



Technicalities: Classes, etc.

“Remember, things take time.”

—Piet Hein

In this chapter, we keep our focus on our main tool for programming: the C++ programming language. We present language technicalities, mostly related to user-defined types, that is, to classes and enumerations. Much of the presentation of language features takes the form of the gradual improvement of a **Date** type. That way, we also get a chance to demonstrate some useful class design techniques.

9.1 User-defined types	9.5 Enumerations
9.2 Classes and members	9.5.1 “Plain” enumerations
9.3 Interface and implementation	9.6 Operator overloading
9.4 Evolving a class	9.7 Class interfaces
9.4.1 <code>struct</code> and functions	9.7.1 Argument types
9.4.2 Member functions and constructors	9.7.2 Copying
9.4.3 Keep details private	9.7.3 Default constructors
9.4.4 Defining member functions	9.7.4 <code>const</code> member functions
9.4.5 Referring to the current object	9.7.5 Members and “helper functions”
9.4.6 Reporting errors	
	9.8 The <code>Date</code> class

9.1 User-defined types

The C++ language provides you with some built-in types, such as `char`, `int`, and `double` (§A.8). A type is called built-in if the compiler knows how to represent objects of the type and which operations can be done on it (such as `+` and `*`) without being told by declarations supplied by a programmer in source code.

Types that are not built-in are called *user-defined types* (UDTs). They can be standard library types – available to all C++ programmers as part of every ISO standard C++ implementation – such as `string`, `vector`, and `ostream` (Chapter 10), or types that we build for ourselves, such as `Token` and `Token_stream` (§6.5 and §6.6). As soon as we get the necessary technicalities under our belt, we’ll build graphics types such as `Shape`, `Line`, and `Text` (Chapter 13). The standard library types are as much a part of the language as the built-in types, but we still consider them user-defined because they are built from the same primitives and with the same techniques as the types we built ourselves; the standard library builders have no special privileges or facilities that you don’t have. Like the built-in types, most user-defined types provide operations. For example, `vector` has `[]` and `size()` (§4.6.1, §B.4.8), `ostream` has `<<`, `Token_stream` has `get()` (§6.8), and `Shape` has `add(Point)` and `set_color()` (§14.2).

Why do we build types? The compiler does not know all the types we might like to use in our programs. It couldn’t, because there are far too many useful types – no language designer or compiler implementer could know them all. We invent new ones every day. Why? What are types good for? Types are good for directly representing ideas in code. When we write code, the ideal is to represent our ideas directly in our code so that we, our colleagues, and the compiler can understand what we wrote. When we want to do integer arithmetic, `int` is a great help; when we want to manipulate text, `string` is a great help; when we want to manipulate calculator input, `Token` and `Token_stream` are a great help. The help comes in two forms:

- *Representation:* A type “knows” how to represent the data needed in an object.
- *Operations:* A type “knows” what operations can be applied to objects.

Many ideas follow this pattern: “something” has data to represent its current value – sometimes called the current *state* – and a set of operations that can be applied. Think of a computer file, a web page, a toaster, a music player, a coffee cup, a car engine, a cell phone, a telephone directory; all can be characterized by some data and all have a more or less fixed set of standard operations that you can perform. In each case, the result of the operation depends on the data – the “current state” – of an object.

So, we want to represent such an “idea” or “concept” in code as a data structure plus a set of functions. The question is: “Exactly how?” This chapter presents the technicalities of the basic ways of doing that in C++.

C++ provides two kinds of user-defined types: classes and enumerations. The class is by far the most general and important, so we first focus on classes. A class directly represents a concept in a program. A *class* is a (user-defined) type that specifies how objects of its type are represented, how those objects can be created, how they are used, and how they can be destroyed (see §17.5). If you think of something as a separate entity, it is likely that you should define a class to represent that “thing” in your program. Examples are vector, matrix, input stream, string, FFT (fast Fourier transform), valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, and clock.

In C++ (as in most modern languages), a class is the key building block for large programs – and very useful for small ones as well, as we saw for our calculator (Chapters 6 and 7).

9.2 Classes and members

A class is a user-defined type. It is composed of built-in types, other user-defined types, and functions. The parts used to define the class are called *members*. A class has zero or more members. For example:

```
class X {  
public:  
    int m;                                // data member  
    int mf(int v) { int old = m; m=v; return old; } // function member  
};
```

Members can be of various types. Most are either data members, which define the representation of an object of the class, or function members, which provide

operations on such objects. We access members using the *object.member* notation. For example:

X var;	<i>// var is a variable of type X</i>
var.m = 7;	<i>// assign to var's data member m</i>
int x = var.mf(9);	<i>// call var's member function mf()</i>

You can read **var.m** as **var's m**. Most people pronounce it “**var dot m**” or “**var's m**.” The type of a member determines what operations we can do on it. We can read and write an **int** member, call a function member, etc.

A member function, such as **X's mf()**, does not need to use the **var.m** notation. It can use the plain member name (**m** in this example). Within a member function, a member name refers to the member of that name in the object for which the member function was called. Thus, in the call **var.mf(9)**, the **m** in the definition of **mf()** refers to **var.m**.

9.3 Interface and implementation



Usually, we think of a class as having an interface plus an implementation. The interface is the part of the class's declaration that its users access directly. The implementation is that part of the class's declaration that its users access only indirectly through the interface. The public interface is identified by the label **public:** and the implementation by the label **private:**. You can think of a class declaration like this:

```
class X {    // this class's name is X
public:
    // public members:
    // – the interface to users (accessible by all)
    // functions
    // types
    // data (often best kept private)
private:
    // private members:
    // – the implementation details (used by members of this class only)
    // functions
    // types
    // data
};
```

Class members are private by default; that is,

```
class X {
    int mf(int);
    // ...
};
```

means

```
class X {
private:
    int mf(int);
    // ...
};
```

so that

```
X x;           // variable x of type X
int y = x.mf(); // error: mf is private (i.e., inaccessible)
```

A user cannot directly refer to a private member. Instead, we have to go through a public function that can use it. For example:

```
class X {
    int m;
    int mf(int);
public:
    int f(int i) { m=i; return mf(i); }
};

X x;
int y = x.f(2);
```

We use **private** and **public** to represent the important distinction between an interface (the user's view of the class) and implementation details (the implementer's view of the class). We explain that and give lots of examples as we go along. Here we'll just mention that for something that's just data, this distinction doesn't make sense. So, there is a useful simplified notation for a class that has no private implementation details. A **struct** is a **class** where members are public by default:

```
struct X {
    int m;
    // ...
};
```

means

```
class X {
public:
    int m;
    // ...
};
```

structs are primarily used for data structures where the members can take any value; that is, we can't define any meaningful invariant (§9.4.3).

9.4 Evolving a class

Let's illustrate the language facilities supporting classes and the basic techniques for using them by showing how – and why – we might evolve a simple data structure into a class with private implementation details and supporting operations. We use the apparently trivial problem of how to represent a date (such as August 14, 1954) in a program. The need for dates in many programs is obvious (commercial transactions, weather data, calendar programs, work records, inventory management, etc.). The only question is how we might represent them.

9.4.1 struct and functions

How would we represent a date? When asked, most people answer, “Well, how about the year, the month, and the day of the month?” That's not the only answer and not always the best answer, but it's good enough for our uses, so that's what we'll do. Our first attempt is a simple **struct**:

```
// simple Date (too simple?)
struct Date {
    int y;    // year
    int m;    // month in year
    int d;    // day of month
};

Date today; // a Date variable (a named object)
```

A **Date** object, such as **today**, will simply be three **ints**:

Date:	
y:	2005
m:	12
d:	24

There is no “magic” relying on hidden data structures anywhere related to a **Date** – and that will be the case for every version of **Date** in this chapter.

So, we now have **Dates**; what can we do with them? We can do everything in the sense that we can access the members of **today** (and any other **Date**) and read and write them as we like. The snag is that nothing is really convenient. Just about anything that we want to do with a **Date** has to be written in terms of reads and writes of those members. For example:

```
// set today to December 24, 2005
today.y = 2005;
today.m = 24;
today.d = 12;
```

This is tedious and error-prone. Did you spot the error? Everything that’s tedious is error-prone! For example, does this make sense?

```
Date x;
x.y = -3;
x.m = 13;
x.d = 32;
```

Probably not, and nobody would write that – or would they? How about

```
Date y;
y.y = 2000;
y.m = 2;
y.d = 29;
```

Was year 2000 a leap year? Are you sure?

What we do then is to provide some helper functions to do the most common operations for us. That way, we don’t have to repeat the same code over and over again and we won’t make, find, and fix the same mistakes over and over again. For just about every type, initialization and assignment are among the most common operations. For **Date**, increasing the value of the **Date** is another common operation, so we write

```
// helper functions:

void init_day(Date& dd, int y, int m, int d)
{
    // check that (y,m,d) is a valid date
    // if it is, use it to initialize dd
}
```

```
void add_day(Date& dd, int n)
{
    // increase dd by n days
}
```

We can now try to use **Date**:

```
void f()
{
    Date today;
    init_day(today, 12, 24, 2005); // oops! (no day 2005 in year 12)
    add_day(today,1);
}
```

First we note the usefulness of such “operations” – here implemented as helper functions. Checking that a date is valid is sufficiently difficult and tedious that if we didn’t write a checking function once and for all, we’d skip the check occasionally and get buggy programs. Whenever we define a type, we want some operations for it. Exactly how many operations we want and of which kind will vary. Exactly how we provide them (as functions, member functions, or operators) will also vary, but whenever we decide to provide a type, we ask ourselves, “Which operations would we like for this type?”

9.4.2 Member functions and constructors

We provided an initialization function for **Dates**, one that provided an important check on the validity of **Dates**. However, checking functions are of little use if we fail to use them. For example, assume that we have defined the output operator **<<** for a **Date** (§9.8):

```
void f()
{
    Date today;
    // ...
    cout << today << '\n'; // use today
    // ...
    init_day(today,2008,3,30);
    // ...
    Date tomorrow;
    tomorrow.y = today.y;
    tomorrow.m = today.m;
    tomorrow.d = today.d+1; // add 1 to today
    cout << tomorrow << '\n'; // use tomorrow
}
```

Here, we “forgot” to immediately initialize `today` and “someone” used it before we got around to calling `init_day()`. “Someone else” decided that it was a waste of time to call `add_day()` – or maybe hadn’t heard of it – and constructed `tomorrow` by hand. As it happens, this is bad code – very bad code. Sometimes, probably most of the time, it works, but small changes lead to serious errors. For example, writing out an uninitialized `Date` will produce garbage output, and incrementing a day by simply adding `1` to its member `d` is a time bomb: when `today` is the last day of the month, the increment yields an invalid date. The worst aspect of this “very bad code” is that it doesn’t look bad.

This kind of thinking leads to a demand for an initialization function that can’t be forgotten and for operations that are less likely to be overlooked. The basic tool for that is *member functions*, that is, functions declared as members of the class within the class body. For example:

```
// simple Date
// guarantee initialization with constructor
// provide some notational convenience
struct Date {
    int y, m, d;                                // year, month, day
    Date(int y, int m, int d);                  // check for valid date and initialize
    void add_day(int n);                         // increase the Date by n days
};
```

A member function with the same name as its class is special. It is called a *constructor* and will be used for initialization (“construction”) of objects of the class. It is an error – caught by the compiler – to forget to initialize an object of a class that has a constructor that requires an argument, and there is a special convenient syntax for doing such initialization:

<code>Date my_birthday;</code>	<i>// error: my_birthday not initialized</i>
<code>Date today {12,24,2007};</code>	<i>// oops! run-time error</i>
<code>Date last {2000,12,31};</code>	<i>// OK (colloquial style)</i>
<code>Date next = {2014,2,14};</code>	<i>// also OK (slightly verbose)</i>
<code>Date christmas = Date{1976,12,24};</code>	<i>// also OK (verbose style)</i>

The attempt to declare `my_birthday` fails because we didn’t specify the required initial value. The attempt to declare `today` will pass the compiler, but the checking code in the constructor will catch the illegal date at run time (`{12,24,2007}` – there is no day 2007 of the 24th month of year 12).

The definition of `last` provides the initial value – the arguments required by `Date`’s constructor – as a `{ }` list immediately after the name of the variable. That’s the most common style of initialization of variables of a class that has a

constructor requiring arguments. We can also use the more verbose style where we explicitly create an object (here, `Date{1976,12,24}`) and then use that to initialize the variable using the `=` initializer syntax. Unless you actually like typing, you'll soon tire of that.

We can now try to use our newly defined variables:

```
last.add_day(1);  
add_day(2); // error: what date?
```

Note that the member function `add_day()` is called for a particular `Date` using the dot member-access notation. We'll show how to define member functions in §9.4.4.

In C++98, people used parentheses to delimit the initializer list, so you will see a lot of code like this:

Date last(2000,12,31); // OK (old colloquial style)

We prefer `{ }` for initializer lists because that clearly indicates when initialization (construction) is done, and also because that notation is more widely useful. The notation can also be used for built-in types. For example:

```
int x {7}; // OK (modern initializer list style)
```

Optionally, we can use a `=` before the `{ }` list:

Date next = {2014,2,14}; // also OK (slightly verbose)

Some find this combination of older and new style more readable.

9.4.3 Keep details private

We still have a problem: What if someone forgets to use the member function `add_day()`? What if someone decides to change the month directly? After all, we “forgot” to provide a facility for that:

```
Date birthday {1960,12,31}; // December 31, 1960  
++birthday.d; // ouch! Invalid date  
// (birthday.d==32 makes today invalid)
```

```
Date today {1970,2,3};  
today.m = 14; // ouch! Invalid date  
// (today.m==14 makes today invalid)
```

As long as we leave the representation of **Date** accessible to everybody, somebody will – by accident or design – mess it up; that is, someone will do something that produces an invalid value. In this case, we created a **Date** with a value that doesn't correspond to a day on the calendar. Such invalid objects are time bombs; it is just a matter of time before someone innocently uses the invalid value and gets a run-time error or – usually worse – produces a bad result.

Such concerns lead us to conclude that the representation of **Date** should be inaccessible to users except through the public member functions that we supply. Here is a first cut:

```
// simple Date (control access)
class Date {
    int y, m, d; // year, month, day
public:
    Date(int y, int m, int d); // check for valid date and initialize
    void add_day(int n); // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};
```

We can use it like this:

```
Date birthday {1970, 12, 30}; // OK
birthday.m = 14; // error: Date::m is private
cout << birthday.month() << '\n'; // we provided a way to read m
```

The notion of a “valid **Date**” is an important special case of the idea of a valid value. We try to design our types so that values are guaranteed to be valid; that is, we hide the representation, provide a constructor that creates only valid objects, and design all member functions to expect valid values and leave only valid values behind when they return. The value of an object is often called its *state*, so the idea of a valid value is often referred to as a *valid state* of an object.

The alternative is for us to check for validity every time we use an object, or just hope that nobody left an invalid value lying around. Experience shows that “hoping” can lead to “pretty good” programs. However, producing “pretty good” programs that occasionally produce erroneous results and occasionally crash is no way to win friends and respect as a professional. We prefer to write code that can be demonstrated to be correct.

A rule for what constitutes a valid value is called an *invariant*. The invariant for **Date** (“A **Date** must represent a day in the past, present, or future”) is unusually

hard to state precisely: remember leap years, the Gregorian calendar, time zones, etc. However, for simple realistic uses of **Dates** we can do it. For example, if we are analyzing internet logs, we need not be bothered with the Gregorian, Julian, or Mayan calendars. If we can't think of a good invariant, we are probably dealing with plain data. If so, use a **struct**.

9.4.4 Defining member functions

So far, we have looked at **Date** from the point of view of an interface designer and a user. But sooner or later, we have to implement those member functions. First, here is a subset of the **Date** class reorganized to suit the common style of providing the public interface first:

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int y, int m, int d); // constructor: check for valid date and initialize
    void add_day(int n); // increase the Date by n days
    int month();
    // ...
private:
    int y, m, d; // year, month, day
};
```

People put the public interface first because the interface is what most people are interested in. In principle, a user need not look at the implementation details. In reality, we are typically curious and have a quick look to see if the implementation looks reasonable and if the implementer used some technique that we could learn from. However, unless we are the implementers, we do tend to spend much more time with the public interface. The compiler doesn't care about the order of class function and data members; it takes the declarations in any order you care to present them.

When we define a member outside its class, we need to say which class it is a member of. We do that using the *class_name::member_name* notation:

```
Date::Date(int yy, int mm, int dd) // constructor
    :y{yy}, m{mm}, d{dd} // note: member initializers
{
}

void Date::add_day(int n)
{
    // ...
}
```

```
int month()           // oops: we forgot Date::
{
    return m;        // not the member function, can't access m
}
```

The **:y{yy}, m{mm}, d{dd}** notation is how we initialize members. It is called a (member) initializer list. We could have written

```
Date::Date(int yy, int mm, int dd) // constructor
{
    y = yy;
    m = mm;
    d = dd;
}
```

but then we would in principle first have default initialized the members and then assigned values to them. We would then also open the possibility of accidentally using a member before it was initialized. The **:y{yy}, m{mm}, d{dd}** notation more directly expresses our intent. The distinction is exactly the same as the one between

```
int x;      // first define the variable x
// . . .
x = 2;      // later assign to x
```

and

```
int x {2}; // define and immediately initialize with 2
```

We can also define member functions right in the class definition:

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd)
        :y{yy}, m{mm}, d{dd}
    {
        // . . .
    }

    void add_day(int n)
    {
        // . . .
    }
}
```

```
int month() { return m; }

// ...
private:
    int y, m, d; // year, month, day
};
```

The first thing we notice is that the class declaration became larger and “messier.” In this example, the code for the constructor and `add_day()` could be a dozen or more lines each. This makes the class declaration several times larger and makes it harder to find the interface among the implementation details. Consequently, we don’t define large functions within a class declaration.

However, look at the definition of `month()`. That’s straightforward and shorter than the version that places `Date::month()` out of the class declaration. For such short, simple functions, we might consider writing the definition right in the class declaration.

Note that `month()` can refer to `m` even though `m` is defined after (below) `month()`. A member can refer to a function or data member of its class independently of where in the class that other member is declared. The rule that a name must be declared before it is used is relaxed within the limited scope of a class.

Writing the definition of a member function within the class definition has three effects:

- The function will be *inline*; that is, the compiler will try to generate code for the function at each point of call rather than using function-call instructions to use common code. This can be a significant performance advantage for functions, such as `month()`, that hardly do anything but are used a lot.
- All uses of the class will have to be recompiled whenever we make a change to the body of an inlined function. If the function body is out of the class declaration, recompilation of users is needed only when the class declaration is itself changed. Not recompiling when the body is changed can be a huge advantage in large programs.
- The class definition gets larger. Consequently, it can be harder to find the members among the member function definitions.

The obvious rule of thumb is: Don’t put member function bodies in the class declaration unless you know that you need the performance boost from inlining tiny functions. Large functions, say five or more lines of code, don’t benefit from inlining and make a class declaration harder to read. We rarely inline a function that consists of more than one or two expressions.

9.4.5 Referring to the current object

Consider a simple use of the **Date** class so far:

```
class Date {
    // ...
    int month() { return m; }
    // ...
private:
    int y, m, d;    // year, month, day
};

void f(Date d1, Date d2)
{
    cout << d1.month() << ' ' << d2.month() << '\n';
}
```

How does **Date::month()** know to return the value of **d1.m** in the first call and **d2.m** in the second? Look again at **Date::month()**; its declaration specifies no function argument! How does **Date::month()** know for which object it was called? A class member function, such as **Date::month()**, has an implicit argument which it uses to identify the object for which it is called. So in the first call, **m** correctly refers to **d1.m** and in the second call it refers to **d2.m**. See §17.10 for more uses of this implicit argument.

9.4.6 Reporting errors

What do we do when we find an invalid date? Where in the code do we look for invalid dates? From §5.6, we know that the answer to the first question is “Throw an exception,” and the obvious place to look is where we first construct a **Date**. If we don’t create invalid **Dates** and also write our member functions correctly, we will never have a **Date** with an invalid value. So, we’ll prevent users from ever creating a **Date** with an invalid state:

```
// simple Date (prevent invalid dates)
class Date {
public:
    class Invalid { };           // to be used as exception
    Date(int y, int m, int d);   // check for valid date and initialize
    // ...
private:
    int y, m, d;                // year, month, day
    bool is_valid();              // return true if date is valid
};
```

We put the testing of validity into a separate `is_valid()` function because checking for validity is logically distinct from initialization and because we might want to have several constructors. As you can see, we can have private functions as well as private data:

```
Date::Date(int yy, int mm, int dd)
    : y{yy}, m{mm}, d{dd}           // initialize data members
{
    if (!is_valid()) throw Invalid{}; // check for validity
}

bool Date::is_valid()           // return true if date is valid
{
    if (m<1 || 12<m) return false;
    // ...
}
```

Given that definition of `Date`, we can write

```
void f(int x, int y)
try {
    Date dxy {2004,x,y};
    cout << dxy << '\n';           // see §9.8 for a declaration of <<
    dxy.add_day(2);
}
catch(Date::Invalid) {
    error("invalid date");        // error() defined in §5.6.3
}
```

We now know that `<<` and `add_day()` will have a valid `Date` on which to operate.

Before completing the evolution of our `Date` class in §9.7, we'll take a detour to describe a couple of general language facilities that we'll need to do that well: enumerations and operator overloading.

9.5 Enumerations



An `enum` (an *enumeration*) is a very simple user-defined type, specifying its set of values (its *enumerators*) as symbolic constants. For example:

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
```

The “body” of an enumeration is simply a list of its enumerators. The `class` in `enum class` means that the enumerators are in the scope of the enumeration. That is, to refer to `jan`, we have to say `Month::jan`.

You can give a specific representation value for an enumerator, as we did for `jan` here, or leave it to the compiler to pick a suitable value. If you leave it to the compiler to pick, it’ll give each enumerator the value of the previous enumerator plus one. Thus, our definition of `Month` gave the months consecutive values starting with `1`. We could equivalently have written

```
enum class Month {
    jan=1, feb=2, mar=3, apr=4, may=5, jun=6,
    jul=7, aug=8, sep=9, oct=10, nov=11, dec=12
};
```

However, that’s tedious and opens the opportunity for errors. In fact, we made two typing errors before getting this latest version right; it is better to let the compiler do simple, repetitive “mechanical” things. The compiler is better at such tasks than we are, and it doesn’t get bored.

If we don’t initialize the first enumerator, the count starts with `0`. For example:

```
enum class Day {
    monday, tuesday, wednesday, thursday, friday, saturday, sunday
};
```

Here `monday` is represented as `0` and `sunday` is represented as `6`. In practice, starting with `0` is often a good choice.

We can use our `Month` like this:

```
Month m = Month::feb;

Month m2 = feb;           // error: feb is not in scope
m = 7;                   // error: can't assign an int to a Month
int n = m;                // error: can't assign a Month to an int
Month mm = Month(7);       // convert int to Month (unchecked)
```

`Month` is a separate type from its “underlying type” `int`. Every `Month` has an equivalent integer value, but most `ints` do not have a `Month` equivalent. For example, we really do want this initialization to fail:

```
Month bad = 9999;          // error: can't convert an int to a Month
```

If you insist on using the `Month(9999)` notation, on your head be it! In many cases, C++ will not try to stop a programmer from doing something potentially



silly when the programmer explicitly insists; after all, the programmer might actually know better. Note that you cannot use the `Month{9999}` notation because that would allow only values that could be used in an initialization of a `Month`, and `ints` cannot.

Unfortunately, we cannot define a constructor for an enumeration to check initializer values, but it is trivial to write a simple checking function:

```
Month int_to_month(int x)
{
    if (x<int(Month::jan) || int(Month::dec)<x) error("bad month");
    return Month(x);
}
```

We use the `int(Month::jan)` notation to get the `int` representation of `Month::jan`. Given that, we can write

```
void f(int m)
{
    Month mm = int_to_month(m);
    // ...
}
```

What do we use enumerations for? Basically, an enumeration is useful whenever we need a set of related named integer constants. That happens all the time when we try to represent sets of alternatives (`up, down; yes, no, maybe; on, off; n, ne, e, se, s, sw, w, nw`) or distinctive values (`red, blue, green, yellow, maroon, crimson, black`).

9.5.1 “Plain” enumerations

In addition to the `enum classes`, also known as *scoped enumerations*, there are “plain” enumerations that differ from scoped enumerations by implicitly “exporting” their enumerators to the scope of the enumeration and allowing implicit conversions to `int`. For example:

```
enum Month {                                // note: no "class"
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;                            // OK: feb in scope
Month m2 = Month::feb;                    // also OK
m = 7;                                    // error: can't assign an int to a Month
```

```
int n = m;           // OK: we can assign a Month to an int
Month mm = Month(7); // convert int to Month (unchecked)
```

Obviously, “plain” **enums** are less strict than **enum classes**. Their enumerators can “pollute” the scope in which their enumerator is defined. That can be a convenience, but it occasionally leads to surprises. For example, if you try to use this **Month** together with the **iostream** formatting mechanisms (§11.2.1), you will find that **dec** for December clashes with **dec** for decimal.

Similarly, having an enumeration value convert to **int** can be a convenience (it saves us from being explicit when we want a conversion to **int**), but occasionally it leads to surprises. For example:

```
void my_code(Month m)
{
    If (m==17) do_something();           // huh: 17th month?
    If (m==monday) do_something_else(); // huh: compare month to
                                         // Monday?
}
```

If **Month** is an **enum class**, neither condition will compile. If **monday** is an enumerator of a “plain” **enum**, rather than an **enum class**, the comparison of a month to Monday would succeed, most likely with undesirable results.

Prefer the simpler and safer **enum classes** to “plain” **enums**, but expect to find “plain” **enums** in older code: **enum classes** are new in C++11.

9.6 Operator overloading

You can define almost all C++ operators for class or enumeration operands. That’s often called *operator overloading*. We use it when we want to provide conventional notation for a type we design. For example:

```
enum class Month {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

Month operator++(Month& m)           // prefix increment operator
{
    m = (m==Dec) ? Jan : Month(int(m)+1); // "wrap around"
    return m;
}
```

The `? :` construct is an “arithmetic if”: `m` becomes `Jan` if `(m==Dec)` and `Month(int(m)+1)` otherwise. It is a reasonably elegant way of expressing the fact that months “wrap around” after December. The `Month` type can now be used like this:

```
Month m = Sep;
++m;      // m becomes Oct
++m;      // m becomes Nov
++m;      // m becomes Dec
++m;      // m becomes Jan ("wrap around")
```

You might not think that incrementing a `Month` is common enough to warrant a special operator. That may be so, but how about an output operator? We can define one like this:

```
vector<string> month_tbl;

ostream& operator<<(ostream& os, Month m)
{
    return os << month_tbl[int(m)];
}
```

This assumes that `month_tbl` has been initialized somewhere so that (for example) `month_tbl[int(Month::mar)]` is "March" or some other suitable name for that month; see §10.11.3.

 You can define just about any operator provided by C++ for your own types, but only existing operators, such as `+`, `-`, `*`, `/`, `%`, `[]`, `()`, `^`, `!`, `&`, `<`, `<=`, `>`, and `>=`. You cannot define your own operators; you might like to have `**` or `$=` as operators in your program, but C++ won’t let you. You can define operators only with their conventional number of operands; for example, you can define unary `-`, but not unary `<=` (less than or equal), and binary `+`, but not binary `!` (not). Basically, the language allows you to use the existing syntax for the types you define, but not to extend that syntax.

An overloaded operator must have at least one user-defined type as operand:

```
int operator+(int,int); // error: you can't overload built-in +
Vector operator+(const Vector&, const Vector &);    // OK
Vector operator+=(const Vector&, int);           // OK
```

 It is generally a good idea not to define operators for a type unless you are really certain that it makes a big positive change to your code. Also, define operators only with their conventional meaning: `+` should be addition, binary `*`

multiplication, `[]` access, `()` call, etc. This is just advice, not a language rule, but it is good advice: conventional use of operators, such as `+` for addition, can significantly help us understand a program. After all, such use is the result of hundreds of years of experience with mathematical notation. Conversely, obscure operators and unconventional use of operators can be a significant distraction and a source of errors. We will not elaborate on this point. Instead, in the following chapters, we will simply use operator overloading in a few places where we consider it appropriate.

Note that the most interesting operators to overload aren't `+`, `-`, `*`, and `/` as people often assume, but `=`, `==`, `!=`, `<`, `[]` (subscript), and `()` (call).

9.7 Class interfaces

We have argued that the public interface and the implementation parts of a class should be separated. As long as we leave open the possibility of using `structs` for types that are “plain old data,” few professionals would disagree. However, how do we design a good interface? What distinguishes a good public interface from a mess? Part of that answer can be given only by example, but there are a few general principles that we can list and that are given some support in C++:

- Keep interfaces complete.
- Keep interfaces minimal.
- Provide constructors.
- Support copying (or prohibit it) (see §14.2.4).
- Use types to provide good argument checking.
- Identify nonmodifying member functions (see §9.7.4).
- Free all resources in the destructor (see §17.5).

See also §5.5 (how to detect and report run-time errors).

The first two principles can be summarized as “Keep the interface as small as possible, but no smaller.” We want our interface to be small because a small interface is easy to learn and easy to remember, and the implementer doesn’t waste a lot of time implementing unnecessary and rarely used facilities. A small interface also means that when something is wrong, there are only a few functions to check to find the problem. On average, the more public member functions are, the harder it is to find bugs – and please don’t get us started on the complexities of debugging classes with public data. But of course, we want a complete interface; otherwise, it would be useless. We couldn’t use an interface that didn’t allow us to do all we really needed.

Let’s look at the other – less abstract and more directly supported – ideals.

9.7.1 Argument types

When we defined the constructor for `Date` in §9.4.3, we used three `ints` as the arguments. That caused some problems:

```
Date d1 {4,5,2005};    // oops: year 4, day 2005
Date d2 {2005,4,5};    // April 5 or May 4?
```

The first problem (an illegal day of the month) is easily dealt with by a test in the constructor. However, the second (a month vs. day-of-the-month confusion) can't be caught by code written by the user. The second problem is simply that the conventions for writing month and day-in-month differ; for example, 4/5 is April 5 in the United States and May 4 in England. Since we can't calculate our way out of this, we must do something else. The obvious solution is to use a `Month` type:

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

// simple Date (use Month type)
class Date {
public:
    Date(int y, Month m, int d);    // check for valid date and initialize
    // ...
private:
    int y;                          // year
    Month m;                       // month
    int d;                          // day
};
```

When we use a `Month` type, the compiler will catch us if we swap month and day, and using an enumeration as the `Month` type also gives us symbolic names to use. It is usually easier to read and write symbolic names than to play around with numbers, and therefore less error-prone:

```
Date dx1 {1998, 4, 3};           // error: 2nd argument not a Month
Date dx2 {1998, 4, Month::mar};   // error: 2nd argument not a Month
Date dx2 {4, Month::mar, 1998};   // oops: run-time error: day 1998
Date dx2 {Month::mar, 4, 1998};   // error: 2nd argument not a Month
Date dx3 {1998, Month::mar, 30};  // OK
```

This takes care of most “accidents.” Note the use of the qualification of the enumerator `mar` with the enumeration name: `Month::mar`. We don't say `Month.mar`

because **Month** isn't an object (it's a type) and **mar** isn't a data member (it's an enumerator – a symbolic constant). Use `::` after the name of a class, enumeration, or namespace (§8.7) and `.` (dot) after an object name.

When we have a choice, we catch errors at compile time rather than at run time. We prefer for the compiler to find the error rather than for us to try to figure out exactly where in the code a problem occurred. Also, errors caught at compile time don't require checking code to be written and executed.

Thinking like that, could we catch the swap of the day of the month and the year also? We could, but the solution is not as simple or as elegant as for **Month**; after all, there was a year 4 and you might want to represent it. Even if we restricted ourselves to modern times there would probably be too many relevant years for us to list them all in an enumeration.

Probably the best we could do (without knowing quite a lot about the intended use of **Date**) would be something like this:

```
class Year {                                     // year in [min:max) range
    static const int min = 1800;
    static const int max = 2200;
public:
    class Invalid {};
    Year(int x) : y{x} { if (x<min || max<=x) throw Invalid{}; }
    int year() { return y; }
private:
    int y;
};

class Date {
public:
    Date(Year y, Month m, int d);           // check for valid date and initialize
    // ...
private:
    Year y;
    Month m;
    int d; // day
};
```

Now we get

```
Date dx1 {Year{1998}, 4, 3};                // error: 2nd argument not a Month
Date dx2 {Year{1998}, 4, Month::mar};          // error: 2nd argument not a Month
Date dx2 {4, Month::mar, Year{1998}};        // error: 1st argument not a Year
Date dx2 {Month::mar, 4, Year{1998}};          // error: 2nd argument not a Month
Date dx3 {Year{1998}, Month::mar, 30};         // OK
```

This weird and unlikely error would still not be caught until run time:

```
Date dx2 {Year{4}, Month::mar, 1998};      // run-time error: Year::Invalid
```

Is the extra work and notation to get years checked worthwhile? Naturally, that depends on the constraints on the kind of problem you are solving using **Date**, but in this case we doubt it and won't use class **Year** as we go along.

When we program, we always have to ask ourselves what is good enough for a given application. We usually don't have the luxury of being able to search "forever" for the perfect solution after we have already found one that is good enough. Search further, and we might even come up with something that's so elaborate that it is worse than the simple early solution. This is one meaning of the saying "The best is the enemy of the good" (Voltaire).

Note the use of **static const** in the definitions of **min** and **max**. This is the way we define symbolic constants of integer types within classes. For a class member, we use **static** to make sure that there is just one copy of the value in the program, rather than one per object of the class. In this case, because the initializer is a constant expression, we could have used **constexpr** instead of **const**.

9.7.2 Copying

We always have to create objects; that is, we must always consider initialization and constructors. Arguably they are the most important members of a class: to write them, you have to decide what it takes to initialize an object and what it means for a value to be valid (what is the invariant?). Just thinking about initialization will help you avoid errors.

The next thing to consider is often: Can we copy our objects? And if so, how do we copy them?

For **Date** or **Month**, the answer is that we obviously want to copy objects of that type and that the meaning of *copy* is trivial: just copy all of the members. Actually, this is the default case. So as long as you don't say anything else, the compiler will do exactly that. For example, if you copy a **Date** as an initializer or right-hand side of an assignment, all its members are copied:

```
Date holiday {1978, Month::jul, 4};      // initialization
Date d2 = holiday;
Date d3 = Date{1978, Month::jul, 4};
holiday = Date{1978, Month::dec, 24};    // assignment
d3 = holiday;
```

This will all work as expected. The **Date{1978, Month::dec, 24}** notation makes the appropriate unnamed **Date** object, which you can then use appropriately. For example:

```
cout << Date{1978, Month::dec, 24};
```

This is a use of a constructor that acts much as a literal for a class type. It often comes in as a handy alternative to first defining a variable or **const** and then using it once.

What if we don't want the default meaning of copying? We can either define our own (see §18.3) or **delete** the copy constructor and copy assignment (see §14.2.4).

9.7.3 Default constructors

Uninitialized variables can be a serious source of errors. To counter that problem, we have the notion of a constructor to guarantee that every object of a class is initialized. For example, we declared the constructor **Date::Date(int,Month,int)** to ensure that every **Date** is properly initialized. In the case of **Date**, that means that the programmer must supply three arguments of the right types. For example:

Date d0;	<i>// error: no initializer</i>
Date d1 {};	<i>// error: empty initializer</i>
Date d2 {1998};	<i>// error: too few arguments</i>
Date d3 {1,2,3,4};	<i>// error: too many arguments</i>
Date d4 {1,"jan",2};	<i>// error: wrong argument type</i>
Date d5 {1,Month::jan,2};	<i>// OK: use the three-argument constructor</i>
Date d6 {d5};	<i>// OK: use the copy constructor</i>

Note that even though we defined a constructor for **Date**, we can still copy **Dates**.

Many classes have a good notion of a default value; that is, there is an obvious answer to the question "What value should it have if I didn't give it an initializer?" For example:

string s1;	<i>// default value: the empty string " "</i>
vector<string> v1;	<i>// default value: the empty vector; no elements</i>

This looks reasonable. It even works the way the comments indicate. That is achieved by giving **vector** and **string** *default constructors* that implicitly provide the desired initialization.

For a type **T**, **T{}** is the notation for the default value, as defined by the default constructor, so we could write

string s1 = string{};	<i>// default value: the empty string " "</i>
vector<string> v1 = vector<string>{};	<i>// default value: the empty vector; // no elements</i>

However, we prefer the equivalent and colloquial

string s1;	<i>// default value: the empty string " "</i>
vector<string> v1;	<i>// default value: the empty vector; no elements</i>

For built-in types, such as `int` and `double`, the default constructor notation means `0`, so `int{}` is a complicated way of saying `0`, and `double{}` a long-winded way of saying `0.0`.

Using a default constructor is not just a matter of looks. Imagine that we could have an uninitialized `string` or `vector`.

```
string s;
for (int i=0; i<s.size(); ++i) // oops: loop an undefined number of times
    s[i] = toupper(s[i]);      // oops: read and write a random memory location

vector<string> v;
v.push_back("bad");          // oops: write to random address
```

If the values of `s` and `v` were genuinely undefined, `s` and `v` would have no notion of how many elements they contained or (using the common implementation techniques; see §17.5) where those elements were supposed to be stored. The results would be use of random addresses – and that can lead to the worst kind of errors. Basically, without a constructor, we cannot establish an invariant – we cannot ensure that the values in those variables are valid (§9.4.3). We must insist that such variables are initialized. We could insist on an initializer and then write

```
string s1 = "";
vector<string> v1 {};
```

But we don't think that's particularly pretty. For `string`, `""` is rather obvious for “empty string.” For `vector`, `{}` is pretty for a vector with no elements. However, for many types, it is not easy to find a reasonable notation for a default value. For many types, it is better to define a constructor that gives meaning to the creation of an object without an explicit initializer. Such a constructor takes no arguments and is called a *default constructor*.

There isn't an obvious default value for dates. That's why we haven't defined a default constructor for `Date` so far, but let's provide one (just to show we can):

```
class Date {
public:
    // ...
    Date();    // default constructor
    // ...
private:
    int y;
    Month m;
    int d;
};
```

We have to pick a default date. The first day of the 21st century might be a reasonable choice:

```
Date::Date()
    :y{2001}, m{Month::jan}, d{1}
{
}
```

Instead of placing the default values for members in the constructor, we could place them on the members themselves:

```
class Date {
public:
    // ...
    Date();           // default constructor
    Date(year, Month, day);
    Date(int y);     // January 1 of year y
    // ...
private:
    int y {2001};
    Month m {Month::jan};
    int d {1};
};
```

That way, the default values are available to every constructor. For example:

```
Date::Date(int y)           // January 1 of year y
    :y{yy}
{
    if (!is_valid()) throw Invalid(); // check for validity
}
```

Because **Date(int)** does not explicitly initialize the month (**m**) or the day (**d**), the specified initializers (**Month::jan** and **1**) are implicitly used. An initializer for a class member specified as part of the member declaration is called an *in-class initializer*.

If we didn't like to build the default value right into the constructor code, we could use a constant (or a variable). To avoid a global variable and its associated initialization problems, we use the technique from §8.6.2:

```
const Date& default_date()
{
    static Date dd {2001, Month::jan, 1};
    return dd;
}
```



We used `static` to get a variable (`dd`) that is created only once, rather than each time `default_date()` is called, and initialized the first time `default_date()` is called. Given `default_date()`, it is trivial to define a default constructor for `Date`:

```
Date::Date()
: y{default_date().year()},
  m{default_date().month()},
  d{default_date().day()}

{ }
}
```

Note that the default constructor does not need to check its value; the constructor for `default_date` already did that. Given this default `Date` constructor, we can now define nonempty vectors of `Dates` without listing element values:

```
vector<Date> birthdays(10); // ten elements with the default Date value,
// Date{}
```

Without the default constructor, we would have had to be explicit:

```
vector<Date> birthdays(10, default_date()); // ten default Dates

vector<Date> birthdays2 = { // ten default Dates
    default_date(), default_date(), default_date(), default_date(), default_
    date(),
    default_date(), default_date(), default_date(), default_date(), default_
    date()
};
```

We use parentheses, `()`, when specifying the element counts for a `vector`, rather than the `{ }` initializer-list notation, to avoid confusion in the case of a `vector<int>` (§18.2).

9.7.4 `const` member functions

Some variables are meant to be changed – that’s why we call them “variables” – but some are not; that is, we have “variables” representing immutable values. Those, we typically call *constants* or just `consts`. Consider:

```
void some_function(Date& d, const Date& start_of_term)
{
    int a = d.day(); // OK
    int b = start_of_term.day(); // should be OK (why?)
```

```

d.add_day(3);           // fine
start_of_term.add_day(3); // error
}

```

Here we intend **d** to be mutable, but **start_of_term** to be immutable; it is not acceptable for **some_function()** to change **start_of_term**. How would the compiler know that? It knows because we told it by declaring **start_of_term const**. So far, so good, but then why is it OK to read the **day** of **start_of_term** using **day()**? As the definition of **Date** stands so far, **start_of_term.day()** is an error because the compiler does not know that **day()** doesn't change its **Date**. We never told it, so the compiler assumes that **day()** may modify its **Date** and reports an error.

We can deal with this problem by classifying operations on a class as modifying and nonmodifying. That's a pretty fundamental distinction that helps us understand a class, but it also has a very practical importance: operations that do not modify the object can be invoked for **const** objects. For example:

```

class Date {
public:
    // ...
    int day() const;           // const member: can't modify the object
    Month month() const;      // const member: can't modify the object
    int year() const;          // const member: can't modify the object

    void add_day(int n);       // non-const member: can modify the object
    void add_month(int n);     // non-const member: can modify the object
    void add_year(int n);      // non-const member: can modify the object

private:
    int y;                    // year
    Month m;
    int d;                    // day of month
};

Date d {2000, Month::jan, 20};
const Date cd {2001, Month::feb, 21};

cout << d.day() << " - " << cd.day() << '\n'; // OK
d.add_day(1);                                     // OK
cd.add_day(1);                                    // error: cd is a const

```

We use **const** right after the argument list in a member function declaration to indicate that the member function can be called for a **const** object. Once we have

declared a member function **const**, the compiler holds us to our promise not to modify the object. For example:

```
int Date::day() const
{
    ++d;      // error: attempt to change object from const member function
    return d;
}
```

Naturally, we don't usually try to "cheat" in this way. What the compiler provides for the class implementer is primarily protection against accident, which is particularly useful for more complex code.

9.7.5 Members and “helper functions”

When we design our interfaces to be minimal (though complete), we have to leave out lots of operations that are merely useful. A function that can be simply, elegantly, and efficiently implemented as a freestanding function (that is, as a non-member function) should be implemented outside the class. That way, a bug in that function cannot directly corrupt the data in a class object. Not accessing the representation is important because the usual debug technique is "Round up the usual suspects"; that is, when something goes wrong with a class, we first look at the functions that directly access the representation: one of those almost certainly did it. If there are a dozen such functions, we will be much happier than if there were 50.

Fifty functions for a **Date** class! You must wonder if we are kidding. We are not: a few years ago I surveyed a number of commercially used **Date** libraries and found them full of functions like **next_Sunday()**, **next_workday()**, etc. Fifty is not an unreasonable number for a class designed for the convenience of the users rather than for ease of comprehension, implementation, and maintenance.

Note also that if the representation changes, only the functions that directly access the representation need to be rewritten. That's another strong practical reason for keeping interfaces minimal. In our **Date** example, we might decide that an integer representing the number of days since January 1, 1900, is a much better representation for our uses than (year,month,day). Only the member functions would have to be changed.

Here are some examples of *helper functions*:

```
Date next_Sunday(const Date& d)
{
    // access d using d.day(), d.month(), and d.year()
    // make new Date to return
}
```

```

Date next_weekday(const Date& d) { /* ... */ }

bool leapyear(int y) { /* ... */ }

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}

```

Helper functions are also called *convenience functions*, *auxiliary functions*, and many other things. The distinction between these functions and other nonmember functions is logical; that is, “helper function” is a design concept, not a programming language concept. The helper functions often take arguments of the classes that they are helpers of. There are exceptions, though: note `leapyear()`. Often, we use namespaces to identify a group of helper functions; see §8.7:

```

namespace Chrono {
    enum class Month { /* ... */};
    class Date { /* ... */};
    bool is_date(int y, Month m, int d);      // true for valid date
    Date next_Sunday(const Date& d) { /* ... */}
    Date next_weekday(const Date& d) { /* ... */}
    bool leapyear(int y) { /* ... */}           // see exercise 10
    bool operator==(const Date& a, const Date& b) { /* ... */ }
    // ...
}

```

Note the `==` and `!=` functions. They are typical helpers. For many classes, `==` and `!=` make obvious sense, but since they don’t make sense for all classes, the compiler can’t write them for you the way it writes the copy constructor and copy assignment.

Note also that we introduced a helper function `is_date()`. That function replaces `Date::is_valid()` because checking whether a date is valid is largely independent of the representation of a `Date`. For example, we don’t need to know how `Date` objects are represented to know that “January 30, 2008” is a valid date and “February 30, 2008” is not. There still may be aspects of a date that depend on

the representation (e.g., can we represent “January 30, 1066”?), but (if necessary) **Date**'s constructor can take care of that.

9.8 The Date class

So, let's just put it all together and see what that **Date** class might look like when we combine all of the ideas/concerns. Where a function's body is just a `...` comment, the actual implementation is tricky (please don't try to write those just yet). First we place the declarations in a header **Chrono.h**:

```
// file Chrono.h

namespace Chrono {

enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

class Date {
public:
    class Invalid { }; // to throw as exception

    Date(int y, Month m, int d); // check for valid date and initialize
    Date(); // default constructor
    // the default copy operations are fine

    // nonmodifying operations:
    int day() const { return d; }
    Month month() const { return m; }
    int year() const { return y; }

    // modifying operations:
    void add_day(int n);
    void add_month(int n);
    void add_year(int n);

private:
    int y;
    Month m;
    int d;
};

bool is_date(int y, Month m, int d); // true for valid date
```

```

bool leapyear(int y);           // true if y is a leap year

bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);

ostream& operator<<(ostream& os, const Date& d);

istream& operator>>(istream& is, Date& dd);

}
```

// Chrono

The definitions go into **Chrono.cpp**:

```

// Chrono.cpp
#include "Chrono.h"

namespace Chrono {
// member function definitions:

Date::Date(int yy, Month mm, int dd)
    : y{yy}, m{mm}, d{dd}
{
    if (!is_date(yy,mm,dd)) throw Invalid();
}

const Date& default_date()
{
    static Date dd {2001,Month::jan,1};    // start of 21st century
    return dd;
}

Date::Date()
:y{default_date().year()},
m{default_date().month()},
d{default_date().day()}
{
}

void Date:: add_day(int n)
{
    // ...
}

```

```
void Date::add_month(int n)
{
    // ...
}

void Date::add_year(int n)
{
    if (m==feb && d==29 && !leapyear(y+n)) {      // beware of leap years!
        m = mar;                                // use March 1 instead of February 29
        d = 1;
    }
    y+=n;
}

// helper functions:

bool is_date(int y, Month m, int d)
{
    // assume that y is valid

    if (d<=0) return false;           // d must be positive
    if (m<Month::jan || Month::dec<m) return false;

    int days_in_month = 31;          // most months have 31 days

    switch (m) {
        case Month::feb:           // the length of February varies
            days_in_month = (leapyear(y))?29:28;
            break;
        case Month::apr: case Month::jun: case Month::sep: case Month::nov:
            days_in_month = 30;       // the rest have 30 days
            break;
    }

    if (days_in_month<d) return false;

    return true;
}

bool leapyear(int y)
{
    // see exercise 10
}
```

```
bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}

ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
        << ',' << d.month()
        << ',' << d.day() << ')';
}

istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1!= '(' || ch2!=',' || ch3!=',' || ch4!=')') { // oops: format error
        is.clear(ios_base::failbit); // set the fail bit
    }
    return is;
}

dd = Date(y, Month(m),d); // update dd

return is;
}

enum class Day {
    sunday, monday, tuesday, wednesday, thursday, friday, saturday
};

Day day_of_week(const Date& d)
{
    // ...
}
```

Date next_Sunday(const Date& d)

```
{  
    // ...  
}
```

Date next_weekday(const Date& d)

```
{  
    // ...  
}  
  
} // Chrono
```

The functions implementing `>>` and `<<` for **Date** will be explained in detail in §10.8 and §10.9.



Drill

This drill simply involves getting the sequence of versions of **Date** to work. For each version define a **Date** called **today** initialized to June 25, 1978. Then, define a **Date** called **tomorrow** and give it a value by copying **today** into it and increasing its day by one using `add_day()`. Finally, output **today** and **tomorrow** using a `<<` defined as in §9.8.

Your check for a valid date may be very simple. Feel free to ignore leap years. However, don't accept a month that is not in the [1,12] range or day of the month that is not in the [1,31] range. Test each version with at least one invalid date (e.g., 2004, 13, -5).

1. The version from §9.4.1
2. The version from §9.4.2
3. The version from §9.4.3
4. The version from §9.7.1
5. The version from §9.7.4

Review

1. What are the two parts of a class, as described in the chapter?
2. What is the difference between the interface and the implementation in a class?
3. What are the limitations and problems of the original **Date struct** that is created in the chapter?

4. Why is a constructor used for the `Date` type instead of an `init_day()` function?
5. What is an invariant? Give examples.
6. When should functions be put in the class definition, and when should they be defined outside the class? Why?
7. When should operator overloading be used in a program? Give a list of operators that you might want to overload (each with a reason).
8. Why should the public interface to a class be as small as possible?
9. What does adding `const` to a member function do?
10. Why are “helper functions” best placed outside the class definition?

Terms

built-in types	enumerator	representation
class	helper function	struct
const	implementation	structure
constructor	in-class initializer	user-defined types
destructor	inlining	valid state
enum	interface	
enumeration	invariant	

Exercises

1. List sets of plausible operations for the examples of real-world objects in §9.1 (such as toaster).
2. Design and implement a `Name_pairs` class holding (name,age) pairs where name is a `string` and age is a `double`. Represent that as a `vector<string>` (called `name`) and a `vector<double>` (called `age`) member. Provide an input operation `read_names()` that reads a series of names. Provide a `read_ages()` operation that prompts the user for an age for each name. Provide a `print()` operation that prints out the `(name[i],age[i])` pairs (one per line) in the order determined by the `name` vector. Provide a `sort()` operation that sorts the `name` vector in alphabetical order and reorganizes the `age` vector to match. Implement all “operations” as member functions. Test the class (of course: test early and often).
3. Replace `Name_pair::print()` with a (global) operator `<<` and define `==` and `!=` for `Name_pairs`.
4. Look at the headache-inducing last example of §8.4. Indent it properly and explain the meaning of each construct. Note that the example doesn’t do anything meaningful; it is pure obfuscation.
5. This exercise and the next few require you to design and implement a `Book` class, such as you can imagine as part of software for a library. Class

- Book** should have members for the ISBN, title, author, and copyright date. Also store data on whether or not the book is checked out. Create functions for returning those data values. Create functions for checking a book in and out. Do simple validation of data entered into a **Book**; for example, accept ISBNs only of the form **n-n-n-x** where **n** is an integer and **x** is a digit or a letter. Store an ISBN as a **string**.
6. Add operators for the **Book** class. Have the **==** operator check whether the ISBN numbers are the same for two books. Have **!=** also compare the ISBN numbers. Have a **<<** print out the title, author, and ISBN on separate lines.
 7. Create an enumerated type for the **Book** class called **Genre**. Have the types be fiction, nonfiction, periodical, biography, and children. Give each book a **Genre** and make appropriate changes to the **Book** constructor and member functions.
 8. Create a **Patron** class for the library. The class will have a user's name, library card number, and library fees (if owed). Have functions that access this data, as well as a function to set the fee of the user. Have a helper function that returns a Boolean (**bool**) depending on whether or not the user owes a fee.
 9. Create a **Library** class. Include vectors of **Books** and **Patrons**. Include a **struct** called **Transaction**. Have it include a **Book**, a **Patron**, and a **Date** from the chapter. Make a vector of **Transactions**. Create functions to add books to the library, add patrons to the library, and check out books. Whenever a user checks out a book, have the library make sure that both the user and the book are in the library. If they aren't, report an error. Then check to make sure that the user owes no fees. If the user does, report an error. If not, create a **Transaction**, and place it in the vector of **Transactions**. Also write a function that will return a vector that contains the names of all **Patrons** who owe fees.
 10. Implement **leapyear()** from §9.8.
 11. Design and implement a set of useful helper functions for the **Date** class with functions such as **next_workday()** (assume that any day that is not a Saturday or a Sunday is a workday) and **week_of_year()** (assume that week 1 is the week with January 1 in it and that the first day of a week is a Sunday).
 12. Change the representation of a **Date** to be the number of days since January 1, 1970 (known as day 0), represented as a **long int**, and re-implement the functions from §9.8. Be sure to reject dates outside the range we can represent that way (feel free to reject days before day 0, i.e., no negative days).
 13. Design and implement a rational number class, **Rational**. A rational number has two parts: a numerator and a denominator, for example, 5/6

(five-sixths, also known as approximately .83333). Look up the definition if you need to. Provide assignment, addition, subtraction, multiplication, division, and equality operators. Also, provide a conversion to **double**. Why would people want to use a **Rational** class?

14. Design and implement a **Money** class for calculations involving dollars and cents where arithmetic has to be accurate to the last cent using the 4/5 rounding rule (.5 of a cent rounds up; anything less than .5 rounds down). Represent a monetary amount as a number of cents in a **long int**, but input and output as dollars and cents, e.g., \$123.45. Do not worry about amounts that don't fit into a **long int**.
15. Refine the **Money** class by adding a currency (given as a constructor argument). Accept a floating-point initializer as long as it can be exactly represented as a **long int**. Don't accept illegal operations. For example, **Money*Money** doesn't make sense, and **USD1.23+DKK5.00** makes sense only if you provide a conversion table defining the conversion factor between U.S. dollars (USD) and Danish kroner (DKK).
16. Define an input operator (**>>**) that reads monetary amounts with currency denominations, such as **USD1.23** and **DKK5.00**, into a **Money** variable. Also define a corresponding output operator (**>>**).
17. Give an example of a calculation where a **Rational** gives a mathematically better result than **Money**.
18. Give an example of a calculation where a **Rational** gives a mathematically better result than **double**.

Postscript

There is a lot to user-defined types, much more than we have presented here. User-defined types, especially classes, are the heart of C++ and the key to many of the most effective design techniques. Most of the rest of the book is about the design and use of classes. A class – or a set of classes – is the mechanism through which we represent our concepts in code. Here we primarily introduced the language-technical aspects of classes; elsewhere we focus on how to elegantly express useful ideas as classes.

Part II

Input and Output



Input and Output Streams

“Science is what we have learned about how to keep from fooling ourselves.”

—Richard P. Feynman

In this chapter and the next, we present the C++ standard library facilities for handling input and output from a variety of sources: I/O streams. We show how to read and write files, how to deal with errors, how to deal with formatted input, and how to provide and use I/O operators for user-defined types. This chapter focuses on the basic model: how to read and write individual values, and how to open, read, and write whole files. The final example illustrates the kinds of considerations that go into a larger piece of code. The next chapter addresses details.

10.1 Input and output	10.8 User-defined output operators
10.2 The I/O stream model	10.9 User-defined input operators
10.3 Files	10.10 A standard input loop
10.4 Opening a file	10.11 Reading a structured file
10.5 Reading and writing a file	10.11.1 In-memory representation
10.6 I/O error handling	10.11.2 Reading structured values
10.7 Reading a single value	10.11.3 Changing representations
10.7.1 Breaking the problem into manageable parts	
10.7.2 Separating dialog from function	

10.1 Input and output



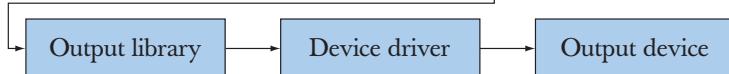
Without data, computing is pointless. We need to get data into our program to do interesting computations and we need to get the results out again. In §4.1, we mentioned the bewildering variety of data sources and targets for output. If we don't watch out, we'll end up writing programs that can receive input only from a specific source and deliver output only to a specific output device. That may be acceptable (and sometimes even necessary) for specialized applications, such as a digital camera or a sensor for an engine fuel injector, but for more common tasks, we need a way to separate the way our program reads and writes from the actual input and output devices used. If we had to directly address each kind of device, we'd have to change our program each time a new screen or disk came on the market, or limit our users to the screens and disks we happen to like. That would be absurd.

Most modern operating systems separate the detailed handling of I/O devices into device drivers, and programs then access the device drivers through an I/O library that makes I/O from/to different sources appear as similar as possible. Generally, the device drivers are deep in the operating system where most users don't see them, and the I/O library provides an abstraction of I/O so that the programmer doesn't have to think about devices and device drivers:

Data source:



Data destination:



When a model like this is used, input and output can be seen as streams of bytes (characters) handled by the input/output library. More complex forms of I/O require specialized expertise and are beyond the scope of this book. Our job as programmers of an application then becomes

1. To set up I/O streams to the appropriate data sources and destinations
2. To read and write from/to those streams

The details of how our characters are actually transmitted to/from the devices are dealt with by the I/O library and the device drivers. In this chapter and the next, we'll see how I/O consisting of streams of formatted data is done using the C++ standard library.

From the programmer's point of view there are many different kinds of input and output. One classification is

- Streams of (many) data items (usually to/from files, network connections, recording devices, or display devices)
- Interactions with a user at a keyboard
- Interactions with a user through a graphical interface (outputting objects, receiving mouse clicks, etc.)

This classification isn't the only classification possible, and the distinction between the three kinds of I/O isn't as clear as it might appear. For example, if a stream of output characters happens to be an HTTP document aimed at a browser, the result looks remarkably like user interaction and can contain graphical elements. Conversely, the results of interactions with a GUI (graphical user interface) may be presented to a program as a sequence of characters. However, this classification fits our tools: the first two kinds of I/O are provided by the C++ standard library I/O streams and supported rather directly by most operating systems. We have been using the `iostream` library since Chapter 1 and will focus on that for this and the next chapter. The graphical output and graphical user interactions are served by a variety of different libraries, and we will focus on that kind of I/O in Chapters 12 to 16.

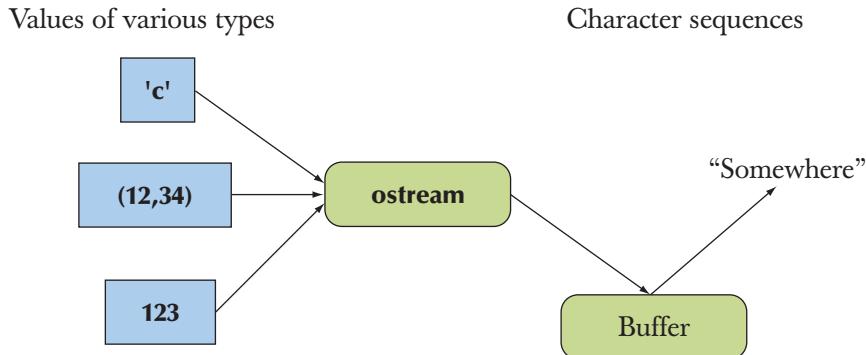
10.2 The I/O stream model

The C++ standard library provides the type `istream` to deal with streams of input and the type `ostream` to deal with streams of output. We have used the standard `istream` called `cin` and the standard `ostream` called `cout`, so we know the basics of how to use this part of the standard library (usually called the `iostream` library).

An `ostream`

- Turns values of various types into character sequences
- Sends those characters "somewhere" (such as to a console, a file, the main memory, or another computer)

We can represent an **ostream** graphically like this:

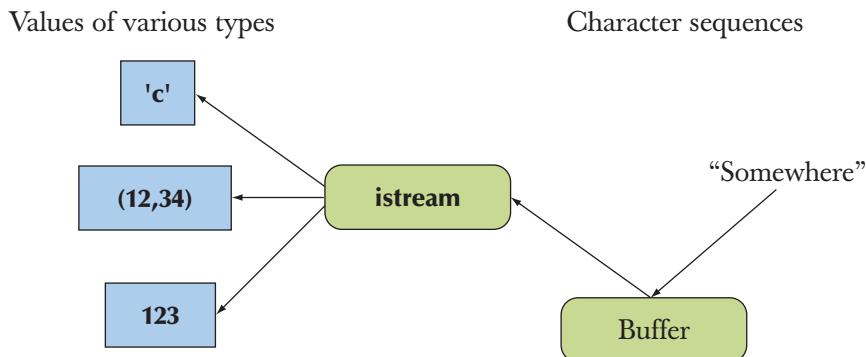


The buffer is a data structure that the **ostream** uses internally to store the data you give it while communicating with the operating system. If you notice a “delay” between your writing to an **ostream** and the characters appearing at their destination, it’s usually because they are still in the buffer. Buffering is important for performance, and performance is important if you deal with large amounts of data.

An **istream**

- Turns character sequences into values of various types
- Gets those characters from somewhere (such as a console, a file, the main memory, or another computer)

We can represent an **istream** graphically like this:



As with an **ostream**, an **istream** uses a buffer to communicate with the operating system. With an **istream**, the buffering can be quite visible to the user. When you use an **istream** that is attached to a keyboard, what you type is left in the buffer until you hit Enter (return/newline), and you can use the erase (Backspace) key “to change your mind” (until you hit Enter).

One of the major uses of output is to produce data for humans to read. Think of email messages, scholarly articles, web pages, billing records, business reports, contact lists, tables of contents, equipment status readouts, etc. Therefore, **ostreams** provide many features for formatting text to suit various tastes. Similarly, much input is written by humans or is formatted to make it easy for humans to read it. Therefore, **istreams** provide features for reading the kind of output produced by **ostreams**. We'll discuss formatting in §11.2 and how to read non-character input in §11.3.2. Most of the complexity related to input has to do with how to handle errors. To be able to give more realistic examples, we'll start by discussing how the iostream model relates to files of data.

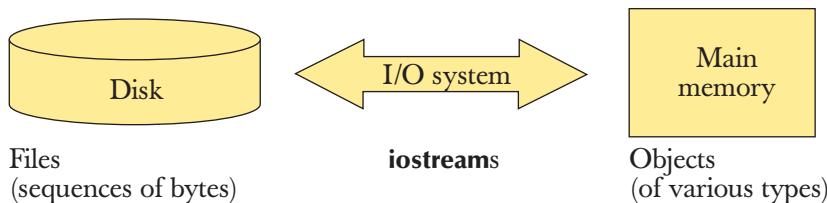
10.3 Files

We typically have much more data than can fit in the main memory of our computer, so we store most of it on disks or other large-capacity storage devices. Such devices also have the desirable property that data doesn't disappear when the power is turned off – the data is persistent. At the most basic level, a file is simply a sequence of bytes numbered from 0 upward:



A file has a format; that is, it has a set of rules that determine what the bytes mean. For example, if we have a text file, the first 4 bytes will be the first four characters. On the other hand, if we have a file that uses a binary representation of integers, those very same first 4 bytes will be taken to be the (binary) representation of the first integer (see §11.3.2). The format serves the same role for files on disk as types serve for objects in main memory. We can make sense of the bits in a file if (and only if) we know its format (see §11.2–3).

For a file, an **ostream** converts objects in main memory into streams of bytes and writes them to disk. An **istream** does the opposite; that is, it takes a stream of bytes from disk and composes objects from them:



Most of the time, we assume that these “bytes on disk” are in fact characters in our usual character set. That is not always so, but we can get an awfully long way

with that assumption, and other representations are not that hard to deal with. We also talk as if all files were on disks (that is, on rotating magnetic storage). Again, that's not always so (think of flash memory), but at this level of programming the actual storage makes no difference. That's one of the beauties of the file and stream abstractions.

To read a file, we must

1. Know its name
2. Open it (for reading)
3. Read in the characters
4. Close it (though that is typically done implicitly)

To write a file, we must

1. Name it
2. Open it (for writing) or create a new file of that name
3. Write out our objects
4. Close it (though that is typically done implicitly)

We already know the basics of reading and writing because an **ostream** attached to a file behaves exactly as **cout** for what we have done so far, and an **istream** attached to a file behaves exactly as **cin** for what we have done so far. We'll present operations that can only be done for files later (§11.3.3), but for now we'll just see how to open files and then concentrate on operations and techniques that apply to all **ostreams** and all **istreams**.

10.4 Opening a file

If you want to read from a file or write to a file you have to open a stream specifically for that file. An **ifstream** is an **istream** for reading from a file, an **ofstream** is an **ostream** for writing to a file, and an **fstream** is an **iostream** that can be used for both reading and writing. Before a file stream can be used it must be attached to a file. For example:

```
cout << "Please enter input file name: ";
string iname;
cin >> iname;
ifstream ist {iname};      // ist is an input stream for the file named name
if (!ist) error("can't open input file ",iname);
```

Defining an **ifstream** with a name string opens the file of that name for reading. The test of **!ist** checks if the file was properly opened. After that, we can read from

the file exactly as we would from any other **istream**. For example, assuming that the input operator, **>>**, was defined for a type **Point**, we could write

```
vector<Point> points;
for (Point p; ist>>p; )
    points.push_back(p);
```

Output to files is handled in a similar fashion by **ofstreams**. For example:

```
cout << "Please enter name of output file: ";
string oname;
cin >> oname;
ofstream ost {oname};      // ost is an output stream for a file named oname
if (!ost) error("can't open output file ",oname);
```

Defining an **ofstream** with a name string opens the file with that name for writing. The test of **!ost** checks if the file was properly opened. After that, we can write to the file exactly as we would to any other **ostream**. For example:

```
for (int p : points)
    ost << '(' << p.x << ',' << p.y << ")\\n";
```

When a file stream goes out of scope its associated file is closed. When a file is closed its associated buffer is “flushed”; that is, the characters from the buffer are written to the file.

It is usually best to open files early in a program before any serious computation has taken place. After all, it is a waste to do a lot of work just to find that we can’t complete it because we don’t have anywhere to write our results.

Opening the file implicitly as part of the creation of an **ostream** or an **istream** and relying on the scope of the stream to take care of closing the file is the ideal. For example:

```
void fill_from_file(vector<Point>& points, string& name)
{
    ifstream ist {name};      // open file for reading
    if (!ist) error("can't open input file ",name);
    // . . . use ist . . .
    // the file is implicitly closed when we leave the function
}
```



You can also perform explicit `open()` and `close()` operations (§B.7.1). However, relying on scope minimizes the chances of someone trying to use a file stream before it has been attached to a stream or after it was closed. For example:

```
ifstream ifs;
// ...
ifs >> foo;           // won't succeed: no file opened for ifs
// ...
ifs.open(name,ios_base::in); // open file named name for reading
// ...
ifs.close();          // close file
// ...
ifs >> bar;          // won't succeed: ifs' file was closed
// ...
```

In real-world code the problems would typically be much harder to spot. Fortunately, you can't open a file stream a second time without first closing it. For example:

```
fstream fs;
fs.open("foo", ios_base::in) ; // open for input
// close() missing
fs.open("foo", ios_base::out); // won't succeed: fs is already open
if (!fs) error("impossible");
```

Don't forget to test a stream after opening it.

Why would you use `open()` or `close()` explicitly? Well, occasionally the lifetime of a connection to a file isn't conveniently limited by a scope so you have to. But that's rare enough for us not to have to worry about it here. More to the point, you'll find such use in code written by people using styles from languages and libraries that don't have the scoped idiom used by `iostreams` (and the rest of the C++ standard library).

As we'll see in Chapter 11, there is much more to files, but for now we know enough to use them as a data source and a destination for data. That'll allow us to write programs that would be unrealistic if we assumed that a user had to directly type in all the input. From a programmer's point of view, a great advantage of a file is that you can repeatedly read it during debugging until your program works correctly.

10.5 Reading and writing a file

Consider how you might read a set of results of some measurements from a file and represent them in memory. These might be the temperature readings from a weather station:

```
0 60.7
1 60.6
2 60.3
3 59.22
...
```

This data file contains a sequence of (hour,temperature) pairs. The hours are numbered **0** to **23** and the temperatures are in Fahrenheit. No further formatting is assumed; that is, the file does not contain any special header information (such as where the reading was taken), units for the values, punctuation (such as parentheses around each pair of values), or termination indicator. This is the simplest case.

We could represent a temperature reading by a **Reading** type:

```
struct Reading {           // a temperature reading
    int hour;             // hour after midnight [0:23]
    double temperature;   // in Fahrenheit
};
```

Given that, we could read like this:

```
vector<Reading> temps;      // store the readings here
int hour;
double temperature;
while (ist >> hour >> temperature) {
    if (hour < 0 || 23 < hour) error("hour out of range");
    temps.push_back(Reading{hour,temperature});
}
```

This is a typical input loop. The **istream** called **ist** could be an input file stream (**ifstream**) as shown in the previous section, (an alias for) the standard input stream (**cin**), or any other kind of **istream**. For code like this, it doesn't matter exactly from where the **istream** gets its data. All that our program cares about is that **ist** is an **istream** and that the data has the expected format. The next section addresses the interesting question of how to detect errors in the input data and what we can do after detecting a format error.

Writing to a file is usually simpler than reading from one. Again, once a stream is initialized we don't have to know exactly what kind of stream it is. In particular, we can use the output file stream (**ofstream**) from the section above just like any other **ostream**. For example, we might want to output the readings with each pair of values in parentheses:

```
for (int i=0; i<temps.size(); ++i)
    ost << '(' << temps[i].hour << ',' << temps[i].temperature << ")\n";
```

The resulting program would then be reading the original temperature reading file and producing a new file with the data in (hour,temperature) format.

Because the file streams automatically close their files when they go out of scope, the complete program becomes

```
#include "std_lib_facilities.h"

struct Reading {                                // a temperature reading
    int hour;                                // hour after midnight [0:23]
    double temperature;                      // in Fahrenheit
};

int main()
{
    cout << "Please enter input file name: ";
    string iname;
    cin >> iname;
    ifstream ist {iname};           // ist reads from the file named iname
    if (!ist) error("can't open input file ",iname);

    string oname;
    cout << "Please enter name of output file: ";
    cin >> oname;
    ofstream ost {oname};           // ost writes to a file named oname
    if (!ost) error("can't open output file ",oname);

    vector<Reading> temps;      // store the readings here
    int hour;
    double temperature;
    while (ist >> hour >> temperature) {
        if (hour < 0 || 23 < hour) error("hour out of range");
        temps.push_back(Reading{hour,temperature});
    }

    for (int i=0; i<temps.size(); ++i)
        ost << '(' << temps[i].hour << ','
            << temps[i].temperature << ")\\n";
}
```

10.6 I/O error handling

When dealing with input we must expect errors and deal with them. What kind of errors? And how? Errors occur because humans make mistakes (misunderstand-

ing instructions, mistyping, letting the cat walk on the keyboard, etc.), because files fail to meet specifications, because we (as programmers) have the wrong expectations, etc. The possibilities for input errors are limitless! However, an **istream** reduces all to four possible cases, called the *stream state*:

Stream states	
good()	The operations succeeded.
eof()	We hit end of input (“end of file”).
fail()	Something unexpected happened (e.g., we looked for a digit and found ' x ').
bad()	Something unexpected and serious happened (e.g., a disk read error).

Unfortunately, the distinction between **fail()** and **bad()** is not precisely defined and subject to varying opinions among programmers defining I/O operations for new types. However, the basic idea is simple: If an input operation encounters a simple format error, it lets the stream **fail()**, assuming that you (the user of our input operation) might be able to recover. If, on the other hand, something really nasty, such as a bad disk read, happens, the input operation lets the stream go **bad()**, assuming that there is nothing much you can do except to abandon the attempt to get data from that stream. A stream that is **bad()** is also **fail()**. This leaves us with this general logic:

```
int i = 0;
cin >> i;
if (!cin) { // we get here (only) if an input operation failed
    if (cin.fail()) error("cin is bad"); // stream corrupted: let's get out of here!
    if (cin.eof()) {
        // no more input
        // this is often how we want a sequence of input operations to end
    }
    if (cin.fail()) { // stream encountered something unexpected
        cin.clear(); // make ready for more input
        // somehow recover
    }
}
```

The **!cin** can be read as “**cin** is not good” or “Something went wrong with **cin**” or “The state of **cin** is not **good()**.” It is the opposite of “The operation succeeded.” Note the **cin.clear()** where we handle **fail()**. When a stream has failed, we might be able to recover. To try to recover, we explicitly take the stream out of the **fail()** state, so that we can look at characters from it again; **clear()** does that – after **cin.clear()** the state of **cin** is **good()**.

Here is an example of how we might use the stream state. Consider how to read a sequence of integers that may be terminated by the character * or an “end of file” (Ctrl+Z on Windows, Ctrl+D on Unix) into a `vector`. For example:

```
1 2 3 4 5 *
```

This could be done using a function like this:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
    // read integers from ist into v until we reach eof() or terminator
{
    for (int i; ist >> i; ) v.push_back(i);
    if (ist.eof()) return;           // fine: we found the end of file

    if (ist.bad()) error("ist is bad"); // stream corrupted; let's get out of here!
    if (ist.fail()) { // clean up the mess as best we can and report the problem
        ist.clear(); // clear stream state,
                      // so that we can look for terminator
        char c;
        ist>>c;      // read a character, hopefully terminator
        if (c != terminator) { // unexpected character
            ist.unget(); // put that character back
            ist.clear(ios_base::failbit); // set the state to fail()
        }
    }
}
```

Note that when we didn’t find the terminator, we still returned. After all, we may have collected some data and the caller of `fill_vector()` may be able to recover from a `fail()`. Since we cleared the state to be able to examine the character, we have to set the stream state back to `fail()`. We do that with `ist.clear(ios_base::failbit)`. Note this potentially confusing use of `clear()`: `clear()` with an argument actually sets the `iostream` state flags (bits) mentioned and (only) clears flags not mentioned. By setting the state to `fail()`, we indicate that we encountered a format error, rather than something more serious. We put the character back into `ist` using `unget()`; the caller of `fill_vector()` might have a use for it. The `unget()` function is a shorter version of `putback()` (§6.8.2, §B.7.3) that relies on the stream remembering which character it last produced, so that you don’t have to mention it.

If you called `fill_vector()` and want to know what terminated the read, you can test for `fail()` and `eof()`. You could also catch the `runtime_error` exception thrown by `error()`, but it is understood that getting more data from `istream` in the `bad()` state is unlikely. Most callers won’t bother. This implies that in almost all



cases the only thing we want to do if we encounter **bad()** is to throw an exception. To make life easier, we can tell an **istream** to do that for us:

```
// make ist throw if it goes bad
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

The notation may seem odd, but the effect is simply that from that statement onward, **ist** will throw the standard library exception **ios_base::failure** if it goes **bad()**. We need to execute that **exceptions()** call only once in a program. That'll allow us to simplify all input loops on **ist** by ignoring **bad()**:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
    // read integers from ist into v until we reach eof() or terminator
{
    for (int i; ist >> i; ) v.push_back(i);
    if (ist.eof()) return; // fine: we found the end of file

    // not good() and not bad() and not eof(), ist must be fail()
    ist.clear();           // clear stream state

    char c;
    ist>>c;              // read a character, hopefully terminator

    if (c != terminator) { // ouch: not the terminator, so we must fail
        ist.unget();       // maybe my caller can use that character
        ist.clear(ios_base::failbit); // set the state to fail()
    }
}
```

The **ios_base** that appears here and there is the part of an **iostream** that holds constants such as **badbit**, exceptions such as **failure**, and other useful stuff. You refer to them using the **::** operator, for example, **ios_base::badbit** (§B.7.2). We don't plan to go into the **iostream** library in that much detail; it could take a whole course to explain all of **iostreams**. For example, **iostreams** can handle different character sets, implement different buffering strategies, and also contain facilities for formatting monetary amounts in various languages; we once had a bug report relating to the formatting of Ukrainian currency. You can read up on whatever bits you need to know about if you need to; see *The C++ Programming Language* by Stroustrup and *Standard C++ IOStreams and Locales* by Langer.

You can test an **ostream** for exactly the same states as an **istream: good()**, **fail()**, **eof()**, and **bad()**. However, for the kinds of programs we write here, errors are much rarer for output than for input, so we don't do it as often. For programs where output devices have a more significant chance of being unavailable, filled,

or broken, we would test after each output operation just as we test after each input operation.

10.7 Reading a single value

So, we know how to read a series of values ending with the end of file or a terminator. We'll show more examples as we go along, but let's just have a look at the ever popular idea of repeatedly asking for a value until an acceptable one is entered. This example will allow us to examine several common design choices. We'll discuss these alternatives through a series of alternative solutions to the simple problem of "how to get an acceptable value from the user." We start with an unpleasantly messy obvious "first try" and proceed through a series of improved versions. Our fundamental assumption is that we are dealing with interactive input where a human is typing input and reading the messages from the program. Let's ask for an integer in the range 1 to 10 (inclusive):

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n) { // read
    if (1<=n && n<=10) break; // check range
    cout << "Sorry "
    << n << " is not in the [1:10] range; please try again\n";
}
// ... use n here ...
```

This is pretty ugly, but it "sort of works." If you don't like using the **break** (§A.6), you can combine the reading and the range checking:

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n && !(1<=n && n<=10)) // read and check range
    cout << "Sorry "
    << n << " is not in the [1:10] range; please try again\n";
// ... use n here ...
```

However, that's just a cosmetic change. Why does it only "sort of work"? It works if the user carefully enters integers. If the user is a poor typist and hits **t** rather than **6** (**t** is just below **6** on most keyboards), the program will leave the loop without changing the value of **n**, so that **n** will have an out-of-range value. We wouldn't call that quality code. A joker (or a diligent tester) might also send an "end of file" from the keyboard (Ctrl+Z on a Windows machine and Ctrl+D on a Unix machine). Again, we'd leave the loop with **n** out of range. In other words, to get a robust read we have to deal with three problems:

1. The user typing an out-of-range value
2. Getting no value (end of file)
3. The user typing something of the wrong type (here, not an integer)

What do we want to do in those three cases? That's often the question when writing a program: What do we really want? Here, for each of those three errors, we have three alternatives:

1. Handle the problem in the code doing the read.
2. Throw an exception to let someone else handle the problem (potentially terminating the program).
3. Ignore the problem.

As it happens, those are three very common alternatives for dealing with an error condition. Thus, this is a good example of the kind of thinking we have to do about errors.

It is tempting to say that the third alternative, ignoring the problem, is always unacceptable, but that would be patronizing. If I'm writing a trivial program for my own use, I can do whatever I like, including forgetting about error checking with potential nasty results. However, for a program that I might want to use for more than a few hours after I wrote it, I would probably be foolish to leave such errors, and if I want to share that program with anyone, I should not leave such holes in the error checking in the code. Please note that we deliberately use the first-person singular here; “we” would be misleading. We do not consider alternative 3 acceptable even when just two people are involved.

The choice between alternatives 1 and 2 is genuine; that is, in a given program there can be good reasons to choose either way. First we note that in most programs there is no local and elegant way to deal with no input from a user sitting at the keyboard: after the input stream is closed, there isn't much point in asking the user to enter a number. We could reopen `cin` (using `cin.clear()`), but the user is unlikely to have closed that stream by accident (how would you hit Ctrl+Z by accident?). If the program wants an integer and finds “end of file,” the part of the program trying to read the integer must usually give up and hope that some other part of the program can cope; that is, our code requesting input from the user must throw an exception. This implies that the choice is not between throwing exceptions and handling problems locally, but a choice of which problems (if any) we should handle locally.

10.7.1 Breaking the problem into manageable parts

Let's try handling both an out-of-range input and an input of the wrong type locally:

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
```

```

while (true) {
    cin >> n;
    if (cin) {           // we got an integer; now check it
        if (1<=n && n<=10) break;
        cout << "Sorry "
            << n << " is not in the [1:10] range; please try again\n";
    }
    else if (cin.fail()) {      // we found something that wasn't an integer
        cin.clear();          // set the state back to good();
                                // we want to look at the characters
        cout << "Sorry, that was not a number; please try again\n";
        for (char ch; cin>>ch && !isdigit(ch); ) // throw away non-digits
            /* nothing */ ;
        if (!cin) error("no input"); // we didn't find a digit: give up
        cin.unget(); // put the digit back, so that we can read the number
    }
    else {
        error("no input");           // eof or bad: give up
    }
}
// if we get here n is in [1:10]

```

This is messy, and rather long-winded. In fact, it is so messy that we could not recommend that people write such code each time they needed an integer from a user. On the other hand, we do need to deal with the potential errors because people do make them, so what can we do? The reason that the code is messy is that code dealing with several different concerns is all mixed together:

- Reading values
- Prompting the user for input
- Writing error messages
- Skipping past “bad” input characters
- Testing the input against a range

The way to make code clearer is often to separate logically distinct concerns into separate functions. For example, we can separate out the code for recovering after seeing a “bad” (i.e., unexpected) character:

```

void skip_to_int()
{
    if (cin.fail()) {           // we found something that wasn't an integer
        cin.clear();          // we'd like to look at the characters
    }
}

```

```

        for (char ch; cin>>ch; ) { // throw away non-digits
            if (isdigit(ch) || ch=='-') {
                cin.unget(); // put the digit back,
                               // so that we can read the number
                return;
            }
        }
    }
    error("no input"); // eof or bad: give up
}

```

Given the `skip_to_int()` “utility function,” we can write

```

cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (true) {
    if (cin>>n) { // we got an integer; now check it
        if (1<=n && n<=10) break;
        cout << "Sorry " << n
              << " is not in the [1:10] range; please try again\n";
    }
    else {
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
// if we get here n is in [1:10]

```

This code is better, but it is still too long and too messy to use many times in a program. We’d never get it consistently right, except after (too) much testing.

What operation would we really like to have? One plausible answer is “a function that reads an `int`, any `int`, and another that reads an `int` of a given range”:

```

int get_int(); // read an int from cin
int get_int(int low, int high); // read an int in [low:high] from cin

```

If we had those, we would at least be able to use them simply and correctly. They are not that hard to write:

```

int get_int()
{
    int n = 0;
    while (true) {

```

```

    if (cin >> n) return n;
    cout << "Sorry, that was not a number; please try again\n";
    skip_to_int();
}
}

```

Basically, `get_int()` stubbornly keeps reading until it finds some digits that it can interpret as an integer. If we want to get out of `get_int()`, we must supply an integer or end of file (and end of file will cause `get_int()` to throw an exception).

Using that general `get_int()`, we can write the range-checking `get_int()`:

```

int get_int(int low, int high)
{
    cout << "Please enter an integer in the range "
        << low << " to " << high << " (inclusive):\n";

    while (true) {
        int n = get_int();
        if (low <= n && n <= high) return n;
        cout << "Sorry "
            << n << " is not in the [" << low << ':' << high
            << "] range; please try again\n";
    }
}

```

This `get_int()` is as stubborn as the other. It keeps getting `ints` from the non-range `get_int()` until the `int` it gets is in the expected range.

We can now reliably read integers like this:

```

int n = get_int(1,10);
cout << "n: " << n << '\n';

int m = get_int(2,300);
cout << "m: " << m << '\n';

```

Don't forget to catch exceptions somewhere, though, if you want decent error messages for the (probably rare) case when `get_int()` really couldn't read a number for us.

10.7.2 Separating dialog from function

The `get_int()` functions still mix up reading with writing messages to the user. That's probably good enough for a simple program, but in a large program we might want to vary the messages written to the user. We might want to call `get_int()` like this:

```
int strength = get_int(1,10, "enter strength", "Not in range, try again");
cout << "strength: " << strength << '\n';

int altitude = get_int(0,50000,
    "Please enter altitude in feet",
    "Not in range, please try again");
cout << "altitude: " << altitude << "f above sea level\n";
```

We could implement that like this:

```
int get_int(int low, int high, const string& greeting, const string& sorry)
{
    cout << greeting << ": [" << low << ':' << high << "]\n";

    while (true) {
        int n = get_int();
        if (low <= n && n <= high) return n;
        cout << sorry << ": [" << low << ':' << high << "]\n";
    }
}
```

It is hard to compose arbitrary messages, so we “stylized” the messages. That’s often acceptable, and composing really flexible messages, such as are needed to support many natural languages (e.g., Arabic, Bengali, Chinese, Danish, English, and French), is not a task for a novice.

Note that our solution is still incomplete: the `get_int()` without a range still “blabbers.” The deeper point here is that “utility functions” that we use in many parts of a program shouldn’t have messages “hardwired” into them. Further, library functions that are meant for use in many programs shouldn’t write to the user at all – after all, the library writer may not even know that the program in which the library runs is used on a machine with a human watching. That’s one reason that our `error()` function doesn’t just write an error message (§5.6.3); in general, we wouldn’t know where to write.

10.8 User-defined output operators

Defining the output operator, `<<`, for a given type is typically trivial. The main design problem is that different people might prefer the output to look different, so it is hard to agree on a single format. However, even if no single output format is good enough for all uses, it is often a good idea to define `<<` for a user-defined type. That way, we can at least trivially write out objects of the type during debugging and early development. Later, we might provide a more sophisticated `<<` that allows a user to provide formatting information. Also, if we want

output that looks different from what a `<<` provides, we can simply bypass the `<<` and write out the individual parts of the user-defined type the way we happen to like them in our application.

Here is a simple output operator for `Date` from §9.8 that simply prints the year, month, and day comma-separated in parentheses:

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
        << ',' << d.month()
        << ',' << d.day() << ')';
}
```

This will print August 30, 2004, as `(2004,8,30)`. This simple list-of-elements representation is what we tend to use for types with a few members unless we have a better idea or more specific needs.

In §9.6, we mention that a user-defined operator is handled by calling its function. Here we can see an example of how that's done. Given the definition of `<<` for `Date`, the meaning of

```
cout << d1;
```

where `d1` is a `Date` is the call

```
operator<<(cout,d1);
```

Note how `operator<<()` takes an `ostream&` as its first argument and returns it again as its return value. That's the way the output stream is passed along so that you can “chain” output operations. For example, we could output two dates like this:

```
cout << d1 << d2;
```

This will be handled by first resolving the first `<<` and after that the second `<<:`

```
cout << d1 << d2;      // means operator<<(cout,d1) << d2;
                         // means operator<<(operator<<(cout,d1),d2);
```

That is, first output `d1` to `cout` and then output `d2` to the output stream that is the result of the first output operation. In fact, we can use any of those three variants to write out `d1` and `d2`. We know which one is easier to read, though.

10.9 User-defined input operators

Defining the input operator, `>>`, for a given type and input format is basically an exercise in error handling. It can therefore be quite tricky.

Here is a simple input operator for the `Date` from §9.8 that will read dates as written by the operator `<<` defined above:

```
istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1 != ' ' || ch2 != ' ' || ch3 != ' ' || ch4 != ' ') { // oops: format error
        is.clear(ios_base::failbit);
        return is;
    }
    dd = Date{y,Date::Month(m),d}; // update dd
    return is;
}
```

This `>>` will read items like `(2004,8,20)` and try to make a `Date` out of those three integers. As ever, input is harder to deal with than output. There is simply more that can – and often does – go wrong with input than with output.

If this `>>` doesn't find something in the `(integer, integer, integer)` format, it will leave the stream in a not-good state (`fail`, `eof`, or `bad`) and leave the target `Date` unchanged. The `clear()` member function is used to set the state of the `istream`. Obviously, `ios_base::failbit` puts the stream into the `fail()` state. Leaving the target `Date` unchanged in case of a failure to read is the ideal; it tends to lead to cleaner code. The ideal is for an `operator>>()` not to consume (throw away) any characters that it didn't use, but that's too difficult in this case: we might have read lots of characters before we caught a format error. As an example, consider `(2004, 8, 30)`. Only when we see the final `}` do we know that we have a format error on our hands and we cannot in general rely on putting back many characters. One character `unget()` is all that's universally guaranteed. If this `operator>>()` reads an invalid `Date`, such as `(2004,8,32)`, `Date`'s constructor will throw an exception, which will get us out of this `operator>>()`.

10.10 A standard input loop

In §10.5, we saw how we could read and write files. However, that was before we looked more carefully at errors (§10.6), so the input loop simply assumed that

we could read a file from its beginning until end of file. That can be a reasonable assumption, because we often apply separate checks to ensure that a file is valid. However, we often want to check our reads as we go along. Here is a general strategy, assuming that `ist` is an `istream`:

```
for (My_type var; ist>>var; ) {    // read until end of file
    // maybe check that var is valid
    // do something with var
}
// we can rarely recover from bad; don't try unless you really have to:
if (ist.bad()) error("bad input stream");
if (ist.fail() {
    // was it an acceptable terminator?
}
// carry on: we found end of file
```

That is, we read a sequence of values into variables and when we can't read any more values, we check the stream state to see why. As in §10.6, we can improve this a bit by letting the `istream` throw an exception of type `failure` if it goes bad. That saves us the bother of checking for it all the time:

```
// somewhere: make ist throw an exception if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

We could also decide to designate a character as a terminator:

```
for (My_type var; ist>>var; ) {    // read until end of file
    // maybe check that var is valid
    // do something with var
}
if (ist.fail() {      // use ';' as terminator and/or separator
    ist.clear();
    char ch;
    if (!(ist>>ch && ch==';)) error("bad termination of input");
}
// carry on: we found end of file or a terminator
```

If we don't want to accept a terminator – that is, to accept only end of file as the end – we simply delete the test before the call of `error()`. However, terminators are very useful when you read files with nested constructs, such as a file of monthly readings containing daily readings, containing hourly readings, etc., so we'll keep considering the possibility of a terminating character.

Unfortunately, that code is still a bit messy. In particular, it is tedious to repeat the terminator test if we read a lot of files. We could write a function to deal with that:

```
// somewhere: make ist throw if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);

void end_of_loop(istream& ist, char term, const string& message)
{
    if (ist.fail()) {           // use term as terminator and/or separator
        ist.clear();
        char ch;
        if (ist>>ch && ch==term) return;      // all is fine
        error(message);
    }
}
```

This reduces the input loop to

```
for (My_type var; ist>>var; ) {          // read until end of file
    // maybe check that var is valid

    // ... do something with var ...
}
end_of_loop(ist,'|',"bad termination of file");    // test if we can continue

// carry on: we found end of file or a terminator
```

The `end_of_loop()` does nothing unless the stream is in the `fail()` state. We consider that simple enough and general enough for many purposes.

10.11 Reading a structured file

Let's try to use this "standard loop" for a concrete example. As usual, we'll use the example to illustrate widely applicable design and programming techniques. Assume that you have a file of temperature readings that has been structured like this:

- A file holds years (of months of readings).
 - A year starts with `{ year` followed by an integer giving the year, such as `1900`, and ends with `}`.

- A year holds months (of days of readings).
 - A month starts with { **month** followed by a three-letter month name, such as **jan**, and ends with }.
- A reading holds a time and a temperature.
 - A reading starts with a (followed by day of the month, hour of the day, and temperature and ends with a).

For example:

```
{ year 1990 }
{year 1991 { month jun }}
{ year 1992 { month jan ( 1 0 61.5 ) } {month feb (1 1 64) (2 2 65.2) } }
{year 2000
 { month feb (1 1 68 ) (2 3 66.66 ) ( 1 0 67.2)}
 {month dec (15 15 -9.2 ) (15 14 -8.8) (14 0 -2) }
 }
```

This format is somewhat peculiar. File formats often are. There is a move toward more regular and hierarchically structured files (such as HTML and XML files) in the industry, but the reality is still that we can rarely control the input format offered by the files we need to read. The files are the way they are, and we just have to read them. If a format is too awful or files contain too many errors, we can write a format conversion program to produce a format that suits our main program better. On the other hand, we can typically choose the in-memory representation of data to suit our needs, and we can often pick output formats to suit needs and tastes.

So, let's assume that we have been given the temperature reading format above and have to live with it. Fortunately, it has self-identifying components, such as years and months (a bit like HTML or XML). On the other hand, the format of individual readings is somewhat unhelpful. For example, there is no information that could help us if someone flipped a day-of-the-month value with an hour of day or if someone produced a file with temperatures in Celsius and the program expected them in Fahrenheit or vice versa. We just have to cope.

10.11.1 In-memory representation

How should we represent this data in memory? The obvious first choice is three classes, **Year**, **Month**, and **Reading**, to exactly match the input. **Year** and **Month** are obviously useful when manipulating the data; we want to compare temperatures of different years, calculate monthly averages, compare different months of a year, compare the same month of different years, match up temperature readings with

sunshine records and humidity readings, etc. Basically, **Year** and **Month** match the way we think about temperatures and weather in general: **Month** holds a month's worth of information and **Year** holds a year's worth of information. But what about **Reading**? That's a low-level notion matching some piece of hardware (a sensor). The data of a **Reading** (day of month, hour of day, temperature) is "odd" and makes sense only within a **Month**. It is also unstructured: we have no promise that readings come in day-of-the-month or hour-of-the-day order. Basically, whenever we want to do anything of interest with the readings we have to sort them.

For representing the temperature data in memory, we make these assumptions:

- If we have any readings for a month, then we tend to have lots of readings for that month.
- If we have any readings for a day, then we tend to have lots of readings for that day.

When that's the case, it makes sense to represent a **Year** as a **vector** of 12 **Months**, a **Month** as a **vector** of about 30 **Days**, and a **Day** as 24 temperatures (one per hour). That's simple and easy to manipulate for a wide variety of uses. So, **Day**, **Month**, and **Year** are simple data structures, each with a constructor. Since we plan to create **Months** and **Days** as part of a **Year** before we know what temperature readings we have, we need to have a notion of "not a reading" for an hour of a day for which we haven't (yet) read data.

```
const int not_a_reading = -7777; // less than absolute zero
```

Similarly, we noticed that we often had a month without data, so we introduced the notion "not a month" to represent that directly, rather than having to search through all the days to be sure that no data was lurking somewhere:

```
const int not_a_month = -1;
```

The three key classes then become

```
struct Day {
    vector<double> hour {vector<double>(24, not_a_reading);
}
```

That is, a **Day** has 24 hours, each initialized to **not_a_reading**.

```
struct Month {                                // a month of temperature readings
    int month {not_a_month};                // [0:11] January is 0
    vector<Day> day {32};                  // [1:31] one vector of readings per day
}
```

We “waste” `day[0]` to keep the code simple.

```
struct Year {           // a year of temperature readings, organized by month
    int year;          // positive == A.D.
    vector<Month> month {12}; // [0:11] January is 0
};
```

Each class is basically a simple `vector` of “parts,” and `Month` and `Year` have an identifying member `month` and `year`, respectively.

There are several “magic constants” here (for example, `24`, `32`, and `12`). We try to avoid such literal constants in code. These are pretty fundamental (the number of months in a year rarely changes) and will not be used in the rest of the code. However, we left them in the code primarily so that we could remind you of the problem with “magic constants”; symbolic constants are almost always preferable (§7.6.1). Using `32` for the number of days in a month definitely requires explanation; `32` is obviously “magic” here.

Why didn’t we write

```
struct Day {
    vector<double> hour {24,not_a_reading};
};
```

That would have been simpler, but unfortunately, we would have gotten a `vector` of two elements (`24` and `-1`). When we want to specify the number of elements for a `vector` for which an integer can be converted to the element type, we unfortunately have to use the `()` initializer syntax (§18.2).

10.11.2 Reading structured values

The `Reading` class will be used only for reading input and is even simpler:

```
struct Reading {
    int day;
    int hour;
    double temperature;
};

istream& operator>>(istream& is, Reading& r)
// read a temperature reading from is into r
// format: ( 3 4 9.7 )
// check format, but don't bother with data validity
{
```

```

char ch1;
if (is>>ch1 && ch1!='{' {           // could it be a Reading?
        is.unget();
        is.clear(ios_base::failbit);
        return is;
}

char ch2;
int d;
int h;
double t;
is >> d >> h >> t >> ch2;
if (!is || ch2!='}') error("bad reading"); // messed-up reading
r.day = d;
r.hour = h;
r.temperature = t;
return is;
}

```

Basically, we check if the format begins plausibly, and if it doesn't we set the file state to **fail()** and return. This allows us to try to read the information in some other way. On the other hand, if we find the format wrong after having read some data so that there is no real chance of recovering, we bail out with **error()**.

The **Month** input operation is much the same, except that it has to read an arbitrary number of **Readings** rather than a fixed set of values (as **Reading**'s **>>** did):

```

istream& operator>>(istream& is, Month& m)
// read a month from is into m
// format: { month feb . . . }
{
    char ch = 0;
    if (is >> ch && ch!='{' {
            is.unget();
            is.clear(ios_base::failbit);      // we failed to read a Month
            return is;
    }

    string month_marker;
    string mm;
    is >> month_marker >> mm;
    if (!is || month_marker!="month") error("bad start of month");
    m.month = month_to_int(mm);

```

```

int duplicates = 0;
int invalids = 0;
for (Reading r; is >> r; ) {
    if (is_valid(r)) {
        if (m.day[r.day].hour[r.hour] != not_a_reading)
            ++duplicates;
        m.day[r.day].hour[r.hour] = r.temperature;
    }
    else
        ++invalids;
}
if (invalids) error("invalid readings in month",invalids);
if (duplicates) error("duplicate readings in month", duplicates);
end_of_loop(is,!","bad end of month");
return is;
}

```

We'll get back to `month_to_int()` later; it converts the symbolic notation for a month, such as `jun`, to a number in the `[0:11]` range. Note the use of `end_of_loop()` from §10.10 to check for the terminator. We keep count of invalid and duplicate `Readings`; someone might be interested.

`Month`'s `>>` does a quick check that a `Reading` is plausible before storing it:

```

constexpr int implausible_min = -200;
constexpr int implausible_max = 200;

bool is_valid(const Reading& r)
// a rough test
{
    if (r.day<1 || 31<r.day) return false;
    if (r.hour<0 || 23<r.hour) return false;
    if (r.temperature<implausible_min|| implausible_max<r.temperature)
        return false;
    return true;
}

```

Finally, we can read `Years`. `Year`'s `>>` is similar to `Month`'s `>>`:

```

istream& operator>>(istream& is, Year& y)
// read a year from is into y
// format: { year 1972 . . . }
{

```

```

char ch;
is >> ch;
if (ch != '{') {
    is.unget();
    is.clear(ios::failbit);
    return is;
}

string year_marker;
int yy;
is >> year_marker >> yy;
if (!is || year_marker != "year") error("bad start of year");
y.year = yy;

while(true) {
    Month m;           // get a clean m each time around
    if (!(is >> m)) break;
    y.month[m.month] = m;
}

end_of_loop(is, '}', "bad end of year");
return is;
}

```

We would have preferred “boringly similar” to just “similar,” but there is a significant difference. Have a look at the read loop. Did you expect something like the following?

```

for (Month m; is >> m; )
    y.month[m.month] = m;

```

You probably should have, because that’s the way we have written all the read loops so far. That’s actually what we first wrote, and it’s wrong. The problem is that **operator>>(istream& is, Month& m)** doesn’t assign a brand-new value to **m**; it simply adds data from **Readings** to **m**. Thus, the repeated **is>>m** would have kept adding to our one and only **m**. Oops! Each new month would have gotten all the readings from all previous months of that year. We need a brand-new, clean **Month** to read into each time we do **is>>m**. The easiest way to do that was to put the definition of **m** inside the loop so that it would be initialized each time around. The alternatives would have been for **operator>>(istream&**

is, Month& m) to assign an empty month to **m** before reading into it, or for the loop to do that:

```
for (Month m; is >> m; ) {
    y.month[m.month] = m;
    m = Month{}; // "reinitialize" m
}
```

Let's try to use it:

```
// open an input file:
cout << "Please enter input file name\n";
string iname;
cin >> iname;
ifstream ifs {iname};
if (!ifs) error("can't open input file", iname);

ifs.exceptions(ifs.exceptions()|ios_base::badbit); // throw for bad()

// open an output file:
cout << "Please enter output file name\n";
string oname;
cin >> oname;
ofstream ofs {oname};
if (!ofs) error("can't open output file", oname);

// read an arbitrary number of years:
vector<Year> ys;
while(true) {
    Year y; // get a freshly initialized Year each time around
    if (!(ifs >> y)) break;
    ys.push_back(y);
}
cout << "read " << ys.size() << " years of readings\n";

for (Year& y : ys) print_year(ofs, y);
```

We leave **print_year()** as an exercise.

10.11.3 Changing representations

To get **Month**'s **>>** to work, we need to provide a way of reading symbolic representations of the month. For symmetry, we'll provide a matching write using a symbolic representation. The tedious way would be to write an **if**-statement convert:

```

if (s=="jan")
    m = 1;
else if (s=="feb")
    m = 2;
...

```

This is not just tedious; it also builds the names of the months into the code. It would be better to have those in a table somewhere so that the main program could stay unchanged even if we had to change the symbolic representation. We decided to represent the input representation as a `vector<string>` plus an initialization function and a lookup function:

```

vector<string> month_input_tbl = {
    "jan", "feb", "mar", "apr", "may", "jun", "jul",
    "aug", "sep", "oct", "nov", "dec"
};

int month_to_int(string s)
// is s the name of a month? If so return its index [0:11] otherwise -1
{
    for (int i=0; i<12; ++i) if (month_input_tbl[i]==s) return i;
    return -1;
}

```

In case you wonder: the C++ standard library does provide a simpler way to do this. See §21.6.1 for a `map<string,int>`.

When we want to produce output, we have the opposite problem. We have an `int` representing a month and would like a symbolic representation to be printed. Our solution is fundamentally similar, but instead of using a table to go from `string` to `int`, we use one to go from `int` to `string`:

```

vector<string> month_print_tbl = {
    "January", "February", "March", "April", "May", "June", "July",
    "August", "September", "October", "November", "December"
};

string int_to_month(int i)
// months [0:11]
{
    if (i<0 || 12<=i) error("bad month index");
    return month_print_tbl[i];
}

```

So, did you actually read all of that code and the explanations? Or did your eyes glaze over and skip to the end? Remember that the easiest way of learning to write good code is to read a lot of code. Believe it or not, the techniques we used for this example are simple, but not trivial to discover without help. Reading data is fundamental. Writing loops correctly (initializing every variable used correctly) is fundamental. Converting between representations is fundamental. That is, you *will* learn to do such things. The only questions are whether you'll learn to do them well and whether you learn the basic techniques before losing too much sleep.



Drill

1. Start a program to work with points, discussed in §10.4. Begin by defining the data type **Point** that has two coordinate members **x** and **y**.
2. Using the code and discussion in §10.4, prompt the user to input seven (x,y) pairs. As the data is entered, store it in a **vector** of **Points** called **original_points**.
3. Print the data in **original_points** to see what it looks like.
4. Open an **ofstream** and output each point to a file named **mydata.txt**. On Windows, we suggest the **.txt** suffix to make it easier to look at the data with an ordinary text editor (such as WordPad).
5. Close the **ofstream** and then open an **ifstream** for **mydata.txt**. Read the data from **mydata.txt** and store it in a new **vector** called **processed_points**.
6. Print the data elements from both **vectors**.
7. Compare the two **vectors** and print **Something's wrong!** if the number of elements or the values of elements differ.

Review

1. When dealing with input and output, how is the variety of devices dealt with in most modern computers?
2. What, fundamentally, does an **istream** do?
3. What, fundamentally, does an **ostream** do?
4. What, fundamentally, is a file?
5. What is a file format?
6. Name four different types of devices that can require I/O for a program.
7. What are the four steps for reading a file?
8. What are the four steps for writing a file?
9. Name and define the four stream states.

10. Discuss how the following input problems can be resolved:
 - a. The user typing an out-of-range value
 - b. Getting no value (end of file)
 - c. The user typing something of the wrong type
11. In what way is input usually harder than output?
12. In what way is output usually harder than input?
13. Why do we (often) want to separate input and output from computation?
14. What are the two most common uses of the **istream** member function **clear()**?
15. What are the usual function declarations for `<<` and `>>` for a user-defined type **X**?

Terms

bad()	good()	ostream
buffer	ifstream	output device
clear()	input device	output operator
close()	input operator	stream state
device driver	iostream	structured file
eof()	istream	terminator
fail()	ofstream	unget()
file	open()	

Exercises

1. Write a program that produces the sum of all the numbers in a file of whitespace-separated integers.
2. Write a program that creates a file of data in the form of the temperature **Reading** type defined in §10.5. For testing, fill the file with at least 50 “temperature readings.” Call this program **store_temps.cpp** and the file it creates **raw_temps.txt**.
3. Write a program that reads the data from **raw_temps.txt** created in exercise 2 into a vector and then calculates the mean and median temperatures in your data set. Call this program **temp_stats.cpp**.
4. Modify the **store_temps.cpp** program from exercise 2 to include a temperature suffix **c** for Celsius or **f** for Fahrenheit temperatures. Then modify the **temp_stats.cpp** program to test each temperature, converting the Celsius readings to Fahrenheit before putting them into the vector.
5. Write the function **print_year()** mentioned in §10.11.2.
6. Define a **Roman_int** class for holding Roman numerals (as **ints**) with a `<<` and `>>`. Provide **Roman_int** with an **as_int()** member that returns the

- `int` value, so that if `r` is a `Roman_int`, we can write `cout << "Roman"`
`<< r << " equals "` `<< r.as_int() << '\n';`
7. Make a version of the calculator from Chapter 7 that accepts Roman numerals rather than the usual Arabic ones, for example, `XXI + CIV == CXXV.`
 8. Write a program that accepts two file names and produces a new file that is the contents of the first file followed by the contents of the second; that is, the program concatenates the two files.
 9. Write a program that takes two files containing sorted whitespace-separated words and merges them, preserving order.
 10. Add a command `from x` to the calculator from Chapter 7 that makes it take input from a file `x`. Add a command `to y` to the calculator that makes it write its output (both standard output and error output) to file `y`. Write a collection of test cases based on ideas from §7.3 and use that to test the calculator. Discuss how you would use these commands for testing.
 11. Write a program that produces the sum of all the whitespace-separated integers in a text file. For example, `bears: 17 elephants 9 end` should output `26.`

Postscript

Much of computing involves moving lots of data from one place to another, for example, copying text from a file to a screen or moving music from a computer onto an MP3 player. Often, some transformation of the data is needed on the way. The iostream library is a way of handling many such tasks where the data can be seen as a sequence (a stream) of values. Input and output can be a surprisingly large part of common programming tasks. This is partly because we (or our programs) need a lot of data and partly because the point where data enters a system is a place where lots of errors can happen. So, we must try to keep our I/O simple and try to minimize the chances that bad data “slips through” into our system.



11

Customizing Input and Output

“Keep it simple:
as simple as possible,
but no simpler.”

—Albert Einstein

In this chapter, we concentrate on how to adapt the general iostream framework presented in Chapter 10 to specific needs and tastes. This involves a lot of messy details dictated by human sensibilities to what they read and also practical constraints on the uses of files. The final example shows the design of an input stream for which you can specify the set of separators.

11.1 Regularity and irregularity**11.2 Output formatting****11.2.1 Integer output****11.2.2 Integer input****11.2.3 Floating-point output****11.2.4 Precision****11.2.5 Fields****11.3 File opening and positioning****11.3.1 File open modes****11.3.2 Binary files****11.3.3 Positioning in files****11.4 String streams****11.5 Line-oriented input****11.6 Character classification****11.7 Using nonstandard separators****11.8 And there is so much more**

11.1 Regularity and irregularity

The `iostream` library – the input/output part of the ISO C++ standard library – provides a unified and extensible framework for input and output of text. By “text” we mean just about anything that can be represented as a sequence of characters. Thus, when we talk about input and output we can consider the integer **1234** as text because we can write it using the four characters **1**, **2**, **3**, and **4**.

So far, we have treated all input sources as equivalent. Sometimes, that’s not enough. For example, files differ from other input sources (such as communications connections) in that we can address individual bytes. Similarly, we worked on the assumption that the type of an object completely determined the layout of its input and output. That’s not quite right and wouldn’t be sufficient. For example, we often want to specify the number of digits used to represent a floating-point number on output (its precision). This chapter presents a number of ways in which we can tailor input and output to our needs.

As programmers, we prefer regularity; treating all in-memory objects uniformly, treating all input sources equivalently, and imposing a single standard on the way to represent objects entering and exiting the system give the cleanest, simplest, most maintainable, and often the most efficient code. However, our programs exist to serve humans, and humans have strong preferences. Thus, as programmers we must strive for a balance between program complexity and accommodation of users’ personal tastes.

11.2 Output formatting

People care a lot about apparently minor details of the output they have to read. For example, to a physicist **1.25** (rounded to two digits after the dot) can be very

different from **1.24670477**, and to an accountant (**1.25**) can be legally different from (**1.2467**) and totally different from **1.25** (in financial documents, parentheses are sometimes used to indicate losses, that is, negative values). As programmers, we aim at making our output as clear and as close as possible to the expectations of the “consumers” of our program. Output streams (**ostreams**) provide a variety of ways for formatting the output of built-in types. For user-defined types, it is up to the programmer to define suitable `<<` operations.

There seem to be an infinite number of details, refinements, and options for output and quite a few for input. Examples are the character used for the decimal point (usually dot or comma), the way to output monetary values, a way to represent true as the word **true** (or **vrai** or **sandt**) rather than the number **1** when output, ways to deal with non-ASCII character sets (such as Unicode), and a way to limit the number of characters read into a string. These facilities tend to be uninteresting until you need them, so we’ll leave their description to manuals and specialized works such as Langer, *Standard C++ IOStreams and Locales*; Chapters 38 and 39 of *The C++ Programming Language* by Stroustrup; and §22 and §27 of the ISO C++ standard. Here we’ll present the most frequently useful features and a few general concepts.

11.2.1 Integer output

Integer values can be output as octal (the base-8 number system), decimal (our usual base-10 number system), and hexadecimal (the base-16 number system). If you don’t know about these systems, read §A.2.1.1 before proceeding here. Most output uses decimal. Hexadecimal is popular for outputting hardware-related information. The reason is that a hexadecimal digit exactly represents a 4-bit value. Thus, two hexadecimal digits can be used to present the value of an 8-bit byte, four hexadecimal digits give the value of 2 bytes (that’s often a half word), and eight hexadecimal digits can present the value of 4 bytes (that’s often the size of a word or a register). When C++’s ancestor C was first designed (in the 1970s), octal was popular for representing bit patterns, but now it’s rarely used.

We can specify the output (decimal) value **1234** to be decimal, hexadecimal (often called “hex”), and octal:

```
cout << 1234 << "\t(decimal)\n"
    << hex << 1234 << "\t(hexadecimal)\n"
    << oct << 1234 << "\t(octal)\n";
```

The `\t` character is “tab” (short for “tabulation character”). This prints

```
1234 (decimal)
4d2 (hexadecimal)
2322 (octal)
```

The notations `<< hex` and `<< oct` do not output values. Instead, `<< hex` informs the stream that any further integer values should be displayed in hexadecimal and `<< oct` informs the stream that any further integer values should be displayed in octal. For example:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n'; // the octal base is still in effect
```

This produces

```
1234 4d2 2322
2322 // integers will continue to show as octal until changed
```

Note that the last output is octal; that is, `oct`, `hex`, and `dec` (for decimal) persist (“stick,” “are sticky”) – they apply to every integer value output until we tell the stream otherwise. Terms such as `hex` and `oct` that are used to change the behavior of a stream are called *manipulators*.

TRY THIS



Output your birth year in decimal, hexadecimal, and octal form. Label each value. Line up your output in columns using the tab character. Now output your age.

Seeing values of a base different from 10 can often be confusing. For example, unless we tell you otherwise, you’ll assume that **11** represents the (decimal) number 11, rather than 9 (**11** in octal) or 17 (**11** in hexadecimal). To alleviate such problems, we can ask the `ostream` to show the base of each integer printed. For example:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << showbase << dec;      // show bases
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

This prints

```
1234 4d2 2322
1234 0x4d2 02322
```

So, decimal numbers have no prefix, octal numbers have the prefix **0**, and hexadecimal values have the prefix **0x** (or **0X**). This is exactly the notation for integer literals in C++ source code. For example:

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

In decimal form, this will print

1234 1234 1234

As you might have noticed, **showbase** persists, just like **oct** and **hex**. The manipulator **noshowbase** reverses the action of **showbase**, reverting to the default, which shows each number without its base.

In summary, the integer output manipulators are:

Integer output manipulations	
oct	use base-8 (octal) notation
dec	use base-10 (decimal) notation
hex	use base-16 (hexadecimal) notation
showbase	prefix 0 for octal and 0x for hexadecimal
noshowbase	don't use prefixes

11.2.2 Integer input

By default, **>>** assumes that numbers use the decimal notation, but you can tell it to read hexadecimal or octal integers:

```
int a;
int b;
int c;
int d;
cin >> a >> hex >> b >> oct >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

If you type in

1234 4d2 2322 2322

this will print

1234 1234 1234 1234

Note that this implies that **oct**, **dec**, and **hex** “stick” for input, just as they do for output.

TRY THIS

Complete the code fragment above to make it into a program. Try the suggested input; then type in

1234 1234 1234 1234

Explain the results. Try other inputs to see what happens.

You can get `>>` to accept and correctly interpret the `0` and `0x` prefixes. To do that, you “unset” all the defaults. For example:

```
cin.unsetf(ios::dec); // don't assume decimal (so that 0x can mean hex)
cin.unsetf(ios::oct); // don't assume octal (so that 12 can mean twelve)
cin.unsetf(ios::hex); // don't assume hexadecimal (so that 12 can mean twelve)
```

The stream member function `unsetf()` clears the flag (or flags) given as argument. Now, if you write

```
cin >>a >> b >> c >> d;
```

and enter

1234 0x4d2 02322 02322

you get

1234 1234 1234 1234

11.2.3 Floating-point output

If you deal directly with hardware, you’ll need hexadecimal (or possibly octal) notation. Similarly, if you deal with scientific computation, you must deal with the formatting of floating-point values. They are handled using `iostream` manipulators in a manner very similar to that of integer values. For example:

```
cout << 1234.56789 << "\t\t(defaultfloat)\n"      // \t\t to line up columns
<< fixed << 1234.56789 << "\t(fixed)\n"
<< scientific << 1234.56789 << "\t(scientific)\n";
```

This prints

1234.57	(general)
1234.567890	(fixed)
1.234568e+003	(scientific)

The manipulators **fixed**, **scientific**, and **defaultfloat** are used to select floating-point formats; **defaultfloat** is the default format (also known as the *general format*). Now, we can write

```

cout << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n';           // floating format "sticks"
cout << defaultfloat << 1234.56789 << '\t' // the default format for
                                                // floating-point output
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';

```

This prints

1234.57	1234.567890	1.234568e+003
1.234568e+003		// scientific manipulator "sticks"
1234.57	1234.567890	1.234568e+003

In summary, the basic floating-point output-formatting manipulators are:

Floating-point formats

fixed	use fixed-point notation
scientific	use mantissa and exponent notation; the mantissa is always in the [1:10) range; that is, there is a single nonzero digit before the decimal point
defaultfloat	choose fixed or scientific to give the numerically most accurate representation, within the precision of defaultfloat

11.2.4 Precision

By default, a floating-point value is printed using six total digits using the **defaultfloat** format. The most appropriate format is chosen and the number is rounded to give the best approximation that can be printed using only six digits (the default precision for the **defaultfloat** format). For example:

1234.567 prints as **1234.57**

1.2345678 prints as **1.23457**



The rounding rule is the usual 4/5 rule: 0 to 4 round down (toward zero) and 5 to 9 round up (away from zero). Note that floating-point formatting applies only to floating-point numbers, so

1234567 prints as **1234567** (because it's an integer)

1234567.0 prints as **1.23457e+006**

In the latter case, the **ostream** determines that **1234567.0** cannot be printed using the **fixed** format using only six digits and switches to **scientific** format to preserve the most accurate representation. Basically the **defaultfloat** format chooses between **scientific** and **fixed** formats to present the user with the most accurate representation of a floating-point value within the precision of the **general** format, which defaults to six total digits.

TRY THIS



Write some code to print the number **1234567.89** three times, first using **defaultfloat**, then **fixed**, then **scientific**. Which output form presents the user with the most accurate representation? Explain why.

A programmer can set the precision using the manipulator **setprecision()**. For example:

```
cout << 1234.56789 << '\t'  
    << fixed << 1234.56789 << '\t'  
    << scientific << 1234.56789 << '\n';  
cout << defaultfloat << setprecision(5)  
    << 1234.56789 << '\t'  
    << fixed << 1234.56789 << '\t'  
    << scientific << 1234.56789 << '\n';  
cout << defaultfloat << setprecision(8)  
    << 1234.56789 << '\t'  
    << fixed << 1234.56789 << '\t'  
    << scientific << 1234.56789 << '\n';
```

This prints (note the rounding)

```
1234.57      1234.567890      1.234568e+003  
1234.6 1234.56789      1.23457e+003  
1234.5679 1234.56789000  1.23456789e+003
```

The precision is defined as:

Floating-point precision

defaultfloat	precision is the total number of digits
scientific	precision is the number of digits after the decimal point
fixed	precision is the number of digits after the decimal point

Use the default (**defaultfloat**) format with precision 6) unless there is a reason not to. The usual reason not to is “Because we need greater accuracy of the output.”

11.2.5 Fields

Using **scientific** and **fixed** formats, a programmer can control exactly how much space a value takes up on output. That’s clearly useful for printing tables, etc. The equivalent mechanism for integer values is called *fields*. You can specify exactly how many character positions an integer value or string value will occupy using the “set field width” manipulator **setw()**. For example:

```
cout << 123456           // no field used
    << '|' << setw(4) << 123456 << '|'
    << setw(8) << 123456 << '|'      // set field width to 8
    << 123456 << "\n";            // field sizes don't stick
```

This prints

```
123456|123456| 123456|123456|
```

Note first the two spaces before the third occurrence of **123456**. That’s what we would expect for a six-digit number in an eight-character field. However, **123456** did not get truncated to fit into a four-character field. Why not? **|1234|** or **|3456|** might be considered plausible outputs for the four-character field. However, that would have completely changed the value printed without any warning to the poor reader that something had gone wrong. The **ostream** doesn’t do that; instead it breaks the output format. Bad formatting is almost always preferable to “bad output data.” In the most common uses of fields (such as printing out a table), the “overflow” is visually very noticeable, so that it can be corrected.

Fields can also be used for floating-point numbers and strings. For example:

```
cout << 12345 << '|' << setw(4) << 12345 << '|'
    << setw(8) << 12345 << '|' << 12345 << "\n";
cout << 1234.5 << '|' << setw(4) << 1234.5 << '|'
    << setw(8) << 1234.5 << '|' << 1234.5 << "\n";
```

```
cout << "asdfg" << '|' << setw(4) << "asdfg" << '|'
<< setw(8) << "asdfg" << '|' << "asdfg" << "\n";
```

This prints

```
12345|12345| 12345|12345|
1234.5|1234.5| 1234.5|1234.5|
asdfg|asdfg|  asdfg|asdfg|
```

Note that the field width “doesn’t stick.” In all three cases, the first and the last values are printed in the default “as many characters as it takes” format. In other words, unless you set the field width immediately before an output operation, the notion of “field” is not used.

TRY THIS



Make a simple table including the last name, first name, telephone number, and email address for yourself and at least five of your friends. Experiment with different field widths until you are satisfied that the table is well presented.

11.3 File opening and positioning



As seen from C++, a file is an abstraction of what the operating system provides. As described in §10.3, a file is simply a sequence of bytes numbered from 0 upward:



The question is how we access those bytes. Using **iostreams**, this is largely determined when we open a file and associate a stream with it. The properties of a stream determine what operations we can perform after opening the file, and their meaning. The simplest example of this is that if we open an **istream** for a file, we can read from the file, whereas if we open a file with an **ostream**, we can write to it.

11.3.1 File open modes

You can open a file in one of several modes. By default, an **ifstream** opens its file for reading and an **ofstream** opens its file for writing. That takes care of most common needs. However, you can choose between several alternatives:

File stream open modes	
ios_base::app	append (i.e., add to the end of the file)
ios_base::ate	“at end” (open and seek to end)
ios_base::binary	binary mode — beware of system-specific behavior
ios_base::in	for reading
ios_base::out	for writing
ios_base::trunc	truncate file to 0 length

A file mode is optionally specified after the name of the file. For example:

```
ofstream of1 {name1};           // defaults to ios_base::out
ifstream if1 {name2};           // defaults to ios_base::in

ofstream ofs {name, ios_base::app};   // ofstreams by default include
                                         // io_base::out
fstream fs {"myfile", ios_base::in|ios_base::out};    // both in and out
```

The `|` in that last example is the “bitwise or” operator (§A.5.5) that can be used to combine modes as shown. The `app` option is popular for writing log files where you always add to the end.

In each case, the exact effect of opening a file may depend on the operating system, and if an operating system cannot honor a request to open a file in a certain way, the result will be a stream that is not in the `good()` state:

```
if (!fs)      // oops: we couldn't open that file that way
```

The most common reason for a failure to open a file for reading is that the file doesn’t exist (at least not with the name we used):

```
ifstream ifs {"redungs";
if (!ifs)      // error: can't open "readings" for reading
```

In this case, we guess that a spelling error might be the problem.

Note that typically, an operating system will create a new file if you try to open a nonexistent file for output, but (fortunately) not if you try to open a nonexistent file for input:

```
ofstream ofs {"no-such-file";          // create new file called no-such-file
ifstream ifs {"no-file-of-this-name";    // error: ifs will not be good()
```

Try not to be clever with file open modes. Operating systems don't handle "unusual" mode consistently. When you can, stick to reading from files opened as **istreams** and writing to files opened as **ostreams**.

11.3.2 Binary files

In memory, we can represent the number 123 as an integer value or as a string value. For example:

```
int n = 123;  
string s = "123";
```

In the first case, **123** is stored as a (binary) number in an amount of memory that is the same as for all other **ints** (4 bytes, that is, 32 bits, on a PC). Had we chosen the value **12345** instead, the same 4 bytes would have been used. In the second case, **123** is stored as a string of three characters. Had we chosen the string value **"12345"** it would have used five characters (plus the fixed overhead for managing a **string**). We could illustrate this like this (using the ordinary decimal and character representation, rather than the binary representation actually used within the computer):

123 as characters:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	1	2	3	?	?	?	?	?
1	2	3	?	?	?	?	?		
12345 as characters:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>?</td><td>?</td><td>?</td></tr></table>	1	2	3	4	5	?	?	?
1	2	3	4	5	?	?	?		
123 as binary:	<table border="1"><tr><td>123</td><td></td></tr></table>	123							
123									
12345 as binary:	<table border="1"><tr><td>12345</td><td></td></tr></table>	12345							
12345									

When we use a character representation, we must use some character to represent the end of a number in memory, just as we do on paper: 123456 is one number and 123 456 are two numbers. On "paper," we use the space character to represent the end of the number. In memory, we could do the same:

123456 as characters:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td></td><td>?</td></tr></table>	1	2	3	4	5	6		?
1	2	3	4	5	6		?		
123 456 as characters:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td></td><td>4</td><td>5</td><td>6</td><td></td></tr></table>	1	2	3		4	5	6	
1	2	3		4	5	6			

The distinction between storing fixed-size binary representation (e.g., an **int**) and variable-size character string representation (e.g., a **string**) also occurs in files. By default, **iostreams** deal with character representations; that is, an **istream** reads a sequence of characters and turns it into an object of the desired type. An **ostream** takes an object of a specified type and transforms it into a sequence of characters which it writes out. However, it is possible to request **istream** and **ostream** to

simply copy bytes to and from files. That's called *binary I/O* and is requested by opening a file with the mode **ios_base::binary**. Here is an example that reads and writes binary files of integers. The key lines that specifically deal with "binary" are explained below:

```
int main()
{
    // open an istream for binary input from a file:
    cout << "Please enter input file name\n";
    string iname;
    cin >> iname;
    ifstream ifs {iname,ios_base::binary};      // note: stream mode
        // binary tells the stream not to try anything clever with the bytes
    if (!ifs) error("can't open input file ",iname);

    // open an ostream for binary output to a file:
    cout << "Please enter output file name\n";
    string oname;
    cin >> oname;
    ofstream ofs {oname,ios_base::binary};      // note: stream mode
        // binary tells the stream not to try anything clever with the bytes
    if (!ofs) error("can't open output file ",oname);

    vector<int> v;

    // read from binary file:
    for(int x; ifs.read(as_bytes(x),sizeof(int)); ) // note: reading bytes
        v.push_back(x);

    // . . . do something with v . . .

    // write to binary file:
    for(int x : v)
        ofs.write(as_bytes(x),sizeof(int));      // note: writing bytes
    return 0;
}
```

We open the files using **ios_base::binary** as the stream mode:

```
ifstream ifs {iname, ios_base::binary};

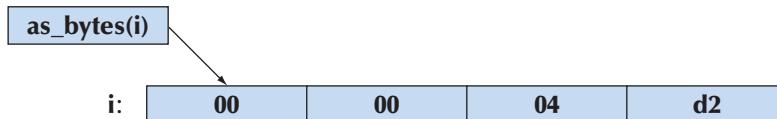
ofstream ofs {oname, ios_base::binary};
```

In both cases, we chose the trickier, but often more compact, binary representation. When we move from character-oriented I/O to binary I/O, we give up our usual `>>` and `<<` operators. Those operators specifically turn values into character sequences using the default conventions (e.g., the string `"asdf"` turns into the characters `a, s, d, f` and the integer `123` turns into the characters `1, 2, 3`). If we wanted that, we wouldn't need to say `binary` – the default would suffice. We use `binary` only if we (or someone else) thought that we somehow could do better than the default. We use `binary` to tell the stream not to try anything clever with the bytes.

What “cleverness” might we do to an `int`? The obvious is to store a 4-byte `int` in 4 bytes; that is, we can look at the representation of the `int` in memory (a sequence of 4 bytes) and transfer those bytes to the file. Later, we can read those bytes back the same way and reassemble the `int`:

```
ifs.read(as_bytes(i), sizeof(int))           // note: reading bytes
ofs.write(as_bytes(v[i]), sizeof(int))        // note: writing bytes
```

The `ostream write()` and the `istream read()` both take an address (supplied here by `as_bytes()`) and a number of bytes (characters) which we obtained by using the operator `sizeof`. That address should refer to the first byte of memory holding the value we want to read or write. For example, if we had an `int` with the value `1234`, we would get the 4 bytes (using hexadecimal notation) `00, 00, 04, d2`:



The `as_bytes()` function is needed to get the address of the first byte of an object's representation. It can – using language facilities yet to be explained (§17.8 and §19.3) – be defined like this:

```
template<class T>
char* as_bytes(T& i)           // treat a T as a sequence of bytes
{
    void* addr = &i;           // get the address of the first byte
                                // of memory used to store the object
    return static_cast<char*>(addr); // treat that memory as bytes
}
```

The (unsafe) type conversion using `static_cast` is necessary to get to the “raw bytes” of a variable. The notion of addresses will be explored in some detail in Chapters 17 and 18. Here, we just show how to treat any object in memory as a sequence of bytes for the use of `read()` and `write()`.

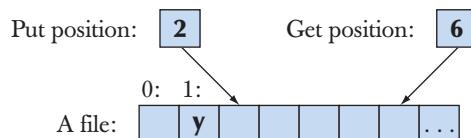
This binary I/O is messy, somewhat complicated, and error-prone. However, as programmers we don't always have the freedom to choose file formats, so occasionally we must use binary I/O simply because that's the format someone chose for the files we need to read or write. Alternatively, there may be a good logical reason for choosing a non-character representation. A typical example is an image or a sound file, for which there is no reasonable character representation: a photograph or a piece of music is basically just a bag of bits.

The character I/O provided by default by the iostream library is portable, human readable, and reasonably supported by the type system. Use it when you have a choice and don't mess with binary I/O unless you really have to.

11.3.3 Positioning in files

Whenever you can, just read and write files from the beginning to the end. That's the easiest and least error-prone way. Many times, when you feel that you have to make a change to a file, the better solution is to produce a new file containing the change.

However, if you must, you can use positioning to select a specific place in a file for reading or writing. Basically, every file that is open for reading has a "read/get position" and every file that is open for writing has a "write/put position":



This can be used like this:

```
fstream fs {name}; // open for input and output
if (!fs) error("can't open ", name);

fs.seekg(5); // move reading position (g for "get") to 5 (the 6th character)
char ch;
fs>>ch; // read and increment reading position
cout << "character[5] is " << ch << ' (' << int(ch) << ")\n";

fs.seekp(1); // move writing position (p for "put") to 1
fs<<'y'; // write and increment writing position
```

Note that **seekg()** and **seekp()** increment their respective positions, so the figure represents the state of the program *after* execution.

Please be careful: there is next to no run-time error checking when you use positioning. In particular, it is undefined what happens if you try to seek (using `seekg()` or `seekp()`) beyond the end of a file, and operating systems really do differ in what happens then.

11.4 String streams

You can use a `string` as the source of an `istream` or the target for an `ostream`. An `istream` that reads from a `string` is called an `istringstream` and an `ostream` that stores characters written to it in a `string` is called an `ostringstream`. For example, an `istringstream` is useful for extracting numeric values from a `string`:

```
double str_to_double(string s)
    // if possible, convert characters in s to floating-point value
{
    istringstream is {s};           // make a stream so that we can read from s
    double d;
    is >> d;
    if (!is) error("double format error: ",s);
    return d;
}

double d1 = str_to_double("12.4");           // testing
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("twelve point three"); // will call error()
```

If we try to read beyond the end of an `istringstream`'s string, the `istringstream` will go into `eof()` state. This means that we can use “the usual input loop” for an `istringstream`; an `istringstream` really is a kind of `istream`.

Conversely, an `ostringstream` can be useful for formatting output for a system that requires a simple `string` argument, such as a GUI system (see §16.5). For example:

```
void my_code(string label, Temperature temp)
{
    ...
    ostringstream os;           // stream for composing a message
    os << setw(8) << label << ":" 
        << fixed << setprecision(5) << temp.temp << temp.unit;
    someobject.display(Point(100,100), os.str().c_str());
    ...
}
```

The `str()` member function of `ostringstream` returns the `string` composed by output operations to an `ostringstream`. The `c_str()` is a member function of `string` that returns a C-style string as required by many system interfaces.

The `stringstreams` are generally used when we want to separate actual I/O from processing. For example, a `string` argument for `str_to_double()` will usually originate in a file (e.g., a web log) or from a keyboard. Similarly, the message we composed in `my_code()` will eventually end up written to an area of a screen. For example, in §11.7, we use a `stringstream` to filter undesirable characters out of our input. Thus, `stringstreams` can be seen as a mechanism for tailoring I/O to special needs and tastes.

A simple use of an `ostringstream` is to construct strings by concatenation. For example:

```
int seq_no = get_next_number();           // get the number of a log file
ostringstream name;
name << "myfile" << seq_no << ".log"; // e.g., myfile17.log
ofstream logfile{name.str()};            // e.g., open myfile17.log
```

Usually, we initialize an `istringstream` with a `string` and then read the characters from that `string` using input operations. Conversely, we typically initialize an `ostringstream` to the empty `string` and then fill it using output operations. There is a more direct way of accessing characters in a `stringstream` that is sometimes useful: `ss.str()` returns a copy of `ss`'s string, and `ss.str(s)` sets `ss`'s string to a copy of `s`. §11.7 shows an example where `ss.str(s)` is essential.

11.5 Line-oriented input

A `>>` operator reads into objects of a given type according to that type's standard format. For example, when reading into an `int`, `>>` will read until it encounters something that's not a digit, and when reading into a `string`, `>>` will read until it encounters whitespace. The standard library `istream` library also provides facilities for reading individual characters and whole lines. Consider:

```
string name;
cin >> name;                      // input: Dennis Ritchie
cout << name << '\n';             // output: Dennis
```

What if we wanted to read everything on that line at once and decide how to format it later? That could be done using the function `getline()`. For example:

```
string name;
getline(cin, name);                // input: Dennis Ritchie
cout << name << '\n';             // output: Dennis Ritchie
```

Now we have the whole line. Why would we want that? A good answer would be “Because we want to do something that can’t be done by `>>`.” Often, the answer is a poor one: “Because the user typed a whole line.” If that’s the best you can think of, stick to `>>`, because once you have the line entered, you usually have to parse it somehow. For example:

```
string first_name;
string second_name;
stringstream ss {name};
ss>>first_name;           // input Dennis
ss>>second_name;         // input Ritchie
```

Reading directly into `first_name` and `second_name` would have been simpler.

One common reason for wanting to read a whole line is that the definition of whitespace isn’t always appropriate. Sometimes, we want to consider a newline as different from other whitespace characters. For example, a text communication with a game might consider a line a sentence, rather than relying on conventional punctuation:

```
go left until you see a picture on the wall to your right
remove the picture and open the door behind it. take the bag from there
```

In that case, we’d first read a whole line and then extract individual words from that.

```
string command;
getline(cin,command);           // read the line

stringstream ss {command};
vector<string> words;
for (string s; ss>>s; )
    words.push_back(s); // extract the individual words
```

On the other hand, had we had a choice, we would most likely have preferred to rely on some proper punctuation rather than a line break.

11.6 Character classification



Usually, we read integers, floating-point numbers, words, etc. as defined by format conventions. However, we can – and sometimes must – go down a level of abstraction and read individual characters. That’s more work, but when we read individual characters, we have full control over what we are doing. Consider

tokenizing an expression (§7.8.2). For example, we want **1+4*x<=y/z*5** to be separated into the eleven tokens

```
1 + 4 * x <= y / z * 5
```

We could use **>>** to read the numbers, but trying to read the identifiers as strings would cause **x<=y** to be read as one string (since **<** and **=** are not whitespace characters) and **z*** to be read as one string (since ***** isn't a whitespace character either). Instead, we could write

```
for (char ch; cin.get(ch); ) {
    if (isspace(ch)) {      // if ch is whitespace
        // do nothing (i.e., skip whitespace)
    }
    if (isdigit(ch)) {
        // read a number
    }
    else if (isalpha(ch)) {
        // read an identifier
    }
    else {
        // deal with operators
    }
}
```

The **istream::get()** function reads a single character into its argument. It does not skip whitespace. Like **>>**, **get()** returns a reference to its **istream** so that we can test its state.

When we read individual characters, we usually want to classify them: Is this character a digit? Is this character uppercase? And so forth. There is a set of standard library functions for that:

Character classification	
isspace(c)	Is c whitespace (' ', '\t', '\n', etc.)?
isalpha(c)	Is c a letter ('a'.. 'z', 'A'.. 'Z') (note: not '_')?
isdigit(c)	Is c a decimal digit ('0'.. '9')?
isxdigit(c)	Is c a hexadecimal digit (decimal digit or 'a'.. 'f' or 'A'.. 'F')?
isupper(c)	Is c an uppercase letter?
islower(c)	Is c a lowercase letter?
isalnum(c)	Is c a letter or a decimal digit?
iscntrl(c)	Is c a control character (ASCII 0..31 and 127)?

Character classification (*continued*)

ispunct(c)	Is c not a letter, digit, whitespace, or invisible control character?
isprint(c)	Is c printable (ASCII '!'.. '~')?
isgraph(c)	Is isalpha(c) or isdigit(c) or ispunct(c) (note: not space)?

Note that the classifications can be combined using the “or” operator (`||`). For example, **isalnum(c)** means **isalpha(c)||isdigit(c)**; that is, “Is **c** either a letter or a digit?”

In addition, the standard library provides two useful functions for getting rid of case differences:

Character case

toupper(c)	c or c 's uppercase equivalent
tolower(c)	c or c 's lowercase equivalent

These are useful when you want to ignore case differences. For example, in input from a user **Right**, **right**, and **rigHT** most likely mean the same thing (**rigHT** most likely being the result of an unfortunate hit on the Caps Lock key). After applying **tolower()** to each character in each of those strings, we get **right** for each. We can do that for an arbitrary **string**:

```
void tolower(string& s)      // put s into lower case
{
    for (char& x : s) x = tolower(x);
}
```

We use pass-by-reference (§8.5.5) to actually change the **string**. Had we wanted to keep the old string we could have written a function to make a lowercase copy.

Prefer **tolower()** to **toupper()** because that works better for text in some natural languages, such as German, where not every lowercase character has an uppercase equivalent.

11.7 Using nonstandard separators

This section provides a semi-realistic example of the use of **iostreams** to solve a real problem. When we read strings, words are by default separated by whitespace. Unfortunately, **istream** doesn't offer a facility for us to define what characters make up whitespace or in some other way directly change how **>>** reads a string. So, what do we do if we need another definition of whitespace? Consider the

example from §4.6.3 where we read in “words” and compared them. Those words were whitespace-separated, so if we read

As planned, the guests arrived; then,

We would get the “words”

```
As
planned,
the
guests
arrived;
then,
```

This is not what we’d find in a dictionary: **planned**, and **arrived**; are not words. They are words plus distracting and irrelevant punctuation characters. For most purposes we must treat punctuation just like whitespace. How might we get rid of such punctuation? We could read characters, remove the punctuation characters – or turn them into whitespace – and then read the “cleaned-up” input again:

```
string line;
getline(cin,line);           // read into line
for (char& ch : line)       // replace each punctuation character by a space
    switch(ch) {
        case ';': case '!': case ',': case '?': case '!':
            ch = ' ';
    }

stringstream ss(line);        // make an istream ss reading from line
vector<string> vs;
for (string word; ss>>word; ) // read words without punctuation characters
    vs.push_back(word);
```

Using that to read the line, we get the desired

```
As
planned
the
guests
arrived
then
```

Unfortunately, the code above is messy and rather special-purpose. What would we do if we had another definition of punctuation? Let's provide a more general and useful way of removing unwanted characters from an input stream. What would that be? What would we like our user code to look like? How about

```
ps.whitespace(";,:."); // treat semicolon, colon, comma, and dot as whitespace
for (string word; ps>>word; )
    vs.push_back(word);
```

How would we define a stream that would work like **ps**? The basic idea is to read words from an ordinary input stream and then treat the user-specified “whitespace” characters as whitespace; that is, we do not give “whitespace” characters to the user, we just use them to separate words. For example,

as.not

should be the two words

**as
not**

We can define a class to do that for us. It must get characters from an **istream** and have a **>>** operator that works just like **istream**'s except that we can tell it which characters it should consider to be whitespace. For simplicity, we will not provide a way of treating existing whitespace characters (space, newline, etc.) as non-whitespace; we'll just allow a user to specify additional “whitespace” characters. Nor will we provide a way to completely remove the designated characters from the stream; as before, we will just turn them into whitespace. Let's call that class **Punct_stream**:

```
class Punct_stream { // like an istream, but the user can add to
    // the set of whitespace characters
public:
    Punct_stream(istream& is)
        : source{is}, sensitive{true} {}

    void whitespace(const string& s) // make s the whitespace set
        { white = s; }
    void add_white(char c) { white += c; } // add to the whitespace set
    bool is_whitespace(char c); // is c in the whitespace set?
```

```

void case_sensitive(bool b) { sensitive = b; }
bool is_case_sensitive() { return sensitive; }

Punct_stream& operator>>(string& s);
operator bool();
private:
    istream& source;           // character source
    istringstream buffer;        // we let buffer do our formatting
    string white;              // characters considered "whitespace"
    bool sensitive;            // is the stream case-sensitive?
};


```

The basic idea is – just as in the example above – to read a line at a time from the **istream**, convert “whitespace” characters into spaces, and then use the **istringstream** to do formatting. In addition to dealing with user-defined whitespace, we have given **Punct_stream** a related facility: if we ask it to, using **case_sensitive()**, it can convert case-sensitive input into non-case-sensitive input. For example, if we ask, we can get a **Punct_stream** to read

Man bites dog!

as

```

man
bites
dog

```

Punct_stream’s constructor takes the **istream** to be used as a character source and gives it the local name **source**. The constructor also defaults the stream to the usual case-sensitive behavior. We can make a **Punct_stream** that reads from **cin** regarding semicolon, colon, and dot as whitespace, and that turns all characters into lower case:

```

Punct_stream ps {cin};           // ps reads from cin
ps.whitespace(";:.");             // semicolon, colon, and dot are also whitespace
ps.case_sensitive(false);          // not case-sensitive

```

Obviously, the most interesting operation is the input operator **>>**. It is also by far the most difficult to define. Our general strategy is to read a whole line from the **istream** into a string (called **line**). We then convert all of “our” whitespace characters to the space character (‘ ’). That done, we put the line into the **istringstream**

called **buffer**. Now we can use the usual whitespace-separating **>>** to read from **buffer**. The code looks a bit more complicated than this because we simply try reading from the **buffer** and try to fill it only when we find it empty:

```
Punct_stream& Punct_stream::operator>>(string& s)
{
    while (!(buffer>>s)) {           // try to read from buffer
        if (buffer.bad() || !source.good()) return *this;
        buffer.clear();

        string line;
        getline(source,line);          // get a line from source

        // do character replacement as needed:
        for (char& ch : line)
            if (is whitespace(ch))
                ch = ' ';               // to space
            else if (!sensitive)
                ch = tolower(ch);       // to lower case

        buffer.str(line);              // put string into stream
    }
    return *this;
}
```

Let's consider this bit by bit. Consider first the somewhat unusual

```
while (!(buffer>>s)) {
```

If there are characters in the **istringstream** called **buffer**, the read **buffer>>s** will work, and **s** will receive a “whitespace”-separated word; then there is nothing more to do. That will happen as long as there are characters in **buffer** for us to read. However, when **buffer>>s** fails – that is, if **!(buffer>>s)** – we must replenish **buffer** from **source**. Note that the **buffer>>s** read is in a loop; after we have tried to replenish **buffer**, we need to try another read, so we get

```
while (!(buffer>>s)) {           // try to read from buffer
    if (buffer.bad() || !source.good()) return *this;
    buffer.clear();

    // replenish buffer
}
```

If `buffer` is `bad()` or the source has a problem, we give up; otherwise, we clear `buffer` and try again. We need to clear `buffer` because we get into that “replenish loop” only if a read failed, typically because we hit `eof()` for `buffer`; that is, there were no more characters in `buffer` for us to read. Dealing with stream state is always messy and it is often the source of subtle errors that require tedious debugging. Fortunately the rest of the replenish loop is pretty straightforward:

```

string line;
getline(source,line);           // get a line from source

// do character replacement as needed:
for (char& ch : line)
    if (is_whitespace(ch))
        ch = ' ';                // to space
    else if (!sensitive)
        ch = tolower(ch);        // to lower case

buffer.str(line);             // put string into stream

```

We read a line into `line`. Then we look at each character of that line to see if we need to change it. The `is_whitespace()` function is a member of `Punct_stream`, which we'll define later. The `tolower()` function is a standard library function doing the obvious, such as turning `A` into `a` (see §11.6).

Once we have a properly processed `line`, we need to get it into our `istringstream`. That's what `buffer.str(line)` does; it can be read as “Set the `istringstream` `buffer`'s `string` to `line`.”

Note that we “forgot” to test the state of `source` after reading from it using `getline()`. We don't need to because we will eventually reach the `!source.good()` test at the top of the loop.

As ever, we return a reference to the stream itself, `*this`, as the result of `>>`; see §17.10.

Testing for whitespace is easy; we just compare a character to each character of the string that holds our whitespace set:

```

bool Punct_stream::is_whitespace(char c)
{
    for (char w : white)
        if (c==w) return true;
    return false;
}

```

Remember that we left the **istringstream** to deal with the usual whitespace characters (e.g., newline and space) in the usual way, so we don't need to do anything special about those.

This leaves one mysterious function:

```
Punct_stream::operator bool()
{
    return !(source.fail() || source.bad() && source.good());
}
```

The conventional use of an **istream** is to test the result of **>>**. For example:

```
while (ps>>s) { /* . . . */ }
```

That means that we need a way of looking at the result of **ps>>s** as a Boolean value. The result of **ps>>s** is a **Punct_stream**, so we need a way of implicitly turning a **Punct_stream** into a **bool**. That's what **Punct_stream**'s **operator bool()** does. A member function called **operator bool()** defines a conversion to **bool**. In particular, it returns **true** if the operation on the **Punct_stream** succeeded.

Now we can write our program:

```
int main()
// given text input, produce a sorted list of all words in that text
// ignore punctuation and case differences
// eliminate duplicates from the output
{
    Punct_stream ps {cin};
    ps.whitespace(";,.?!(){}<>/&$@#%^*|~"); // note \" means " in string
    ps.case_sensitive(false);

    cout << "please enter words\n";
    vector<string> vs;
    for (string word; ps>>word; )
        vs.push_back(word); // read words

    sort(vs.begin(),vs.end()); // sort in lexicographical order
    for (int i=0; i<vs.size(); ++i) // write dictionary
        if (i==0 || vs[i]!=vs[i-1]) cout << vs[i] << '\n';
}
```

This will produce a properly sorted list of words from input. The test

```
if (i==0 || vs[i]!=vs[i-1])
```

will suppress duplicates. Feed this program the input

There are only two kinds of languages: languages that people complain about, and languages that people don't use.

and it will output

```
about  
and  
are  
complain  
don't  
kind  
languages  
of  
only  
people  
that  
there  
two  
use
```

Why did we get **don't** and not **dont**? We left the single quote out of the **whitespace()** call.

Caution: **Punct_stream** behaves like an **istream** in many important and useful ways, but it isn't really an **istream**. For example, we can't ask for its state using **rdstate()**, **eof()** isn't defined, and we didn't bother providing a **>>** that reads integers. Importantly, we cannot pass a **Punct_stream** to a function expecting an **istream**. Could we define a **Punct_istream** that really is an **istream**? We could, but we don't yet have the programming experience, the design concepts, and the language facilities required to pull off that stunt (if you – much later – want to return to this problem, you have to look up stream buffers in an expert-level guide or manual).

Did you find **Punct_stream** easy to read? Did you find the explanations easy to follow? Do you think you could have written it yourself? If you were a genuine novice a few days ago, the honest answer is likely to be “No, no, no!” or even “NO, no! Nooo!! – Are you crazy?” We understand – and the answer to the last question/outburst is “No, at least we think not.” The purpose of the example is

- To show a somewhat realistic problem and solution
- To show what can be achieved with relatively modest means
- To provide an easy-to-use solution to an apparently easy problem
- To illustrate the distinction between the interface and the implementation



To become a programmer, you need to read code, and not just carefully polished solutions to educational problems. This is an example. In another few days or weeks, this will become easy for you to read, and you will be looking at ways to improve the solution.

One way to think of this example is as equivalent to a teacher having dropped some genuine English slang into an English-for-beginners course to give a bit of color and enliven the proceedings.

11.8 And there is so much more



The details of I/O seem infinite. They probably are, since they are limited only by human inventiveness and capriciousness. For example, we have not considered the complexity implied by natural languages. What is written as **12.35** in English will be conventionally represented as **12,35** in most other European languages. Naturally, the C++ standard library provides facilities for dealing with that and many other natural-language-specific aspects of I/O. How do you write Chinese characters? How do you compare strings written using Malayalam characters? There are answers, but they are far beyond the scope of this book. If you need to know, look in more specialized or advanced books (such as Langer, *Standard C++ IOStreams and Locales*, and Stroustrup, *The C++ Programming Language*) and in library and system documentation. Look for “locale”; that’s the term usually applied to facilities for dealing with natural language differences.

Another source of complexity is buffering: the standard library **iostreams** rely on a concept called **streambuf**. For advanced work – whether for performance or functionality – with **iostreams** these **streambufs** are unavoidable. If you feel the need to define your own **iostreams** or to tune **iostreams** to new data sources/sinks, see Chapter 38 of *The C++ Programming Language* by Stroustrup or your system documentation.

When using C++, you may also encounter the C standard **printf()/scanf()** family of I/O functions. If you do, look them up in §27.6, §B.10.2, or in the excellent C textbook by Kernighan and Ritchie (*The C Programming Language*) or one of the innumerable sources on the web. Each language has its own I/O facilities; they all vary, most are quirky, but most reflect (in various odd ways) the same fundamental concepts that we have presented in Chapters 10 and 11.

The standard library I/O facilities are summarized in Appendix B.

The related topic of graphical user interfaces (GUIs) is described in Chapters 12–16.



Drill

1. Start a program called **Test_output.cpp**. Declare an integer **birth_year** and assign it the year you were born.
2. Output your **birth_year** in decimal, hexadecimal, and octal form.
3. Label each value with the name of the base used.
4. Did you line up your output in columns using the tab character? If not, do it.
5. Now output your age.
6. Was there a problem? What happened? Fix your output to decimal.
7. Go back to 2 and cause your output to show the base for each output.
8. Try reading as octal, hexadecimal, etc.:

```
cin >> a >> oct >> b >> hex >> c >> d;  
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n' ;
```

Run this code with the input

```
1234 1234 1234 1234
```

Explain the results.

9. Write some code to print the number **1234567.89** three times, first using **defaultfloat**, then **fixed**, then **scientific** forms. Which output form presents the user with the most accurate representation? Explain why.
10. Make a simple table including last name, first name, telephone number, and email address for yourself and at least five of your friends. Experiment with different field widths until you are satisfied that the table is well presented.

Review

1. Why is I/O tricky for a programmer?
2. What does the notation **<< hex** do?
3. What are hexadecimal numbers used for in computer science? Why?
4. Name some of the options you may want to implement for formatting integer output.
5. What is a manipulator?
6. What is the prefix for decimal? For octal? For hexadecimal?
7. What is the default output format for floating-point values?
8. What is a field?
9. Explain what **setprecision()** and **setw()** do.
10. What is the purpose of file open modes?
11. Which of the following manipulators does not “stick”: **hex**, **scientific**, **setprecision()**, **showbase**, **setw**?

12. What is the difference between character I/O and binary I/O?
13. Give an example of when it would probably be beneficial to use a binary file instead of a text file.
14. Give two examples where a **stringstream** can be useful.
15. What is a file position?
16. What happens if you position a file position beyond the end of file?
17. When would you prefer line-oriented input to type-specific input?
18. What does **isalnum(c)** do?

Terms

binary	hexadecimal	octal
character classification	irregularity	output formatting
decimal	line-oriented input	regularity
defaultfloat	manipulator	scientific
file positioning	nonstandard separator	setprecision()
fixed	noshowbase	showbase

Exercises

1. Write a program that reads a text file and converts its input to all lower case, producing a new file.
2. Write a program that given a file name and a word outputs each line that contains that word together with the line number. Hint: **getline()**.
3. Write a program that removes all vowels from a file (“disemvowels”). For example, **Once upon a time!** becomes **nc pn tm!**. Surprisingly often, the result is still readable; try it on your friends.
4. Write a program called **multi_input.cpp** that prompts the user to enter several integers in any combination of octal, decimal, or hexadecimal, using the **0** and **0x** base suffixes; interprets the numbers correctly; and converts them to decimal form. Then your program should output the values in properly spaced columns like this:

```
0x43 hexadecimal converts to 67 decimal
0123 octal      converts to 83 decimal
65 decimal      converts to 65 decimal
```

5. Write a program that reads strings and for each string outputs the character classification of each character, as defined by the character classification functions presented in §11.6. Note that a character can have several classifications (e.g., **x** is both a letter and an alphanumeric).
6. Write a program that replaces punctuation with whitespace. Consider **.** (dot), **;** (semicolon), **,** (comma), **?** (question mark), **-** (dash), **'** (single

- quote) punctuation characters. Don't modify characters within a pair of double quotes (""). For example, “ - **don't use the as-if rule.**” becomes “ **don t use the as if rule** ”.
7. Modify the program from the previous exercise so that it replaces **don't** with **do not**, **can't** with **cannot**, etc.; leaves hyphens within words intact (so that we get “ **do not use the as-if rule** ”); and converts all characters to lower case.
 8. Use the program from the previous exercise to make a dictionary (as an alternative to the approach in §11.7). Run the result on a multi-page text file, look at the result, and see if you can improve the program to make a better dictionary.
 9. Split the binary I/O program from §11.3.2 into two: one program that converts an ordinary text file into binary and one program that reads binary and converts it to text. Test these programs by comparing a text file with what you get by converting it to binary and back.
 10. Write a function `vector<string> split(const string& s)` that returns a `vector` of whitespace-separated substrings from the argument `s`.
 11. Write a function `vector<string> split(const string& s, const string& w)` that returns a `vector` of whitespace-separated substrings from the argument `s`, where whitespace is defined as “ordinary whitespace” plus the characters in `w`.
 12. Reverse the order of characters in a text file. For example, **asdfghjkl** becomes **lkjhgfdsa**. Warning: There is no really good, portable, and efficient way of reading a file backward.
 13. Reverse the order of words (defined as whitespace-separated strings) in a file. For example, **Norwegian Blue parrot** becomes **parrot Blue Norwegian**. You are allowed to assume that all the strings from the file will fit into memory at once.
 14. Write a program that reads a text file and writes out how many characters of each character classification (§11.6) are in the file.
 15. Write a program that reads a file of whitespace-separated numbers and outputs a file of numbers using scientific format and precision 8 in four fields of 20 characters per line.
 16. Write a program to read a file of whitespace-separated numbers and output them in order (lowest value first), one value per line. Write a value only once, and if it occurs more than once write the count of its occurrences on its line. For example, **7 5 5 7 3 117 5** should give

3
5 3
7 2
117

Postscript



Input and output are messy because our human tastes and conventions have not followed simple-to-state rules and straightforward mathematical laws. As programmers, we are rarely in a position to dictate that our users depart from their preferences, and when we are, we should typically be less arrogant than to think that we can provide a simple alternative to conventions built up over time. Consequently, we must expect, accept, and adapt to a certain messiness of input and output while still trying to keep our programs as simple as possible – but no simpler.



A Display Model

“The world was black and white then.
[It] didn’t turn color
until sometime in the 1930s.”

—Calvin’s dad

This chapter presents a display model (the output part of GUI), giving examples of use and fundamental notions such as screen coordinates, lines, and color. **Line**, **Lines**, **Polygons**, **Axis**, and **Text** are examples of **Shapes**. A **Shape** is an object in memory that we can display and manipulate on a screen. The next two chapters will explore these classes further, with Chapter 13 focusing on their implementation and Chapter 14 on design issues.

- 12.1 Why graphics?**
- 12.2 A display model**
- 12.3 A first example**
- 12.4 Using a GUI library**
- 12.5 Coordinates**
- 12.6 Shapes**

- 12.7 Using Shape primitives**
 - 12.7.1 Graphics headers and main**
 - 12.7.2 An almost blank window**
 - 12.7.3 Axis**
 - 12.7.4 Graphing a function**
 - 12.7.5 Polygons**
 - 12.7.6 Rectangles**
 - 12.7.7 Fill**
 - 12.7.8 Text**
 - 12.7.9 Images**
 - 12.7.10 And much more**
- 12.8 Getting this to run**
 - 12.8.1 Source files**

12.1 Why graphics?

Why do we spend four chapters on graphics and one on GUIs (graphical user interfaces)? After all, this is a book about programming, not a graphics book. There is a huge number of interesting software topics that we don't discuss, and we can at best scratch the surface on the topic of graphics. So, "Why graphics?" Basically, graphics is a subject that allows us to explore several important areas of software design, programming, and programming language facilities:

- *Graphics are useful.* There is much more to programming than graphics and much more to software than code manipulated through a GUI. However, in many areas good graphics are either essential or very important. For example, we wouldn't dream of studying scientific computing, data analysis, or just about any quantitative subject without the ability to graph data. Chapter 15 gives simple (but general) facilities for graphing data.
- *Graphics are fun.* There are few areas of computing where the effect of a piece of code is as immediately obvious and – when finally free of bugs – as pleasing. We'd be tempted to play with graphics even if it wasn't useful!
- *Graphics provide lots of interesting code to read.* Part of learning to program is to read lots of code to get a feel for what good code is like. Similarly, the way to become a good writer of English involves reading a lot of books, articles, and quality newspapers. Because of the direct correspondence between what we see on the screen and what we write in our programs, simple graphics code is more readable than most kinds of code of similar complexity. This chapter will prove that you can read graphics code after a few minutes of introduction; Chapter 13 will demonstrate how you can write it after another couple of hours.

- *Graphics are a fertile source of design examples.* It is actually hard to design and implement a good graphics and GUI library. Graphics are a very rich source of concrete and practical examples of design decisions and design techniques. Some of the most useful techniques for designing classes, designing functions, separating software into layers (of abstraction), and constructing libraries can be illustrated with a relatively small amount of graphics and GUI code.
- *Graphics provide a good introduction to what is commonly called object-oriented programming and the language features that support it.* Despite rumors to the contrary, object-oriented programming wasn't invented to be able to do graphics (see Chapter 22), but it was soon applied to that, and graphics provide some of the most accessible examples of object-oriented designs.
- *Some of the key graphics concepts are nontrivial.* So they are worth teaching, rather than leaving it to your own initiative (and patience) to seek out information. If we did not show how graphics and GUI were done, you might consider them "magic," thus violating one of the fundamental aims of this book.

12.2 A display model

The iostream library is oriented toward reading and writing streams of characters as they might appear in a list of numeric values or a book. The only direct supports for the notion of graphical position are the newline and tab characters. You can embed notions of color and two-dimensional positions, etc. in a one-dimensional stream of characters. That's what layout (typesetting, "markup") languages such as Troff, TeX, Word, HTML, and XML (and their associated graphical packages) do. For example:

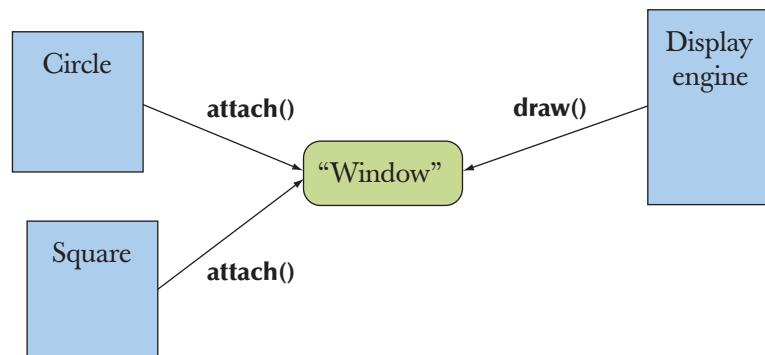
```
<hr>
<h2>
Organization
</h2>
This list is organized in three parts:
<ul>
    <li><b>Proposals</b>, numbered EPddd, . . .</li>
    <li><b>Issues</b>, numbered Elddd, . . .</li>
    <li><b>Suggestions</b>, numbered ESddd, . . .</li>
</ul>
<p>We try to . . .
<p>
```

This is a piece of HTML specifying a header (`<h2> . . . </h2>`), a list (` . . . `) with list items (` . . . `), and a paragraph (`<p>`). We left out most

of the actual text because it is irrelevant here. The point is that you can express layout notions in plain text, but the connection between the characters written and what appears on the screen is indirect, governed by a program that interprets those “markup” commands. Such techniques are fundamentally simple and immensely useful (just about everything you read has been produced using them), but they also have their limitations.

In this chapter and the next four, we present an alternative: a notion of graphics and of graphical user interfaces that is directly aimed at a computer screen. The fundamental concepts are inherently graphical (and two-dimensional, adapted to the rectangular area of a computer screen), such as coordinates, lines, rectangles, and circles. The aim from a programming point of view is a direct correspondence between the objects in memory and the images on the screen.

The basic model is as follows: We compose objects with basic objects provided by a graphics system, such as lines. We “attach” these graphics objects to a `window` object, representing our physical screen. A program that we can think of as the display itself, as “a display engine,” as “our graphics library,” as “the GUI library,” or even (humorously) as “the small gnome writing on the back of the screen,” then takes the objects we have attached to our window and draws them on the screen:



The “display engine” draws lines on the screen, places strings of text on the screen, colors areas of the screen, etc. For simplicity, we’ll use the phrase “our GUI library” or even “the system” for the display engine even though our GUI library does much more than just drawing the objects. In the same way that our code lets the GUI library do most of the work for us, the GUI library delegates much of its work to the operating system.

12.3 A first example

Our job is to define classes from which we can make objects that we want to see on the screen. For example, we might want to draw a graph as a series of connected lines. Here is a small program presenting a very simple version of that:

```
#include "Simple_window.h" // get access to our window library
#include "Graph.h"          // get access to our graphics library facilities

int main()
{
    using namespace Graph_lib; // our graphics facilities are in Graph_lib

    Point tl {100,100};           // to become top left corner of window

    Simple_window win {tl,600,400,"Canvas"}; // make a simple window

    Polygon poly;                // make a shape (a polygon)

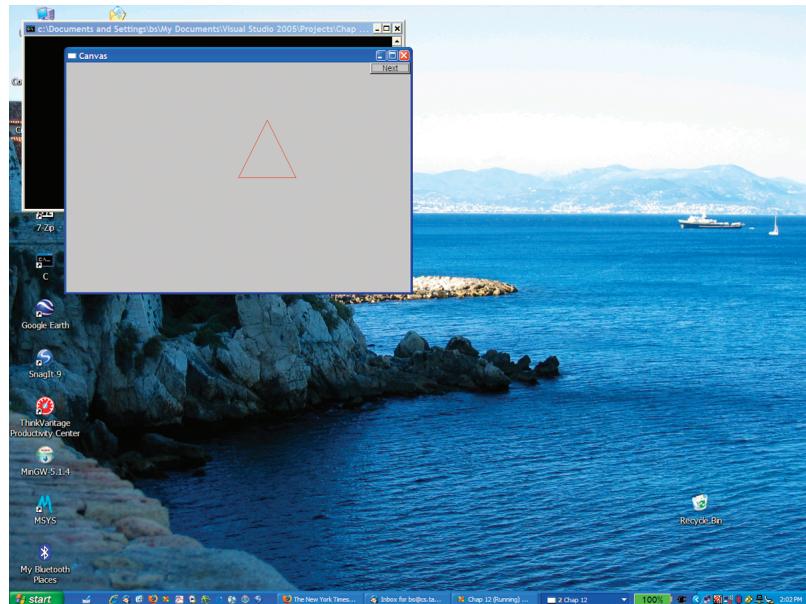
    poly.add(Point{300,200});     // add a point
    poly.add(Point{350,100});     // add another point
    poly.add(Point{400,200});     // add a third point

    poly.set_color(Color::red);   // adjust properties of poly

    win.attach (poly);           // connect poly to the window

    win.wait_for_button();       // give control to the display engine
}
```

When we run this program, the screen looks something like this:



Let's go through the program line by line to see what was done. First we include the headers for our graphics interface libraries:

```
#include "Simple_window.h"      // get access to our window library
#include "Graph.h"              // get access to our graphics library facilities
```

Then, in **main()**, we start by telling the compiler that our graphics facilities are to be found in **Graph_lib**:

```
using namespace Graph_lib;      // our graphics facilities are in Graph_lib
```

Then, we define a point that we will use as the top left corner of our window:

```
Point tl{100,100};            // to become top left corner of window
```

Next, we create a window on the screen:

```
Simple_window win{tl,600,400,"Canvas"};    // make a simple window
```

We use a class representing a window in our **Graph_lib** interface library called **Simple_window**. The name of this particular **Simple_window** is **win**; that is, **win** is a variable of class **Simple_window**. The initializer list for **win** starts with the point to be used as the top left corner, **tl**, followed by **600** and **400**. Those are the width and height, respectively, of the window, as displayed on the screen, measured in pixels. We'll explain in more detail later, but the main point here is that we specify a rectangle by giving its width and height. The string **Canvas** is used to label the window. If you look, you can see the word **Canvas** in the top left corner of the window's frame.

Next, we put an object in the window:

```
Polygon poly;                  // make a shape (a polygon)

poly.add(Point{300,200});       // add a point
poly.add(Point{350,100});       // add another point
poly.add(Point{400,200});       // add a third point
```

We define a polygon, **poly**, and then add points to it. In our graphics library, a **Polygon** starts empty and we can add as many points to it as we like. Since we added three points, we get a triangle. A point is simply a pair of values giving the *x* and *y* (horizontal and vertical) coordinates within a window.

Just to show off, we then color the lines of our polygon red:

```
poly.set_color(Color::red);      // adjust properties of poly
```

Finally, we attach **poly** to our window, **win**:

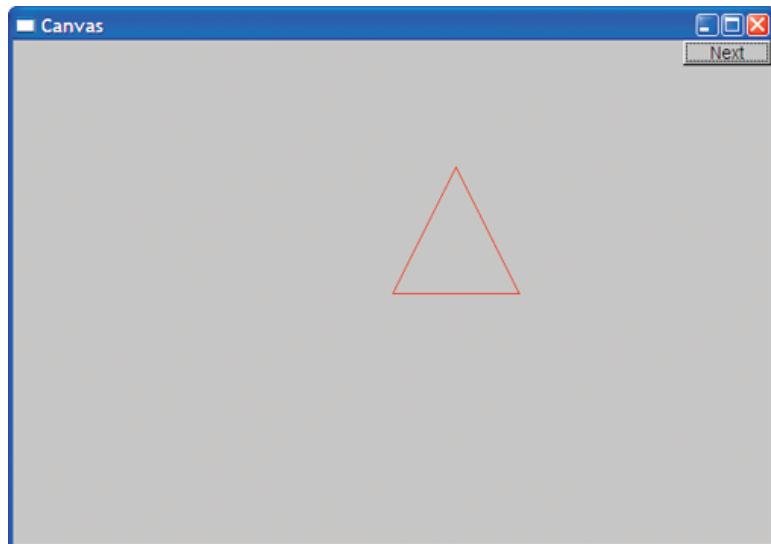
```
win.attach(poly); // connect poly to the window
```

If the program wasn't so fast, you would notice that so far nothing had happened to the screen: nothing at all. We created a window (an object of class **Simple_window**, to be precise), created a polygon (called **poly**), painted that polygon red (**Color::red**), and attached it to the window (called **win**), but we have not yet asked for that window to be displayed on the screen. That's done by the final line of the program:

```
win.wait_for_button(); // give control to the display engine
```

To get a GUI system to display objects on the screen, you have to give control to "the system." Our **wait_for_button()** does that, and it also waits for you to "press" ("click") the "Next" button of our **Simple_window** before proceeding. This gives you a chance to look at the window before the program finishes and the window disappears. When you press the button, the program terminates, closing the window.

In isolation, our window looks like this:



You'll notice that we "cheated" a bit. Where did that button labeled "Next" come from? We built it into our **Simple_window** class. In Chapter 16, we'll move from **Simple_window** to "plain" **Window**, which has no potentially spurious facilities

built in, and show how we can write our own code to control interaction with a window.

For the next three chapters, we'll simply use that "Next" button to move from one "display" to the next when we want to display information in stages ("frame by frame").

You are so used to the operating system putting a frame around each window that you might not have noticed it specifically. However, the pictures in this and the following chapters were produced on a Microsoft Windows system, so you get the usual three buttons on the top right "for free." This can be useful: if your program gets in a real mess (as it surely will sometimes during debugging), you can kill it by hitting the **X** button. When you run your program on another system, a different frame will be added to fit that system's conventions. Our only contribution to the frame is the label (here, [Canvas](#)).

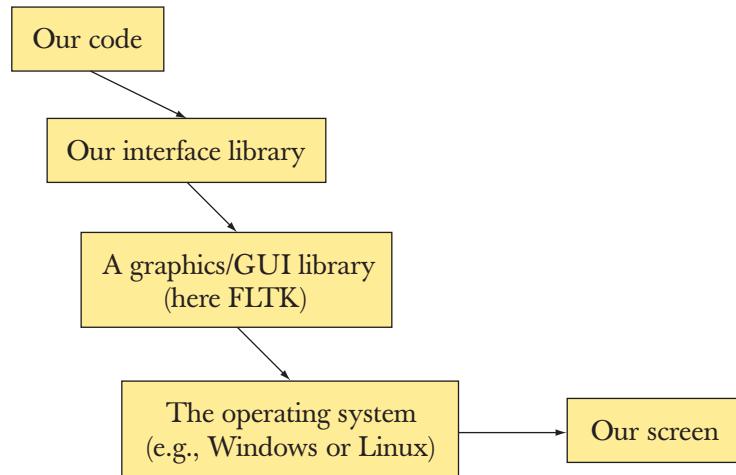
12.4 Using a GUI library

In this book, we will not use the operating system's graphical and GUI (graphical user interface) facilities directly. Doing so would limit our programs to run on a single operating system and would also force us to deal directly with a lot of messy details. As with text I/O, we'll use a library to smooth over operating system differences, I/O device variations, etc. and to simplify our code. Unfortunately, C++ does not provide a standard GUI library the way it provides the standard stream I/O library, so we use one of the many available C++ GUI libraries. So as not to tie you directly into one of those GUI libraries, and to save you from hitting the full complexity of a GUI library all at once, we use a set of simple interface classes that can be implemented in a couple of hundred lines of code for just about any GUI library.

The GUI toolkit that we are using (indirectly for now) is called FLTK (Fast Light Tool Kit, pronounced "full tick") from www.fltk.org. Our code is portable wherever FLTK is used (Windows, Unix, Mac, Linux, etc.). Our interface classes can also be re-implemented using other toolkits, so code using them is potentially even more portable.

The programming model presented by our interface classes is far simpler than what common toolkits offer. For example, our complete graphics and GUI interface library is about 600 lines of C++ code, whereas the extremely terse FLTK documentation is 370 pages. You can download that from www.fltk.org, but we don't recommend you do that just yet. You can do without that level of detail for a while. The general ideas presented in Chapters 12–16 can be used with any popular GUI toolkit. We will of course explain how our interface classes map to FLTK so that you will (eventually) see how you can use that (and similar toolkits) directly, if necessary.

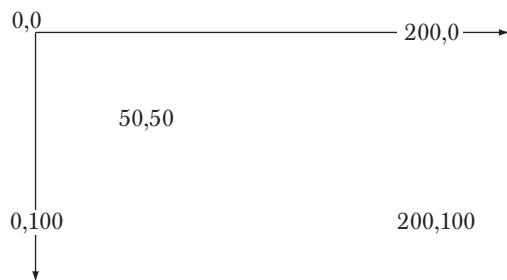
We can illustrate the parts of our “graphics world” like this:



Our interface classes provide a simple and user-extensible basic notion of two-dimensional shapes with limited support for the use of color. To drive that, we present a simple notion of GUI based on “callback” functions triggered by the use of user-defined buttons, etc. on the screen (Chapter 16).

12.5 Coordinates

A computer screen is a rectangular area composed of pixels. A pixel is a tiny spot that can be given some color. The most common way of modeling a screen in a program is as a rectangle of pixels. Each pixel is identified by an x (horizontal) coordinate and a y (vertical) coordinate. The x coordinates start with 0, indicating the leftmost pixel, and increase (toward the right) to the rightmost pixel. The y coordinates start with 0, indicating the topmost pixel, and increase (toward the bottom) to the lowest pixel:





Please note that y coordinates “grow downward.” Mathematicians, in particular, find this odd, but screens (and windows) come in many sizes, and the top left point is about all that they have in common.

The number of pixels available depends on the screen: 1024-by-768, 1280-by-1024, 1400-by-1050, and 1600-by-1200 are common screen sizes.

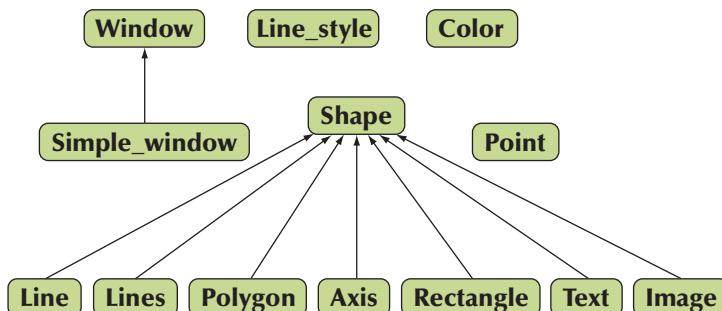
In the context of interacting with a computer using a screen, a window is a rectangular region of the screen devoted to some specific purpose and controlled by a program. A window is addressed exactly like a screen. Basically, we see a window as a small screen. For example, when we said

```
Simple_window win {tl,600,400,"Canvas"};
```

we requested a rectangular area 600 pixels wide and 400 pixels high that we can address as 0–599 (left to right) and 0–399 (top to bottom). The area of a window that you can draw on is commonly referred to as a *canvas*. The 600-by-400 area refers to “the inside” of the window, that is, the area inside the system-provided frame; it does not include the space the system uses for the title bar, quit button, etc.

12.6 Shapes

Our basic toolbox for drawing on the screen consists of about a dozen classes:



An arrow indicates that the class pointing can be used where the class pointed to is required. For example, a **Polygon** can be used where a **Shape** is required; that is, a **Polygon** is a kind of **Shape**.

We will start out presenting and using

- **Simple_window, Window**
- **Shape, Text, Polygon, Line, Lines, Rectangle, Function**, etc.
- **Color, Line_style, Point**
- **Axis**

Later (Chapter 16), we'll add GUI (user interaction) classes:

- **Button**, **In_box**, **Menu**, etc.

We could easily add many more classes (for some definition of “easy”), such as

- **Spline**, **Grid**, **Block_chart**, **Pie_chart**, etc.

However, defining or describing a complete GUI framework with all its facilities is beyond the scope of this book.

12.7 Using Shape primitives

In this section, we will walk you through some of the primitive facilities of our graphics library: **Simple_window**, **Window**, **Shape**, **Text**, **Polygon**, **Line**, **Lines**, **Rectangle**, **Color**, **Line_style**, **Point**, **Axis**. The aim is to give you a broad view of what you can do with those facilities, but not yet a detailed understanding of any of those classes. In the next chapters, we explore the design of each.

We will now walk through a simple program, explaining the code line by line and showing the effect of each on the screen. When you run the program you'll see how the image changes as we add shapes to the window and modify existing shapes. Basically, we are “animating” the progress through the code by looking at the program as it is executed.

12.7.1 Graphics headers and main

First, we include the header files defining our interface to the graphics and GUI facilities:

```
#include "Window.h"           // a plain window
#include "Graph.h"
```

or

```
#include "Simple_window.h"    // if we want that "Next" button
#include "Graph.h"
```

As you probably guessed, **Window.h** contains the facilities related to windows and **Graph.h** the facilities related to drawing shapes (including text) into windows. These facilities are defined in the **Graph_lib** namespace. To simplify notation we use a namespace directive to make the names from **Graph_lib** directly available in our program:

```
using namespace Graph_lib;
```

As usual, **main()** contains the code we want to execute (directly or indirectly) and deals with exceptions:

```
int main ()
try
{
    // . . . here is our code . . .

}
catch(exception& e) {
    // some error reporting
    return 1;
}
catch(...){
    // some more error reporting
    return 2;
}
```

For this **main()** to compile, we need to have **exception** defined. We get that if we include **std_lib_facilities.h** as usual, or we could start to deal directly with standard headers and include **<stdexcept>**.

12.7.2 An almost blank window

We will not discuss error handling here (see Chapter 5, in particular, §5.6.3) but go straight to the graphics within **main()**:

```
Point tl{100,100};           // top left corner of our window

Simple_window win {tl,600,400,"Canvas"};
    // screen coordinate tl for top left corner
    // window size(600*400)
    // title: Canvas
win.wait_for_button();   // display!
```

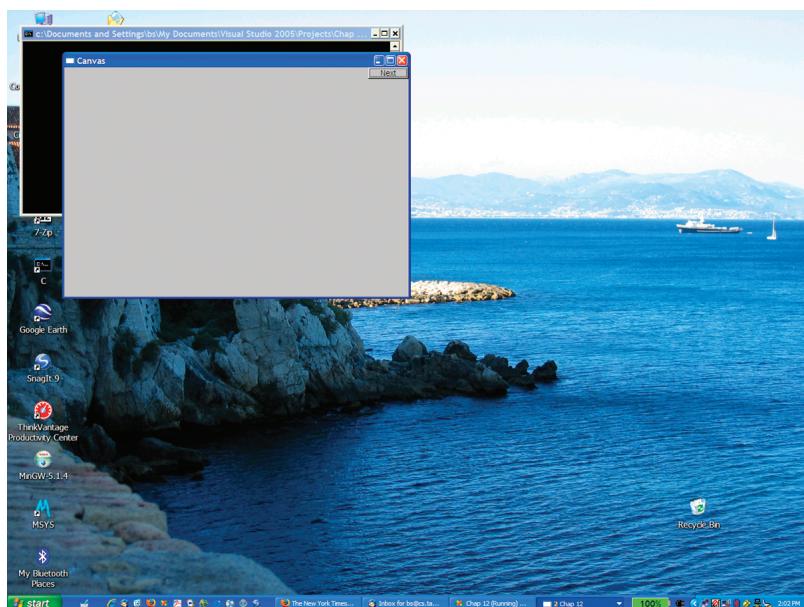
This creates a **Simple_window**, that is, a window with a “Next” button, and displays it on the screen. Obviously, we need to have #included the header **Simple_window.h** rather than **Window.h** to get **Simple_window**. Here we are specific about where on the screen the window should go: its top left corner goes at **Point{100,100}**. That’s near, but not too near, the top left corner of the screen.

Obviously, **Point** is a class with a constructor that takes a pair of integers and interprets them as an (x,y) coordinate pair. We could have written

```
Simple_window win {Point{100,100},600,400,"Canvas"};
```

However, we want to use the point (100,100) several times so it is more convenient to give it a symbolic name. The 600 is the width and 400 is the height of the window, and **Canvas** is the label we want put on the frame of the window.

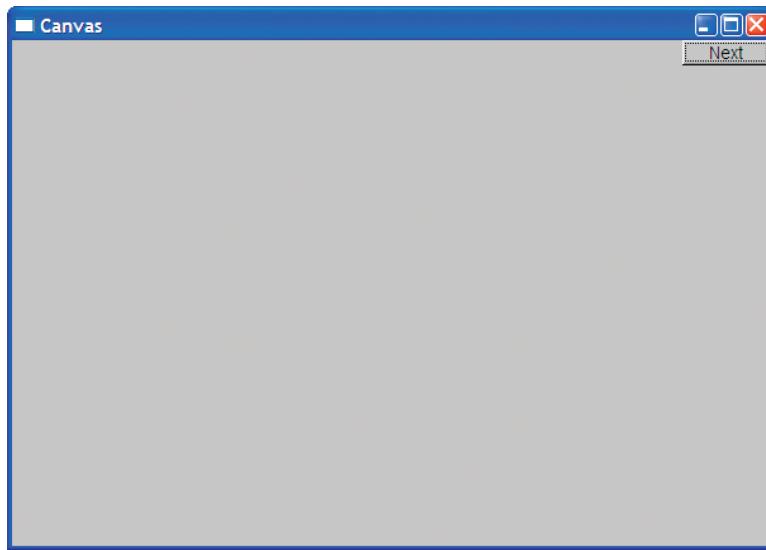
To actually get the window drawn on the screen, we have to give control to the GUI system. We do this by calling **win.wait_for_button()** and the result is:



In the background of our window, we see a laptop screen (somewhat cleaned up for the occasion). For people who are curious about irrelevant details, we can tell you that I took the photo standing near the Picasso library in Antibes looking across the bay to Nice. The black console window partially hidden behind is the one running our program. Having a console window is somewhat ugly and unnecessary, but it has the advantage of giving us an effective way of killing our window if a partially debugged program gets into an infinite loop and refuses to go away. If you look carefully, you'll notice that we have the Microsoft C++ compiler running, but you could just as well have used some other compiler (such as Borland or GNU).



For the rest of the presentation we will eliminate the distractions around our window and just show that window by itself:



The actual size of the window (in inches) depends on the resolution of your screen. Some screens have bigger pixels than other screens.

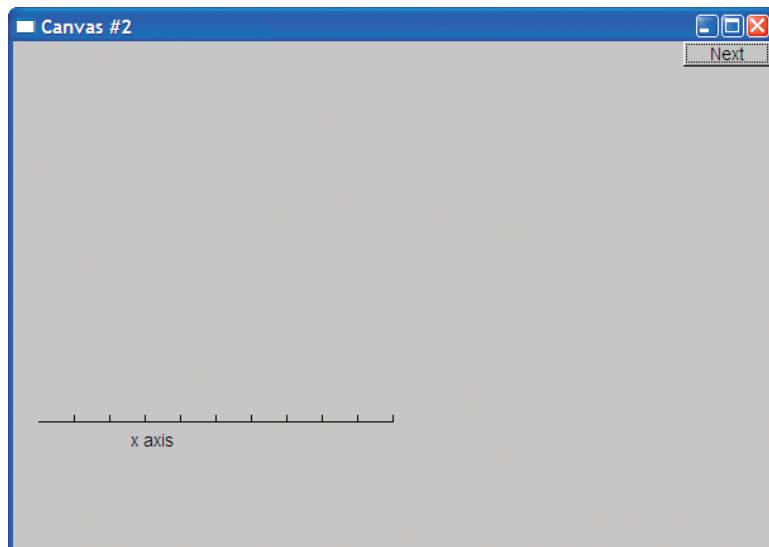
12.7.3 Axis

An almost blank window isn't very interesting, so we'd better add some information. What would we like to display? Just to remind you that graphics is not all fun and games, we will start with something serious and somewhat complicated: an axis. A graph without axes is usually a disgrace. You just don't know what the data represents without axes. Maybe you explained it all in some accompanying text, but it is far safer to add axes; people often don't read the explanation and often a nice graphical representation gets separated from its original context. So, a graph needs axes:

```
Axis xa {Axis::x, Point{20,300}, 280, 10, "x axis"};      // make an Axis
// an Axis is a kind of Shape
// Axis::x means horizontal
// starting at (20,300)
// 280 pixels long
// 10 "notches"
// label the axis "x axis"
```

```
win.attach(xa);           // attach xa to the window, win
win.set_label("Canvas #2"); // relabel the window
win.wait_for_button();     // display!
```

The sequence of actions is: make the axis object, add it to the window, and finally display it:



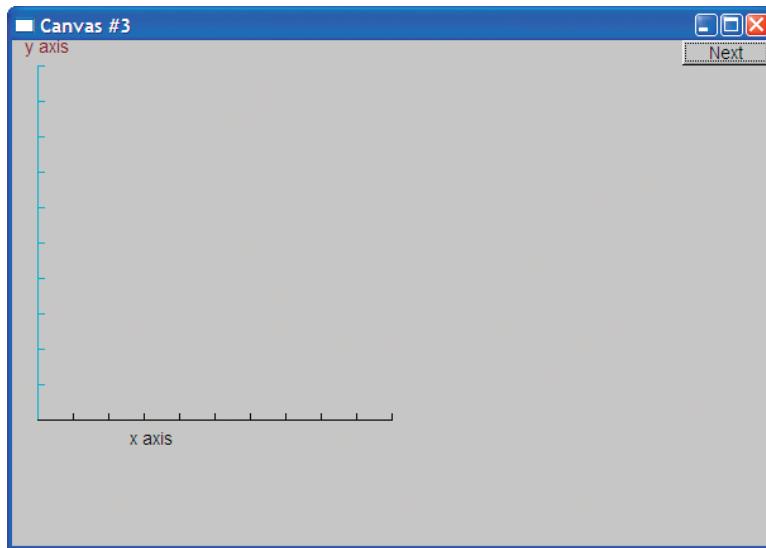
We can see that an **Axis::x** is a horizontal line. We see the required number of “notches” (10) and the label “x axis.” Usually, the label will explain what the axis and the notches represent. Naturally, we chose to place the *x* axis somewhere near the bottom of the window. In real life, we’d represent the height and width by symbolic constants so that we could refer to “just above the bottom” as something like **y_max-bottom_margin** rather than by a “magic constant,” such as **300** (§4.3.1, §15.6.2).

To help identify our output we relabeled the screen to **Canvas #2** using **Window**’s member function **set_label()**.

Now, let’s add a *y* axis:

```
Axis ya {Axis::y, Point{20,300}, 280, 10, "y axis";
ya.set_color(Color::cyan);           // choose a color
ya.label.set_color(Color::dark_red); // choose a color for the text
win.attach(ya);
win.set_label("Canvas #3");
win.wait_for_button();             // display!
```

Just to show off some facilities, we colored our y axis cyan and our label dark red.



We don't actually think that it is a good idea to use different colors for x and y axes. We just wanted to show you how you can set the color of a shape and of individual elements of a shape. Using lots of color is not necessarily a good idea. In particular, novices tend to use color with more enthusiasm than taste.

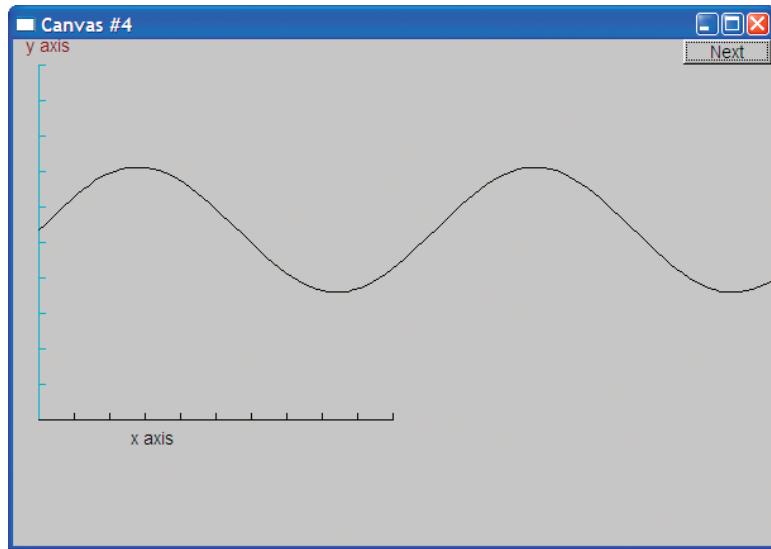
12.7.4 Graphing a function

What next? We now have a window with axes, so it seems a good idea to graph a function. We make a shape representing a sine function and attach it:

```
Function sine {sin,0,100,Point{20,150},1000,50,50}; // sine curve
    // plot sin() in the range [0:100) with (0,0) at (20,150)
    // using 1000 points; scale x values *50, scale y values *50

win.attach(sine);
win.set_label("Canvas #4");
win.wait_for_button();
```

Here, the **Function** named **sine** will draw a sine curve using the standard library function **sin()** to generate values. We explain details about how to graph functions in §15.3. For now, just note that to graph a function we have to say where it starts (a **Point**) and for what set of input values we want to see it (a range), and we need to give some information about how to squeeze that information into our window (scaling):



Note how the curve simply stops when it hits the edge of the window. Points drawn outside our window rectangle are simply ignored by the GUI system and never seen.

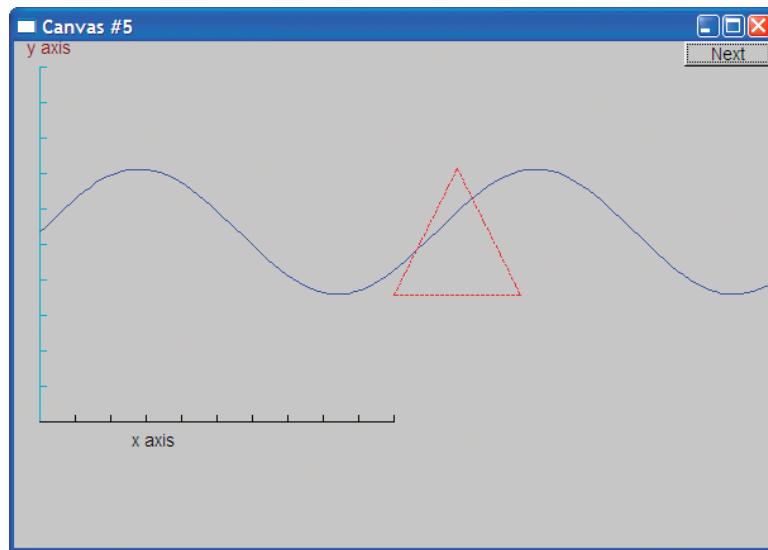
12.7.5 Polygons

A graphed function is an example of data presentation. We'll see much more of that in Chapter 15. However, we can also draw different kinds of objects in a window: geometric shapes. We use geometric shapes for graphical illustrations, to indicate user interaction elements (such as buttons), and generally to make our presentations more interesting. A **Polygon** is characterized by a sequence of points, which the **Polygon** class connects by lines. The first line connects the first point to the second, the second line connects the second point to the third, and the last line connects the last point to the first:

```
sine.set_color(Color::blue); // we changed our mind about sine's color  
  
Polygon poly; // a polygon; a Polygon is a kind of Shape  
poly.add(Point{300,200}); // three points make a triangle  
poly.add(Point{350,100});  
poly.add(Point{400,200});  
  
poly.set_color(Color::red);  
poly.set_style(Line_style::dash);  
win.attach(poly);
```

```
win.set_label("Canvas #5");
win.wait_for_button();
```

This time we change the color of the sine curve (`sine`) just to show how. Then, we add a triangle, just as in our first example from §12.3, as an example of a polygon. Again, we set a color, and finally, we set a style. The lines of a `Polygon` have a “style.” By default that is solid, but we can also make those lines dashed, dotted, etc. as needed (see §13.5). We get



12.7.6 Rectangles

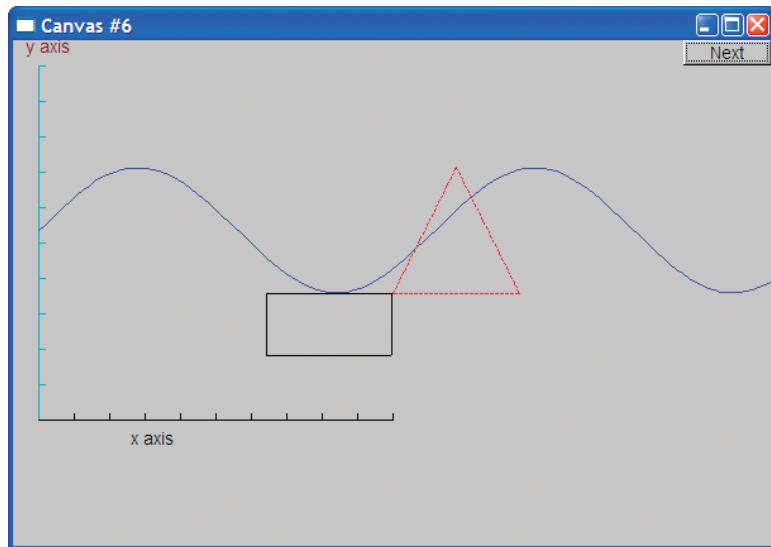
A screen is a rectangle, a window is a rectangle, and a piece of paper is a rectangle. In fact, an awful lot of the shapes in our modern world are rectangles (or at least rectangles with rounded corners). There is a reason for this: a rectangle is the simplest shape to deal with. For example, it's easy to describe (top left corner plus width plus height, or top left corner plus bottom right corner, or whatever), it's easy to tell whether a point is inside a rectangle or outside it, and it's easy to get hardware to draw a rectangle of pixels fast.

So, most higher-level graphics libraries deal better with rectangles than with other closed shapes. Consequently, we provide `Rectangle` as a class separate from

the **Polygon** class. A **Rectangle** is characterized by its top left corner plus a width and height:

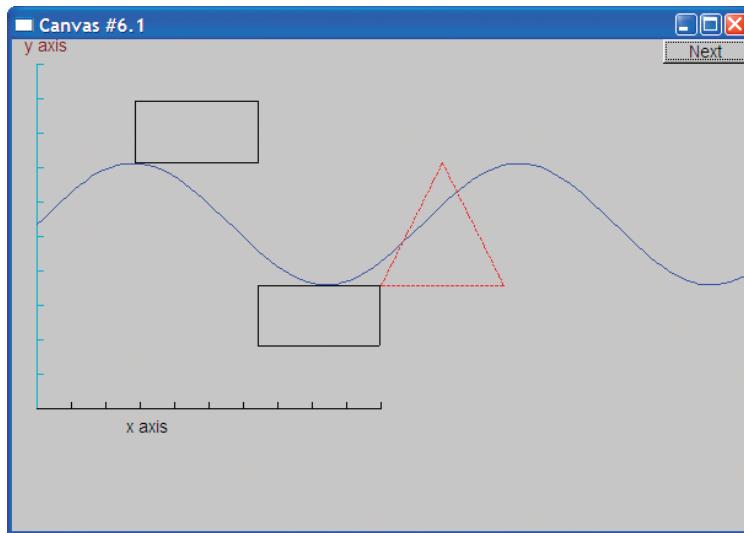
```
Rectangle r {Point{200,200}, 100, 50};      // top left corner, width, height
win.attach(r);
win.set_label("Canvas #6");
win.wait_for_button();
```

From that, we get



Please note that making a polyline with four points in the right places is not enough to make a **Rectangle**. It is easy to make a **Closed_polyline** that looks like a **Rectangle** on the screen (you can even make an **Open_polyline** that looks just like a **Rectangle**); for example:

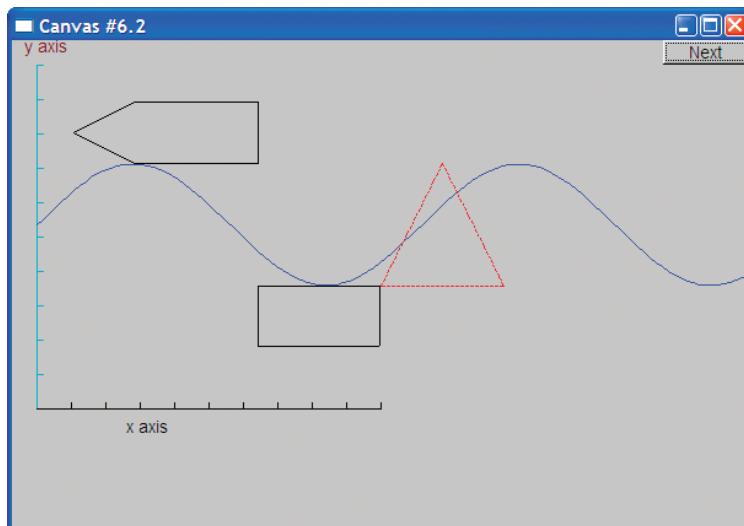
```
Closed_polyline poly_rect;
poly_rect.add(Point{100,50});
poly_rect.add(Point{200,50});
poly_rect.add(Point{200,100});
poly_rect.add(Point{100,100});
win.attach(poly_rect);
```



In fact, the *image* on the screen of such a **poly_rect** is a rectangle. However, the **poly_rect** object in memory is not a **Rectangle** and it does not “know” anything about rectangles. The simplest way to prove that is to add another point:

```
poly_rect.add(Point{50,75});
```

No rectangle has five points:



It is important for our reasoning about our code that a **Rectangle** doesn’t just happen to look like a rectangle on the screen; it maintains the fundamental guarantees

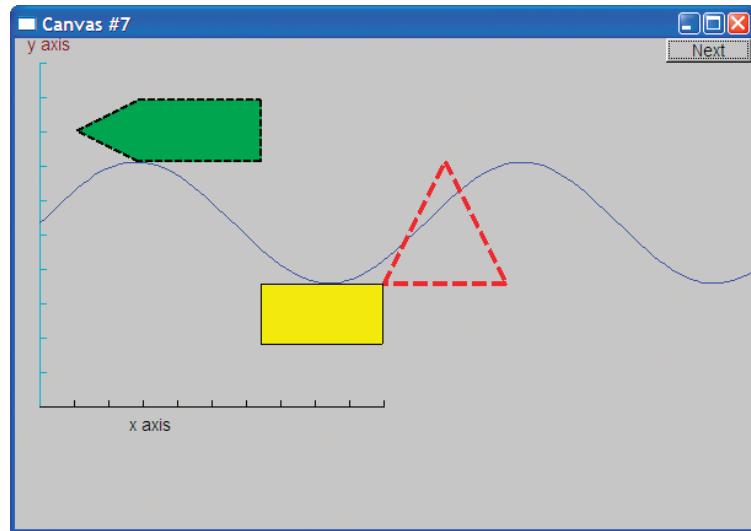
of a rectangle (as we know them from geometry). We write code that depends on a **Rectangle** really being a rectangle on the screen and staying that way.

12.7.7 Fill

We have been drawing our shapes as outlines. We can also “fill” a rectangle with color:

```
r.set_fill_color(Color::yellow);      // color the inside of the rectangle
poly.set_style(Line_style(Line_style::dash,4));
poly_rect.set_style(Line_style(Line_style::dash,2));
poly_rect.set_fill_color(Color::green);
win.set_label("Canvas #7");
win.wait_for_button();
```

We also decided that we didn’t like the line style of our triangle (**poly**), so we set its line style to “fat (thickness four times normal) dashed.” Similarly, we changed the style of **poly_rect** (now no longer looking like a rectangle):



If you look carefully at **poly_rect**, you’ll see that the outline is printed on top of the fill.

It is possible to fill any closed shape (see §13.9). Rectangles are just special in how easy (and fast) they are to fill.

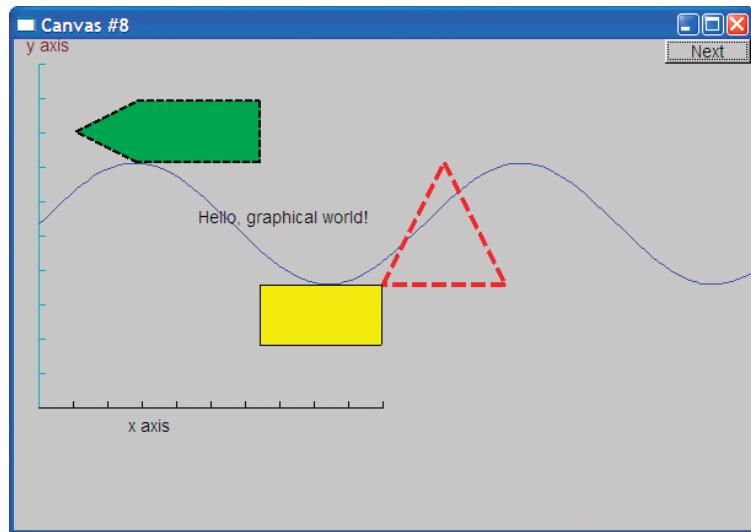
12.7.8 Text

Finally, no system for drawing is complete without a simple way of writing text – drawing each character as a set of lines just doesn’t cut it. We label the



window itself, and axes can have labels, but we can also place text anywhere using a **Text** object:

```
Text t {Point{150,150}, "Hello, graphical world!"};
win.attach(t);
win.set_label("Canvas #8");
win.wait_for_button();
```

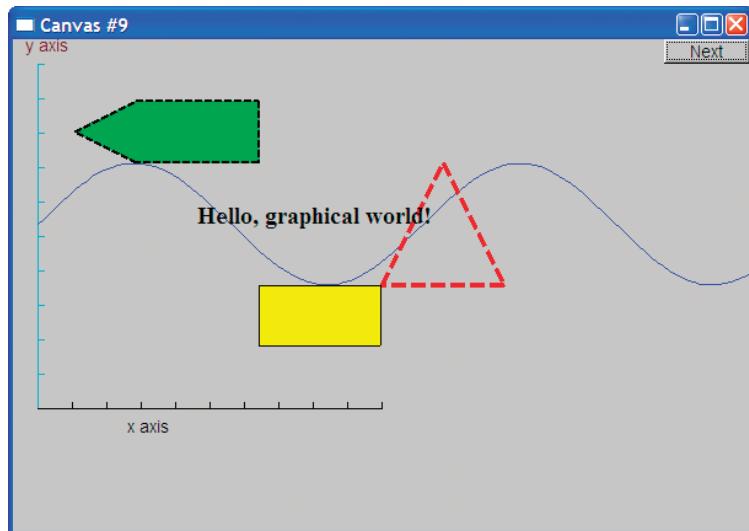


From the primitive graphics elements you see in this window, you can build displays of just about any complexity and subtlety. For now, just note a peculiarity of the code in this chapter: there are no loops, no selection statements, and all data was “hardwired” in. The output was just composed of primitives in the simplest possible way. Once we start composing these primitives using data and algorithms, things will start to get interesting.

We have seen how we can control the color of text: the label of an **Axis** (§12.7.3) is simply a **Text** object. In addition, we can choose a font and set the size of the characters:

```
t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Canvas #9");
win.wait_for_button();
```

We enlarged the characters of the **Text** string **Hello, graphical world!** to point size 20 and chose the Times font in bold:

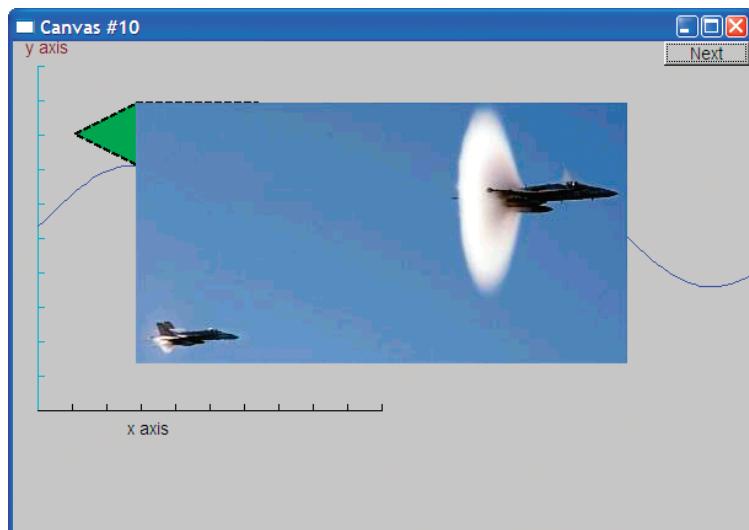


12.7.9 Images

We can also load images from files:

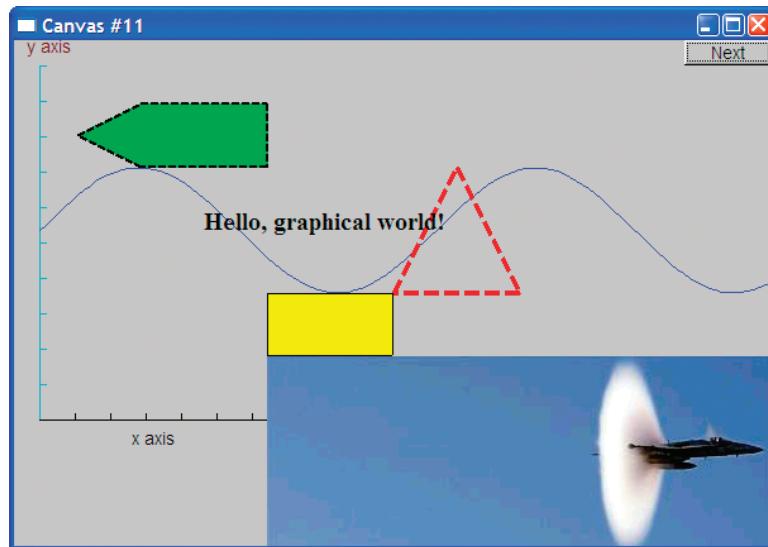
```
Image ii {Point{100,50}, "image.jpg"};      // 400*212-pixel jpg
win.attach(ii);
win.set_label("Canvas #10");
win.wait_for_button();
```

As it happens, the file called **image.jpg** is a photo of two planes breaking the sound barrier:



That photo is relatively large and we placed it right on top of our text and shapes. So, to clean up our window a bit, let us move it a bit out of the way:

```
ii.move(100,200);
win.set_label("Canvas #11");
win.wait_for_button();
```



Note how the parts of the photo that didn't fit in the window are simply not represented. What would have appeared outside the window is “clipped” away.

12.7.10 And much more

And here, without further comment, is some more code:

```
Circle c {Point{100,200},50};
Ellipse e {Point{100,200}, 75,25};
e.set_color(Color::dark_red);
Mark m {Point{100,200},'x'};

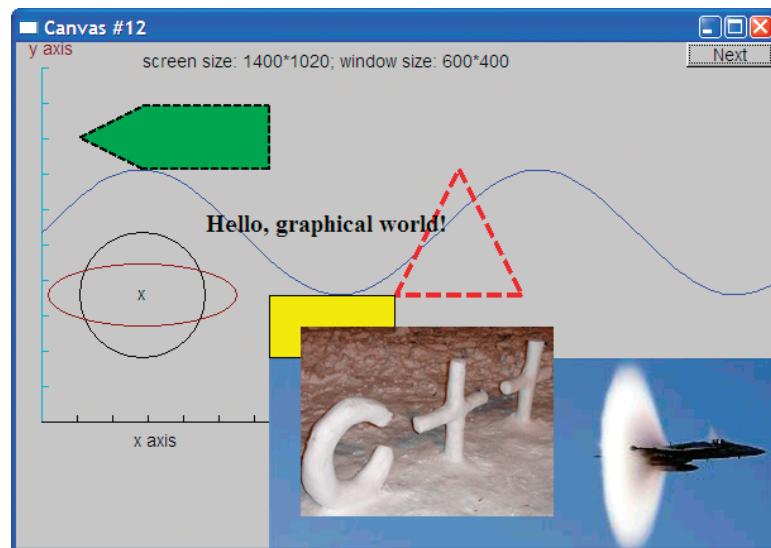
ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Text sizes {Point{100,20},oss.str()};

Image cal {Point{225,225}, "snow_cpp.gif"};      // 320*240-pixel gif
cal.set_mask(Point{40,40},200,150);           // display center part of image
```

```
win.attach(c);
win.attach(m);
win.attach(e);

win.attach(sizes);
win.attach(cal);
win.set_label("Canvas #12");
win.wait_for_button();
```

Can you guess what this code does? Is it obvious?



The connection between the code and what appears on the screen is direct. If you don't yet see how that code caused that output, it soon will become clear. Note the way we used an `ostringstream` (§11.4) to format the text object displaying sizes.



12.8 Getting this to run

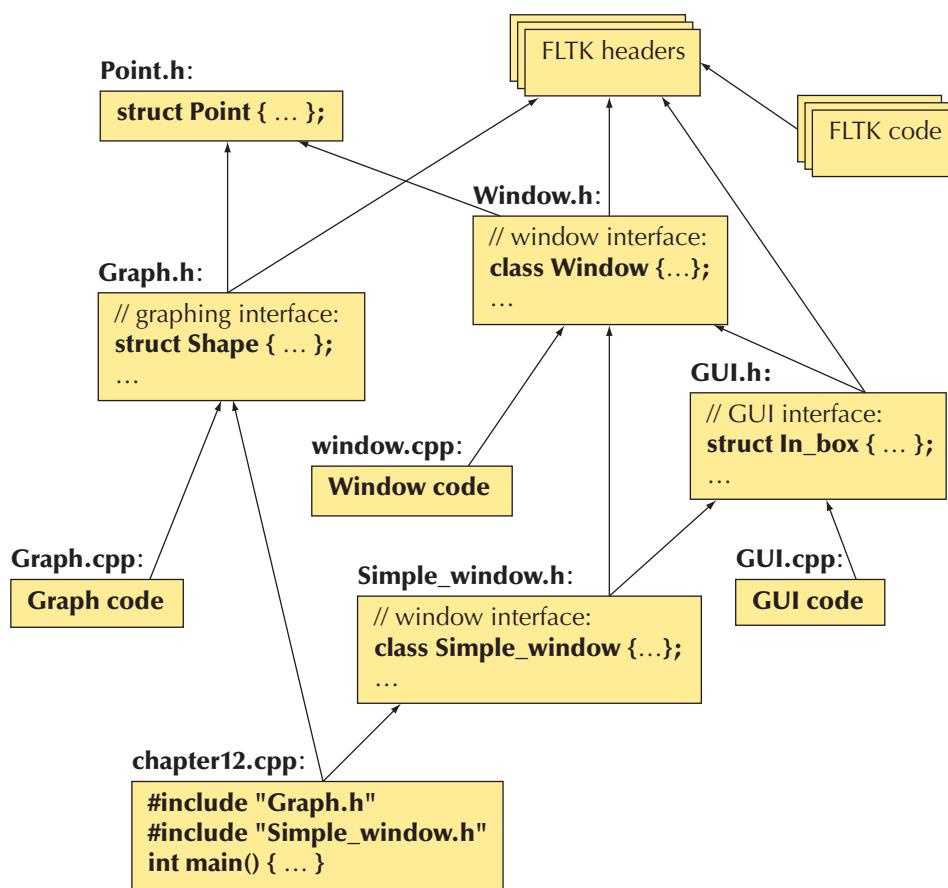
We have seen how to make a window and how to draw various shapes in it. In the following chapters, we'll see how those `Shape` classes are defined and show more ways of using them.

Getting this program to run requires more than the programs we have presented so far. In addition to our code in `main()`, we need to get the interface library code compiled and linked to our code, and finally, nothing will run unless the FLTK library (or whatever GUI system we use) is installed and correctly linked to ours.

One way of looking at the program is that it has four distinct parts:

- Our program code (`main()`, etc.)
- Our interface library (`Window`, `Shape`, `Polygon`, etc.)
- The FLTK library
- The C++ standard library

Indirectly, we also use the operating system. Leaving out the OS and the standard library, we can illustrate the organization of our graphics code like this:



Appendix D explains how to get all of this to work together.

12.8.1 Source files

Our graphics and GUI interface library consists of just five header files and three code files:

- Headers:
 - **Point.h**
 - **Window.h**
 - **Simple_window.h**
 - **Graph.h**
 - **GUI.h**
- Code files:
 - **Window.cpp**
 - **Graph.cpp**
 - **GUI.cpp**

Until Chapter 16, you can ignore the GUI files.



Drill

The drill is the graphical equivalent to the “Hello, World!” program. Its purpose is to get you acquainted with the simplest graphical output tools.

1. Get an empty **Simple_window** with the size 600 by 400 and a label **My window** compiled, linked, and run. Note that you have to link the FLTK library as described in Appendix D; **#include Graph.h** and **Simple_window.h** in your code; and include **Graph.cpp** and **Window.cpp** in your project.
2. Now add the examples from §12.7 one by one, testing between each added subsection example.
3. Go through and make one minor change (e.g., in color, in location, or in number of points) to each of the subsection examples.

Review

1. Why do we use graphics?
2. When do we try not to use graphics?
3. Why is graphics interesting for a programmer?

4. What is a window?
5. In which namespace do we keep our graphics interface classes (our graphics library)?
6. What header files do you need to do basic graphics using our graphics library?
7. What is the simplest window to use?
8. What is the minimal window?
9. What's a window label?
10. How do you label a window?
11. How do screen coordinates work? Window coordinates? Mathematical coordinates?
12. What are examples of simple “shapes” that we can display?
13. What command attaches a shape to a window?
14. Which basic shape would you use to draw a hexagon?
15. How do you write text somewhere in a window?
16. How would you put a photo of your best friend in a window (using a program you wrote yourself)?
17. You made a **Window** object, but nothing appears on your screen. What are some possible reasons for that?
18. You have made a shape, but it doesn't appear in the window. What are some possible reasons for that?

Terms

color	graphics	JPEG
coordinates	GUI	line style
display	GUI library	software layer
fill color	HTML	window
FLTK	image	XML

Exercises

We recommend that you use **Simple_window** for these exercises.

1. Draw a rectangle as a **Rectangle** and as a **Polygon**. Make the lines of the **Polygon** red and the lines of the **Rectangle** blue.
2. Draw a 100-by-30 **Rectangle** and place the text “Howdy!” inside it.
3. Draw your initials 150 pixels high. Use a thick line. Draw each initial in a different color.
4. Draw a 3-by-3 tic-tac-toe board of alternating white and red squares.
5. Draw a red 1/4-inch frame around a rectangle that is three-quarters the height of your screen and two-thirds the width.

6. What happens when you draw a **Shape** that doesn't fit inside its window? What happens when you draw a **Window** that doesn't fit on your screen? Write two programs that illustrate these two phenomena.
7. Draw a two-dimensional house seen from the front, the way a child would: with a door, two windows, and a roof with a chimney. Feel free to add details; maybe have "smoke" come out of the chimney.
8. Draw the Olympic five rings. If you can't remember the colors, look them up.
9. Display an image on the screen, e.g., a photo of a friend. Label the image both with a title on the window and with a caption in the window.
10. Draw the file diagram from §12.8.
11. Draw a series of regular polygons, one inside the other. The innermost should be an equilateral triangle, enclosed by a square, enclosed by a pentagon, etc. For the mathematically adept only: let all the points of each **N**-polygon touch sides of the **(N+1)**-polygon. Hint: The trigonometric functions are found in `<cmath>` (§24.8, §B.9.2).
12. A superellipse is a two-dimensional shape defined by the equation

$$\left| \frac{x}{a} \right|^m + \left| \frac{y}{b} \right|^n = 1; \quad m, n > 0.$$

- Look up *superellipse* on the web to get a better idea of what such shapes look like. Write a program that draws "starlike" patterns by connecting points on a superellipse. Take **a**, **b**, **m**, **n**, and **N** as arguments. Select **N** points on the superellipse defined by **a**, **b**, **m**, and **n**. Make the points equally spaced for some definition of "equal." Connect each of those **N** points to one or more other points (if you like you can make the number of points to which to connect a point another argument or just use **N-1**, i.e., all the other points).
13. Find a way to add color to the lines from the previous exercise. Make some lines one color and other lines another color or other colors.

Postscript



The ideal for program design is to have our concepts directly represented as entities in our program. So, we often represent ideas by classes, real-world entities by objects of classes, and actions and computations by functions. Graphics is a domain where this idea has an obvious application. We have concepts, such as circles and polygons, and we represent them in our program as class **Circle** and class **Polygon**. Where graphics is unusual is that when writing a graphics program, we also have the opportunity to see objects of those classes on the screen; that is, the

state of our program is directly represented for us to observe – in most applications we are not that lucky. This direct correspondence between ideas, code, and output is what makes graphics programming so attractive. Please do remember, though, that graphics are just illustrations of the general idea of using classes to directly represent concepts in code. That idea is far more general and useful: just about anything we can think of can be represented in code as a class, an object of a class, or a set of classes.



Graphics Classes

“A language that doesn’t change the way you think isn’t worth learning.”

—Traditional

Chapter 12 gave an idea of what we could do in terms of graphics using a set of simple interface classes, and how we can do it. This chapter presents many of the classes offered. The focus here is on the design, use, and implementation of individual interface classes such as **Point**, **Color**, **Polygon**, and **Open polyline** and their uses. The following chapter will present ideas for designing sets of related classes and will also present more implementation techniques.

- | | |
|--|---------------------------------------|
| 13.1 Overview of graphics classes | 13.10 Managing unnamed objects |
| 13.2 Point and Line | 13.11 Text |
| 13.3 Lines | 13.12 Circle |
| 13.4 Color | 13.13 Ellipse |
| 13.5 Line_style | 13.14 Marked_polyline |
| 13.6 Open_polyline | 13.15 Marks |
| 13.7 Closed_polyline | 13.16 Mark |
| 13.8 Polygon | 13.17 Images |
| 13.9 Rectangle | |

13.1 Overview of graphics classes

Graphics and GUI libraries provide lots of facilities. By “lots” we mean hundreds of classes, often with dozens of functions applying to each. Reading a description, manual, or documentation is a bit like looking at an old-fashioned botany textbook listing details of thousands of plants organized according to obscure classifying traits. It is daunting! It can also be exciting – looking at the facilities of a modern graphics/GUI library can make you feel like a child in a candy store, but it can be hard to figure out where to start and what is really good for you.

One purpose of our interface library is to reduce the shock delivered by the complexity of a full-blown graphics/GUI library. We present just two dozen classes with hardly any operations. Yet they allow you to produce useful graphical output. A closely related goal is to introduce key graphics and GUI concepts through those classes. Already, you can write programs displaying results as simple graphics. After this chapter, your range of graphics programs will have increased to exceed most people’s initial requirements. After Chapter 14, you’ll understand most of the design techniques and ideas involved so that you can deepen your understanding and extend your range of graphical expression as needed. You can do so either by adding to the facilities described here or by adopting another C++ graphics/GUI library.

The key interface classes are:

Graphics interface classes	
Color	used for lines, text, and filling shapes
Line_style	used to draw lines
Point	used to express locations on a screen and within a Window

Graphics interface classes (<i>continued</i>)	
Line	a line segment as we see it on the screen, defined by its two end Points
Open_polyline	a sequence of connected line segments defined by a sequence of Points
Closed_polyline	like an Open_polyline , except that a line segment connects the last Point to the first
Polygon	a Closed_polyline where no two line segments intersect
Text	a string of characters
Lines	a set of line segments defined by pairs of Points
Rectangle	a common shape optimized for quick and convenient display
Circle	a circle defined by a center and a radius
Ellipse	an ellipse defined by a center and two axes
Function	a function of one variable graphed in a range
Axis	a labeled axis
Mark	a point marked by a character (such as x or o)
Marks	a sequence of points indicated by marks (such as x and o)
Marked_polyline	an Open_polyline with its points indicated by marks
Image	the contents of an image file

Chapter 15 examines **Function** and **Axis**. Chapter 16 presents the main GUI interface classes:

GUI interface classes	
Window	an area of the screen in which we display our graphics objects
Simple_window	a window with a “Next” button
Button	a rectangle, usually labeled, in a window that we can press to run one of our functions
In_box	a box, usually labeled, in a window into which a user can type a string
Out_box	a box, usually labeled, in a window into which our program can write a string
Menu	a vector of Buttons

The source code is organized into files like this:

Graphics interface source files	
Point.h	Point
Graph.h	all other graphics interface classes
Window.h	Window
Simple_window.h	Simple_window
GUI.h	Button and the other GUI classes
Graph.cpp	definitions of functions from Graph.h
Window.cpp	definitions of functions from Window.h
GUI.cpp	definitions of functions from GUI.h

In addition to the graphics classes, we present a class that happens to be useful for holding collections for **Shapes** or **Widgets**:

A container of Shapes or Widgets	
Vector_ref	a vector with an interface that makes it convenient for holding unnamed elements

When you read the following sections, please don't move too fast. There is little that isn't pretty obvious, but the purpose of this chapter isn't just to show you some pretty pictures – you see prettier pictures on your computer screen or television every day. The main points of this chapter are

- To show the correspondence between code and the pictures produced.
- To get you used to reading code and thinking about how it works.
- To get you to think about the design of code – in particular to think about how to represent concepts as classes in code. Why do those classes look the way they do? How else could they have looked? We made many, many design decisions, most of which could reasonably have been made differently, in some cases radically differently.

So please don't rush. If you do, you'll miss something important and you might then find the exercises unnecessarily hard.

13.2 Point and Line

The most basic part of any graphics system is the point. To define *point* is to define how we organize our geometric space. Here, we use a conventional, computer-

oriented layout of two-dimensional points defined by (x,y) integer coordinates. As described in §12.5, x coordinates go from **0** (representing the left-hand side of the screen) to **x_max()** (representing the right-hand side of the screen); y coordinates go from **0** (representing the top of the screen) to **y_max()** (representing the bottom of the screen).

As defined in **Point.h**, **Point** is simply a pair of **ints** (the coordinates):

```
struct Point {
    int x, y;
};

bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
bool operator!=(Point a, Point b) { return !(a==b); }
```

In **Graph.h**, we find **Shape**, which we describe in detail in Chapter 14, and **Line**:

```
struct Line : Shape {                                // a Line is a Shape defined by two Points
    Line(Point p1, Point p2);                      // construct a Line from two Points
};
```

A **Line** is a kind of **Shape**. That's what **: Shape** means. **Shape** is called a *base class* for **Line** or simply a *base* of **Line**. Basically, **Shape** provides the facilities needed to make the definition of **Line** simple. Once we have a feel for the particular shapes, such as **Line** and **Open_polyline**, we'll explain what that implies (Chapter 14).

A **Line** is defined by two **Points**. Leaving out the “scaffolding” (**#includes**, etc. as described in §12.3), we can create lines and cause them to be drawn like this:

```
// draw two lines

constexpr Point x {100,100};

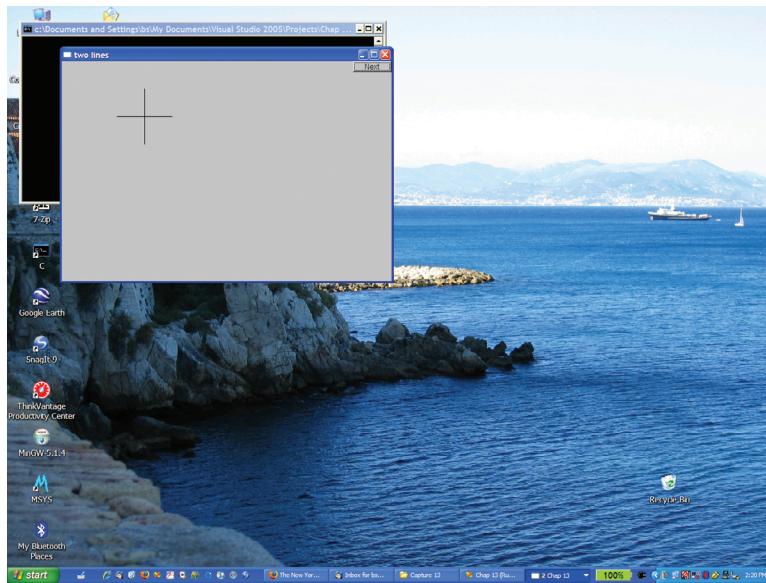
Simple_window win1 {x,600,400,"two lines"};

Line horizontal {x,Point{200,100}};          // make a horizontal line
Line vertical {Point{150,50},Point{150,150}}; // make a vertical line

win1.attach(horizontal);                    // attach the lines to the window
win1.attach(vertical);

win1.wait_for_button();                     // display!
```

Executing that, we get



As a user interface designed for simplicity, **Line** works quite well. You don't need to be Einstein to guess that

```
Line vertical {Point{150,50},Point{150,150}};
```

creates a (vertical) line from (150,50) to (150,150). There are, of course, implementation details, but you don't have to know those to make **Lines**. The implementation of **Line**'s constructor is correspondingly simple:

```
Line::Line(Point p1, Point p2) // construct a line from two points
{
    add(p1); // add p1 to this shape
    add(p2); // add p2 to this shape
}
```

That is, it simply “adds” two points. Adds to what? And how does a **Line** get drawn in a window? The answer lies in the **Shape** class. As we'll describe in Chapter 14, **Shape** can hold points defining lines, knows how to draw lines defined by pairs of **Points**, and provides a function **add()** that allows an object to add a **Point** to its **Shape**. The key point (*sic!*) here is that defining **Line** is trivial. Most of the implementation work is done by “the system” so that we can concentrate on writing simple classes that are easy to use.

From now on we'll leave out the definition of the **Simple_window** and the calls of **attach()**. Those are just more "scaffolding" that we need for a complete program but that adds little to the discussion of specific **Shapes**.

13.3 Lines

As it turns out, we rarely draw just one line. We tend to think in terms of objects consisting of many lines, such as triangles, polygons, paths, mazes, grids, bar graphs, mathematical functions, graphs of data, etc. One of the simplest such "composite graphical object classes" is **Lines**:

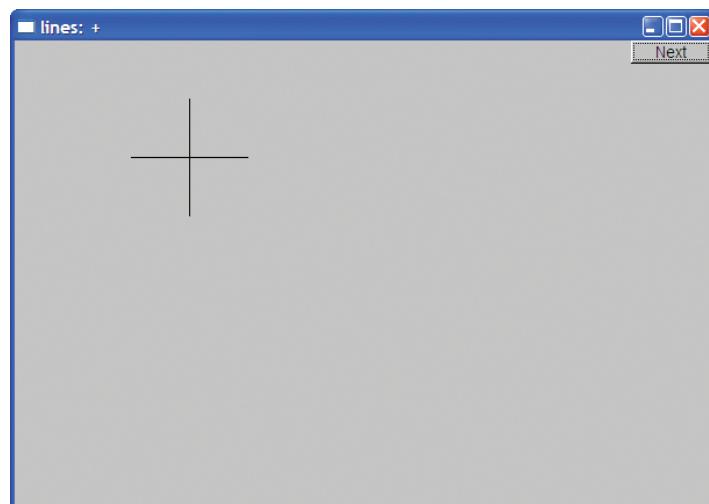
```
struct Lines : Shape {                                // related lines
    Lines() {}                                         // empty
    Lines(initializer_list<Point> lst);           // initialize from a list of Points

    void draw_lines() const;
    void add(Point p1, Point p2);                  // add a line defined by two points
};
```

A **Lines** object is simply a collection of lines, each defined by a pair of **Points**. For example, had we considered the two lines from the **Line** example in §13.2 as part of a single graphical object, we could have defined them like this:

```
Lines x;
x.add(Point{100,100}, Point{200,100};          // first line: horizontal
x.add(Point{150,50}, Point{150,150};          // second line: vertical
```

This gives output that is indistinguishable (to the last pixel) from the **Line** version:



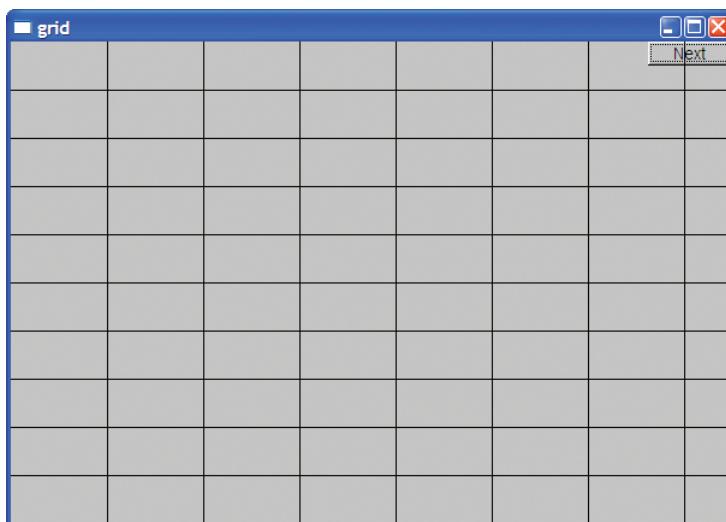
The only way we can tell that this is a different window is that we labeled them differently.

The difference between a set of **Line** objects and a set of lines in a **Lines** object is completely one of our view of what's going on. By using **Lines**, we have expressed our opinion that the two lines belong together and should be manipulated together. For example, we can change the color of all lines that are part of a **Lines** object with a single command. On the other hand, we can give lines that are individual **Line** objects different colors. As a more realistic example, consider how to define a grid. A grid consists of a number of evenly spaced horizontal and vertical lines. However, we think of a grid as one "thing," so we define those lines as part of a **Lines** object, which we call **grid**:

```
int x_size = win3.x_max();           // get the size of our window
int y_size = win3.y_max();
int x_grid = 80;
int y_grid = 40;

Lines grid;
for (int x=x_grid; x<x_size; x+=x_grid)
    grid.add(Point{x,0},Point{x,y_size}); // vertical line
for (int y = y_grid; y<y_size; y+=y_grid)
    grid.add(Point{0,y},Point{x_size,y}); // horizontal line
```

Note how we get the dimension of our window using **x_max()** and **y_max()**. This is also the first example where we are writing code that computes which objects we want to display. It would have been unbearably tedious to define this grid by defining one named variable for each grid line. From that code, we get



Let's return to the design of **Lines**. How are the member functions of class **Lines** implemented? **Lines** provides just two constructors and two operations.

The **add()** function simply adds a line defined by a pair of points to the set of lines to be displayed:

```
void Lines::add(Point p1, Point p2)
{
    Shape::add(p1);
    Shape::add(p2);
}
```

Yes, the **Shape::** qualification is needed because otherwise the compiler would see **add(p1)** as an (illegal) attempt to call **Lines'** **add()** rather than **Shape'** **add()**.

The **draw_lines()** function draws the lines defined using **add()**:

```
void Lines::draw_lines() const
{
    if (color().visibility())
        for (int i=1; i<number_of_points(); i+=2)
            fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

That is, **Lines::draw_lines()** takes two points at a time (starting with points 0 and 1) and draws the line between them using the underlying library's line-drawing function (**fl_line()**). Visibility is a property of the **Lines'** **Color** object (§13.4), so we have to check that the lines are meant to be visible before drawing them.

As we explain in Chapter 14, **draw_lines()** is called by "the system." We don't need to check that the number of points is even — **Lines'** **add()** can add only pairs of points. The functions **number_of_points()** and **point()** are defined in class **Shape** (§14.2) and have their obvious meaning. These two functions provide read-only access to a **Shape**'s points. The member function **draw_lines()** is defined to be **const** (see §9.7.4) because it doesn't modify the shape.

The default constructor for **Lines** simply creates an empty object (containing no lines): the model of starting out with no points and then **add**ing points as needed is more flexible than any constructor could be. However, we also added a constructor taking an **initializer_list** of pairs of **Points**, each defining a line. Given that initializer-list constructor (§18.2), we can simply define **Lines** starting out with 0, 1, 2, , 3, . . . lines. For example, the first **Lines** example could be written like this:

```
Lines x = {
    {Point{100,100}, Point{200,100}},      // first line: horizontal
    {Point{150,50}, Point{150,150}}                // second line: vertical
};
```



or even like this:

```
Lines x = {  
    {{100,100}, {200,100}},    // first line: horizontal  
    {{150,50}, {150,150}}     // second line: vertical  
};
```

The initializer-list constructor is easily defined:

```
void Lines::Lines(initializer_list<pair<Point,Point>> lst)  
{  
    for (auto p : lst) add(p.first,p.second);  
}
```

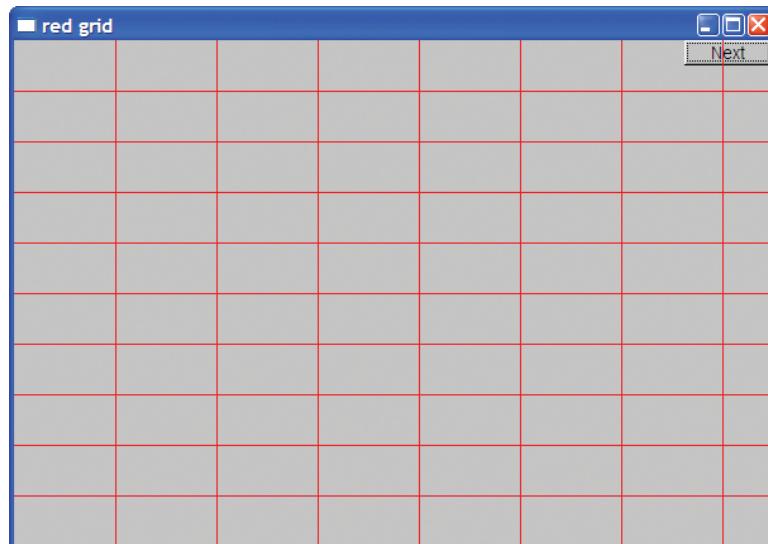
The **auto** is a placeholder for the type **pair<Point,Point>**, and **first** and **second** are the names of a pair's first and second members. The types **initializer_list** and **pair** are defined in the standard library (§B.6.4, §B.6.3).

13.4 Color

Color is the type we use to represent color. We can use **Color** like this:

```
grid.set_color(Color::red);
```

This colors the lines defined in **grid** red so that we get



Color defines the notion of a color and gives symbolic names to a few of the more common colors:

```

struct Color {
    enum Color_type {
        red=FL_RED,
        blue=FL_BLUE,
        green=FL_GREEN,
        yellow=FL_YELLOW,
        white=FL_WHITE,
        black=FL_BLACK,
        magenta=FL_MAGENTA,
        cyan=FL_CYAN,
        dark_red=FL_DARK_RED,
        dark_green=FL_DARK_GREEN,
        dark_yellow=FL_DARK_YELLOW,
        dark_blue=FL_DARK_BLUE,
        dark_magenta=FL_DARK_MAGENTA,
        dark_cyan=FL_DARK_CYAN
    };

    enum Transparency { invisible = 0, visible=255 };

    Color(Color_type cc) :c{Fl_Color(cc)}, v{visible} {}
    Color(Color_type cc, Transparency vv) :c{Fl_Color(cc)}, v{vv} {}
    Color(int cc) :c{Fl_Color(cc)}, v{visible} {}
    Color(Transparency vv) :c{Fl_Color()}, v{vv} {} // default color

    int as_int() const { return c; }

    char visibility() const { return v; }
    void set_visibility(Transparency vv) { v=vv; }
private:
    char v; // invisible and visible for now
    Fl_Color c;
};

```

The purpose of **Color** is

- To hide the implementation's notion of color, FLTK's **Fl_Color** type
- To map between **Fl_Color** and **Color_type** values

- To give the color constants a scope
- To provide a simple version of transparency (visible and invisible)

You can pick colors

- From the list of named colors, for example, `Color::dark_blue`.
- By picking from a small “palette” of colors that most screens display well by specifying a value in the range 0–255; for example, `Color(99)` is a dark green. For a code example, see §13.9.
- By picking a value in the RGB (red, green, blue) system, which we will not explain here. Look it up if you need it. In particular, a search for “RGB color” on the web gives many sources, such as http://en.wikipedia.org/wiki/RGB_color_model and www.rapidtables.com/web/color/RGB_Color.htm. See also exercises 13 and 14.



Note the use of constructors to allow `Colors` to be created either from the `Color_type` or from a plain `int`. The member `c` is initialized by each constructor. You could argue that `c` is too short and too obscure a name to use, but since it is used only within the small scope of `Color` and not intended for general use, that’s probably OK. We made the member `c` private to protect it from direct use from our users. For our representation of the data member `c` we use the FLTK type `Fl_Color` that we don’t really want to expose to our users. However, looking at a color as an `int` representing its RGB (or other) value is very common, so we supplied `as_int()` for that. Note that `as_int()` is a `const` member because it doesn’t actually change the `Color` object that it is used for.

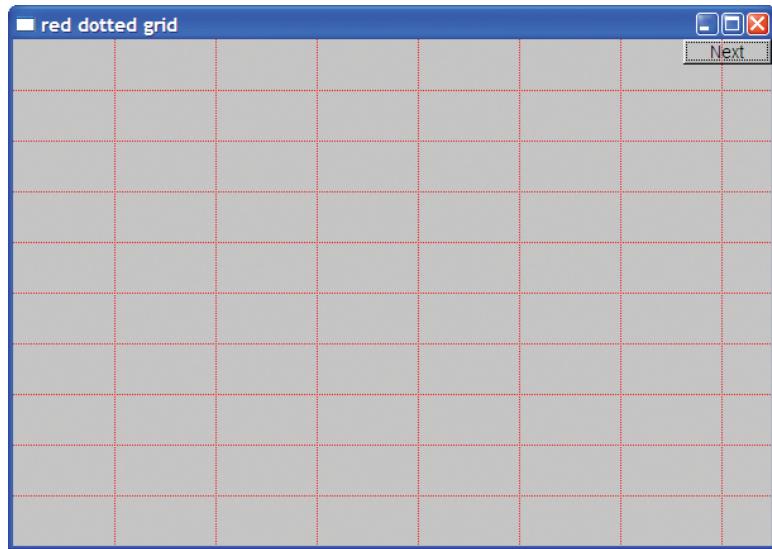
The transparency is represented by the member `v` which can hold the values `Color::visible` and `Color::invisible`, with their obvious meaning. It may surprise you that an “invisible color” can be useful, but it can be most useful to have part of a composite shape invisible.

13.5 Line_style

When we draw several lines in a window, we can distinguish them by color, by style, or by both. A line style is the pattern used to outline the line. We can use `Line_style` like this:

```
grid.set_style(Line_style::dot);
```

This displays the lines in `grid` as a sequence of dots rather than a solid line:



That “thinned out” the grid a bit, making it more discreet. By adjusting the width (thickness), we can adjust the grid lines to suit our taste and needs.

The **Line_style** type looks like this:

```
struct Line_style {
    enum Line_style_type {
        solid=FL_SOLID,           // -----
        dash=FL_DASH,             // - - -
        dot=FL_DOT,               // .....
        dashdot=FL_DASHDOT,       // - . - .
        dashdotdot=FL_DASHDOTDOT, // -.-.
    };
    Line_style(Line_style_type ss) :s{ss}, w{0} { }
    Line_style(Line_style_type lst, int ww) :s{lst}, w{ww} { }
    Line_style(int ss) :s{ss}, w{0} { }

    int width() const { return w; }
    int style() const { return s; }
private:
    int s;
    int w;
};
```

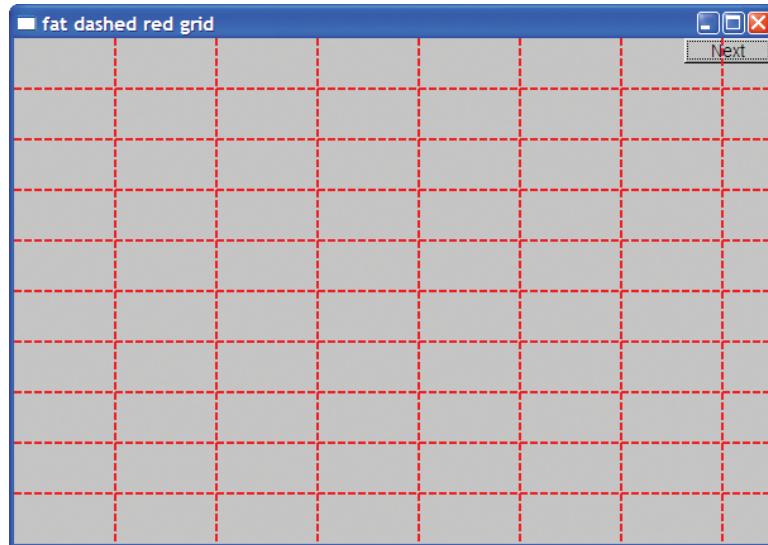
The programming techniques for defining **Line_style** are exactly the same as the ones we used for **Color**. Here, we hide the fact that FLTK uses plain **ints** to represent line styles. Why is something like that worth hiding? Because it is exactly such a detail that might change as a library evolves. The next FLTK release might very well have a **FL_linestyle** type, or we might retarget our interface classes to some other GUI library. In either case, we wouldn't like to have our code and our users' code littered with plain **ints** that we just happened to know represent line styles.

Most of the time, we don't worry about style at all; we just rely on the default (default width and solid lines). This default line width is defined by the constructors in the cases where we don't specify one explicitly. Setting defaults is one of the things that constructors are good for, and good defaults can significantly help users of a class.

Note that **Line_style** has two “components”: the style proper (e.g., use dashed or solid lines) and width (the thickness of the line used). The width is measured in integers. The default width is 1. We can request a fat dashed line like this:

```
grid.set_style(Line_style{Line_style::dash,2});
```

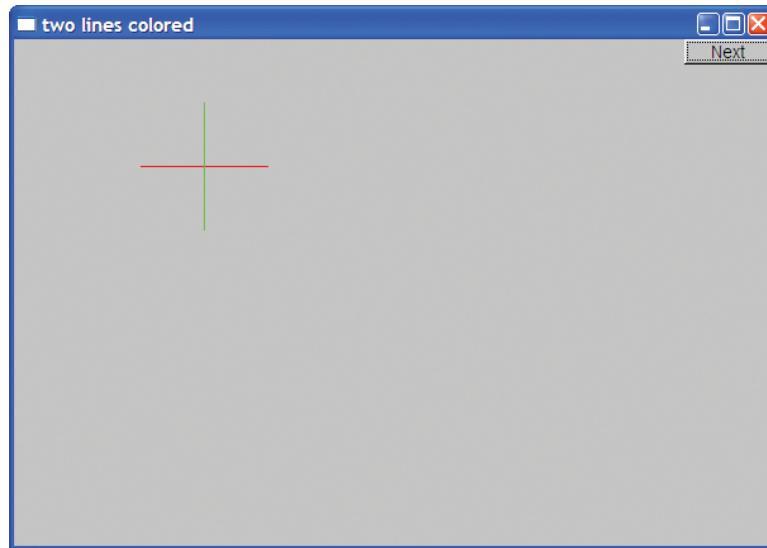
This produces



Note that color and style apply to all lines of a shape. That is one of the advantages of grouping many lines into a single graphics object, such as a [Lines](#), [Open_polyline](#), or [Polygon](#). If we want to control the color or style for lines separately, we must define them as separate [Lines](#). For example:

```
horizontal.set_color(Color::red);
vertical.set_color(Color::green);
```

This gives us

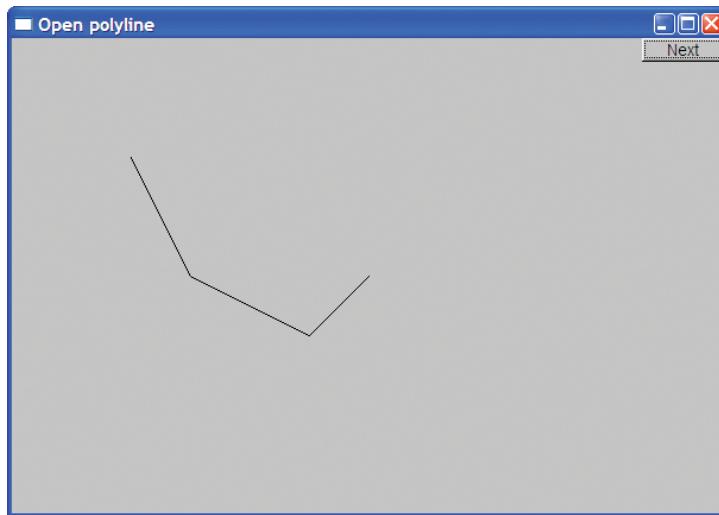


13.6 Open_polyline

An [Open_polyline](#) is a shape that is composed of a series of connected line segments defined by a series of points. *Poly* is the Greek word for “many,” and *polyline* is a fairly conventional name for a shape composed of many lines. For example:

```
Open_polyline opl = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

This draws the shape that you get by connecting the four points:



Basically, an **Open_polyline** is a fancy word for what we encountered in kindergarten playing “Connect the Dots.”

Class **Open_polyline** is defined like this:

```
struct Open_polyline : Shape {      // open sequence of lines
    using Shape::Shape;           // use Shape's constructors (§A.16)
    void add(Point p) { Shape::add(p); }
};
```

Open_polyline inherits from **Shape**. **Open_polyline**’s **add()** function is there to allow the users of an **Open_polyline** to access the **add()** from **Shape** (that is, **Shape::add()**). We don’t even need to define a **draw_lines()** because **Shape** by default interprets the **Points add()**ed as a sequence of connected lines.

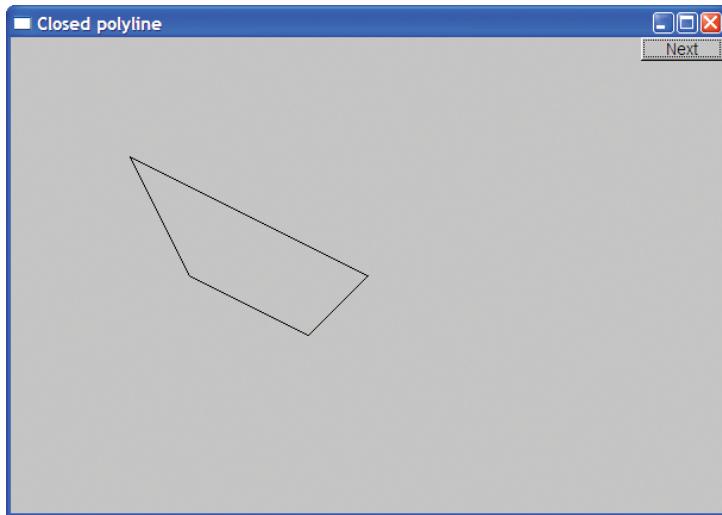
The declaration **using Shape::Shape** is a **using** declaration. It says that an **Open_polyline** can use the constructors defined for **Shape**. **Shape** has a default constructor (§9.7.3) and an initializer-list constructor (§18.2), so the **using** declaration is simply a shorthand for defining those two constructors for **Open_polyline**. As for **Lines**, the initializer-list constructor is there as a shorthand for an initial sequence of **add()**s.

13.7 Closed_polyline

A **Closed_polyline** is just like an **Open_polyline**, except that we also draw a line from the last point to the first. For example, we could use the same points we used for the **Open_polyline** in §13.6 for a **Closed_polyline**:

```
Closed_polyline cpl = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

The result is (of course) identical to that of §13.6 except for that final closing line:



The definition of **Closed_polyline** is

```
struct Closed_polyline : Open_polyline {           // closed sequence of lines
    using Open_polyline::Open_polyline;          // use Open_polyline's
                                                    // constructors (§A.16)
    void draw_lines() const;
};

void Closed_polyline::draw_lines() const
{
    Open_polyline::draw_lines();                  // first draw the "open polyline part"

    // then draw closing line:
    if (2<number_of_points() && color().visibility())
        fl_line(point(number_of_points()-1).x,
            point(number_of_points()-1).y,
            point(0).x,
            point(0).y);
}
```

The **using** declaration (§A.16) says that **Closed_polyline** has the same constructors as **Open_polyline**. **Closed_polyline** needs its own **draw_lines()** to draw that closing line connecting the last point to the first.

We only have to do the little detail where **Closed_polyline** differs from what **Open_polyline** offers. That's important and is sometimes called "programming by difference." We need to program only what's different about our derived class (here, **Closed_polyline**) compared to what a base class (here, **Open_polyline**) offers.

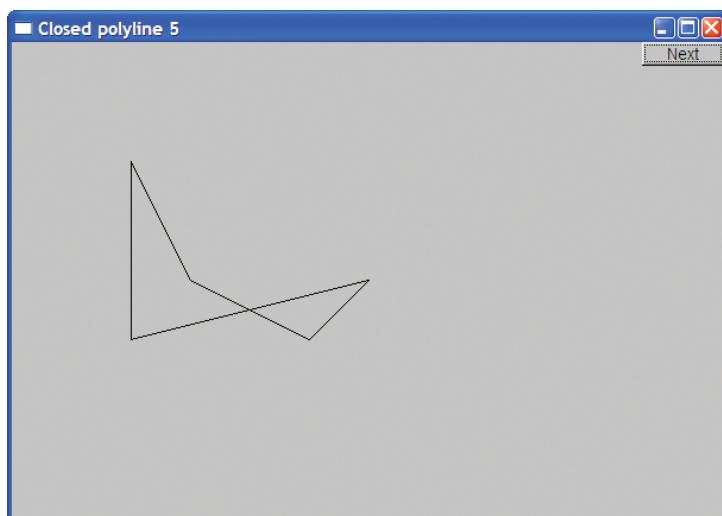
So how do we draw that closing line? We use the FLTK line-drawing function **fl_line()**. It takes four **ints** representing two points. So, here the underlying graphics library is again used. Note, however, that – as in every other case – the mention of FLTK is kept within the implementation of our class rather than being exposed to our users. No user code needs to mention **fl_line()** or to know about interfaces where points appear implicitly as integer pairs. If we wanted to, we could replace FLTK with another GUI library with very little impact on our users' code.

13.8 Polygon

A **Polygon** is very similar to a **Closed_polyline**. The only difference is that for **Polygons** we don't allow lines to cross. For example, the **Closed_polyline** above is a polygon, but we can add another point:

```
cpl.add(Point{100,250});
```

The result is



According to classical definitions, this **Closed_polyline** is not a polygon. How do we define **Polygon** so that we correctly capture the relationship to **Closed_polyline** without violating the rules of geometry? The presentation above contains a strong hint. A **Polygon** is a **Closed_polyline** where lines do not cross. Alternatively, we could emphasize the way a shape is built out of points and say that a **Polygon** is a **Closed_polyline** where we cannot add a **Point** that defines a line segment that intersects one of the existing lines of the **Polygon**.

Given that idea, we define **Polygon** like this:

```
struct Polygon : Closed_polyline {      // closed sequence of nonintersecting
                                         // lines
    using Closed_polyline::Closed_polyline; // use Closed_polyline's
                                         // constructors
    void add(Point p);
    void draw_lines() const;
};

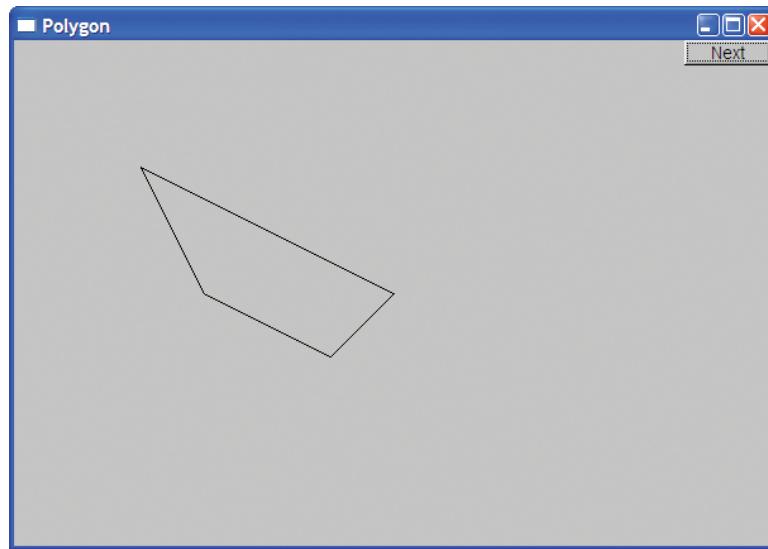
void Polygon::add(Point p)
{
    // check that the new line doesn't intersect existing lines (code not shown)
    Closed_polyline::add(p);
}
```

Here we inherit **Closed_polyline**'s definition of **draw_lines()**, thus saving a fair bit of work and avoiding duplication of code. Unfortunately, we have to check each **add()**. That yields an inefficient (order N^2) algorithm – defining a **Polygon** with N points requires $N*(N-1)/2$ calls of **intersect()**. In effect, we have made the assumption that the **Polygon** class will be used for polygons of a low number of points. For example, creating a **Polygon** with 24 **Points** involves $24*(24-1)/2 == 276$ calls of **intersect()**. That's probably acceptable, but if we wanted a polygon with 2000 points it would cost us about 2,000,000 calls, and we might look for a better algorithm, which might require a modified interface.

Using the initializer-list constructor, we can create a polygon like this:

```
Polygon poly = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

Obviously, this creates a **Polygon** that (to the last pixel) is identical to our original **Closed_polyline**:



Ensuring that a **Polygon** really represents a polygon turned out to be surprisingly messy. The check for intersection that we left out of **Polygon::add()** is arguably the most complicated in the whole graphics library. If you are interested in fiddly coordinate manipulation of geometry, have a look at the code.

The trouble is that **Polygon**'s invariant “the points represent a polygon” can't be verified until all points have been defined; that is, we are not – as strongly recommended – establishing **Polygon**'s invariant in its constructor. We considered removing **add()** and requiring that a **Polygon** be completely specified by an initializer list with at least three points, but that would have complicated uses where a program generated a sequence of points.

13.9 Rectangle

The most common shape on a screen is a rectangle. The reasons for that are partly cultural (most of our doors, windows, pictures, walls, bookcases, pages, etc. are also rectangular) and partly technical (keeping a coordinate within rectangular space is simpler than for any other shaped space). Anyway, rectangles are

so common that GUI systems support them directly rather than treating them simply as polygons that happen to have four corners and right angles.

```
struct Rectangle : Shape {
    Rectangle(Point xy, int ww, int hh);
    Rectangle(Point x, Point y);
    void draw_lines() const;

    int height() const { return h; }
    int width() const { return w; }
private:
    int h; // height
    int w; // width
};
```

We can specify a rectangle by two points (top left and bottom right) or by one point (top left) and a width and a height. The constructors can be defined like this:

```
Rectangle::Rectangle(Point xy, int ww, int hh)
    : w{ww}, h{hh}
{
    if (h<=0 || w<=0)
        error("Bad rectangle: non-positive side");
    add(xy);
}

Rectangle::Rectangle(Point x, Point y)
    :w{y.x-x.x}, h{y.y-x.y}
{
    if (h<=0 || w<=0)
        error("Bad rectangle: first point is not top left");
    add(x);
}
```

Each constructor initializes the members **h** and **w** appropriately (using the member initialization syntax; see §9.4.4) and stores away the top left corner point in the **Rectangle**'s base **Shape** (using **add()**). In addition, it does a simple sanity check: we don't really want **Rectangles** with negative width or height.



One of the reasons that some graphics/GUI systems treat rectangles as special is that the algorithm for determining which pixels are inside a rectangle is far simpler – and therefore far faster – than for other shapes, such as **Polygons** and **Circles**. Consequently, the notion of “fill color” – that is, the color of the space inside the rectangle – is more commonly used for rectangles than for other shapes. We can set the fill color in a constructor or by the operation **set_fill_color()** (provided by **Shape** together with the other services related to color):

```
Rectangle rect00 {Point{150,100},200,100};  
Rectangle rect11 {Point{50,50},Point{250,150}};  
Rectangle rect12 {Point{50,150},Point{250,250}}; // just below rect11  
Rectangle rect21 {Point{250,50},200,100}; // just to the right of rect11  
Rectangle rect22 {Point{250,150},200,100}; // just below rect21  
  
rect00.set_fill_color(Color::yellow);  
rect11.set_fill_color(Color::blue);  
rect12.set_fill_color(Color::red);  
rect21.set_fill_color(Color::green);
```

This produces

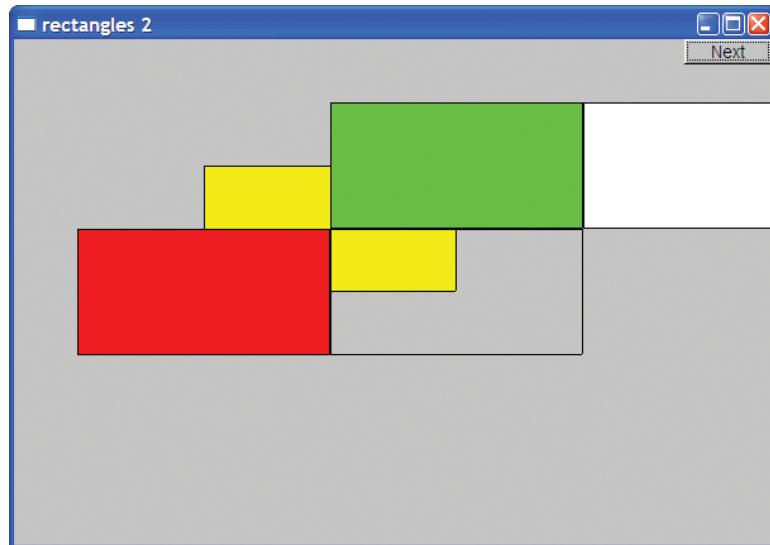


When you don't have a fill color, the rectangle is transparent; that's how you can see a corner of the yellow `rect00`.

We can move shapes around in a window (§14.2.3). For example:

```
rect11.move(400,0);      // to the right of rect21
rect11.set_fill_color(Color::white);
win12.set_label("rectangles 2");
```

This produces

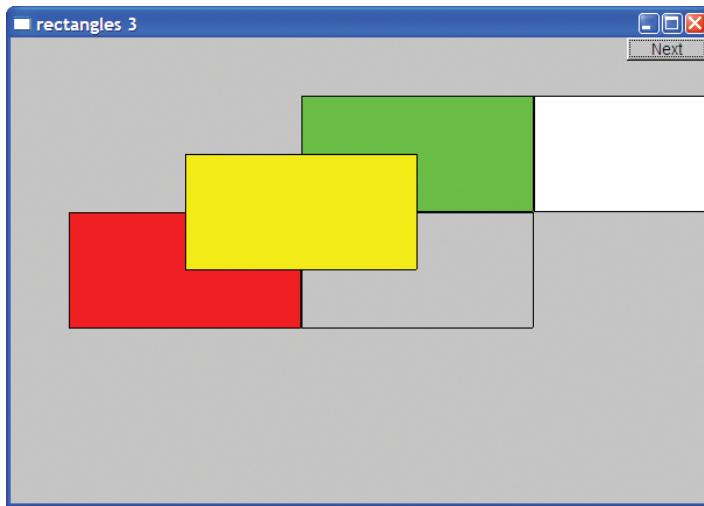


Note how only part of the white `rect11` fits in the window. What doesn't fit is “clipped”; that is, it is not shown anywhere on the screen.

Note also how shapes are placed one on top of another. This is done just like you would put sheets of paper on a table. The first one you put will be on the bottom. Our **Window** (§E.3) provides a simple way of reordering shapes. You can tell a window to put a shape on top (using **Window::put_on_top()**). For example:

```
win12.put_on_top(rect00);
win12.set_label("rectangles 3");
```

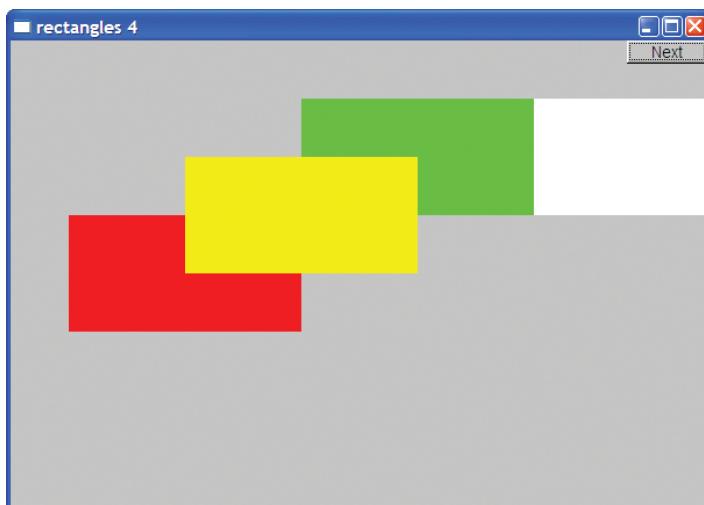
This produces



Note that we can see the lines that make up the rectangles even though we have filled (all but one of) them. If we don't like those outlines, we can remove them:

```
rect00.set_color(Color::invisible);
rect11.set_color(Color::invisible);
rect12.set_color(Color::invisible);
rect21.set_color(Color::invisible);
rect22.set_color(Color::invisible);
```

We get



Note that with both fill color and line color set to `invisible`, `rect22` can no longer be seen.

Because it has to deal with both line color and fill color, `Rectangle`'s `draw_lines()` is a bit messy:

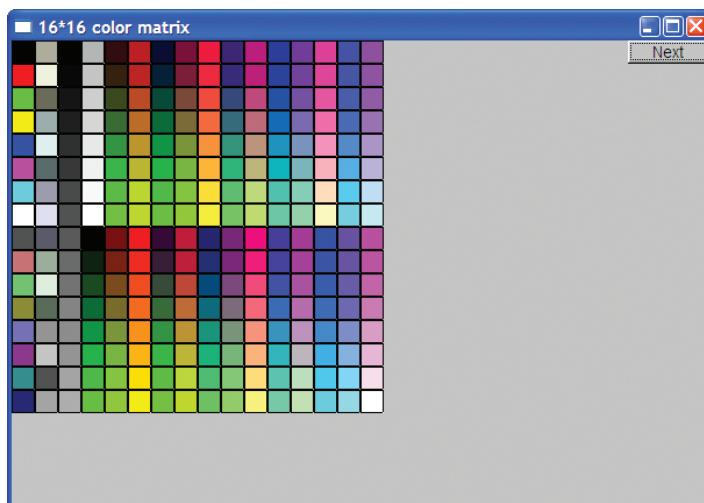
```
void Rectangle::draw_lines() const
{
    if (fill_color().visibility()) { // fill
        fl_color(fill_color().as_int());
        fl_rectf(point(0).x,point(0).y,w,h);
    }

    if (color().visibility()) { // lines on top of fill
        fl_color(color().as_int());
        fl_rect(point(0).x,point(0).y,w,h);
    }
}
```

As you can see, FLTK provides functions for drawing rectangle fill (`fl_rectf()`) and rectangle outlines (`fl_rect()`). By default, we draw both (with the lines/outline on top).

13.10 Managing unnamed objects

So far, we have named all our graphical objects. When we want lots of objects, this becomes infeasible. As an example, let us draw a simple color chart of the 256 colors in FLTK's palette; that is, let's make 256 colored squares and draw them in a 16-by-16 matrix that shows how colors with similar color values relate. First, here is the result:



Naming those 256 squares would not only be tedious, it would be silly. The obvious “name” of the top left square is its location in the matrix (0,0), and any other square is similarly identified (“named”) by a coordinate pair (i,j). What we need for this example is the equivalent of a matrix of objects. We thought of using a **vector<Rectangle>**, but that turned out to be not quite flexible enough. For example, it can be useful to have a collection of unnamed objects (elements) that are not all of the same type. We discuss that flexibility issue in §14.3. Here, we’ll just present our solution: a **vector** type that can hold named and unnamed objects:

```
template<class T> class Vector_ref {
public:
    // ...
    void push_back(T&);      // add a named object
    void push_back(T*);      // add an unnamed object

    T& operator[](int i);    // subscripting: read and write access
    const T& operator[](int i) const;

    int size() const;
};
```

The way you use it is very much like a standard library **vector**:

```
Vector_ref<Rectangle> rect;

Rectangle x {Point{100,200},Point{200,300}};
rect.push_back(x);          // add named

rect.push_back(new Rectangle{Point{50,60},Point{80,90}}); // add unnamed

for (int i=0; i<rect.size(); ++i) rect[i].move(10,10);           // use rect
```

We explain the **new** operator in Chapter 17, and the implementation of **Vector_ref** is presented in Appendix E. For now, it is sufficient to know that we can use it to hold unnamed objects. Operator **new** is followed by the name of a type (here, **Rectangle**) optionally followed by an initializer list (here, **{Point{50,60},Point{80,90}}**). Experienced programmers will be relieved to hear that we did not introduce a memory leak in this example.

Given **Rectangle** and **Vector_ref**, we can play with colors. For example, we can draw a simple color chart of the 256 colors shown above:

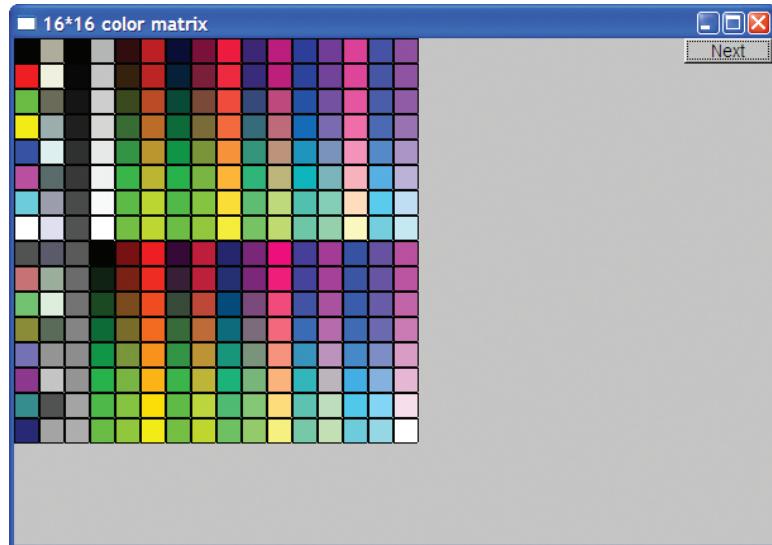
```

Vector_ref<Rectangle> vr;

for (int i = 0; i<16; ++i)
    for (int j = 0; j<16; ++j) {
        vr.push_back(new Rectangle{Point{i*20,j*20},20,20});
        vr[vr.size()-1].set_fill_color(Color{i*16+j});
        win20.attach(vr[vr.size()-1]);
    }
}

```

We make a **Vector_ref** of 256 **Rectangles**, organized graphically in the **Window** as a 16-by-16 matrix. We give the **Rectangles** the colors 0, 1, 2, 3, 4, and so on. After each **Rectangle** is created, we attach it to the window, so that it will be displayed:



13.11 Text

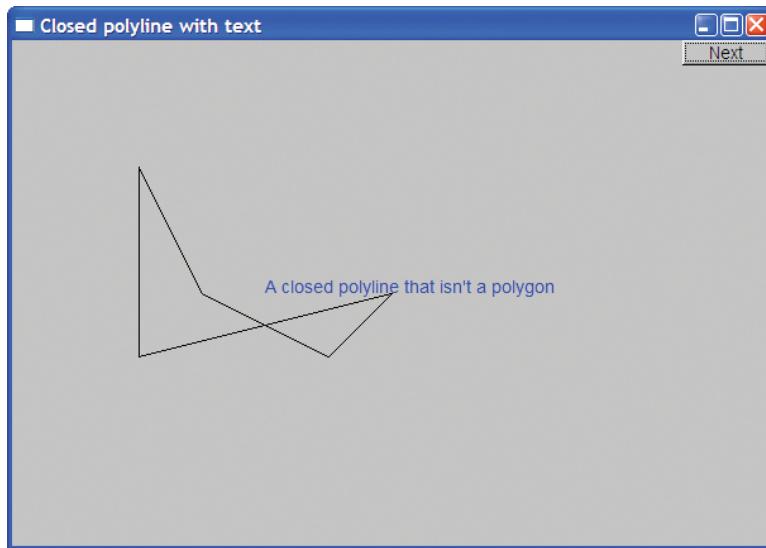
Obviously, we want to be able to add text to our displays. For example, we might want to label our “odd” **Closed_polyline** from §13.8:

```

Text t {Point{200,200}, "A closed polyline that isn't a polygon"};
t.set_color(Color::blue);

```

We get



Basically, a **Text** object defines a line of text starting at a **Point**. The **Point** will be the bottom left corner of the text. The reason for restricting the string to be a single line is to ensure portability across systems. Don't try to put in a newline character; it may or may not be represented as a newline in your window. String streams (§11.4) are useful for composing **strings** for display in **Text** objects (examples in §12.7.7 and §12.7.8). **Text** is defined like this:

```
struct Text : Shape {
    // the point is the bottom left of the first letter
    Text(Point x, const string& s)
        : lab{s}
        { add(x); }

    void draw_lines() const;

    void set_label(const string& s) { lab = s; }
    string label() const { return lab; }

    void set_font(Font f) { fnt = f; }
    Font font() const { return fnt; }

    void set_font_size(int s) { fnt_sz = s; }
    int font_size() const { return fnt_sz; }
```

```
private:
    string lab;      // label
    Font fnt {fl_font()};
    int fnt_sz {(fnt_size()<14)?14:fnt_size()} ;
};
```

If you want the font character size to be less than 14 or larger than the FLTK default, you have to explicitly set it. This is an example of a test protecting a user from possible variations in the behavior of an underlying library. In this case, an update of FLTK changed its default in a way that broke existing programs by making the characters tiny, and we decided to prevent that problem.

We provide the initializers as member initializers, rather than as part of the constructors' initializer lists, because the initializers do not depend on constructor arguments.

Text has its own **draw_lines()** because only the **Text** class knows how its string is stored:

```
void Text::draw_lines() const
{
    fl_draw(lab.c_str(),point(0).x,point(0).y);
}
```

The color of the characters is determined exactly like the lines in shapes composed of lines (such as **Open_polyline** and **Circle**), so you can choose a color using **set_color()** and see what color is currently used by **color()**. The character size and font are handled analogously. There is a small number of predefined fonts:

```
class Font { // character font
public:
    enum Font_type {
        helvetica=FL_HELVETICA,
        helvetica_bold=FL_HELVETICA_BOLD,
        helvetica_italic=FL_HELVETICA_ITALIC,
        helvetica_bold_italic=FL_HELVETICA_BOLD_ITALIC,
        courier=FL_COURIER,
        courier_bold=FL_COURIER_BOLD,
        courier_italic=FL_COURIER_ITALIC,
        courier_bold_italic=FL_COURIER_BOLD_ITALIC,
        times=FL_TIMES,
        times_bold=FL_TIMES_BOLD,
        times_italic=FL_TIMES_ITALIC,
        times_bold_italic=FL_TIMES_BOLD_ITALIC,
        symbol=FL_SYMBOL,
```

```

screen=FL_SCREEN,
screen_bold=FL_SCREEN_BOLD,
zapf_dingbats=FL_ZAPF_DINGBATS
};

Font(Font_type ff) :f{ff} { }

Font(int ff) :f{ff} { }

int as_int() const { return f; }

private:
    int f;
};

```

The style of class definition used to define **Font** is the same as we used to define **Color** (§13.4) and **Line_style** (§13.5).

13.12 Circle

Just to show that the world isn't completely rectangular, we provide class **Circle** and class **Ellipse**. A **Circle** is defined by a center and a radius:

```

struct Circle : Shape {
    Circle(Point p, int rr);      // center and radius

    void draw_lines() const;

    Point center() const ;
    int radius() const { return r; }
    void set_radius(int rr)
    {
        set_point(0,Point{center().x-rr,center().y-rr}); // maintain
                                                       // the center
        r = rr;
    }
private:
    int r;
};

```

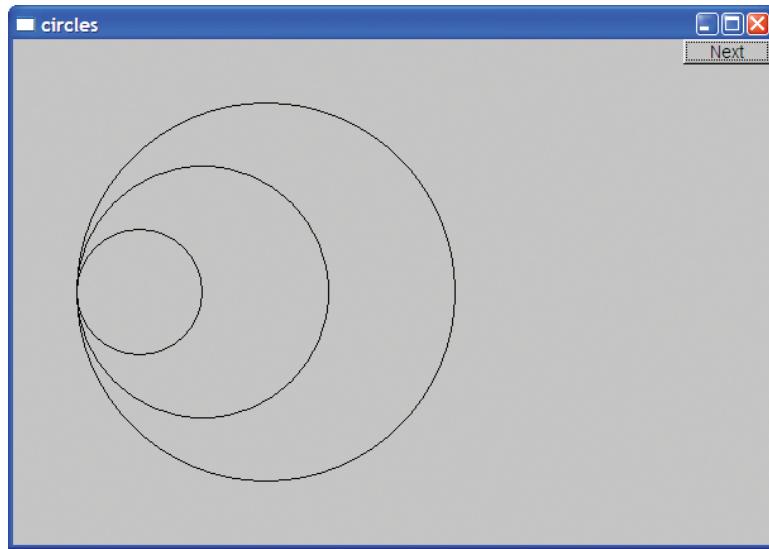
We can use **Circle** like this:

```

Circle c1 {Point{100,200},50};
Circle c2 {Point{150,200},100};
Circle c3 {Point{200,200},150};

```

This produces three circles of different sizes aligned with their centers in a horizontal line:



The main peculiarity of **Circle**'s implementation is that the point stored is not the center, but the top left corner of the square bounding the circle. We could have stored either but chose the one FLTK uses for its optimized circle-drawing routine. That way, **Circle** provides another example of how a class can be used to present a different (and supposedly nicer) view of a concept than its implementation:

```
Circle::Circle(Point p, int rr)      // center and radius
    :r{rr}
{
    add(Point{p.x-r,p.y-r});        // store top left corner
}

Point Circle::center() const
{
    return {point(0).x+r, point(0).y+r};
}

void Circle::draw_lines() const
{
    if (color().visibility())
        fl_arc(point(0).x,point(0).y,r+r,r+r,0,360);
}
```

Note the use of `fl_arc()` to draw the circle. The initial two arguments specify the top left corner, the next two arguments specify the width and the height of the smallest rectangle that encloses the circle, and the final two arguments specify the beginning and end angle to be drawn. A circle is drawn by going the full 360 degrees, but we can also use `fl_arc()` to draw parts of a circle (and parts of an ellipse); see exercise 1.

13.13 Ellipse

An ellipse is similar to `Circle` but is defined with both a major and a minor axis, instead of a radius; that is, to define an ellipse, we give the center's coordinates, the distance from the center to a point on the x axis, and the distance from the center to a point on the y axis:

```
struct Ellipse : Shape {
    Ellipse(Point p, int w, int h); // center, max and min distance from center

    void draw_lines() const;

    Point center() const;
    Point focus1() const;
    Point focus2() const;

    void set_major(int ww)
    {
        set_point(0,Point{center().x-ww,center().y-h}); // maintain
                                                       // the center
        w = ww;
    }
    int major() const { return w; }

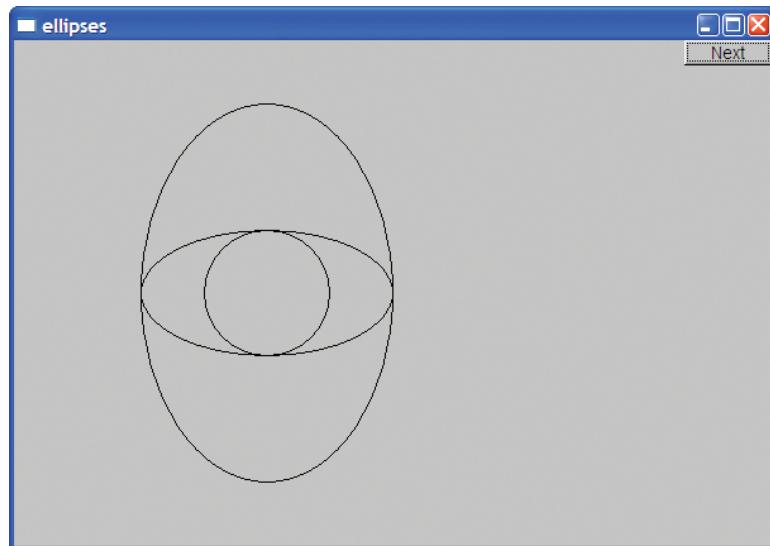
    void set_minor(int hh)
    {
        set_point(0,Point{center().x-w,center().y-hh}); // maintain
                                                       // the center
        h = hh;
    }
    int minor() const { return h; }

private:
    int w;
    int h;
};
```

We can use **Ellipse** like this:

```
Ellipse e1 {Point{200,200},50,50};  
Ellipse e2 {Point{200,200},100,50};  
Ellipse e3 {Point{200,200},100,150};
```

This gives us three ellipses with a common center but different-size axes:



Note that an **Ellipse** with **major()==minor()** looks exactly like a circle.

Another popular view of an ellipse specifies two foci plus a sum of distances from a point to the foci. Given an **Ellipse**, we can compute a focus. For example:

```
Point focus1() const  
{  
    if (h<=w) // foci are on the x axis:  
        return {center().x+int(sqrt(double(w*w-h*h))),center().y};  
    else // foci are on the y axis:  
        return {center().x,center().y+int(sqrt(double(h*h-w*w)))};  
}
```



Why is a **Circle** not an **Ellipse**? Geometrically, every circle is an ellipse, but not every ellipse is a circle. In particular, a circle is an ellipse where the two foci are equal. Imagine that we defined our **Circle** to be an **Ellipse**. We could do that at the cost of needing an extra value in its representation (a circle is defined by a point and a radius; an ellipse needs a center and a pair of axes). We don't like space overhead where we don't need it, but the primary reason for our **Circle** not being an **Ellipse** is that we couldn't define it so without somehow disabling **set_major()** and **set_minor()**. After all, it would not be a circle (as a mathematician would recognize it) if we could use **set_major()** to get **major() != minor()** – at least it would no longer be a circle after we had done that. We can't have an object that is of one type sometimes (i.e., when **major() != minor()**) and another type some other time (i.e., when **major() == minor()**). What we can have is an object (an **Ellipse**) that can look like a circle sometimes. A **Circle**, on the other hand, never morphs into an ellipse with two unequal axes.



When we design classes, we have to be careful not to be too clever and not to be deceived by our “intuition” into defining classes that don't make sense as classes in our code. Conversely, we have to take care that our classes represent some coherent concept and are not just a collection of data and function members.



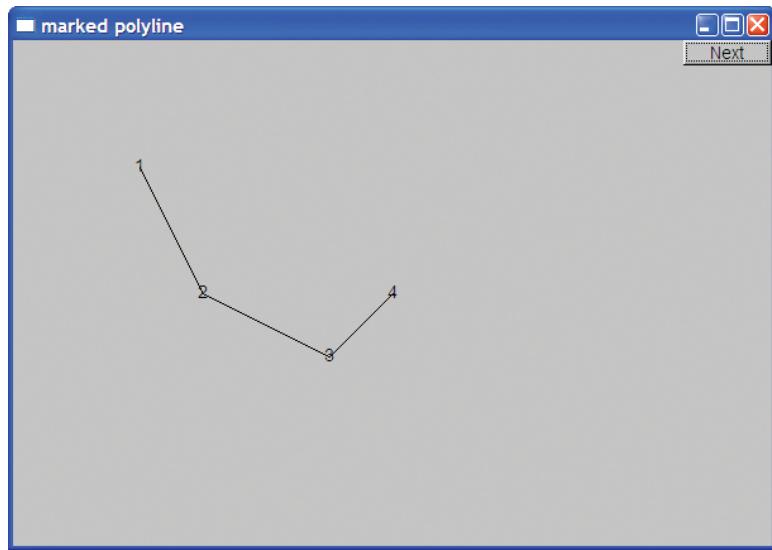
Just throwing code together without thinking about what ideas/concepts we are representing is “hacking” and leads to code that we can't explain and that others can't maintain. If you don't feel altruistic, remember that “others” might be you in a few months' time. Such code is also harder to debug.

13.14 Marked_polyline

We often want to “label” points on a graph. One way of displaying a graph is as an open polyline, so what we need is an open polyline with “marks” at the points. A **Marked_polyline** does that. For example:

```
Marked_polyline mpl {"1234"};
mpl.add(Point{100,100});
mpl.add(Point{150,200});
mpl.add(Point{250,250});
mpl.add(Point{300,200});
```

This produces



The definition of **Marked_polyline** is

```
struct Marked_polyline : Open_polyline {
    Marked_polyline(const string& m) : mark{m} { if (m=="") mark = "*"; }
    Marked_polyline(const string& m, initializer_list<Point> lst);
    void draw_lines() const;
private:
    string mark;
};
```

By deriving from **Open_polyline**, we get the handling of **Points** “for free”; all we have to do is to deal with the marks. In particular, **draw_lines()** becomes

```
void Marked_polyline::draw_lines() const
{
    Open_polyline::draw_lines();
    for (int i=0; i<number_of_points(); ++i)
        draw_mark(point(i),mark[i%mark.size()]);
}
```

The call `Open_polyline::draw_lines()` takes care of the lines, so we just have to deal with the “marks.” We supply the marks as a string of characters and use them in order: the `mark[i%mark.size()]` selects the character to be used next by cycling through the characters supplied when the `Marked_polyline` was created. The `%` is the modulo (remainder) operator. This `draw_lines()` uses a little helper function `draw_mark()` to actually output a letter at a given point:

```
void draw_mark(Point xy, char c)
{
    constexpr int dx = 4;
    constexpr int dy = 4;

    string m {1,c}; // string holding the single char c
    fl_draw(m.c_str(),xy.x-dx,xy.y+dy);
}
```

The `dx` and `dy` constants are used to center the letter over the point. The `string m` is constructed to contain the single character `c`.

The constructor that takes an initializer list simply forwards the list `Open_polyline`’s initializer-list constructor:

```
Marked_polyline(const string& m, initializer_list<Point> lst)
    :Open_polyline{lst},
    mark{m}
{
    if (m=="") mark = "*";
}
```

The test for the empty string is needed to avoid `draw_lines()` trying to access a character that isn’t there.

Given the constructor that takes an initializer list, we can abbreviate the example to

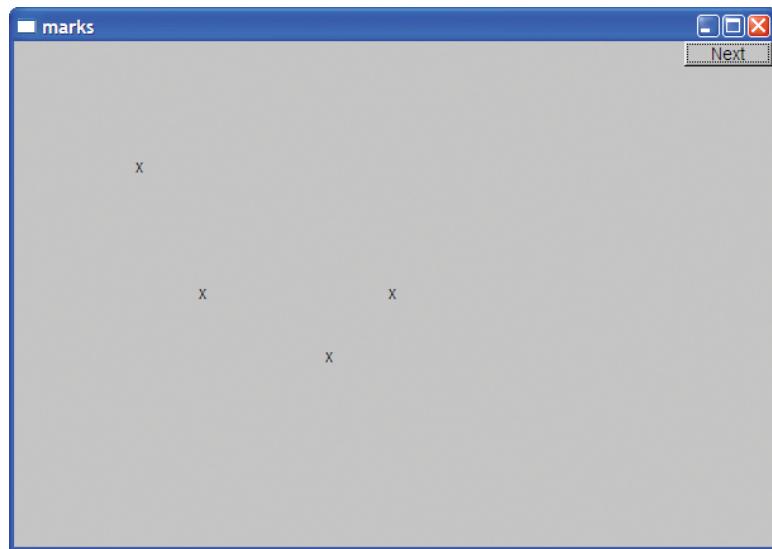
```
Marked_polyline mpl {"1234",{{100,100}, {150,200}, {250,250}, {300,200}}};
```

13.15 Marks

Sometimes, we want to display marks without lines connecting them. We provide the class `Marks` for that. For example, we can mark the four points we have used for our various examples without connecting them with lines:

```
Marks pp {"x",{{100,100}, {150,200}, {250,250}, {300,200}}};
```

This produces



One obvious use of **Marks** is to display data that represents discrete events so that drawing connecting lines would be inappropriate. An example would be (height, weight) data for a group of people.

A **Marks** is simply a **Marked_polyline** with the lines **invisible**:

```
struct Marks : Marked_polyline {
    Marks(const string& m)
        :Marked_polyline{m}
    {
        set_color(Color{Color::invisible});
    }

    Marked_polyline(const string& m, initializer_list<Point> lst)
        : Marked_polyline{m,lst}
    {
        set_color(Color{Color::invisible});
    }
};
```

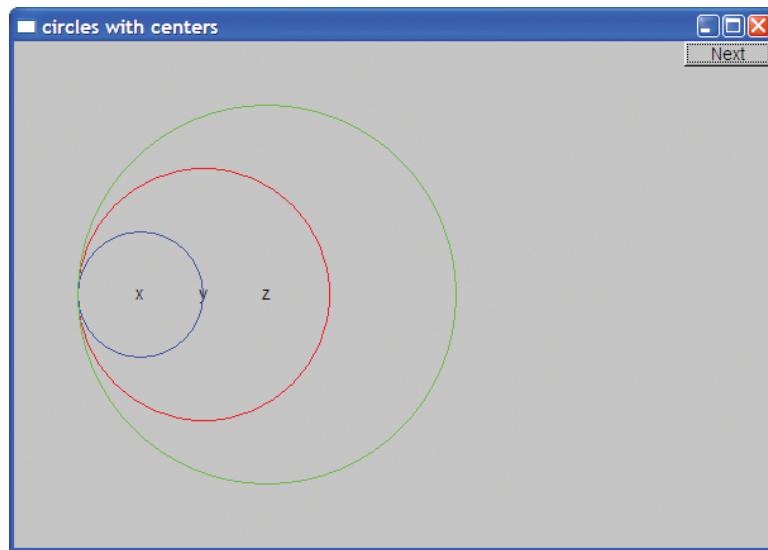
The **:Marked_polyline{m}** notation is used to initialize the **Marked_polyline** part of a **Marks** object. This notation is a variant of the syntax used to initialize members (§9.4.4).

13.16 Mark

A **Point** is simply a location in a **Window**. It is not something we draw or something we can see. If we want to mark a single **Point** so that we can see it, we can indicate it by a pair of lines as in §13.2 or by using **Marks**. That's a bit verbose, so we have a simple version of **Marks** that is initialized by a point and a character. For example, we could mark the centers of our circles from §13.12 like this:

```
Mark m1 {Point{100,200},'x'};  
Mark m2 {Point{150,200},'y'};  
Mark m3 {Point{200,200},'z'};  
c1.set_color(Color::blue);  
c2.set_color(Color::red);  
c3.set_color(Color::green);
```

This produces



A **Mark** is simply a **Marks** with its initial (and typically only) point given immediately:

```
struct Mark : Marks {  
    Mark(Point xy, char c) : Marks{string{1,c}}  
    {  
        add(xy);  
    }  
};
```

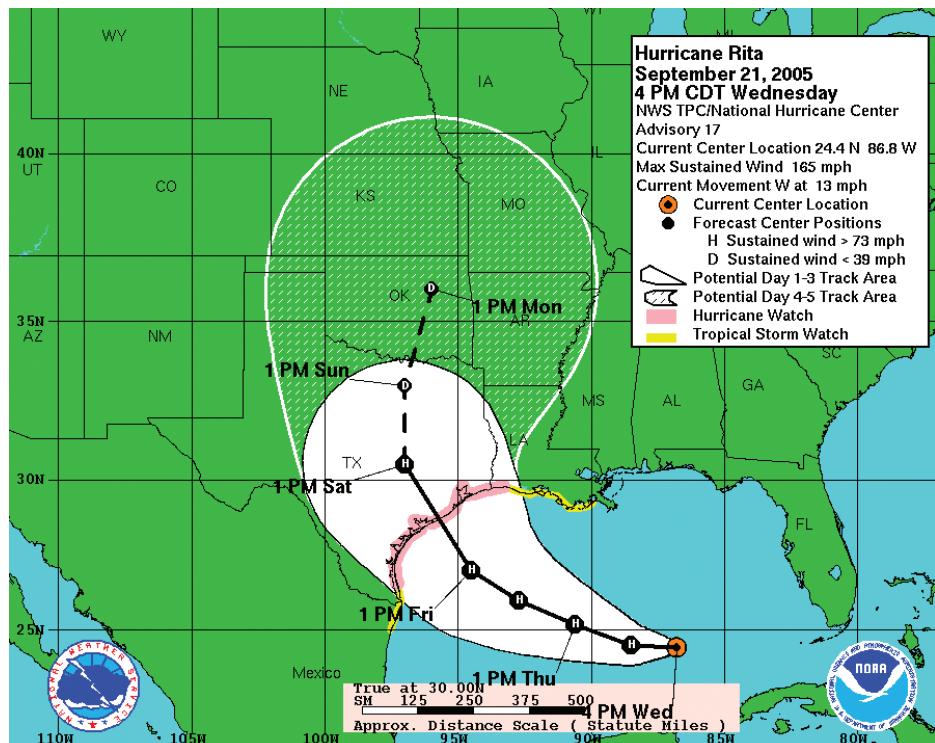
The `string{1,c}` is a constructor for `string`, initializing the `string` to contain the single character `c`.

All `Mark` provides is a convenient notation for creating a `Marks` object with a single point marked with a single character. Is `Mark` worth our effort to define it? Or is it just “spurious complication and confusion”? There is no clear, logical answer. We went back and forth on this question, but in the end decided that it was useful for users and the effort to define it was minimal.

Why use a character as a “mark”? We could have used any small shape, but characters provide a useful and simple set of marks. It is often useful to be able to use a variety of “marks” to distinguish different sets of points. Characters such as `x`, `o`, `+`, and `*` are pleasantly symmetric around a center.

13.17 Images

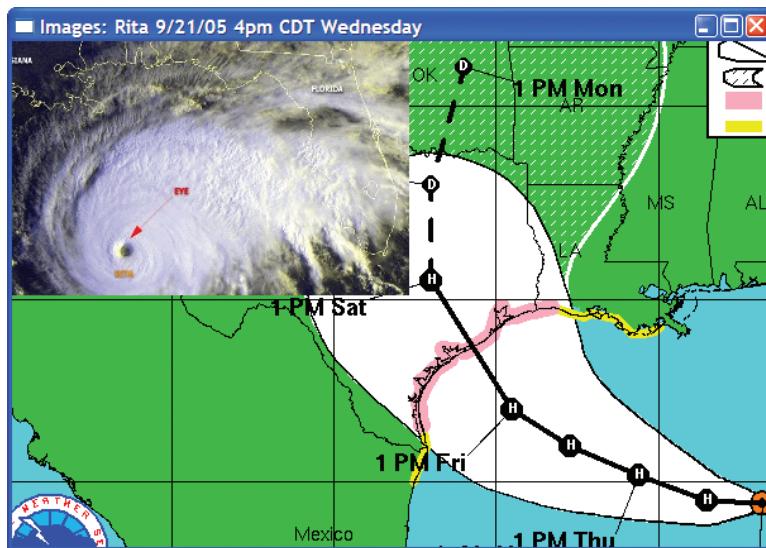
The average personal computer holds thousands of images in files and can access millions more over the web. Naturally, we want to display some of those images in even quite simple programs. For example, here is an image ([rita_path.gif](#)) of the projected path of Hurricane Rita as it approached the Texas Gulf Coast:



We can select part of that image and add a photo of Rita as seen from space ([rita.jpg](#)):

```
Image rita {Point{0,0}, "rita.jpg"};
Image path {Point{0,0}, "rita_path.gif"};
path.set_mask(Point{50,250},600,400); // select likely landfall

win.attach(path);
win.attach(rita);
```



The [set_mask\(\)](#) operation selects a sub-picture of an image to be displayed. Here, we selected a (600,400)-pixel image from [rita_path.gif](#) (loaded as [path](#)) with its top leftmost point at [path](#)'s point (50,250). Selecting only part of an image for display is so common that we chose to support it directly.

Shapes are laid down in the order they are attached, like pieces of paper on a desk, so we got [path](#) “on the bottom” simply by attaching it before [rita](#).

Images can be encoded in a bewildering variety of formats. Here we deal with only two of the most common, JPEG and GIF:

```
enum class Suffix { none, jpg, gif };
```

In our graphics interface library, we represent an image in memory as an object of class **Image**:

```
struct Image : Shape {
    Image(Point xy, string file_name, Suffix e = Suffix::none);
    ~Image() { delete p; }
    void draw_lines() const;
    void set_mask(Point xy, int ww, int hh)
        { w=ww; h=hh; cx=xy.x; cy=xy.y; }
private:
    int w,h; // define "masking box" within image relative to position (cx,cy)
    int cx,cy;
    Fl_Image* p;
    Text fn;
};
```

The **Image** constructor tries to open a file with the name given to it. Then it tries to create a picture using the encoding specified as an optional argument or (more often) as a file suffix. If the image cannot be displayed (e.g., because the file wasn't found), the **Image** displays a **Bad_image**. The definition of **Bad_image** looks like this:

```
struct Bad_image : Fl_Image {
    Bad_image(int h, int w) : Fl_Image{h,w,0} {}
    void draw(int x,int y, int, int, int, int) { draw_empty(x,y); }
};
```

The handling of images within a graphics library is quite complicated, but the main complexity of our graphics interface class **Image** is in the file handling in the constructor:

```
// somewhat overelaborate constructor
// because errors related to image files can be such a pain to debug
Image::Image(Point xy, string s, Suffix e)
    :w{0}, h{0}, fn{xy,""}
{
    add(xy);

    if (!can_open(s)) { // can we open s?
        fn.set_label("cannot open \\""+s+"");
    }
```

```

    p = new Bad_image(30,20);      // the "error image"
    return;
}

if (e == Suffix::none) e = get_encoding(s);

switch(e) {      // check if it is a known encoding
case Suffix::jpg:
    p = new Fl_JPEG_Image{s.c_str()};
    break;
case Suffix::gif:
    p = new Fl_GIF_Image{s.c_str()};
    break;
default:          // unsupported image encoding
    fn.set_label("unsupported file type '"+s+"'");
    p = new Bad_image(30,20);      // the "error image"
}
}

```

We use the suffix to pick the kind of object we create to hold the image (a **Fl_JPEG_Image** or a **Fl_GIF_Image**). We create that implementation object using **new** and assign it to a pointer. This is an implementation detail (see Chapter 17 for a discussion of operator **new** and pointers) related to the organization of FLTK and is of no fundamental importance here. FLTK uses C-style strings, so we have to use **s.c_str()** rather than plain **s**.

Now, we just have to implement **can_open()** to test if we can open a named file for reading:

```

bool can_open(const string& s)
    // check if a file named s exists and can be opened for reading
{
    ifstream ff(s);
    return ff;
}

```

Opening a file and then closing it again is a fairly clumsy way of portably separating errors related to “can’t open the file” from errors related to the format of the data in the file.

You can look up the **get_encoding()** function, if you like. It simply looks for a suffix and looks up that suffix in a table of known suffixes. That lookup table is a standard library **map** (see §21.6).



Drill

1. Make an 800-by-1000 **Simple_window**.
2. Put an 8-by-8 grid on the leftmost 800-by-800 part of that window (so that each square is 100 by 100).
3. Make the eight squares on the diagonal starting from the top left corner red (use **Rectangle**).
4. Find a 200-by-200-pixel image (JPEG or GIF) and place three copies of it on the grid (each image covering four squares). If you can't find an image that is exactly 200 by 200, use **set_mask()** to pick a 200-by-200 section of a larger image. Don't obscure the red squares.
5. Add a 100-by-100 image. Have it move around from square to square when you click the "Next" button. Just put **wait_for_button()** in a loop with some code that picks a new square for your image.

Review

1. Why don't we "just" use a commercial or open-source graphics library directly?
2. About how many classes from our graphics interface library do you need to do simple graphics output?
3. What are the header files needed to use the graphics interface library?
4. What classes define closed shapes?
5. Why don't we just use **Line** for every shape?
6. What do the arguments to **Point** indicate?
7. What are the components of **Line_style**?
8. What are the components of **Color**?
9. What is RGB?
10. What are the differences between two **Lines** and a **Lines** containing two lines?
11. What properties can you set for every **Shape**?
12. How many sides does a **Closed_polyline** defined by five **Points** have?
13. What do you see if you define a **Shape** but don't attach it to a **Window**?
14. How does a **Rectangle** differ from a **Polygon** with four **Points** (corners)?
15. How does a **Polygon** differ from a **Closed_polyline**?
16. What's on top: fill or outline?
17. Why didn't we bother defining a **Triangle** class (after all, we did define **Rectangle**)?
18. How do you move a **Shape** to another place in a **Window**?
19. How do you label a **Shape** with a line of text?

20. What properties can you set for a text string in a **Text**?
21. What is a font and why do we care?
22. What is **Vector_ref** for and how do we use it?
23. What is the difference between a **Circle** and an **Ellipse**?
24. What happens if you try to display an **Image** given a file name that doesn't refer to a file containing an image?
25. How do you display part of an image?

Terms

closed shape	image	point
color	image encoding	polygon
ellipse	invisible	polyline
fill	JPEG	unnamed object
font	line	Vector_ref
font size	line style	visible
GIF	open shape	

Exercises

For each “define a class” exercise, display a couple of objects of the class to demonstrate that they work.

1. Define a class **Arc**, which draws a part of an ellipse. Hint: **fl_arc()**.
2. Draw a box with rounded corners. Define a class **Box**, consisting of four lines and four arcs.
3. Define a class **Arrow**, which draws a line with an arrowhead.
4. Define functions **n()**, **s()**, **e()**, **w()**, **center()**, **ne()**, **se()**, **sw()**, and **nw()**. Each takes a **Rectangle** argument and returns a **Point**. These functions define “connection points” on and in the rectangle. For example, **nw(r)** is the northwest (top left) corner of a **Rectangle** called **r**.
5. Define the functions from exercise 4 for a **Circle** and an **Ellipse**. Place the connection points on or outside the shape but not outside the bounding rectangle.
6. Write a program that draws a class diagram like the one in §12.6. It will simplify matters if you start by defining a **Box** class that is a rectangle with a text label.
7. Make an RGB color chart (e.g., search the web for “RGB color chart”).
8. Define a class **Regular_hexagon** (a regular hexagon is a six-sided polygon with all sides of equal length). Use the center and the distance from the center to a corner point as constructor arguments.
9. Tile a part of a window with **Regular_hexagons** (use at least eight hexagons).

10. Define a class **Regular_polygon**. Use the center, the number of sides (>2), and the distance from the center to a corner as constructor arguments.
11. Draw a 300-by-200-pixel ellipse. Draw a 400-pixel-long x axis and a 300-pixel-long y axis through the center of the ellipse. Mark the foci. Mark a point on the ellipse that is not on one of the axes. Draw the two lines from the foci to the point.
12. Draw a circle. Move a mark around on the circle (let it move a bit each time you hit the “Next” button).
13. Draw the color matrix from §13.10, but without lines around each color.
14. Define a right triangle class. Make an octagonal shape out of eight right triangles of different colors.
15. “Tile” a window with small right triangles.
16. Do the previous exercise, but with hexagons.
17. Do the previous exercise, but using hexagons of a few different colors.
18. Define a class **Poly** that represents a polygon but checks that its points really do make a polygon in its constructor. Hint: You’ll have to supply the points to the constructor.
19. Define a class **Star**. One parameter should be the number of points. Draw a few stars with differing numbers of points, differing line colors, and differing fill colors.

Postscript

Chapter 12 showed how to be a user of classes. This chapter moves us one level up the “food chain” of programmers: here we become tool builders in addition to being tool users.



Graphics Class Design

“Functional, durable, beautiful.”

—Vitruvius

The purpose of the graphics chapters is dual: we want to provide useful tools for displaying information, but we also use the family of graphical interface classes to illustrate general design and implementation techniques. In particular, this chapter presents some ideas of interface design and the notion of inheritance. Along the way, we have to take a slight detour to examine the language features that most directly support object-oriented programming: class derivation, virtual functions, and access control. We don’t believe that design can be discussed in isolation from use and implementation, so our discussion of design is rather concrete. Maybe you’d better think of this chapter as “Graphics Class Design and Implementation.”

14.1 Design principles

- 14.1.1 Types
- 14.1.2 Operations
- 14.1.3 Naming
- 14.1.4 Mutability

14.2 Shape

- 14.2.1 An abstract class
- 14.2.2 Access control
- 14.2.3 Drawing shapes
- 14.2.4 Copying and mutability

14.3 Base and derived classes

- 14.3.1 Object layout
 - 14.3.2 Deriving classes and defining virtual functions
 - 14.3.3 Overriding
 - 14.3.4 Access
 - 14.3.5 Pure virtual functions
- 14.4 Benefits of object-oriented programming**

14.1 Design principles

What are the design principles for our graphics interface classes? First: What kind of question is that? What are “design principles” and why do we need to look at those instead of getting on with the serious business of producing neat pictures?

14.1.1 Types

Graphics is an example of an application domain. So, what we are looking at here is an example of how to present a set of fundamental application concepts and facilities to programmers (like us). If the concepts are presented confusingly, inconsistently, incompletely, or in other ways poorly represented in our code, the difficulty of producing graphical output is increased. We want our graphics classes to minimize the effort of a programmer trying to learn and to use them.

Our ideal of program design is to represent the concepts of the application domain directly in code. That way, if you understand the application domain, you understand the code and vice versa. For example:

- **Window** – a window as presented by the operating system
- **Line** – a line as you see it on the screen
- **Point** – a coordinate point
- **Color** – as you see it on the screen
- **Shape** – what’s common for all shapes in our graphics/GUI view of the world

The last example, **Shape**, is different from the rest in that it is a generalization, a purely abstract notion. We never see just a shape on the screen; we see a particular shape, such as a line or a hexagon. You’ll find that reflected in the definition of our types: try to make a **Shape** variable and the compiler will stop you.

The set of our graphics interface classes is a library; the classes are meant to be used together and in combination. They are meant to be used as examples to

follow when you define classes to represent other graphical shapes and as building blocks for such classes. We are not just defining a set of unrelated classes, so we can't make design decisions for each class in isolation. Together, our classes present a view of how to do graphics. We must ensure that this view is reasonably elegant and coherent. Given the size of our library and the enormity of the domain of graphical applications, we cannot hope for completeness. Instead, we aim for simplicity and extensibility.

In fact, no class library directly models all aspects of its application domain. That's not only impossible; it is also pointless. Consider writing a library for displaying geographical information. Do you want to show vegetation? National, state, and other political boundaries? Road systems? Railroads? Rivers? Highlight social and economic data? Seasonal variations in temperature and humidity? Wind patterns in the atmosphere above? Airline routes? Mark the locations of schools? The locations of fast-food "restaurants"? Local beauty spots? "All of that!" may be a good answer for a comprehensive geographical application, but it is not an answer for a single display. It may be an answer for a library supporting such geographical applications, but it is unlikely that such a library could also cover other graphical applications such as freehand drawing, editing photographic images, scientific visualization, and aircraft control displays.

So, as ever, we have to decide what's important to us. In this case, we have to decide which kind of graphics/GUI we want to do well. Trying to do everything is a recipe for failure. A good library directly and cleanly models its application domain from a particular perspective, emphasizes some aspects of the application, and deemphasizes others.

The classes we provide here are designed for simple graphics and simple graphical user interfaces. They are primarily aimed at users who need to present data and graphical output from numeric/scientific/engineering applications. You can build your own classes "on top of" ours. If that is not enough, we expose sufficient FLTK details in our implementation for you to get an idea of how to use that (or a similar "full-blown" graphics/GUI library) directly, should you so desire. However, if you decide to go that route, wait until you have absorbed Chapters 17 and 18. Those chapters contain information about pointers and memory management that you need for successful direct use of most graphics/GUI libraries.

One key decision is to provide a lot of "little" classes with few operations. For example, we provide **Open_polyline**, **Closed_polyline**, **Polygon**, **Rectangle**, **Marked_polyline**, **Marks**, and **Mark** where we could have provided a single class (possibly called "polyline") with a lot of arguments and operations that allowed us to specify which kind of polyline an object was and possibly even mutate a polyline from one kind to another. The extreme of this kind of thinking would be to provide every kind of shape as part of a single class **Shape**. We think that using many small classes most closely and most usefully models our domain of graphics. A single class providing "everything" would leave the user messaging

with data and options without a framework to help understanding, debugging, and performance.

14.1.2 Operations

We provide a minimum of operations as part of each class. Our ideal is the minimal interface that allows us to do what we want. Where we want greater convenience, we can always provide it in the form of added nonmember functions or yet another class.

We want the interfaces of our classes to show a common style. For example, all functions performing similar operations in different classes have the same name, take arguments of the same types, and where possible require those arguments in the same order. Consider the constructors: if a shape requires a location, it takes a **Point** as its first argument:

```
Line ln {Point{100,200},Point{300,400}};
Mark m {Point{100,200},'x'};      // display a single point as an 'x'
Circle c {Point{200,200},250};
```

All functions that deal with points use class **Point** to represent them. That would seem obvious, but many libraries exhibit a mixture of styles. For example, imagine a function for drawing a line. We could use one of two styles:

```
void draw_line(Point p1, Point p2);           // from p1 to p2 (our style)
void draw_line(int x1, int y1, int x2, int y2); // from (x1,y1) to (x2,y2)
```

We could even allow both, but for consistency, improved type checking, and improved readability we use the first style exclusively. Using **Point** consistently also saves us from confusion between coordinate pairs and the other common pair of integers: width and height. For example, consider:

```
draw_rectangle(Point{100,200}, 300, 400);    // our style
draw_rectangle(100,200,300,400);               // alternative
```

The first call draws a rectangle with a point, width, and height. That's reasonably easy to guess, but how about the second call? Is that a rectangle defined by points (100,200) and (300,400)? A rectangle defined by a point (100,200), a width 300, and a height 400? Something completely different (though plausible to someone)? Using the **Point** type consistently avoids such confusion.

Incidentally, if a function requires a width and a height, they are always presented in that order (just as we always give an *x* coordinate before a *y* coordinate). Getting such little details consistent makes a surprisingly large difference to the ease of use and the avoidance of run-time errors.

Logically identical operations have the same name. For example, every function that adds points, lines, etc. to any kind of shape is called `add()`, and any function that draws lines is called `draw_lines()`. Such uniformity helps us remember (by offering fewer details to remember) and helps us when we design new classes (“just do the usual”). Sometimes, it even allows us to write code that works for many different types, because the operations on those types have an identical pattern. Such code is called *generic*; see Chapters 19–21.

14.1.3 Naming

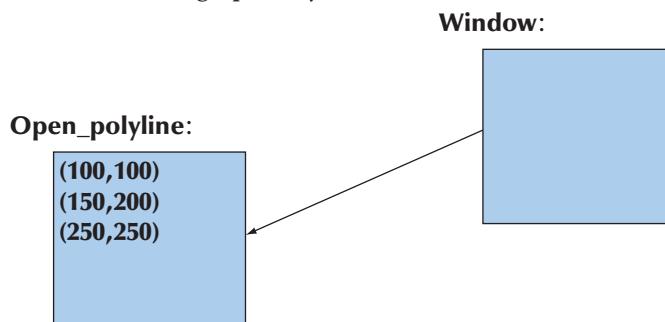
Logically different operations have different names. Again, that would seem obvious, but consider: why do we “attach” a `Shape` to a `Window`, but “add” a `Line` to a `Shape`? In both cases, we “put something into something,” so shouldn’t that similarity be reflected by a common name? No. The similarity hides a fundamental difference. Consider:

```
Open_polyline opl;
opl.add(Point{100,100});
opl.add(Point{150,200});
opl.add(Point{250,250});
```

Here, we copy three points into `opl`. The shape `opl` does not care about “our” points after a call to `add()`; it keeps its own copies. In fact, we rarely keep copies of the points – we leave that to the shape. On the other hand, consider:

```
win.attach(opl);
```

Here, we create a connection between the window `win` and our shape `opl`; `win` does not make a copy of `opl` – it keeps a reference to `opl`. So, it is our responsibility to keep `opl` valid as long as `win` uses it. That is, we must not exit `opl`’s scope while `win` is using `opl`. We can update `opl` and the next time `win` comes to draw `opl` our changes will appear on the screen. We can illustrate the difference between `attach()` and `add()` graphically:



Basically, `add()` uses pass-by-value (copies) and `attach()` uses pass-by-reference (shares a single object). We could have chosen to copy graphical objects into **Windows**. However, that would have given a different programming model, which we would have indicated by using `add()` rather than `attach()`. As it is, we just “attach” a graphics object to a **Window**. That has important implications. For example, we can’t create an object, attach it, allow the object to be destroyed, and expect the resulting program to work:

```
void f(Simple_window& w)
{
    Rectangle r {Point{100,200},50,30};
    w.attach(r);
} // oops, the lifetime of r ends here

int main()
{
    Simple_window win {Point{100,100},600,400,"My window"};
    // ...
    f(win);      // asking for trouble
    // ...
    win.wait_for_button();
}
```

By the time we have exited from `f()` and reached `wait_for_button()`, there is no `r` for the `win` to refer to and display. In Chapter 17, we’ll show how to create objects within a function and have them survive after the return from the function. Until then, we must avoid attaching objects that don’t survive until the call of `wait_for_button()`. We have **Vector_ref** (§13.10, §E.4) to help with that.

Note that had we declared `f()` to take its **Window** as a **const** reference argument (as recommended in §8.5.6), the compiler would have prevented our mistake: we can’t `attach(r)` to a **const Window** because `attach()` needs to make a change to the **Window** to record the **Window**’s interest in `r`.

14.1.4 Mutability

When we design a class, “Who can modify the data (representation)?” and “How?” are key questions that we must answer. We try to ensure that modification to the state of an object is done only by its own class. The **public/private** distinction is key to this, but we’ll show examples where a more flexible/subtle mechanism (**protected**) is employed. This implies that we can’t just give a class a data member, say a **string** called **label**; we must also consider if it should be possible to modify it after construction, and if so, how. We must also decide if

code other than our class's member functions needs to read the value of **label**, and if so, how. For example:

```
struct Circle {  
    // ...  
private:  
    int r;    // radius  
};  
  
Circle c{Point{100,200},50};  
c.r = -9;      // OK? No — compile-time error: Circle::r is private
```

As you might have noticed in Chapter 13, we decided to prevent direct access to most data members. Not exposing the data directly gives us the opportunity to check against “silly” values, such as a **Circle** with a negative radius. For simplicity of implementation, we take only limited advantage of this opportunity, so do be careful with your values. The decision not to consistently and completely check reflects a desire to keep the code short for presentation and the knowledge that if a user (you, us) supplies “silly” values, the result is simply a messed-up image on the screen and not corruption of precious data.

We treat the screen (seen as a set of **Windows**) purely as an output device. We can display new objects and remove old ones, but we never ask “the system” for information that we don't (or couldn't) know ourselves from the data structures we have built up representing our images.

14.2 Shape

Class **Shape** represents the general notion of something that can appear in a **Window** on a screen:

- It is the notion that ties our graphical objects to our **Window** abstraction, which in turn provides the connection to the operating system and the physical screen.
- It is the class that deals with color and the style used to draw lines. To do that it holds a **Line_style**, a **Color** for lines, and a **Color** for fill.
- It can hold a sequence of **Points** and has a basic notion of how to draw them.

Experienced designers will recognize that a class doing three things probably has problems with generality. However, here, we need something far simpler than the most general solution.

We'll first present the complete class and then discuss its details:

```

class Shape {          // deals with color and style and holds sequence of lines
public:
    void draw() const;           // deal with color and draw lines
    virtual void move(int dx, int dy); // move the shape +=dx and +=dy

    void set_color(Color col);
    Color color() const;

    void set_style(Line_style sty);
    Line_style style() const;

    void set_fill_color(Color col);
    Color fill_color() const;

    Point point(int i) const;      // read-only access to points
    int number_of_points() const;

    Shape(const Shape&) = delete; // prevent copying
    Shape& operator=(const Shape&) = delete;

    virtual ~Shape() {}
protected:
    Shape() {}
    Shape(initializer_list<Point> lst); // add() the Points to this Shape

    virtual void draw_lines() const; // draw the appropriate lines
    void add(Point p);             // add p to points
    void set_point(int i, Point p); // points[i]=p;
private:
    vector<Point> points;        // not used by all shapes
    Color lcolor {fl_color();} // color for lines and characters (with default)
    Line_style ls {0};
    Color fcolor {Color::invisible}; // fill color
};

```

This is a relatively complex class designed to support a wide variety of graphics classes and to represent the general concept of a shape on the screen. However, it still has only four data members and 15 functions. Furthermore, those functions are all close to trivial so that we can concentrate on design issues. For the rest of this section we will go through the members one by one and explain their role in the design.

14.2.1 An abstract class

Consider first `Shape`'s constructors:

protected:

```
Shape() {}
Shape(initializer_list<Point> lst); // add() the Points to this Shape
```

The constructors are **protected**. That means that they can only be used directly from classes derived from `Shape` (using the `:Shape` notation). In other words, `Shape` can only be used as a base for classes, such as `Line` and `Open_polyline`. The purpose of that **protected**: is to ensure that we don't make `Shape` objects directly. For example:

```
Shape ss; // error: cannot construct Shape
```

`Shape` is designed to be a base class only. In this case, nothing particularly nasty would happen if we allowed people to create `Shape` objects directly, but by limiting use, we keep open the possibility of modifications to `Shape` that would render it unsuitable for direct use. Also, by prohibiting the direct creation of `Shape` objects, we directly model the idea that we cannot have/see a general shape, only particular shapes, such as `Circle` and `Closed_polyline`. Think about it! What does a shape look like? The only reasonable response is the counter question “What shape?” The notion of a shape that we represent by `Shape` is an abstract concept. That's an important and frequently useful design notion, so we don't want to compromise it in our program. Allowing users to directly create `Shape` objects would do violence to our ideal of classes as direct representations of concepts.

The default constructor sets the members to their default values. Here again, the underlying library used for implementation, FLTK, “shines through.” However, FLTK's notions of color and style are not mentioned directly by the uses. They are only part of the implementation of our `Shape`, `Color`, and `Line_style` classes. The `vector<Points>` defaults to an empty vector.

The initializer-list constructor also uses the default initializers, and then `add()`s the elements of its argument list to the `Shape`:

```
Shape::Shape(initializer_list<Point> lst)
{
    for (Point p : list) add(p);
}
```

A class is *abstract* if it can be used only as a base class. The other – more common – way of achieving that is called a *pure virtual function*; see §14.3.5. A class that can be used to create objects – that is, the opposite of an abstract class – is called

a *concrete* class. Note that *abstract* and *concrete* are simply technical words for an everyday distinction. We might go to the store to buy a camera. However, we can't just ask for a camera and take it home. What brand of camera? Which particular model camera? The word *camera* is a generalization; it refers to an abstract notion. An Olympus E-M5 refers to a specific kind of camera, which we (in exchange for a large amount of cash) might acquire a particular instance of: a particular camera with a unique serial number. So, “camera” is much like an abstract (base) class; “Olympus E-M5” is much like a concrete (derived) class, and the actual camera in my hand (if I bought it) would be much like an object.

The declaration

```
virtual ~Shape() {}
```

defines a virtual destructor. We won't use that for now, so we leave the explanation to §17.5.2, where we show a use.

14.2.2 Access control

Class **Shape** declares all data members **private**:

```
private:
    vector<Point> points;
    Color lcolor {fl_color();      // color for lines and characters (with default)
    Line_style ls {0};
    Color fcolor {Color::invisible};    // fill color
```

The initializers for the data members don't depend on constructor arguments, so I specified them in the data member declarations. As ever, the default value for a vector is “empty” so I didn't have to be explicit about that. The constructor will apply those default values.

Since the data members of **Shape** are declared **private**, we need to provide access functions. There are several possible styles for doing this. We chose one that we consider simple, convenient, and readable. If we have a member representing a property **X**, we provide a pair of functions **X()** and **set_X()** for reading and writing, respectively. For example:

```
void Shape::set_color(Color col)
{
    lcolor = col;

}

Color Shape::color() const
```

```

{
    return lcolor;
}

```

The main inconvenience of this style is that you can't give the member variable the same name as its readout function. As ever, we chose the most convenient names for the functions because they are part of the public interface. It matters far less what we call our **private** variables. Note the way we use **const** to indicate that the readout functions do not modify their **Shape** (§9.7.4).

Shape keeps a vector of **Points**, called **points**, that a **Shape** maintains in support of its derived classes. We provide the function **add()** for adding **Points** to **points**:

```

void Shape::add(Point p)      // protected
{
    points.push_back(p);
}

```

Naturally, **points** starts out empty. We decided to provide **Shape** with a complete functional interface rather than giving users – even member functions of classes derived from **Shape** – direct access to data members. To some, providing a functional interface is a no-brainer, because they feel that making any data member of a class **public** is bad design. To others, our design seems overly restrictive because we don't allow direct write access to all members of derived classes.

A shape derived from **Shape**, such as **Circle** and **Polygon**, knows what its points mean. The base class **Shape** does not “understand” the points; it only stores them. Therefore, the derived classes need control over how points are added. For example:

- **Circle** and **Rectangle** do not allow a user to add points; that just wouldn't make sense. What would be a rectangle with an extra point? (§12.7.6)
- **Lines** allows only pairs of points to be added (and not an individual point; §13.3).
- **Open_polyline** and **Marks** allow any number of points to be added.
- **Polygon** allows a point to be added only by an **add()** that checks for intersections (§13.8).

We made **add()** **protected** (that is, accessible from a derived class only) to ensure that derived classes take control over how points are added. Had **add()** been **public** (everybody can add points) or **private** (only **Shape** can add points), this close match of functionality to our idea of shapes would not have been possible.



Similarly, we made `set_point()` **protected**. In general, only a derived class can know what a point means and whether it can be changed without violating an invariant. For example, if we have a `Regular_hexagon` class defined as a set of six points, changing just a single point would make the resulting figure “not a regular hexagon.” On the other hand, if we changed one of the points of a rectangle, the result would still be a rectangle. In fact, we didn’t find a need for `set_point()` in our example classes and code, so `set_point()` is provided just to ensure that the rule that we can read and set every attribute of a `Shape` holds. For example, if we wanted a `Mutable_rectangle`, we could derive it from `Rectangle` and provide operations to change the points.

We made the vector of `Points`, `points`, **private** to protect it against undesired modification. To make it useful, we also need to provide access to it:

```
void Shape::set_point(int i, Point p)    // not used; not necessary so far
{
    points[i] = p;
}

Point Shape::point(int i) const
{
    return points[i];
}

int Shape::number_of_points() const
{
    return points.size();
}
```

In derived class member functions, these functions are used like this:

```
void Lines::draw_lines() const
    // draw lines connecting pairs of points
{
    for (int i=1; i<number_of_points(); i+=2)
        fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

 You might worry about all those trivial access functions. Are they not inefficient? Do they slow down the program? Do they increase the size of the generated code?

No, they will all be compiled away (“inlined”) by the compiler. Calling `number_of_points()` will take up exactly as many bytes of memory and execute exactly as many instructions as calling `points.size()` directly.

These access control considerations and decisions are important. We could have provided this close-to-minimal version of `Shape`:

```
struct Shape {          // close-to-minimal definition — too simple — not used
    Shape();
    Shape(initializer_list<Point>);
    void draw() const;           // deal with color and call draw_lines
    virtual void draw_lines() const; // draw the appropriate lines
    virtual void move(int dx, int dy); // move the shape +=dx and +=dy
    virtual ~Shape();

    vector<Point> points;      // not used by all shapes
    Color lcolor;
    Line_style ls;
    Color fcolor;
};
```

What value did we add by those extra 12 member functions and two lines of access specifications (`private:` and `protected:`)? The basic answer is that protecting the representation ensures that it doesn’t change in ways unanticipated by a class designer so that we can write better classes with less effort. This is the argument about “invariants” (§9.4.3). Here, we’ll point out such advantages as we define classes derived from `Shape`. One simple example is that earlier versions of `Shape` used

```
Fl_Color lcolor;
int line_style;
```

This turned out to be too limiting (an `int` line style doesn’t elegantly support line width, and `Fl_Color` doesn’t accommodate `invisible`) and led to some messy code. Had these two variables been `public` and used in a user’s code, we could have improved our interface library only at the cost of breaking that code (because it mentioned the names `lcolor` and `line_style`).

In addition, the access functions often provide notational convenience. For example, `s.add(p)` is easier to read and write than `s.points.push_back(p)`.

14.2.3 Drawing shapes

We have now described almost all but the real heart of class **Shape**:

```
void draw() const; // deal with color and call draw_lines()
virtual void draw_lines() const; // draw the lines appropriately
```

Shape's most basic job is to draw shapes. We could remove all other functionality from **Shape** or leave it with no data of its own without doing major conceptual harm (see §14.4), but drawing is **Shape**'s essential business. It does so using FLTK and the operating system's basic machinery, but from a user's point of view, it provides just two functions:

- **draw()** applies style and color and then calls **draw_lines()**.
- **draw_lines()** puts pixels on the screen.

The **draw()** function doesn't use any novel techniques. It simply calls FLTK functions to set the color and style to what is specified in the **Shape**, calls **draw_lines()** to do the actual drawing on the screen, and then tries to restore color and style to what they were before the call:

```
void Shape::draw() const
{
    Fl_Color oldc = fl_color();
    // there is no good portable way of retrieving the current style
    fl_color(lcolor.as_int()); // set color
    fl_line_style(ls.style(),ls.width()); // set style
    draw_lines();
    fl_color(oldc); // reset color (to previous)
    fl_line_style(0); // reset line style to default
}
```

Unfortunately, FLTK doesn't provide a way of obtaining the current style, so the style is just set to a default. That's the kind of compromise we sometimes have to accept as the cost of simplicity and portability. We didn't think it worthwhile to try to implement that facility in our interface library.

Note that **Shape::draw()** doesn't handle fill color or the visibility of lines. Those are handled by the individual **draw_lines()** functions that have a better idea of how to interpret them. In principle, all color and style handling could be delegated to the individual **draw_lines()** functions, but that would be quite repetitive.

Now consider how we might handle **draw_lines()**. If you think about it for a bit, you'll realize that it would be hard for a **Shape** function to draw all that

needs to be drawn for every kind of shape. To do so would require that every last pixel of each shape should somehow be stored in the **Shape** object. If we kept the **vector<Point>** model, we'd have to store an awful lot of points. Worse, "the screen" (that is, the graphics hardware) already does that – and does it better.

To avoid that extra work and extra storage, **Shape** takes another approach: it gives each **Shape** (that is, each class derived from **Shape**) a chance to define what it means to draw it. A **Text**, **Rectangle**, or **Circle** class may have a clever way of drawing itself. In fact, most such classes do. After all, such classes "know" exactly what they are supposed to represent. For example, a **Circle** is defined by a point and a radius, rather than, say, a lot of line segments. Generating the required bits for a **Circle** from the point and radius if and when needed isn't really all that hard or expensive. So **Circle** defines its own **draw_lines()** which we want to call instead of **Shape**'s **draw_lines()**. That's what the **virtual** in the declaration of **Shape::draw_lines()** means:

```
struct Shape {
    // ...
    virtual void draw_lines() const; // let each derived class define its
                                    // own draw_lines() if it so chooses
    // ...
};

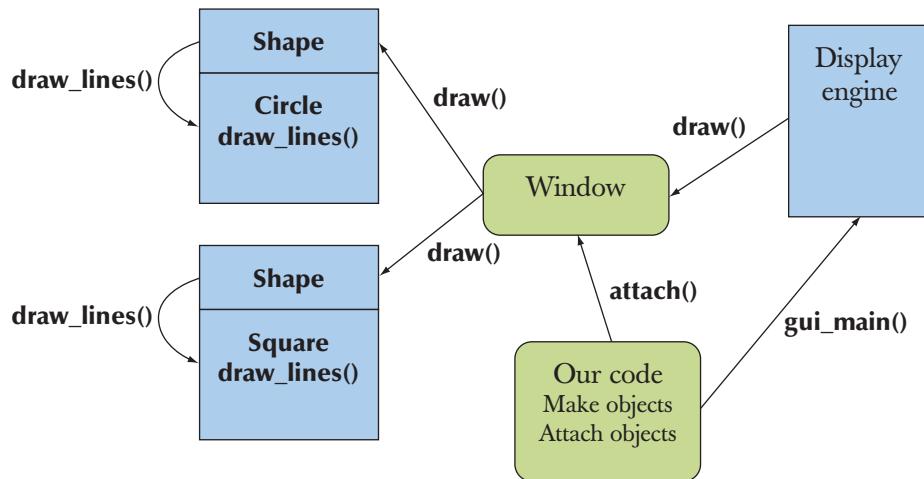
struct Circle : Shape {
    // ...
    void draw_lines() const; // "override" Shape::draw_lines()
    // ...
};
```

So, **Shape**'s **draw_lines()** must somehow invoke one of **Circle**'s functions if the **Shape** is a **Circle** and one of **Rectangle**'s functions if the **Shape** is a **Rectangle**. That's what the word **virtual** in the **draw_lines()** declaration ensures: if a class derived from **Shape** has defined its own **draw_lines()** (with the same type as **Shape**'s **draw_lines()**), that **draw_lines()** will be called rather than **Shape**'s **draw_lines()**. Chapter 13 shows how that's done for **Text**, **Circle**, **Closed_polyline**, etc. Defining a function in a derived class so that it can be used through the interfaces provided by a base is called *overriding*.

Note that despite its central role in **Shape**, **draw_lines()** is **protected**; it is not meant to be called by "the general user" – that's what **draw()** is for – but simply as an "implementation detail" used by **draw()** and the classes derived from **Shape**.

This completes our display model from §12.2. The system that drives the screen knows about **Window**. **Window** knows about **Shape** and can call **Shape**'s

draw(). Finally, **draw()** invokes the **draw_lines()** for the particular kind of shape. A call of **gui_main()** in our user code starts the display engine.



What **gui_main()**? So far, we haven't actually seen **gui_main()** in our code. Instead we use **wait_for_button()**, which invokes the display engine in a more simple-minded manner.

Shape's **move()** function simply moves every point stored relative to the current position:

```

void Shape::move(int dx, int dy) // move the shape +=dx and +=dy
{
    for (int i = 0; i<points.size(); ++i) {
        points[i].x+=dx;
        points[i].y+=dy;
    }
}
  
```

Like **draw_lines()**, **move()** is virtual because a derived class may have data that needs to be moved and that **Shape** does not know about. For example, see **Axis** (§12.7.3 and §15.4).

The **move()** function is not logically necessary for **Shape**; we just provided it for convenience and to provide another example of a virtual function. Every kind of **Shape** that has points that it didn't store in its **Shape** must define its own **move()**.

14.2.4 Copying and mutability

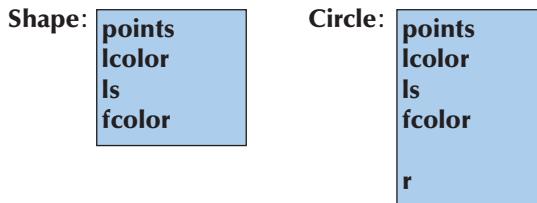
The `Shape` class declared the copy constructor and the copy assignment `=delete`:

```
Shape(const Shape&) =delete; // prevent copying
Shape& operator=(const Shape&) =delete;
```

The effect is to eliminate the otherwise default copy operations. For example:

```
void my_fct(Open_polyline& op, const Circle& c)
{
    Open_polyline op2 = op; // error: Shape's copy constructor is deleted
    vector<Shape> v;
    v.push_back(c); // error: Shape's copy constructor is deleted
    // ...
    op = op2; // error: Shape's assignment is deleted
}
```

But copying is useful in so many places! Just look at that `push_back()`; without copying, it is hard even to use `vectors` (`push_back()` puts a *copy* of its argument into its `vector`). Why would anyone make trouble for programmers by preventing copying? You prohibit the default copy operations for a type if they are likely to cause trouble. As a prime example of “trouble,” look at `my_fct()`. We cannot copy a `Circle` into a `Shape`-size element “slot” in `v`; a `Circle` has a radius but `Shape` does not, so `sizeof(Shape)<sizeof(Circle)`. If that `v.push_back(c)` were allowed, the `Circle` would be “sliced” and any future use of the resulting `Shape` element would most likely lead to a crash; the `Circle` operations would assume a radius member (`r`) that hadn’t been copied:



The copy construction of `op2` and the assignment to `op` suffer from exactly the same problem. Consider:

```
Marked_polyline mp {"x"};
Circle c(p,10);
my_fct(mp,c); // the Open_polyline argument refers to a Marked_polyline
```

Now the copy operations of the **Open_polyline** would “slice” **mp**’s **string** member **mark** away.

Basically, class hierarchies plus pass-by-reference and default copying do not mix. When you design a class that is meant to be a base class in a hierarchy, disable its copy constructor and copy assignment using **=delete** as was done for **Shape**.

Slicing (yes, that’s really a technical term) is not the only reason to prevent copying. There are quite a few concepts that are best represented without copy operations. Remember that the graphics system has to remember where a **Shape** is stored to display it to the screen. That’s why we “attach” **Shapes** to a **Window**, rather than copy. For example, if a **Window** held only a copy of a **Shape**, rather than a reference to the **Shape**, changes to the original would not affect the copy. So if we changed the **Shape**’s color, the **Window** would not notice the change and would display its copy with the unchanged color. A copy would in a very real sense not be as good as its original.

If we want to copy objects of types where the default copy operations have been disabled, we can write an explicit function to do the job. Such a copy function is often called **clone()**. Obviously, you can write a **clone()** only if the functions for reading members are sufficient for expressing what is needed to construct a copy, but that is the case for all **Shapes**.

14.3 Base and derived classes

Let’s take a more technical view of base and derived classes; that is, let us for this section (only) change the focus of discussion from programming, application design, and graphics to programming language features. When designing our graphics interface library, we relied on three key language mechanisms:

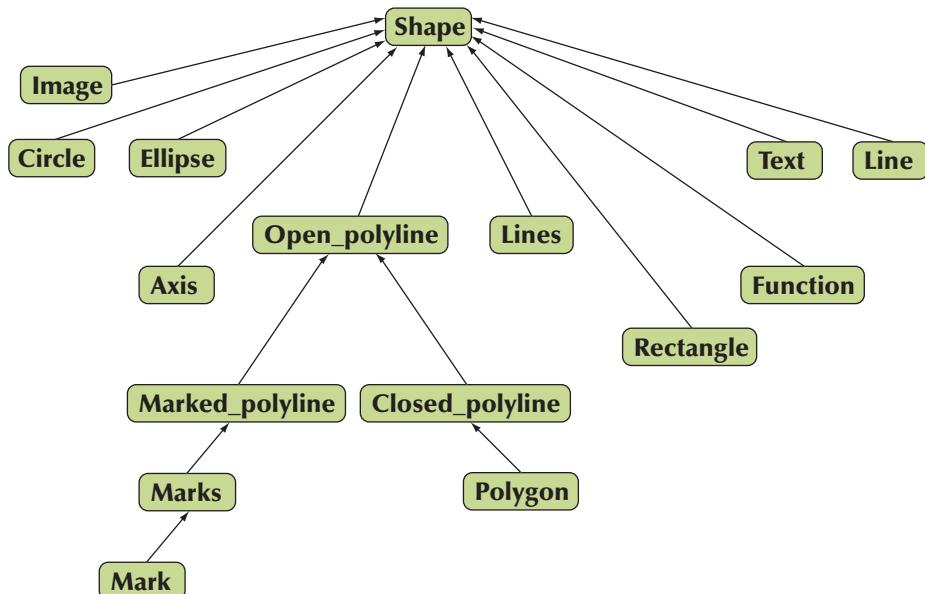
- *Derivation*: a way to build one class from another so that the new class can be used in place of the original. For example, **Circle** is derived from **Shape**, or in other words, “a **Circle** is a kind of **Shape**” or “**Shape** is a base of **Circle**.” The derived class (here, **Circle**) gets all of the members of its base (here, **Shape**) in addition to its own. This is often called *inheritance* because the derived class “inherits” all of the members of its base. In some contexts, a derived class is called a *subclass* and a base class is called a *superclass*.
- *Virtual functions*: the ability to define a function in a base class and have a function of the same name and type in a derived class called when a user calls the base class function. For example, when **Window** calls **draw_lines()** for a **Shape** that is a **Circle**, it is the **Circle**’s **draw_lines()** that is executed, rather than **Shape**’s own **draw_lines()**. This is often called *run-time*

polymorphism, dynamic dispatch, or run-time dispatch because the function called is determined at run time based on the type of the object used.

- *Private and protected members:* We kept the implementation details of our classes private to protect them from direct use that could complicate maintenance. That's often called *encapsulation*.

The use of inheritance, run-time polymorphism, and encapsulation is the most common definition of *object-oriented programming*. Thus, C++ directly supports object-oriented programming in addition to other programming styles. For example, in Chapters 20–21, we'll see how C++ supports generic programming. C++ borrowed – with explicit acknowledgments – its key mechanisms from Simula67, the first language to directly support object-oriented programming (see Chapter 22).

That was a lot of technical terminology! But what does it all mean? And how does it actually work on our computers? Let's first draw a simple diagram of our graphics interface classes showing their inheritance relationships:

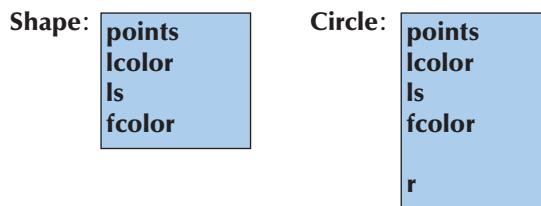


The arrows point from a derived class to its base. Such diagrams help visualize class relationships and often decorate the blackboards of programmers. Compared to commercial frameworks this is a tiny “class hierarchy” with only 16 classes, and only in the case of **Open polyline**'s many descendants is the hierarchy more than one deep. Clearly the common base (**Shape**) is the most important

class here, even though it represents an abstract concept so that we never directly make a shape.

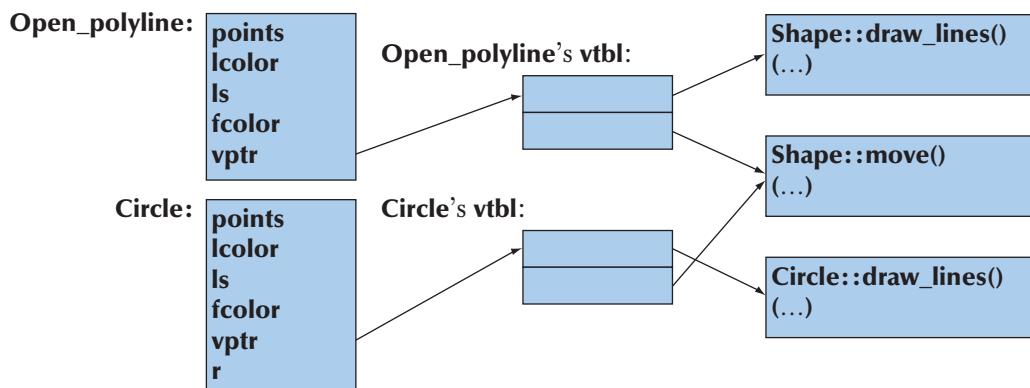
14.3.1 Object layout

How are objects laid out in memory? As we saw in §9.4.1, members of a class define the layout of objects: data members are stored one after another in memory. When inheritance is used, the data members of a derived class are simply added after those of a base. For example:



A **Circle** has the data members of a **Shape** (after all, it is a kind of **Shape**) and can be used as a **Shape**. In addition, **Circle** has “its own” data member **r** placed after the inherited data members.

To handle a virtual function call, we need (and have) one more piece of data in a **Shape** object: something to tell which function is really invoked when we call **Shape**'s **draw_lines()**. The way that is usually done is to add the address of a table of functions. This table is usually referred to as the **vtbl** (for “virtual table” or “virtual function table”) and its address is often called the **vptr** (for “virtual pointer”). We discuss pointers in Chapters 17–18; here, they act like references. A given implementation may use different names for **vtbl** and **vptr**. Adding the **vptr** and the **vtbls** to the picture we get



Since `draw_lines()` is the first virtual function, it gets the first slot in the `vtbl`, followed by that of `move()`, the second virtual function. A class can have as many virtual functions as you want it to have; its `vtbl` will be as large as needed (one slot per virtual function). Now when we call `x.draw_lines()`, the compiler generates a call to the function found in the `draw_lines()` slot in the `vtbl` for `x`. Basically, the code just follows the arrows on the diagram. So if `x` is a `Circle`, `Circle::draw_lines()` will be called. If `x` is of a type, say `Open_polyline`, that uses the `vtbl` exactly as `Shape` defined it, `Shape::draw_lines()` will be called. Similarly, `Circle` didn't define its own `move()` so `x.move()` will call `Shape::move()` if `x` is a `Circle`. Basically, code generated for a virtual function call simply finds the `vptr`, uses that to get to the right `vtbl`, and calls the appropriate function there. The cost is about two memory accesses plus the cost of an ordinary function call. This is simple and fast.

`Shape` is an abstract class so you can't actually have an object that's just a `Shape`, but an `Open_polyline` will have exactly the same layout as a "plain shape" since it doesn't add a data member or define a virtual function. There is just one `vtbl` for each class with a virtual function, not one for each object, so the `vtbls` tend not to add significantly to a program's object code size.

Note that we didn't draw any non-virtual functions in this picture. We didn't need to because there is nothing special about the way such functions are called and they don't increase the size of objects of their type.

Defining a function of the same name and type as a virtual function from a base class (such as `Circle::draw_lines()`) so that the function from the derived class is put into the `vtbl` instead of the version from the base is called *overriding*. For example, `Circle::draw_lines()` overrides `Shape::draw_lines()`.

Why are we telling you about `vtbls` and memory layout? Do you need to know about that to use object-oriented programming? No. However, many people strongly prefer to know how things are implemented (we are among those), and when people don't understand something, myths spring up. We have met people who were terrified of virtual functions "because they are expensive." Why? How expensive? Compared to what? Where would the cost matter? We explain the implementation model for virtual functions so that you won't have such fears. If you need a virtual function call (to select among alternatives at run time), you can't code the functionality to be any faster or to use less memory using other language features. You can see that for yourself.

14.3.2 Deriving classes and defining virtual functions

We specify that a class is to be a derived class by mentioning a base after the class name. For example:

```
struct Circle : Shape /* ... */;
```



By default, the members of a **struct** are **public** (§9.3), and that will include **public** members of a base. We could equivalently have said

```
class Circle : public Shape { public: /* . . . */};
```

These two declarations of **Circle** are completely equivalent, but you can have many long and fruitless discussions with people about which is better. We are of the opinion that time can be spent more productively on other topics.

Beware of forgetting **public** when you need it. For example:

```
class Circle : Shape { public: /* . . . */}; // probably a mistake
```

This would make **Shape** a **private** base of **Circle**, making **Shape**'s **public** functions inaccessible for a **Circle**. That's unlikely to be what you meant. A good compiler will warn about this likely error. There are uses for **private** base classes, but those are beyond the scope of this book.

A virtual function must be declared **virtual** in its class declaration, but if you place the function definition outside the class, the keyword **virtual** is neither required nor allowed out there. For example:

```
struct Shape {
    // ...
    virtual void draw_lines() const;
    virtual void move();
    // ...
};

virtual void Shape::draw_lines() const /* . . . */ // error
void Shape::move() /* . . . */ // OK
```

14.3.3 Overriding



When you want to override a virtual function, you must use exactly the same name and type as in the base class. For example:

```
struct Circle : Shape {
    void draw_lines(int) const; // probably a mistake (int argument?)
    void drawlines() const; // probably a mistake (misspelled name?)
    void draw_lines(); // probably a mistake (const missing?)
    // ...
};
```

Here, the compiler will see three functions that are independent of `Shape::draw_lines()` (because they have a different name or a different type) and won't override them. A good compiler will warn about these likely mistakes. There is nothing you can or must say in an overriding function to ensure that it actually overrides a base class function.

The `draw_lines()` example is real and can therefore be hard to follow in all details, so here is a purely technical example that illustrates overriding:

```
struct B {
    virtual void f() const { cout << "B::f ";
    void g() const { cout << "B::g "; // not virtual
};

struct D : B {
    void f() const { cout << "D::f "; // overrides B::f
    void g() { cout << "D::g ";
};

struct DD : D {
    void f() { cout << "DD::f "; // doesn't override D::f (not const)
    void g() const { cout << "DD::g ";
};
```

Here, we have a small class hierarchy with (just) one virtual function `f()`. We can try using it. In particular, we can try to call `f()` and the non-virtual `g()`, which is a function that doesn't know what type of object it had to deal with except that it is a `B` (or something derived from `B`):

```
void call(const B& b)
    // a D is a kind of B, so call() can accept a D
    // a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}

int main()
{
    B b;
    D d;
    DD dd;
```

```

call(b);
call(d);
call(dd);

b.f();
b.g();

d.f();
d.g();

dd.f();
dd.g();
}

```

You'll get

B::f B::g D::f B::g D::f B::f B::g D::f D::g DD::f DD::g

When you understand why, you'll know the mechanics of inheritance and virtual functions.

Obviously, it can be hard to keep track of which derived class functions are meant to override which base class functions. Fortunately, we can get compiler help to check. We can explicitly declare that a function is meant to override. Assuming that the derived class functions were meant to override, we can say so by adding **override** and the example becomes

```

struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; } // not virtual
};

struct D : B {
    void f() const override { cout << "D::f "; } // overrides B::f
    void g() override { cout << "D::g "; } // error: no virtual B::g to override
};

struct DD : D {
    void f() override { cout << "DD::f "; } // error: doesn't override
    // D::f (not const)
    void g() const override { cout << "DD::g "; } // error: no virtual D::g
    // to override
};

```

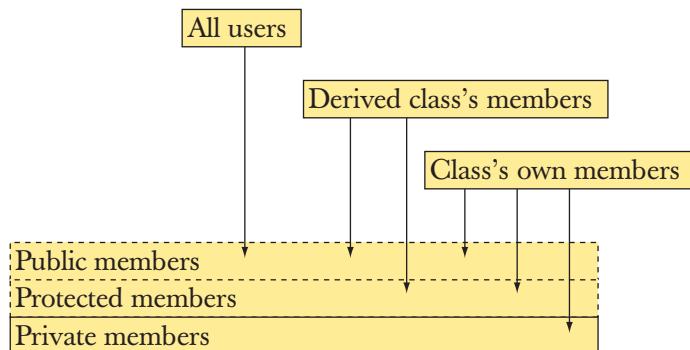
Explicit use of **override** is particularly useful in large, complicated class hierarchies.

14.3.4 Access

C++ provides a simple model of access to members of a class. A member of a class can be

- *Private*: If a member is **private**, its name can be used only by members of the class in which it is declared.
- *Protected*: If a member is **protected**, its name can be used only by members of the class in which it is declared and members of classes derived from that.
- *Public*: If a member is **public**, its name can be used by all functions.

Or graphically:



A base can also be **private**, **protected**, or **public**:

- If a base of class **D** is **private**, its **public** and **protected** member names can be used only by members of **D**.
- If a base of class **D** is **protected**, its **public** and **protected** member names can be used only by members of **D** and members of classes derived from **D**.
- If a base is **public**, its public member names can be used by all functions.

These definitions ignore the concept of “friend” and a few minor details, which are beyond the scope of this book. If you want to become a language lawyer you need to study Stroustrup, *The Design and Evolution of C++* and *The C++ Programming Language*, and the ISO C++ standard. We don’t recommend becoming a language lawyer (someone knowing every little detail of the language definition); being a programmer (a software developer, an engineer, a user, whatever you prefer to call someone who actually uses the language) is much more fun and typically much more useful to society.

14.3.5 Pure virtual functions

An abstract class is a class that can be used only as a base class. We use abstract classes to represent concepts that are abstract; that is, we use abstract classes for concepts that are generalizations of common characteristics of related entities.

Thick books of philosophy have been written trying to precisely define *abstract concept* (or *abstraction* or *generalization* or . . .). However you define it philosophically, the notion of an abstract concept is immensely useful. Examples are “animal” (as opposed to any particular kind of animal), “device driver” (as opposed to the driver for any particular kind of device), and “publication” (as opposed to any particular kind of book or magazine). In programs, abstract classes usually define interfaces to groups of related classes (*class hierarchies*).

In §14.2.1, we saw how to make a class abstract by declaring its constructor **protected**. There is another – and much more common – way of making a class abstract: state that one or more of its virtual functions needs to be overridden in some derived class. For example:

```
class B {                                // abstract base class
public:
    virtual void f() =0;      // pure virtual function
    virtual void g() =0;
};

B b;                                // error: B is abstract
```

The curious **=0** notation says that the virtual functions **B::f()** and **B::g()** are “pure”; that is, they must be overridden in some derived class. Since **B** has pure virtual functions, we cannot create an object of class **B**. Overriding the pure virtual functions solves this “problem”:

```
class D1 : public B {
public:
    void f() override;
    void g() override;
};

D1 d1;                                // OK
```

Note that unless all pure virtual functions are overridden, the resulting class is still abstract:

```
class D2 : public B {  
public:  
    void f() override;  
    // no g()  
};  
  
D2 d2;      // error: D2 is (still) abstract  
  
class D3 : public D2 {  
public:  
    void g() override;  
};  
  
D3 d3;      // OK
```

Classes with pure virtual functions tend to be pure interfaces; that is, they tend to have no data members (the data members will be in the derived classes) and consequently have no constructors (if there are no data members to initialize, a constructor is unlikely to be needed).



14.4 Benefits of object-oriented programming

When we say that **Circle** is derived from **Shape**, or that **Circle** is a kind of **Shape**, we do so to obtain (either or both)

- *Interface inheritance:* A function expecting a **Shape** (usually as a reference argument) can accept a **Circle** (and can use a **Circle** through the interface provided by **Shape**).
- *Implementation inheritance:* When we define **Circle** and its member functions, we can take advantage of the facilities (such as data and member functions) offered by **Shape**.

A design that does not provide interface inheritance (that is, a design for which an object of a derived class cannot be used as an object of its **public** base class) is a poor and error-prone design. For example, we might define a class called **Never_do_this** with **Shape** as its **public** base. Then we could override **Shape::draw_lines()** with a function that didn't draw the shape, but instead moved its center 100 pixels to the left. That "design" is fatally flawed because even though **Never_do_this**



provides the interface of a **Shape**, its implementation does not maintain the semantics (meaning, behavior) required of a **Shape**. Never do that!

Interface inheritance gets its name because its benefits come from code using the interface provided by a base class (“an interface”; here, **Shape**) and not having to know about the derived classes (“implementations”; here, classes derived from **Shape**).

Implementation inheritance gets its name because the benefits come from the simplification in the implementation of derived classes (e.g., **Circle**) provided by the facilities offered by the base class (here, **Shape**).

Note that our graphics design critically depends on interface inheritance: the “graphics engine” calls **Shape::draw()** which in turn calls **Shape**’s virtual function **draw_lines()** to do the real work of putting images on the screen. Neither the “graphics engine” nor indeed class **Shape** knows which kinds of shapes exist. In particular, our “graphics engine” (FLTK plus the operating system’s graphics facilities) was written and compiled years before our graphics classes! We just define particular shapes and **attach()** them to **Windows** as **Shapes** (**Window::attach()** takes a **Shape&** argument; see §E.3). Furthermore, since class **Shape** doesn’t know about your graphics classes, you don’t need to recompile **Shape** each time you define a new graphics interface class.

In other words, we can add new **Shapes** to a program without modifying existing code. This is a holy grail of software design/development/maintenance: extension of a system without modifying it. There are limits to which changes we can make without modifying existing classes (e.g., **Shape** offers a rather limited range of services), and the technique doesn’t apply well to all programming problems (see, for example, Chapters 17–19 where we define **vector**; inheritance has little to offer for that). However, interface inheritance is one of the most powerful techniques for designing and implementing systems that are robust in the face of change.

Similarly, implementation inheritance has much to offer, but it is no panacea. By placing useful services in **Shape**, we save ourselves the bother of repeating work over and over again in the derived classes. That can be most significant in real-world code. However, it comes at the cost that any change to the interface of **Shape** or any change to the layout of the data members of **Shape** necessitates a recompilation of all derived classes and their users. For a widely used library, such recompilation can be simply infeasible. Naturally, there are ways of gaining most of the benefits while avoiding most of the problems; see §14.3.5.



Drill

Unfortunately, we can't construct a drill for the understanding of general design principles, so here we focus on the language features that support object-oriented programming.

1. Define a class **B1** with a virtual function **vf()** and a non-virtual function **f()**. Define both of these functions within class **B1**. Implement each function to output its name (e.g., **B1::vf()**). Make the functions **public**. Make a **B1** object and call each function.
2. Derive a class **D1** from **B1** and override **vf()**. Make a **D1** object and call **vf()** and **f()** for it.
3. Define a reference to **B1** (a **B1&**) and initialize that to the **D1** object you just defined. Call **vf()** and **f()** for that reference.
4. Now define a function called **f()** for **D1** and repeat 1–3. Explain the results.
5. Add a pure virtual function called **pvf()** to **B1** and try to repeat 1–4. Explain the result.
6. Define a class **D2** derived from **D1** and override **pvf()** in **D2**. Make an object of class **D2** and invoke **f()**, **vf()**, and **pvf()** for it.
7. Define a class **B2** with a pure virtual function **pvf()**. Define a class **D21** with a **string** data member and a member function that overrides **pvf()**; **D21::pvf()** should output the value of the **string**. Define a class **D22** that is just like **D21** except that its data member is an **int**. Define a function **f()** that takes a **B2&** argument and calls **pvf()** for its argument. Call **f()** with a **D21** and a **D22**.

Review

1. What is an application domain?
2. What are ideals for naming?
3. What can we name?
4. What services does a **Shape** offer?
5. How does an abstract class differ from a class that is not abstract?
6. How can you make a class abstract?
7. What is controlled by access control?
8. What good can it do to make a data member **private**?
9. What is a virtual function and how does it differ from a non-virtual function?
10. What is a base class?
11. What makes a class derived?
12. What do we mean by object layout?
13. What can you do to make a class easier to test?

14. What is an inheritance diagram?
15. What is the difference between a **protected** member and a **private** one?
16. What members of a class can be accessed from a class derived from it?
17. How does a pure virtual function differ from other virtual functions?
18. Why would you make a member function virtual?
19. Why would you make a virtual member function pure?
20. What does overriding mean?
21. How does interface inheritance differ from implementation inheritance?
22. What is object-oriented programming?

Terms

abstract class	mutability	pure virtual function
access control	object layout	subclass
base class	object-oriented override	superclass
derived class	polymorphism	virtual function
dispatch	private	virtual function call
encapsulation	protected	virtual function table
inheritance	public	

Exercises

1. Define two classes **Smiley** and **Frowny**, which are both derived from class **Circle** and have two eyes and a mouth. Next, derive classes from **Smiley** and **Frowny** which add an appropriate hat to each.
2. Try to copy a **Shape**. What happens?
3. Define an abstract class and try to define an object of that type. What happens?
4. Define a class **Immobile_Circle**, which is just like **Circle** but can't be moved.
5. Define a **Striped_rectangle** where instead of fill, the rectangle is "filled" by drawing one-pixel-wide horizontal lines across the inside of the rectangle (say, draw every second line like that). You may have to play with the width of lines and the line spacing to get a pattern you like.
6. Define a **Striped_circle** using the technique from **Striped_rectangle**.
7. Define a **Striped_closed_polyline** using the technique from **Striped_rectangle** (this requires some algorithmic inventiveness).
8. Define a class **Octagon** to be a regular octagon. Write a test that exercises all of its functions (as defined by you or inherited from **Shape**).
9. Define a **Group** to be a container of **Shapes** with suitable operations applied to the various members of the **Group**. Hint: **Vector_ref**. Use a

- Group** to define a checkers (draughts) board where pieces can be moved under program control.
10. Define a class **Pseudo_window** that looks as much like a **Window** as you can make it without heroic efforts. It should have rounded corners, a label, and control icons. Maybe you could add some fake “contents,” such as an image. It need not actually do anything. It is acceptable (and indeed recommended) to have it appear within a **Simple_window**.
 11. Define a **Binary_tree** class derived from **Shape**. Give the number of levels as a parameter (**levels==0** means no nodes, **levels==1** means one node, **levels==2** means one top node with two sub-nodes, **levels==3** means one top node with two sub-nodes each with two sub-nodes, etc.). Let a node be represented by a small circle. Connect the nodes by lines (as is conventional). P.S. In computer science, trees grow downward from a top node (amusingly, but logically, often called the root).
 12. Modify **Binary_tree** to draw its nodes using a virtual function. Then, derive a new class from **Binary_tree** that overrides that virtual function to use a different representation for a node (e.g., a triangle).
 13. Modify **Binary_tree** to take a parameter (or parameters) to indicate what kind of line to use to connect the nodes (e.g., an arrow pointing down or a red arrow pointing up). Note how this exercise and the last use two alternative ways of making a class hierarchy more flexible and useful.
 14. Add an operation to **Binary_tree** that adds text to a node. You may have to modify the design of **Binary_tree** to implement this elegantly. Choose a way to identify a node; for example, you might give a string "**lrlr**" for navigating left, right, right, left, and right down a binary tree (the root node would match both an initial **l** and an initial **r**).
 15. Most class hierarchies have nothing to do with graphics. Define a class **Iterator** with a pure virtual function **next()** that returns a **double*** (see Chapter 17). Now derive **Vector_iterator** and **List_iterator** from **Iterator** so that **next()** for a **Vector_iterator** yields a pointer to the next element of a **vector<double>** and **List_iterator** does the same for a **list<double>**. You initialize a **Vector_iterator** with a **vector<double>** and the first call of **next()** yields a pointer to its first element, if any. If there is no next element, return 0. Test this by using a function **void print(Iterator&)** to print the elements of a **vector<double>** and a **list<double>**.
 16. Define a class **Controller** with four virtual functions **on()**, **off()**, **set_level(int)**, and **show()**. Derive at least two classes from **Controller**. One should be a simple test class where **show()** prints out whether the class is set to on or off and what is the current level. The second derived class should somehow control the line color of a **Shape**; the exact meaning of “level” is up to you. Try to find a third “thing” to control with such a **Controller** class.

17. The exceptions defined in the C++ standard library, such as `exception`, `runtime_error`, and `out_of_range` (§5.6.3), are organized into a class hierarchy (with a useful virtual function `what()` returning a string supposedly explaining what went wrong). Search your information sources for the C++ standard exception class hierarchy and draw a class hierarchy diagram of it.

Postscript

The ideal for software is not to build a single program that does everything. The ideal is to build a lot of classes that closely reflect our concepts and that work together to allow us to build our applications elegantly, with minimal effort (relative to the complexity of our task), with adequate performance, and with confidence that the results produced are correct. Such programs are comprehensible and maintainable in a way that code that was simply thrown together to get a particular job done as quickly as possible is not. Classes, encapsulation (as supported by `private` and `protected`), inheritance (as supported by class derivation), and run-time polymorphism (as supported by virtual functions) are among our most powerful tools for structuring systems.





Graphing Functions and Data

“The best is the enemy of the good.”

—Voltaire

If you are in any empirical field, you need to graph data. If you are in any field that uses math to model phenomena, you need to graph functions. This chapter discusses basic mechanisms for such graphics. As usual, we show the use of the mechanisms and also discuss their design. The key examples are graphing a function of one argument and displaying values read from a file.

15.1 Introduction**15.2 Graphing simple functions****15.3 Function**

- 15.3.1 Default arguments
- 15.3.2 More examples
- 15.3.3 Lambda expressions

15.4 Axis**15.5 Approximation****15.6 Graphing data**

- 15.6.1 Reading a file
- 15.6.2 General layout
- 15.6.3 Scaling data
- 15.6.4 Building the graph

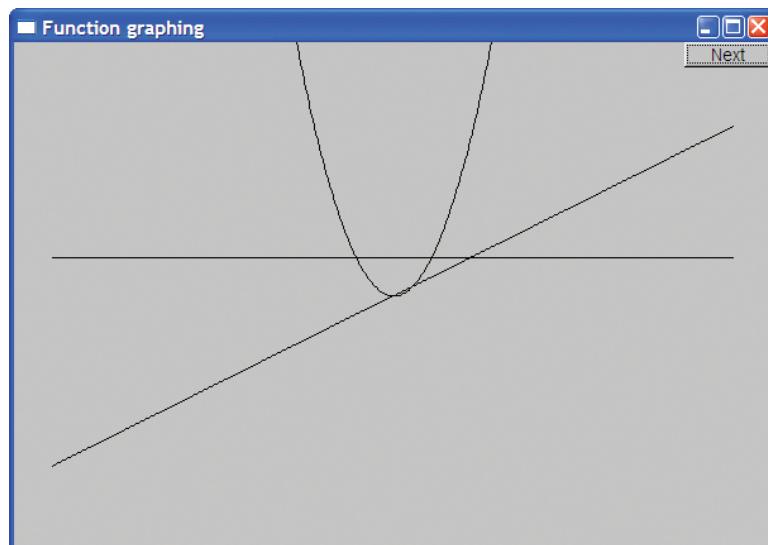


15.1 Introduction

Compared to the professional software systems you'll use if such visualization becomes your main occupation, the facilities presented here are primitive. Our primary aim is not elegance of output, but an understanding of how such graphical output can be produced and of the programming techniques used. You'll find the design techniques, programming techniques, and basic mathematical tools presented here of longer-term value than the graphics facilities presented. Therefore, please don't skim too quickly over the code fragments – they contain more of interest than just the shapes they compute and draw.

15.2 Graphing simple functions

Let's start. Let's look at examples of what we can draw and what code it takes to draw them. In particular, look at the graphics interface classes used. Here, first, are a parabola, a horizontal line, and a sloping line:



Actually, since this chapter is about graphing functions, that horizontal line isn't just a horizontal line; it is what we get from graphing the function

```
double one(double) { return 1; }
```

This is about the simplest function we could think of: it is a function of one argument that for every argument returns **1**. Since we don't need that argument to compute the result, we need not name it. For every **x** passed as an argument to **one()** we get the **y** value **1**; that is, the line is defined by $(x,y) == (x,1)$ for all **x**.

Like all beginning mathematical arguments, this is somewhat trivial and pedantic, so let's look at a slightly more complicated function:

```
double slope(double x) { return x/2; }
```

This is the function that generated the sloping line. For every **x**, we get the **y** value **x/2**. In other words, $(x,y) == (x,x/2)$. The point where the two lines cross is **(2,1)**.

Now we can try something more interesting, the square function that seems to reappear regularly in this book:

```
double square(double x) { return x*x; }
```

If you remember your high school geometry (and even if you don't), this defines a parabola with its lowest point at **(0,0)** and symmetric on the **y** axis. In other words, $(x,y) == (x,x*x)$. So, the lowest point where the parabola touches the sloping line is **(0,0)**.

Here is the code that drew those three functions:

```
constexpr int xmax = 600;           // window size
constexpr int ymax = 400;

constexpr int x_orig = xmax/2;     // position of (0,0) is center of window
constexpr int y_orig = ymax/2;
constexpr Point orig {x_orig,y_orig};

constexpr int r_min = -10;         // range [-10:11]
constexpr int r_max = 11;

constexpr int n_points = 400;      // number of points used in range

constexpr int x_scale = 30;        // scaling factors
constexpr int y_scale = 30;

Simple_window win {Point{100,100},xmax,ymax,"Function graphing"};
```

```
Function s {one,r_min,r_max,orig,n_points,x_scale,y_scale};  

Function s2 {slope,r_min,r_max,orig,n_points,x_scale,y_scale};  

Function s3 {square,r_min,r_max,orig,n_points,x_scale,y_scale};  
  

win.attach(s);  

win.attach(s2);  

win.attach(s3);  

win.wait_for_button();
```

First, we define a bunch of constants so that we won't have to litter our code with "magic constants." Then, we make a window, define the functions, attach them to the window, and finally give control to the graphics system to do the actual drawing.

All of this is repetition and "boilerplate" except for the definitions of the three **Functions**, **s**, **s2**, and **s3**:

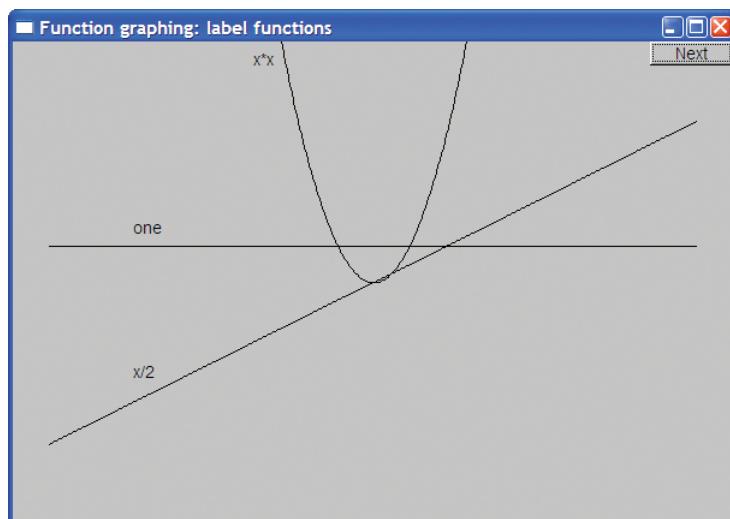
```
Function s {one,r_min,r_max,orig,n_points,x_scale,y_scale};  

Function s2 {slope,r_min,r_max,orig,n_points,x_scale,y_scale};  

Function s3 {square,r_min,r_max,orig,n_points,x_scale,y_scale};
```

Each **Function** specifies how its first argument (a function of one **double** argument returning a **double**) is to be drawn in a window. The second and third arguments give the range of **x** (the argument to the function to be graphed). The fourth argument (here, **orig**) tells the **Function** where the origin (**0,0**) is to be located within the window.

If you think that the many arguments are confusing, we agree. Our ideal is to have as few arguments as possible, because having many arguments confuses and provides opportunities for bugs. However, here we need them. We'll explain the last three arguments later (§15.3). First, however, let's label our graphs:

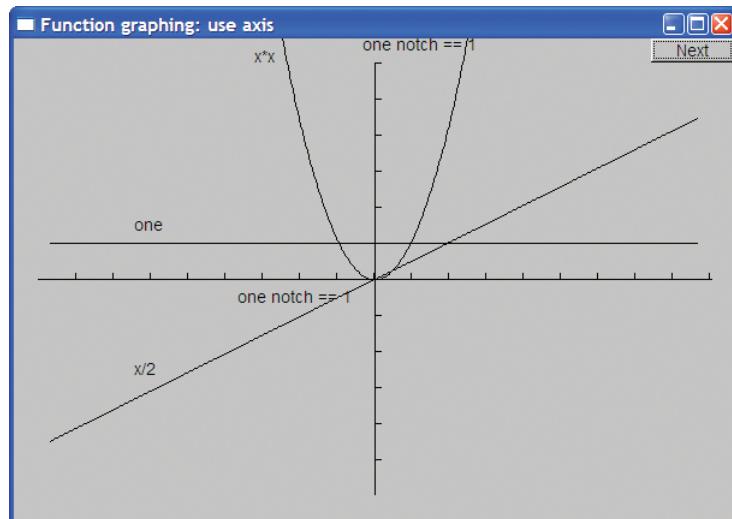


We always try to make our graphs self-explanatory. People don't always read the surrounding text and good diagrams get moved around, so that the surrounding text is "lost." Anything we put in as part of the picture itself is most likely to be noticed and – if reasonable – most likely to help the reader understand what we are displaying. Here, we simply put a label on each graph. The code for "labeling" was three **Text** objects (see §13.11):

```
Text ts {Point{100,y_orig-40}, "one"};
Text ts2 {Point{100,y_orig+y_orig/2-20}, "x/2"};
Text ts3 {Point{x_orig-100,20}, "x*x"};
win.set_label("Function graphing: label functions");
win.wait_for_button();
```

From now on in this chapter, we'll omit the repetitive code for attaching shapes to the window, labeling the window, and waiting for the user to hit "Next."

However, that picture is still not acceptable. We noticed that **x/2** touches **x*x** at **(0,0)** and that **one** crosses **x/2** at **(2,1)** but that's far too subtle; we need axes to give the reader an unsubtle clue about what's going on:



The code for the axes was two **Axis** objects (§15.4):

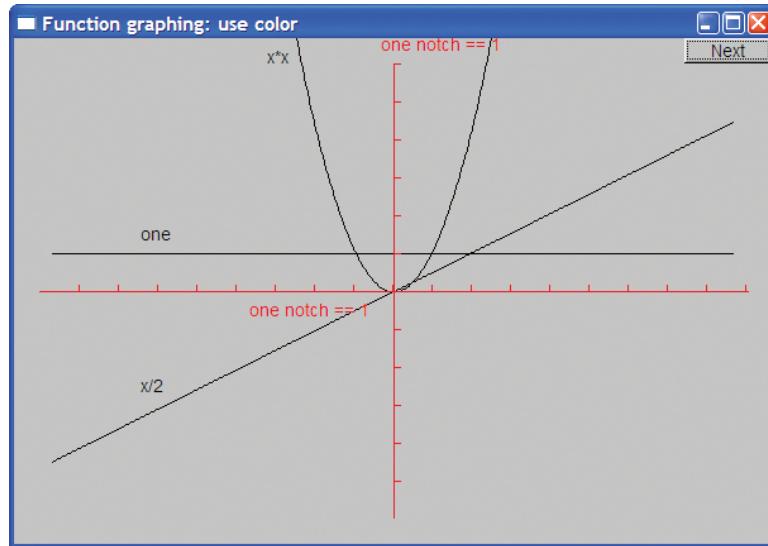
```
constexpr int xlength = xmax-40; // make the axis a bit smaller than the window
constexpr int ylength = ymax-40;

Axis x {Axis::x, Point{20,y_orig},
        xlength, xlength/x_scale, "one notch == 1"};
Axis y {Axis::y, Point{x_orig, ylength+20},
        ylength, ylength/y_scale, "one notch == 1"};
```

Using `xlength/x_scale` as the number of notches ensures that a notch represents the values 1, 2, 3, etc. Having the axes cross at **(0,0)** is conventional. If you prefer them along the left and bottom edges as is conventional for the display of data (see §15.6), you can of course do that instead. Another way of distinguishing the axes from the data is to use color:

```
x.set_color(Color::red);
y.set_color(Color::red);
```

And we get



This is acceptable, though for aesthetic reasons, we'd probably want a bit of empty space at the top to match what we have at the bottom and sides. It might also be a better idea to push the label for the x axis further to the left. We left these blemishes so that we could mention them – there are always more aesthetic details that we can work on. One part of a programmer's art is to know when to stop and use the time saved on something better (such as learning new techniques or sleep). Remember: "The best is the enemy of the good."

15.3 Function

The **Function** graphics interface class is defined like this:

```
struct Function : Shape {
    // the function parameters are not stored
```

```
Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale = 25, double yscale = 25);
};
```

Function is a **Shape** with a constructor that generates a lot of line segments and stores them in its **Shape** part. Those line segments approximate the values of function **f**. The values of **f** are calculated **count** times for values equally spaced in the **[r1:r2]** range:

```
Function::Function(Fct f, double r1, double r2, Point xy,
                  int count, double xscale, double yscale)
// graph f(x) for x in [r1:r2] using count line segments with (0,0) displayed at xy
// x coordinates are scaled by xscale and y coordinates scaled by yscale
{
    if (r2-r1<=0) error("bad graphing range");
    if (count <=0) error("non-positive graphing count");
    double dist = (r2-r1)/count;
    double r = r1;
    for (int i = 0; i<count; ++i) {
        add(Point{xy.x+int(r*xscale),xy.y-int(f(r)*yscale)});
        r += dist;
    }
}
```

The **xscale** and **yscale** values are used to scale the *x* coordinates and the *y* coordinates, respectively. We typically need to scale our values to make them fit appropriately into a drawing area of a window.

Note that a **Function** object doesn't store the values given to its constructor, so we can't later ask a function where its origin is, redraw it with different scaling, etc. All it does is to store points (in its **Shape**) and draw itself on the screen. If we wanted the flexibility to change a **Function** after construction, we would have to store the values we wanted to change (see exercise 2).

What is the type **Fct** that we used to represent a function argument? It is a variant of a standard library type called **std::function** that can "remember" a function to be called later. **Fct** requires its argument to be a **double** and its return type to be a **double**.

15.3.1 Default Arguments

Note the way the **Function** constructor arguments **xscale** and **yscale** were given initializers in the declaration. Such initializers are called *default arguments* and their values are used if a caller doesn't supply values. For example:

```
Function s {one, r_min, r_max,orig, n_points, x_scale, y_scale};
Function s2 {slope, r_min, r_max, orig, n_points, x_scale}; // no yscale
```

```
Function s3 {square, r_min, r_max, orig, n_points}; // no xscale, no yscale
Function s4 {sqrt, r_min, r_max, orig}; // no count, no xscale, no yscale
```

This is equivalent to

```
Function s {one, r_min, r_max, orig, n_points, x_scale, y_scale};
Function s2 {slope, r_min, r_max, orig, n_points, x_scale, 25};
Function s3 {square, r_min, r_max, orig, n_points, 25, 25};
Function s4 {sqrt, r_min, r_max, orig, 100, 25, 25};
```

Default arguments are used as an alternative to providing several overloaded functions. Instead of defining one constructor with three default arguments, we could have defined four constructors:

```
struct Function : Shape { // alternative, not using default arguments
    Function(Fct f, double r1, double r2, Point orig,
        int count, double xscale, double yscale);
    // default scale of y:
    Function(Fct f, double r1, double r2, Point orig,
        int count, double xscale);
    // default scale of x and y:
    Function(Fct f, double r1, double r2, Point orig, int count);
    // default count and default scale of x or y:
    Function(Fct f, double r1, double r2, Point orig);
};
```

 It would have been more work to define four constructors, and with the four-constructor version, the nature of the default is hidden in the constructor definitions rather than being obvious from the declaration. Default arguments are frequently used for constructors but can be useful for all kinds of functions. You can only define default arguments for trailing parameters. For example:

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale, double yscale); // error
};
```

If a parameter has a default argument, all subsequent parameters must also have one:

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale=25, double yscale=25);
};
```

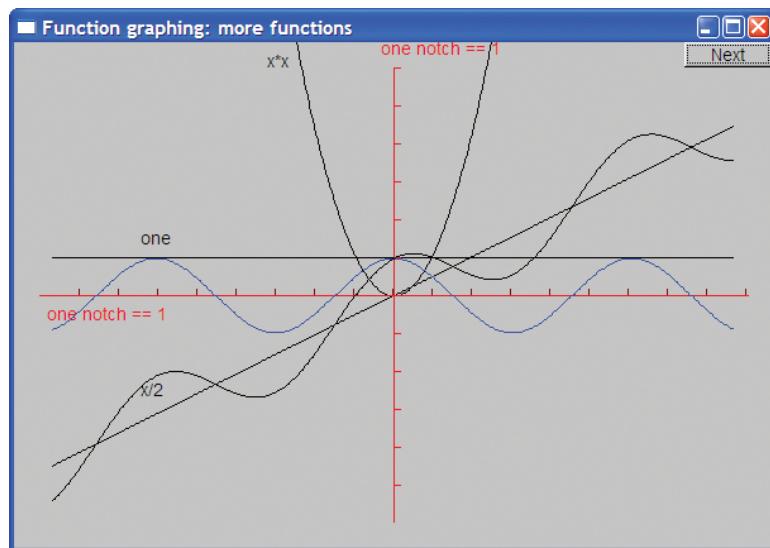
Sometimes, picking good default arguments is easy. Examples of that are the default for `string` (the empty `string`) and the default for `vector` (the empty `vector`). In other cases, such as `Function`, choosing a default is less easy; we found the ones we used after a bit of experimentation and a failed attempt. Remember, you don't have to provide default arguments, and if you find it hard to provide one, just leave it to your user to specify that argument.

15.3.2 More examples

We added a couple more functions, a simple cosine (`cos`) from the standard library, and – just to show how we can compose functions – a sloping cosine that follows the $x/2$ slope:

```
double sloping_cos(double x) { return cos(x)+slope(x); }
```

Here is the result:



The code is

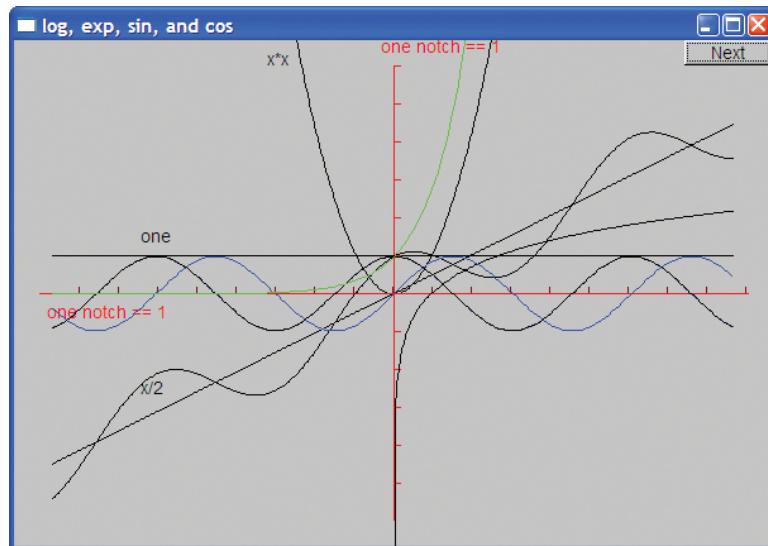
```
Function s4 {cos,r_min,r_max,orig,400,30,30};
s4.set_color(Color::blue);
Function s5 {sloping_cos, r_min,r_max,orig,400,30,30};
x.label.move(-160,0);
x.notches.set_color(Color::dark_red);
```

In addition to adding those two functions, we also moved the x axis's label and (just to show how) slightly changed the color of its notches.

Finally, we graph a log, an exponential, a sine, and a cosine:

```
Function f1 {log,0.000001,r_max,orig,200,30,30};      // log() logarithm, base e
Function f2 {sin,r_min,r_max,orig,200,30,30};        // sin()
f2.set_color(Color::blue);
Function f3 {cos,r_min,r_max,orig,200,30,30};        // cos()
Function f4 {exp,r_min,r_max,orig,200,30,30};        // exp() exponential  $e^x$ 
```

Since **log(0)** is undefined (mathematically, minus infinity), we started the range for **log** at a small positive number. The result is



Rather than labeling those functions we used color.

Standard mathematical functions, such as **cos()**, **sin()**, and **sqrt()**, are declared in the standard library header **<cmath>**. See §24.8 and §B.9.2 for lists of the standard mathematical functions.

15.3.3 Lambda expressions

It can get tedious to define a function just to have it to pass as an argument to a **Function**. Consequently, C++ offers a notation for defining something that acts as a function in the argument position where it is needed. For example, we could define the **sloping_cos** shape like this:

```
Function s5 {[](double x) { return cos(x)+slope(x); },
r_min,r_max,orig,400,30,30};
```

The `[](double x) { return cos(x)+slope(x); }` is a lambda expression; that is, it is an unnamed function defined right where it is needed as an argument. The `[]` is called a *lambda introducer*. After the lambda introducer, the lambda expression specifies what arguments are required (the argument list) and what actions are to be performed (the function body). The return type can be deduced from the lambda body. Here, the return type is `double` because that's the type of `cos(x)+slope(x)`. Had we wanted to, we could have specified the return type explicitly:

```
Function s5 [](double x) -> double { return cos(x)+slope(x); },
r_min,r_max,orig,400,30,30};
```

Specifying the return type for a lambda expression is rarely necessary. The main reason for that is that lambda expressions should be kept simple to avoid becoming a source of errors and confusion. If a piece of code does something significant, it should be given a name and probably requires a comment to be comprehensible to people other than the original programmer. We recommend using named functions for anything that doesn't easily fit on a line or two.

The lambda introducer can be used to give the lambda expression access to local variables; see §15.5. See also §21.4.3.



15.4 Axis

We use `Axis` wherever we present data (e.g., §15.6.4) because a graph without information that allows us to understand its scale is most often suspect. An `Axis` consists of a line, a number of “notches” on that line, and a text label. The `Axis` constructor computes the axis line and (optionally) the lines used as notches on that line:

```
struct Axis : Shape {
    enum Orientation { x, y, z };
    Axis(Orientation d, Point xy, int length,
        int number_of_notches=0, string label = "");

    void draw_lines() const override;
    void move(int dx, int dy) override;
    void set_color(Color c);

    Text label;
    Lines notches;
};
```

The **label** and **notches** objects are left public so that a user can manipulate them. For example, you can give the notches a different color from the line and **move()** the **label** to a more convenient location. **Axis** is an example of an object composed of several semi-independent objects.

The **Axis** constructor places the lines and adds the “notches” if **number_of_notches** is greater than zero:

```
Axis::Axis(Orientation d, Point xy, int length, int n, string lab)
    :label(Point{0,0},lab)
{
    if (length<0) error("bad axis length");
    switch (d){
        case Axis::x:
            { Shape::add(xy);                      // axis line
              Shape::add(Point{xy.x+length,xy.y});

                if (0<n) {           // add notches
                    int dist = length/n;
                    int x = xy.x+dist;
                    for (int i = 0; i<n; ++i) {
                        notches.add(Point{xy.x,y},Point{x,xy.y-5});
                        x += dist;
                    }
                }
            }

            label.move(length/3,xy.y+20);      // put the label under the line
            break;
        }
        case Axis::y:
            { Shape::add(xy);                      // a y axis goes up
              Shape::add(Point{xy.x,xy.y-length});

                if (0<n) {           // add notches
                    int dist = length/n;
                    int y = xy.y-dist;
                    for (int i = 0; i<n; ++i) {
                        notches.add(Point{xy.x,y},Point{xy.x+5,y});
                        y -= dist;
                    }
                }
            }
    }
}
```

```

        label.move(xy.x-10,xy.y-length-10);      // put the label at top
        break;
    }
    case Axis::z:
        error("z axis not implemented");
    }
}

```

Compared to much real-world code, this constructor is very simple, but please have a good look at it because it isn't quite trivial and it illustrates a few useful techniques. Note how we store the line in the **Shape** part of the **Axis** (using **Shape::add()**) but the notches are stored in a separate object (**notches**). That way, we can manipulate the line and the notches independently; for example, we can give each its own color. Similarly, a label is placed in a fixed position relative to its axes, but since it is a separate object, we can always move it to a better spot. We use the enumeration **Orientation** to provide a convenient and non-error-prone notation for users.

Since an **Axis** has three parts, we must supply functions for when we want to manipulate an **Axis** as a whole. For example:

```

void Axis::draw_lines() const
{
    Shape::draw_lines();
    notches.draw();    // the notches may have a different color from the line
    label.draw();      // the label may have a different color from the line
}

```

We use **draw()** rather than **draw_lines()** for **notches** and **label** to be able to use the color stored in them. The line is stored in the **Axis::Shape** itself and uses the color stored there.

We can set the color of the line, the notches, and the label individually, but stylistically it's usually better not to, so we provide a function to set all three to the same:

```

void Axis::set_color(Color c)
{
    Shape::set_color(c);
    notches.set_color(c);
    label.set_color(c);
}

```

Similarly, `Axis::move()` moves all the parts of the `Axis` together:

```
void Axis::move(int dx, int dy)
{
    Shape::move(dx,dy);
    notches.move(dx,dy);
    label.move(dx,dy);
}
```

15.5 Approximation

Here we give another small example of graphing a function: we “animate” the calculation of an exponential function. The purpose is to help you get a feel for mathematical functions (if you haven’t already), to show the way graphics can be used to illustrate computations, to give you some code to read, and finally to warn about a common problem with computations.

One way of computing an exponential function is to compute the series

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots$$

The more terms of this sequence we calculate, the more precise our value of e^x becomes; that is, the more terms we calculate, the more digits of the result will be mathematically correct. What we will do is to compute this sequence and graph the result after each term. The exclamation point here is used with the common mathematical meaning: factorial; that is, we graph these functions in order:

```
exp0(x) = 0          // no terms
exp1(x) = 1          // one term
exp2(x) = 1+x        // two terms; pow(x,1)/fac(1)==x
exp3(x) = 1+x+pow(x,2)/fac(2)
exp4(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)
exp5(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)+pow(x,4)/fac(4)
...

```

Each function is a slightly better approximation of ex than the one before it. Here, `pow(x,n)` is the standard library function that returns xn . There is no factorial function in the standard library, so we must define our own:

```
int fac(int n)          // factorial(n); n!
{
    int r = 1;
    while (n>1) {
        r*=n;
        --n;
    }
}
```

```

    }
    return r;
}

```

For an alternative implementation of `fac()`, see exercise 1. Given `fac()`, we can compute the n th term of the series like this:

```
double term(double x, int n) { return pow(x,n)/fac(n); } // nth term of series
```

Given `term()`, calculating the exponential to the precision of `n` terms is now easy:

```
double expe(double x, int n)      // sum of n terms for x
{
    double sum = 0;
    for (int i=0; i<n; ++i) sum+=term(x,i);
    return sum;
}
```

Let's use that to produce some graphics. First, we'll provide some axes and the "real" exponential, the standard library `exp()`, so that we can see how close our approximation using `expe()` is:

```
Function real_exp {exp,r_min,r_max,orig,200,x_scale,y_scale};
real_exp.set_color(Color::blue);
```

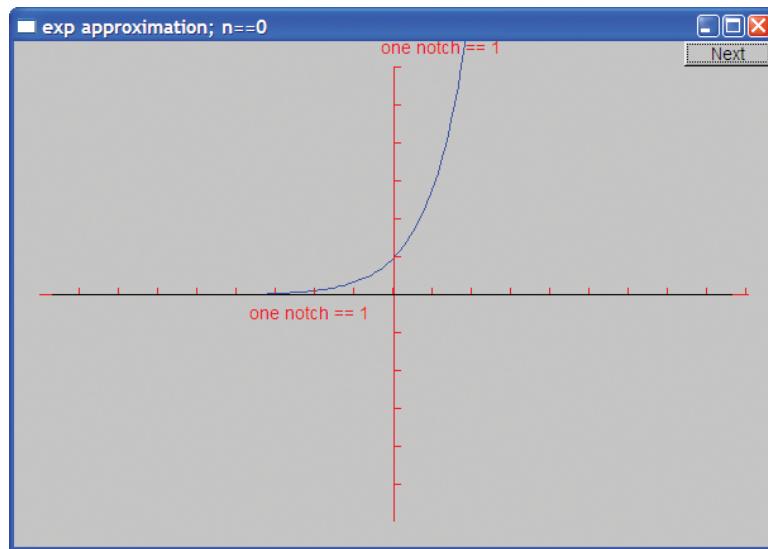
But how can we use `expe()`? From a programming point of view, the difficulty is that our graphing class, `Function`, takes a function of one argument and `expe()` needs two arguments. Given C++, as we have seen it so far, there is no really elegant solution to this problem. However, lambda expressions provide a way (§15.3.3). Consider:

```
for (int n = 0; n<50; ++n) {
    ostringstream ss;
    ss << "exp approximation; n==" << n ;
    win.set_label(ss.str());
    // get next approximation:
    Function e {[n](double x) { return expe(x,n); },
                r_min,r_max,orig,200,x_scale,y_scale};
    win.attach(e);
    win.wait_for_button();
    win.detach(e);
}
```

The lambda introducer, [n], says that the lambda expression may access the local variable n. That way, a call of `expe(x,n)` gets its n when its **Function** is created and its x from each call from within the **Function**.

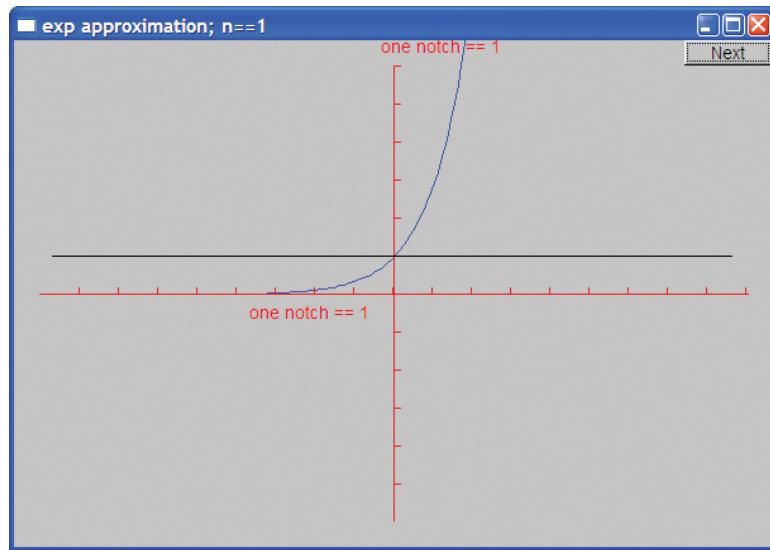
Note the final `detach(e)` in that loop. The scope of the **Function** object e is the block of the `for`-statement. Each time we enter that block we get a new **Function** called e, and each time we exit the block that e goes away, to be replaced by the next. The window must not remember the old e because it will have been destroyed. Thus, `detach(e)` ensures that the window does not try to draw a destroyed object.

This first gives a window with just the axes and the “real” exponential rendered in blue:

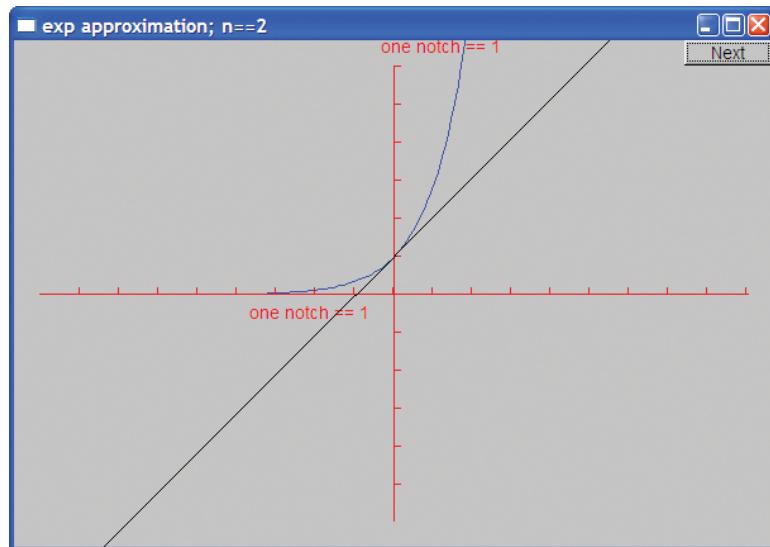


We see that `exp(0)` is 1 so that our blue “real exponential” crosses the y axis at **(0,1)**.

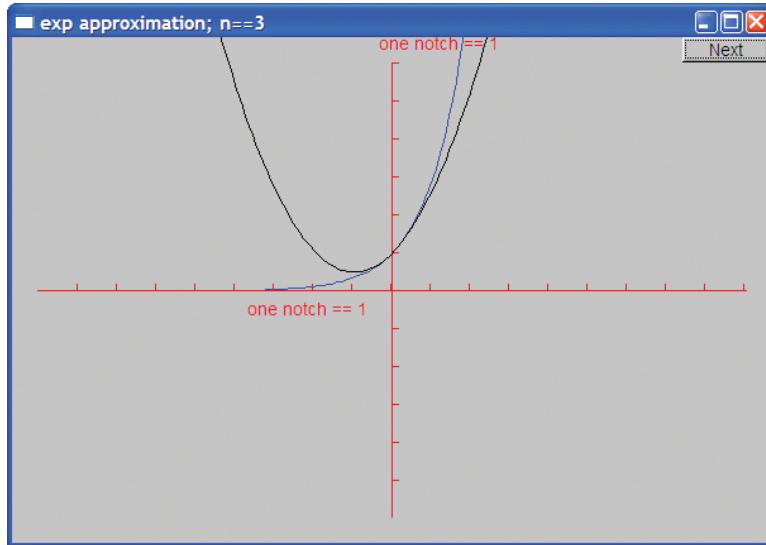
If you look carefully, you’ll see that we actually drew the zero term approximation (`exp0(x)==0`) as a black line right on top of the x axis. Hitting “Next,” we get the approximation using just one term. Note that we display the number of terms used in the approximation in the window label:



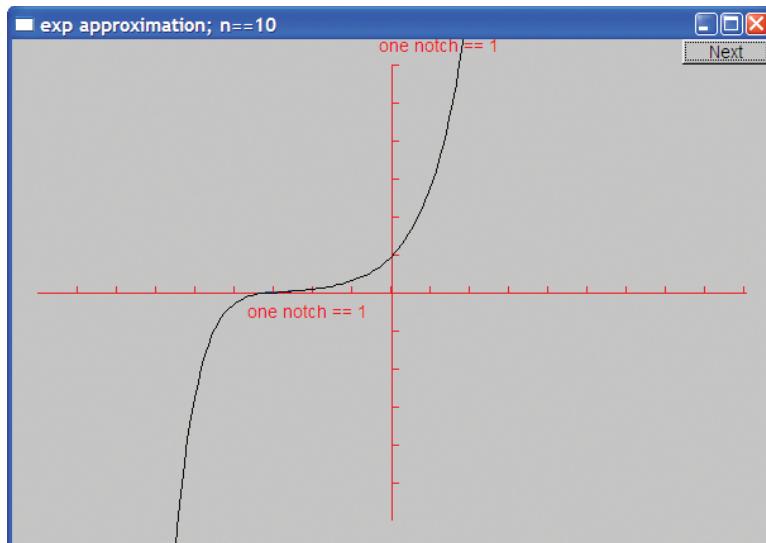
That's the function **exp1(x)==1**, the approximation using just one term of the sequence. It matches the exponential perfectly at **(0,1)**, but we can do better:



With two terms ($1+x$), we get the diagonal crossing the y axis at $(0,1)$. With three terms ($1+x+\text{pow}(x,2)/\text{fac}(2)$), we can see the beginning of a convergence:

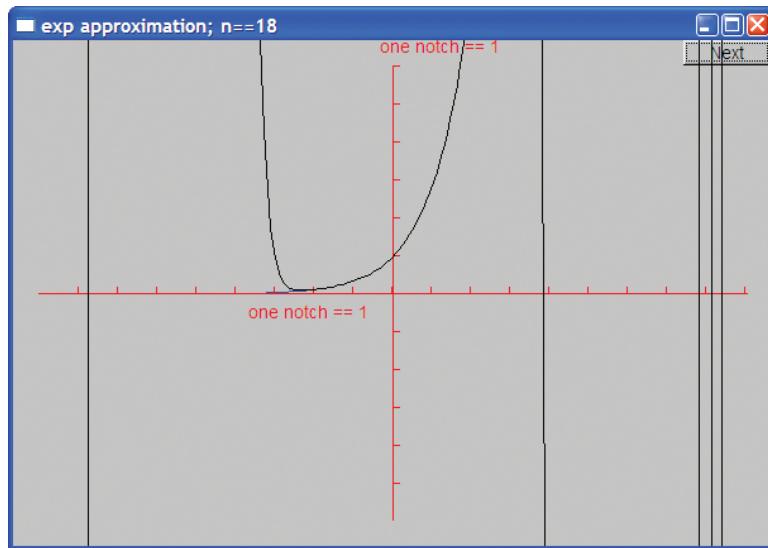


With ten terms we are doing rather well, especially for values larger than -3:



If we don't think too much about it, we might believe that we could get better and better approximations simply by using more and more terms. However, there are

limits, and after 13 terms something strange starts to happen. First, the approximations start to get slightly worse, and at 18 terms vertical lines appear:



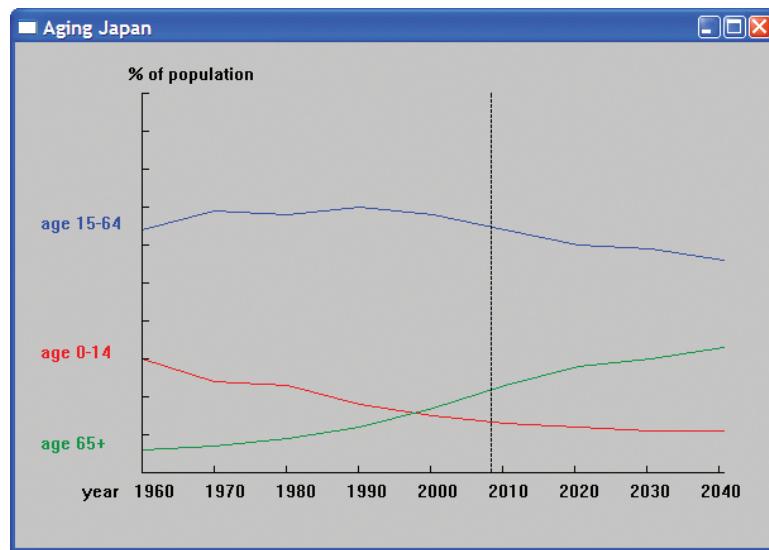
Remember, the computer's arithmetic is not pure math. Floating-point numbers are simply as good an approximation to real numbers as we can get with a fixed number of bits. An `int` overflows if you try to place a too-large integer in it, whereas a `double` stores an approximation. When I saw the strange output for larger numbers of terms, I first suspected that our calculation started to produce values that couldn't be represented as `doubles`, so that our results started to diverge from the mathematically correct answers. Later, I realized that `fac()` was producing values that couldn't be stored in an `int`. Modifying `fac()` to produce a `double` solved the problem. For more information, see exercise 11 of Chapter 5 and §24.2.

This last picture is also a good illustration of the principle that “it looks OK” isn’t the same as “tested.” Before giving a program to someone else to use, first test it beyond what at first seems reasonable. Unless you know better, running a program slightly longer or with slightly different data could lead to a real mess – as in this case.

15.6 Graphing data

Displaying data is a highly skilled and highly valued craft. When done well, it combines technical and artistic aspects and can add significantly to our understanding of complex phenomena. However, that also makes graphing a huge area

that for the most part is unrelated to programming techniques. Here, we'll just show a simple example of displaying data read from a file. The data shown represents the age groups of Japanese people over almost a century. The data to the right of the 2008 line is a projection:



We'll use this example to discuss the programming problems involved in presenting such data:

- Reading a file
- Scaling data to fit the window
- Displaying the data
- Labeling the graph

We will not go into artistic details. Basically, this is “graphs for geeks,” not “graphical art.” Clearly, you can do better artistically when you need to.

Given a set of data, we must consider how best to display it. To simplify, we will only deal with data that is easy to display using two dimensions, but that's a huge part of the data most people deal with. Note that bar graphs, pie charts, and similar popular displays really are just two-dimensional data displayed in a fancy way. Three-dimensional data can often be handled by producing a series of two-dimensional images, by superimposing several two-dimensional graphs onto a single window (as is done in the “Japanese age” example), or by labeling indi-

vidual points with information. If we want to go beyond that, we'll have to write new graphics classes or adopt another graphics library.

So, our data is basically pairs of values, such as **(year, number of children)**. If we have more data, such as **(year, number of children, number of adults, number of elderly)**, we simply have to decide which pair of values – or pairs of values – we want to draw. In our example, we simply graphed **(year, number of children)**, **(year, number of adults)**, and **(year, number of elderly)**.

There are many ways of looking at a set of **(x,y)** pairs. When considering how to graph such a set it is important to consider whether one value is in some way a function of the other. For example, for a **(year, steel production)** pair it would be quite reasonable to consider the steel production a function of the year and display the data as a continuous line. **Open polyline** (§13.6) is the obvious choice for graphing such data. If **y** should not be seen as a function of **x**, for example **(gross domestic product per person, population of country)**, **Marks** (§13.15) can be used to plot unconnected points.

Now, back to our Japanese age distribution example.

15.6.1 Reading a file

The file of age distributions consists of lines like this:

```
( 1960 : 30 64 6 )
(1970 : 24 69 7 )
(1980 : 23 68 9 )
```

The first number after the colon is the percentage of children (age 0–14) in the population, the second is the percentage of adults (age 15–64), and the third is the percentage of the elderly (age 65+). Our job is to read those. Note that the formatting of the data is slightly irregular. As usual, we have to deal with such details.

To simplify that task, we first define a type **Distribution** to hold a data item and an input operator to read such data items:

```
struct Distribution {
    int year, young, middle, old;
};

istream& operator>>(istream& is, Distribution& d)
// assume format: ( year : young middle old )
{
    char ch1 = 0;
    char ch2 = 0;
```

```

char ch3 = 0;
Distribution dd;

if (is >> ch1 >> dd.year
    >> ch2 >> dd.young >> dd.middle >> dd.old
    >> ch3) {
    if (ch1 != '(' || ch2 != ':' || ch3 != ')') {
        is.clear(ios_base::failbit);
        return is;
    }
}
else
    return is;
d = dd;
return is;
}

```

This is a straightforward application of the ideas from Chapter 10. If this code isn't clear to you, please review that chapter. We didn't need to define a **Distribution** type and a **>>** operator. However, it simplifies the code compared to a brute-force approach of "just read the numbers and graph them." Our use of **Distribution** splits the code up into logical parts to help comprehension and debugging. Don't be shy about introducing types "just to make the code clearer." We define classes to make the code correspond more directly to the way we think about the concepts in our code. Doing so even for "small" concepts that are used only very locally in our code, such as a line of data representing the age distribution for a year, can be most helpful.

Given **Distribution**, the read loop becomes

```

string file_name = "japanese-age-data.txt";
ifstream ifs {file_name};
if (!ifs) error("can't open ",file_name);

// . . .

for (Distribution d; ifs>>d; ) {
    if (d.year<base_year || end_year<d.year)
        error("year out of range");
    if (d.young+d.middle+d.old != 100)
        error("percentages don't add up");
    // . . .
}

```

That is, we try to open the file **japanese-age-data.txt** and exit the program if we don't find that file. It is often a good idea *not* to "hardwire" a file name into the source code the way we did here, but we consider this program an example of a small "one-off" effort, so we don't burden the code with facilities that are more appropriate for long-lived applications. On the other hand, we did put **japanese-age-data.txt** into a named **string** variable so the program is easy to modify if we want to use it – or some of its code – for something else.

The read loop checks that the year read is in the expected range and that the percentages add up to 100. That's a basic sanity check for the data. Since **>>** checks the format of each individual data item, we didn't bother with further checks in the main loop.

15.6.2 General layout

So what do we want to appear on the screen? You can see our answer at the beginning of §15.6. The data seems to ask for three **Open_polyline**s – one for each age group. These graphs need to be labeled, and we decided to write a "caption" for each line at the left-hand side of the window. In this case, that seemed clearer than the common alternative: to place the label somewhere along the line itself. In addition, we use color to distinguish the graphs and associate their labels.

We want to label the *x* axis with the years. The vertical line through the year 2008 indicates where the graph goes from hard data to projected data.

We decided to just use the window's label as the title for our graph.

Getting graphing code both correct and good-looking can be surprisingly tricky. The main reason is that we have to do a lot of fiddly calculations of sizes and offsets. To simplify that, we start by defining a set of symbolic constants that defines the way we use our screen space:

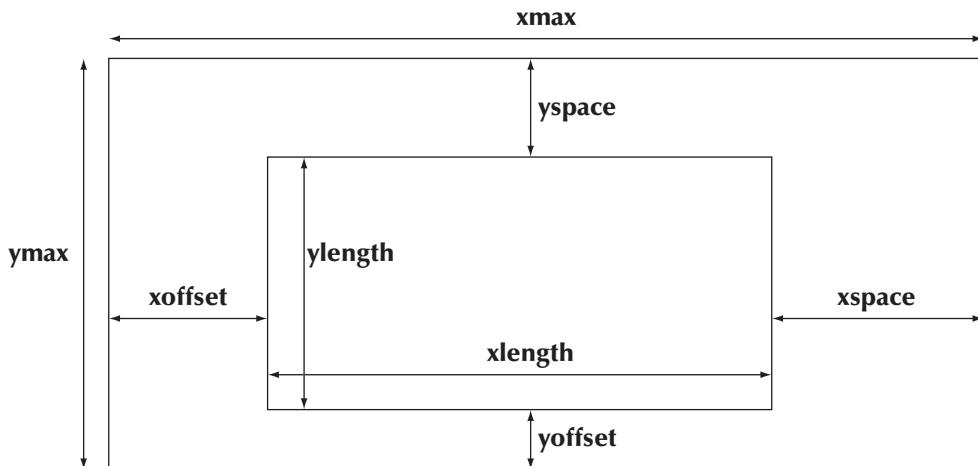
```
constexpr int xmax = 600; // window size
constexpr int ymax = 400;

constexpr int xoffset = 100; // distance from left-hand side of window to y axis
constexpr int yoffset = 60; // distance from bottom of window to x axis

constexpr int xspace = 40; // space beyond axis
constexpr int yspace = 40;

constexpr int xlength = xmax-xoffset-xspace; // length of axes
constexpr int ylength = ymax-yoffset-yspace;
```

Basically this defines a rectangular space (the window) with another rectangle (defined by the axes) within it:



We find that without such a “schematic view” of where things are in our window and the symbolic constants that define it, we get lost and become frustrated when our output doesn’t reflect our wishes.

15.6.3 Scaling data

Next we need to define how to fit our data into that space. We do that by scaling the data so that it fits into the space defined by the axes. To do that we need the scaling factors that are the ratio between the data range and the axis range:

```
constexpr int base_year = 1960;
constexpr int end_year = 2040;

constexpr double xscale = double(xlength)/(end_year-base_year);
constexpr double yscale = double(ylength)/100;
```

We want our scaling factors (**xscale** and **yscale**) to be floating-point numbers – or our calculations could be subject to serious rounding errors. To avoid integer division, we convert our lengths to **double** before dividing (§4.3.3).

We can now place a data point on the *x* axis by subtracting its base value (**1960**), scaling with **xscale**, and adding the **xoffset**. A *y* value is dealt with similarly. We find that we can never remember to do that quite right when we try to do it repeatedly. It may be a trivial calculation, but it is fiddly and verbose. To

simplify the code and minimize that chance of error (and minimize frustrating debugging), we define a little class to do the calculation for us:

```
class Scale {          // data value to coordinate conversion
    int cbase;        // coordinate base
    int vbase;        // base of values
    double scale;
public:
    Scale(int b, int vb, double s) :cbase{b}, vbase{vb}, scale{s} {}
    int operator()(int v) const { return cbase + (v-vbase)*scale; } // see §21.4
};
```

We want a class because the calculation depends on three constant values that we wouldn't like to unnecessarily repeat. Given that, we can define

```
Scale xs {xoffset,base_year,xscale};
Scale ys {ymax-yoffset,0,-yscale};
```

Note how we make the scaling factor for **ys** negative to reflect the fact that *y* coordinates grow downward whereas we usually prefer higher values to be represented by higher points on a graph. Now we can use **xs** to convert a year to an *x* coordinate. Similarly, we can use **ys** to convert a percentage to a *y* coordinate.

15.6.4 Building the graph

Finally, we have all the prerequisites for writing the graphing code in a reasonably elegant way. We start creating a window and placing the axes:

```
Window win {Point{100,100},xmax,ymax,"Aging Japan"};

Axis x {Axis::x, Point{xoffset,ymax-yoffset}, xlength,
(end_year-base_year)/10,
"year 1960 1970 1980 1990 "
"2000 2010 2020 2030 2040"};
x.label.move(-100,0);

Axis y {Axis::y, Point{xoffset,ymax-yoffset}, ylength, 10,"% of population"};

Line current_year {Point{xs(2008),ys(0)},Point{xs(2008),ys(100)}};
current_year.set_style(Line_style::dash);
```

The axes cross at **Point{xoffset,ymax-yoffset}** representing **(1960,0)**. Note how the notches are placed to reflect the data. On the *y* axis, we have ten notches each representing 10% of the population. On the *x* axis, each notch represents ten years, and the exact number of notches is calculated from **base_year** and **end_year** so that if we change that range, the axis would automatically be recalculated. This is one benefit of avoiding “magic constants” in the code. The label on the *x* axis violates that rule: it is simply the result of fiddling with the label string until the numbers were in the right position under the notches. To do better, we would have to look to a set of individual labels for individual “notches.”

Please note the curious formatting of the label string. We used two adjacent string literals:

```
"year 1960 1970 1980 1990 "
"2000 2010 2020 2030 2040"
```

Adjacent string literals are concatenated by the compiler, so that’s equivalent to

```
"year 1960 1970 1980 1990 2000 2010 2020 2030 2040"
```

That can be a useful “trick” for laying out long string literals to make our code more readable.

The **current_year** is a vertical line that separates hard data from projected data. Note how **xs** and **ys** are used to place and scale the line just right.

Given the axes, we can proceed to the data. We define three **Open_polyline**s and fill them in the read loop:

```
Open_polyline children;
Open_polyline adults;
Open_polyline aged;

for (Distribution d; ifs>>d; ) {
    if (d.year<base_year || end_year<d.year) error("year out of range");
    if (d.young+d.middle+d.old != 100)
        error("percentages don't add up");
    const int x = xs{d.year};
    children.add(Point{x,ys(d.young)});
    adults.add(Point{x,ys(d.middle)}));
    aged.add(Point{x,ys(d.old)}));
}
```

The use of `xs` and `ys` makes scaling and placement of the data trivial. “Little classes,” such as **Scale**, can be immensely important for simplifying notation and avoiding unnecessary repetition – thereby increasing readability and increasing the likelihood of correctness.

To make the graphs more readable, we label each and apply color:

```
Text children_label {Point{20,children.point(0).y}, "age 0-14"};
children.set_color(Color::red);
children_label.set_color(Color::red);

Text adults_label {Point{20,adults.point(0).y}, "age 15-64"};
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);

Text aged_label {Point{20,aged.point(0).y}, "age 65+"};
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
```

Finally, we need to attach the various **Shapes** to the **Window** and start the GUI system (§14.2.3):

```
win.attach(children);
win.attach(adults);
win.attach(aged);

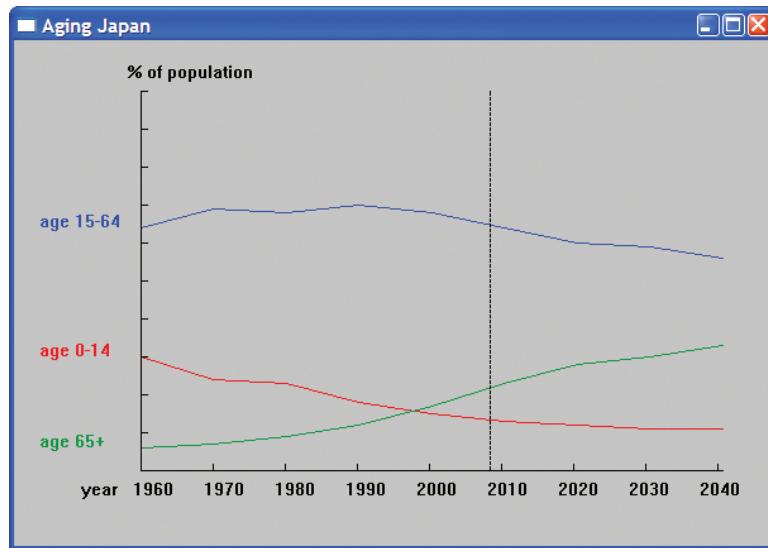
win.attach(children_label);
win.attach(adults_label);
win.attach(aged_label);

win.attach(x);
win.attach(y);
win.attach(current_year);

gui_main();
```

All the code could be placed inside **main()**, but we prefer to keep the helper classes **Scale** and **Distribution** outside together with **Distribution**’s input operator.

In case you have forgotten what we were producing, here is the output again:



Drill

Function graphing drill:

1. Make an empty 600-by-600 **Window** labeled “Function graphs.”
2. Note that you’ll need to make a project with the properties specified in the “installation of FLTK” note from the course website.
3. You’ll need to move **Graph.cpp** and **Window.cpp** into your project.
4. Add an *x* axis and a *y* axis each of length 400, labeled “1 = 20 pixels” and with a notch every 20 pixels. The axes should cross at (300,300).
5. Make both axes red.

In the following, use a separate **Shape** for each function to be graphed:

1. Graph the function **double one(double x) { return 1; }** in the range [-10,11] with (0,0) at (300,300) using 400 points and no scaling (in the window).
2. Change it to use *x* scale 20 and *y* scale 20.
3. From now on use that range, scale, etc. for all graphs.
4. Add **double slope(double x) { return x/2; }** to the window.
5. Label the slope with a **Text "x/2"** at a point just above its bottom left end point.

6. Add `double square(double x) { return x*x; }` to the window.
7. Add a cosine to the window (don't write a new function).
8. Make the cosine blue.
9. Write a function `sloping_cos()` that adds a cosine to `slope()` (as defined above) and add it to the window.

Class definition drill:

1. Define a `struct Person` containing a `string` name and an `int` age.
2. Define a variable of type `Person`, initialize it with "Goofy" and 63, and write it to the screen (`cout`).
3. Define an input (`>>`) and an output (`<<`) operator for `Person`; read in a `Person` from the keyboard (`cin`) and write it out to the screen (`cout`).
4. Give `Person` a constructor initializing `name` and `age`.
5. Make the representation of `Person` private, and provide `const` member functions `name()` and `age()` to read the name and age.
6. Modify `>>` and `<<` to work with the redefined `Person`.
7. Modify the constructor to check that `age` is [0:150) and that `name` doesn't contain any of the characters ; : " ' [] * & ^ % \$ # @ !. Use `error()` in case of error. Test.
8. Read a sequence of `Persons` from input (`cin`) into a `vector<Person>`; write them out again to the screen (`cout`). Test with correct and erroneous input.
9. Change the representation of `Person` to have `first_name` and `second_name` instead of `name`. Make it an error not to supply both a first and a second name. Be sure to fix `>>` and `<<` also. Test.

Review

1. What is a function of one argument?
2. When would you use a (continuous) line to represent data? When do you use (discrete) points?
3. What function (mathematical formula) defines a slope?
4. What is a parabola?
5. How do you make an x axis? A y axis?
6. What is a default argument and when would you use one?
7. How do you add functions together?
8. How do you color and label a graphed function?
9. What do we mean when we say that a series approximates a function?
10. Why would you sketch out the layout of a graph before writing the code to draw it?
11. How would you scale your graph so that the input will fit?
12. How would you scale the input without trial and error?

13. Why would you format your input rather than just having the file contain “the numbers”?
14. How do you plan the general layout of a graph? How do you reflect that layout in your code?

Terms

approximation	function	scaling
default argument	lambda	screen layout

Exercises

1. Here is another way of defining a factorial function:

```
int fac(int n) { return n>1 ? n*fac(n-1) : 1; } // factorial n!
```

It will do `fac(4)` by first deciding that since `4>1` it must be `4*fac(3)`, and that's obviously `4*3*fac(2)`, which again is `4*3*2*fac(1)`, which is `4*3*2*1`. Try to see that it works. A function that calls itself is said to be *recursive*. The alternative implementation in §15.5 is called *iterative* because it iterates through the values (using `while`). Verify that the recursive `fac()` works and gives the same results as the iterative `fac()` by calculating the factorial of 0, 1, 2, 3, 4, up until and including 20. Which implementation of `fac()` do you prefer, and why?

2. Define a class `Fct` that is just like `Function` except that it stores its constructor arguments. Provide `Fct` with “reset” operations, so that you can use it repeatedly for different ranges, different functions, etc.
3. Modify `Fct` from the previous exercise to take an extra argument to control precision or whatever. Make the type of that argument a template parameter for extra flexibility.
4. Graph a sine (`sin()`), a cosine (`cos()`), the sum of those (`sin(x)+cos(x)`), and the sum of the squares of those (`sin(x)*sin(x)+cos(x)*cos(x)`) on a single graph. Do provide axes and labels.
5. “Animate” (as in §15.5) the series `1-1/3+1/5-1/7+1/9-1/11+ . . .`. It is known as Leibniz’s series and converges to $\pi/4$.
6. Design and implement a bar graph class. Its basic data is a `vector<double>` holding N values, and each value should be represented by a “bar” that is a rectangle where the height represents the value.
7. Elaborate the bar graph class to allow labeling of the graph itself and its individual bars. Allow the use of color.
8. Here is a collection of heights in centimeters together with the number of people in a group of that height (rounded to the nearest 5cm): (170,7), (175,9), (180,23), (185,17), (190,6), (195,1). How would you graph that

- data? If you can't think of anything better, do a bar graph. Remember to provide axes and labels. Place the data in a file and read it from that file.
9. Find another data set of heights (an inch is 2.54cm) and graph them with your program from the previous exercise. For example, search the web for "height distribution" or "height of people in the United States" and ignore a lot of rubbish or ask your friends for their heights. Ideally, you don't have to change anything for the new data set. Calculating the scaling from the data is a key idea. Reading in labels from input also helps minimize changes when you want to reuse code.
 10. What kind of data is unsuitable for a line graph or a bar graph? Find an example and find a way of displaying it (e.g., as a collection of labeled points).
 11. Find the average maximum temperatures for each month of the year for two or more locations (e.g., Cambridge, England, and Cambridge, Massachusetts; there are lots of towns called "Cambridge") and graph them together. As ever, be careful with axes, labels, use of color, etc.

Postscript

Graphical representation of data is important. We simply understand a well-crafted graph better than the set of numbers that was used to make it. Most people, when they need to draw a graph, use someone else's code – a library. How are such libraries constructed and what do you do if you don't have one handy? What are the fundamental ideas underlying "an ordinary graphing tool"? Now you know: it isn't magic or brain surgery. We covered only two-dimensional graphs; three-dimensional graphing is also very useful in science, engineering, marketing, etc. and can be even more fun. Explore it someday!



Graphical User Interfaces

“Computing is not about
computers any more.
It is about living.”

—Nicholas Negroponte

A graphical user interface (GUI) allows a user to interact with a program by pressing buttons, selecting from menus, entering data in various ways, and displaying textual and graphical entities on a screen. That’s what we are used to when we interact with our computers and with websites. In this chapter, we show the basics of how code can be written to define and control a GUI application. In particular, we show how to write code that interacts with entities on the screen using callbacks. Our GUI facilities are built “on top of” system facilities. The low-level features and interfaces are presented in Appendix E, which uses features and techniques presented in Chapters 17 and 18. Here we focus on usage.

16.1 User interface alternatives	16.5 An example
16.2 The “Next” button	16.6 Control inversion
16.3 A simple window	16.7 Adding a menu
16.3.1 A callback function	16.8 Debugging GUI code
16.3.2 A wait loop	
16.3.3 A lambda expression as a callback	
16.4 Button and other Widgets	
16.4.1 Widgets	
16.4.2 Buttons	
16.4.3 In_box and Out_box	
16.4.4 Menus	

16.1 User interface alternatives



Every program has a user interface. A program running on a small gadget may be limited to input from a couple of push buttons and to a blinking light for output. Other computers are connected to the outside world only by a wire. Here, we will consider the common case in which our program communicates with a user who is watching a screen and using a keyboard and a pointing device (such as a mouse). In this case, we as programmers have three main choices:

- *Use console input and output:* This is a strong contender for technical/professional work where the input is simple and textual, consisting of commands and short data items (such as file names and simple data values). If the output is textual, we can display it on the screen or store it in files. The C++ standard library **iostreams** (Chapters 10–11) provide suitable and convenient mechanisms for this. If graphical output is needed, we can use a graphics display library (as shown in Chapters 12–15) without making dramatic changes to our programming style.
- *Use a graphical user interface (GUI) library:* This is what we do when we want our user interaction to be based on the metaphor of manipulating objects on the screen (pointing, clicking, dragging and dropping, hovering, etc.). Often (but not always), that style goes together with a high degree of graphically displayed information. Anyone who has used a modern computer knows examples where that is convenient. Anyone who wants to match the “feel” of Windows/Mac applications must use a GUI style of interaction.
- *Use a web browser interface:* For that, we need to use a markup (layout) language, such as HTML, and usually a scripting language. Showing how to

do this is beyond the scope of this book, but it is often the ideal for applications that require remote access. In that case, the communication between the program and the screen is again textual (using streams of characters). A browser is a GUI application that translates some of that text into graphical elements and translates the mouse clicks, etc. into textual data that can be sent back to the program.

To many, the use of GUI is the essence of modern programming, and sometimes the interaction with objects on the screen is considered the central concern of programming. We disagree: GUI is a form of I/O, and separation of the main logic of an application from I/O is among our major ideals for software. Wherever possible, we prefer to have a clean interface between our main program logic and the parts of the program we use to get input and produce output. Such a separation allows us to change the way a program is presented to a user, to port our programs to use different I/O systems, and – most importantly – to think about the logic of the program and its interaction with users separately.

That said, GUI is important and interesting from several perspectives. This chapter explores both the ways we can integrate graphical elements into our applications and how we can keep interface concerns from dominating our thinking.

16.2 The “Next” button

How did we provide that “Next” button that we used to drive the graphics examples in Chapters 12–15? There, we do graphics in a window using a button. Obviously, that is a simple form of GUI programming. In fact, it is so simple that some would argue that it isn’t “true GUI.” However, let’s see how it was done because it will lead directly into the kind of programming that everyone recognizes as GUI programming.

Our code in Chapters 12–15 is conventionally structured like this:

```
// create objects and/or manipulate objects, display them in Window win:  
win.wait_for_button();  
  
// create objects and/or manipulate objects, display them in Window win:  
win.wait_for_button();  
  
// create objects and/or manipulate objects, display them in Window win:  
win.wait_for_button();
```

Each time we reach `wait_for_button()`, we can look at our objects on the screen until we hit the button to get the output from the next part of the program. From the point of view of program logic, this is no different from a program that writes

lines of output to a screen (a console window), stopping now and then to receive input from the keyboard. For example:

```
// define variables and/or compute values, produce output  
cin >> var; // wait for input  
  
// define variables and/or compute values, produce output  
cin >> var; // wait for input  
  
// define variables and/or compute values, produce output  
cin >> var; // wait for input
```

From an implementation point of view, these two kinds of programs are quite different. When your program executes `cin >> var`, it stops and waits for “the system” to bring back characters you typed. However, the system (the graphical user interface system) that looks after your screen and tracks the mouse as you use it works on a rather different model: the GUI keeps track of where the mouse is and what the user is doing with the mouse (clicking, etc.). When your program wants an action, it must

- Tell the GUI what to look for (e.g., “Someone clicked the ‘Next’ button”)
- Tell what is to be done when someone does that
- Wait until the GUI detects an action that the program is interested in

What is new and different here is that the GUI does not just return to our program; it is designed to respond in different ways to different user actions, such as clicking on one of many buttons, resizing windows, redrawing the window after it has been obscured by another, and popping up pop-up menus.

For starters, we just want to say, “Please wake me up when someone clicks my button”; that is, “Please continue executing my program when someone clicks the mouse button and the cursor is in the rectangular area where the image of my button is displayed.” This is just about the simplest action we could imagine. However, such an operation isn’t provided by “the system” so we wrote one ourselves. Seeing how that is done is the first step in understanding GUI programming.

16.3 A simple window

Basically, “the system” (which is a combination of a GUI library and the operating system) continuously tracks where the mouse is and whether its buttons are pressed or not. A program can express interest in an area of the screen and ask “the system” to call a function when “something interesting” happens. In this particular

case, we ask the system to call one of our functions (a “callback function”) when the mouse button is clicked “on our button.” To do that we must

- Define a button
- Get it displayed
- Define a function for the GUI to call
- Tell the GUI about that button and that function
- Wait for the GUI to call our function

Let’s do that. A button is part of a **Window**, so (in **Simple_window.h**) we define our class **Simple_window** to contain a member **next_button**:

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button(); // simple event loop
private:
    Button next_button; // the “Next” button
    bool button_pushed; // implementation detail

    static void cb_next(Address, Address); // callback for next_button
    void next(); // action to be done when next_button is pressed
};
```

Obviously, **Simple_window** is derived from **Graph_lib**’s **Window**. All our windows must be derived directly or indirectly from **Graph_lib::Window** because it is the class that (through FLTK) connects our notion of a window with the system’s window implementation. For details of **Window**’s implementation, see §E.3.

Our button is initialized in **Simple_window**’s constructor:

```
Simple_window::Simple_window(Point xy, int w, int h, const string& title)
    : Window{xy, w, h, title},
    next_button{Point{x_max() - 70, 0}, 70, 20, "Next", cb_next},
    button_pushed{false}
{
    attach(next_button);
}
```

Unsurprisingly, **Simple_window** passes its location (**xy**), size (**w,h**), and title (**title**) on to **Graph_lib**’s **Window** to deal with. Next, the constructor initializes **next_button** with a location (**Point{x_max() - 70, 0}**; that’s roughly the top right corner), a size (**70,20**), a label (**"Next"**), and a “callback” function (**cb_next**). The first four

parameters exactly parallel what we do for a **Window**: we place a rectangular shape on the screen and label it.

Finally, we **attach()** our **next_button** to our **Simple_window**; that is, we tell the window that it must display the button in its position and make sure that the GUI system knows about it.

The **button_pushed** member is a pretty obscure implementation detail; we use it to keep track of whether the button has been pushed since last we executed **next()**. In fact, just about everything here is implementation details, and therefore declared **private**. Ignoring the implementation details, we see

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button(); // simple event loop

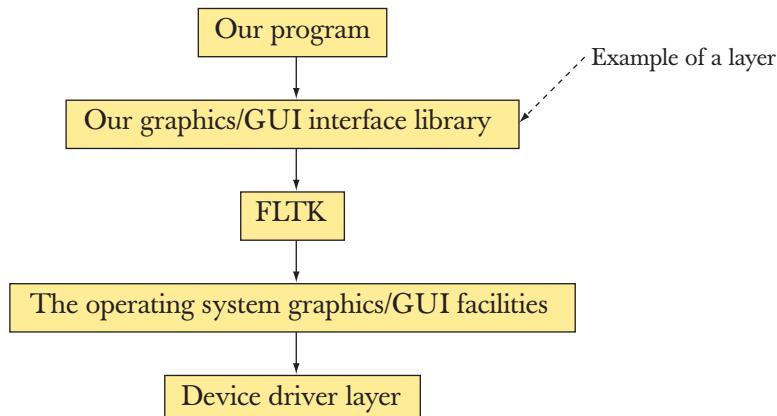
    // ...
};
```

That is, a user can make a window and wait for its button to be pushed.

16.3.1 A callback function

The function **cb_next()** is the new and interesting bit here. This is the function that we want the GUI system to call when it detects a click on our button. Since we give the function to the GUI for the GUI to “call back to us,” it’s commonly called a *callback* function. We indicate **cb_next()**’s intended use with the prefix **cb_** for “callback.” That’s just to help us – no language or library requires that naming convention. Obviously, we chose the name **cb_next** because it is to be the callback for our “Next” button. The definition of **cb_next** is an ugly piece of “boilerplate.”

Before showing that code, let’s consider what is going on here:



Our program runs on top of several “layers” of code. It uses our graphics library that we implement using the FLTK library, which is implemented using operating system facilities. In a system, there may be even more layers and sub-layers. Somehow, a click detected by the mouse’s device driver has to cause our function `cb_next()` to be called. We pass the address of `cb_next()` and the address of our `Simple_window` down through the layers of software; some code “down there” then calls `cb_next()` when the “Next” button is pressed.

The GUI system (and the operating system) can be used by programs written in a variety of languages, so it cannot impose some nice C++ style on all users. In particular, it does not know about our `Simple_window` class or our `Button` class. In fact, it doesn’t know about classes or member functions at all. The type required for a callback function is chosen so that it is usable from the lowest level of programming, including C and assembler. A callback function returns no value and takes two addresses as its arguments. We can declare a C++ member function that obeys those rules like this:

```
static void cb_next(Address, Address); // callback for next_button
```

The keyword `static` is there to make sure that `cb_next()` can be called as an ordinary function, that is, not as a C++ member function invoked for a specific object. Having the system call a proper C++ member function would have been much nicer. However, the callback interface has to be usable from many languages, so this is what we get: a `static` member function. The `Address` arguments specify that `cb_next()` takes arguments that are addresses of “something in memory.” C++ references are unknown to most languages, so we can’t use those. The compiler isn’t told what the types of those “somethings” are. We are close to the hardware here and don’t get the usual help from the language. “The system” will invoke a callback function with the first argument being the address of the GUI entity (`Widget`) for which the callback was triggered. We won’t use that first argument, so we don’t bother to name it. The second argument is the address of the window containing that `Widget`; for `cb_next()`, that will be our `Simple_window`. We can use that information like this:

```
void Simple_window::cb_next(Address, Address pw)
// call Simple_window::next() for the window located at pw
{
    reference_to<Simple_window>(pw).next();
}
```

The `reference_to<Simple_window>(pw)` tells the compiler that the address in `pw` is to be considered the address of a `Simple_window`; that is, we can use `reference_to<Simple_window>(pw)` as a reference to a `Simple_window`. In

Chapters 17 and 18, we will return to the issue of addressing memory. In §E.1, we present the (by then, trivial) definition of `reference_to`. For now, we are just glad that we finally obtained a reference to our `Simple_window` so that we can access our data and functions exactly as we like and are used to. Finally, we get out of this system-dependent code as quickly as possible by calling our member function `next()`.

We could have written all the code we wanted to execute in `cb_next()`, but we – like most good GUI programmers – prefer to keep messy low-level stuff separate from our nice user code, so we handle a callback with two functions:

- `cb_next()` simply maps the system conventions for a callback into a call to an ordinary member function (`next()`).
- `next()` does what we want done (without having to know about the messy conventions of callbacks).

The fundamental reason for using two functions here is the general principle that “a function should perform a single logical action”: `cb_next()` gets us out of the low-level system-dependent part of the system and `next()` performs our desired action. Whenever we want a callback (from “the system”) to one of our windows, we define such a pair of functions; for example, see §16.5–7. Before going further, let’s repeat what is going on here:

- We define our `Simple_window`.
- `Simple_window`’s constructor registers its `next_button` with the GUI system.
- When we click the image of `next_button` on the screen, the GUI calls `cb_next()`.
- `cb_next()` converts the low-level system information into a call of our member function `next()` for our window.
- `next()` performs whatever action we want done in response to the button click.

That’s a rather elaborate way of getting a function called. But remember that we are dealing with the basic mechanism for communicating an action of a mouse (or other hardware device) to a program. In particular:

- There are typically many programs running.
- The program is written long after the operating system.
- The program is written long after the GUI library.
- The program can be written in a language that is different from that used in the operating system.

- The technique deals with all kinds of interactions (not just our little button push).
- A window can have many buttons; a program can have many windows.

However, once we understand how **next()** is called, we basically understand how to deal with every action in a program with a GUI interface.

16.3.2 A wait loop

So, in this – our simplest – case, what do we want done by **Simple_window**'s **next()** each time the button is “pressed”? Basically, we want an operation that stops the execution of our program at some point, giving us a chance to see what has been done so far. And, we want **next()** to restart our program after that wait:

```
// create some objects and/or manipulate some objects, display them in a window
win.wait_for_button();      // next() causes the program to proceed from here
// create some objects and/or manipulate some objects
```

Actually, that's easily done. Let's first define **wait_for_button()**:

```
void Simple_window::wait_for_button()
    // modified event loop:
    // handle all events (as per default), quit when button_pushed becomes true
    // this allows graphics without control inversion
{
    while (!button_pushed) Fl::wait();
    button_pushed = false;
    Fl::redraw();
}
```

Like most GUI systems, FLTK provides a function that stops a program until something happens. The FLTK version is called **wait()**. Actually, **wait()** takes care of lots of things because our program gets woken up whenever anything that affects it happens. For example, when running under Microsoft Windows, it is the job of a program to redraw its window when it is being moved or becomes visible after having been hidden by another window. It is also the job of the **Window** to handle resizing. The **Fl::wait()** handles all of these tasks in the default manner. Each time **wait()** has dealt with something, it returns to give our code a chance to do something.



So, when someone clicks our “Next” button, `wait()` calls `cb_next()` and returns (to our “wait loop”). To proceed in `wait_for_button()`, `next()` just has to set the Boolean variable `button_pushed` to `true`. That’s easy:

```
void Simple_window::next()
{
    button_pushed = true;
}
```

Of course we also need to define `button_pushed` somewhere:

```
bool button_pushed; // initialized to false in the constructor
```

After waiting, `wait_for_button()` needs to reset `button_pushed` and `redraw()` the window to make sure that any changes we made can be seen on the screen. So that’s what it did.

16.3.3 A lambda expression as a callback

So for each action on a `Widget`, we have to define two functions: one to map from the system’s notion of a callback and one to do our desired action. Consider:

```
struct Simple_window : Graph_lib::Window {
    Simple_window{Point xy, int w, int h, const string& title};

    void wait_for_button(); // simple event loop
private:
    Button next_button; // the “Next” button
    bool button_pushed; // implementation detail

    static void cb_next(Address, Address); // callback for next_button
    void next(); // action to be done when next_button is pressed
};
```

By using a lambda expression (§15.3.3), we can eliminate the need to explicitly declare the mapping function `cb_next()`. Instead, we define the mapping in `Simple_window`’s constructor:

```
Simple_window::Simple_window(Point xy, int w, int h, const string& title)
    :Window{xy,w,h,title},
    next_button{Point{x_max()-70,0}, 70, 20, "Next",
        [](Address, Address pw) { reference_to<Simple_window>
            (pw).next(); }
```

```

},
    button_pushed{false}
{
    attach(next_button);
}

```

16.4 Button and other Widgets

We define a **Button** like this:

```

struct Button : Widget {
    Button(Point xy, int w, int h, const string& label, Callback cb);
    void attach(Window&);
};

```

So, a **Button** is a **Widget** with a location (**xy**), a size (**w,h**), a text label (**label**), and a callback (**cb**). Basically, anything that appears on a screen with an action (e.g., a callback) associated is a **Widget**.



16.4.1 Widgets

Yes, *widget* really is a technical term. A more descriptive, but less evocative, name for a widget is a *control*. We use widgets to define forms of interaction with a program through a GUI (graphical user interface). Our **Widget** interface class looks like this:

```

class Widget {
    // Widget is a handle to an FL_widget — it is *not* an FL_widget
    // we try to keep our interface classes at arm's length from FLTK
public:
    Widget(Point xy, int w, int h, const string& s, Callback cb);

    virtual void move(int dx,int dy);
    virtual void hide();
    virtual void show();
    virtual void attach(Window&) = 0;

    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;
}

```

protected:

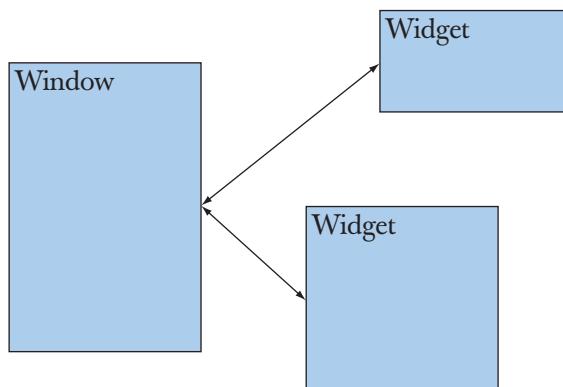
```
Window* own; // every Widget belongs to a Window
FL_Widget* pw; // connection to the FLTK Widget
};
```

A **Widget** has two interesting functions that we can use for **Button** (and also for any other class derived from **Widget**, e.g., a **Menu**; see §16.7):

- **hide()** makes the **Widget** invisible.
- **show()** makes the **Widget** visible again.

A **Widget** starts out visible.

Just like a **Shape**, we can **move()** a **Widget** in its **Window**, and we must **attach()** it to a **Window** before it can be used. Note that we declared **attach()** to be a pure virtual function (§14.3.5): every class derived from **Widget** must define what it means for it to be attached to a **Window**. In fact, it is in **attach()** that the system-level widgets are created. The **attach()** function is called from **Window** as part of its implementation of **Window**'s own **attach()**. Basically, connecting a window and a widget is a delicate little dance where each has to do its own part. The result is that a window knows about its widgets and that each widget knows about its window:



Note that a **Window** doesn't know what kind of **Widgets** it deals with. As described in §14.4, we are using basic object-oriented programming to ensure that a **Window** can deal with every kind of **Widget**. Similarly, a **Widget** doesn't know what kind of **Window** it deals with.

We have been slightly sloppy, leaving data members accessible. The **own** and **pw** members are strictly for the implementation of derived classes so we have declared them **protected**.

The definitions of **Widget** and of the widgets we use here (**Button**, **Menu**, etc.) are found in **GUI.h**.

16.4.2 Buttons

A **Button** is the simplest **Widget** we deal with. All it does is to invoke a callback when we click on it:

```
class Button : public Widget {  
public:  
    Button(Point xy, int ww, int hh, const string& s, Callback cb)  
        :Widget{xy,ww,hh,s,cb} {}  
  
    void attach(Window& win);  
};
```

That's all. The **attach()** function contains all the (relatively) messy FLTK code. We have banished the explanation to Appendix E (not to be read until after Chapters 17 and 18). For now, please just note that defining a simple **Widget** isn't particularly difficult.

We do not deal with the somewhat complicated and messy issue of how buttons (and other **Widgets**) look on the screen. The problem is that there is a near infinity of choices and that some styles are mandated by certain systems. Also, from a programming technique point of view, nothing really new is needed for expressing the looks of buttons. If you get desperate, we note that placing a **Shape** on top of a button doesn't affect the button's ability to function – and you know how to make a shape look like anything at all.

16.4.3 In_box and Out_box

We provide two **Widgets** for getting text in and out of our program:

```
struct In_box : Widget {  
    In_box(Point xy, int w, int h, const string& s)  
        :Widget{xy,w,h,s,0} {}  
    int get_int();  
    string get_string();  
  
    void attach(Window& win);  
};
```

```
struct Out_box : Widget {
    Out_box(Point xy, int w, int h, const string& s)
        :Widget{xy,w,h,s,0} {}
    void put(int);
    void put(const string&);

    void attach(Window& win);
};
```

An **In_box** can accept text typed into it, and we can read that text as a string using **get_string()** or as an integer using **get_int()**. If you want to know if text has been entered, you can read using **get_string()** and see if you get the empty string:

```
string s = some_inbox.get_string();
if (s == "") {
    // deal with missing input
}
```

An **Out_box** is used to present some message to a user. In analogy to **In_box**, we can **put()** either integers or strings. §16.5 gives examples of the use of **In_box** and **Out_box**.

We could have provided **get_floating_point()**, **get_complex()**, etc., but we did not bother because you can take the string, stick it into a **stringstream**, and do any input formatting you like that way (§11.4).

16.4.4 Menus

We offer a very simple notion of a menu:

```
struct Menu : Widget {
    enum Kind { horizontal, vertical };
    Menu(Point xy, int w, int h, Kind kk, const string& label);
    Vector_ref<Button> selection;
    Kind k;
    int offset;
    int attach(Button& b);           // attach Button to Menu
    int attach(Button* p);          // attach new Button to Menu

    void show()                  // show all buttons
    {
        for (Button& b : selection) b.show();
    }
```

```

void hide();           // hide all buttons
void move(int dx, int dy); // move all buttons

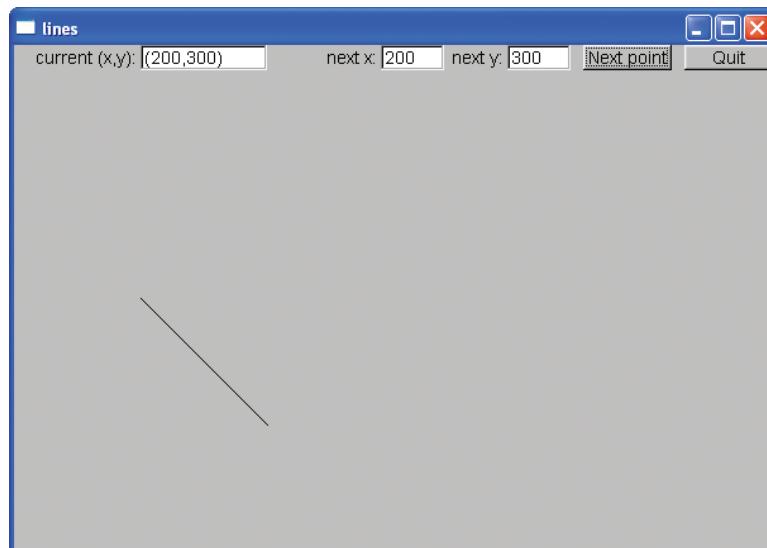
void attach(Window& win); // attach all buttons to Window win
};

```

A **Menu** is basically a vector of buttons. As usual, the **Point xy** is the top left corner. The width and height are used to resize buttons as they are added to the menu. For examples, see §16.5 and §16.7. Each menu button (“a menu item”) is an independent **Widget** presented to the **Menu** as an argument to **attach()**. In turn, **Menu** provides an **attach()** operation to attach all of its **Buttons** to a **Window**. The **Menu** keeps track of its **Buttons** using a **Vector_ref** (§13.10, §E.4). If you want a “pop-up” menu, you have to make it yourself; see §16.7.

16.5 An example

To get a better feel for the basic GUI facilities, consider the window for a simple application involving input, output, and a bit of graphics:

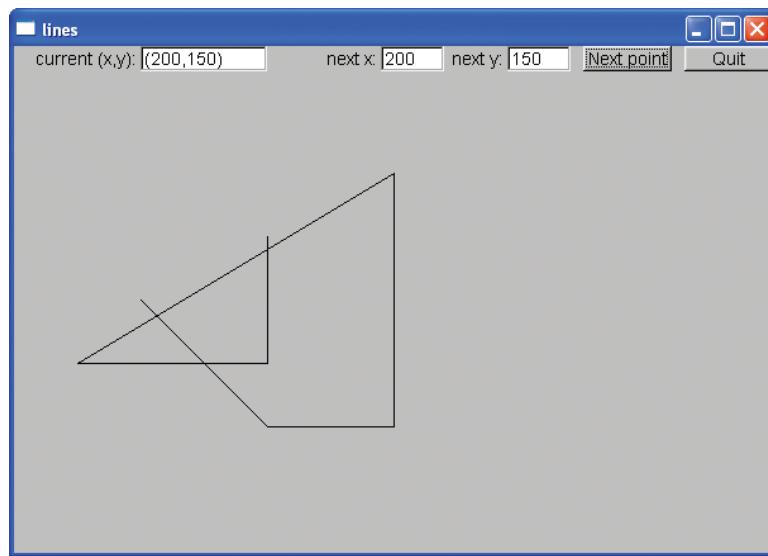


This program allows a user to display a sequence of lines (an open polyline; §13.6) specified as a sequence of coordinate pairs. The idea is that the user repeatedly enters (x,y) coordinates in the “next x” and “next y” boxes; after each pair the user hits the “Next point” button.

Initially, the “current (x,y) ” box is empty and the program waits for the user to enter the first coordinate pair. That done, the starting point appears in the

“current (x,y)” box, and each new coordinate pair entered results in a line being drawn: a line from the current point (which has its coordinates displayed in the “current (x,y)” box) to the newly entered (x,y) is drawn, and that (x,y) becomes the new current point.

This draws an open polyline. When the user tires of this activity, there is the “Quit” button for exiting. That’s pretty straightforward, and the program exercises several useful GUI facilities: text input and output, line drawing, and multiple buttons. The window above shows the result after entering two coordinate pairs; after seven we can get this:



Let’s define a class for representing such windows. It is pretty straightforward:

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
    Open_polyline lines;
private:
    Button next_button;      // add (next_x,next_y) to lines
    Button quit_button;
    In_box next_x;
    In_box next_y;
    Out_box xy_out;

    void next();
    void quit();
};
```

The line is represented as an **Open_polyline**. The buttons and boxes are declared (as **Buttons**, **In_boxes**, and **Out_boxes**), and for each button a member function implementing the desired action is defined. We decided to eliminate the “boilerplate” callback function and use lambdas instead.

Lines_window's constructor initializes everything:

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    :Window{xy,w,h,title},
    next_button{Point{x_max()-150,0}, 70, 20, "Next point",
        [](Address, Address pw) {reference_to<Lines_window>(pw).next();},
    quit_button{Point{x_max()-70,0}, 70, 20, "Quit",
        [](Address, Address pw) {reference_to<Lines_window>(pw).quit();},
    next_x{Point{x_max()-310,0}, 50, 20, "next x:"},
    next_y{Point{x_max()-210,0}, 50, 20, "next y:"},
    xy_out{Point{100,0}, 100, 20, "current (x,y):"}
{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    attach(lines);
}
```

That is, each widget is constructed and then attached to the window.

The “Quit” button deletes the **Window**. That's done using the curious FLTK idiom of simply hiding it:

```
void Lines_window::quit()
{
    hide();      // curious FLTK idiom to delete window
}
```

All the real work is done in the “Next point” button: it reads a pair of coordinates, updates the **Open_polyline**, updates the position readout, and redraws the window:

```
void Lines_window::next()
{
    int x = next_x.get_int();
    int y = next_y.get_int();
```

```

    lines.add(Point{x,y});

    // update current position readout:
    ostringstream ss;
    ss << '(' << x << ',' << y << ')';
    xy_out.put(ss.str());

    redraw();
}

```

That's all pretty obvious. We get integer coordinates from the **In_boxes** using **get_int()**. We use an **ostringstream** to format the string to be put into the **Out_box**; the **str()** member function lets us get to the string within the **ostringstream**. The final **redraw()** here is needed to present the results to the user; until a **Window**'s **redraw()** is called, the old image remains on the screen.

So what's odd and different about this program? Let's see its **main()**:

```

#include "GUI.h"

int main()
try {
    Lines_window win {Point{100,100},600,400,"lines"};
    return gui_main();
}
catch(exception& e) {
    cerr << "exception: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Some exception\n";
    return 2;
}

```

There is basically nothing there! The body of **main()** is just the definition of our window, **win**, and a call to a function **gui_main()**. There is not another function, **if**, **switch**, or loop – nothing of the kind of code we saw in Chapters 6 and 7 – just a definition of a variable and a call to the function **gui_main()**, which is itself just a call of FLTK's **run()**. Looking further, we can find that **run()** is simply the infinite loop

```
while(wait());
```

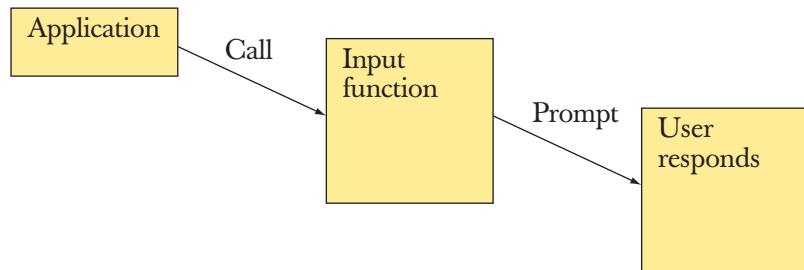
Except for a few implementation details postponed to Appendix E, we have seen all of the code that makes our “lines” program run. We have seen all of the fundamental logic. So what happens?



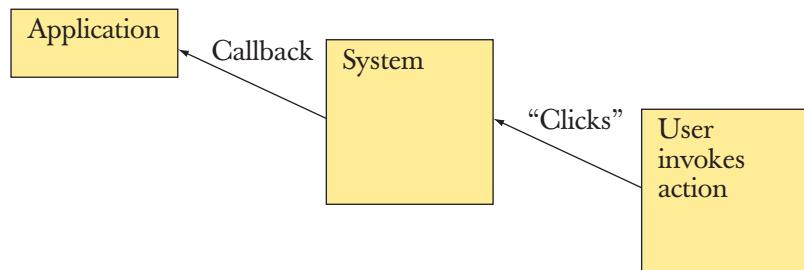
16.6 Control inversion

What happened was that we moved the control of the order of execution from the program to the widgets: whichever widget the user activates, runs. For example, click on a button and its callback runs. When that callback returns, the program settles back, waiting for the user to do something else. Basically, `wait()` tells “the system” to look out for the widgets and invoke the appropriate callbacks. In theory, `wait()` could tell you, the programmer, which widget requested attention and leave it to you to call the appropriate function. However, in FLTK and most other GUI systems, `wait()` simply invokes the appropriate callback, saving you the bother of writing code to select it.

A “conventional program” is organized like this:



A “GUI program” is organized like this:





One implication of this “control inversion” is that the order of execution is completely determined by the actions of the user. This complicates both program organization and debugging. It is hard to imagine what a user will do and hard to imagine every possible effect of a random sequence of callbacks. This makes systematic testing a nightmare (see Chapter 26). The techniques for dealing with that are beyond the scope of this book, but we encourage you to be extra careful with code driven by users through callbacks. In addition to the obvious control flow problems, there are also problems of visibility and difficulties with keeping track of which widget is connected to what data. To minimize hassle, it is essential to keep the GUI portion of a program simple and to build a GUI program incrementally, testing at each stage. When working on a GUI program, it is almost essential to draw little diagrams of the objects and their interactions.

How does the code triggered by the various callbacks communicate? The simplest way is for the functions to operate on data stored in the window, as was done in the example in §16.5. There, the `Lines_window`'s `next()` function, invoked by pressing the “Next point” button, reads data from the `In_boxes` (`next_x` and `next_y`) and updates the `lines` member variable and the `Out_box` (`xy_out`). Obviously, a function invoked by a callback can do anything: it could open files, connect to the web, etc. However, for now, we'll just consider the simple case in which we hold our data in a window.

16.7 Adding a menu

Let's explore the control and communication issues raised by “control inversion” by providing a menu for our “lines” program. First, we'll simply provide a menu that allows the user to change the color of all lines in the `lines` member variable. We add the menu `color_menu` and its callbacks:

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);

    Open_polyline lines;
    Menu color_menu;

    static void cb_red(Address, Address);      // callback for red button
    static void cb_blue(Address, Address);     // callback for blue button
    static void cb_black(Address, Address);    // callback for black button

    // the actions:
    void red_pressed() { change(Color::red); }
    void blue_pressed() { change(Color::blue); }
    void black_pressed() { change(Color::black); }
```

```

void change(Color c) { lines.set_color(c); }

// . . . as before . . .
};

```

Writing all of those almost identical callback functions and “action” functions is tedious. However, it is conceptually simple, and offering something that’s significantly simpler to type in is beyond the scope of this book. If you prefer, you can eliminate the **cb_** functions by using lambdas (§16.3.3). When a menu button is pressed, it changes the lines to the requested color.

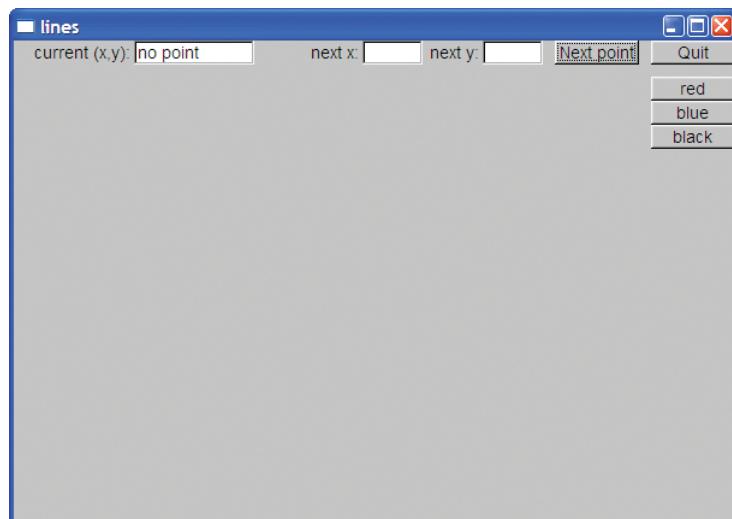
Having defined the **color_menu** member, we need to initialize it:

```

Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    :Window(xy,w,h,title),
// . . . as before . . .
    color_menu{Point{x_max()-70,40},70,20,Menu::vertical,"color"}
{
    // . . . as before . . .
    color_menu.attach(new Button{Point{0,0},0,0,"red",cb_red});
    color_menu. attach(new Button{Point{0,0},0,0,"blue",cb_blue});
    color_menu. attach(new Button{Point{0,0},0,0,"black",cb_black});
    attach(color_menu);
}

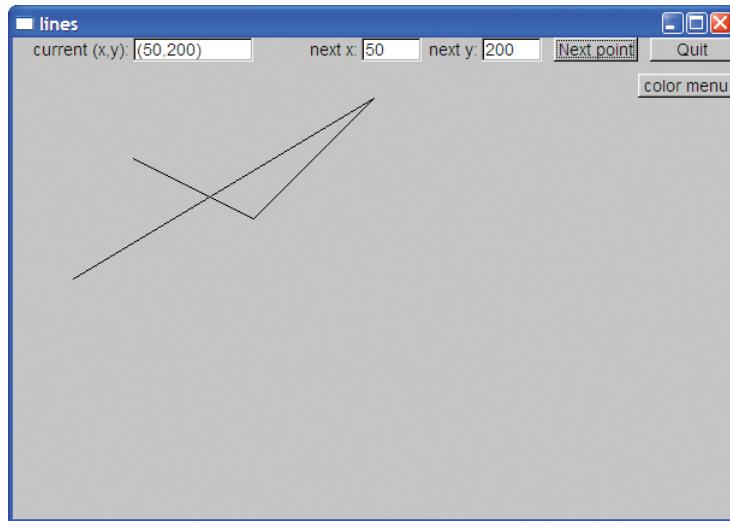
```

The buttons are dynamically attached to the menu (using **attach()**) and can be removed and/or replaced as needed. **Menu::attach()** adjusts the size and location of the button and attaches it to the window. That’s all, and we get

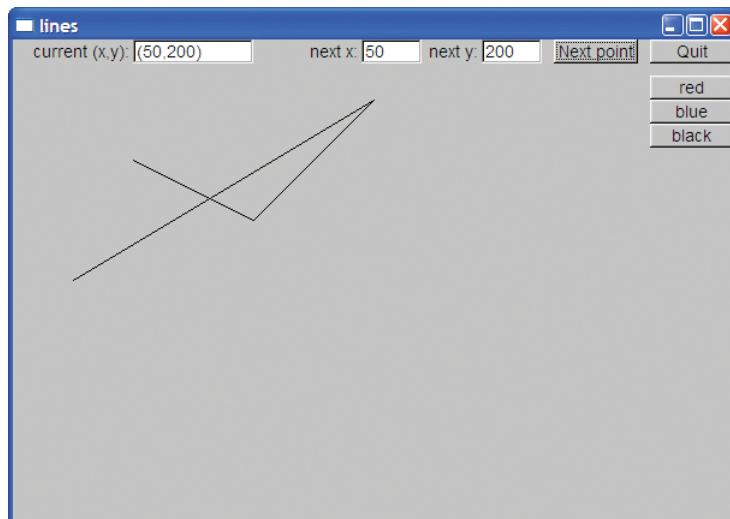


Having played with this for a while, we decided that what we really wanted was a “pop-up menu”; that is, we didn’t want to spend precious screen space on a menu except when we are using it. So, we added a “color menu” button. When we press that, up pops the color menu, and when we have made a selection, the menu is again hidden and the button appears.

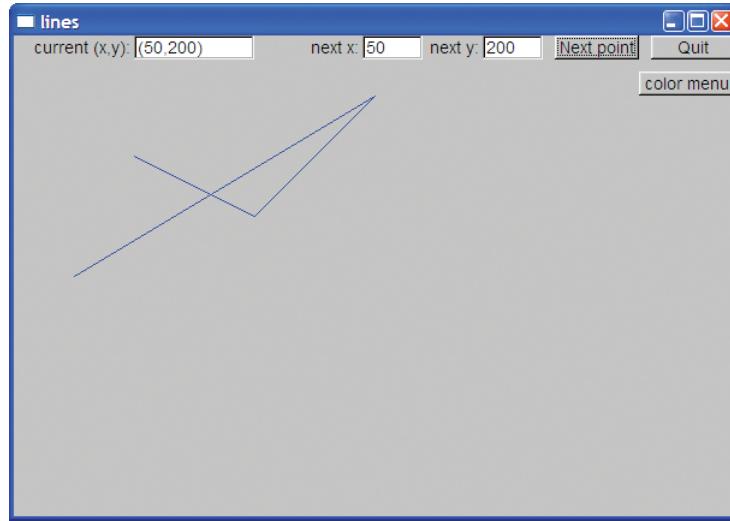
Here first is the window after we have added a few lines:



We see the new “color menu” button and some (black) lines. Press “color menu” and the menu appears:



Note that the “color menu” button is now hidden. We don’t need it until we are finished with the menu. Press “blue” and we get



The lines are now blue and the “color menu” button has reappeared.

To achieve this we added the “color menu” button and modified the “pressed” functions to adjust the visibility of the menu and the button. Here is the complete **Lines_window** after all of our modifications:

```

struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
private:
    // data:
    Open_polyline lines;

    // widgets:
    Button next_button;      // add (next_x,next_y) to lines
    Button quit_button;      // end program
    In_box next_x;
    In_box next_y;
    Out_box xy_out;
    Menu color_menu;
    Button menu_button;

    void change(Color c) { lines.set_color(c); }

    void hide_menu() { color_menu.hide(); menu_button.show(); }
  
```

```

// actions invoked by callbacks:
void red_pressed() { change(Color::red); hide_menu(); }
void blue_pressed() { change(Color::blue); hide_menu(); }
void black_pressed() { change(Color::black); hide_menu(); }
void menu_pressed() { menu_button.hide(); color_menu.show(); }
void next();
void quit();

// callback functions:
static void cb_red(Address, Address);
static void cb_blue(Address, Address);
static void cb_black(Address, Address);
static void cb_menu(Address, Address);
static void cb_next(Address, Address);
static void cb_quit(Address, Address);

};


```

Note how all but the constructor is private. Basically, that **Window** class is the program. All that happens, happens through its callbacks, so no code from outside the window is needed. We sorted the declarations a bit hoping to make the class more readable. The constructor provides arguments to all of its sub-objects and attaches them to the window:

```

Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    :Window{xy,w,h,title},
    next_button{Point{x_max()-150,0}, 70, 20, "Next point", cb_next},
    quit_button{Point{x_max()-70,0}, 70, 20, "Quit", cb_quit},
    next_x{Point{x_max()-310,0}, 50, 20, "next x:"},
    next_y{Point{x_max()-210,0}, 50, 20, "next y:"},
    xy_out{Point{100,0}, 100, 20, "current (x,y):"},
    color_menu{Point{x_max()-70,30},70,20,Menu::vertical,"color"},
    menu_button{Point{x_max()-80,30}, 80, 20, "color menu", cb_menu}

{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    xy_out.put("no point");
    color_menu.attach(new Button{Point{0,0},0,0,"red",cb_red});
    color_menu.attach(new Button{Point{0,0},0,0,"blue",cb_blue});
    color_menu.attach(new Button{Point{0,0},0,0,"black",cb_black});
    attach(color_menu);
}

```

```
color_menu.hide();
attach(menu_button);
attach(lines);
}
```

Note that the initializers are in the same order as the data member definitions. That's the proper order in which to write the initializers. In fact, member initializers are always executed in the order their data members were declared. Some compilers (helpfully) give a warning if a base or member constructor is specified out of order.



16.8 Debugging GUI code

Once a GUI program starts working it is often quite easy to debug: what you see is what you get. However, there is often a most frustrating period before the first shapes and widgets start appearing in a window or even before a window appears on the screen. Try this `main()`:

```
int main()
{
    Lines_window {Point{100,100},600,400,"lines"};
    return gui_main();
}
```

Do you see the error? Whether you see it or not, you should try it; the program will compile and run, but instead of the `Lines_window` giving you a chance to draw lines, you get at most a flicker on the screen. How do you find errors in such a program?



- By carefully using well-tried program parts (classes, function, libraries)
- By simplifying all new code, by slowly “growing” a program from its simplest version, by carefully looking over the code line by line
- By checking all linker settings
- By comparing the code to already working programs
- By explaining the code to a friend

The one thing that you will find it hard to do is to trace the execution of the code. If you have learned to use a debugger, you have a chance, but just inserting “output statements” will not work in this case – the problem is that no output appears. Even debuggers will have problems because there are several things going on at once (“multi-threading”) – your code is not the only code trying to interact with the screen. Simplification of the code and a systematic approach to understanding the code are key.



So what was the problem? Here is the correct version (from §16.5):

```
int main()
{
    Lines_window win{Point{100,100},600,400,"lines"};
    return gui_main();
}
```

We “forgot” the name of the **Lines_window**, **win**. Since we didn’t actually need that name that seemed reasonable, but the compiler then decided that since we didn’t use that window, it could immediately destroy it. Oops! That window existed for something on the order of a millisecond. No wonder we missed it.

Another common problem is to put one window *exactly* on top of another. This obviously (or rather not at all obviously) looks as if there is only one window. Where did the other window go? We can spend significant time looking for nonexistent bugs in the code. The same problem can occur if we put one shape on top of another.

Finally – to make matters still worse – exceptions don’t always work as we would like them to when we use a GUI library. Since our code is managed by a GUI library, an exception we throw may never reach our handler – the library or the operating system may “eat” it (that is, they may rely on error-handling mechanisms that differ from C++ exceptions and may indeed be completely oblivious of C++).

Common problems found during debugging include **Shapes** and **Widgets** not showing because they were not attached and objects misbehaving because they have gone out of scope. Consider how a programmer might factor out the creation and attachment of buttons in a menu:

```
// helper function for loading buttons into a menu
void load_disaster_menu(Menu& m)
{
    Point orig {0,0};
    Button b1 {orig,0,0,"flood",cb_flood};
    Button b2 {orig,0,0,"fire",cb_fire};
    // ...
    m.attach(b1);
    m.attach(b2);
    // ...
}
```

```
int main()
{
    // ...
    Menu disasters {Point{100,100},60,20,Menu::horizontal,"disasters"};
    load_disaster_menu(disasters);
    win.attach(disasters);
    // ...
}
```

This will not work. All those buttons are local to the `load_disaster_menu` function and attaching them to a menu will not change that. An explanation can be found in §18.6.4 (*Don't return a pointer to a local variable*), and an illustration of the memory layout for local variables is presented in §8.5.8. The essence of the story is that after `load_disaster_menu()` has returned, those local objects have been destroyed and the `disasters` menu refers to nonexistent (destroyed) objects. The result is likely to be surprising and not pretty. The solution is to use unnamed objects created by `new` instead of named local objects:

```
// helper function for loading buttons into a menu
void load_disaster_menu(Menu& m)
{
    Point orig {0,0};
    m.attach(new Button{orig,0,0,"flood",cb_flood});
    m.attach(new Button{orig,0,0,"fire",cb_fire});
    // ...
}
```

The correct solution is even simpler than the (all too common) bug.



Drill

1. Make a completely new project with linker settings for FLTK (as described in Appendix D).
2. Using the facilities of `Graph_lib`, type in the line-drawing program from §16.5 and get it to run.
3. Modify the program to use a pop-up menu as described in §16.7 and get it to run.
4. Modify the program to have a second menu for choosing line styles and get it to run.

Review

1. Why would you want a graphical user interface?
2. When would you want a non-graphical user interface?
3. What is a software layer?
4. Why would you want to layer software?
5. What is the fundamental problem when communicating with an operating system from C++?
6. What is a callback?
7. What is a widget?
8. What is another name for widget?
9. What does the acronym FLTK mean?
10. How do you pronounce FLTK?
11. What other GUI toolkits have you heard of?
12. Which systems use the term *widget* and which prefer *control*?
13. What are examples of widgets?
14. When would you use an inbox?
15. What is the type of the value stored in an inbox?
16. When would you use a button?
17. When would you use a menu?
18. What is control inversion?
19. What is the basic strategy for debugging a GUI program?
20. Why is debugging a GUI program harder than debugging an “ordinary program using streams for I/O”?

Terms

button	dialog box	visible/hidden
callback	GUI	waiting for input
console I/O	menu	wait loop
control	software layer	widget
control inversion	user interface	

Exercises

1. Make a **My_window** that’s a bit like **Simple_window** except that it has two buttons, **next** and **quit**.
2. Make a window (based on **My_window**) with a 4-by-4 checkerboard of square buttons. When pressed, a button performs a simple action, such as printing its coordinates in an output box, or turns a slightly different color (until another button is pressed).

3. Place an **Image** on top of a **Button**; move both when the button is pushed. Use this random number generator from **std_lib_facilities.h** to pick a new location for the “image button”:

```
#include<random>

inline int rand_int(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>{min,max}(ran);
}
```

It returns a random **int** in the range [**min,max**]).

4. Make a menu with items that make a circle, a square, an equilateral triangle, and a hexagon, respectively. Make an input box (or two) for giving a coordinate pair, and place the shape made by pressing a menu item at that coordinate. Sorry, no drag and drop.
5. Write a program that draws a shape of your choice and moves it to a new point each time you click “Next.” The new point should be determined by a coordinate pair read from an input stream.
6. Make an “analog clock,” that is, a clock with hands that move. You get the time of day from the operating system through a library call. A major part of this exercise is to find the functions that give you the time of day and a way of waiting for a short period of time (e.g., a second for a clock tick) and to learn to use them based on the documentation you found. Hint: **clock()**, **sleep()**.
7. Using the techniques developed in the previous exercises, make an image of an airplane “fly around” in a window. Have a “Start” and a “Stop” button.
8. Provide a currency converter. Read the conversion rates from a file on startup. Enter an amount in an input window and provide a way of selecting currencies to convert to and from (e.g., a pair of menus).
9. Modify the calculator from Chapter 7 to get its input from an input box and return its results in an output box.
10. Provide a program where you can choose among a set of functions (e.g., **sin()** and **log()**), provide parameters for those functions, and then graph them.

Postscript

GUI is a huge topic. Much of it has to do with style and compatibility with existing systems. Furthermore, much has to do with a bewildering variety of widgets (such as a GUI library offering many dozens of alternative button styles) that

would make a traditional botanist feel quite at home. However, little of that has to do with fundamental programming techniques, so we won't proceed in that direction. Other topics, such as scaling, rotation, morphing, three-dimensional objects, shadowing, etc., require sophistication in graphical and/or mathematical topics which we don't assume here.

One thing you should be aware of is that most GUI systems provide a "GUI builder" that allows you to design your window layouts graphically and attach callbacks and actions to buttons, menus, etc. specified graphically. For many applications, such a GUI builder is well worth using to reduce the tedium of writing "scaffolding code" such as our callbacks. However, always try to understand how the resulting programs work. Sometimes, the generated code is equivalent to what you have seen in this chapter. Sometimes more elaborate and/or expensive mechanisms are used.

Part III

Data and Algorithms



Vector and Free Store

“Use **vector** as the default!”

—Alex Stepanov

This chapter and the next four 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 two is the design and implementation of the most common and most useful STL container: **vector**.

17.1 Introduction	17.6 Access to elements
17.2 <code>vector</code> basics	17.7 Pointers to class objects
17.3 Memory, addresses, and pointers	17.8 Messing with types: <code>void*</code> and casts
17.3.1 The <code>sizeof</code> operator	17.9 Pointers and references
17.4 Free store and pointers	17.9.1 Pointer and reference parameters
17.4.1 Free-store allocation	17.9.2 Pointers, references, and inheritance
17.4.2 Access through pointers	17.9.3 An example: lists
17.4.3 Ranges	17.9.4 List operations
17.4.4 Initialization	17.9.5 List use
17.4.5 The null pointer	
17.4.6 Free-store deallocation	
17.5 Destructors	17.10 The <code>this</code> pointer
17.5.1 Generated destructors	17.10.1 More link use
17.5.2 Destructors and free store	

17.1 Introduction

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()`, 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 will 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>`, the operation `v.push_back(2.3)` adds 2.3 to a sequence of `doubles` 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 two chapters, we show how to build `vector` from the basic language facilities available to every programmer. Doing so 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 the `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 (Chapters 20 and 21). These facilities, called algorithms, 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 libraries. We have already seen and used one of the standard library's most useful algorithms: `sort()`.

We approach the standard library `vector` through a series of increasingly sophisticated `vector` implementations. First, we build a very simple `vector`. Then, we see what's undesirable about that `vector` and fix it. When we have done that a few times, we reach a `vector` implementation that is roughly equivalent to 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 closely 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 17 (this chapter):* How can we deal with varying amounts of memory? In particular, how can different `vectors` have different numbers of elements and how can a single `vector` have different numbers of elements at different times? This leads us to examine free store (heap storage), pointers, casts (explicit type conversion), and references.
- *Chapter 18:* How can we copy `vectors`? How can we provide a subscript operation for them? We also introduce arrays and explore their relation to pointers.
- *Chapter 19:* How can we have `vectors` 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.

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 will also (re)use many of the language facilities and programming techniques we have already seen. Occasionally, we'll 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 allows our computer's memory to function.

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, you would not just have to “believe in magic”; you would be unable to implement a new container (should you need one, and that’s not uncommon). Also, you would be unable to read huge amounts of C and 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.

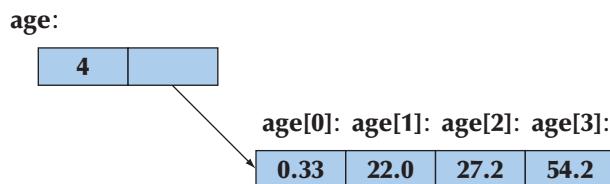
More philosophically, I am among the large group of computer professionals who are of the opinion that if you lack a basic and practical understanding of how a program maps onto a computer’s memory and operations, you will have problems getting a solid grasp of higher-level topics, such as data structures,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

17.2 vector basics

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

```
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;
```

Obviously, 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 a **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, age0, age1, age2, age3;
    // ...
};
```

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

	age:
size:	age[0]: age[1]: age[2]: age[3]:
4	0.33 22.0 27.2 54.2

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 **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:

```
// a very simplified vector of doubles (like vector<double>)
class vector {
    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
                                         // store s in sz
    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.”

17.3 Memory, addresses, and pointers



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:



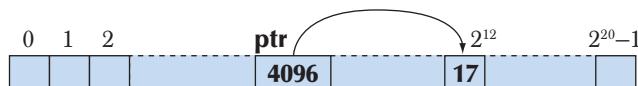
Everything we put in memory has an address. For example:

```
int var = 17;
```

This will set aside an "int-size" 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; hexadecimal notation (§A.2.1.1) is popular for pointer values.

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)
```

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 and logical) 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*
```

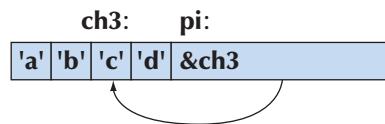
Similarly, a pointer to **char** (a **char***) is not a pointer to **int** (an **int***). For example:

```
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;   // point to ch3, a char-size piece of memory
                  // error: we cannot assign a char* to an int*
                  // but let's pretend we could
*pi = 12345;     // write to an int-size 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, but this is one of the very few protections offered by the compiler at this low level of programming.

In the unlikely case that you really need to convert an **int** to a pointer or to convert one pointer type to another, you can use **reinterpret_cast**; see §17.8.

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” (see Chapter 25). 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 Chapters 18–19, we show how to get back to a more comfortable level of abstraction by implementing a **vector**.

17.3.1 The **sizeof** operator

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

```

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 `ints` and high-performance architectures with 64-bit `ints` 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

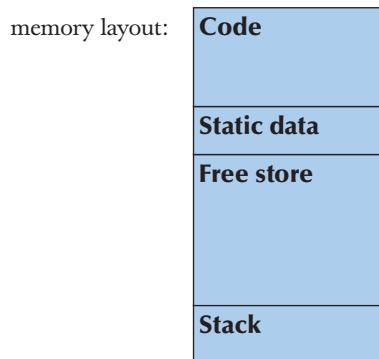
The explanation will become obvious over this chapter and the next (see also §19.2.1), but clearly, `sizeof` is not counting the elements.

17.4 Free store and pointers

Consider the implementation of `vector` from the end of §17.2. 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 (that's called *stack storage* or *automatic storage*). The rest of the computer's



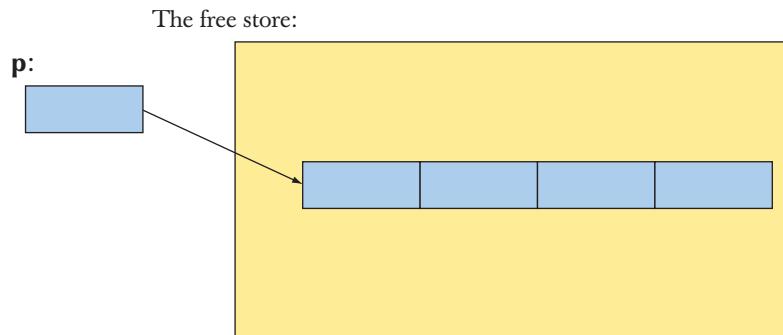
memory is potentially available for other uses; it is “free.” We can illustrate that graphically:



The C++ language makes this “free store” (also called the *heap*) 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), 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: double* assigned to 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**.

17.4.1 Free-store allocation

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

- 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.

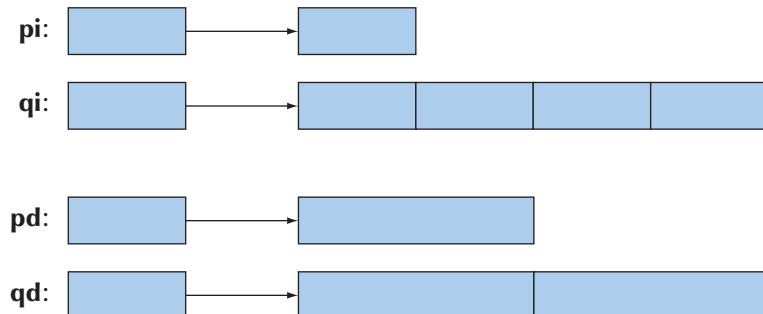


The **new** operator can allocate individual elements or sequences (arrays) of elements. For example:

```
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 not? After all, we can assign an **int** to a **double** and a **double** to an **int**. The reason is the **[]** operator. It relies on the size of the element type to figure out where to find an element. For example, **qi[2]** is two **int** sizes further on in memory than **qi[0]**, and **qd[2]** is two **double** sizes further on in memory than **qd[0]**. If the size of an **int** is different from the size of **double**, as it is on many computers, 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."

17.4.2 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[2];          // read the 3rd object pointed to by p
```

Unsurprisingly, the subscript operator 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:

<code>p[0]:</code>	<code>p[1]:</code>	<code>p[2]:</code>	<code>p[3]:</code>
8.8		9.9	4.4

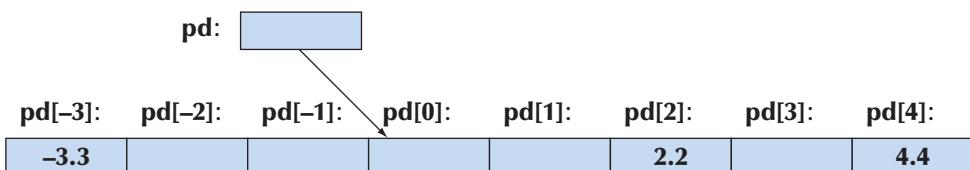
This is exactly the simple and optimally efficient mechanism for accessing memory that we need to implement a `vector`.

17.4.3 Ranges

The major problem with pointers is that a pointer doesn't "know" how many elements it points to. Consider:

```
double* pd = new double[3];
pd[2] = 2.2;
pd[4] = 4.4;
pd[-3] = -3.3;
```

Does `pd` have a third element `pd[2]`? Does it have a fifth element `pd[4]`? If we look at the definition of `pd`, 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[-3]` as if the location three **doubles** before what `pd` points to was part of our allocation:



We have no idea what the memory locations marked `pd[-3]` and `pd[4]` are used for. However, we do know that they weren't meant to be used as part of our array of three **doubles** 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 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.

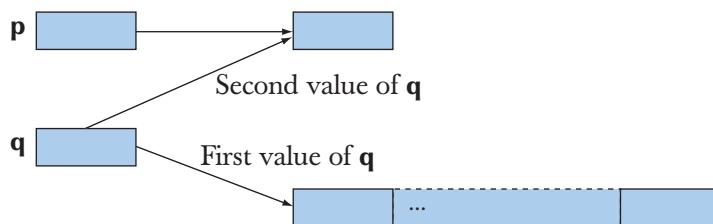
One thing that can make it hard to prevent out-of-range access is that we can assign one `double*` to another `double*` independently of how many objects

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 = p;                      // let q point to the same as p
double d = q[700];              // 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. 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.

17.4.4 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.

Memory allocated by **new** is not initialized for built-in types. If you don’t like that for a single object, you can specify a value, as we did for **p2**: ***p2 is 5.5**. Note the use of **()** for initialization. This contrasts to the use of **[]** to indicate “array.”

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.

When we define our own types, we have better control of initialization. If a type **X** has a default constructor, 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 typically the most common case.

17.4.5 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
```

When assigned to a pointer, the value zero is called the *null pointer*, and 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 `deleted` (see §17.4.6). 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
```



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?

The name `nullptr` for the null pointer is new in C++11, so 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`.

17.4.6 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) // leaks memory
{
```

```

double* p = new double[max];
double* res = new double[res_size];
// use p to calculate results to be put in res
return res;
}

double* r = calc(100,1000);

```

As written, each call of `calc()` “leaks” the `doubles` allocated for `p`. For example, the call `calc(100,1000)` will render the space needed for **1000 doubles** unusable for the rest of the program.

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. The example now becomes

```

double* calc(int res_size, int max)
// the caller is responsible for the memory allocated for res
{
    double* p = new double[max];
    double* res = new double[res_size];
// use p to calculate results to be put in res
    delete[] p; // we don't need that memory anymore: free it
    return res;
}

double* r = calc(100,1000);
// use r
delete[] r; // we don't need that memory anymore: free it

```

Incidentally, this example demonstrates one of the major reasons for using the free store: we can create objects in a function and pass them back to a caller.

There are two forms of `delete`:

- `delete p` frees the memory for an individual object allocated by `new`.
- `delete[] p` frees the memory for an array of objects allocated by `new`.

It is the programmer’s tedious job to use the right version.

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

```

int* p = new int{5};
delete p; // fine: p points to an object created by new
// . . . no use of p here . . .
delete p; // error: p points to memory owned by the free-store manager

```



There are two problems with the second `delete p`:

- You don't own the object pointed to anymore so the free-store manager may have changed its internal data structure in such a way that it can't correctly execute `delete p` again.
- The free-store manager may have "recycled" the memory pointed to by `p` so that `p` now points to another object; deleting that other object (owned by some other part of the program) will cause errors in your program.

Both problems occur in a real program; they are not just theoretical possibilities.

Deleting the null pointer doesn't do anything (because the null pointer doesn't point to an object), so deleting the null pointer is harmless. For example:

```
int* p = nullptr;
delete p;      // fine: no action needed
delete p;      // also fine (still no action needed)
```

Why do we have to bother with freeing memory? Can't the compiler figure out when we don't need a piece of memory anymore and just recycle it without human intervention? It can. That's called *automatic garbage collection* or just *garbage collection*. Unfortunately, automatic garbage collection is not cost-free and not ideal for all kinds of applications. If you really need automatic garbage collection, you can plug a garbage collector into your C++ program. Good garbage collectors are available (see www.stroustrup.com/C++.html). However, in this book we assume that you have to deal with your own "garbage," and we show how to do so conveniently and efficiently.

When is it important not to leak memory? A program that needs to run "forever" can't afford any memory leaks. An operating system is an example of a program that "runs forever," and so are most embedded systems (see Chapter 25). A library should not leak memory because someone might use it as part of a system that shouldn't leak memory. In general, it is simply a good idea not to leak. Many programmers consider leaks as proof of sloppiness. However, that's slightly overstating the point. When you run a program under an operating system (Unix, Windows, whatever), all memory is automatically returned to the system at the end of the program. It follows that if you know that your program will not use more memory than is available, you might reasonably decide to "leak" until the operating system does the deallocation for you. However, if you decide to do that, be sure that your memory consumption estimate is correct, or people will have good reason to consider you sloppy.

17.5 Destructors

Now we know how to store the elements for a **vector**. We simply allocate sufficient space for the elements on the free store and access them through a pointer:

```
// a very simplified vector of doubles
class vector {
    int sz;                                // the size
    double* elem;                         // a pointer to the elements
public:
    vector(int s)                          // constructor
        :sz{s},                            // initialize sz
        elem{new double[s]}              // initialize elem
    {
        for (int i=0; i<s; ++i) elem[i]=0; // initialize elements
    }
    int size() const { return sz; }          // the current size
    // ...
};
```

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**.

Note that we initialize the elements to their default value (**0.0**). The standard library **vector** does that, so we thought it best to do the same from the start.

Unfortunately, our first primitive **vector** leaks memory. In the constructor, it allocates memory for the elements using **new**. To follow the rule stated in §17.4, we must make sure that this memory is freed using **delete**. Consider:

```
void f(int n)
{
    vector v(n);           // allocate n doubles
    // ...
}
```

When we leave **f()**, the elements created on the free store by **v** are not freed. We could define a **clean_up()** operation for **vector** and call that:

```
void f2(int n)
{
```

```

vector v(n);           // define a vector (which allocates another n ints)
// . . . use v . . .
v.clean_up();          // clean_up() deletes elem
}

```

That would work. However, one of the most common problems with the free store is that people forget to **delete**. The equivalent problem would arise for **clean_up()**; people would forget to call it. We can do better than that. 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*. 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. For example:

```

// a very simplified vector of doubles
class vector {
    int sz;                      // the size
    double* elem;                // a pointer to the elements
public:
    vector(int s)                // constructor
        :sz{s}, elem{new double[s]} // allocate memory
    {
        for (int i=0; i<s; ++i) elem[i]=0; // initialize elements
    }

    ~vector()                    // destructor
        { delete[] elem; }          // free memory
    // ...
}

```

Given that, we can write

```

void f3(int n)
{
    double* p = new double[n];      // allocate n doubles
    vector v(n);                  // the vector allocates n doubles
    // . . . use p and v . . .
    delete[ ] p;                 // deallocate p's doubles
} // vector automatically cleans up after v

```

Suddenly, that `delete[]` looks rather tedious and error-prone! Given `vector`, there is no reason to allocate memory using `new` just to deallocate it using `delete[]` at the end of a function. That's what `vector` does and does better. In particular, a `vector` cannot forget to call its destructor to deallocate the memory used for the elements.

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 `iostreams` 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.

17.5.1 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;
    // initialize fred
    // 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.

The destructors for members – and for bases – 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.).

17.5.2 Destructors and free store

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

- Whatever resources a class object needs to function, it acquires 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 all resources 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 §19.5. Here, we will examine an important application that comes from the use of free-store and class hierarchies in combination. Consider:

```
Shape* fct()
{
    Text tt {Point{200,200}, "Annemarie"};
    // ...
    Shape* p = new Text{Point{100,100}, "Nicholas"};
    return p;
}

void f()
{
    Shape* q = fct();
    // ...
    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. Inside `fct()`, the `Text` (§13.11) 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 §17.5.1. But what about the `Text` object that was returned from `fct()`? The calling function `f()` 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 §14.2.1, 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 (§14.3.1) – 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.

As a rule of thumb: if you have a class with a **virtual** function, it needs a **virtual** destructor. The reason is:

1. If a class has a **virtual** function it is likely to be used as a base class, and
2. If it is a base class its derived class is likely to be allocated using **new**, and
3. If a derived class object is allocated using **new** and manipulated through a pointer to its base, then
4. It is likely to be **deleted** through a pointer to its base

Note that destructors are invoked implicitly 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.

17.6 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:

```
// a very simplified vector of doubles
class vector {
    int sz;                                // the size
    double* elem;                           // a pointer to the elements
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
};
```

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

Now we can make a `vector` of `doubles` 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 aim to start small and simple and then grow our programs step by step, testing along the way. As ever, this strategy of growth and repeated testing minimizes errors and debugging.

17.7 Pointers to class objects

The notion of “pointer” is general, so we can point to just about anything we can place in memory. For example, we can use pointers to `vectors` exactly as we use pointers to `chars`:

```
vector* f(int s)
{
    vector* p = new vector(s); // allocate a vector on free store
    // fill *p
    return p;
}

void ff()
{
    vector* q = f(4);
    // use *q
    delete q; // free vector on free store
}
```

Note that when we **delete** a **vector**, its destructor is called. For example:

```
vector* p = new vector(s);           // allocate a vector on free store
delete p;                           // deallocate
```

Creating the **vector** on the free store, the **new** operator

- First allocates memory for a **vector**
- Then invokes the **vector**'s constructor to initialize that **vector**; the constructor allocates memory for the **vector**'s elements and initializes those elements

Deleting the **vector**, the **delete** operator

- First invokes the **vector**'s destructor; the destructor invokes the destructors for the elements (if they have destructors) and then deallocates the memory used for the **vector**'s elements
- Then deallocates the memory used for the **vector**

Note how nicely that works recursively (see §8.5.8). Using the real (standard library) **vector** we can also do

```
vector<vector<double>>* p = new vector<vector<double>>(10);
delete p;
```

Here, **delete p** invokes the destructor for **vector<vector<double>>**; this destructor in turn invokes the destructor for its **vector<double>** elements, and all is neatly cleaned up, leaving no object undestroyed and leaking no memory.

Because **delete** invokes destructors (for types, such as **vector**, that have one), **delete** is often said to destroy objects, not just deallocate them.

As usual, please remember that a “naked” **new** outside a constructor is an opportunity to forget to **delete** the object that **new** created. Unless you have a good (that is, really simple, such as **Vector_ref** from §13.10 and §E.4) strategy for deleting objects, try to keep **news** in constructors and **deletes** in destructors.

So far, so good, but how do we access the members of a **vector**, given only a pointer? Note that all classes support the operator **.** (dot) for accessing members, given the name of an object:

```
vector v(4);
int x = v.size();
double d = v.get(3);
```

Similarly, all classes support the operator `->` (arrow) for accessing members, given a pointer to an object:

```
vector* p = new vector(4);
int x = p->size();
double d = p->get(3);
```

Like `.` (dot), `->` (arrow) can be used for both data members and function members. Since built-in types, such as `int` and `double`, have no members, `->` doesn't apply to built-in types. Dot and arrow are often called *member access operators*.

17.8 Messing with types: `void*` and casts

Using pointers and free-store-allocated arrays, we are very close to the hardware. Basically, our operations on pointers (initialization, assignment, `*`, and `[]`) map directly to machine instructions. At this level, the language offers only a bit of notational convenience and the compile-time consistency offered by the type system. Occasionally, we have to give up even that last bit of protection.

Naturally, we don't want to make do without the protection of the type system, but sometimes there is no logical alternative (e.g., we need to interact with another language that doesn't know about C++'s types). There are also an unfortunate number of cases where we need to interface with old code that wasn't designed with static type safety in mind. For that, we need two things:

- A type of pointer that points to memory without knowing what kinds of objects reside in that memory
- An operation to tell the compiler what kind of type to assume (without proof) for memory pointed to by one of those pointers



The type `void*` means "pointer to some memory that the compiler doesn't know the type of." We use `void*` when we want to transmit an address between pieces of code that really don't know each other's types. Examples are the "address" arguments of a callback function (§16.3.1) and the lowest level of memory allocators (such as the implementation of the `new` operator).

There are no objects of type `void`, but as we have seen, we use `void` to mean "no value returned":

```
void v;      // error: there are no objects of type void
void f();    // f() returns nothing — f() does not return an object of type void
```

A pointer to any object type can be assigned to a `void*`. For example:

```
void* pv1 = new int;           // OK: int* converts to void*
void* pv2 = new double[10]; // OK: double* converts to void*
```

Since the compiler doesn't know what a `void*` points to, we must tell it:

```
void f(void* pv)
{
    void* pv2 = pv;           // copying is OK (copying is what void*s are for)
    double* pd = pv;         // error: cannot convert void* to double*
    *pv = 7;                 // error: cannot dereference a void*
                            // (we don't know what type of object it points to)
    pv[2] = 9;               // error: cannot subscript a void*
    int* pi = static_cast<int*>(pv); // OK: explicit conversion
    // ...
}
```

A `static_cast` can be used to explicitly convert between related pointer types, such as `void*` and `double*` (§A.5.7). The name `static_cast` is a deliberately ugly name for an ugly (and dangerous) operation – use it only when absolutely necessary. You shouldn't find it necessary very often – if at all. An operation such as `static_cast` is called an *explicit type conversion* (because that's what it does) or colloquially a *cast* (because it is used to support something that's broken).

C++ offers two casts that are potentially even nastier than `static_cast`:

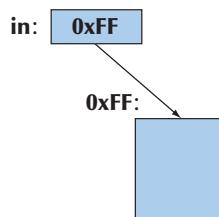
- `reinterpret_cast` can cast between unrelated types, such as `int` and `double*`.
- `const_cast` can “cast away `const`.”

For example:

```
Register* in = reinterpret_cast<Register*>(0xff);

void f(const Buffer* p)
{
    Buffer* b = const_cast<Buffer*>(p);
    // ...
}
```

The first example is the classical necessary and proper use of a `reinterpret_cast`. We tell the compiler that a certain part of memory (the memory starting with location `0xFF`) is to be considered a `Register` (presumably with special semantics). Such code is necessary when you write things like device drivers.





In the second example, `const_cast` strips the `const` from the `const Buffer*` called `p`. Presumably, we know what we are doing.

At least `static_cast` can't mess with the pointer/integer distinction or with "const-ness," so prefer `static_cast` if you feel the need for a cast. When you think you need a cast, reconsider: Is there a way to write the code without the cast? Is there a way to redesign that part of the program so that the cast is not needed? Unless you are interfacing to other people's code or to hardware, there usually is a way. If not, expect subtle and nasty bugs. Don't expect code using `reinterpret_cast` to be portable.



17.9 Pointers and references

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).
- To get a pointer you generally need to use `new` or `&`.
- To access an object pointed to by a pointer you use `*` or `[]`.
- Assignment to a reference changes the value of the object referred to (not the reference itself).
- You cannot make a reference refer to a different object after initialization.
- Assignment of references does deep copy (assigns to the referred-to object); assignment of pointers does not (assigns to the pointer object itself).
- Beware of null pointers.

For example:

```
int x = 10;
int* p = &x;           // you need & to get a pointer
*p = 7;                // use * to assign to x through p
int x2 = *p;           // read x through p
int* p2 = &x2;          // get a pointer to another int
p2 = p;                // p2 and p both point to x
p = &x2;                // make p point to another object
```

The corresponding example for references is

```
int y = 10;
int& r = y;           // the & is in the type, not in the initializer
r = 7;                 // assign to y through r (no * needed)
int y2 = r;             // read y through r (no * needed)
```

```

int& r2 = y2;           // get a reference to another int
r2 = r;                // the value of y is assigned to y2
r = &y2;               // error: you can't change the value of a reference
                          // (no assignment of an int* to an int&)

```

Note the last example; it is not just this construct that will fail – there is no way to get a reference to refer to a different object after initialization. If you need to point to something different, use a pointer. For ideas of how to use pointers, see §17.9.3.

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.

17.9.1 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” has a move constructor (§18.3.4) 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.

Using a pointer argument alerts the programmer that something might be changed. For example:

```

int x = 7;
incr_p(&x)           // the & is needed
incr_r(x);

```

The need to use **&** in **incr_p(&x)** alerts the user that **x** might be changed. In contrast, **incr_r(x)** “looks innocent.” This leads to a slight preference for the pointer version.



On the other hand, 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:

```
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 5 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. Note: That's not the case for an 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”:



- For tiny objects prefer pass-by-value.
- 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 §8.5.6.

17.9.2 Pointers, references, and inheritance

In §14.3, we saw how a derived class, such as `Circle`, could be used where an object of its public base class `Shape` was required. We can express that idea in terms of pointers or references: a `Circle*` can be implicitly converted to a `Shape*` because `Shape` is a public base of `Circle`. For example:

```
void rotate(Shape* s, int n);           // rotate *s n degrees

Shape* p = new Circle{Point{100,100},40};
Circle c {Point{200,200},50};
```

```
rotate(p,35);
rotate(&c,45);
```

And similarly for references:

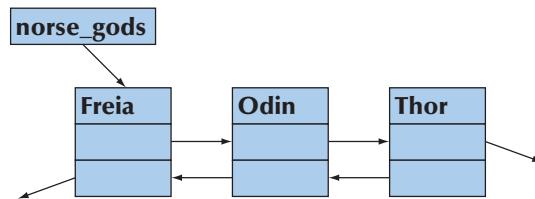
```
void rotate(Shape& s, int n);           // rotate s n degrees

Shape& r = c;
rotate(r,55);
rotate(*p,65);
rotate(c,75);
```

This is crucial for most object-oriented programming techniques (§14.3–4).

17.9.3 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:



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

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 the null pointer 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{"Freia",nullptr,norse_gods};
norse_gods->succ->prev = norse_gods;
```

We built that list by creating the **Links** and tying them together as in the picture: first Thor, then Odin as the predecessor of Thor, and finally Freia 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 **Links**, 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("Freia"));
```

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()**.

Note that we used default arguments (§15.3.1, §A.9.2) to save users from mentioning predecessors and successors in every constructor use.

17.9.4 List operations

The standard library provides a **list** class, which we will describe in §20.4. 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 (see exercise 11)
}

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;
    if (0<n) {
        while (n--) {
            if (p->succ == nullptr) return nullptr;
            p = p->succ;
        }
    }
    else if (n<0) {
        while (n++) {
            if (p->prev == nullptr) return nullptr;
            p = p->prev;
        }
    }
    return p;
}

```

Note the use of the postfix **n++**. This form of increment (“post-increment”) yields the value before the increment as its value.

17.9.5 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("Freia"));

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.

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

```

{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }

```

17.10 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 member functions. 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);                      // finds s in list
    const Link* find(const string& s) const;           // finds s in const list (see §18.5.1)

    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 **Links**) – 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 previous global **insert()** and modifying it suitably:

```
Link* Link::insert(Link* n)           // insert n before p; return n
{
    Link* p = this;                  // pointer to this object
    if (n==nullptr) return p;        // nothing to insert
    if (p==nullptr) return n;        // nothing to insert into
    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**:

```
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
        //...
    }
};
```

17.10.1 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{"Freia"});

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.” Correct the name of the god of war:

```
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_gods->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

```
{ Freia, 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 §9.7.5.



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. We can do better than that – by defining a class **List** – but designs along the lines presented here are very common. The standard library **list** is presented in §20.4.



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_array10(ostream& os, int* a)** that prints out the values of **a** (assumed to have ten 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. 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**.
8. Allocate an array of 20 **ints** on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
9. Did you remember to delete the arrays? (If not, do it.)
10. Do 5, 6, and 8 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. Using **print_array()** from the last drill:

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 a variable **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 dereference operator and why do we need one?
5. What is an address? How are memory addresses manipulated in C++?
6. What information about a pointed-to object does a pointer have? What useful information does it lack?
7. What can a pointer point to?
8. What is a leak?
9. What is a resource?
10. How can we initialize a pointer?
11. What is a null pointer? When do we need to use one?
12. When do we need a pointer (instead of a reference or a named object)?
13. What is a destructor? When do we want one?
14. When do we want a **virtual** destructor?
15. How are destructors for members called?
16. What is a cast? When do we need to use one?
17. How do we access a member of a class through a pointer?
18. What is a doubly-linked list?
19. What is **this** and when do we need to use it?

Terms

address	destructor	nullptr
address of: &	free store	pointer
allocation	link	range
cast	list	resource leak
container	member access: ->	subscripting
contents of: *	member destructor	subscript: []
deallocation	memory	this
delete	memory leak	type conversion
delete[]	new	virtual destructor
dereference	null pointer	void*

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. 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 functions. 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.
4. Write a function, `char* strdup(const char*)`, that copies a C-style string into memory it allocates on the free store. Do not use any standard library functions.
5. Write a function, `char* findx(const char* s, const char* x)`, that finds the first occurrence of the C-style string `x` in `s`.
6. This chapter does not say 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?
7. 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.
8. Do exercise 7 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).
9. 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.
10. Look at your solution of exercise 7. 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?
11. Complete the "list of gods" example from §17.10.1 and run it.
12. Why did we define two versions of `find()`?
13. Modify the `Link` class from §17.10.1 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 `Links` 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.

14. Could the “list of gods” example from §17.10.1 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.

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 we’d like to know how that’s done. There are programming languages that don’t provide facilities equivalent to pointers and thus 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, etc. often have the strangest ideas of how their programming language facilities work; such wrong ideas can lead to code that’s “interestingly poor.”



Vectors and Arrays

“Caveat emptor!”

—Good advice

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

18.1 Introduction**18.2 Initialization****18.3 Copying**

- 18.3.1 Copy constructors
- 18.3.2 Copy assignments
- 18.3.3 Copy terminology
- 18.3.4 Moving

18.4 Essential operations

- 18.4.1 Explicit constructors
- 18.4.2 Debugging constructors and destructors

18.5 Access to `vector` elements

- 18.5.1 Overloading on `const`

18.6 Arrays

- 18.6.1 Pointers to array elements
- 18.6.2 Pointers and arrays
- 18.6.3 Array initialization
- 18.6.4 Pointer problems

18.7 Examples: palindrome

- 18.7.1 Palindromes using `string`
- 18.7.2 Palindromes using arrays
- 18.7.3 Palindromes using pointers

18.1 Introduction

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

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

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

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

This chapter focuses on the notion of copying. This is an important but rather technical point: What do we mean by copying a nontrivial object? To what extent

are the copies independent after a copy operation? What copy operations are there? How do we specify them? And how do they relate to other fundamental operations, such as initialization and cleanup?

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

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

18.2 Initialization

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

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

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

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

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

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

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

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

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

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

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

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

Now we can write

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

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

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

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

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

```
vector v1 = {1,2,3};    // three elements 1.0, 2.0, 3.0
vector v12 {1,2,3};    // three elements 1.0, 2.0, 3.0
```

The difference is purely one of style.

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

18.3 Copying

Consider again our incomplete **vector**:

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

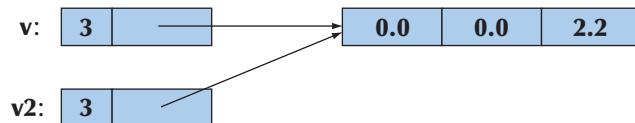
Let's try to copy one of these vectors:

```
void f(int n)
{
    vector v(3);           // define a vector of 3 elements
    v.set(2,2.2);         // set v[2] to 2.2
```

```
vector v2 = v;      // what happens here?
// ...
}
```

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

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



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

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

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

```
delete[] elem;
```

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

18.3.1 Copy constructors

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

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

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

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

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

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

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

Given this copy constructor, consider again our example:

```
vector v2 = v;
```

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



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

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

This will output **0 0**.

Instead of saying

```
vector v2 = v;
```

we could equally well have said

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

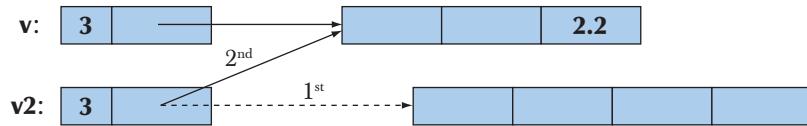
When **v** (the initializer) and **v2** (the variable being initialized) are of the same type and that type has copying conventionally defined, those two notations mean exactly the same thing and you can use whichever notation you like better.

18.3.2 Copy assignments

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

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

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



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

```

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

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

```

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

```

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

```

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

```

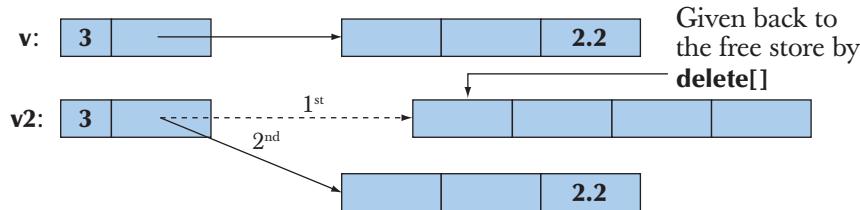
delete[] elem;                         // deallocate old space

```

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

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

We can represent the result graphically like this:



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

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

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

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

18.3.3 Copy terminology

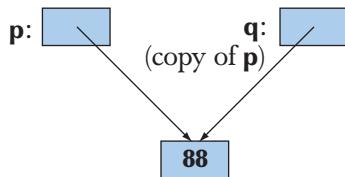
Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer (or reference) or copy the information pointed to (referred to):

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

Here is an example of shallow copy:

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

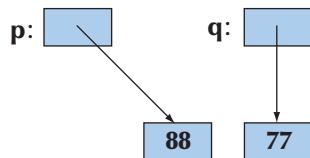
We can illustrate that like this:



In contrast, we can do a deep copy:

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

We can illustrate that like this:



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



18.3.4 Moving

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

```
vector fill(istream& is)
{
```

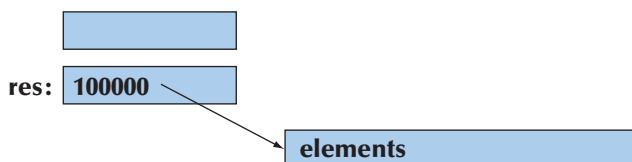
```

vector res;
for (double x; is>>x; ) res.push_back(x);
return res;
}

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

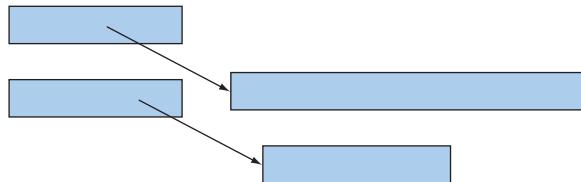
```

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



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

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



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

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

```

class vector {
    int sz;
    double* elem;

```

```
public:
    vector(vector&& a);           // move constructor
    vector& operator=(vector&&); // move assignment
    // ...
};
```

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

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

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

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

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

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

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

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

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

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

18.4 Essential operations

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

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

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

```

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

```

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

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

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

```

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

```

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

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



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

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

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

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

18.4.1 Explicit constructors

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

```
class complex {  
public:  
    complex(double);           // defines double-to-complex conversion  
    complex(double,double);  
    // ...  
};  
  
complex z1 = 3.14;           // OK: convert 3.14 to (3.14,0)  
complex z2 = complex{1.2, 3.4};
```

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

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

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

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

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

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

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

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

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

18.4.2 Debugging constructors and destructors

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

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

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

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

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

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

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

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

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

X copy(X a) { return a; }

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

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

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

struct XX { X a; X b; };

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

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

Try executing that.

TRY THIS



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

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

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





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

18.5 Access to vector elements

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

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

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

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

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

TRY THIS



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

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

```
class vector {
    int sz;           // the size
    double* elem;    // a pointer to the elements
```

```
public:
    // ...
    double* operator[](int n) { return &elem[n]; }      // return pointer
};
```

Given that definition, we can write

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

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

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

Now we can write

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

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

18.5.1 Overloading on const

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

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

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

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

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

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

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

18.6 Arrays

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

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

Now, you might have detected that we have a not-so-subtle bias in favor of `vectors` over arrays. Use `std::vector` where you have a choice – and you have a choice in most contexts. However, arrays existed long before `vectors` and are roughly equivalent to what is offered in other languages (notably C), so you must know

arrays, and know them well, to be able to cope with older code and with code written by people who don't appreciate the advantages of **vector**.

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

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

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

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

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

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

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

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

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

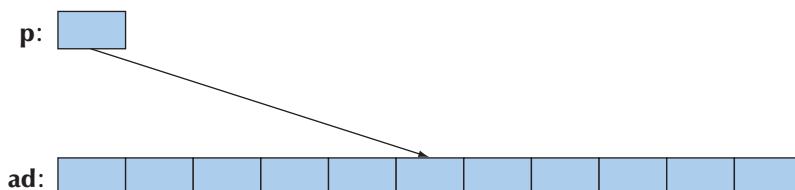
But couldn't the compiler see that `lac` has just 20 elements so that `lac[200]` is an error? A compiler could, but as far as we know no production compiler does. The problem is that keeping track of array bounds at compile time is impossible in general, and catching errors in the simplest cases (like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

18.6.1 Pointers to array elements

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

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

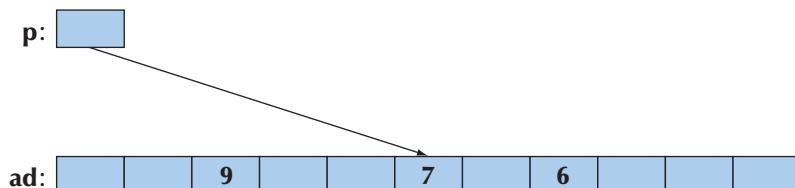
We now have a pointer `p` to the `double` known as `ad[5]`:



We can subscript and dereference that pointer:

```
*p = 7;
p[2] = 6;
p[-3] = 9;
```

We get

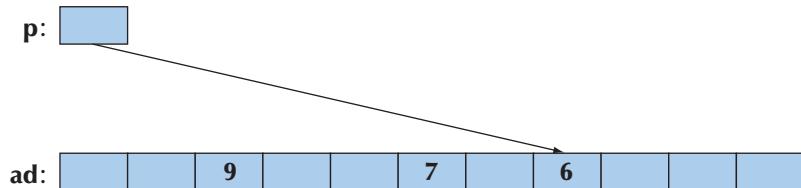


That is, we can subscript the pointer with both positive and negative numbers. As long as the resulting element is in range, all is well. However, access outside the range of the array pointed into is illegal (as with free-store-allocated arrays; see §17.4.3). Typically, access outside an array is not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

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

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

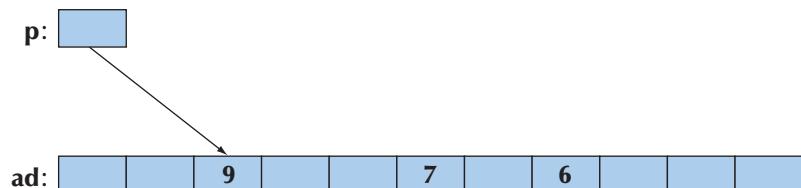
We get



And

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

We get



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

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

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

The most common use of pointer arithmetic is incrementing a pointer (using ****++****) to point to the next element and decrementing a pointer (using ****--****) to point

to the previous element. For example, we could print the value of `ad`'s elements like this:

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

Or backward:

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

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

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

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

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

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

18.6.2 Pointers and arrays

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

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

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

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

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

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

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

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

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

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

char lots [100000];

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

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

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

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

Finally! A problem that the compiler will catch!

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

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

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

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

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

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

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

18.6.3 Array initialization

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

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

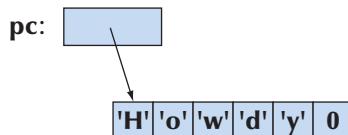
Count those characters. There are five, but **ac** becomes an array of six characters because the compiler adds a terminating zero character at the end of a string literal:

ac:	'B'	'e'	'o'	'r'	'n'	0
-----	-----	-----	-----	-----	-----	---

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

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

Graphically:



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

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

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

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

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

Note that the number of elements of `ai` is six (not seven) and the number of elements for `chars` is three (not four) – the “add a `0` at the end” rule is for literal character strings only. If an array isn't given a size, that size is deduced from the initializer list. That's a rather useful feature. If there are fewer initializer values

than array elements (as in the definitions of `ai2` and `ad`), the remaining elements are initialized by the element type's default value.

18.6.4 Pointer problems

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

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

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

Don't access through the null pointer:



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

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

```
int* p = fct_that_can_return_a_nullptr();

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

and

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

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

Do initialize your pointers:

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

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

Don't access nonexistent array elements:

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

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

Don't access through a deleted pointer:

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

The **delete p** or the code after it may have scribbled all over ***p** or used it for something else. Of all of these problems, we consider this one the hardest to

systematically avoid. The most effective defense against this problem is not to have “naked” **news** that require “naked” **deletes**: use **new** and **delete** in constructors and destructors or use a container, such as **Vector_ref** (§E.4), to handle **deletes**.

Don't return a pointer to a local variable:

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

// ...

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

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

Consider a logically equivalent example:

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

// ...

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

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

It is common for programmers to underestimate these problems. However, many experienced programmers have been defeated by the innumerable varia-



tions and combinations of these simple array and pointer problems. The solution is not to litter your code with pointers, arrays, **news**, and **deletes**. If you do, “being careful” simply isn’t enough in realistically sized programs. Instead, rely on vectors, RAII (“Resource Acquisition Is Initialization”; see §19.5), and other systematic approaches to the management of memory and other resources.

18.7 Examples: palindrome

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

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

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

18.7.1 Palindromes using `string`

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

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

We return **true** if we reach the middle without finding a difference. We suggest that you look at this code to convince yourself that it is correct when there are no

letters in the string, just one letter in the string, an even number of letters in the string, and an odd number of letters in the string. Of course, we should not just rely on logic to see that our code is correct. We should also test. We can exercise `is_palindrome()` like this:

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

Basically, the reason we are using a `string` is that “`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 dffd sa` to be palindromes.

18.7.2 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,
                        // add zero after the last character read into buffer
    return is;
}

```

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

```

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

```

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

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



18.7.3 Palindromes using pointers

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

```

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

```

```

    }
    return true;
}

```



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

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

```

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

```

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

```

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

```

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



Drill

In this chapter, we have two drills: one to exercise arrays and one to exercise **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()**:
 - a. Define a local **int** array **la** of ten **ints**.
 - b. Copy the values from **ga** into **la**.
 - c. Print out the elements of **la**.
 - d. Define a pointer **p** to **int** and initialize it with an array allocated on the free store with the same number of elements as the argument array.
 - e. Copy the values from the argument array into the free-store array.
 - f. Print out the elements of the free-store array.
 - g. Deallocate the free-store array.
4. In **main()**:
 - a. Call **f()** with **ga** as its argument.
 - b. Define an array **aa** with ten elements, and initialize it with the first ten factorial values (1, 2*1, 3*2*1, 4*3*2*1, etc.).
 - c. Call **f()** with **aa** as its argument.

Standard library **vector** drill:

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

Review

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

Terms

array	deep copy	move assignment
array initialization	default constructor	move construction
copy assignment	essential operations	palindrome
copy constructor	<code>explicit</code> constructor	shallow copy

Exercises

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

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

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

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

“I smell the wumpus”: It’s in an adjoining room.

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

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

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

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

Postscript

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



Vector, Templates, and Exceptions

“Success is never final.”

—Winston Churchill

This chapter completes the design and implementation of the most common and most useful STL container: **vector**. Here, we show how to implement containers where the number of elements can vary, how to specify containers where the element type is a parameter, and how to deal with range errors. As usual, the techniques used are generally applicable, rather than simply restricted to the implementation of **vector**, or even to the implementation of containers. Basically, we show how to deal safely with varying amounts of data of a variety of types. In addition, we add a few doses of realism as design lessons. The techniques rely on templates and exceptions, so we show how to define templates and give the basic techniques for resource management that are the keys to good use of exceptions.

19.1 The problems	19.4 Range checking and exceptions
19.2 Changing size	19.4.1 An aside: design considerations
19.2.1 Representation	19.4.2 A confession: macros
19.2.2 <code>reserve</code> and <code>capacity</code>	19.5 Resources and exceptions
19.2.3 <code>resize</code>	19.5.1 Potential resource management problems
19.2.4 <code>push_back</code>	19.5.2 Resource acquisition is initialization
19.2.5 Assignment	19.5.3 Guarantees
19.2.6 Our <code>vector</code> so far	19.5.4 <code>unique_ptr</code>
19.3 Templates	19.5.5 Return by moving
19.3.1 Types as template parameters	19.5.6 RAI for <code>vector</code>
19.3.2 Generic programming	
19.3.3 Concepts	
19.3.4 Containers and inheritance	
19.3.5 Integers as template parameters	
19.3.6 Template argument deduction	
19.3.7 Generalizing <code>vector</code>	

19.1 The problems

At the end of Chapter 18, our `vector` 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
- 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)

That's all good and useful, but to reach the level of sophistication we expect (based on experience with the standard library `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:

```
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;
cin>>n;
vc.resize(n);               // make vc have n elements
```

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 is 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 **vector<Number>**.

Obviously, 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, but 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:

```
// read elements into a vector without using push_back:
vector<double>* p = new vector<double>(10);
int n = 0;           // number of elements
for (double d; cin>>d; ) {
    if (n==p->size()) {
        vector<double>* q = new vector<double>(p->size()*2);
        copy(p->begin(), p->end(), q->begin());
```

```

    delete p;
    p = q;
}
(*p)[n] = d;
++n;
}

```

That's not pretty. Are you convinced that we got it right? How can you be sure? Note how we suddenly started to use pointers and explicit memory management. What we did was to imitate the style of programming we have to use when we are “close to the machine,” using only the basic memory management techniques dealing with fixed-size objects (arrays; see §18.6). One of the reasons to use containers, such as `vector`, is to do better than that; that is, we want `vector` to handle such size changes internally to save us – its users – the bother and the chance to make mistakes. In other words, we prefer containers that can grow to hold the exact number of elements we happen to need. For example:

```

vector<double> vd;
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 containers, but how.

 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. People who try that tend to have to rewrite code (if they carefully and systematically checked for overflows) and deal with disasters (if they were careless with their checking).

Obviously, not all `vectors` have the same type of elements. We need `vectors` of **doubles**, temperature readings, records (of various kinds), `strings`, operations, GUI buttons, shapes, dates, pointers to windows, etc. The possibilities are endless.

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 `vectors`? 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 **vectors** and **strings** (a **string** is a container of characters) extensively. In the next chapters, we will see **lists**, **maps** (a **map** is a tree of pairs of values), and matrices. 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.

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.

19.2 Changing size

What facilities for changing size does the standard library **vector** offer? It provides three simple operations. Given

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

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()** (§B.4.7), but here we will just see how we can implement those three operations for our **vector**.

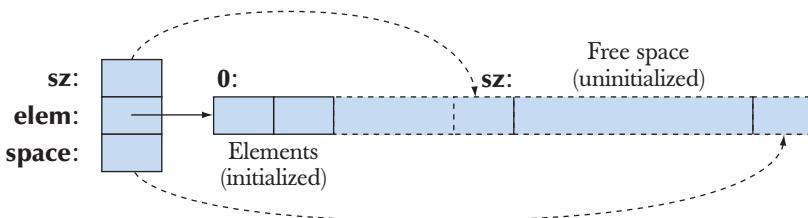
19.2.1 Representation

In §19.1, we showed the simplest 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:

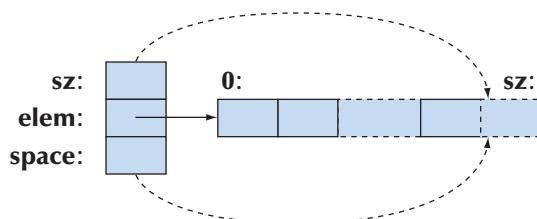
```
class vector {
    int sz;           // number of elements
    double* elem;    // address of first element
    int space;        // number of elements plus “free space”/“slots”
                        // for new elements (“the current allocation”)
public:
    // ...
};
```

We can represent this graphically like this:



Since we count elements starting with **0**, we represent **sz** (the number of elements) as referring to one beyond the last element and **space** as referring 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”:

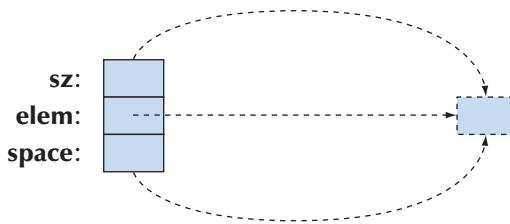


We don’t start allocating extra slots until we begin changing the number of elements. Typically, **space==sz**, so there is no memory overhead unless we use **push_back()**.

The default constructor (creating a **vector** with no elements) sets the integer members to **0** and the pointer member to **nullptr**:

```
vector::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.

19.2.2 `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) return;           // never decrease allocation
    double* p = new double[newalloc];      // allocate new space
    for (int i=0; i<sz; ++i) p[i] = elem[i]; // copy old elements
    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.

19.2.3 `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) elem[i] = 0;      // initialize new elements
    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`?

19.2.4 `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)
        reserve(8);           // start with space for 8 elements
```

```

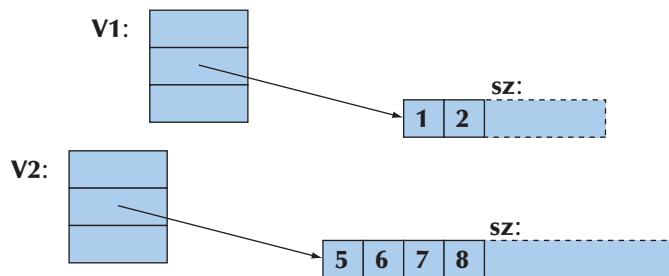
else if (sz==space)
    reserve(2*space); // get more space
elem[sz] = d;        // add d at end
++sz;                // increase the size (sz is the number of elements)
}

```

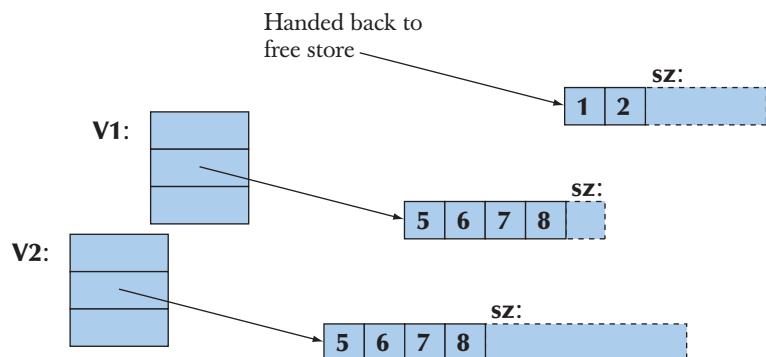
In other words, if we have no spare space, we double the size of the allocation. In practice that turns out to be a very good choice for the vast majority of uses of **vector**, and that's the strategy used by most implementations of the standard library **vector**.

19.2.5 Assignment

We could have defined 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 §18.3.2 we decided that vector assignment should have the most 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, 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) p[i] = a.elem[i]; // copy elements
    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 §17.10.

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) return *this; // self-assignment, no work needed

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

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

```

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` line. 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.

19.2.6 Our vector so far

Now we have an almost real `vector` of `doubles`:

```

// an almost real vector 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(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);

};

```

Note how it has the essential operations (§18.4): constructor, default constructor, copy operations, 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()`).

19.3 Templates

But we don't just want `vectors` of `doubles`; we want to freely specify the element type for our `vectors`. For example:

```

vector<double>
vector<int>
vector<Month>
vector<Window*>          // vector of pointers to Windows
vector<vector<Record>>   // vector of vectors of Records
vector<char>

```

To do that, we must see how to define templates. We have used templates from day one, but until now we haven't had a need to define one. The standard library provides what we have needed so far, but we mustn't believe in magic, so we need to examine how the designers and implementers of the standard library provided facilities such as the `vector` type and the `sort()` function (§21.1, §B.5.4). This is not just of theoretical interest, because – as usual – the tools and techniques used for the standard library are among the most useful for our own code. For example, in Chapters 21 and 22, we show how templates can be used for implementing the standard library containers and algorithms. In Chapter 24, we show how to design matrices for scientific computation.

Basically, a *template* is a mechanism that allows a programmer to use types as parameters for a class or a function. The compiler then generates a specific class or function when we later provide specific types as arguments.

19.3.1 Types as template parameters

We want to make the element type a parameter to `vector`. So we take our `vector` and replace `double` with `T` where `T` is a parameter that can be given “values” such as `double`, `int`, `string`, `vector<Record>`, and `Window*`. The C++ notation for introducing a type parameter `T` is the `template<typename T>` prefix, meaning “for all types `T`.” For example:

```
// an almost real vector of Ts:
template<typename T>
class vector {           // read "for all types T" (just like in math)
    int sz;                // the size
    T* elem;              // a pointer to the elements
    int space;             // size + free space
public:
    vector() : sz{0}, elem{nullptr}, space{0} {}
    explicit vector(int s) :sz{s}, elem{new T[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;      // elements are initialized
    }

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

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

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

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

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

    void resize(int newsize);            // growth
    void push_back(const T & d);
    void reserve(int newalloc);

};
```

That's just our `vector` of `doubles` from §19.2.6 with `double` replaced by the template parameter `T`. We can use this class template `vector` like this:

```
vector<double> vd;           // T is double
vector<int> vi;              // T is int
vector<double*> vpd;         // T is double*
vector<vector<int>> vvi;     // T is vector<int>, in which T is int
```

One way of thinking about what a compiler does when we use a template is that it generates the class with the actual type (the template argument) in place of the template parameter. For example, when the compiler sees `vector<char>` in the code, it (somewhere) generates something like this:



```
class vector_char {
    int sz;                      // the size
    char* elem;                  // a pointer to the elements
    int space;                   // size + free space
public:
    vector() : sz{0}, elem{nullptr}, space{0} { }
    explicit vector_char(int s) :sz{s}, elem{new char[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;           // elements are initialized
    }

    vector_char(const vector_char&);                // copy constructor
    vector_char& operator=(const vector_char&);    // copy assignment

    vector_char(vector_char&&);                   // move constructor
    vector_char& operator=(vector_char&&);        // move assignment

    ~vector_char();                                // destructor

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

    int size() const;                            // the current size
    int capacity() const;

    void resize(int newsize);                  // growth
    void push_back(const char& d);
    void reserve(int newalloc);

};
```

For `vector<double>`, the compiler generates roughly the `vector` (of `double`) from §19.2.6 (using a suitable internal name meaning `vector<double>`).

Sometimes, we call a class template a *type generator*. The process of generating types (classes) from a class template given template arguments is called *specialization* or *template instantiation*. For example, `vector<char>` and `vector<Poly_line*>` are said to be specializations of `vector`. In simple cases, such as our `vector`, instantiation is a pretty simple process. In the most general and advanced cases, template instantiation is horrendously complicated. Fortunately for the user of templates, that complexity is in the domain of the compiler writer, not the template user. Template instantiation (generation of template specializations) takes place at compile time or link time, not at run time.

Naturally, we can use member functions of such a class template. For example:

```
void fct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Norah");
    // ...
}
```

When such a member function of a class template is used, the compiler generates the appropriate function. For example, when the compiler sees `v.push_back("Norah")`, it generates a function

```
void vector<string>::push_back(const string& d) /* . . . */
```

from the template definition

```
template<typename T> void vector<T>::push_back(const T& d) /* . . . */;
```

That way, there is a function for `v.push_back("Norah")` to call. In other words, when you need a function for given object and argument types, the compiler will write it for you based on its template.

Instead of writing `template<typename T>`, you can write `template<class T>`. The two constructs mean exactly the same thing, but some prefer `typename` “because it is clearer” and “because nobody gets confused by `typename` thinking that you can’t use a built-in type, such as `int`, as a template argument.” We are of the opinion that `class` already means type, so it makes no difference. Also, `class` is shorter.

19.3.2 Generic programming

Templates are the basis for generic programming in C++. In fact, the simplest definition of “generic programming” in C++ is “using templates.” That definition

is a bit too simpleminded, though. We should not define fundamental programming concepts in terms of programming language features. Programming language features exist to support programming techniques – not the other way around. As with most popular notions, there are many definitions of “generic programming.” We think that the most useful simple definition is

Generic programming: Writing code that works with a variety of types presented as arguments, as long as those argument types meet specific syntactic and semantic requirements.

For example, the elements of a `vector` must be of a type that we can copy (by copy construction and copy assignment), and in Chapters 20 and 21 we will see templates that require arithmetic operations on their arguments. When what we parameterize is a class, we get a *class template*, what is often called a *parameterized type* or a *parameterized class*. When what we parameterize is a function, we get a *function template*, what is often called a *parameterized function* and sometimes also called an *algorithm*. Thus, generic programming is sometimes referred to as “algorithm-oriented programming”; the focus of the design is more the algorithms than the data types they use.

Since the notion of parameterized types is so central to programming, let’s explore the somewhat bewildering terminology a bit further. That way we have a chance of not getting too confused when we meet such notions in other contexts.

This form of generic programming relying on explicit template parameters is often called *parametric polymorphism*. In contrast, the polymorphism you get from using class hierarchies and virtual functions is called *ad hoc polymorphism* and that style of programming is called *object-oriented programming* (§14.3–4). The reason that both styles of programming are called *polymorphism* is that each style relies on the programmer to present many versions of a concept by a single interface. *Polymorphism* is Greek for “many shapes,” referring to the many different types you can manipulate through a common interface. In the `Shape` examples from Chapters 16–19 we literally accessed many shapes (such as `Text`, `Circle`, and `Polygon`) through the interface defined by `Shape`. When we use `vectors`, we use many `vectors` (such as `vector<int>`, `vector<double>`, and `vector<Shape*>`) through the interface defined by the `vector` template.

There are several differences between object-oriented programming (using class hierarchies and virtual functions) and generic programming (using templates). The most obvious is that the choice of function invoked when you use generic programming is determined by the compiler at compile time, whereas for object-oriented programming, it is not determined until run time. For example:

```
v.push_back(x);      // put x into the vector v
s.draw();           // draw the shape s
```

For `v.push_back(x)` the compiler will determine the element type for `v` and use the appropriate `push_back()`, but for `s.draw()` the compiler will indirectly call some `draw()` function (using `s`'s `vtbl`; see §14.3.1). This gives object-oriented programming a degree of freedom that generic programming lacks, but leaves run-of-the-mill generic programming more regular, easier to understand, and better performing (hence the “ad hoc” and “parametric” labels).

To sum up:

- *Generic programming*: supported by templates, relying on compile-time resolution
- *Object-oriented programming*: supported by class hierarchies and virtual functions, relying on run-time resolution



Combinations of the two are possible and useful. For example:

```
void draw_all(vector<Shape*>& v)
{
    for (int i = 0; i < v.size(); ++i) v[i]→draw();
}
```

Here we call a virtual function (`draw()`) on a base class (`Shape`) using a virtual function – that's certainly object-oriented programming. However, we also kept `Shape*`'s in a `vector`, which is a parameterized type, so we also used (simple) generic programming.

So – assuming you have had your fill of philosophy for now – what do people actually use templates for? For unsurpassed flexibility and performance:

- Use templates where performance is essential (e.g., numerics and hard real time; see Chapters 24 and 25).
- Use templates where flexibility in combining information from several types is essential (e.g., the C++ standard library; see Chapters 20–21).



19.3.3 Concepts

Templates have many useful properties, such as great flexibility and near-optimal performance, but unfortunately they are not perfect. As usual, the benefits have corresponding weaknesses. For templates, the main problem is that the flexibility and performance come at the cost of poor separation between the “inside” of a template (its definition) and its interface (its declaration). This manifests itself in poor error diagnostics – often spectacularly poor error messages. Sometimes, these error messages come much later in the compilation process than we would prefer.



When compiling a use of a template, the compiler “looks into” the template and also into the template arguments. It does so to get the information to generate optimal code. To have all that information available, current compilers tend to require that a template must be fully defined wherever it is used. That includes all of its member functions and all template functions called from those. Consequently, template writers tend to place template definitions in header files. This is not actually required by the standard, but until radically improved implementations are widely available, we recommend that you do so for your own templates: place the definition of any template that is to be used in more than one translation unit in a header file.

Initially, write only very simple templates yourself and proceed carefully to gain experience. One useful development technique is to do as we did for `vector`: First develop and test a class using specific types. Once that works, replace the specific types with template parameters and test with a variety of template arguments. Use template-based libraries, such as the C++ standard library, for generality, type safety, and performance. Chapters 20 and 21 are devoted to the containers and algorithms of the standard library and will give you examples of the use of templates.

C++14 provides a mechanism for vastly improved checking of template interfaces. For example, in C++11 we write

```
template<typename T>      // for all types T
class vector {
    ...
};
```

We cannot precisely state what is expected of an argument type `T`. The standard says what these requirements are, but only in English, rather than in code that the compiler can understand. We call a set of requirements on a template argument a *concept*. A template argument must meet the requirements, the concepts, of the template to which it is applied. For example, a `vector` requires that its elements can be copied or moved, can have their address taken, and be default constructed (if needed). In other words, an element must meet a set of requirements, which we could call `Element`. In C++14, we can make that explicit:

```
template<typename T>      // for all types T
    requires Element<T>() // such that T is an Element
class vector {
    ...
};
```

This shows that a concept is really a type predicate, that is, a compile-time-evaluated (**constexpr**) function that returns **true** if the type argument (here, **T**) has the properties required by the concept (here, **Element**) and **false** if it does not. This is a bit long-winded, but a shorthand notation brings us to

```
template<Element T>           // for all types T, such that Element<T>() is true
class vector {
    // ...
};
```

If we don't have a C++14 compiler that supports concepts, we can specify our requirements in names and in comments:

```
template<typename Elem>      // requires Element<Elem>()
class vector {
    // ...
};
```

The compiler doesn't understand our names or read our comments, but being explicit about concepts helps us think about our code, improves our design of generic code, and helps other programmers understand our code. As we go along, we will use some common and useful concepts:

- **Element<E>()**: **E** can be an element in a container.
- **Container<C>()**: **C** can hold **Elements** and be accessed as a [**begin()**:**end()**] sequence.
- **Forward_iterator<For>()**: **For** can be used to traverse a sequence [**b**:**e**] (like a linked list, a vector, or an array).
- **Input_iterator<In>()**: **In** can be used to read a sequence [**b**:**e**] once only (like an input stream).
- **Output_iterator<Out>()**: A sequence can be output using **Out**.
- **Random_access_iterator<Ran>()**: **Ran** can be used to read and write a sequence [**b**:**e**] repeatedly and supports subscripting using **[]**.
- **Allocator<A>()**: **A** can be used to acquire and release memory (like the free store).
- **Equal_comparable<T>()**: We can compare two **T**s for equality using **==** to get a Boolean result.
- **Equal_comparable<T,U>()**: We can compare a **T** to a **U** for equality using **==** to get a Boolean result.

- **Predicate<P,T>()**: We can call **P** with an argument of type **T** to get a Boolean result.
- **Binary_predicate<P,T>()**: We can call **P** with two arguments of type **T** to get a Boolean result.
- **Binary_predicate<P,T,U>()**: We can call **P** with arguments of types **T** and **U** to get a Boolean result.
- **Less_comparable<L,T>()**: We can use **L** to compare two **T**s for less than using **<** to get a Boolean result.
- **Less_comparable<L,T,U>()**: We can use **L** to compare a **T** to a **U** for less than using **<** to get a Boolean result.
- **Binary_operation<B,T,U>()**: We can use **B** to do an operation on two **T**s.
- **Binary_operation<B,T,U>()**: We can use **B** to do an operation on a **T** and a **U**.
- **Number<N>()**: **N** behaves like a number, supporting **+**, **-**, *****, and **/**.

For standard library containers and algorithms, these concepts (and many more) are specified in excruciating detail. Here, especially in Chapters 20 and 21, we will use them informally to document our containers and algorithms.

A container type and an iterator type, **T**, have a value type (written as **Value_type<T>**), which is the element type. Often, that **Value_type<T>** is a member type **T::value_type**; see **vector** and **list** (§20.5).

19.3.4 Containers and inheritance

There is one kind of combination of object-oriented programming and generic programming that people always try, but it doesn't work: attempting to use a container of objects of a derived class as a container of objects of a base class. For example:

```
vector<Shape> vs;
vector<Circle> vc;
vs = vc;           // error: vector<Shape> required
void f(vector<Shape>&);
f(vc);           // error: vector<Shape> required
```

 But why not? After all, you say, I can convert a **Circle** to a **Shape**! Actually, no, you can't. You can convert a **Circle*** to a **Shape*** and a **Circle&** to a **Shape&**, but we deliberately disabled assignment of **Shapes**, so that you wouldn't have to wonder what would happen if you put a **Circle** with a radius into a **Shape** variable that doesn't have a radius (§14.2.4). What would have happened – had we allowed

it – would have been what is called “slicing” and is the class object equivalent of integer truncation (§3.9.2).

So we try again using pointers:

```
vector<Shape*> vps;
vector<Circle*> vpc;
vps = vpc;           // error: vector<Shape*> required
void f(vector<Shape*>&);
f(vpc);             // error: vector<Shape*> required
```

Again, the type system resists; why? Consider what **f()** might do:

```
void f(vector<Shape*>& v)
{
    v.push_back(new Rectangle{Point{0,0},Point{100,100}});
}
```

Obviously, we can put a **Rectangle*** into a **vector<Shape*>**. However, if that **vector<Shape*>** was elsewhere considered to be a **vector<Circle*>**, someone would get a nasty surprise. In particular, had the compiler accepted the example above, what would a **Rectangle*** be doing in **vpc**? Inheritance is a powerful and subtle mechanism and templates do not implicitly extend its reach. There are ways of using templates to express inheritance, but they are beyond the scope of this book. Just remember that “**D** is a **B**” does not imply “**C<D>** is a **C**” for an arbitrary template **C** – and we should value that as a protection against accidental type violations. See also §25.4.4.

19.3.5 Integers as template parameters

Obviously, it is useful to parameterize classes with types. How about parameterizing classes with “other things,” such as integer values and string values? Basically, any kind of argument can be useful, but we’ll consider only type and integer parameters. Other kinds of parameters are less frequently useful, and C++’s support for other kinds of parameters is such that their use requires quite detailed knowledge of language features.

Consider an example of the most common use of an integer value as a template argument, a container where the number of elements is known at compile time:

```
template<typename T, int N> struct array {
    T elem[N];           // hold elements in member array
    // rely on the default constructors, destructor, and assignment
```

```

T& operator[] (int n);           // access: return reference
const T& operator[] (int n) const;

T* data() { return elem; }      // conversion to T*
const T* data() const { return elem; }

int size() const { return N; }
};

```

We can use **array** (see also §20.7) like this:

```

array<int,256> gb;           // 256 integers
array<double,6> ad = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
const int max = 1024;

void some_fct(int n)
{
    array<char,max> loc;
    array<char,n> oops;      // error: the value of n not known to compiler
    // ...
    array<char,max> loc2 = loc; // make backup copy
    // ...
    loc = loc2;              // restore
    // ...
}

```

Clearly, **array** is very simple – much simpler and less powerful than **vector** – so why would anyone want to use an **array** rather than a **vector**? One answer is “efficiency.” We know the size of an **array** at compile time, so the compiler can allocate static memory (for global objects, such as **gb**) and stack memory (for local objects, such as **loc**) rather than using the free store. When we do range checking, the checks can be against constants (the size parameter **N**). For most programs the efficiency improvement is insignificant, but if you are writing a crucial system component, such as a network driver, even a small difference can matter. More importantly, some programs simply can’t be allowed to use the free store. Such programs are typically embedded systems programs and/or safety-critical programs (see Chapter 25). In such programs, **array** gives us many of the advantages of **vector** without violating a critical restriction (no free-store use).

Let’s ask the opposite question: not “Why can’t we just use **vector**?” but “Why not just use built-in arrays?” As we saw in §18.6, arrays can be rather ill behaved: they don’t know their own size, they convert to pointers at the slightest provocation, they don’t copy properly; like **vector**, **array** doesn’t have those problems. For example:

```
double* p = ad;           // error: no implicit conversion to pointer
double* q = ad.data();    // OK: explicit conversion

template<typename C> void printout(const C& c)    // function template
{
    for (int i = 0; i < c.size(); ++i) cout << c[i] << '\n';
}
```

This `printout()` can be called by an `array` as well as a `vector`:

```
printout(ad);           // call with array
vector<int> vi;
// ...
printout(vi);          // call with vector
```

This is a simple example of generic programming applied to data access. It works because the interface used for `array` and `vector` (`size()` and subscripting) is the same. Chapters 20 and 21 will explore this style of programming in some detail.

19.3.6 Template argument deduction

For a class template, you specify the template arguments when you create an object of some specific class. For example:

```
array<char,1024> buf;           // for buf, T is char and N is 1024  
array<double,10> b2;           // for b2, T is double and N is 10
```

For a function template, the compiler usually deduces the template arguments from the function arguments. For example:



Technically, `fill(buf, 'x')` is shorthand for `fill<char,1024>(buf, 'x')`, and `fill(b2, 0)` is shorthand for `fill<double,10>(b2, 0)`, but fortunately we don't often have to be that specific. The compiler figures it out for us.

19.3.7 Generalizing `vector`

When we generalized `vector` from a class “`vector` of `double`” to a template “`vector` of `T`,” we didn't review the definitions of `push_back()`, `resize()`, and `reserve()`. We must do that now because as they are defined in §19.2.2 and §19.2.3 they make assumptions that are true for `doubles`, but not true for all types that we'd like to use as `vector` element types:

- How do we handle a `vector<X>` where `X` doesn't have a default value?
- How do we ensure that elements are destroyed when we are finished with them?

 Must we solve those problems? We could say, “Don't try to make `vectors` of types without default values” and “Don't use `vectors` for types with destructors in ways that cause problems.” For a facility that is aimed at “general use,” such restrictions are annoying to users and give the impression that the designer hasn't thought the problem through or doesn't really care about users. Often, such suspicions are correct, but the designers of the standard library didn't leave these warts in place. To mirror the standard library `vector`, we must solve these two problems.

We can handle types without a default by giving the user the option to specify the value to be used when we need a “default value”:

```
template<typename T> void vector<T>::resize(int newsize, T def = T());
```

That is, use `T()` as the default value unless the user says otherwise. For example:

```
vector<double> v1;
v1.resize(100);           // add 100 copies of double(), that is, 0.0
v1.resize(200, 0.0);     // add 100 copies of 0.0 — mentioning 0.0 is redundant
v1.resize(300, 1.0);     // add 100 copies of 1.0

struct No_default {
    No_default(int);      // the only constructor for No_default
    ...
};

vector<No_default> v2(10); // error: tries to make 10 No_default()'s
vector<No_default> v3;
```

```
v3.resize(100, No_default(2));           // add 100 copies of No_default(2)
v3.resize(200);                         // error: tries to add 100 No_default()s
```

The destructor problem is harder to address. Basically, we need to deal with something really awkward: a data structure consisting of some initialized data and some uninitialized data. So far, we have gone a long way to avoid uninitialized data and the programming errors that usually accompany it. Now – as implementers of **vector** – we have to face that problem so that we – as users of **vector** – don't have to in our applications.

First, we need to find a way of getting and manipulating uninitialized storage. Fortunately, the standard library provides a class **allocator**, which provides uninitialized memory. A slightly simplified version looks like this:

```
template<typename T> class allocator {
public:
    // . . .
    T* allocate(int n);                  // allocate space for n objects of type T
    void deallocate(T* p, int n);        // deallocate n objects of type T starting at p

    void construct(T* p, const T& v);    // construct a T with the value v in p
    void destroy(T* p);                 // destroy the T in p
};
```

Should you need the full story, have a look in *The C++ Programming Language, <memory>* (§B.1.1), or the standard. However, what is presented here shows the four fundamental operations that allow us to

- Allocate memory of a size suitable to hold an object of type **T** without initializing
- Construct an object of type **T** in uninitialized space
- Destroy an object of type **T**, thus returning its space to the uninitialized state
- Deallocate uninitialized space of a size suitable for an object of type **T**

Unsurprisingly, an **allocator** is exactly what we need for implementing **vector<T> ::reserve()**. We start by giving **vector** an allocator parameter:

```
template<typename T, typename A = allocator<T>> class vector {
    A alloc;                      // use allocate to handle memory for elements
    // . . .
};
```

Except for providing an **allocator** – and using the standard one by default instead of using **new** – all is as before. As users of **vector**, we can ignore allocators until we find ourselves needing a **vector** that manages memory for its elements in some unusual way. As implementers of **vector** and as students trying to understand fundamental problems and learn fundamental techniques, we must see how a **vector** can deal with uninitialized memory and present properly constructed objects to its users. The only code affected is **vector<T>::reserve()**:

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;           // never decrease allocation
    T* p = alloc.allocate(newalloc);        // allocate new space
    for (int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]);    // copy
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);          // destroy
    alloc.deallocate(elem,space);            // deallocate old space
    elem = p;
    space = newalloc;
}
```

We move an element to the new space by constructing a copy in uninitialized space and then destroying the original. We can't use assignment because for types such as **string**, assignment assumes that the target area has been initialized.

Given **reserve()**, **vector<T,A>::push_back()** is simple to write:

```
template<typename T, typename A>
void vector<T,A>::push_back(const T& val)
{
    if (space==0) reserve(8);           // start with space for 8 elements
    else if (sz==space) reserve(2*space); // get more space
    alloc.construct(&elem[sz],val);       // add val at end
    ++sz;                           // increase the size
}
```

Similarly, **vector<T,A>::resize()** is not too difficult:

```
template<typename T, typename A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    for (int i=sz; i<newszie; ++i) alloc.construct(&elem[i],val); // construct
```

```

        for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]);      // destroy
        sz = newsize;
    }

```

Note that because some types do not have a default constructor, we again provide the option to supply a value to be used as an initial value for new elements.

The other new thing here is the destruction of “surplus elements” in the case where we are resizing to a smaller **vector**. Think of the destructor as turning a typed object into “raw memory.”

“Messing with allocators” is pretty advanced stuff, and tricky. Leave it alone until you are ready to become an expert.



19.4 Range checking and exceptions

We look at our **vector** so far and find (with horror?) that access isn’t range checked. The implementation of **operator[]** is simply

```

template<typename T, typename A> T& vector<T,A>::operator[](int n)
{
    return elem[n];
}

```

So, consider:

```

vector<int> v(100);
v[-200] = v[200];           // oops!
int i;
cin>>i;
v[i] = 999;                 // maul an arbitrary memory location

```

This code compiles and runs, accessing memory not owned by our **vector**. This could mean big trouble! In a real program, such code is unacceptable. Let’s try to improve our **vector** to deal with this problem. The simplest approach would be to add a checked access operation, called **at()**:

```

struct out_of_range { /* ... */ }; // class used to report range access errors

template<typename T, typename A = allocator<T>> class vector {
    // ...
    T& at(int n);                      // checked access
    const T& at(int n) const;           // checked access
}

```

```

T& operator[](int n);           // unchecked access
const T& operator[](int n) const; // unchecked access
// ...
};

template<typename T, typename A > T& vector<T,A>::at(int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<typename T, typename A > T& vector<T,A>::operator[](int n)
    // as before
{
    return elem[n];
}

```

Given that, we could write

```

void print_some(vector<int>& v)
{
    int i = -1;
    while(cin>>i && i!= -1)
        try {
            cout << "v[" << i << "]==" << v.at(i) << "\n";
        }
        catch(out_of_range) {
            cout << "bad index: " << i << "\n";
        }
}

```

Here, we use `at()` to get range-checked access, and we catch `out_of_range` in case of an illegal access.

The general idea is to use subscripting with `[]` when we know that we have a valid index and `at()` when we might have an out-of-range index.

19.4.1 An aside: design considerations

So far, so good, but why didn't we just add the range check to `operator[]()`? Well, the standard library `vector` provides checked `at()` and unchecked `operator[]()` as shown here. Let's try to explain how that makes some sense. There are basically four arguments:

1. *Compatibility*: People have been using unchecked subscripting since long before C++ had exceptions.
2. *Efficiency*: You can build a checked-access operator on top of an optimally fast unchecked-access operator, but you cannot build an optimally fast access operator on top of a checked-access operator.
3. *Constraints*: In some environments, exceptions are unacceptable.
4. *Optional checking*: The standard doesn't actually say that you can't range check `vector`, so if you want checking, use an implementation that checks.

19.4.1.1 Compatibility

People really, really don't like to have their old code break. For example, if you have a million lines of code, it could be a very costly affair to rework it all to use exceptions correctly. We can argue that the code would be better for the extra work, but then we are not the ones who have to pay (in time or money). Furthermore, maintainers of existing code usually argue that unchecked code may be unsafe in principle, but their particular code has been tested and used for years and all the bugs have already been found. We can be skeptical about that argument, but again nobody who hasn't had to make such decisions about real code should be too judgmental. Naturally, there was no code using the standard library `vector` before it was introduced into the C++ standard, but there were many millions of lines of code that used very similar `vectors` that (being pre-standard) didn't use exceptions. Much of that code was later modified to use the standard.

19.4.1.2 Efficiency

Yes, range checking can be a burden in extreme cases, such as buffers for network interfaces and matrices in high-performance scientific computations. However, the cost of range checking is rarely a concern in the kind of "ordinary computing" that most of us spend most of our time on. Thus, we recommend and use a range-checked implementation of `vector` whenever we can.

19.4.1.3 Constraints

Again, the argument holds for some programmers and some applications. In fact, it holds for a whole lot of programmers and shouldn't be lightly ignored. However, if you are starting a new program in an environment that doesn't involve hard real time (see §25.2.1), prefer exception-based error handling and range-checked `vectors`.

19.4.1.4 Optional checking

The ISO C++ standard simply states that out-of-range `vector` access is not guaranteed to have any specific semantics, and that such access should be avoided. It

is perfectly standards-conforming to throw an exception when a program tries an out-of-range access. So, if you like `vector` to throw and don't need to be concerned by the first three reasons for a particular application, use a range-checked implementation of `vector`. That's what we are doing for this book.

The long and the short of this is that real-world design can be messier than we would prefer, but there are ways of coping.

19.4.2 A confession: macros

Like our `vector`, most implementations of the standard library `vector` don't guarantee to range check the subscript operator `[]` but provide `at()` that checks. So where did those `std::out_of_range` exceptions in our programs come from? Basically, we chose "option 4" from §19.4.1: a `vector` implementation is not obliged to range check `[]`, but it is not prohibited from doing so either, so we arranged for checking to be done. What you might have been using is our debug version, called `Vector`, which does check `[]`. That's what we use when we develop code. It cuts down on errors and debug time at little cost to performance:

```
struct Range_error : out_of_range {    // enhanced vector range error reporting
    int index;
    Range_error(int i) :out_of_range("Range error"), index(i) { }
};

template<typename T> struct Vector : public std::vector<T> {
    using size_type = typename std::vector<T>::size_type;
    using vector<T>::vector;           // use vector<T>'s constructors (§20.5)

    T& operator[](size_type i)          // rather than return at(i);
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }

    const T& operator[](size_type i) const
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
};
```

We use `Range_error` to make the offending index available for debugging. Deriving from `std::vector` gives us all of `vector`'s member functions for `Vector`. The first `using` introduces a convenient synonym for `std::vector`'s `size_type`; see §20.5. The second `using` gives us all of `vector`'s constructors for `Vector`.

This **Vector** has been useful in debugging nontrivial programs. The alternative is to use a systematically checked implementation of the complete standard library **vector** – in fact, that *may* indeed be what you have been using; we have no way of knowing exactly what degree of checking your compiler and library provide (beyond what the standard guarantees).

In **std_lib_facilities.h**, we use the nasty trick (a macro substitution) of redefining **vector** to mean **Vector**:

```
// disgusting macro hack to get a range-checked vector:  
#define vector Vector
```

That means that whenever you wrote **vector**, the compiler saw **Vector**. This trick is nasty because what you see looking at the code is not what the compiler sees. In real-world code, macros are a significant source of obscure errors (§27.8, §A.17.2).

We did the same to provide range-checked access for **string**.

Unfortunately, there is no standard, portable, and clean way of getting range checking from an implementation of **vector**'s []. It is, however, possible to do a much cleaner and more complete job of a range-checked **vector** (and **string**) than we did. However, that usually involves replacement of a vendor's standard library implementation, adjusting installation options, or messing with standard library source code. None of those options is appropriate for a beginner's first week of programming – and we used **string** in Chapter 2.

19.5 Resources and exceptions

So, **vector** can throw exceptions, and we recommend that when a function cannot perform its required action, it throws an exception to tell that to its callers (Chapter 5). Now is the time to consider what to do when we write code that must deal with exceptions thrown by **vector** operations and other functions that we call. The naive answer – “Use a **try**-block to catch the exception, write an error message, and then terminate the program” – is too crude for most nontrivial systems.

One of the fundamental principles of programming is that if we acquire a resource, we must – somehow, directly or indirectly – return it to whatever part of the system manages that resource. Examples of resources are

- Memory
- Locks
- File handles
- Thread handles
- Sockets
- Windows



Basically, we define a resource as something that is acquired and must be given back (released) or reclaimed by some “resource manager.” The simplest example is free-store memory that we acquire using `new` and return to the free store using `delete`. For example:

```
void suspicious(int s, int x)
{
    int* p = new int[s];      // acquire memory
    // ...
    delete[] p;              // release memory
}
```

As we saw in §17.4.6, we have to remember to release the memory, and that’s not always easy to do. When we add exceptions to the picture, resource leaks can become common; all it takes is ignorance or some lack of care. In particular, we view code, such as `suspicious()`, that explicitly uses `new` and assigns the resulting pointer to a local variable with great suspicion.

We call an object, such as a `vector`, that is responsible for releasing a resource the *owner* or a *handle* of the resource for which it is responsible.



19.5.1 Potential resource management problems



One reason for suspicion of apparently innocuous pointer assignments such as

```
int* p = new int[s];      // acquire memory
```

is that it can be hard to verify that the `new` has a corresponding `delete`. At least `suspicious()` has a `delete[] p;` statement that might release the memory, but let’s imagine a few things that might cause that release not to happen. What could we put in the `...` part to cause a memory leak? The problematic examples we find should give you cause for thought and make you suspicious of such code. They should also make you appreciate the simple and powerful alternative to such code.

Maybe `p` no longer points to the object when we get to the `delete`:

```
void suspicious(int s, int x)
{
    int* p = new int[s];      // acquire memory
    // ...
    if (x) p = q;            // make p point to another object
    // ...
    delete[] p;              // release memory
}
```

We put that **if (x)** there to be sure that you couldn't know whether we had changed the value of **p**. Maybe we never get to the **delete**:

```
void suspicious(int s, int x)
{
    int* p = new int[s];           // acquire memory
    // ...
    if (x) return;
    // ...
    delete[] p;                 // release memory
}
```

Maybe we never get to the **delete** because we threw an exception:

```
void suspicious(int s, int x)
{
    int* p = new int[s];           // acquire memory
    vector<int> v;
    // ...
    if (x) p[x] = v.at(x);
    // ...
    delete[] p;                 // release memory
}
```

It is this last possibility that concerns us most here. When people first encounter this problem, they tend to consider it a problem with exceptions rather than a resource management problem. Having misclassified the root cause, they come up with a solution that involves catching the exception:

```
void suspicious(int s, int x)      // messy code
{
    int* p = new int[s];           // acquire memory
    vector<int> v;
    // ...
    try {
        if (x) p[x] = v.at(x);
        // ...
    } catch (...) {                // catch every exception
        delete[] p;               // release memory
        throw;                  // re-throw the exception
    }
    // ...
    delete[] p;                 // release memory
}
```

This solves the problem at the cost of some added code and a duplication of the resource release code (here, `delete[] p;`). In other words, this solution is ugly; worse, it doesn't generalize well. Consider acquiring more resources:

```
void suspicious(vector<int>& v, int s)
{
    int* p = new int[s];
    vector<int> v1;
    // ...
    int* q = new int[s];
    vector<double> v2;
    // ...
    delete[] p;
    delete[] q;
}
```

Note that if `new` fails to find free-store memory to allocate, it will throw the standard library exception `bad_alloc`. The `try . . . catch` technique works for this example also, but you'll need several `try`-blocks, and the code is repetitive and ugly. We don't like repetitive and ugly code because "repetitive" translates into code that is a maintenance hazard, and "ugly" translates into code that is hard to get right, hard to read, and a maintenance hazard.

TRY THIS



Add `try`-blocks to this last example to ensure that all resources are properly released in all cases where an exception might be thrown.

19.5.2 Resource acquisition is initialization

Fortunately, we don't need to plaster our code with complicated `try . . . catch` statements to deal with potential resource leaks. Consider:

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // ...
}
```



This is better. More importantly, it is *obviously* better. The resource (here, free-store memory) is acquired by a constructor and released by the matching destruc-

tor. We actually solved this particular “exception problem” when we solved the memory leak problems for vectors. The solution is general; it applies to all kinds of resources: acquire a resource in the constructor for some object that manages it, and release it again in the matching destructor. Examples of resources that are usually best dealt with in this way include database locks, sockets, and I/O buffers (**iostreams** do it for you). This technique is usually referred to by the awkward phrase “Resource Acquisition Is Initialization,” abbreviated to RAI^I.

Consider the example above. Whichever way we leave **f()**, the destructors for **p** and **q** are invoked appropriately: since **p** and **q** aren’t pointers, we can’t assign to them, a **return**-statement will not prevent the invocation of destructors, and neither will throwing an exception. This general rule holds: when the thread of execution leaves a scope, the destructors for every fully constructed object and sub-object are invoked. An object is considered constructed when its constructor completes. Exploring the detailed implications of those two statements might cause a headache, but they simply mean that constructors and destructors are invoked as needed.

In particular, use **vector** rather than explicit **new** and **delete** when you need a nonconstant amount of storage within a scope.

19.5.3 Guarantees

What can we do where we can’t keep the **vector** within a single scope (and its sub-scopes)? For example:

```
vector<int>* make_vec() // make a filled vector
{
    vector<int>* p = new vector<int>; // we allocate on free store
    // . . . fill the vector with data; this may throw an exception . .
    return p;
}
```

This is an example of a common kind of code: we call a function to construct a complicated data structure and return that data structure as the result. The snag is that if an exception is thrown while “filling” the **vector**, **make_vec()** leaks that **vector**. An unrelated problem is that if the function succeeds, someone will have to **delete** the object returned by **make_vec()** (see §17.4.6).

We can add a **try**-block to deal with the possibility of a **throw**:

```
vector<int>* make_vec() // make a filled vector
{
    vector<int>* p = new vector<int>; // we allocate on free store
    try {
        // fill the vector with data; this may throw an exception
        return p;
    }
```

```
catch (...) {
    delete p;           // do our local cleanup
    throw;             // re-throw to allow our caller to deal with the fact
                       // that make_vec() couldn't do what was
                       // required of it
}
```

This `make_vec()` function illustrates a very common style of error handling: it tries to do its job and if it can't, it cleans up any local resources (here the `vector` on the free store) and indicates failure by throwing an exception. Here, the exception thrown is one that some other function (e.g., `vector::at()`) threw; `make_vec()` simply re-throws it using `throw;`. This is a simple and effective way of dealing with errors and can be used systematically.

- *The basic guarantee:* The purpose of the `try...catch` code is to ensure that `make_vec()` either succeeds or throws an exception without having leaked any resources. That's often called the *basic guarantee*. All code that is part of a program that we expect to recover from an exception `throw` should provide the basic guarantee. All standard library code provides the basic guarantee.
- *The strong guarantee:* If, in addition to providing the basic guarantee, a function also ensures that all observable values (all values not local to the function) are the same after failure as they were when we called the function, that function is said to provide the *strong guarantee*. The strong guarantee is the ideal when we write a function: either the function succeeded at doing everything it was asked to do or else nothing happened except that an exception was thrown to indicate failure.
- *The no-throw guarantee:* Unless we could do simple operations without any risk of failing and throwing an exception, we would not be able to write code to meet the basic guarantee and the strong guarantee. Fortunately, essentially all built-in facilities in C++ provide the no-throw guarantee: they simply can't throw. To avoid throwing, simply avoid `throw` itself, `new`, and `dynamic_cast` of reference types (§A.5.7).

The basic guarantee and the strong guarantee are most useful for thinking about correctness of programs. RAII is essential for implementing code written according to those ideals simply and with high performance.

Naturally, we should always avoid undefined (and usually disastrous) operations, such as dereferencing 0, dividing by 0, and accessing an array beyond its range. Catching exceptions does not save you from violations of the fundamental language rules.

19.5.4 unique_ptr

So, `make_vec()` is a useful kind of function that obeys the basic rules for good resource management in the presence of exceptions. It provides the basic guarantee – as all good functions should – when we want to recover from exception throws. Unless something nasty is done with nonlocal data in the “fill the `vector` with data” part, it even provides the strong guarantee. However, that `try...catch` code is still ugly. The solution is obvious: somehow we must use RAII; that is, we need to provide an object to hold that `vector<int>` so that it can delete the `vector` if an exception occurs. In `<memory>`, the standard library provides `unique_ptr` for that:

```
vector<int>* make_vec()           // make a filled vector
{
    unique_ptr<vector<int>> p {new vector<int>}; // allocate on free store
    // ... fill the vector with data; this may throw an exception ...
    return p.release();           // return the pointer held by p
}
```

A `unique_ptr` is an object that holds a pointer. We immediately initialize it with the pointer we got from `new`. You can use `->` and `*` on a `unique_ptr` exactly like a built-in pointer (e.g., `p->at(2)` or `(*p).at(2)`), so we think of `unique_ptr` as a kind of pointer. However, the `unique_ptr` owns the object pointed to: when the `unique_ptr` is destroyed, it `deletes` the object it points to. That means that if an exception is thrown while the `vector<int>` is being filled, or if we return prematurely from `make_vec`, the `vector<int>` is properly destroyed. The `p.release()` extracts the contained pointer (to the `vector<int>`) from `p` so that we can return it, and it also makes `p` hold the `nullptr` so that destroying `p` (as is done by the `return`) does not destroy anything.

Using `unique_ptr` simplifies `make_vec()` immensely. Basically, it makes `make_vec()` as simple as the naive but unsafe version. Importantly, having `unique_ptr` allows us to repeat our recommendation to look upon explicit `try`-blocks with suspicion; most can – as in `make_vec()` – be replaced by some variant of the “Resource Acquisition Is Initialization” technique.

The version of `make_vec()` that uses a `unique_ptr` is fine, except that it still returns a pointer, so that someone still has to remember to `delete` that pointer. Returning a `unique_ptr` would solve that:

```
unique_ptr<vector<int>> make_vec()           // make a filled vector
{
    unique_ptr<vector<int>> p {new vector<int>}; // allocate on free store
    // ... fill the vector with data; this may throw an exception ...
    return p;
}
```

A `unique_ptr` is very much like an ordinary pointer, but it has one significant restriction: you cannot assign one `unique_ptr` to another to get two `unique_ptr`s to the same object. That has to be so, or confusion could arise about which `unique_ptr` owned the pointed-to object and had to `delete` it. For example:

```
void no_good()
{
    unique_ptr<X> p { new X };
    unique_ptr<X> q {p};      // error: fortunately
    // ...
} // here p and q both delete the X
```

If you want to have a “smart” pointer that both guarantees deletion and can be copied, use a `shared_ptr` (§B.6.5). However, that is a more heavyweight solution that involves a use count to ensure that the last copy destroyed destroys the object referred to.

A `unique_ptr` has the interesting property of having no overhead compared to an ordinary pointer.

19.5.5 Return by moving

The technique of returning a lot of information by placing it on the free store and returning a pointer to it is very common. It is also a source of a lot of complexity and one of the major sources of memory management errors: Who `deletes` a pointer to the free store returned from a function? Are we sure that a pointer to an object on the free store is properly `deleted` in case of an exception? Unless we are systematic about the management of pointers (or use “smart” pointers such as `unique_ptr` and `shared_ptr`), the answer will be something like “Well, we think so,” and that’s not good enough.

Fortunately, when we added move operations to `vector`, we solved that problem for `vectors`: just use a move constructor to get the ownership of the elements out of the function. For example:

```
vector<int> make_vec() // make a filled vector
{
    vector<int> res;
    // ... fill the vector with data; this may throw an exception ...
    return res;           // the move constructor efficiently transfers ownership
}
```

This (final) version of `make_vec()` is the simplest and the one we recommend. The move solution generalizes to all containers and further still to all resource handles. For example, `fstream` uses this technique to keep track of file handles. The move

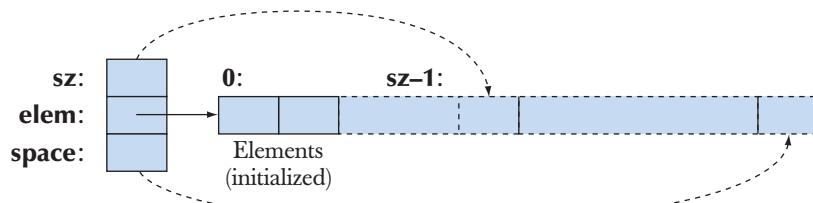
solution is simple and general. Using resource handles simplifies code and eliminates a major source of errors. Compared to the direct use of pointers, the run-time overhead of using such handles is nothing, or very minor and predictable.

19.5.6 RAII for vector

Even using a smart pointer, such as `unique_ptr`, may seem to be a bit ad hoc. How can we be sure that we have spotted all pointers that require protection? How can we be sure that we have released all pointers to objects that should not be destroyed at the end of a scope? Consider `reserve()` from §19.3.7:

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;      // never decrease allocation
    T* p = alloc.allocate(newalloc);  // allocate new space
    for (int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]); // copy
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);        // destroy
    alloc.deallocate(elem,space);    // deallocate old space
    elem = p;
    space = newalloc;
}
```

Note that the copy operation for an old element, `alloc.construct(&p[i],elem[i])`, might throw an exception. So, `p` is an example of the problem we warned about in §19.5.1. Ouch! We could apply the `unique_ptr` solution. A better solution is to step back and realize that “memory for a `vector`” is a resource; that is, we can define a class `vector_base` to represent the fundamental concept we have been using all the time, the picture with the three elements defining a `vector`’s memory use:



In code, that is (after adding the allocator for completeness)

```
template<typename T, typename A>
struct vector_base {
    A alloc;           // allocator
    T* elem;          // start of allocation
```

```

int sz;           // number of elements
int space;        // amount of allocated space

vector_base(const A& a, int n)
    : alloc{a}, elem{alloc.allocate(n)}, sz{n}, space{n}{ }
~vector_base() { alloc.deallocate(elem,space); }

};

```

Note that **vector_base** deals with memory rather than (typed) objects. Our **vector** implementation can use that to hold objects of the desired element type. Basically, **vector** is simply a convenient interface to **vector_base**:

```

template<typename T, typename A = allocator<T>>
class vector : private vector_base<T,A> {
public:
    // ...
};

```

We can then rewrite **reserve()** to something simpler and more correct:

```

template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=this->space) return;      // never decrease allocation
    vector_base<T,A> b(this->alloc,newalloc); // allocate new space
    uninitialized_copy(b.elem,&b.elem[this->sz],this->elem); // copy
    for (int i=0; i<this->sz; ++i)
        this->alloc.destroy(&this->elem[i]); // destroy old
    swap<vector_base<T,A>>(*this,b);       // swap representations
}

```

We use the standard library function **uninitialized_copy** to construct copies of the elements from **b** because it correctly handles throws from an element copy constructor and because calling a function is simpler than writing a loop. When we exit **reserve()**, the old allocation is automatically freed by **vector_base**'s destructor if the copy operation succeeded. If instead that exit is caused by the copy operation throwing an exception, the new allocation is freed. The **swap()** function is a standard library algorithm (from **<algorithm>**) that exchanges the value of two objects. We used **swap<vector_base<T,A>>(*this,b)** rather than the simpler **swap(*this,b)** because ***this** and **b** are of different types (**vector** and **vector_base**, respectively), so we had to be explicit about which swap specialization we wanted. Similarly, we have to explicitly use **this->** when we refer to a member of the base

class `vector_base<T,A>` from a member of the derived class `vector<T,A>`, such as `vector<T,A>::reserve()`.

TRY THIS



Modify `reserve` to use `unique_ptr`. Remember to release before returning. Compare that solution to the `vector_base` one. Consider which is easier to write and which is easier to get correct.



Drill

1. Define `template<typename T> struct S { T val; };`.
2. Add a constructor, so that you can initialize with a `T`.
3. Define variables of types `S<int>`, `S<char>`, `S<double>`, `S<string>`, and `S<vector<int>>`; initialize them with values of your choice.
4. Read those values and print them.
5. Add a function template `get()` that returns a reference to `val`.
6. Put the definition of `get()` outside the class.
7. Make `val` private.
8. Do 4 again using `get()`.
9. Add a `set()` function template so that you can change `val`.
10. Replace `set()` with an `S<T>::operator=(const T&)`. Hint: Much simpler than §19.2.5.
11. Provide `const` and non-`const` versions of `get()`.
12. Define a function `template<typename T> read_val(T& v)` that reads from `cin` into `v`.
13. Use `read_val()` to read into each of the variables from 3 except the `S<vector<int>>` variable.
14. Bonus: Define input and output operators (`>>` and `<<`) for `vector<T>s`. For both input and output use a `{ val, val, val }` format. That will allow `read_val()` to also handle the `S<vector<int>>` variable.

Remember to test after each step.

Review

1. Why would we want to change the size of a `vector`?
2. Why would we want to have different element types for different `vectors`?

3. Why don't we just always define a **vector** with a large enough size for all eventualities?
4. How much spare space do we allocate for a new **vector**?
5. When must we copy **vector** elements to a new location?
6. Which **vector** operations can change the size of a **vector** after construction?
7. What is the value of a **vector** after a copy?
8. Which two operations define copy for **vector**?
9. What is the default meaning of copy for class objects?
10. What is a template?
11. What are the two most useful types of template arguments?
12. What is generic programming?
13. How does generic programming differ from object-oriented programming?
14. How does **array** differ from **vector**?
15. How does **array** differ from the built-in array?
16. How does **resize()** differ from **reserve()**?
17. What is a resource? Define and give examples.
18. What is a resource leak?
19. What is RAI? What problem does it address?
20. What is **unique_ptr** good for?

Terms

#define	owner	specialization
at()	push_back()	strong guarantee
basic guarantee	RAII	template
exception	resize()	template parameter
guarantees	resource	this
handle	re-throw	throw;
instantiation	self-assignment	unique_ptr
macro	shared_ptr	

Exercises

For each exercise, create and test (with output) a couple of objects of the defined classes to demonstrate that your design and implementation actually do what you think they do. Where exceptions are involved, this can require careful thought about where errors can occur.

1. Write a template function **f()** that adds the elements of one **vector<T>** to the elements of another; for example, **f(v1,v2)** should do **v1[i]+=v2[i]** for each element of **v1**.
2. Write a template function that takes a **vector<T> vt** and a **vector<U> vu** as arguments and returns the sum of all **vt[i]*vu[i]**s.

3. Write a template class **Pair** that can hold a pair of values of any type. Use this to implement a simple symbol table like the one we used in the calculator (§7.8).
4. Modify class **Link** from §17.9.3 to be a template with the type of value as the template argument. Then redo exercise 13 from Chapter 17 with **Link<God>**.
5. Define a class **Int** having a single member of class **int**. Define constructors, assignment, and operators **+**, **-**, *****, **/** for it. Test it, and improve its design as needed (e.g., define operators **<<** and **>>** for convenient I/O).
6. Repeat the previous exercise, but with a class **Number<T>** where **T** can be any numeric type. Try adding **%** to **Number** and see what happens when you try to use **%** for **Number<double>** and **Number<int>**.
7. Try your solution to exercise 2 with some **Numbers**.
8. Implement an allocator (§19.3.7) using the basic allocation functions **malloc()** and **free()** (§B.11.4). Get **vector** as defined by the end of §19.4 to work for a few simple test cases. Hint: Look up “placement **new**” and “explicit call of destructor” in a complete C++ reference.
9. Re-implement **vector::operator=()** (§19.2.5) using an allocator (§19.3.7) for memory management.
10. Implement a simple **unique_ptr** supporting only a constructor, destructor, **->**, *****, and **release()**. In particular, don’t try to implement an assignment or a copy constructor.
11. Design and implement a **counted_ptr<T>** that is a type that holds a pointer to an object of type **T** and a pointer to a “use count” (an **int**) shared by all counted pointers to the same object of type **T**. The use count should hold the number of counted pointers pointing to a given **T**. Let the **counted_ptr**’s constructor allocate a **T** object and a use count on the free store. Let **counted_ptr**’s constructor take an argument to be used as the initial value of the **T** elements. When the last **counted_ptr** for a **T** is destroyed, **counted_ptr**’s destructor should **delete** the **T**. Give the **counted_ptr** operations that allow us to use it as a pointer. This is an example of a “smart pointer” used to ensure that an object doesn’t get destroyed until after its last user has stopped using it. Write a set of test cases for **counted_ptr** using it as an argument in calls, container elements, etc.
12. Define a **File_handle** class with a constructor that takes a **string** argument (the file name), opens the file in the constructor, and closes it in the destructor.
13. Write a **Tracer** class where its constructor prints a string and its destructor prints a string. Give the strings as constructor arguments. Use it to see where RAII management objects will do their job (i.e., experiment with **Tracers** as local objects, member objects, global objects, objects allocated

- by `new`, etc.). Then add a copy constructor and a copy assignment so that you can use `Tracer` objects to see when copying is done.
14. Provide a GUI interface and a bit of graphical output to the “Hunt the Wumpus” game from the exercises in Chapter 18. Take the input in an input box and display a map of the part of the cave currently known to the player in a window.
 15. Modify the program from the previous exercise to allow the user to mark rooms based on knowledge and guesses, such as “maybe bats” and “bottomless pit.”
 16. Sometimes, it is desirable that an empty `vector` be as small as possible. For example, someone might use `vector<vector<vector<int>>` a lot but have most element vectors empty. Define a `vector` so that `sizeof(vector<int>) == sizeof(int*)`, that is, so that the `vector` itself consists only of a pointer to a representation consisting of the elements, the number of elements, and the `space` pointer.

Postscript

Templates and exceptions are immensely powerful language features. They support programming techniques of great flexibility – mostly by allowing people to separate concerns, that is, to deal with one problem at a time. For example, using templates, we can define a container, such as `vector`, separately from the definition of an element type. Similarly, using exceptions, we can write the code that detects and signals an error separately from the code that handles that error. The third major theme of this chapter, changing the size of a `vector`, can be seen in a similar light: `push_back()`, `resize()`, and `reserve()` allow us to separate the definition of a `vector` from the specification of its size.



Containers and Iterators

“Write programs that do one thing
and do it well. Write programs
to work together.”

—Doug McIlroy

This chapter and the next present the STL, the containers and algorithms part of the C++ standard library. The STL is an extensible framework for dealing with data in a C++ program. After a first simple example, we present the general ideals and the fundamental concepts. We discuss iteration, linked-list manipulation, and STL containers. The key notions of sequence and iterator are used to tie containers (data) together with algorithms (processing). This chapter lays the groundwork for the general, efficient, and useful algorithms presented in the next chapter. As an example, it also presents a framework for text editing as a sample application.

20.1 Storing and processing data	20.6 An example: a simple text editor
20.1.1 Working with data	20.6.1 Lines
20.1.2 Generalizing code	20.6.2 Iteration
20.2 STL ideals	20.7 <code>vector</code>, <code>list</code>, and <code>string</code>
20.3 Sequences and iterators	20.7.1 <code>insert</code> and <code>erase</code>
20.3.1 Back to the example	20.8 Adapting our <code>vector</code> to the STL
20.4 Linked lists	20.9 Adapting built-in arrays to the STL
20.4.1 List operations	20.10 Container overview
20.4.2 Iteration	20.10.1 Iterator categories
20.5 Generalizing <code>vector</code> yet again	
20.5.1 Container traversal	
20.5.2 <code>auto</code>	

20.1 Storing and processing data

Before looking into dealing with larger collections of data items, let's consider a simple example that points to ways of handling a large class of data-processing problems. Jack and Jill are each measuring vehicle speeds, which they record as floating-point values. Jack was brought up as a C programmer and stores his values in an array, whereas Jill stores hers in a `vector`. Now we'd like to use their data in our program. How might we do this?

We could have Jack's and Jill's programs write out the values to a file and then read them back into our program. That way, we are completely insulated from their choices of data structures and interfaces. Often, such isolation is a good idea, and if that's what we decide to do we can use the techniques from Chapters 10–11 for input and a `vector<double>` for our calculations.

But, what if using files isn't a good option for the task we want to do? Let's say that the data-gathering code is designed to be invoked as a function call to deliver a new set of data every second. Once a second, we call Jack's and Jill's functions to deliver data for us to process:

```
double* get_from_jack(int* count); // Jack puts doubles into an array and
                                   // returns the number of elements in *count
vector<double>* get_from_jill();   // Jill fills the vector

void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
```

```

vector<double>* jill_data = get_from_jill();
// . . . process . . .
delete[] jack_data;
delete jill_data;
}

```

The assumption is that the data is stored on the free store and that we should delete it when we are finished using it. Another assumption is that we can't rewrite Jack's and Jill's code, or wouldn't want to.

20.1.1 Working with data

Clearly, this is a somewhat simplified example, but it is not dissimilar to a vast number of real-world problems. If we can handle this example elegantly, we can handle a huge number of common programming problems. The fundamental problem here is that we don't control the way in which our "data suppliers" store the data they give us. It's our job to either work with the data in the form in which we get it or to read it and store it the way we like better.

What do we want to do with that data? Sort it? Find the highest value? Find the average value? Find every value over 65? Compare Jill's data with Jack's? See how many readings there were? The possibilities are endless, and when writing a real program we will simply do the computation required. Here, we just want to do something to learn how to handle data and do computations involving lots of data. Let's first do something really simple: find the element with the highest value in each data set. We can do that by inserting this code in place of the "... process ..." comment in **fct()**:

```

// . . .
double h = -1;
double* jack_high; // jack_high will point to the element with the highest value
double* jill_high; // jill_high will point to the element with the highest value
for (int i=0; i<jack_count; ++i)
    if (h<jack_data[i]) {
        jack_high = &jack_data[i]; // save address of largest element
        h = jack_data[i]; // update "largest element"
    }

h = -1;
for (int i=0; i<jill_data->size(); ++i)
    if (h<(*jill_data)[i]) {
        jill_high = &(*jill_data)[i]; // save address of largest element
        h = (*jill_data)[i]; // update "largest element"
    }

```

```

cout << "Jill's max: " << *jill_high
<< "; Jack's max: " << *jack_high;

// ...

```

Note the ugly notation we use to access Jill's data: `(*jill_data)[i]`. The function `get_from_jill()` returns a pointer to a `vector`, a `vector<double>*`. To get to the data, we first have to dereference the pointer to get to the `vector`, `*jill_data`, then we can subscript that. However, `*jill_data[i]` isn't what we want; that means `*(jill_data[i])` because `[]` binds tighter than `*`, so we need the parentheses around `*jill_data` and get `(*jill_data)[i]`.

TRY THIS



If you were able to change Jill's code, how would you redesign its interface to get rid of the ugliness?

20.1.2 Generalizing code



What we would like is a uniform way of accessing and manipulating data so that we don't have to write our code differently each time we get data presented to us in a slightly different way. Let's look at Jack's and Jill's code as examples of how we can make our code more abstract and uniform.

Obviously, what we do for Jack's data strongly resembles what we do for Jill's. However, there are some annoying differences: `jack_count` vs. `jill_data->size()` and `jack_data[i]` vs. `(*jill_data)[i]`. We could eliminate the latter difference by introducing a reference:

```

vector<double>& v = *jill_data;
for (int i=0; i<v.size(); ++i)
    if (h<v[i]) {
        jill_high = &v[i];
        h = v[i];
    }
}

```

This is tantalizingly close to the code for Jack's data. What would it take to write a function that could do the calculation for Jill's data as well as for Jack's? We can think of several ways (see exercise 3), but for reasons of generality which will become clear over the next two chapters, we chose a solution based on pointers:

```

double* high(double* first, double* last)
// return a pointer to the element in [first, last) that has the highest value

```

```

{
    double h = -1;
    double* high;
    for(double* p = first; p!=last; ++p)
        if (h<*p) { high = p; h = *p; }
    return high;
}

```

Given that, we can write

```

double* jack_high = high(jack_data,jack_data+jack_count);
vector<double>& v = *jill_data;
double* jill_high = high(&v[0],&v[0]+v.size());

```

This looks better. We don't introduce so many variables and we write the loop and the loop body only once (in `high()`). If we want to know the highest values, we can look at `*jack_high` and `*jill_high`. For example:

```

cout << "Jill's max: " << *jill_high
     << "; Jack's max: " << *jack_high;

```

Note that `high()` relies on a vector storing its elements in an array, so that we can express our “find highest element” algorithm in terms of pointers into an array.

TRY THIS



We left two potentially serious errors in this little program. One can cause a crash, and the other will give wrong answers if `high()` is used in many other programs where it might have been useful. The general techniques that we describe below will make them obvious and show how to systematically avoid them. For now, just find them and suggest remedies.

This `high()` function is limited in that it is a solution to a single specific problem:

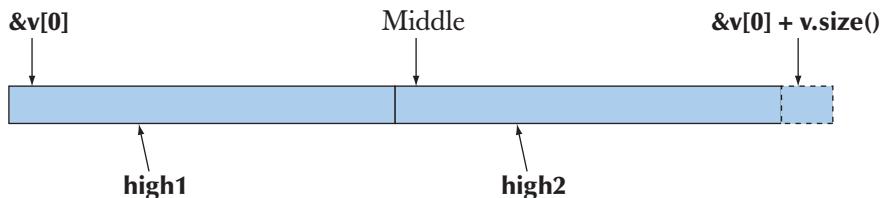
- It works for arrays only. We rely on the elements of a `vector` being stored in an array, but there are many more ways of storing data, such as `lists` and `maps` (see §20.4 and §21.6.1).
- It can be used for `vectors` and arrays of `doubles`, but not for arrays or `vectors` with other element types, such as `vector<double*>` and `char[10]`.
- It finds the element with the highest value, but there are many more simple calculations that we want to do on such data.

Let's explore how we can support this kind of calculation on sets of data in far greater generality.

Please note that by deciding to express our “find highest element” algorithm in terms of pointers, we “accidentally” generalized it to do more than we required: we can – as desired – find the highest element of an array or a **vector**, but we can also find the highest element in part of an array or in part of a **vector**. For example:

```
// ...
vector<double>& v = *jill_data;
double* middle = &v[0]+v.size()/2;
double* high1 = high(&v[0], middle);           // max of first half
double* high2 = high(middle, &v[0]+v.size());   // max of second half
// ...
```

Here **high1** will point to the element with the largest value in the first half of the vector and **high2** will point to the element with the largest value in the second half. Graphically, it will look something like this:



We used pointer arguments for **high()**. That's a bit low-level and can be error-prone. We suspect that for many programmers, the obvious function for finding the element with the largest value in a **vector** would look like this:

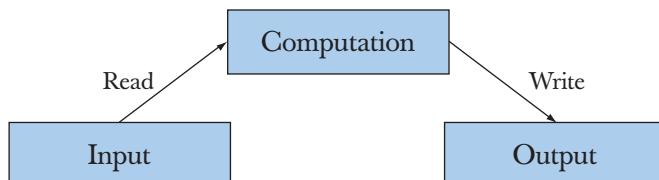
```
double* find_highest(vector<double>& v)
{
    double h = -1;
    double* high = 0;
    for (int i=0; i<v.size(); ++i)
        if (h<v[i]) { high = &v[i]; h = v[i]; }
    return high;
}
```

However, that wouldn't give us the flexibility we “accidentally” obtained from **high()** – we can't use **find_highest()** to find the element with the highest value in part of a **vector**. We actually achieved a practical benefit from writing a function that could be used for both arrays and **vectors** by “messing with pointers.” We will remember that: generalization can lead to functions that are useful for more problems.

20.2 STL ideals

The C++ standard library provides a framework for dealing with data as sequences of elements, called the STL. STL is usually said to be an acronym for “standard template library.” The STL is the part of the ISO C++ standard library that provides containers (such as `vector`, `list`, and `map`) and generic algorithms (such as `sort`, `find`, and `accumulate`). Thus we can – and do – refer to facilities, such as `vector`, as being part of both “the STL” and “the standard library.” Other standard library features, such as `ostream` (Chapter 10) and C-style string functions (§B.11.3), are not part of the STL. To better appreciate and understand the STL, we will first consider the problems we must address when dealing with data and the ideals we have for a solution.

There are two major aspects of computing: the computation and the data. Sometimes we focus on the computation and talk about `if`-statements, loops, functions, error handling, etc. At other times, we focus on the data and talk about arrays, vectors, strings, files, etc. However, to get useful work done we need both. A large amount of data is incomprehensible without analysis, visualization, and searching for “the interesting bits.” Conversely, we can compute as much as we like, but it’s going to be tedious and sterile unless we have some data to tie our computation to something real. Furthermore, the “computation part” of our program has to elegantly interact with the “data part.”



When we talk about data in this way, we think of lots of data: dozens of `Shapes`, hundreds of temperature readings, thousands of log records, millions of points, billions of web pages, etc.; that is, we talk about processing containers of data, streams of data, etc. In particular, this is not a discussion of how best to choose a couple of values to represent a small object, such as a complex number, a temperature reading, or a circle. For such types, see Chapters 9, 11, and 14.

Consider some simple examples of something we’d like to do with “a lot of data”:

- Sort the words in dictionary order.
- Find a number in a phone book, given a name.
- Find the highest temperature.
- Find all values larger than 8800.

- Find the first occurrence of the value 17.
- Sort the telemetry records by unit number.
- Sort the telemetry records by time stamp.
- Find the first value larger than “Petersen.”
- Find the largest amount.
- Find the first difference between two sequences.
- Compute the pair-wise product of the elements of two sequences.
- Find the highest temperature for each day in a month.
- Find the top ten best sellers in the sales records.
- Count the number of occurrences of “Stroustrup” on the web.
- Compute the sum of the elements.

Note that we can describe each of these tasks without actually mentioning how the data is stored. Clearly, we must be dealing with something like lists, vectors, files, input streams, etc. for these tasks to make sense, but we don’t have to know the details about how the data is stored (or gathered) to talk about what to do with it. What is important is the type of the values or objects (the element type), how we access those values or objects, and what we want to do with them.

These kinds of tasks are very common. Naturally, we want to write code performing such tasks simply and efficiently. Conversely, the problems for us as programmers are:

- There is an infinite variation of data types (“kinds of data”).
- There is a bewildering number of ways to store collections of data elements.
- There is a huge variety of tasks we’d like to do with collections of data.

To minimize the effect of these problems, we’d like our code to take advantage of commonalities among types, among the ways of storing data, and among our processing tasks. In other words, we want to generalize our code to cope with these kinds of variations. We really don’t want to hand-craft each solution from scratch; that would be a tedious waste of time.

To get an idea of what support we would like for writing our code, consider a more abstract view of what we do with data:

- Collect data into containers
 - Such as **vector**, **list**, and array
- Organize data
 - For printing
 - For fast access

- Retrieve data items
 - By index (e.g., the 42nd element)
 - By value (e.g., the first record with the “age field” 7)
 - By properties (e.g., all records with the “temperature field” >32 and <100)
- Modify a container
 - Add data
 - Remove data
 - Sort (according to some criteria)
- Perform simple numeric operations (e.g., multiply all elements by 1.7)

We'd like to do these things without getting sucked into a swamp of details about differences among containers, differences in ways of accessing elements, and differences among element types. If we can do that, we'll have come a long way toward our goal of simple and efficient use of large amounts of data.

Looking back at the programming tools and techniques from the previous chapters, we note that we can (already) write programs that are similar independently of the data type used:

- Using an `int` isn't all that different from using a `double`.
- Using a `vector<int>` isn't all that different from using a `vector<string>`.
- Using an array of `double` isn't all that different from using a `vector<double>`.

We'd like to organize our code so that we have to write new code only when we want to do something really new and different. In particular, we'd like to provide code for common programming tasks so that we don't have to rewrite our solution each time we find a new way of storing the data or find a slightly different way of interpreting the data.

- Finding a value in a `vector` isn't all that different from finding a value in an array.
- Looking for a `string` ignoring case isn't all that different from looking at a `string` considering uppercase letters different from lowercase ones.
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values.
- Copying a file isn't all that different from copying a `vector`.

We want to build on these observations to write code that's

- Easy to read
- Easy to modify

- Regular
- Short
- Fast

To minimize our programming work, we would like

- Uniform access to data
 - Independently of how it is stored
 - Independently of its type
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
 - Retrieval of data
 - Addition of data
 - Deletion of data
- Standard versions of the most common algorithms
 - Such as copy, find, search, sort, sum, . . .

The STL provides that, and more. We will look at it not just as a very useful set of facilities, but also as an example of a library designed for maximal flexibility and performance. The STL was designed by Alex Stepanov to provide a framework for general, correct, and efficient algorithms operating on data structures. The ideal was the simplicity, generality, and elegance of mathematics.

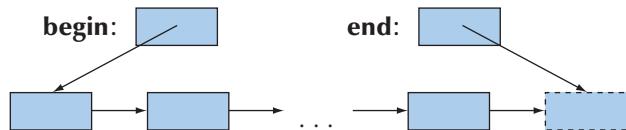
The alternative to dealing with data using a framework with clearly articulated ideals and principles is for each programmer to craft each program out of the basic language facilities using whatever ideas seem good at the time. That's a lot of extra work. Furthermore, the result is often an unprincipled mess; rarely is the result a program that is easily understood by people other than its original designer, and only by chance is the result code that we can use in other contexts.

Having considered the motivation and the ideals, let's look at the basic definitions of the STL, and then finally get to the examples that'll show us how to approximate those ideals – to write better code for dealing with data and to do so with greater ease.

20.3 Sequences and iterators

The central concept of the STL is the sequence. From the STL point of view, a collection of data is a sequence. A sequence has a beginning and an end. We can traverse a sequence from its beginning to its end, optionally reading or writing the

value of each element. We identify the beginning and the end of a sequence by a pair of iterators. An *iterator* is an object that identifies an element of a sequence. We can think of a sequence like this:



Here, **begin** and **end** are iterators; they identify the beginning and the end of the sequence. An STL sequence is what is usually called “half-open”; that is, the element identified by **begin** is part of the sequence, but the **end** iterator points one beyond the end of the sequence. The usual mathematical notation for such sequences (ranges) is **[begin:end)**. The arrows from one element to the next indicate that if we have an iterator to one element we can get an iterator to the next.

What is an iterator? An iterator is a rather abstract notion:

- An iterator points to (refers to) an element of a sequence (or one beyond the last element).
- You can compare two iterators using **==** and **!=**.
- You can refer to the value of the element pointed to by an iterator using the unary ***** operator (“dereference” or “contents of”).
- You can get an iterator to the next element by using **++**.

For example, if **p** and **q** are iterators to elements of the same sequence:

Basic standard iterator operations

p==q	true if and only if p and q point to the same element or both point to one beyond the last element
p!=q	!(p==q)
*p	refers to the element pointed to by p
*p=val	writes to the element pointed to by p
val=*p	reads from the element pointed to by p
++p	makes p refer to the next element in the sequence or to one beyond the last element

Clearly, the idea of an iterator is related to the idea of a pointer (§17.4). In fact, a pointer to an element of an array is an iterator. However, many iterators are not just pointers; for example, we could define a range-checked iterator that throws an exception if you try to make it point outside its **[begin:end)** sequence or

dereference `end`. It turns out that we get enormous flexibility and generality from having iterator as an abstract notion rather than as a specific type. This chapter and the next will give several examples.

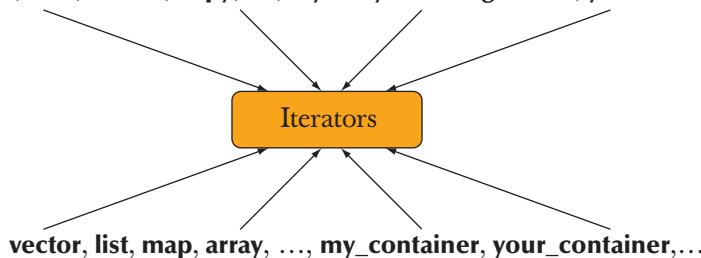
TRY THIS



Write a function `void copy(int* f1, int* e1, int* f2)` that copies the elements of an array of `ints` defined by `[f1:e1]` into another `[f2:f2+(e1-f1)]`. Use only the iterator operations mentioned above (not subscripting).

Iterators are used to connect our code (algorithms) to our data. The writer of the code knows about the iterators (and not about the details of how the iterators actually get to the data), and the data provider supplies iterators rather than exposing details about how the data is stored to all users. The result is pleasingly simple and offers an important degree of independence between algorithms and containers. To quote Alex Stepanov: “The reason STL algorithms and containers work so well together is that they don’t know anything about each other.” Instead, both understand about sequences defined by pairs of iterators.

`sort, find, search, copy, ..., my_very_own_algorithm, your_code, ...`



In other words, my algorithms no longer have to know about the bewildering variety of ways of storing and accessing data; they just have to know about iterators. Conversely, if I’m a data provider, I no longer have to write code to serve a bewildering variety of users; I just have to implement an iterator for my data. At the most basic level, an iterator is defined by just the `*`, `++`, `==`, and `!=` operators. That makes them simple and fast.

The STL framework consists of about ten containers and about 60 algorithms connected by iterators (see Chapter 21). In addition, many organizations and individuals provide containers and algorithms in the style of the STL. The STL is probably the currently best-known and most widely used example of generic programming (§19.3.2). If you know the basic concepts and a few examples, you can use the rest.

20.3.1 Back to the example

Let's see how we can express the "find the element with the largest value" problem using the STL notion of a sequence:

```
template<typename Iterator>
Iterator high(Iterator first, Iterator last)
    // return an iterator to the element in [first:last) that has the highest value
{
    Iterator high = first;
    for (Iterator p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
```

Note that we eliminated the local variable **h** that we had used to hold the highest value seen so far. When we don't know the name of the actual type of the elements of the sequence, the initialization by **-1** seems completely arbitrary and odd. That's because it was arbitrary and odd! It was also an error waiting to happen: in our example **-1** worked only because we happened not to have any negative velocities. We knew that "magic constants," such as **-1**, are bad for code maintenance (§4.3.1, §7.6.1, §10.11.1, etc.). Here, we see that they can also limit the utility of a function and can be a sign of incomplete thought about the solution; that is, "magic constants" can be – and often are – a sign of sloppy thinking.

Note that this "generic" **high()** can be used for any element type that can be compared using **<**. For example, we could use **high()** to find the lexicographically last string in a **vector<string>** (see exercise 7).

The **high()** template function can be used for any sequence defined by a pair of iterators. For example, we can exactly replicate our example program:

```
double* get_from_jack(int* count);    // Jack puts doubles into an array and
                                         // returns the number of elements in *count
vector<double>* get_from_jill();      // Jill fills the vector

void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();

    double* jack_high = high(jack_data,jack_data+jack_count);
    vector<double>& v = *jill_data;
    double* jill_high = high(&v[0],&v[0]+v.size());
```

```

cout << "Jill's high " << *jill_high << "; Jack's high " << *jack_high;
// ...
delete[] jack_data;
delete jill_data;
}

```

For the two calls here, the **Iterator** template argument type for **high()** is **double***. Apart from (finally) getting the code for **high()** correct, there is apparently no difference from our previous solution. To be precise, there is no difference in the code that is executed, but there is a most important difference in the generality of our code. The templated version of **high()** can be used for every kind of sequence that can be described by a pair of iterators. Before looking at the detailed conventions of the STL and the useful standard algorithms that it provides to save us from writing common tricky code, let's consider a couple of more ways of storing collections of data elements.

TRY THIS

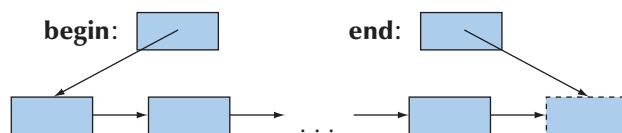


We again left a serious error in that program. Find it, fix it, and suggest a general remedy for that kind of problem.

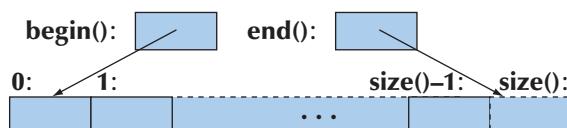
20.4 Linked lists



Consider again the graphical representation of the notion of a sequence:



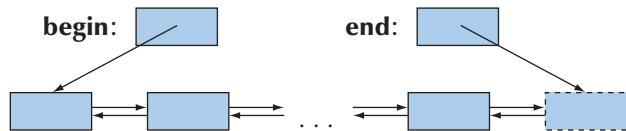
Compare it to the way we visualize a **vector** in memory:



Basically, the subscript **0** identifies the same element as does the iterator **v.begin()**, and the subscript **v.size()** identifies the one-beyond-the-last element also identified by the iterator **v.end()**.

The elements of the **vector** are consecutive in memory. That's not required by STL's notion of a sequence, and it so happens that there are many algorithms where we would like to insert an element in between two existing elements without moving those existing elements. The graphical representation of the abstract notion suggests the possibility of inserting elements (and of deleting elements) without moving other elements. The STL notion of iterators supports that.

The data structure most directly suggested by the STL sequence diagram is called a *linked list*. The arrows in the abstract model are usually implemented as pointers. An element of a linked list is part of a “link” consisting of the element and one or more pointers. A linked list where a link has just one pointer (to the next link) is called a *singly-linked list* and a list where a link has pointers to both the previous and the next link is called a *doubly-linked list*. We will sketch the implementation of a doubly-linked list, which is what the C++ standard library provides under the name of **list**. Graphically, it can be represented like this:



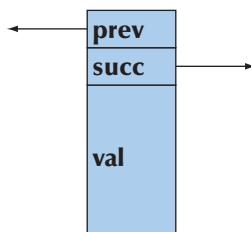
This can be represented in code as

```

template<typename Elem>
struct Link {
    Link* prev;           // previous link
    Link* succ;          // successor (next) link
    Elem val;            // the value
};

template<typename Elem> struct list {
    Link<Elem>* first;
    Link<Elem>* last;   // one beyond the last link
};
  
```

The layout of a **Link** is



There are many ways of implementing linked lists and presenting them to users. A description of the standard library version can be found in Appendix B. Here, we'll just outline the key properties of a list – you can insert and delete elements without disturbing existing elements – show how we can iterate over a list, and give an example of list use.

When you try to think about lists, we strongly encourage you to draw little diagrams to visualize the operations you are considering. Linked-list manipulation really is a topic where a picture is worth 1K words.

20.4.1 List operations

What operations do we need for a list?

- The operations we have for `vector` (constructors, size, etc.), except subscripting
- Insert (add an element) and erase (remove an element)
- Something that can be used to refer to elements and to traverse the list: an iterator

In the STL, that iterator type is a member of its class, so we'll do the same:

```
template<typename Elem>
class list {
    // representation and implementation details
public:
    class iterator;           // member type: iterator

    iterator begin();         // iterator to first element
    iterator end();           // iterator to one beyond last element

    iterator insert(iterator p, const Elem& v); // insert v into list after p
    iterator erase(iterator p);                  // remove p from the list

    void push_back(const Elem& v);             // insert v at end
    void push_front(const Elem& v);            // insert v at front
    void pop_front();                          // remove the first element
    void pop_back();                           // remove the last element

    Elem& front();                            // the first element
    Elem& back();                             // the last element

    // ...
};
```

Just as “our” `vector` is not the complete standard library `vector`, this `list` is not the complete definition of the standard library `list`. There is nothing wrong with this `list`; it simply isn’t complete. The purpose of “our” `list` is to convey an understanding of what linked lists are, how a `list` might be implemented, and how to use the key features. For more information see Appendix B or an expert-level C++ book.

The iterator is central to the definition of an STL `list`. Iterators are used to identify places for insertion and elements for removal (erasure). They are also used for “navigating” through a list rather than using subscripting. This use of iterators is very similar to the way we used pointers to traverse arrays and vectors in §20.1 and §20.3.1. This style of iterators is the key to the standard library algorithms (§21.1–3).

Why not subscripting for `list`? We could subscript a list, but it would be a surprisingly slow operation: `lst[1000]` would involve starting from the first element and then visiting each link along the way until we reached element number **1000**. If we want to do that, we can do it ourselves (or use `advance()`; see §20.6.2). Consequently, the standard library `list` doesn’t provide the innocuous-looking subscript syntax.

We made `list`’s iterator type a member (a nested class) because there was no reason for it to be global. It is used only with `lists`. Also, this allows us to name every container’s iterator type `iterator`. In the standard library, we have `list<T>::iterator`, `vector<T>::iterator`, `map<K,V>::iterator`, and so on.

20.4.2 Iteration

The `list` iterator must provide `*`, `++`, `==`, and `!=`. Since the standard library `list` is a doubly-linked list, it also provides `--` for iterating “backward” toward the front of the list:

```
template<typename Elem>      // requires Element<Elem>() (§19.3.3)
class list<Elem>::iterator {
    Link<Elem>* curr;        // current link
public:
    iterator(Link<Elem>* p) : curr{p} {}

    iterator& operator++() {curr = curr->succ; return *this;} // forward
    iterator& operator--() { curr = curr->prev; return *this;} // backward
    Elem& operator*() { return curr->val;} // get value (dereference)

    bool operator==(const iterator& b) const { return curr==b.curr; }
    bool operator!=(const iterator& b) const { return curr!=b.curr; }
};
```

These functions are short and simple, and obviously efficient: there are no loops, no complicated expressions, and no “suspicious” function calls. If the implementation isn’t clear to you, just have a quick look at the diagrams above. This **list** iterator is just a pointer to a link with the required operations. Note that even though the implementation (the code) for a **list<Elem>::iterator** is very different from the simple pointer we have used as an iterator for **vectors** and arrays, the meaning (the semantics) of the operations is identical. Basically, the **list** iterator provides suitable **++**, **--**, *****, **==**, and **!=** for a **Link** pointer.

Now look at **high()** again:

```
template<typename Iter> // requires Input_iterator<Iter>() (§19.3.3)
Iterator high(Iter first, Iter last)
    // return an iterator to the element in [first, last) that has the highest value
{
    Iterator high = first;
    for (Iterator p = first; p != last; ++p)
        if (*high < *p) high = p;
    return high;
}
```

We can use it for a **list**:

```
void f()
{
    list<int> lst; for (int x; cin >> x; ) lst.push_front(x);

    list<int>::iterator p = high(lst.begin(), lst.end());
    cout << "the highest value was " << *p << '\n';
}
```

Here, the “value” of the **Iterator** argument is **list<int>::iterator**, and the implementation of **++**, *****, and **!=** has changed dramatically from the array case, but the meaning is still the same. The template function **high()** still traverses the data (here a **list**) and finds the highest value. We can insert an element anywhere in a **list**, so we used **push_front()** to add elements at the front just to show that we could. We could equally well have used **push_back()** as we do for **vectors**.

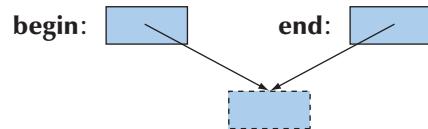
TRY THIS



The standard library **vector** doesn’t provide **push_front()**. Why not? Implement **push_front()** for **vector** and compare it to **push_back()**.

Now, finally, is the time to ask, “But what if the `list` is empty?” In other words, “What if `lst.begin() == lst.end()`?” In that case, `*p` will be an attempt to dereference the one-beyond-the-last element, `lst.end()`: disaster! Or – potentially worse – the result could be a random value that might be mistaken for a correct answer.

The last formulation of the question strongly hints at the solution: we can test whether a list is empty by comparing `begin()` and `end()` – in fact, we can test whether any STL sequence is empty by comparing its beginning and end:



That’s the deeper reason for having `end` point one beyond the last element rather than at the last element: the empty sequence is not a special case. We dislike special cases because – by definition – we have to remember to write special-case code for them.

In our example, we could use that like this:

```

list<int>::iterator p = high(lst.begin(), lst.end());
if (p==lst.end())           // did we reach the end?
    cout << "The list is empty";
else
    cout << "the highest value is " << *p << '\n';
  
```

We use testing the return value against `end()` – indicating “not found” – systematically with STL algorithms.

Because the standard library provides a list, we won’t go further into the implementation here. Instead, we’ll have a brief look at what lists are good for (see exercises 12–14 if you are interested in list implementation details).

20.5 Generalizing vector yet again

Obviously, from the examples in §20.3–4, the standard library `vector` has an `iterator` member type and `begin()` and `end()` member functions (just like `std::list`). However, we did not provide those for our `vector` in Chapter 19. What does it really take for different containers to be used more or less interchangeably in the STL generic programming style presented in §20.3? First, we’ll outline the solution (ignoring allocators to simplify) and then explain it:

```

template<typename T> // requires Element<T>() (§19.3.3)
class vector {
  
```

```

public:
    using size_type = unsigned long;
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;

    // ...

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    size_type size();

    // ...
};


```



A **using** declaration creates an alias for a type; that is, for our **vector**, **iterator** is a synonym, another name, for the type we chose to use as our iterator: **T***. Now, for a **vector** called **v**, we can write

```
vector<int>::iterator p = find(v.begin(), v.end(), 32);
```

and

```
for (vector<int>::size_type i = 0; i < v.size(); ++i) cout << v[i] << '\n';
```

The point is that to write that, we don't actually have to know what types are named by **iterator** and **size_type**. In particular, the code above, because it is expressed in terms of **iterator** and **size_type**, will work with **vectors** where **size_type** is not an **unsigned long** (as it is not on many embedded systems processors) and where **iterator** is not a plain pointer, but a class (as it is on many popular C++ implementations).

The standard defines **list** and the other standard containers similarly. For example:

```

template<typename T>           // requires Element<T>() (§19.3.3)
class list {
public:
    class Link;
    using size_type = unsigned long;

```

```

using value_type = T;
class iterator;           // see §20.4.2
class const_iterator;    // like iterator, but not allowing writes to elements

// ...

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

size_type size();

// ...
}

```

That way, we can write code that does not care whether it uses a **list** or a **vector**. All the standard library algorithms are defined in terms of these member type names, such as **iterator** and **size_type**, so that they don't unnecessarily depend on the implementations of containers or exactly which kind of container they operate on (see Chapter 21).

As an alternative to saying **C::iterator** for some container **C**, we often prefer **Iterator<C>**. This can be achieved through a simple template alias:

```

template<typename C>
using Iterator = typename C::iterator;      // Iterator<C> means typename
                                                // C::iterator

```

The fact that for language-technical reasons we need to prefix **C::iterator** with **typename** to say that **iterator** is a type is part of the reason we prefer **Iterator<C>**. Similarly, we define

```

template<typename C>
using Value_type = typename C::value_type;

```

That way, we can write **Value_type<C>**. These type aliases are not in the standard library, but you can find them in **std_lib_facilities.h**.

A **using** declaration is a C++11 notation for and a generalization of what was known in C and C++ as a **typedef** (§A.16).

20.5.1 Container traversal

Using `size()`, we can traverse one of our `vectors` from its first element to its last. For example:

```
void print1(const vector<double>& v)
{
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << '\n';
}
```

This doesn't work for lists because `list` does not provide subscripting. However, we can traverse a standard library `vector` and `list` using the simpler range-`for`-loop (§4.6.1). For example:

```
void print2(const vector<double>& v, const list<double>& lst)
{
    for (double x : v)
        cout << x << '\n';

    for (double x : lst)
        cout << x << '\n';
}
```



This works for both the standard library containers and for "our" `vector` and `list`. How? The "trick" is that the range-`for`-loop is defined in terms of `begin()` and `end()` functions returning iterators to the first and one beyond the end of our `vector` elements. The range-`for`-loop is simply "syntactic sugar" for a loop over a sequence using iterators. When we defined `begin()` and `end()` for our `vector` and `list` we "accidentally" provided what the range-`for` needed.

20.5.2 auto

When we have to write out loops over a generic structure, naming the iterators can be a real nuisance. Consider:

```
template<typename T>      // requires Element<T>()
void user(vector<T>& v, list<T>& lst)
{
    for (vector<T>::iterator p = v.begin(); p != v.end(); ++p) cout << *p << '\n';

    list<T>::iterator q = find(lst.begin(), lst.end(), T{42});
}
```

The most annoying aspect of this is that the compiler obviously already knows the **iterator** type for the **list** and the **size_type** for the **vector**. Why should we have to tell the compiler what it already knows? Doing so just annoys the poor typists among us and opens opportunities for mistakes. Fortunately, we don't have to: we can declare a variable **auto**, meaning use the type of the **iterator** as the type of the variable:

```
template<typename T>      // requires Element<T>()
void user(vector<T>& v, list<T>& lst)
{
    for (auto p = v.begin(); p!=v.end(); ++p) cout << *p << '\n';

    auto q = find(lst.begin(), lst.end(), T{42});
}
```

Here, **p** is a **vector<T>::iterator** and **q** is a **list<T>::iterator**. We can use **auto** in just about every definition that includes an initializer. For example:

```
auto x = 123;  // x is an int
auto c = 'y';  // c is a char
auto& r = x;  // r is an int&
auto y = r;    // y is an int (references are implicitly dereferenced)
```

Note that a string literal has the type **const char***, so using **auto** for string literals might lead to an unpleasant surprise:

```
auto s1 = "San Antonio";      // s1 is a const char* (Surprise!?)
string s2 = "Fredericksburg"; // s2 is a string
```

When we know exactly which type we want, we can often say so as easily as we can use **auto**.

One common use of **auto** is to specify the loop variable in a range-**for**-loop. Consider:

```
template<typename C>      // requires Container<T>
void print3(const C& cont)
{
    for (const auto& x : cont)
        cout << x << '\n';
}
```

Here, we use `auto` because it is not all that easy to name the element type of the container `cont`. We use `const` because we are not writing to the container elements, and we use `&` (for reference) in case the elements are so large that copying them would be costly.

20.6 An example: a simple text editor

The essential feature of a list is that you can add and remove elements without moving other elements of the list. Let's try a simple example that illustrates that. Consider how to represent the characters of a text document in a simple text editor. The representation should make operations on the document simple and reasonably efficient.

Which operations? Let's assume that a document will fit in your computer's main memory. That way, we can choose any representation that suits us and simply convert it to a stream of bytes when we want to store it in a file. Similarly, we can read a stream of bytes from a file and convert those to our in-memory representation. That decided, we can concentrate on choosing a convenient in-memory representation. Basically, there are five things that our representation must support well:

- Constructing it from a stream of bytes from input
- Inserting one or more characters
- Deleting one or more characters
- Searching for a string
- Generating a stream of bytes for output to a file or a screen

The simplest representation would be a `vector<char>`. However, to add or delete a character we would have to move every following character in the document. Consider:

This is the start of a very long document.

There are lots of . . .

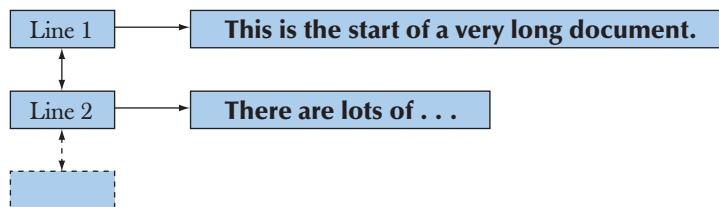
We could add the `t` needed to get

This is the start of a very long document.

There are lots of . . .

However, if those characters were stored in a single `vector<char>`, we'd have to move every character from `h` onward one position to the right. That could be a

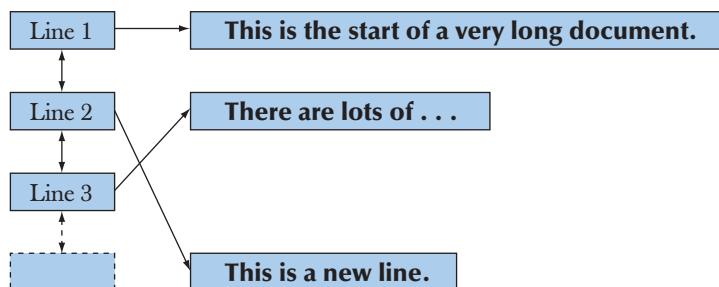
lot of copying. In fact for a 70,000-character-long document (such as this chapter, counting spaces), we would, on average, have to move 35,000 characters to insert or delete a character. The resulting real-time delay is likely to be noticeable and annoying to users. Consequently, we “break down” our representation into “chunks” so that we can change part of the document without moving a lot of characters around. We represent a document as a list of “lines,” `list<Line>`, where a `Line` is a `vector<char>`. For example:



Now, when we inserted that `t`, we only had to move the rest of the characters on that line. Furthermore, when we need to, we can add a new line without moving any characters. For example, we could insert `This is a new line.` after `document`. to get

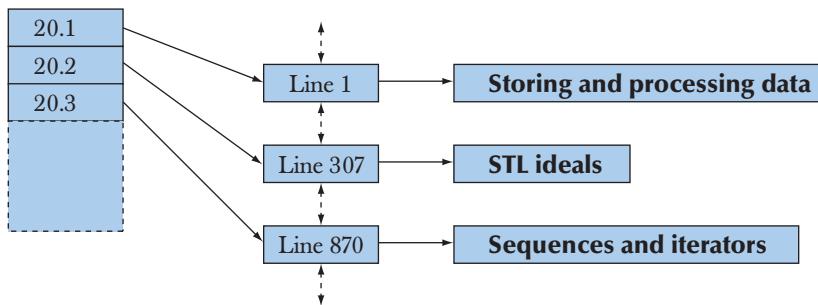
`This is the start of a very long document.`
`This is a new line.`
`There are lots of ...`

All we needed to do was to insert a new “line” in the middle:



The logical reason that it is important to be able to insert new links in a list without moving existing links is that we might have iterators pointing to those links or pointers (and references) pointing to the objects in those links. Such iterators and pointers are unaffected by insertions or deletions of lines. For example, a word

processor may keep a `vector<list<Line>::iterator>` holding iterators to the beginning of every title and subtitle in the current **Document**:



We can add lines to “paragraph 20.2” without invalidating the iterator to “paragraph 20.3.”

In conclusion, we use a **list** of lines rather than a **vector** of lines or a **vector** of all the characters for both logical and performance reasons. Please note that situations where these reasons apply are rather rare so that the “by default, use **vector**” rule of thumb still holds. You need a specific reason to prefer a **list** over a **vector** – even if you think of your data as a list of elements! (See §20.7.) A list is a logical concept that you can represent in your program as a (linked) **list** or as a **vector**. The closest STL analog to our everyday concept of a list (e.g., a to-do list, a list of groceries, or a schedule) is a sequence, and most sequences are best represented as **vectors**.

20.6.1 Lines

How do we decide what’s a “line” in our document? There are three obvious choices:

1. Rely on newline indicators (e.g., `\n`) in user input.
2. Somehow parse the document and use some “natural” punctuation (e.g., `.`).
3. Split any line that grows beyond a given length (e.g., 50 characters) into two.

There are undoubtedly also some less obvious choices. For simplicity, we use alternative 1 here.

We will represent a document in our editor as an object of class **Document**. Stripped of all refinements, our document type looks like this:

```

using Line = vector<char>;           // a line is a vector of characters

struct Document {
    list<Line> line;                  // a document is a list of lines
    Document() { line.push_back(Line{}); }
};

```

Every **Document** starts out with a single empty line: **Document**'s constructor makes an empty line and pushes it into the list of lines.

Reading and splitting into lines can be done like this:

```
istream& operator>>(istream& is, Document& d)
{
    for (char ch; is.get(ch); ) {
        d.line.back().push_back(ch); // add the character
        if (ch=='\n')
            d.line.push_back(Line{}); // add another line
    }
    if (d.line.back().size() d.line.push_back(Line{}); // add final empty line
    return is;
}
```

Both **vector** and **list** have a member function **back()** that returns a reference to the last element. To use it, you have to be sure that there really is a last element for **back()** to refer to – don't use it on an empty container. That's why we defined a **Document** to end with an empty **Line**. Note that we store every character from input, even the newline characters ('\n'). Storing those newline characters greatly simplifies output, but you have to be careful how you define a character count (just counting characters will give a number that includes space and newline characters).

20.6.2 Iteration

If the document was just a **vector<char>** it would be simple to iterate over it. How do we iterate over a list of lines? Obviously, we can iterate over the list using **list<Line>::iterator**. However, what if we wanted to visit the characters one after another without any fuss about line breaks? We could provide an iterator specifically designed for our **Document**:

```
class Text_iterator { // keep track of line and character position within a line
    list<Line>::iterator ln;
    Line::iterator pos;
public:
    // start the iterator at line ll's character position pp:
    Text_iterator(list<Line>::iterator ll, Line::iterator pp)
        :ln{ll}, pos{pp} {}

    char& operator*() { return *pos; }
    Text_iterator& operator++();
```

```

bool operator==(const Text_iterator& other) const
    { return ln==other.ln && pos==other.pos; }
bool operator!=(const Text_iterator& other) const
    { return !(*this==other); }

};

Text_iterator& Text_iterator::operator++()
{
    ++pos;                                // proceed to next character
    if (pos==(*ln).end()) {
        ++ln;                                // proceed to next line
        pos = (*ln).begin();      // bad if ln==line.end(); so make sure it isn't
    }
    return *this;
}

```

To make **Text_iterator** useful, we need to equip class **Document** with conventional **begin()** and **end()** functions:

```

struct Document {
    list<Line> line;

    Text_iterator begin()           // first character of first line
        { return Text_iterator(line.begin(), (*line.begin()).begin()); }
    Text_iterator end()            // one beyond the last character of the last line
    {
        auto last = line.end();
        --last;      // we know that the document is not empty
        return Text_iterator(last, (*last).end());
    }
};

```

We need the curious **(*line.begin()).begin()** notation because we want the beginning of what **line.begin()** points to; we could alternatively have used **line.begin() -> begin()** because the standard library iterators support **->**.

We can now iterate over the characters of a document like this:

```

void print(Document& d)
{
    for (auto p : d) cout << *p;
}

print(my_doc);

```

Presenting the document as a sequence of characters is useful for many things, but usually we traverse a document looking for something more specific than a character. For example, here is a piece of code to delete line `n`:

```
void erase_line(Document& d, int n)
{
    if (n<0 || d.line.size()-1<=n) return;
    auto p = d.line.begin();
    advance(p,n);
    d.line.erase(p);
}
```

A call `advance(p,n)` moves an iterator `p` `n` elements forward; `advance()` is a standard library function, but we could have implemented it ourselves like this:

```
template<typename Iter> // requires Forward_iterator<Iter>
void advance(Iter& p, int n)
{
    while (0<n) { ++p; --n; }
}
```

Note that `advance()` can be used to simulate subscripting. In fact, for a `vector` called `v`, `p=v.begin; advance(p,n); *p=x` is roughly equivalent to `v[n]=x`. Note that “roughly” means that `advance()` laboriously moves past the first `n-1` elements one by one, whereas the subscript goes straight to the `nth` element. For a `list`, we have to use the laborious method. It’s a price we have to pay for the more flexible layout of the elements of a `list`.

For an iterator that can move both forward and backward, such as the iterator for `list`, a negative argument to the standard library `advance()` will move the iterator backward. For an iterator that can handle subscripting, such as the iterator for a `vector`, the standard library `advance()` will go directly to the right element rather than slowly moving along using `++`. Clearly, the standard library `advance()` is a bit smarter than ours. That’s worth noticing: typically, the standard library facilities have had more care and time spent on them than we could afford, so prefer the standard facilities to “home brew.”



TRY THIS



Rewrite `advance()` so that it will “go backward” when you give it a negative argument.

Probably, a search is the kind of iteration that is most obvious to a user. We search for individual words (such as **milkshake** or **Gavin**), for sequences of letters that can't easily be considered words (such as **secret\nhomestead** – i.e., a line ending with **secret** followed by a line starting with **homestead**), for regular expressions (e.g., **[bB]w*w*ne** – i.e., an upper- or lowercase **B** followed by 0 or more letters followed by **ne**; see Chapter 23), etc. Let's show how to handle the second case, finding a string, using our **Document** layout. We use a simple – non-optimal – algorithm:

- Find the first character of our search string in the document.
- See if that character and the following characters match our search string.
- If so, we are finished; if not, we look for the next occurrence of that first character.

For generality, we adopt the STL convention of defining the text in which to search as a sequence defined by a pair of iterators. That way we can use our search function for any part of a document as well as a complete document. If we find an occurrence of our string in the document, we return an iterator to its first character; if we don't find an occurrence, we return an iterator to the end of the sequence:

```
Text_iterator find_txt(Text_iterator first, Text_iterator last, const string& s)
{
    if (s.size()==0) return last; // can't find an empty string
    char first_char = s[0];
    while (true) {
        auto p = find(first, last, first_char);
        if (p==last || match(p, last, s)) return p;
        first = ++p; // look at the next character
    }
}
```

Returning the end of the sequence to indicate “not found” is an important STL convention. The **match()** function is trivial; it just compares two sequences of characters. Try writing it yourself. The **find()** used to look for a character in the sequence of characters is arguably the simplest standard library algorithm (§21.2). We can use our **find_txt()** like this:

```
auto p = find_txt(my_doc.begin(), my_doc.end(), "secret\nhomestead");
if (p==my_doc.end())
    cout << "not found";
else {
    // do something
}
```

Our “text processor” and its operations are very simple. Obviously, we are aiming for simplicity and reasonable efficiency, rather than at providing a “feature-rich” editor. Don’t be fooled into thinking that providing *efficient* insertion, deletion, and search for arbitrary character sequences is trivial, though. We chose this example to illustrate the power and generality of the STL concepts sequence, iterator, and container (such as **list** and **vector**) together with some STL programming conventions (techniques), such as returning the end of a sequence to indicate failure. Note that if we wanted to, we could develop **Document** into an STL container – by providing **Text_iterator** we have done the key part of representing a **Document** as a sequence of values.

20.7 vector, list, and string

Why did we use a **list** for the lines and a **vector** for the characters? More precisely, why did we use a **list** for the sequence of lines and a **vector** for the sequence of characters? Furthermore, why didn’t we use a **string** to hold a line?

We can ask a slightly more general variant of this question. We have now seen four ways to store a sequence of characters:

- **char[]** (array of characters)
- **vector<char>**
- **string**
- **list<char>**

How do we choose among them for a given problem? For really simple tasks, they are interchangeable; that is, they have very similar interfaces. For example, given an iterator, we can walk through each using **++** and use ***** to access the characters. If we look at the code examples related to **Document**, we can actually replace our **vector<char>** with **list<char>** or **string** without any logical problems. Such interchangeability is fundamentally good because it allows us to choose based on performance. However, before we consider performance, we should look at logical properties of these types: what can each do that the others can’t?

- **Elem[]**: Doesn’t know its own size. Doesn’t have **begin()**, **end()**, or any of the other useful container member functions. Can’t be systematically range checked. Can be passed to functions written in C and C-style functions. The elements are allocated contiguously in memory. The size of the array is fixed at compile time. Comparison (**==** and **!=**) and output (**<<**) use the pointer to the first element of the array, not the elements.
- **vector<Elem>**: Can do just about everything, including **insert()** and **erase()**. Provides subscripting. List operations, such as **insert()** and **erase()**, typically involve moving elements (that can be inefficient for large elements and large

numbers of elements). Can be range checked. The elements are allocated contiguously in memory. A **vector** is expandable (e.g., use **push_back()**). Elements of a vector are stored (contiguously) in an array. Comparison operators (**==**, **!=**, **<**, **<=**, **>**, and **>=**) compare elements.

- **string**: Provides all the common and useful operations plus specific text manipulation operations, such as concatenation (**+** and **+=**). The elements are guaranteed to be contiguous in memory. A **string** is expandable. Comparison operators (**==**, **!=**, **<**, **<=**, **>**, and **>=**) compare elements.
- **list<Elem>**: Provides all the common and usual operations, except subscripting. We can **insert()** and **erase()** without moving other elements. Needs two words extra (for link pointers) for each element. A **list** is expandable. Comparison operators (**==**, **!=**, **<**, **<=**, **>**, and **>=**) compare elements.



As we have seen (§17.2, §18.6), arrays are useful and necessary for dealing with memory at the lowest possible level and for interfacing with code written in C (§27.1.2, §27.5). Apart from that, **vector** is preferred because it is easier to use, more flexible, and safer.

TRY THIS



What does that list of differences mean in real code? For each array of **char**, **vector<char>**, **list<char>**, and **string**, define one with the value "**Hello**", pass it to a function as an argument, write out the number of characters in the string passed, try to compare it to "**Hello**" in that function (to see if you really did pass "**Hello**"), and compare the argument to "**Howdy**" to see which would come first in a dictionary. Copy the argument into another variable of the same type.

TRY THIS



Do the previous **Try this** for an array of **int**, **vector<int>**, and **list<int>** each with the value {**1, 2, 3, 4, 5**}.

20.7.1 **insert** and **erase**

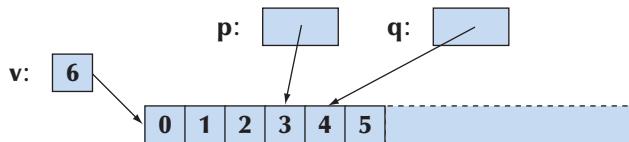
The standard library **vector** is our default choice for a container. It has most of the desired features, so we use alternatives only if we have to. Its main problem is its habit of moving elements when we do list operations (**insert()** and **erase()**); that can be costly when we deal with **vectors** with many elements or **vectors** of large elements. Don't be too worried about that, though. We have been quite happy

reading half a million floating-point values into a **vector** using **push_back()** – measurements confirmed that pre-allocation didn't make a noticeable difference. Always measure before making significant changes in the interest of performance; even for experts, guessing about performance is very hard.

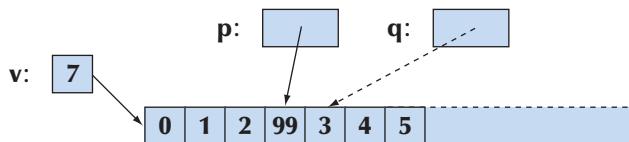
As pointed out in §20.6, moving elements also implies a logical constraint: don't hold iterators or pointers to elements of a **vector** when you do list operations (such as **insert()**, **erase()**, and **push_back()**): if an element moves, your iterator or pointer will point to the wrong element or to no element at all. This is the principal advantage of **lists** (and **maps**; see §21.6) over **vectors**. If you need a collection of large objects or of objects that you point to from many places in a program, consider using a **list**.

Let's compare **insert()** and **erase()** for a **vector** and a **list**. First we take an example designed only to illustrate the key points:

```
vector<int>::iterator p = v.begin();           // take a vector
+p; +p; +p;                                // point to its 4th element
auto q = p;
+q;                                         // point to its 5th element
```

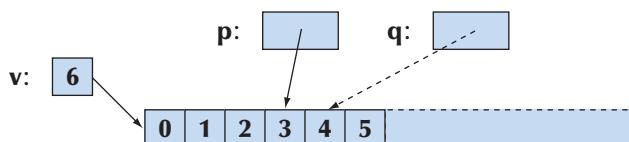


```
p = v.insert(p, 99);                         // p points at the inserted element
```



Note that **q** is now invalid. The elements may have been reallocated as the size of the vector grew. If **v** had spare capacity, so that it grew in place, **q** most likely points to the element with the value **3** rather than the element with the value **4**, but don't try to take advantage of that.

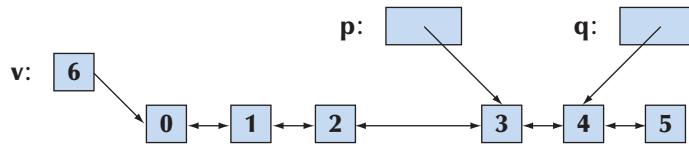
```
p = v.erase(p);                             // p points at the element after the erased one
```



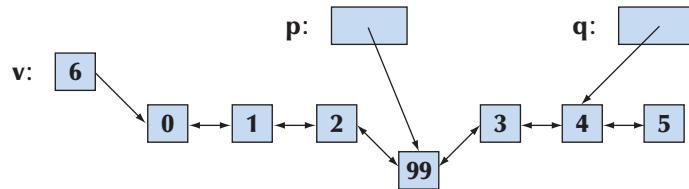
That is, an `insert()` followed by an `erase()` of the inserted element leaves us back where we started, but with `q` invalidated. However, in between, we moved all the elements after the insertion point, and maybe all elements were relocated as `v` grew.

To compare, we'll do exactly the same with a `list`:

```
list<int>::iterator p = v.begin();      // take a list
++p; ++p; ++p;
auto q = p;                          // point to its 4th element
++q;                                  // point to its 5th element
```

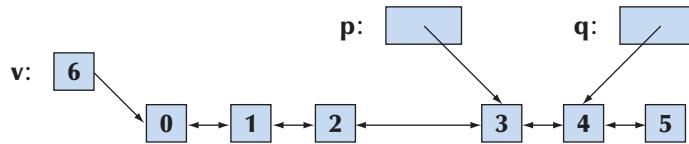


```
p = v.insert(p, 99);      // p points at the inserted element
```



Note that `q` still points to the element with the value 4.

```
p = v.erase(p);          // p points at the element after the erased one
```



Again we find ourselves back where we started. However, for `list` as opposed to `vector`, we didn't move any elements and `q` was valid at all times.

A `list<char>` takes up at least three times as much memory as the other three alternatives – on a PC a `list<char>` uses 12 bytes per element; a `vector<char>` uses 1 byte per element. For large numbers of characters, that can be significant.

In what way is a `vector` superior to a `string`? Looking at the lists of their properties, it seems that a `string` can do all that a `vector` can, and more. That's part of



the problem: since `string` has to do more things, it is harder to optimize. In fact, `vector` tends to be optimized for “memory operations” such as `push_back()`, whereas `string` tends not to be. Instead, `string` tends to be optimized for handling of copying, for dealing with short strings, and for interaction with C-style strings. In the text editor example, we chose `vector` because we were using `insert()` and `delete()`. That is a performance reason, though. The major logical difference is that you can have a `vector` of just about any element type. We have a choice only when we are thinking about characters. In conclusion, prefer `vector` to `string` unless you need string operations, such as concatenation or reading whitespace-separated words.

20.8 Adapting our `vector` to the STL

After adding `begin()`, `end()`, and the type aliases in §20.5, `vector` now just lacks `insert()` and `erase()` to be as close an approximation of `std::vector` as we need it to be:

```
template<typename T, typename A = allocator<T>>
    // requires Element<T>() && Allocator<A>() (§19.3.3)
class vector {
    int sz;           // the size
    T* elem;         // a pointer to the elements
    int space;        // number of elements plus number of free space "slots"
    A alloc;          // use allocate to handle memory for elements
public:
    // . . . all the other stuff from Chapter 19 and §20.5 . .
    using iterator = T*;    // T* is the simplest possible iterator

    iterator insert(iterator p, const T& val);
    iterator erase(iterator p);
};
```

We again used a pointer to the element type, `T*`, as the iterator type. That’s the simplest possible solution. We left providing a range-checked iterator as an exercise (exercise 18).

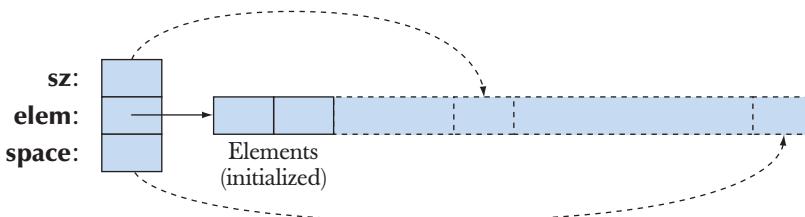
Typically, people don’t provide list operations, such as `insert()` and `erase()`, for data types that keep their elements in contiguous storage, such as `vector`. However, list operations, such as `insert()` and `erase()`, are immensely useful and surprisingly efficient for short `vectors` or small numbers of elements. We have repeatedly seen the usefulness of `push_back()`, which is another operation traditionally associated with lists.



Basically, we implement `vector<T,A>::erase()` by copying all elements after the element we erase (remove, delete). Using the definition of `vector` from §19.3.7 with the additions above, we get

```
template<typename T, typename A>           // requires Element<T>() &&
                                            // Allocator<A>() (§19.3.3)
vector<T,A>::iterator vector<T,A>::erase(iterator p)
{
    if (p==end()) return p;
    for (auto pos = p+1; pos!=end(); ++pos)
        *(pos-1) = *pos;          // copy element "one position to the left"
    alloc.destroy(&*(end()-1)); // destroy surplus copy of last element
    --sz;
    return p;
}
```

It is easier to understand such code if you look at a graphical representation:



The code for `erase()` is quite simple, but it may be a good idea to try out a couple of examples by drawing them on paper. Is the empty `vector` correctly handled? Why do we need the `p==end()` test? What if we erased the last element of a `vector`? Would this code have been easier to read if we had used the subscript notation?

Implementing `vector<T,A>::insert()` is a bit more complicated:

```
template<typename T, typename A>           // requires Element<T>() &&
                                            // Allocator<A>() (§19.3.3)
vector<T,A>::iterator vector<T,A>::insert(iterator p, const T& val)
{
    int index = p->begin();
    if (size()==capacity())
        reserve(size()==0?8:2*size()); // make sure we have space

    // first copy last element into uninitialized space:
    alloc.construct(elem+sz,*back());
```

```

    ++sz;
iterator pp = begin() + index; // the place to put val
for (auto pos = end() - 1; pos != pp; --pos)
    *pos = *(pos - 1);           // copy elements one position to the right
*(begin() + index) = val;      // "insert" val
return pp;
}

```

Please note:

- An iterator may not point outside its sequence, so we use pointers, such as `elem+sz`, for that. That's one reason that allocators are defined in terms of pointers and not iterators.
- When we use `reserve()`, the elements may be moved to a new area of memory. Therefore, we must remember the index at which the element is to be inserted, rather than the iterator to it. When `vector` reallocates its elements, iterators into that `vector` become invalid – you can think of them as pointing to the old memory.
- Our use of the allocator argument, `A`, is intuitive, but inaccurate. If you should ever need to implement a container, you'll have to do some careful reading of the standard.
- It is subtleties like these that make us avoid dealing with low-level memory issues whenever we can. Naturally, the standard library `vector` – and all other standard library containers – get that kind of important semantic detail right. That's one reason to prefer the standard library over “home brew.”

For performance reasons, you wouldn't use `insert()` and `erase()` in the middle of a 100,000-element `vector`; for that, `lists` (and `maps`; see §21.6) are better. However, the `insert()` and `erase()` operations are available for all `vectors`, and their performance is unbeatable when you are just moving a few words of data – or even a few dozen words – because modern computers are really good at this kind of copying; see exercise 20. Avoid (linked) `lists` for representing a list of a few small elements.

20.9 Adapting built-in arrays to the STL

We have repeatedly pointed out the weaknesses of the built-in arrays: they implicitly convert to pointers at the slightest provocation, they can't be copied using assignment, they don't know their own size (§18.6.2), etc. We have also pointed out their main strength: they model physical memory almost perfectly.

To get the best of both worlds, we can build an `array` container that provides the benefits of arrays without the weaknesses. A version of `array` was introduced into the standard as part of a Technical Report. Since a feature from a TR is not required to be part of every implementation, `array` may not be part of the implementation you use. However, the idea is simple and useful:



```

template <typename T, int N> // requires Element<T>()
struct array { // not quite the standard array
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;
    using size_type = unsigned int; // the type of a subscript

    T elems[N];
    // no explicit construct/copy/destroy needed

    iterator begin() { return elems; }
    const_iterator begin() const { return elems; }
    iterator end() { return elems+N; }
    const_iterator end() const { return elems+N; }

    size_type size() const;

    T& operator[](int n) { return elems[n]; }
    const T& operator[](int n) const { return elems[n]; }

    const T& at(int n) const; // range-checked access
    T& at(int n); // range-checked access

    T * data() { return elems; }
    const T * data() const { return elems; }
};

```

This definition isn't complete or completely standards-conforming, but it will give you the idea. It will also give you something to use if your implementation doesn't yet provide the standard `array`. If available, it is in `<array>`. Note that because `array<T,N>` "knows" that its size is `N`, we can (and do) provide assignment, `==`, `!=`, etc. just as for `vector`.

As an example, let's use an array with the STL version of `high()` from §20.4.2:

```

void f()
{
    array<double,6> a = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
    array<double,6>::iterator p = high(a.begin(), a.end());

```

```

    cout << "the highest value was " << *p << '\n';
}

```

Note that we did not think of **array** when we wrote **high()**. Being able to use **high()** for an **array** is a simple consequence of following standard conventions for both.

20.10 Container overview

The STL provides quite a few containers:

Standard containers	
vector	a contiguously allocated sequence of elements; use it as the default container
list	a doubly-linked list; use it when you need to insert and delete elements without moving existing elements
deque	a cross between a list and a vector ; don't use it until you have expert-level knowledge of algorithms and machine architecture
map	a balanced ordered tree; use it when you need to access elements by value (see §21.6.1–3)
multimap	a balanced ordered tree where there can be multiple copies of a key; use it when you need to access elements by value (see §21.6.1–3)
unordered_map	a hash table; an optimized version of map ; use for large maps when you need high performance and can devise a good hash function (see §21.6.4)
unordered_multimap	a hash table where there can be multiple copies of a key; an optimized version of multimap ; use it for large maps when you need high performance and can devise a good hash function (see §21.6.4)
set	a balanced ordered tree; use it when you need to keep track of individual values (see §21.6.5)
multiset	a balanced ordered tree where there can be multiple copies of a key; use it when you need to keep track of individual values (see §21.6.5)
unordered_set	like unordered_map , but just with values, not (key,value) pairs
unordered_multiset	like unordered_multimap , but just with values, not (key,value) pairs
array	a fixed-size array that doesn't suffer most of the problems related to the built-in arrays (see §20.9)

You can look up incredible amounts of additional information on these containers and their use in books and online documentation. Here are a few quality information sources:

Josuttis, Nicholai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 2012. ISBN 978-0321623218. Use only the 2nd edition.

Lippman, Stanley B., Jose Lajoie, and Barbara E. Moo. *The C++ Primer*. Addison-Wesley, 2005. ISBN 0201721481. Use only the 5th edition.

Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, 2012. ISBN 978-0321714114. Use only the 4th edition.

The documentation for the SGI implementation of the STL and the iostream library: www.sgi.com/tech/stl. Note that they also provide complete code.

Do you feel cheated? Do you think we should explain all about containers and their use to you? That's just not possible. There are too many standard facilities, too many useful techniques, and too many useful libraries for you to absorb them all at once. Programming is too rich a field for anyone to know all facilities and techniques – it can also be a noble art. As a programmer, you must acquire the habit of seeking out new information about language facilities, libraries, and techniques. Programming is a dynamic and rapidly developing field, so just being content with what you know and are comfortable with is a recipe for being left behind. “Look it up” is a perfectly reasonable answer to many problems, and as your skills grow and mature, it will more and more often be the answer.

On the other hand, you will find that once you understand `vector`, `list`, and `map` and the standard algorithms presented in Chapter 21, you'll find other STL and STL-style containers easy to use. You'll also find that you have the basic knowledge to understand non-STL containers and code using them.

What is a container? You can find the definition of an STL container in all of the sources above. Here we will just give an informal definition. An STL container

- Is a sequence of elements [`begin()`:`end()`].
- Provides copy operations that copy elements. Copying can be done with assignment or a copy constructor.
- Names its element type `value_type`.
- Has iterator types called `iterator` and `const_iterator`. Iterators provide `*`, `++` (both prefix and postfix), `==`, and `!=` with the appropriate semantics. The iterators for `list` also provide `--` for moving backward in the sequence; that's called a *bidirectional iterator*. The iterators for `vector` also provide `--`, `[]`, `+`, and `-` and are called *random-access iterators*. (See §20.10.1.)

- Provides `insert()` and `erase()`, `front()` and `back()`, `push_back()` and `pop_back()`, `size()`, etc.; `vector` and `map` also provide subscripting (e.g., operator `[]`).
- Provides comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`) that compare the elements. Containers use lexicographical ordering for `<`, `<=`, `>`, and `>=`; that is, they compare the elements in order starting with the first.

The aim of this list is to give you an overview. For more detail see Appendix B. For a more precise specification and complete list, see *The C++ Programming Language* or the standard.

Some data types provide much of what is required from a standard container, but not all. We sometimes refer to those as “almost containers.” The most interesting of those are:

“Almost containers”	
<code>T[n]</code> built-in array	no <code>size()</code> or other member functions; prefer a container, such as <code>vector</code> , <code>string</code> , or <code>array</code> , over a built-in array when you have a choice
<code>string</code>	holds only characters but provides operations useful for text manipulation, such as concatenation (<code>+</code> and <code>+=</code>); prefer the standard string to other strings
<code>valarray</code>	a numerical vector with vector operations, but with many restrictions to encourage high-performance implementations; use only if you do a lot of vector arithmetic

In addition, many people and many organizations have produced containers that meet the standard container requirements, or almost do so.

If in doubt, use `vector`. Unless you have a solid reason not to, use `vector`.



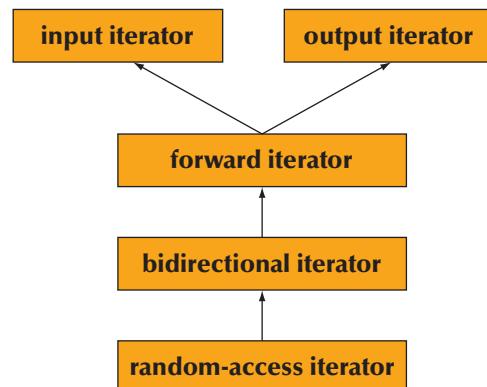
20.10.1 Iterator categories

We have talked about iterators as if all iterators are interchangeable. They are interchangeable if you do only the simplest operations, such as traversing a sequence once reading each value once. If you want to do more, such as iterating backward or subscripting, you need one of the more advanced iterators.

Iterator categories

input iterator	We can iterate forward using <code>++</code> and read element values using <code>*</code> . This is the kind of iterator that <code>istream</code> offers; see §21.7.2. If <code>(*p).m</code> is valid, <code>p->m</code> can be used as a shorthand.
output iterator	We can iterate forward using <code>++</code> and write element values using <code>*</code> . This is the kind of iterator that <code>ostream</code> offers; see §21.7.2.
forward iterator	We can iterate forward repeatedly using <code>++</code> and read and write (unless the elements are <code>const</code> , of course) element values using <code>*</code> . If <code>(*p).m</code> is valid, <code>p->m</code> can be used as a shorthand.
bidirectional iterator	We can iterate forward (using <code>++</code>) and backward (using <code>--</code>) and read and write (unless the elements are <code>const</code>) element values using <code>*</code> . This is the kind of iterator that <code>list</code> , <code>map</code> , and <code>set</code> offer. If <code>(*p).m</code> is valid, <code>p->m</code> can be used as a shorthand.
random-access iterator	We can iterate forward (using <code>++</code>) and backward (using <code>--</code>) and read and write (unless the elements are <code>const</code>) element values using <code>*</code> or <code>[]</code> . We can subscript and add an integer to a random-access iterator using <code>+</code> and subtract an integer using <code>-</code> . We can find the distance between two random-access iterators to the same sequence by subtracting one from the other. This is the kind of iterator that <code>vector</code> offers. If <code>(*p).m</code> is valid, <code>p->m</code> can be used as a shorthand.

From the operations offered, we can see that wherever we can use an output iterator or an input iterator, we can use a forward iterator. A bidirectional iterator is also a forward iterator, and a random-access iterator is also a bidirectional iterator. Graphically, we can represent the iterator categories like this:



Note that since the iterator categories are not classes, this hierarchy is not a class hierarchy implemented using derivation.



Drill

1. Define an array of **ints** with the ten elements { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }.
2. Define a **vector<int>** with those ten elements.
3. Define a **list<int>** with those ten elements.
4. Define a second array, vector, and list, each initialized as a copy of the first array, vector, and list, respectively.
5. Increase the value of each element in the array by 2; increase the value of each element in the vector by 3; increase the value of each element in the list by 5.
6. Write a simple **copy()** operation,

```
template<typename Iter1, typename Iter2>
    // requires Input_iterator<Iter1>() && Output_iterator<Iter2>()
Iter2 copy(Iter1 f1, Iter1 e1, Iter2 f2);
```

that copies **[f1,e1)** to **[f2,f2+(e1-f1))** and returns **f2+(e1-f1)** just like the standard library **copy** function. Note that if **f1==e1** the sequence is empty, so that there is nothing to copy.

7. Use your **copy()** to copy the array into the vector and to copy the list into the array.
8. Use the standard library **find()** to see if the vector contains the value 3 and print out its position if it does; use **find()** to see if the list contains the value 27 and print out its position if it does. The “position” of the first element is 0, the position of the second element is 1, etc. Note that if **find()** returns the end of the sequence, the value wasn’t found.

Remember to test after each step.

Review

1. Why does code written by different people look different? Give examples.
2. What are simple questions we ask of data?
3. What are a few different ways of storing data?
4. What basic operations can we do to a collection of data items?
5. What are some ideals for the way we store our data?
6. What is an STL sequence?
7. What is an STL iterator? What operations does it support?
8. How do you move an iterator to the next element?
9. How do you move an iterator to the previous element?
10. What happens if you try to move an iterator past the end of a sequence?
11. What kinds of iterators can you move to the previous element?
12. Why is it useful to separate data from algorithms?

13. What is the STL?
14. What is a linked list? How does it fundamentally differ from a vector?
15. What is a link (in a linked list)?
16. What does `insert()` do? What does `erase()` do?
17. How do you know if a sequence is empty?
18. What operations does an iterator for a `list` provide?
19. How do you iterate over a container using the STL?
20. When would you use a `string` rather than a `vector`?
21. When would you use a `list` rather than a `vector`?
22. What is a container?
23. What should `begin()` and `end()` do for a container?
24. What containers does the STL provide?
25. What is an iterator category? What kinds of iterators does the STL offer?
26. What operations are provided by a random-access iterator, but not a bidirectional iterator?

Terms

algorithm	empty sequence	singly-linked list
<code>array</code> container	<code>end()</code>	<code>size_type</code>
<code>auto</code>	<code>erase()</code>	STL
<code>begin()</code>	<code>insert()</code>	traversal
container	iteration	<code>using</code>
contiguous	iterator	type alias
doubly-linked list	linked list	<code>value_type</code>
element	sequence	

Exercises

1. If you haven't already, do all **Try this** exercises in the chapter.
2. Get the Jack-and-Jill example from §20.1.2 to work. Use input from a couple of small files to test it.
3. Look at the palindrome examples (§18.7); redo the Jack-and-Jill example from §20.1.2 using that variety of techniques.
4. Find and fix the errors in the Jack-and-Jill example from §20.3.1 by using STL techniques throughout.
5. Define an input and an output operator (`>>` and `<<`) for `vector`.
6. Write a find-and-replace operation for `Documents` based on §20.6.2.
7. Find the lexicographical last string in an unsorted `vector<string>`.
8. Define a function that counts the number of characters in a `Document`.

9. Define a program that counts the number of words in a **Document**. Provide two versions: one that defines *word* as “a whitespace-separated sequence of characters” and one that defines *word* as “a sequence of consecutive alphabetic characters.” For example, with the former definition, **alpha.numeric** and **as12b** are both single words, whereas with the second definition they are both two words.
10. Define a version of the word-counting program where the user can specify the set of whitespace characters.
11. Given a **list<int>** as a (by-reference) parameter, make a **vector<double>** and copy the elements of the list into it. Verify that the copy was complete and correct. Then print the elements sorted in order of increasing value.
12. Complete the definition of **list** from §20.4.1–2 and get the **high()** example to run. Allocate a **Link** to represent one past the end.
13. We don’t really need a “real” one-past-the-end **Link** for a **list**. Modify your solution to the previous exercise to use **0** to represent a pointer to the (nonexistent) one-past-the-end **Link** (**list<Elem>::end()**); that way, the size of an empty list can be equal to the size of a single pointer.
14. Define a singly-linked list, **slist**, in the style of **std::list**. Which operations from **list** could you reasonably eliminate from **slist** because it doesn’t have back pointers?
15. Define a **pvector** to be like a **vector** of pointers except that it contains pointers to objects and its destructor **deletes** each object.
16. Define an **ovector** that is like **pvector** except that the **[]** and ***** operators return a reference to the object pointed to by an element rather than the pointer.
17. Define an **ownership_vector** that hold pointers to objects like **pvector** but provides a mechanism for the user to decide which objects are owned by the vector (i.e., which objects are **deleted** by the destructor). Hint: This exercise is simple if you were awake for Chapter 13.
18. Define a range-checked iterator for **vector** (a random-access iterator).
19. Define a range-checked iterator for **list** (a bidirectional iterator).
20. Run a small timing experiment to compare the cost of using **vector** and **list**. You can find an explanation of how to time a program in §26.6.1. Generate N random **int** values in the range $[0:N]$. As each **int** is generated, insert it into a **vector<int>** (which grows by one element each time). Keep the **vector** sorted; that is, a value is inserted after every previous value that is less than or equal to the new value and before every previous value that is larger than the new value. Now do the same experiment using a **list<int>** to hold the **ints**. For which N is the **list** faster than the **vector**? Try to explain your result. This experiment was first suggested by John Bentley.

Postscript



If we have N kinds of containers of data and M things we'd like to do with them, we can easily end up writing $N*M$ pieces of code. If the data is of K different types, we could even end up with $N*M*K$ pieces of code. The STL addresses this proliferation by having the element type as a parameter (taking care of the K factor) and by separating access to data from algorithms. By using iterators to access data in any kind of container from any algorithm, we can make do with $N+M$ algorithms. This is a huge simplification. For example, if we have 12 containers and 60 algorithms, the brute-force approach would require 720 functions, whereas the STL strategy requires only 60 functions and 12 definitions of iterators: we just saved ourselves 90% of the work. In fact, this underestimates the saved effort because many algorithms take two pairs of iterators and the pairs need not be of the same type (e.g., see exercise 6). In addition, the STL provides conventions for defining algorithms that simplify writing correct code and composable code, so the saving is greater still.



Algorithms and Maps

“In theory, practice is simple.”

—Trygve Reenskaug

This chapter completes our presentation of the fundamental ideas of the STL and our survey of the facilities it offers. Here, we focus on algorithms. Our primary aim is to introduce you to about a dozen of the most useful ones, which will save you days, if not months, of work. Each is presented with examples of its uses and of programming techniques that it supports. Our second aim here is to give you sufficient tools to write your own – elegant and efficient – algorithms if and when you need more than what the standard library and other available libraries have to offer. In addition, we introduce three more containers: **map**, **set**, and **unordered_map**.

21.1 Standard library algorithms	21.6 Associative containers
21.2 The simplest algorithm: <code>find()</code>	21.6.1 <code>map</code>
21.2.1 Some generic uses	21.6.2 <code>map</code> overview
21.3 The general search: <code>find_if()</code>	21.6.3 Another <code>map</code> example
21.4 Function objects	21.6.4 <code>unordered_map</code>
21.4.1 An abstract view of function objects	21.6.5 <code>set</code>
21.4.2 Predicates on class members	21.7 Copying
21.4.3 Lambda expressions	21.7.1 <code>Copy</code>
21.5 Numerical algorithms	21.7.2 <code>Stream iterators</code>
21.5.1 <code>Accumulate</code>	21.7.3 Using a <code>set</code> to keep order
21.5.2 Generalizing <code>accumulate()</code>	21.7.4 <code>copy_if</code>
21.5.3 Inner product	21.8 Sorting and searching
21.5.4 Generalizing <code>inner_product()</code>	21.9 Container algorithms

21.1 Standard library algorithms

The standard library offers about 80 algorithms. All are useful for someone sometimes; we focus on some that are often useful for many and on some that are occasionally very useful for someone:

Selected standard algorithms	
<code>r=find(b,e,v)</code>	<code>r</code> points to the first occurrence of <code>v</code> in <code>[b:e)</code> .
<code>r=find_if(b,e,p)</code>	<code>r</code> points to the first element <code>x</code> in <code>[b:e)</code> so that <code>p(x)</code> is <code>true</code> .
<code>x=count(b,e,v)</code>	<code>x</code> is the number of occurrences of <code>v</code> in <code>[b:e)</code> .
<code>x=count_if(b,e,p)</code>	<code>x</code> is the number of elements in <code>[b:e)</code> so that <code>p(x)</code> is <code>true</code> .
<code>sort(b,e)</code>	Sort <code>[b:e)</code> using <code><</code> .
<code>sort(b,e,p)</code>	Sort <code>[b:e)</code> using <code>p</code> .
<code>copy(b,e,b2)</code>	Copy <code>[b:e)</code> to <code>[b2:b2+(e-b))</code> ; there had better be enough elements after <code>b2</code> .
<code>unique_copy(b,e,b2)</code>	Copy <code>[b:e)</code> to <code>[b2:b2+(e-b))</code> ; don't copy adjacent duplicates.
<code>merge(b,e,b2,e2,r)</code>	Merge two sorted sequences <code>[b2:e2)</code> and <code>[b:e)</code> into <code>[r:r+(e-b)+(e2-b2))</code> .
<code>r=equal_range(b,e,v)</code>	<code>r</code> is the subsequence of the sorted range <code>[b:e)</code> with the value <code>v</code> , basically, a binary search for <code>v</code> .

Selected standard algorithms	
<code>equal(b,e,b2)</code>	Do all elements of <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> compare equal?
<code>x=accumulate(b,e,i)</code>	<code>x</code> is the sum of <code>i</code> and the elements of <code>[b:e)</code> .
<code>x=accumulate(b,e,i,op)</code>	Like the other <code>accumulate</code> , but with the “sum” calculated using <code>op</code> .
<code>x=inner_product(b,e,b2,i)</code>	<code>x</code> is the inner product of <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> .
<code>x=inner_product(b,e,b2,i,op,op2)</code>	Like the other <code>inner_product</code> , but with <code>op</code> and <code>op2</code> instead of <code>+</code> and <code>*</code> .

By default, comparison for equality is done using `==` and ordering is done based on `<` (less than). The standard library algorithms are found in `<algorithm>`. For more information, see §B.5 and the sources listed in §21.2–21.5. These algorithms take one or more sequences. An input sequence is defined by a pair of iterators; an output sequence is defined by an iterator to its first element. Typically an algorithm is parameterized by one or more operations that can be defined as function objects or as functions. The algorithms usually report “failure” by returning the end of an input sequence. For example, `find(b,e,v)` returns `e` if it doesn’t find `v`.

21.2 The simplest algorithm: `find()`

Arguably, the simplest useful algorithm is `find()`. It finds an element with a given value in a sequence:

```
template<typename In, typename T>
    // requires Input_iterator<In>
    //     && Equality_comparable<Value_type<T>>() (§19.3.3)
In find(In first, In last, const T& val)
    // find the first element in [first,last) that equals val
{
    while (first!=last && *first != val) ++first;
    return first;
}
```

Let’s have a look at the definition of `find()`. Naturally, you can use `find()` without knowing exactly how it is implemented – in fact, we have used it already (e.g., §20.6.2). However, the definition of `find()` illustrates many useful design ideas, so it is worth looking at.

First of all, `find()` operates on a sequence defined by a pair of iterators. We are looking for the value `val` in the half-open sequence `[first:last)`. The result returned

by **find()** is an iterator. That result points either to the first element of the sequence with the value **val** or to **last**. Returning an iterator to the one-beyond-the-last element of a sequence is the most common STL way of reporting “not found.” So we can use **find()** like this:

```
void f(vector<int>& v, int x)
{
    auto p = find(v.begin(),v.end(),x);
    if (p!=v.end()) {
        // we found x in v
    }
    else {
        // no x in v
    }
    // ...
}
```

Here, as is common, the sequence consists of all the elements of a container (an STL **vector**). We check the returned iterator against the end of our sequence to see if we found our value. The type of the value returned is the iterator passed as an argument.

To avoid naming the type returned, we used **auto**. An object defined with the “type” **auto** gets the type of its initializer. For example:

```
auto ch = 'c';    // ch is a char
auto d = 2.1;    // d is a double
```

The **auto** type specifier is particularly useful in generic code, such as **find()** where it can be tedious to name the actual type (here, **vector<int>::iterator**).

We now know how to use **find()** and therefore also how to use a bunch of other algorithms that follow the same conventions as **find()**. Before proceeding with more uses and more algorithms, let’s just have a closer look at that definition:

```
template<typename In, typename T>
    // requires Input_iterator<In>()
    //     && Equality_comparable<Value_type<T>>() ($19.3.3)
In find(In first, In last, const T& val)
    // find the first element in [first,last) that equals val
{
    while (first!=last && *first != val) ++first;
    return first;
}
```

Did you find that loop obvious at first glance? We didn't. It is actually minimal, efficient, and a direct representation of the fundamental algorithm. However, until you have seen a few examples, it is not obvious. Let's write it "the pedestrian way" and see how that version compares:

```
template<typename In, typename T>
    // requires Input_iterator<In>
    //     && Equality_comparable<Value_type<T>>() (§19.3.3)
In find(In first, In last, const T& val)
    // find the first element in [first,last) that equals val
{
    for (In p = first; p!=last; ++p)
        if (*p == val) return p;
    return last;
}
```

These two definitions are logically equivalent, and a really good compiler will generate the same code for both. However, in reality many compilers are not good enough to eliminate that extra variable (`p`) and to rearrange the code so that all the testing is done in one place. Why worry and explain? Partly, because the style of the first (and preferred) version of `find()` has become very popular, and you must understand it to read other people's code; partly, because performance matters exactly for small, frequently used functions that deal with lots of data.

TRY THIS



Are you sure those two definitions are logically equivalent? How would you be sure? Try constructing an argument for their being equivalent. That done, try both on some data. A famous computer scientist (Don Knuth) once said, "I have only proven the algorithm correct, not tested it." Even mathematical proofs can contain errors. To be confident, you need to both reason and test.

21.2.1 Some generic uses

The `find()` algorithm is generic. That means that it can be used for different data types. In fact, it is generic in two ways; it can be used for

- Any STL-style sequence
- Any element type



Here are some examples (consult the diagrams in §20.4 if you get confused):

```
void f(vector<int>& v, int x)           // works for vector of int
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) /* we found x */
    // ...
}
```

Here, the iteration operations used by `find()` are those of a `vector<int>::iterator`; that is, `++` (in `++first`) simply moves a pointer to the next location in memory (where the next element of the `vector` is stored) and `*` (in `*first`) dereferences such a pointer. The iterator comparison (in `first!=last`) is a pointer comparison, and the value comparison (in `*first!=val`) simply compares two integers.

Let's try with a `list`:

```
void f(list<string>& v, string x)           // works for list of string
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) /* we found x */
    // ...
}
```

Here, the iteration operations used by `find()` are those of a `list<string>::iterator`. The operators have the required meaning, so that the logic is the same as for the `vector<int>` above. The implementation is very different, though; that is, `++` (in `++first`) simply follows a pointer in the `Link` part of the element to where the next element of the `list` is stored, and `*` (in `*first`) finds the value part of a `Link`. The iterator comparison (in `first!=last`) is a pointer comparison of `Link`'s and the value comparison (in `*first!=val`) compares `strings` using `string`'s `!=` operator.

So, `find()` is extremely flexible: as long as we obey the simple rules for iterators, we can use `find()` to find elements for any sequence we can think of and for any container we care to define. For example, we can use `find()` to look for a character in a `Document` as defined in §20.6:

```
void f(Document& v, char x)           // works for Document of char
{
    Text_iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) /* we found x */
    // ...
}
```

This kind of flexibility is the hallmark of the STL algorithms and makes them more useful than most people imagine when they first encounter them.

21.3 The general search: `find_if()`

We don't actually look for a specific value all that often. More often, we are interested in finding a value that meets some criteria. We could get a much more useful `find` operation if we could define our search criteria ourselves. Maybe we want to find a value larger than 42. Maybe we want to compare strings without taking case (upper case vs. lower case) into account. Maybe we want to find the first odd value. Maybe we want to find a record where the address field is "[17 Cherry Tree Lane](#)".

The standard algorithm that searches based on a user-supplied criterion is `find_if()`:

```
template<typename In, typename Pred>
    // requires Input_iterator<In>() && Predicate<Pred, Value_type<In>>()
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

Obviously (when you compare the source code), it is just like `find()` except that it uses `!pred(*first)` rather than `*first!=val`; that is, it stops searching once the predicate `pred()` succeeds rather than when an element equals a value.

A predicate is a function that returns `true` or `false`. Clearly, `find_if()` requires a predicate that takes one argument so that it can say `pred(*first)`. We can easily write a predicate that checks some property of a value, such as "Does the string contain the letter *x*?" "Is the value larger than 42?" "Is the number odd?" For example, we can find the first odd value in a vector of `ints` like this:

```
bool odd(int x) { return x%2; }      // % is the modulo operator

void f(vector<int>& v)
{
    auto p = find_if(v.begin(), v.end(), odd);
    if (p!=v.end()) { /* we found an odd number */
        // ...
    }
```

For that call of `find_if()`, `find_if()` calls `odd()` for each element until it finds the first odd value. Note that when you pass a function as an argument, you don't add `()` to its name because doing so would call it.

Similarly, we can find the first element of a list with a value larger than 42 like this:

```
bool larger_than_42(double x) { return x>42; }

void f(list<double>& v)
{
    auto p = find_if(v.begin(), v.end(), larger_than_42);
    if (p!=v.end()) { /* we found a value > 42 */
        // ...
    }
```

This last example is not very satisfying, though. What if we next wanted to find an element larger than 41? We would have to write a new function. Find an element larger than 19? Write yet another function. There has to be a better way!

If we want to compare to an arbitrary value `v`, we need somehow to make `v` an implicit argument to `find_if()`'s predicate. We could try (choosing `v_val` as a name that is less likely to clash with other names)

```
double v_val;      // the value to which larger_than_v() compares its argument
bool larger_than_v(double x) { return x>v_val; }

void f(list<double>& v, int x)
{
    v_val = 31;      // set v_val to 31 for the next call of larger_than_v
    auto p = find_if(v.begin(), v.end(), larger_than_v);
    if (p!=v.end()) { /* we found a value > 31 */

        v_val = x;      // set v_val to x for the next call of larger_than_v
        auto q = find_if(v.begin(), v.end(), larger_than_v);
        if (q!=v.end()) { /* we found a value > x */

        // ...
    }
```

 Yuck! We are convinced that people who write such code will eventually get what they deserve, but we pity their users and anyone who gets to maintain their code. Again: there has to be a better way!

TRY THIS


Why are we so disgusted with that use of `v`? Give at least three ways this could lead to obscure errors. List three applications in which you'd particularly hate to find such code.

21.4 Function objects

So, we want to pass a predicate to `find_if()`, and we want that predicate to compare elements to a value we specify as some kind of argument. In particular, we want to write something like this:

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* we found a value > 31 */ }

    auto q = find_if(v.begin(), v.end(), Larger_than(x));
    if (q!=v.end()) { /* we found a value > x */ }

    // ...
}
```

Obviously, `Larger_than` must be something that

- We can call as a predicate, e.g., `pred(*first)`
- Can store a value, such as `31` or `x`, for use when called

For that we need a “function object,” that is, an object that can behave like a function. We need an object because objects can store data, such as the value with which to compare. For example:

```
class Larger_than {
    int v;
public:
    Larger_than(int vv) : v(vv) {}           // store the argument
    bool operator()(int x) const { return x>v; } // compare
};
```

Interestingly, this definition makes the example above work as specified. Now we just have to figure out why it works. When we say **Larger_than(31)** we (obviously) make an object of class **Larger_than** holding **31** in its data member **v**. For example:

```
find_if(v.begin(),v.end(),Larger_than(31))
```

Here, we pass that object to **find_if()** as its parameter called **pred**. For each element of **v**, **find_if()** makes a call

```
pred(*first)
```

This invokes the call operator, called **operator()**, for our function object using the argument ***first**. The result is a comparison of the element's value, ***first**, with **31**.

What we see here is that a function call can be seen as an operator, the “**()** operator,” just like any other operator. The “**()** operator” is also called the *function call operator* and the *application operator*. So **()** in **pred(*first)** is given a meaning by **Larger_than::operator()**, just as subscripting in **v[i]** is given a meaning by **vector::operator[]**.

21.4.1 An abstract view of function objects

We have here a mechanism that allows for a “function” to “carry around” data that it needs. Clearly, function objects provide us with a very general, powerful, and convenient mechanism. Consider a more general notion of a function object:

```
class F {           // abstract example of a function object
    S s;        // state
public:
    F(const S& ss) :s(ss) { /* establish initial state */ }
    T operator() (const S& ss) const
    {
        // do something with ss to s
        // return a value of type T (T is often void, bool, or S)
    }

    const S& state() const { return s; }           // reveal state
    void reset(const S& ss) { s = ss; }           // reset state
};
```

An object of class **F** holds data in its member **s**. If needed, a function object can have many data members. Another way of saying that something holds data is that it “has state.” When we create an **F**, we can initialize that state. Whenever we want to, we can read that state. For **F**, we provided an operation, **state()**, to

read that state and another, `reset()`, to write it. However, when we design a function object we are free to provide any way of accessing its state that we consider appropriate. And, of course, we can directly or indirectly call the function object using the normal function call notation. We defined `F` to take a single argument when it is called, but we can define function objects with as many parameters as we need.

Use of function objects is the main method of parameterization in the STL. We use function objects to specify what we are looking for in searches (§21.3), for defining sorting criteria (§21.4.2), for specifying arithmetic operations in numerical algorithms (§21.5), for defining what it means for values to be equal (§21.8), and for much more. The use of function objects is a major source of flexibility and generality.

Function objects are usually very efficient. In particular, passing a small function object by value to a template function typically leads to optimal performance. The reason is simple, but surprising to people more familiar with passing functions as arguments: typically, passing a function object leads to significantly smaller and faster code than passing a function! This is true only if the object is small (something like zero, one, or two words of data) or passed by reference and if the function call operator is small (e.g., a simple comparison using `<`) and defined to be inline (e.g., has its definition within its class itself). Most of the examples in this chapter – and in this book – follow this pattern. The basic reason for the high performance of small and simple function objects is that they preserve sufficient type information for compilers to generate optimal code. Even older compilers with unsophisticated optimizers can generate a simple “greater-than” machine instruction for the comparison in `Larger_than` rather than calling a function. Calling a function typically takes 10 to 50 times longer than executing a simple comparison operation. In addition, the code for a function call is several times larger than the code for a simple comparison.

21.4.2 Predicates on class members

As we have seen, standard algorithms work well with sequences of elements of basic types, such as `int` and `double`. However, in some application areas, containers of class values are far more common. Consider an example that is key to applications in many areas, sorting a record by several criteria:

```
struct Record {  
    string name;           // standard string for ease of use  
    char addr[24];         // old style to match database layout  
    // ...  
};  
  
vector<Record> vr;
```

Sometimes we want to sort `vr` by name, and sometimes we want to sort it by address. Unless we can do both elegantly and efficiently, our techniques are of limited practical interest. Fortunately, doing so is easy. We can write

```
// ...
sort(vr.begin(), vr.end(), Cmp_by_name());           // sort by name
// ...
sort(vr.begin(), vr.end(), Cmp_by_addr());           // sort by addr
// ...
```

 `Cmp_by_name` is a function object that compares two `Records` by comparing their `name` members. `Cmp_by_addr` is a function object that compares two `Records` by comparing their `addr` members. To allow the user to specify such comparison criteria, the standard library `sort` algorithm takes an optional third argument specifying the sorting criteria. `Cmp_by_name()` creates a `Cmp_by_name` for `sort()` to use to compare `Records`. That looks OK – meaning that we wouldn't mind maintaining code that looked like that. Now all we have to do is to define `Cmp_by_name` and `Cmp_by_addr`:

```
// different comparisons for Record objects:

struct Cmp_by_name {
    bool operator()(const Record& a, const Record& b) const
    { return a.name < b.name; }
};

struct Cmp_by_addr {
    bool operator()(const Record& a, const Record& b) const
    { return strncmp(a.addr, b.addr, 24) < 0; }      // !!!
};
```

The `Cmp_by_name` class is pretty obvious. The function call operator, `operator()`, simply compares the `name` strings using the standard `string`'s `<` operator. However, the comparison in `Cmp_by_addr` is ugly. That is because we chose an ugly representation of the address: an array of 24 characters (not zero terminated). We chose that partly to show how a function object can be used to hide ugly and error-prone code and partly because this particular representation was once presented to me as a challenge: “an ugly and important real-world problem that the STL can’t handle.” Well, the STL could. The comparison function uses the standard C (and C++) library function `strncmp()` that compares fixed-length character arrays, returning a negative number if the second “string” comes lexicographically after the first. Look it up should you ever need to do such an obscure comparison (e.g., §B.11.3).

21.4.3 Lambda expressions

Defining a function object (or a function) in one place in a program and then using it in another can be a bit tedious. In particular, it is a nuisance if the action we want to perform is very easy to specify, easy to understand, and will never again be needed. In that case, we can use a lambda expression (§15.3.3). Probably the best way of thinking about a lambda expression is as a shorthand notation for defining a function object (a class with an operator ()) and then immediately creating an object of it. For example, we could have written

```
// ...
sort(vr.begin(), vr.end(),           // sort by name
      [](const Record& a, const Record& b)
          { return a.name < b.name; })
);
// ...
sort(vr.begin(), vr.end(),           // sort by addr
      [](const Record& a, const Record& b)
          { return strncmp(a.addr, b.addr, 24) < 0; })
);
// ...
```

In this case, we wonder if a named function object wouldn't give more maintainable code. Maybe **Cmp_by_name** and **Cmp_by_addr** have other uses.

However, consider the **find_if()** example from §21.4. There, we needed to pass an operation as an argument and that operation needed to carry data with it:

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* we found a value > 31 */ }

    auto q = find_if(v.begin(), v.end(), Larger_than(x));
    if (q!=v.end()) { /* we found a value > x */ }

    // ...
}
```

Alternatively, and equivalently, we could write

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), [] (double a) { return a>31; });
    if (p!=v.end()) { /* we found a value > 31 */ }
```

```

auto q = find_if(v.begin(), v.end(), [&](double a) { return a>x; });
if (q!=v.end()) /* we found a value > x */

//...
}

```

The comparison to the local variable `x` makes the lambda version attractive.

21.5 Numerical algorithms

Most of the standard library algorithms deal with data management issues: they copy, sort, search, etc. data. However, a few help with numerical computations. These numerical algorithms can be important when you compute, and they serve as examples of how you can express numerical algorithms within the STL framework.

There are just four STL-style standard library numerical algorithms:

Numerical algorithms	
<code>x=accumulate(b,e,i)</code>	Add a sequence of values; e.g., for {a,b,c,d} produce $i+a+b+c+d$. The type of the result <code>x</code> is the type of the initial value <code>i</code> .
<code>x=inner_product(b,e,b2,i)</code>	Multiply pairs of values from two sequences and sum the results; e.g., for {a,b,c,d} and {e,f,g,h} produce $i+a*e+b*f+c*g+d*h$. The type of the result <code>x</code> is the type of the initial value <code>i</code> .
<code>r=partial_sum(b,e,r)</code>	Produce the sequence of sums of the first n elements of a sequence; e.g., for {a,b,c,d} produce {a, a+b, a+b+c, a+b+c+d}.
<code>r=adjacent_difference(b,e,b2,r)</code>	Produce the sequence of differences between elements of a sequence; e.g., for {a,b,c,d} produce {a,b-a,c-b,d-c}.

They are found in `<numeric>`. We'll describe the first two here and leave it for you to explore the other two if you feel the need.

21.5.1 Accumulate

The simplest and most useful numerical algorithm is `accumulate()`. In its simplest form, it adds a sequence of values:

```
template<typename In, typename T>
    // requires Input_iterator<T>() && Number<T>()
T accumulate(In first, In last, T init)
{
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Given an initial value, **init**, it simply adds every value in the **[first:last)** sequence to it and returns the sum. The variable in which the sum is computed, **init**, is often referred to as the *accumulator*. For example:

```
int a[] = { 1, 2, 3, 4, 5 };
cout << accumulate(a, a+sizeof(a)/sizeof(int), 0);
```

This will print **15**, that is, $0+1+2+3+4+5$ (**0** is the initial value). Obviously, **accumulate()** can be used for all kinds of sequences:

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0);
    int sum2 = accumulate(p,p+n,0);
}
```

The type of the result (the sum) is the type of the variable that **accumulate()** uses to hold the accumulator. This gives a degree of flexibility that can be important. For example:

```
void g(int* p, int n)
{
    int s1 = accumulate(p, p+n, 0);           // sum into an int
    long sl = accumulate(p, p+n, long{0});      // sum the ints into a long
    double s2 = accumulate(p, p+n, 0.0);       // sum the ints into a double
}
```

A **long** has more significant digits than an **int** on some computers. A **double** can represent larger (and smaller) numbers than an **int**, but possibly with less precision. We'll revisit the role of range and precision in numerical computations in Chapter 24.



Using the variable in which you want the result as the initializer is a popular idiom for specifying the type of the accumulator:

```
void f(vector<double>& vd, int* p, int n)
{
    double s1 = 0;
    s1 = accumulate(vd.begin(), vd.end(), s1);
    int s2 = accumulate(vd.begin(), vd.end(), s2);      // oops
    float s3 = 0;
    accumulate(vd.begin(), vd.end(), s3);                // oops
}
```



Do remember to initialize the accumulator and to assign the result of `accumulate()` to the variable. In this example, `s2` was used as an initializer before it was itself initialized; the result is therefore undefined. We passed `s3` to `accumulate()` (pass-by-value; see §8.5.3), but the result is never assigned anywhere; that compilation is just a waste of time.

21.5.2 Generalizing `accumulate()`

So, the basic three-argument `accumulate()` adds. However, there are many other useful operations, such as multiply and subtract, that we might like to do on a sequence, so the STL offers a second four-argument version of `accumulate()` where we can specify the operation to be used:

```
template<typename In, typename T, typename BinOp>
// requires Input_iterator<In>() && Number<T>()
//     && Binary_operator<BinOp, Value_type<In>, T>()
T accumulate(In first, In last, T init, BinOp op)
{
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

Any binary operation that accepts two arguments of the accumulator's type can be used here. For example:

```
vector<double> a = { 1.1, 2.2, 3.3, 4.4 };
cout << accumulate(a.begin(), a.end(), 1.0, multiplies<double>());
```

This will print **35.1384**, that is, $1.0*1.1*2.2*3.3*4.4$ (**1.0** is the initial value). The binary operator supplied here, **multiplies<double>()**, is a standard library function object that multiplies; **multiplies<double>** multiplies **doubles**, **multiplies<int>** multiplies **ints**, etc. There are other binary function objects: **plus** (it adds), **minus** (it subtracts), **divides**, and **modulus** (it takes the remainder). They are all defined in **<functional>** (§B.6.2).

Note that for products of floating-point numbers, the obvious initial value is **1.0**.

As in the **sort()** example (§21.4.2), we are often interested in data within class objects, rather than just plain built-in types. For example, we might want to calculate the total cost of items given the unit prices and number of units:

```
struct Record {
    double unit_price;
    int units;           // number of units sold
    // ...
};
```

We can let **accumulate**'s operator extract the **units** from a **Record** element as well as multiplying it to the accumulator value:

```
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units; // calculate price and accumulate
}

void f(const vector<Record>& vr)
{
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // ...
}
```

We were “lazy” and used a function, rather than a function object, to calculate the price – just to show that we could do that also. We tend to prefer function objects

- If they need to store a value between calls, or
- If they are so short that inlining can make a difference (at most a handful of primitive operations)

In this example, we might have chosen a function object for the second reason.

TRY THIS


Define a `vector<Record>`, initialize it with four records of your choice, and compute their total price using the functions above.

21.5.3 Inner product

Take two vectors, multiply each pair of elements with the same subscript, and add all of those products. That's called the *inner product* of the two vectors and is a most useful operation in many areas (e.g., physics and linear algebra; see §24.6). If you prefer code to words, here is the STL version:

```
template<typename In, typename In2, typename T>
    // requires Input_iterator<In> && Input_iterator<In2>
    //     && Number<T> (§19.3.3)
T inner_product(In first, In last, In2 first2, T init)
    // note: this is the way we multiply two vectors (yielding a scalar)
{
    while(first!=last) {
        init = init + (*first) * (*first2);      // multiply pairs of elements
        ++first;
        ++first2;
    }
    return init;
}
```

This generalizes the notion of inner product to any kind of sequence of any type of element. As an example, consider a stock market index. The way that works is to take a set of companies and assign each a “weight.” For example, in the Dow Jones Industrial index Alcoa had a weight of 2.4808 when last we looked. To get the current value of the index, we multiply each company's share price with its weight and add all the resulting weighted prices together. Obviously, that's the inner product of the prices and the weights. For example:

```
// calculate the Dow Jones Industrial index:
vector<double> dow_price = {           // share price for each company
    81.86, 34.69, 54.45,
    // ...
};
```

```

list<double> dow_weight = {           // weight in index for each company
    5.8549, 2.4808, 3.8940,
    // ...
};

double dji_index = inner_product( // multiply (weight,value) pairs and add
    dow_price.begin(), dow_price.end(),
    dow_weight.begin(),
    0.0);

cout << "DJI value " << dji_index << '\n';

```

Note that **inner_product()** takes two sequences. However, it takes only three arguments: only the beginning of the second sequence is mentioned. The second sequence is supposed to have at least as many elements as the first. If not, we have a run-time error. As far as **inner_product()** is concerned, it is OK for the second sequence to have more elements than the first; those “surplus elements” will simply not be used.

The two sequences need not be of the same type, nor do they need to have the same element types. To illustrate this point, we used a **vector** to hold the prices and a **list** to hold the weights.

21.5.4 Generalizing **inner_product()**

The **inner_product()** can be generalized just as **accumulate()** was. For **inner_product()** we need two extra arguments, though: one to combine the accumulator with the new value, exactly as for **accumulate()**, and one for combining the element value pairs:

```

template<typename In, typename In2, typename T, typename BinOp,
         typename BinOp2>
    // requires Input_iterator<In> && Input_iterator<In2> && Number<T>
    //   && Binary_operation<BinOp,T,Value_type<In>()
    //   && Binary_operation<BinOp2,T,Value_type<In2>()
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{  

    while(first!=last) {  

        init = op(init, op2(*first, *first2));  

        ++first;  

        ++first2;  

    }  

    return init;  

}

```

In §21.6.3, we return to the Dow Jones example and use this generalized [inner_product\(\)](#) as part of a more elegant solution.

21.6 Associative containers

After [vector](#), the most useful standard library container is probably the [map](#). A [map](#) is an ordered sequence of (key,value) pairs in which you can look up a value based on a key; for example, `my_phone_book["Nicholas"]` could be the phone number of Nicholas. The only potential competitor to [map](#) in a popularity contest is [unordered_map](#) (see §21.6.4), and that's a [map](#) optimized for keys that are strings. Data structures similar to [map](#) and [unordered_map](#) are known under many names, such as *associative arrays*, *hash tables*, and *red-black trees*. Popular and useful concepts always seem to have many names. In the standard library, we collectively call all such data structures *associative containers*.

The standard library provides eight associative containers:

Associative containers	
map	an ordered container of (key,value) pairs
set	an ordered container of keys
unordered_map	an unordered container of (key,value) pairs
unordered_set	an unordered container of keys
multimap	a map where a key can occur multiple times
multiset	a set where a key can occur multiple times
unordered_multimap	an unordered_map where a key can occur multiple times
unordered_multiset	an unordered_set where a key can occur multiple times

These containers are found in `<map>`, `<set>`, `<unordered_map>`, and `<unordered_set>`.

21.6.1 map

Consider a conceptually simple task: make a list of the number of occurrences of words in a text. The obvious way of doing this is to keep a list of words we have seen together with the number of times we have seen each. When we read a new word, we see if we have already seen it; if we have, we increase its count by one; if not, we insert it in our list and give it the value 1. We could do that using a [list](#) or a [vector](#), but then we would have to do a search for each word we read. That could be slow. A [map](#) stores its keys in a way that makes it easy to see if a key is present, thus making the searching part of our task trivial:

```

int main()
{
    map<string,int> words; // keep (word,frequency) pairs

    for (string s; cin>>s; )
        ++words[s]; // note: words is subscripted by a string

    for (const auto& p : words)
        cout << p.first << ":" << p.second << '\n';
}

```

The really interesting part of the program is `++words[s]`. As we can see from the first line of `main()`, `words` is a `map` of `(string,int)` pairs; that is, `words` maps `strings` to `ints`. In other words, given a `string`, `words` can give us access to its corresponding `int`. So, when we subscript `words` with a `string` (holding a word read from our input), `words[s]` is a reference to the `int` corresponding to `s`. Let's look at a concrete example:

`words["sultan"]`

If we have not seen the string "`sultan`" before, "`sultan`" will be entered into `words` with the default value for an `int`, which is `0`. Now, `words` has an entry ("`sultan`",`0`). It follows that if we haven't seen "`sultan`" before, `++words["sultan"]` will associate the value `1` with the string "`sultan`". In detail: the `map` will discover that "`sultan`" wasn't found, insert a ("`sultan`",`0`) pair, and then `++` will increment that value, yielding `1`.

Now look again at the program: `++words[s]` takes every “word” we get from input and increases its value by one. The first time a new word is seen, it gets the value `1`. Now the meaning of the loop is clear:

```

for (string s; cin>>s; )
    ++words[s]; // note: words is subscripted by a string

```

This reads every (whitespace-separated) word on input and computes the number of occurrences for each. Now all we have to do is to produce the output. We can iterate through a `map`, just like any other STL container. The elements of a `map<string,int>` are of type `pair<string,int>`. The first member of a `pair` is called `first` and the second member `second`, so the output loop becomes

```

for (const auto& p : words)
    cout << p.first << ":" << p.second << '\n';

```

As a test, we can feed the opening statements of the first edition of *The C++ Programming Language* to our program:

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.

We get the output

```
C: 1
C++: 3
C,: 1
Except: 1
In: 1
a: 2
addition: 1
and: 1
by: 1
defining: 1
designed: 1
details,: 1
efficient: 1
enjoyable: 1
facilities: 2
flexible: 1
for: 3
general: 1
is: 2
language: 1
language.: 1
make: 1
minor: 1
more: 1
new: 1
of: 1
programmer.: 1
programming: 3
provided: 1
provides: 1
purpose: 1
```

```
serious: 1
superset: 1
the: 3
to: 2
types.: 1
```

If we don't like to distinguish between upper- and lowercase letters or would like to eliminate punctuation, we can do so: see exercise 13.

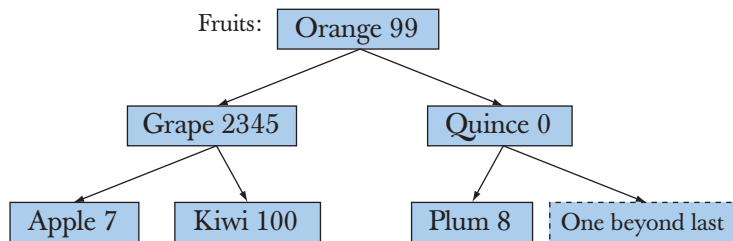
21.6.2 map overview

So what is a **map**? There is a variety of ways of implementing maps, but the STL **map** implementations tend to be balanced binary search trees; more specifically, they are red-black trees. We will not go into the details, but now you know the technical terms, so you can look them up in the literature or on the web, should you want to know more.

A tree is built up from nodes (in a way similar to a list being built from links; see §20.4). A **Node** holds a key, its corresponding value, and pointers to two descendant **Nodes**.

Map node:	Key first Value second Node* left Node* right ...
-----------	---

Here is the way a **map<Fruit,int>** might look in memory assuming we had inserted (Kiwi,100), (Quince,0), (Plum,8), (Apple,7), (Grape,2345), and (Orange,99) into it:



Given that the name of the **Node** member that holds the key value is **first**, the basic rule of a binary search tree is

left->first < first && first < right->first

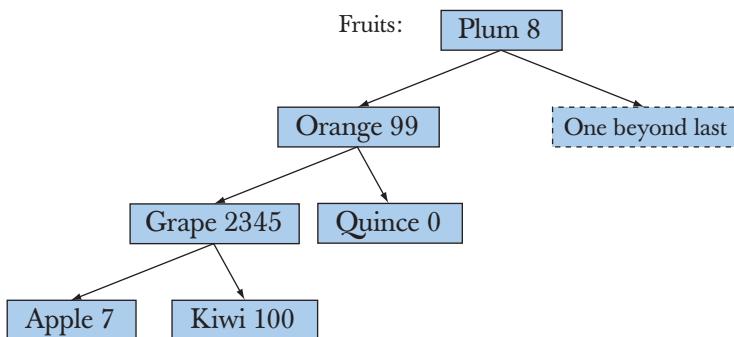
That is, for every node,

- Its left sub-node has a key that is less than the node's key, and
- The node's key is less than the key of its right sub-node



You can verify that this holds for each node in the tree. That allows us to search “down the tree from its root.” Curiously enough, in computer science literature trees grow downward from their roots. In the example, the root node is (Orange, 99). We just compare our way down the tree until we find what we are looking for or the place where it should have been. A tree is called *balanced* when (as in the example above) each sub-tree has approximately as many nodes as every other sub-tree that's equally far from the root. Being balanced minimizes the average number of nodes we have to visit to reach a node.

A **Node** may also hold some more data which the map will use to keep its tree of nodes balanced. A tree is balanced when each node has about as many descendants to its left as to its right. If a tree with N nodes is balanced, we have to at most look at $\log_2(N)$ nodes to find a node. That's much better than the average of $N/2$ nodes we would have to examine if we had the keys in a list and searched from the beginning (the worst case for such a linear search is N). (See also §21.6.4.) For example, have a look at an unbalanced tree:



This tree still meets the criteria that the key of every node is greater than that of its left sub-node and less than that of its right sub-node:

left->first<first && first<right->first

However, this version of the tree is unbalanced, so we now have three “hops” to reach Apple and Kiwi, rather than the two we had in the balanced tree. For trees of many nodes the difference can be very significant, so the trees used to implement **maps** are balanced.

We don't have to understand about trees to use `map`. It is just reasonable to assume that professionals understand at least the fundamentals of their tools. What we do have to understand is the interface to `map` provided by the standard library. Here is a slightly simplified version:

```
template<typename Key, typename Value, typename Cmp = less<Key>>
    // requires Binary_operation<Cmp, Value>() (§19.3.3)
class map {
    // ...
    using value_type = pair<Key,Value>; // a map deals in (Key,Value) pairs

    using iterator = sometype1;           // similar to a pointer to a tree node
    using const_iterator = sometype2;

    iterator begin();                   // points to first element
    iterator end();                    // points one beyond the last element

    Value& operator[](const Key& k);   // subscript with k

    iterator find(const Key& k);        // is there an entry for k?

    void erase(iterator p);             // remove element pointed to by p
    pair<iterator, bool> insert(const value_type&); // insert a (key,value) pair
    // ...
}
```

You can find the real version in `<map>`. You can imagine the iterator to be similar to a `Node*`, but you cannot rely on your implementation using that specific type to implement `iterator`.

The similarity to the interfaces for `vector` and `list` (§20.5 and §B.4) is obvious. The main difference is that when you iterate, the elements are pairs – of type `pair<Key,Value>`. That type is another useful STL type:

```
template<typename T1, typename T2>
struct pair {                                // simplified version of std::pair
    using first_type = T1;
    using second_type = T2;

    T1 first;
    T2 second;
```

```
// ...
};

template<typename T1, typename T2>
pair<T1,T2> make_pair(T1 x, T2 y)
{
    return {x,y};
}
```

We copied the complete definition of `pair` and its useful helper function `make_pair()` from the standard.

Note that when you iterate over a `map`, the elements will come in the order defined by the key. For example, if we iterated over the fruits in the example, we would get

`(Apple,7) (Grape,2345) (Kiwi,100) (Orange,99) (Plum,8) (Quince,0)`

The order in which we inserted those fruits doesn't matter.

The `insert()` operation has an odd return value, which we most often ignore in simple programs. It is a pair of an iterator to the (key,value) element and a `bool` which is `true` if the (key,value) pair was inserted by this call of `insert()`. If the key was already in the map, the insertion fails and the `bool` is `false`.

Note that you can define the meaning of the order used by a map by supplying a third argument (`Cmp` in the map declaration). For example:

`map<string, double, No_case> m;`

`No_case` defines case-insensitive compare; see §21.8. By default the order is defined by `less<Key>`, meaning “less than.”

21.6.3 Another map example

To better appreciate the utility of `map`, let's return to the Dow Jones example from §21.5.3. The code there was correct if and only if all weights appear in the same position in their `vector` as their corresponding name. That's implicit and could easily be the source of an obscure bug. There are many ways of attacking that problem, but one attractive one is to keep each weight together with its company's ticker symbol, e.g., (“AA”,2.4808). A “ticker symbol” is an abbreviation of a company name used where a terse representation is needed. Similarly we can keep the company's ticker symbol together with its share price, e.g., (“AA”,34.69). Finally,

for those of us who don't regularly deal with the U.S. stock market, we can keep the company's ticker symbol together with the company name, e.g., ("AA", "Alcoa Inc."); that is, we could keep three maps of corresponding values.

First we make the (symbol,price) map:

```
map<string,double> dow_price = { // Dow Jones Industrial index (symbol,price);
    // for up-to-date quotes see
    // www.djindexes.com
    {"MMM",81.86},
    {"AA",34.69},
    {"MO",54.45},
    // ...
};
```

The (symbol,weight) map:

```
map<string,double> dow_weight = { // Dow (symbol,weight)
    {"MMM", 5.8549},
    {"AA",2.4808},
    {"MO",3.8940},
    // ...
};
```

The (symbol,name) map:

```
map<string,string> dow_name = { // Dow (symbol,name)
    {"MMM","3M Co."},
    {"AA"} = "Alcoa Inc.",
    {"MO"} = "Altria Group Inc.",
    // ...
};
```

Given those maps, we can conveniently extract all kinds of information. For example:

```
double alcoa_price = dow_price ["AAA"];           // read values from a map
double boeing_price = dow_price ["BA"];

if (dow_price.find("INTC") != dow_price.end()) // find an entry in a map
    cout << "Intel is in the Dow\n";
```

Iterating through a map is easy. We just have to remember that the key is called **first** and the value is called **second**:

```
// write price for each company in the Dow index:
for (const auto& p : dow_price) {
    const string& symbol = p.first;           // the "ticker" symbol
    cout << symbol << '\t'
        << p.second << '\t'
        << dow_name[symbol] << '\n';
}
```

We can even do some computation directly using maps. In particular, we can calculate the index, just as we did in §21.5.3. We have to extract share values and weights from their respective maps and multiply them. We can easily write a function for doing that for any two **map<string,double>**s:

```
double weighted_value(
    const pair<string,double>& a,
    const pair<string,double>& b
        ) // extract values and multiply
{
    return a.second * b.second;
}
```

Now we just plug that function into the generalized version of **inner_product()** and we have the value of our index:

```
double dji_index =
    inner_product(dow_price.begin(), dow_price.end(), // all companies
                  dow_weight.begin(),           // their weights
                  0.0,                         // initial value
                  plus<double>(),
                  weighted_value);            // add (as usual)
                                    // extract values and weights
                                    // and multiply
```



Why might someone keep such data in **maps** rather than **vectors**? We used a **map** to make the association between the different values explicit. That's one common reason. Another is that a **map** keeps its elements in the order defined by its key. When we iterated through **dow** above, we output the symbols in alphabetical order; had we used a **vector** we would have had to sort. The most common reason to use a **map** is simply that we want to look up values based on the key. For large

sequences, finding something using `find()` is far slower than looking it up in a sorted structure, such as a `map`.

TRY THIS



Get this little example to work. Then add a few companies of your own choice, with weights of your choice.

21.6.4 `unordered_map`

To find an element in a `vector`, `find()` needs to examine all the elements from the beginning to the element with the right value or to the end. On average, the cost is proportional to the length of the `vector` (\mathcal{N}); we call that cost $O(\mathcal{N})$.

To find an element in a `map`, the subscript operator needs to examine all the elements of the tree from the root to the element with the right value or to a leaf. On average the cost is proportional to the depth of the tree. A balanced binary tree holding \mathcal{N} elements has a maximum depth of $\log_2(\mathcal{N})$; the cost is $O(\log_2(\mathcal{N}))$. $O(\log_2(\mathcal{N}))$ – that is, cost proportional to $\log_2(\mathcal{N})$ – is actually pretty good compared to $O(\mathcal{N})$:

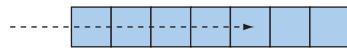
\mathcal{N}	15	128	1023	16,383
$\log_2(\mathcal{N})$	4	7	10	14

The actual cost will depend on how soon in our search we find our values and how expensive comparisons and iterations are. It is usually somewhat more expensive to chase pointers (as the `map` lookup does) than to increment a pointer (as `find()` does in a `vector`).

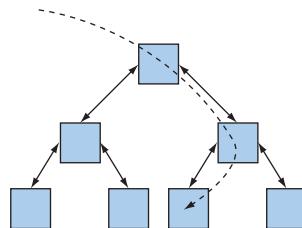
For some types, notably integers and character strings, we can do even better than a `map`'s tree search. We will not go into details, but the idea is that given a key, we compute an index into a `vector`. That index is called a *hash value* and a container that uses this technique is typically called a *hash table*. The number of possible keys is far larger than the number of slots in the hash table. For example, we often use a hash function to map from the billions of possible strings into an index for a `vector` with 1000 elements. This can be tricky, but it can be handled well and is especially useful for implementing large maps. The main virtue of a hash table is that on average the cost of a lookup is (near) constant and independent of the number of elements in the table, that is, $O(1)$. Obviously, that can be a significant advantage for large maps, say a map of 500,000 web addresses. For more information about hash lookup, you can look at the documentation for `unordered_map` (available on the web) or just about any basic text on data structures (look for “hash table” and “hashing”).

We can illustrate lookup in an (unsorted) vector, a balanced binary tree, and a hash table graphically like this:

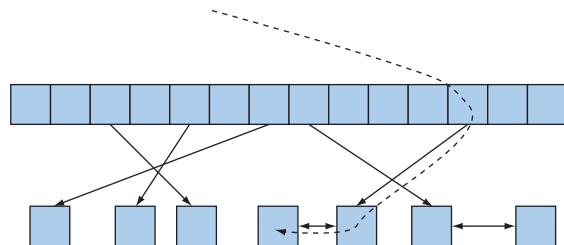
- Lookup in unsorted **vector**:



- Lookup in **map** (balanced binary tree):



- Lookup in **unordered_map** (hash table):



The STL **unordered_map** is implemented using a hash table, just as the STL **map** is implemented using a balanced binary tree, and an STL **vector** is implemented using an array. Part of the utility of the STL is to fit all of these ways of storing and accessing data into a common framework together with algorithms. The rule of thumb is:

- Use **vector** unless you have a good reason not to.
- Use **map** if you need to look up based on a value (and if your key type has a reasonable and efficient less-than operation).
- Use **unordered_map** if you need to do a lot of lookup in a large map and you don't need an ordered traversal (and if you can find a good hash function for your key type).

Here, we will not describe **unordered_map** in any detail. You can use an **unordered_map** with a key of type **string** or **int** exactly like a **map**, except that

when you iterate over the elements, the elements will not be ordered. For example, we could rewrite part of the Dow Jones example from §21.6.3 like this:

```
unordered_map<string,double> dow_price;

for (const auto& p : dow_price) {
    const string& symbol = p.first;           // the "ticker" symbol
    cout << symbol << '\t'
        << p.second << '\t'
        << dow_name[symbol] << '\n';
}
```

Lookup in `dow` might now be faster. However, that would not be significant because there are only 30 companies in that index. Had we been keeping the prices of all the companies on the New York Stock Exchange, we might have noticed a performance difference. We will, however, notice a logical difference: the output from the iteration will now not be in alphabetical order.

The unordered maps are new in the context of the C++ standard and not yet quite “first-class members,” as they are defined in a Technical Report rather than in the standard proper. They are widely available, though, and where they are not you can often find their ancestors, called something like `hash_map`.

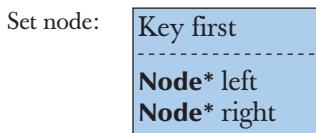
TRY THIS



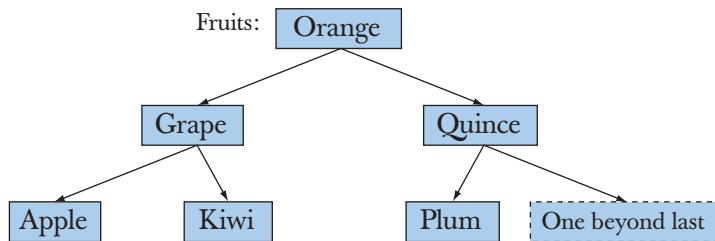
Write a small program using `#include<unordered_map>`. If that doesn’t work, `unordered_map` wasn’t shipped with your C++ implementation. If your C++ implementation doesn’t provide `unordered_map`, you have to download one of the available implementations (e.g., see www.boost.org).

21.6.5 set

We can think of a `set` as a `map` where we are not interested in the values, or rather as a `map` without values. We can visualize a `set` node like this:



We can represent the **set** of fruits used in the **map** example (§21.6.2) like this:



What are sets useful for? As it happens, there are lots of problems that require us to remember if we have seen a value. Keeping track of which fruits are available (independently of price) is one example; building a dictionary is another. A slightly different style of usage is having a set of “records”; that is, the elements are objects that potentially contain “lots of” information – we simply use a member as the key. For example:

```

struct Fruit {
    string name;
    int count;
    double unit_price;
    Date last_sale_date;
    // ...
};

struct Fruit_order {
    bool operator()(const Fruit& a, const Fruit& b) const
    {
        return a.name < b.name;
    }
};

set<Fruit, Fruit_order> inventory; // use Fruit_order(x,y) to compare Fruits
  
```

Here again, we see how using a function object can significantly increase the range of problems for which an STL component is useful.

 Since **set** doesn't have a value type, it doesn't support subscripting (**operator[]**) either. We must use “list operations,” such as **insert()** and **erase()**, instead. Unfortunately, **map** and **set** don't support **push_back()** either – the reason is obvious: the **set** and not the programmer determines where the new value is inserted. Instead use **insert()**. For example:

```
inventory.insert(Fruit("quince",5));
inventory.insert(Fruit("apple",200,0.37));
```

One advantage of `set` over `map` is that you can use the value obtained from an iterator directly. Since there is no (key,value) pair as for `map` (§21.6.3), the dereference operator gives a value of the element type:

```
for (auto p = inventory.begin(), p!=inventory.end(); ++p)
    cout << *p << '\n';
```

Assuming, of course, that you have defined `<<` for `Fruit`. Or we could equivalently write

```
for (const auto& x : inventory)
    cout << x << '\n';
```

21.7 Copying

In §21.2, we deemed `find()` “the simplest useful algorithm.” Naturally, that point can be argued. Many simple algorithms are useful – even some that are trivial to write. Why bother to write new code when you can use what others have written and debugged for you, however simple? When it comes to simplicity and utility, `copy()` gives `find()` a run for its money. The STL provides three versions of copy:

Copy operations	
<code>copy(b,e,b2)</code>	Copy $[b:e)$ to $[b2:b2+(e-b))$.
<code>unique_copy(b,e,b2)</code>	Copy $[b:e)$ to $[b2:b2+(e-b))$; suppress adjacent copies.
<code>copy_if(b,e,b2,p)</code>	Copy $[b:e)$ to $[b2:b2+(e-b))$, but only elements that meet the predicate <code>p</code> .

21.7.1 Copy

The basic copy algorithm is defined like this:

```
template<typename In, typename Out>
// requires Input_iterator<In>() && Output_iterator<Out>()
Out copy(In first, In last, Out res)
{
```

```

while (first!=last) {
    *res = *first;      // copy element
    ++res;
    ++first;
}
return res;
}

```

Given a pair of iterators, `copy()` copies a sequence into another sequence specified by an iterator to its first element. For example:

```

void f(vector<double>& vd, list<int>& li)
    // copy the elements of a list of ints into a vector of doubles
{
    if (vd.size() < li.size()) error("target container too small");
    copy(li.begin(), li.end(), vd.begin());
    // ...
}

```

Note that the type of the input sequence of `copy()` can be different from the type of the output sequence. That's a useful generality of STL algorithms: they work for all kinds of sequences without making unnecessary assumptions about their implementation. We remembered to check that there was enough space in the output sequence to hold the elements we put there. It's the programmer's job to check such sizes. STL algorithms are programmed for maximal generality and optimal performance; they do not (by default) do range checking or other potentially expensive tests to protect their users. At times, you'll wish they did, but when you want checking, you can add it as we did above.

21.7.2 Stream iterators

 You will have heard the phrases “copy to output” and “copy from input.” That’s a nice and useful way of thinking of some forms of I/O, and we can actually use `copy()` to do exactly that.

Remember that a sequence is something

- With a beginning and an end
- Where we can get to the next element using `++`
- Where we can get the value of the current element using `*`

We can easily represent input and output streams that way. For example:

```
ostream_iterator<string> oo{cout}; // assigning to *oo is to write to cout
*oo = "Hello, ";
++oo; // meaning cout << "Hello, "
*oo = "World!\n"; // get ready for next output operation"
// meaning cout << "World!\n"
```

You can imagine how this could be implemented. The standard library provides an **ostream_iterator** type that works like that; **ostream_iterator<T>** is an iterator that you can use to write values of type **T**.

Similarly, the standard library provides the type **istream_iterator<T>** for reading values of type **T**:

```
istream_iterator<string> ii{cin}; // reading *ii is to read a string from cin
string s1 = *ii; // meaning cin >> s1
++ii; // get ready for the next input operation"
string s2 = *ii; // meaning cin >> s2
```

Using **ostream_iterator** and **istream_iterator**, we can use **copy()** for our I/O. For example, we can make a “quick and dirty” dictionary like this:

```
int main()
{
    string from, to;
    cin >> from >> to; // get source and target file names

    ifstream is {from}; // open input stream
    ofstream os {to}; // open output stream

    istream_iterator<string> ii {is}; // make input iterator for stream
    istream_iterator<string> eos; // input sentinel
    ostream_iterator<string> oo {os, "\n"}; // make output iterator for stream

    vector<string> b {ii,eos}; // b is a vector initialized from input
    sort(b.begin(),b.end()); // sort the buffer
    copy(b.begin(),b.end(),oo); // copy buffer to output
}
```

The iterator **eos** is the stream iterator’s representation of “end of input.” When an **istream** reaches end of input (often referred to as **eof**), its **istream_iterator** will equal the default **istream_iterator** (here called **eos**).

Note that we initialized the `vector` by a pair of iterators. As the initializers for a container, a pair of iterators (`a,b`) means “Read the sequence `[a:b)` into the container.” Naturally, the pair of iterators that we used was `(ii, eos)` — the beginning and end of input. That saves us from explicitly using `>>` and `push_back()`. We strongly advise against the alternative



```
vector<string> b(max_size);      // don't guess about the amount of input!
copy(ii,eos,b.begin());
```

People who try to guess the maximum size of input usually find that they have underestimated, and serious problems emerge — for them or for their users — from the resulting buffer overflows. Such overflows are also a source of security problems.

TRY THIS



First get the program as written to work and test it with a small file of, say, a few hundred words. Then try the *emphatically not recommended* version that guesses about the size of input and see what happens when the input buffer `b` overflows. Note that the worst-case scenario is that the overflow led to nothing bad in your particular example, so that you would be tempted to ship it to users.

In our little program, we read in the words and then sorted them. That seemed an obvious way of doing things at the time, but why should we put words in “the wrong place” so that we later have to sort? Worse yet, we find that we store a word and print it as many times as it appears in the input.

We can solve the latter problem by using `unique_copy()` instead of `copy()`. A `unique_copy()` simply doesn’t copy repeated identical values. For example, using plain `copy()` the program will take

the man bit the dog

and produce

```
bit
dog
man
the
the
```

If we used **`unique_copy()`**, the program would write

```
bit
dog
man
the
```

Where did those newlines come from? Outputting with separators is so common that the **`ostream_iterator`**'s constructor allows you to (optionally) specify a string to be printed after each value:

```
ostream_iterator<string> oo {os, "\n"}; // make output iterator for stream
```

Obviously, a newline is a popular choice for output meant for humans to read, but maybe we prefer spaces as separators? We could write

```
ostream_iterator<string> oo {os, " "}; // make output iterator for stream
```

This would give us the output

```
bit dog man the
```

21.7.3 Using a set to keep order

There is an even easier way of getting that output; use a **`set`** rather than a **`vector`**:

```
int main()
{
    string from, to;
    cin >> from >> to; // get source and target file names

    ifstream is {from}; // make input stream
    ofstream os {to}; // make output stream

    set<string> b {istream_iterator<string>(is), istream_iterator<string>{}};
    copy(b.begin(), b.end(), ostream_iterator<string>(os, " ")); // copy buffer
                                                               // to output
}
```

When we insert values into a **`set`**, duplicates are ignored. Furthermore, the elements of a **`set`** are kept in order so no sorting is needed. With the right tools, most tasks are easy.

21.7.4 `copy_if`

The `copy()` algorithm copies unconditionally. The `unique_copy()` algorithm suppresses adjacent elements with the same value. The third copy algorithm copies only elements for which a predicate is true:

```
template<typename In, typename Out, typename Pred>
    // requires Input_iterator<In>() && Output_operator<Out>() &&
    // Predicate<Pred, Value_type<In>>()
Out copy_if(In first, In last, Out res, Pred p)
    // copy elements that fulfill the predicate
{
    while (first!=last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

Using our `Larger_than` function object from §21.4, we can find all elements of a sequence larger than 6 like this:

```
void f(const vector<int>& v)
    // copy all elements with a value larger than 6
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(), Larger_than(6));
    // ...
}
```



Thanks to a mistake I made, this algorithm is missing from the 1998 ISO standard. This mistake has now been remedied, but you can still find implementations without `copy_if`. If so, just use the definition from this section.

21.8 Sorting and searching



Often, we want our data ordered. We can achieve that either by using a data structure that maintains order, such as `map` and `set`, or by sorting. The most common and useful sort operation in the STL is the `sort()` that we have already used several times. By default, `sort()` uses `<` as the sorting criterion, but we can also supply our own criteria:

```
template<typename Ran>
    // requires Random_access_iterator<Ran>()
void sort(Ran first, Ran last);

template<typename Ran, typename Cmp>
    // requires Random_access_iterator<Ran>()
    // && Less_than_comparable<Cmp, Value_type<Ran>>()
void sort(Ran first, Ran last, Cmp cmp);
```

As an example of sorting based on a user-specified criterion, we'll show how to sort strings without taking case into account:

```
struct No_case {      // is lowercase(x) < lowercase(y)?
    bool operator()(const string& x, const string& y) const
    {
        for (int i = 0; i < x.length(); ++i) {
            if (i == y.length()) return false;      // y < x
            char xx = tolower(x[i]);
            char yy = tolower(y[i]);
            if (xx < yy) return true;           // x < y
            if (yy < xx) return false;         // y < x
        }
        if (x.length() == y.length()) return false;   // x == y
        return true;                         // x < y (fewer characters in x)
    }
};

void sort_and_print(vector<string>& vc)
{
    sort(vc.begin(), vc.end(), No_case());

    for (const auto& s : vc)
        cout << s << '\n';
}
```

Once a sequence is sorted, we no longer need to search from the beginning using **find()**; we can use the order to do a binary search. Basically, a binary search works like this:

Assume that we are looking for the value x ; look at the middle element:

- If the element's value equals x , we found it!
- If the element's value is less than x , any element with value x must be to the right, so we look at the right half (doing a binary search on that half).

- If the value of x is less than the element's value, any element with value x must be to the left, so we look at the left half (doing a binary search on that half).
- If we have reached the last element (going left or right) without finding x , then there is no element with that value.

 For longer sequences, a binary search is much faster than `find()` (which is a linear search). The standard library algorithms for binary search are `binary_search()` and `equal_range()`. What do we mean by “longer”? It depends, but ten elements are usually sufficient to give `binary_search()` an advantage over `find()`. For a sequence of 1000 elements, `binary_search()` will be something like 200 times faster than `find()` because its cost is $O(\log_2(N))$; see §21.6.4.

The `binary_search` algorithm comes in two variants:

```
template<typename Ran, typename T>
bool binary_search(Ran first, Ran last, const T& val);

template<typename Ran, typename T, typename Cmp>
bool binary_search(Ran first, Ran last, const T& val, Cmp cmp);
```

 These algorithms require and assume that their input sequence is sorted. If it isn't, “interesting things,” such as infinite loops, might happen. A `binary_search()` simply tells us whether a value is present:

```
void f(vector<string>& vs)      // vs is sorted
{
    if (binary_search(vs.begin(), vs.end(), "starfruit")) {
        // we have a starfruit
    }

    // ...
}
```

 So, `binary_search()` is ideal when all we care about is whether a value is in a sequence or not. If we care about the element we find, we can use `lower_bound()`, `upper_bound()`, or `equal_range()` (§B.5.4, §23.4). In the cases where we care which element is found, the reason is usually that it is an object containing more information than just the key, that there can be many elements with the same key, or that we want to know which element met a search criterion.

21.9 Container algorithms

So, we define standard library algorithms in terms of sequences of elements specified by iterators. An input sequence is defined as a pair of iterators **[b:e)** where **b** points to the first element of the sequence and **e** to the one-past-the-end element of the sequence (§20.3). An output sequence is specified as simply an iterator to its first element. For example:

```
void test(vector<int> & v)
{
    sort(v.begin(),v.end()); // sort v's element from v.begin() to v.end()
}
```

This is nice and general. For example, we can sort half a **vector**:

```
void test(vector<int> & v)
{
    sort(v.begin(),v.begin()+v.size()); // sort first half of v's elements
    sort(v.begin()+v.size(),v.end()); // sort second half of v's elements
}
```

However, specifying the range of elements is a bit verbose, and most of the time, we sort all of a **vector** and not just half. So, most of the time, we want to write

```
void test(vector<int> & v)
{
    sort(v); // sort v
}
```

That variant of **sort()** is not provided by the standard library, but we can define it for ourselves:

```
template<typename C> // requires Container<C>()
void sort(C& c)
{
    std::sort(c.begin(),c.end());
}
```

In fact, we found it so useful that we added it to **std_lib_facilities.h**.

Input sequences are easily handled like that, but to keep things simple, we tend to leave return types as iterators. For example:

```
template<typename C, typename V>      // requires Container<C>()
Iterator<C> find(C& c, Val v)
{
    return std::find(c.begin(),c.end(),v);
}
```

Naturally, `Iterator<C>` is `C`'s iterator type.



Drill

After each operation (as defined by a line of this drill) print the `vector`.

1. Define a `struct Item { string name; int iid; double value; /* ... */};`, make a `vector<Item>`, `vi`, and fill it with ten items from a file.
2. Sort `vi` by name.
3. Sort `vi` by `iid`.
4. Sort `vi` by value; print it in order of decreasing value (i.e., largest value first).
5. Insert `Item("horse shoe",99,12.34)` and `Item("Canon S400", 9988,499.95)`.
6. Remove (erase) two `Items` identified by `name` from `vi`.
7. Remove (erase) two `Items` identified by `iid` from `vi`.
8. Repeat the exercise with a `list<Item>` rather than a `vector<Item>`.

Now try a `map`:

1. Define a `map<string,int>` called `msi`.
2. Insert ten (name,value) pairs into it, e.g., `msi["lecture"]=21`.
3. Output the (name,value) pairs to `cout` in some format of your choice.
4. Erase the (name,value) pairs from `msi`.
5. Write a function that reads value pairs from `cin` and places them in `msi`.
6. Read ten pairs from input and enter them into `msi`.
7. Write the elements of `msi` to `cout`.
8. Output the sum of the (integer) values in `msi`.
9. Define a `map<int,string>` called `mis`.
10. Enter the values from `msi` into `mis`; that is, if `msi` has an element ("lecture",21), `mis` should have an element (21,"lecture").
11. Output the elements of `mis` to `cout`.

More **vector** use:

1. Read some floating-point values (at least 16 values) from a file into a **vector<double>** called **vd**.
2. Output **vd** to **cout**.
3. Make a vector **vi** of type **vector<int>** with the same number of elements as **vd**; copy the elements from **vd** into **vi**.
4. Output the pairs of (**vd[i], vi[i]**) to **cout**, one pair per line.
5. Output the sum of the elements of **vd**.
6. Output the difference between the sum of the elements of **vd** and the sum of the elements of **vi**.
7. There is a standard library algorithm called **reverse** that takes a sequence (pair of iterators) as arguments; reverse **vd**, and output **vd** to **cout**.
8. Compute the mean value of the elements in **vd**; output it.
9. Make a new **vector<double>** called **vd2** and copy all elements of **vd** with values lower than (less than) the mean into **vd2**.
10. Sort **vd**; output it again.

Review

1. What are examples of useful STL algorithms?
2. What does **find()** do? Give at least five examples.
3. What does **count_if()** do?
4. What does **sort(b,e)** use as its sorting criterion?
5. How does an STL algorithm take a container as an input argument?
6. How does an STL algorithm take a container as an output argument?
7. How does an STL algorithm usually indicate “not found” or “failure”?
8. What is a function object?
9. In which ways does a function object differ from a function?
10. What is a predicate?
11. What does **accumulate()** do?
12. What does **inner_product()** do?
13. What is an associative container? Give at least three examples.
14. Is **list** an associative container? Why not?
15. What is the basic ordering property of binary tree?
16. What (roughly) does it mean for a tree to be balanced?
17. How much space per element does a **map** take up?
18. How much space per element does a **vector** take up?
19. Why would anyone use an **unordered_map** when an (ordered) **map** is available?
20. How does a **set** differ from a **map**?

21. How does a **multimap** differ from a **map**?
22. Why use a **copy()** algorithm when we could “just write a simple loop”?
23. What is a binary search?

Terms

accumulate()	find_if()	searching
algorithm	function object	sequence
application: ()	generic	set
associative container	hash function	sort()
balanced tree	inner_product()	sorting
binary_search()	lambda	stream iterator
copy()	lower_bound()	unique_copy()
copy_if()	map	unordered_map
equal_range()	predicate	upper_bound()
find()		

Exercises

1. Go through the chapter and do all **Try this** exercises that you haven’t already done.
2. Find a reliable source of STL documentation and list every standard library algorithm.
3. Implement **count()** yourself. Test it.
4. Implement **count_if()** yourself. Test it.
5. What would we have to do if we couldn’t return **end()** to indicate “not found”? Redesign and re-implement **find()** and **count()** to take iterators to the first and last elements. Compare the results to the standard versions.
6. In the Fruit example in §21.6.5, we copy **Fruits** into the **set**. What if we didn’t want to copy the **Fruits**? We could have a **set<Fruit*>** instead. However, to do that, we’d have to define a comparison operation for that set. Implement the Fruit example using a **set<Fruit*, Fruit_comparison>**. Discuss the differences between the two implementations.
7. Write a binary search function for a **vector<int>** (without using the standard one). You can choose any interface you like. Test it. How confident are you that your binary search function is correct? Now write a binary search function for a **list<string>**. Test it. How much do the two binary search functions resemble each other? How much do you think they would have resembled each other if you had not known about the STL?
8. Take the word-frequency example from §21.6.1 and modify it to output its lines in order of frequency (rather than in lexicographical order). An example line would be **3: C++** rather than **C++: 3**.

9. Define an **Order** class with (customer) name, address, data, and **vector<Purchase>** members. **Purchase** is a class with a (product) **name**, **unit_price**, and **count** members. Define a mechanism for reading and writing **Orders** to and from a file. Define a mechanism for printing **Orders**. Create a file of at least ten **Orders**, read it into a **vector<Order>**, sort it by name (of customer), and write it back out to a file. Create another file of at least ten **Orders** of which about a third are the same as in the first file, read it into a **list<Order>**, sort it by address (of customer), and write it back out to a file. Merge the two files into a third using **std::merge()**.
10. Compute the total value of the orders in the two files from the previous exercise. The value of an individual **Purchase** is (of course) its **unit_price*count**.
11. Provide a GUI interface for entering **Orders** into files.
12. Provide a GUI interface for querying a file of **Orders**; e.g., “Find all orders from **Joe**,” “Find the total value of orders in file **Hardware**,” and “List all orders in file **Clothing**.” Hint: First design a non-GUI interface; then, build the GUI on top of that.
13. Write a program to “clean up” a text file for use in a word query program; that is, replace punctuation with whitespace, put words into lower case, replace *don't* with *do not* (etc.), and remove plurals (e.g., *ships* becomes *ship*). Don't be too ambitious. For example, it is hard to determine plurals in general, so just remove an *s* if you find both *ship* and *ships*. Use that program on a real-world text file with at least 5000 words (e.g., a research paper).
14. Write a program (using the output from the previous exercise) to answer questions such as: “How many occurrences of *ship* are there in a file?” “Which word occurs most frequently?” “Which is the longest word in the file?” “Which is the shortest?” “List all words starting with *s*.” “List all four-letter words.”
15. Provide a GUI for the program from the previous exercise.

Postscript

The STL is the part of the ISO C++ standard library concerned with containers and algorithms. As such it provides very general, flexible, and useful basic tools. It can save us a lot of work: reinventing the wheel can be fun, but it is rarely productive. Unless there are strong reasons not to, use the STL containers and basic algorithms. What is more, the STL is an example of generic programming, showing how concrete problems and concrete solutions can give rise to a collection of powerful and general tools. If you need to manipulate data – and most programmers do – the STL provides an example, a set of ideas, and an approach that often can help.



