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 Chapter 17) 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 (§8.4.3). We must insist that such variables are initialized. We could insist on an initializer and then write

```
string s2 = "";
vector<string> v2 {};
```

That's not particularly pretty and a bit verbose. However, "empty" is a reasonable and useful default for string and vector, so the standard provides it.

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.

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

```
class Date {
public:
        Date()                  // default constructor (takes no arguments)
                :y{Year{2001}}, m{Month::jan}, d{1}
        {
        }
        // ...
}
```

Now we can write:

```
Date d;         // d = {Year{2001},Month::jan,1}
```

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

```
class Date {
public:
        // ...
        Date() {}
        Date(Year y, Month m, int d);
        Date(Year y);                   // January 1 of year y
        // ...
        bool is_valid();
private:
        Year y {2001};
        Month m = Month::jan;
        int d = 1;
};
```

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

```
Date::Date(Year yy)                    // January 1 of year y
        :y{yy}
{
}
```

Because Date(int) does not explicitly initialize the month (m) or the day (d), the specified initializers (Month::jan and 1) are implicitly used. Because we didn't place any constraints on the value of Date::y, we don't need to call is_valid().

An initializer for a class member specified as part of the member declaration is called a *default member initializer* or an *in-class initializer*.

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

```
string{};                  // default value: the empty string ""
vector<string>{};          // default value: the empty vector;       no elements
```

However, in initializations, we prefer the colloquial

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

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. However, {} can be used to shorten initialization of variables:

```
void test()
{
        double x0;         // uninitialized; don't do that
        double x1 {0};     // initialize to 0
        double x2 = 0;     // initialize to 0
        double x3 {};      // initialize to 0
}
```

8.7.4 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 *constant variables* (sic!), *constants* or just consts. Consider:

```
        Date d;
        const Date start_of_term;

        int a = d.day();                       // OK
        int b = start_of_term.day();           // should be OK (why?), but isn't

        d.add_day(3);                          // OK
        start_of_term.add_day(3);              // error
```

Here, d is mutable, but start_of_term in a const. It is not acceptable to change the value of start_of_term. 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, just like add_day() does, and reports an error.

AA We deal with this problem by classifying operations on a class as modifying and nonmodifying. That's a fundamental distinction that helps us understand a class. It also has 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
        Year 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:
        Year y;
        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 deliberately try to "cheat" in this way. However, the compiler helps the class implementer by protecting against accidental violations.

8.7.5 Member functions and helper functions

AA 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 nonmember 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 member 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);
}
```

CC 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 §7.6:

```
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, 2028" is a valid date and "February 30, 2028" is not. There still may be aspects of a date that depend on the representation (e.g., can we represent "October 14, 1066"?), but (if necessary) Date's constructor can take care of that.

8.7.6 The ISO standard

Our Date isn't bad, but it is nowhere near as sophisticated as the facilities for handling time, dates, and time zones in the ISO C++ standard libraries. However, now you know most of the language facilities and design techniques used in the standard-library's chrono component (§20.4) where the equivalent to our Date is called year_month_date. Using that, you can write

```
auto birthday = December/16/1770;        // year_month_day{year{1770},December,day{16}}
```

Date is one of those types that it is useful to build as an exercise, but it is important to throw it away afterwards: The standard's version is not just better designed: importantly, it is also implemented by experts, extensively documented, extensively tested, and known by millions of programmers. Other such useful "exercise classes" with standard-library versions are string and vector. We visit more of the standard library in Chapter 15, Chapter 19, Chapter 20, and Chapter 21.

Drill

Write Day, Month, and their associated functions as described above. Complete the final version of Date with default constructor, is_valid(), Month, Year, etc. Define a Date called today initialized to February 2, 2020. 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 §9.6 and §9.7.

Your check for a valid date, is_valid(), 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 a 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).

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 struct Date from §8.4.1?
[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] What is a default constructor and when do we need one?
[8] What is a default member initializer?
[9] When should operator overloading be used in a program? Give a list of operators that you might want to overload (each with a reason). Which ones can you define in C++?
[10] Why should the public interface to a class be as small as possible?
[11] What does adding const to a member function do?
[12] Why are "helper functions" best placed outside the class definition?
[13] How does an enum class differ from a "plain" enum?

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	enum class
operator overloading	default member initializer		

Exercises

[1] List plausible operations for the examples of real-world objects in §8.1 (e.g., a 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] Do the previous exercise again but implement Name_pairs using a Name_pair class.
[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 to record when a book is checked out. Have it include a Book, a Patron, and a Date. Make a vector of Transactions to keep a record of which books are out. 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(int).

[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 (that is, an int that can hold much larger integers than plain int), and re-implement the Date member functions from §8.4.2. 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 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.

For a good example of an industrial-strength date library, see the standard-library **chrono**. For example, look it up in cppreference.com. It is part of the **std** module. Please don't look too hard. It uses a few advanced features that have yet to be presented.

Part II
Input and Output

Part II first describes how to get numeric and text data from the keyboard and from files, and how to produce corresponding output to the screen and to files. Then, we show how to present numeric data, text, and geometric shapes as graphical output, and how to get input into a program from a graphical user interface (GUI). As part of that, we introduce the fundamental principles and techniques of object-oriented programming.

9

Input and Output Streams

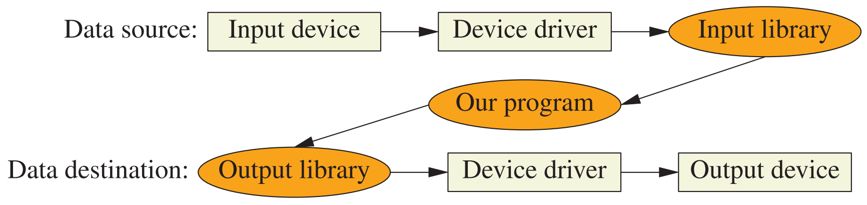
Science is what we have learned about
how to keep from fooling ourselves.
– Richard P. Feynman

In this chapter, 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.

9.1 Input and output

CC 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 §3.1, 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 heat sensor, 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:



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, we'll see how I/O consisting of streams of formatted data is done using the C++ standard library.

CC 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 Chapter 10 to Chapter 14.

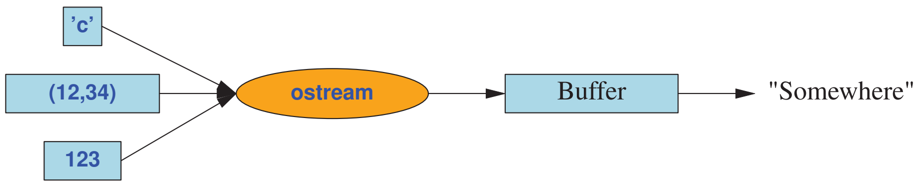
9.2 The I/O stream model

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` **CC**

- 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:



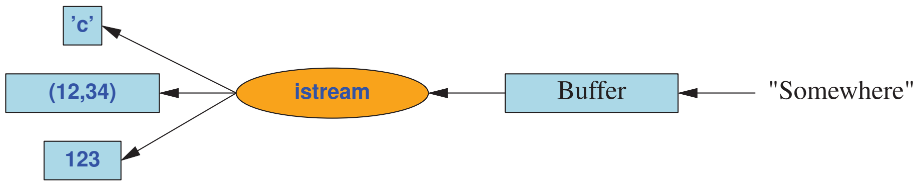
The buffer is a data structure that the `ostream` uses internally to store the data you give it while communicating with the operating system. If you notice a "delay" between your writing to an `ostream` and the characters appearing at their destination, it's usually because they are still in the buffer.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

An `istream` **CC**

[1] Gets characters from somewhere (such as a console, a file, the main memory, or another computer)

[2] Turns those character sequences into values of various types

We can represent an `istream` graphically like this:



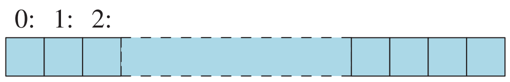
As with an `ostream`, an `istream` uses a buffer to communicate with the operating system. With an

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, **ostream**s 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, **istream**s provide features for reading the kind of output produced by **ostream**s. We’ll discuss formatting in §9.10. 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.

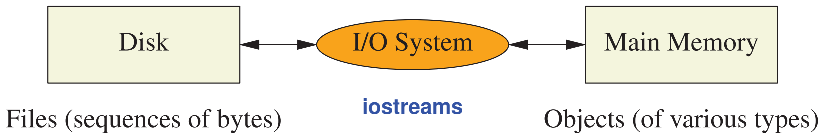
9.3 Files

CC 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. 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.

CC 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 some form of permanent storage (that is, on rotating-magnetic storage or solid-state storage). Again, that’s not always so (think files stored remotely), 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 in PPP2.§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 **ostream**s and all **istream**s.

9.3.1 Opening a file

AA

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);
```

AA

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 (Point 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 is ideal, and a file stream implicitly closes its file upon scope exit; see §15.5. 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
}
```

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

You can also use explicit open() and close() operations. For example:

```
ifstream ifs;
ifs.open(name,ios::in);                     // open file named name for reading
// ...
ifs.close();                                // close file
```

However, that's verbose. Also, relying on scope to control when the file is open minimizes the chances of someone trying to use it before it has been opened or after it was closed.

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, §18.4.2).

9.3.2 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 an (**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 (**ost**) 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 (Reading x : temps)
        ost << '(' << x.hour << ',' << x.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 **AA**

```
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 = −1;
        double temperature = −700;
        while (ist >> hour >> temperature) {
                if (hour < 0 || 23 <hour)
                        error("hour out of range");
                temps.push_back(Reading{hour,temperature});
        }

        for (int x : temps)
                ost << '(' << x.hour << ',' << x.temperature << ")\n";
}
```

9.4 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 (misunderstanding 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” aka "eof").
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).

CC 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.bad())                  // stream corrupted
        error("cin is bad");        // 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 Linux) 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 x; ist>>x; )
        v.push_back(x);
    if (ist.eof())                       // fine: we found the end of file
        return;
    if (ist.bad())                       // stream corrupted;
        error("ist is bad");             // 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 = 0;
        ist >> c;                        // read a character, hopefully terminator
        if (c != terminator) {           // unexpected character
            ist.unget();                 // put that character back
            ist.clear(ios::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::failbit). The ios that appears here and there is the part of an iostream that holds constants such as failbit and other useful stuff to control the behavior of the stream. You refer to them using the :: operator, for example, ios::badbit. The use of clear() with an argument is potentially confusing; clear() sets the iostream state flags (bits) mentioned and 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() (§5.8) 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:

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

The notation may seem odd, but the effect is simply that from that statement onward, ist will throw the standard-library exception ios::failure if it goes bad(). We need to execute that exceptions() call for an istream 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 x; ist>>x; )
        v.push_back(x);
    if (ist.eof())                  // fine: we found the end of file
        return;

    // if we get here ist is not good(), bad(), nor eof(), so ist must be fail()
    ist.clear();                    // clear stream state
    char c = 0;
    ist >> c;                       // read a character, hopefully terminator
    if (c != terminator) {          // ouch: not the terminator, so we must fail
        ist.unget();                // maybe the caller can use that character
        ist.clear(ios::failbit);// set the state to fail()
    }
}
```

Sometimes, we don't want to try recover locally from formatting errors. Then, we can ask for the stream to throw on fail() also:

```
ist.exceptions(ist.exceptions()|ios::badbit|ios::failbit);   // throw on any failure
```

If we want a really minimal fill_vector(), we can also eliminate the check for a terminator and get the simple and obvious loop:

```
void fill_vector(istream& ist, vector<int>& v)
        // read integers from ist into v until we reach eof()
{
        for (int x; ist>>x; )
                v.push_back(x);
}
```

For programs where output devices have a significant chance of being unavailable, filled, or broken, we can test after each output operation just as we test after each input operation. An **ostream** has exactly the same states as an **istream**: **good()**, **fail()**, **eof()**, and **bad()**. However, for output we can often simplify our code by having **bad()** and **fail()** throw exceptions.

9.5 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):

```
int get10()
{
        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)              // check range
                        return n;
                cout << "Sorry " << n << " is not in the [1:10] range; please try again\n";
        }
}
```

This is pretty ugly, but it "sort of works." 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 many 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. 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.

AA 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 the use of 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.

The deeper question is what to consider a valid input. For example, where should we reject "December 32, 202"? Typically, an input routine can handle only relatively simple, format related, problems. Problems related to the meaning of input must typically be handled at a higher-level of a program where the intended use of the data is known. For example, in most programs, year 202 would be a bad ("out of possible range") year, but in a program analyzing historical data it might be a perfectly good value. We will return to such questions repeatedly because validating input is a key part of writing a good program.

9.5.1 Breaking the problem into manageable parts

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

```
int get10()
{
        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)
                                return n;
                        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)			// we didn't find a digit: give up
					error("no input");
				cin.unget();			// put the digit back, so that we can read the number
			}
			else
				error("no input");		// bad: give up
		}
	}
```

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: XX

- 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: AA

```
	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:

```
	int get10()
	{
		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)
                                return n;
                        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();
                }
        }
}
```

This code is better, but it is still too long, too messy, and too special purpose (e.g., why [1:10]?).

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 could 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 **skip_to_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.

9.5.2 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 (§4.6.3); in general, we wouldn't know where to write.

9.6 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 the `Date` from §8.7.4 that simply prints the year, month, and day comma-separated in parentheses:

```
ostream& operator<<(ostream& os, const Date& d)
{
        return os << '(' << d.year()
                        << ',' << as_int(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 §8.6, we mention that a user-defined operator is handled by calling a function. Here, we have 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 (that is `cout`). 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.

9.7 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 §8.7.4 that will read dates as written by the output << 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::failbit);
                return is;
        }
        dd = Date{y,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::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>>().

9.8 A standard input loop

In §9.3.2, we saw how we could read and write files. However, that was before we looked more carefully at errors (§9.4), 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 §9.4, 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::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 many values. We could write a function to deal with that:

```
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 = 0;
                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.

9.9 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, XML, and JSON 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. AA

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.

9.9.1 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 many readings for that month.
- If we have any readings for a day, then we tend to have many 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> temp = vector<double>(24,not_a_reading);          // note: parentheses
};
```

That is, a Day has temperatures for 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 = vector<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 = vector<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.

XX 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 (§6.6.1). 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> temp {24,not_a_reading};            // note: curly brackets
};
```

That would have been simpler, but unfortunately, we would have gotten a vector of two elements (**24** and **–7777**). 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 (§7.2.2).

9.9.2 Reading structured values

The **Reading** class will be used only for reading input and is trivial:

```
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::failbit);
                return is;
        }

        char ch2;
        if ((is >> r.day >> r.hour >> r.temperature >> ch2) && ch2!=')')      // messed-up Reading?
                error("bad reading");
        return is;
}
```

Basically, we check if the format begins plausibly. 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::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 §9.8 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 = 0;
        is >> ch;
        if (ch!='{') {
                is.unget();
                is.clear(ios::failbit);
                return is;
        }

        string year_marker;
        int yy = –1;
        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::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.

9.9.3 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;                  // Months start at 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 = {
      "–not a month–",
      "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 [1:12] otherwise -1
{
      for (int i=1; i<13; ++i)
            if (month_input_tbl[i]==s)
                  return i;
      return 0;
}
```

In case you wonder: the C++ standard library does provide a simpler way to do this. See §20.2 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 = {
      "–not a month–",
      "January", "February", "March", "April", "May", "June", "July",
      "August", "September", "October", "November", "December"
};

string int_to_month(int i)
      // months [1:12]
{
      if (i<1 || 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.

9.10 Formatting

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 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 section presents a number of ways in which we can tailor input and output to our needs.

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 to make our output as clear and as close as possible to the expectations of the "consumers" of our program. Output streams (`ostream`s) 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.

CC 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 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 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 and flexibility is buffering: the standard-library `iostream`s rely on a concept called `streambuf`. For advanced work – whether for performance or functionality – with `iostream`s these `streambuf`s are unavoidable. If you feel the need to define your own `iostream`s or to tune `iostream`s to new data sources/sinks, look them up.

9.10.1 Integer I/O

Integer values can be output textually as binary (base-2), octal (base-8), decimal (our usual base-10 number system), and hexadecimal (base-16). 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 an int). When C++'s ancestor C was first designed (in the 1970s), octal was popular for representing bit patterns, but now it's rarely used.

For example:

```
int x = 1234;
cout << x << " – " << hex << x << " – " << oct << x << " – " << dec << x << '\n';
```

This prints

```
1234 – 4d2 – 2322 – 1234
```

The notation << hex does not output values. Instead, hex informs the stream that any further integer values should be displayed in hexadecimal. They are known as *manipulators* because they manipulate the state of a stream. We also have dec and oct 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.

By default, >> assumes that numbers use the decimal notation:

```
int a = 0;
int b = 0;
int c = 0;
cin >> a >> hex >> b >> oct >> c;
cout << dec;
cout << a << '\t' << b << '\t' << c << '\n';
cout << hex;
cout << a << '\t' << b << '\t' << c << '\n';
```

The '\t' character is called "tab" (short for "tabulation character" – a leftover from the days of mechanical typewriters).

If you type in

```
1234  4d2  2322
```

This prints:

```
1234 1234 1234
4d2  4d2  4d2
```

Note that the operators, such as hex, controlling formatting are "sticky"; that is, they persist until changed so that we can set them once for a stream and have their effect persist for many operations. We don't have to apply them repeatedly.

There is no standard **bin** manipulator to give us binary output. If we want binary output, we have to write one ourselves or use **format()** (§9.10.6).

9.10.2 Floating-point I/O

In scientific computation and many other fields, we 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:

```
constexpr double d = 1234.56789;

cout << "format: " << d << " – "            // use the default format for d
        << hexfloat << d << " – "           // use hexadecimal notation for d
        << scientific << d << " – "         // use 1.123e2 style format for d
        << fixed << d << " – "              // use 123.456 style format for d
        << defaultfloat << d << '\n';       // use the default format for d
```

This prints

```
format: 1234.57 – 0x1.34a4584f4c6e7p+10 – 1.234568e+03 – 1234.567890 – 1234.57
```

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
hexfloat	use scientific notation with hexadecimal for mantissa and exponent

A key property we often want to control is *precision*; that is how many digits are used when printing a floating-point number. 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."

Floating-point values are rounded rather than just truncated, and **precision()** doesn't affect integer output. For example:

```
cout << "precision: " << d << " – " << setprecision(8) << d << " – " << setprecision(16) << d << '\n';
```

This prints:

```
precision: 1234.57 – 1234.5679 – 1234.56789
```

When printing a lot of numbers, we often want them presented in neat rows and columns. This can be achieved by specifying the width of the field into which a value is written using setw(n). For example:

```
cout << "width: " << d << " – " << setw(8) << d <<  " – " << setw(16) << d << '\n';
```

This prints:

```
width: 1234.57 –  1234.57 –         1234.57
```

Note that setw applies just to its following number or string. There are more such controls. If you need them, look them up.

9.10.3 String I/O

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 whole lines. Consider:

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

What if we want to read everything on that line at once and decide how to format it later? That can 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.

9.10.4 Character I/O

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 (§5.8.2). There, we wanted 1+4*x<=y/z*5 to be separated into the eleven tokens CC

```
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)) {
                // do nothing; i.e., skip whitespace (e.g. space or tab)
        }
        else 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)?
`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	
`x=toupper(c)`	`x` becomes `c` or `c`'s uppercase equivalent
`x=tolower(c)`	`x` becomes `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 lowercase
{
        for (char& x : s)
                x = tolower(x);
}
```

We use pass-by-reference (§7.4.5) 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. **AA**

Possibly the most common reason to look at individual characters in a string or an input stream is to separate items; see §20.2.

9.10.5 Extend I/O

The stream I/O is extensible, so we can define << for our own (user-defined) types (§9.7, §9.6). Fortunately, the standard library defines << and >> for quite a few types. In addition to the basic numbers, strings, and characters, << can also handle time and dates: duration, time_point, year_month_date, weekday, month, and zoned_time (§20.4, §20.4.1). For example:

```
cout << "birthday: " << November/28/2021 << '\n';
cout << "zt: " << zoned_time{current_zone(), system_clock::now()} << '\n';
```

This produced:

```
birthday: 2021−11−28
zt: 2021−12−05 11:03:13.5945638 EST
```

The standard also defines << for complex numbers, bitsets (PPP2.§25.5.2), error codes, bools, and pointers (§15.3, §15.4).

9.10.6 format()

It has been credibly argued that printf() is the most popular function in C and a significant factor in C's success. For example:

```
printf("an int %g and a string '%s'\n", 123, "Hello!");
```

This ‘‘format string followed by arguments’’-style was adopted into C from BCPL and has been followed by many languages. Naturally, printf() has always been part of the C++ standard library, but it suffers from lack of type safety and lack of extensibility to handle user-defined types.

However, the standard library provides a type-safe and extensible printf()-style formatting mechanism. The function, format() produces a string:

```
string s = format("Hello, {}!\n", val);
```

‘‘Ordinary characters’’ in the *format string* are simply put into the output string. On the other hand, characters delimited by { and } specify how arguments following the format string are to be inserted into the output string. The simplest format string is the empty string, {}, that takes the next argument from the argument list and inserts it according to its << default (if any). So, if val is "World", we get the iconic "Hello, World!\n". If val is 127 we get "Hello, 127!\n".

The most common use of format() is to output its result:

```
cout << format("Hello, {}\n", val);
```

To see how this works, let's first repeat the examples from (§9.10.1):

```
int x = 1234;
cout << format("{} – {:x} – {:o} – {:d} – {:b}\n", x, x, x, x, x);
```

This gives the same output as the integer example in §9.10.1, except that I added b for binary which is not directly supported by ostream:

```
1234 – 4d2 – 2322 – 1234 – 10011010010
```

A formatting directive is preceded by a colon. The integer formatting alternatives are

- x: hexadecimal
- o: octal
- d: decimal
- b: binary
- none: d

The format string relies on a whole little programming language for specifying how a value is presented. Explaining all of that is beyond the scope of this book. If you need more see some more detailed source, such as https://en.cppreference.com/w/cpp/utility/format/formatter.

For floating-point numbers, the choices described in §9.10.2 are represented by

- a: hexfloat
- e: scientific
- f: fixed
- g: general, with precision 6
- none: general, with default precision

For example:

```
constexpr double d = 1234.56789;
cout << format("format: {} – {:a} – {:e} – {:f} – {:g}\n", d, d, d, d, d);
```

This prints

```
format: 1234.56789 – 1.34a4584f4c6e7p+10 – 1.234568e+03 – 1234.567890 – 1234.57
```

We can also specify how many character positions are used for a value. The "width example" from §9.10.2 can be written like this:

```
cout << format("width: {} – {:8} – {:20} –\n", d, d, d);
```

This prints:

```
width: 1234.56789 – 1234.56789 –           1234.56789 –
```

Precision is specified by a number after a dot:

```
cout << format("precision: {} – {:.8} – {:.20} –\n", d, d, d);
```

This prints:

```
precision: 1234.56789 – 1234.57 – 1234.567890000000034 –
```

You can combine formatting directives. For example:

```
cout << format("– {:12} – {:12.8f} – {:30.20e} –\n", d, d, d);
```

TRY THIS

See what that last statement prints, and explain it. Try some other formats.

9.11 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 values from a formatted **string**: CC

```
Point get_coordinates(const string& s)   // extract {x,y} from "(x,y)"
{
        istringstream is {s};      // make a stream so that we can read from s
        Point xy;
        char left_paren, ch, right_paren;
        is >> left_paren >> xy.x >> ch >> xy.y >> right_paren;
        if (!is || left_paren !='(' || ch!=',' || right_paren!=')')
                error("format error: ",s);
        return xy;
}

// testing:
auto c1 = get_coordinates("(2,3)");
auto c2 = get_coordinates("(   200, 300) ");
auto c3 = get_coordinates("100,400");          // 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**.

The **stringstreams** are generally used when we want to separate actual I/O from processing. For example, a **string** argument for **get_coordinates()** will usually originate from a file (e.g., a Web log), a GUI library, or a keyboard. Similarly, the message we composed in **my_code()** will eventually end up written to an area of a screen. AA

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
```

For logfiles, people often want fixed-length file names. In such cases, we can pad the names with the appropriate number of leading zeros:

```
name << "myfile" << setw(6) << setfill('0') << seq_no << ".log";      // e.g., myfile000017.log
```

Yes, external constraints can make I/O messy.

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.

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. Use **string**s to hold all values, even for the phone numbers. Experiment with different field widths until you are satisfied that the table is well presented.

[11] Defining a data type **Point** that has two coordinate members **x** and **y**. Define **<<** and **>>** for **Point** as discussed in §9.3.1.

[12] Using the code and discussion in §9.3.1, prompt the user to input seven (x,y) pairs. As the data is entered, store it in a **vector<Point>** called **original_points**.

[13] Print the data in **original_points** to see what it looks like.

[14] Open an **ofstream** and output each point to a file named **mydata.txt**. We suggest the **.txt** suffix to make it easier to look at the data with an ordinary text editor if you are using Windows.

[15] Open an **ifstream** for **mydata.txt**. Read the data from **mydata.txt** and store it in a new vector called **processed_points**.

[16] Print the data elements from both `vector`s.

[17] Compare the two `vector`s and print "Something's wrong!" if the number of elements or the values of elements differ.

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 default output format for floating-point values?

[7] Explain what `setprecision()` and `setw()` do.

[8] Which of the following manipulators do not "stick": `hex`, `scientific`, `setprecision()`, `setw()`?

[9] In `format()`, how do you specify where an argument is placed on output?

[10] Give two examples where a `stringstream` can be useful.

[11] When would you prefer line-oriented input to type-specific input?

[12] What does `isalnum(c)` do?

[13] When dealing with input and output, how is the variety of devices dealt with in most modern computers?

[14] What, fundamentally, does an `istream` do?

[15] What, fundamentally, does an `ostream` do?

[16] What, fundamentally, is a file?

[17] What is a file format?

[18] Name four different types of devices that can require I/O for a program.

[19] What are the four steps for reading a file?

[20] What are the four steps for writing a file?

[21] Name and define the four stream states.

[22] 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

[23] In what way is input usually harder than output?

[24] In what way is output usually harder than input?

[25] Why do we (often) want to separate input and output from computation?

[26] What are the two most common uses of the `istream` member function `clear()`?

[27] What are the usual function declarations for `<<` and `>>` for a user-defined type `X`?

[28] How do you specify where an argument is inserted into a format string in `format()`?

[29] What is the notation for bases of decimal values in `format()`?

[30] How do you specify the precision of floating-point values in `format()`?

Terms

binary	hexadecimal	octal	getline()
character classification	output formatting	decimal	line-oriented input
defaultfloat	manipulator	scientific	get()
setprecision()	fixed	<<	>>
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()	format()
tolower()	setw()	setfill()	isdigit()
isalpha()			

Exercises

[1] Write a program that reads a text file and converts its input to all lowercase, producing a new file.

[2] Write a program that given a file name and a word will output 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 prefixes; 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 §9.10.3. 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 lowercase.

[8] Use the program from the previous exercise to make a sorted list of words. 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 list.

[9] Write a function `vector<string> split(const string& s)` that returns a `vector` of whitespace-separated substrings from the argument `s`.

[10] 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`.

[11] 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.

[12] Reverse the order of words (defined as whitespace-separated strings) in a file. For example, `Norwegian Blue parrot` becomes `parrot Blue Norwegian`. Assume that all the strings from the file will fit into memory at once.

[13] Write a program that reads a text file and writes out how many characters of each character classification (§9.10.3) are in the file.

[14] 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.

[15] 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
```

[16] Write a program that produces the sum of all the numbers in a file of whitespace-separated integers.

[17] Write a program that creates a file of data in the form of the temperature `Reading` type defined in §9.3.2. 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.`

[18] 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`.

[19] 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.

[20] Write the function `print_year()` mentioned in §9.9.2.

[21] Define a `Roman_int` class for holding Roman numerals (as `int`s) 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';`.

[22] Make a version of the calculator from Chapter 6 that accepts Roman numerals rather than the usual Arabic ones, for example, XXI + CIV == CXXV.

[23] 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.

[24] Write a program that takes two files containing sorted whitespace-separated words and merges them, preserving order.

[25] Add a command **from x** to the calculator from Chapter 6 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 §6.3 and use that to test the calculator. Discuss how you would use these commands for testing.

[26] 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.

CC 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 must depart from their preferences, and when we are, we should typically be less arrogant than to think that we can provide a superior alternative to conventions built up over decades or centuries. 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.

10

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**, **Polygon**s, **Axis**, and **Text** are examples of **Shape**s. 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 11 focusing on their implementation and Chapter 12 on design issues.

10.1 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 13 gives simple (but general) facilities for graphing data. Also consider browsers, games, animation, scientific visualization, phones, and control displays.
- *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 11 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 PPP2.§22.2.4), but it was soon applied to that, and graphics provide some of the most accessible and tangible 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.

10.2 A display model

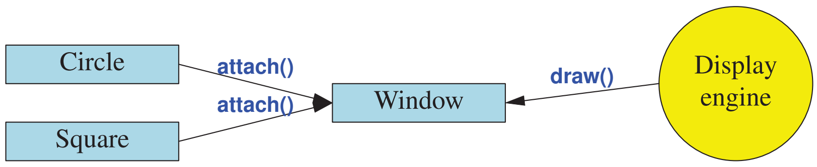
The iostream library is oriented toward reading and writing streams of characters as they might appear in a list of numeric values or a book. The only direct supports for the notion of graphical position are the newline and tab characters. You can embed notions of color and two-dimensional positions, etc. in a one-dimensional stream of characters. That's what layout (typesetting, "markup") languages such as Troff, TeX, Word, Markup, 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 EIddd, ...</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 (`<ul> ... </ul>`) with list items (`<li> ... </li>`), 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 sitting behind the screen," then takes the objects we have attached to our window and draw them on the screen: CC



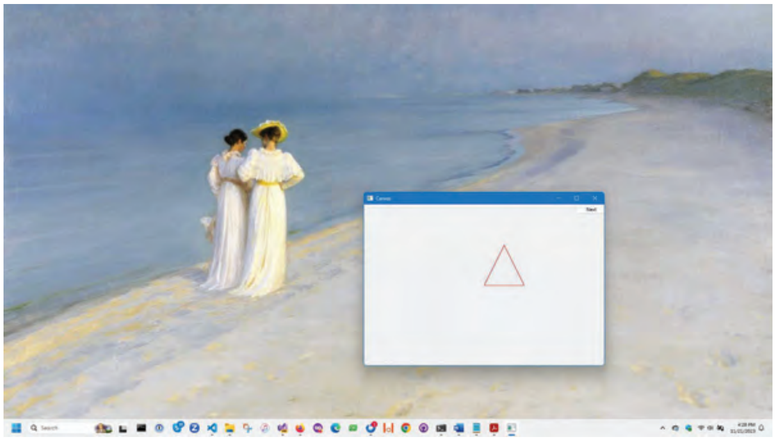
The "display engine" draws lines on the screen, places strings of text on the screen, colors areas of the screen, etc. For simplicity, we'll use the phrase "our GUI library" or even "the system" for the display engine even though our GUI library does much more than just drawing the objects. In the same 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.

10.3 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

        Application app;                            // start a Graphics/GUI application

        Point tl {900,500};                         // 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:



AA In the background of our window, we see a laptop screen (cleaned up for the occasion). For people who are curious about irrelevant details, we can tell you that my background is a famous painting by the Danish painter Peder Severin Krøyer. The ladies are Anna Ancher and Marie Krøyer, both well-known painters. If you look carefully, you'll notice that we have the Microsoft C++ compiler running, but we could just as well have used some other compiler (such as GCC or Clang). Let's go through the program line by line to see what was done.

First we **#include** our graphics interface library:

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

Why don't we use a module **Graph_lib** (§7.7.1)? One reason is at the time of writing not all implementations are up to using modules for this relatively complex task. For example, the system we use to implement our graphics library, Qt, exports its facilities using header files (§7.7.2). Another reason is that there is so much C++ code "out there" using header files (§7.7.2) that we need to show a realistic example somewhere.

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 start our display engine (§10.2):

```
Application app;                              // start a Graphics/GUI application
```

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

```
Point tl {900,500};                           // 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 called **Simple_window** to represent a window in our **Graph_lib** interface library . 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**:

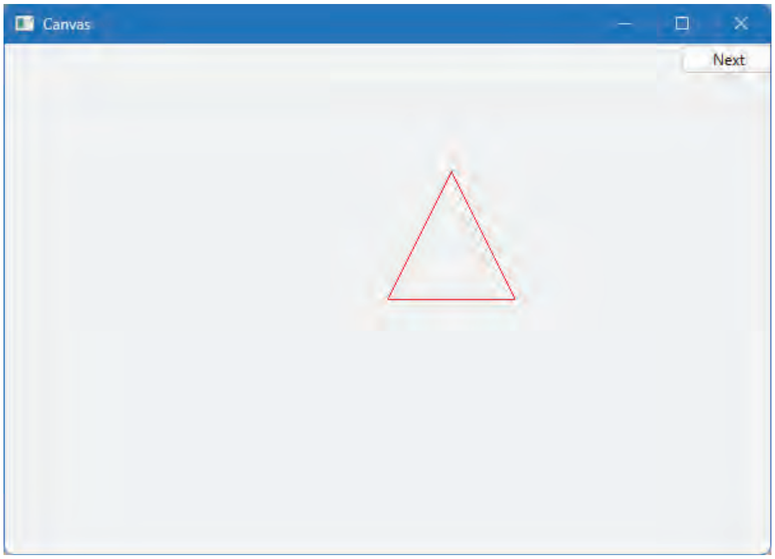
```
win.attach(poly);                        // connect poly to the window
```

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 in the top right corner 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.

For the rest of the Graphics-and-GUI chapters, we eliminate the distractions around our window and just show the window itself:



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 14, 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").

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**).

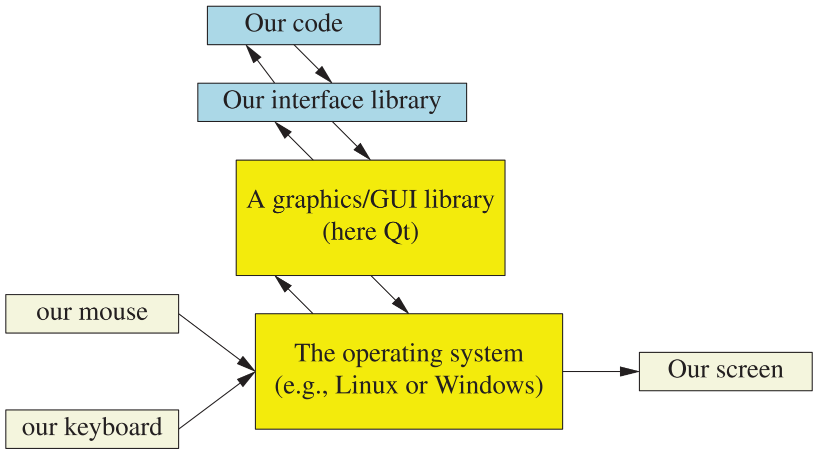
10.4 Using a GUI library

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. CC

The GUI toolkit that we are using (indirectly for now) is called Qt from **www.qt.io**. Our code is portable wherever Qt is available (Windows, Mac, Linux, many embedded systems, phones, browsers, 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 Qt documentation is thousands of pages. You can download Qt from **www.qt.io**, 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 Chapter 10 – Chapter 14 can be used with any popular GUI toolkit. We will of course explain how our interface classes map to Qt so that you will (eventually) see how you can use that (and similar toolkits) directly, if necessary.

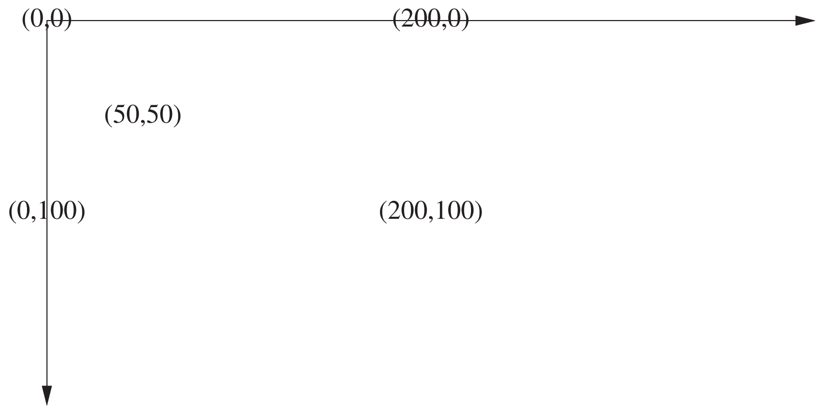
We can illustrate the parts of our "graphics world" like this: CC


Our code
Our interface library
A graphics/GUI library (here Qt)
our mouse
our keyboard
The operating system (e.g., Linux or Windows)
Our screen

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 14).

10.5 Coordinates

CC 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:



XX 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 and varies a lot (e.g., 600-by-1024, 1280-by-1024, 1920-by-1080, 2412-by-1080, and 2880-by-1920).

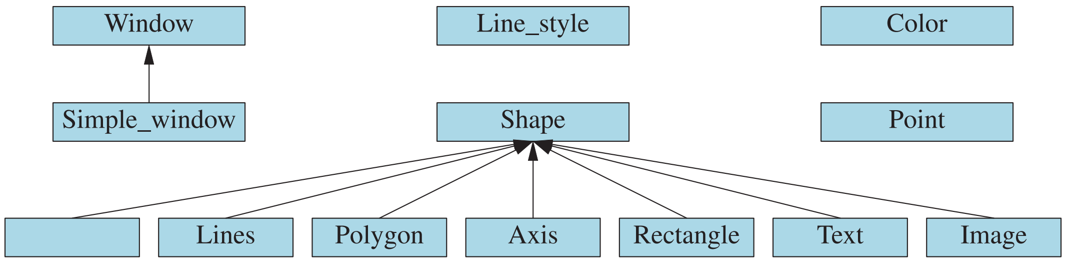
In the context of interacting with a computer using a screen, a window is a rectangular region of the screen devoted to some specific purpose and controlled by a program. A window is addressed exactly like a screen. Basically, we see a window as a small screen. For example, when we said

```
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.

10.6 Shapes

Our basic toolbox for drawing on the screen consists of about a dozen classes, including:



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**, **Circle**, **Ellipse**, etc.
- **Color**, **Line_style**, **Point**
- **Axis**

Later (Chapter 14), 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.

10.7 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.

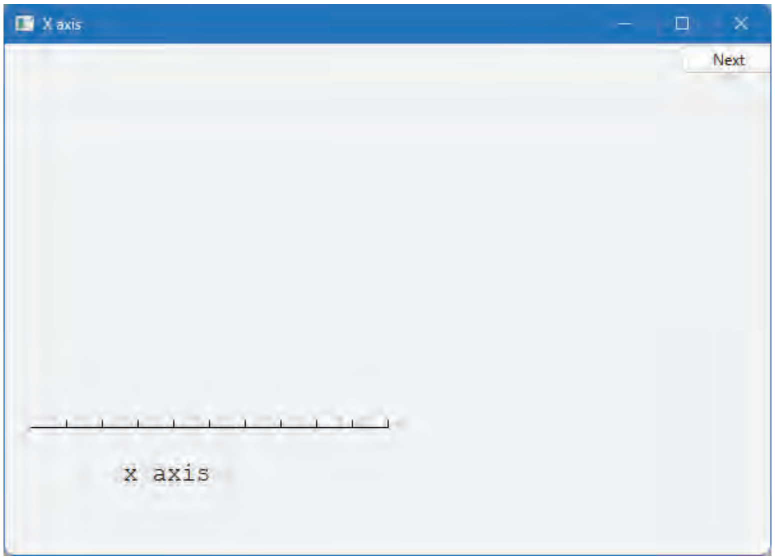
We will now walk through a simple program, explaining the code line by line and showing the effect of each on the screen. When you run the program, you'll see how the image changes as we add shapes to the window and modify existing shapes. Basically, we are "animating" the progress through the code by looking at the program as it is executed.

10.7.1 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
        // with 10 "notches"
        // label the axis "x axis"

win.attach(xa);                 // attach xa to the window, win
win.set_label("X axis");        // re-label 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** (§3.3.1, §13.6.3).

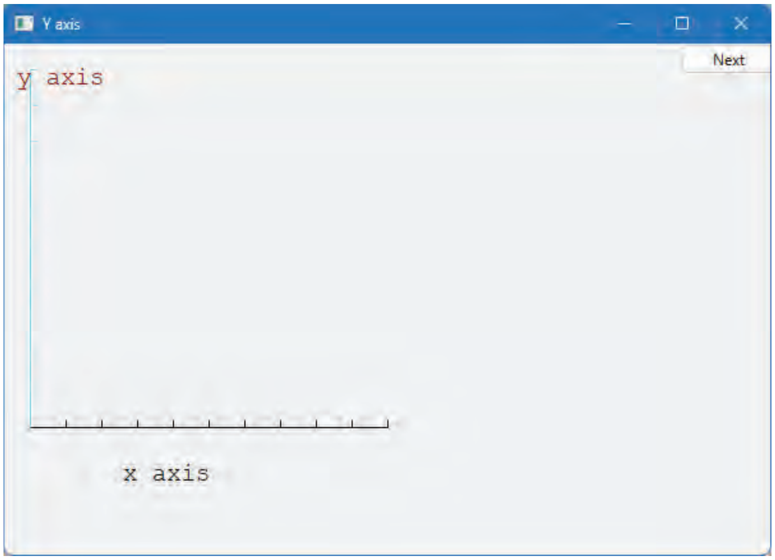
To help identify our output we relabeled the screen to **X axis** using **Window**'s member function **set_label()**.

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 for the y axis
ya.label.set_color(Color::dark_red);    // choose a color for the text
```

```
win.attach(ya);
win.set_label("Y axis");
win.wait_for_button();                    // display!
```

Just to show off some facilities, we colored our *y* axis cyan and our label dark red.



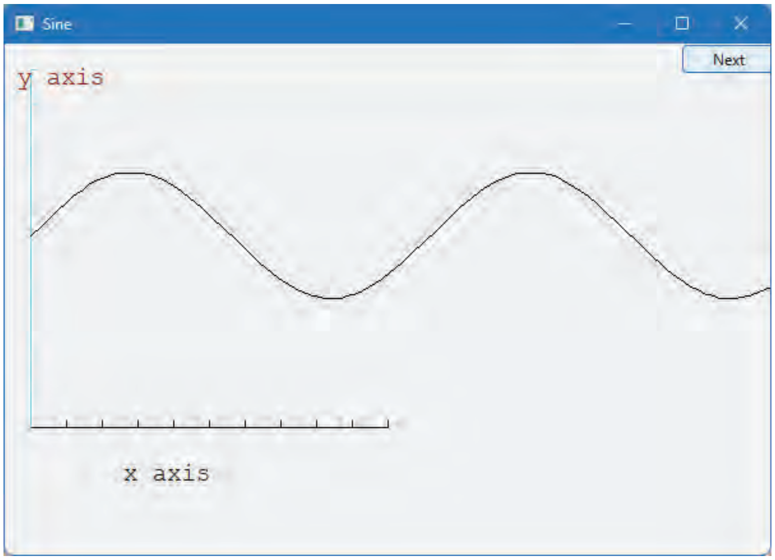
We don't actually think that it is a good idea to use different colors for *x* and *y* axes. We just wanted to show you how you can set the color of a shape and of individual elements of a shape. Using lots of color is not necessarily a good idea. In particular, novices often use color with more enthusiasm than taste.

10.7.2 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:

```
double dsin(double d) { return sin(d); }  // chose the right sin() (§13.3)

Function sine {dsin,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("Sine");
win.wait_for_button();
```

Here, the **Function** named **sine** will draw a sine curve using the standard-library function **sin(double)** to generate values. We explain details about how to graph functions in §13.3. 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.

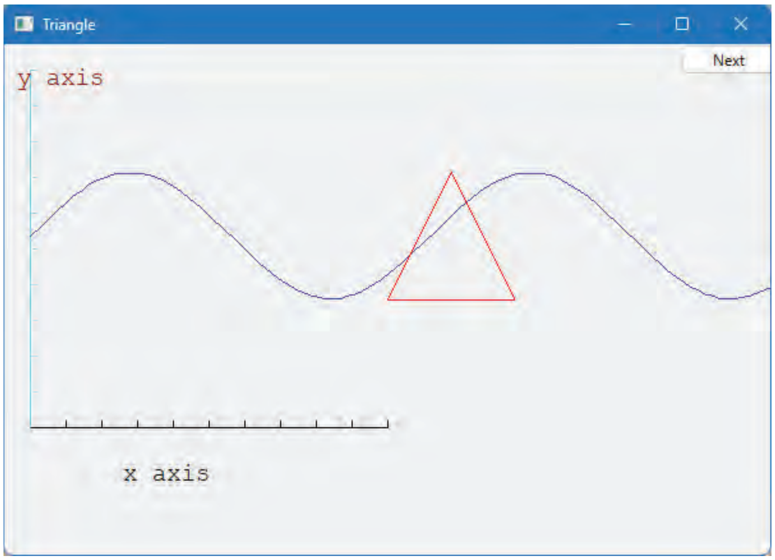
10.7.3 Polygons

A graphed function is an example of data presentation. We'll see much more of that in Chapter 11. 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);
```

```
win.attach(poly);
win.set_label("Triangle");
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 §10.3, 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 (§11.5). We get



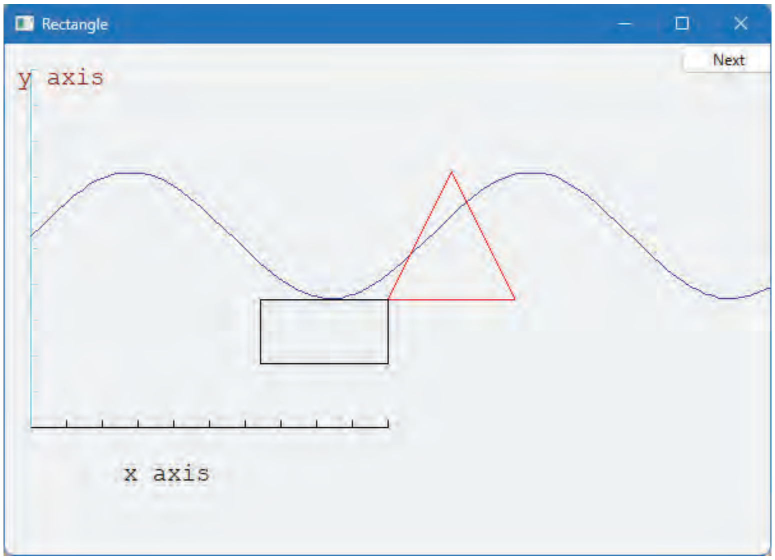
10.7.4 Rectangles

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. CC

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("Rectangle");
win.wait_for_button();
```

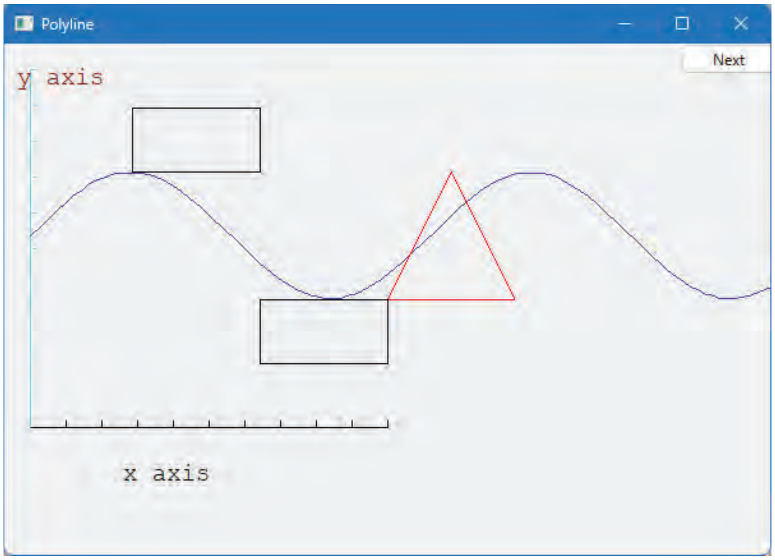
From that, we get



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.set_label("Polyline");
win.attach(poly_rect);
win.wait_for_button();
```

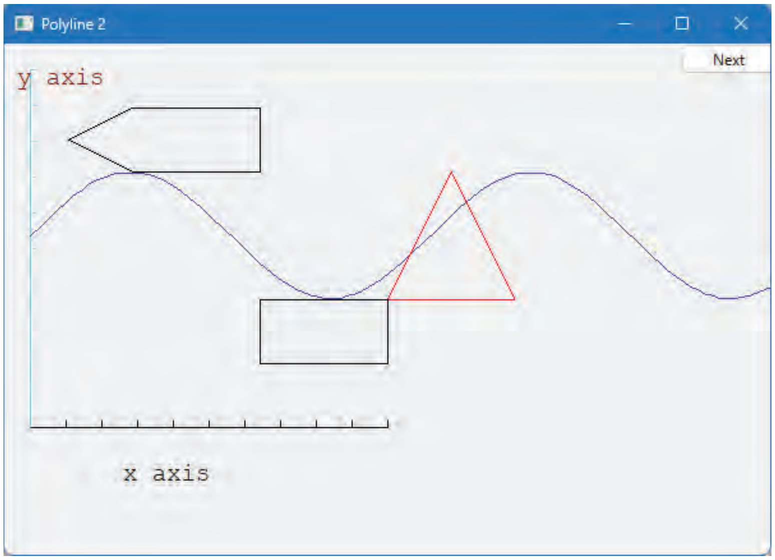
That polygon looks exactly – to the last pixel – like a rectangle:



However, it only looks like a **Rectangle**. No **Rectangle** has four points:

```
poly_rect.add(Point{50,75});
win.set_label("Polyline 2");
win.wait_for_button();
```

No rectangle has five points:



CC In fact, the *image* on the screen of the 4-point poly_rect *is* a rectangle. However, the poly_rect object in memory is not a Rectangle and it does not "know" anything about rectangles.

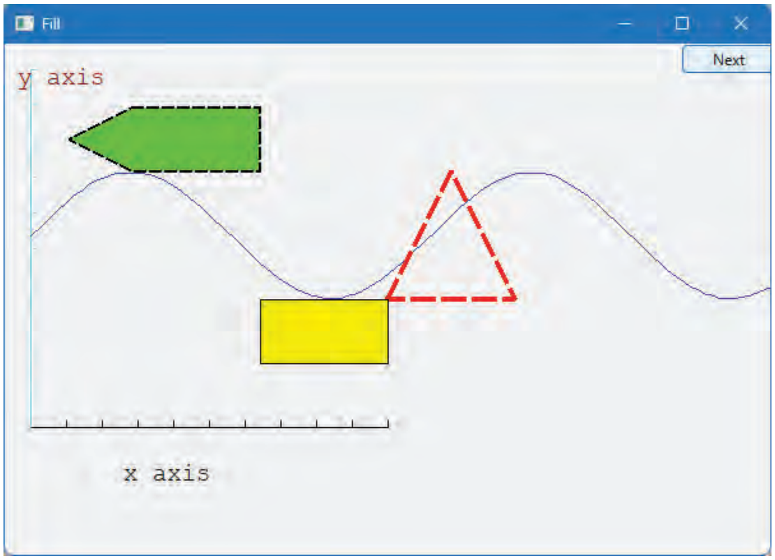
It is important for our reasoning about our code that a Rectangle doesn't just happen to look like a rectangle on the screen; it maintains the fundamental guarantees of a rectangle (as we know them from geometry). We write code that depends on a Rectangle really being a rectangle on the screen and staying that way.

10.7.5 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("Fill");
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) and filled it with green:



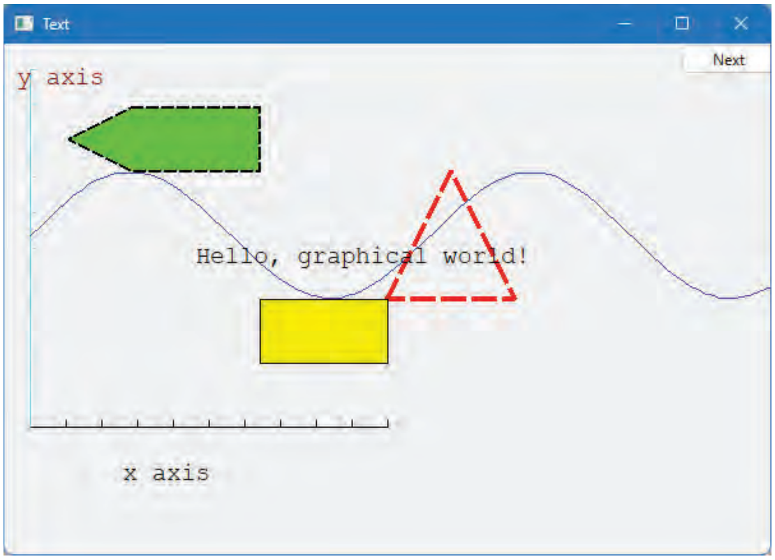
If you look carefully at poly_rect, you'll see that the outline is printed on top of the fill.

It is possible to fill any closed shape (§11.7, §11.7.2). Rectangles are just special in how easy (and fast) they are to fill.

10.7.6 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: CC

```
Text t {Point{150,150}, "Hello, graphical world!"};
win.attach(t);
win.set_label("Text");
win.wait_for_button();
```

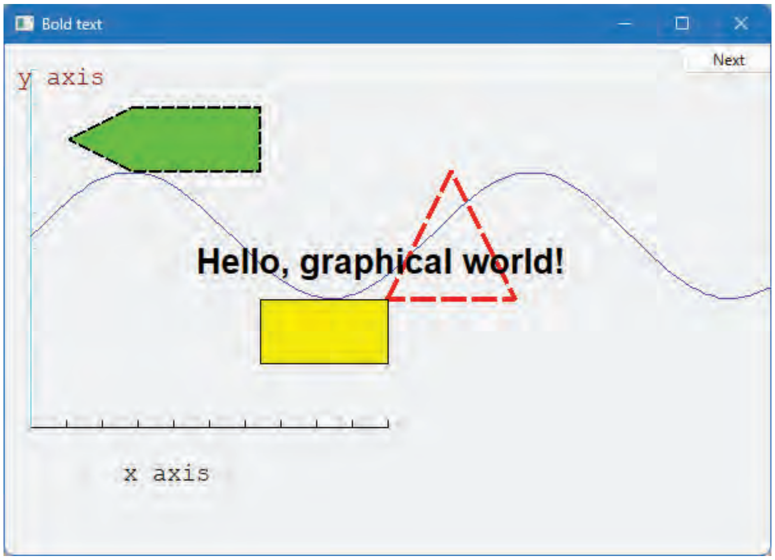


From the primitive graphics elements you see in this window, you can build displays of just about any complexity and subtlety. For now, just note a peculiarity of the code in this chapter: there are no loops, no selection statements, and all data was "hardwired" in. The output was just composed of primitives 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 (§10.7.1) is simply a Text object. In addition, we can choose a font and set the size of the characters:

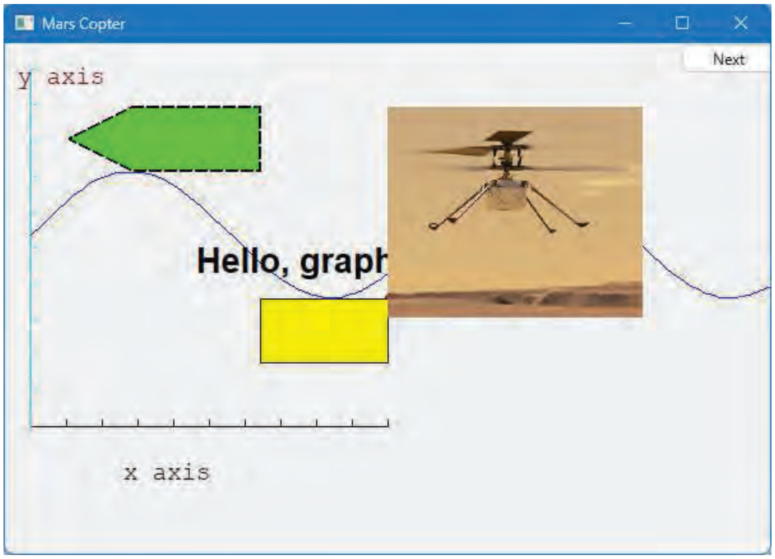
```
t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Bold text");
win.wait_for_button();
```

We enlarged the characters of the Text string Hello, graphical world! to point size 20 and chose the Times font in bold:



10.7.7 Images

We can also load images from files:

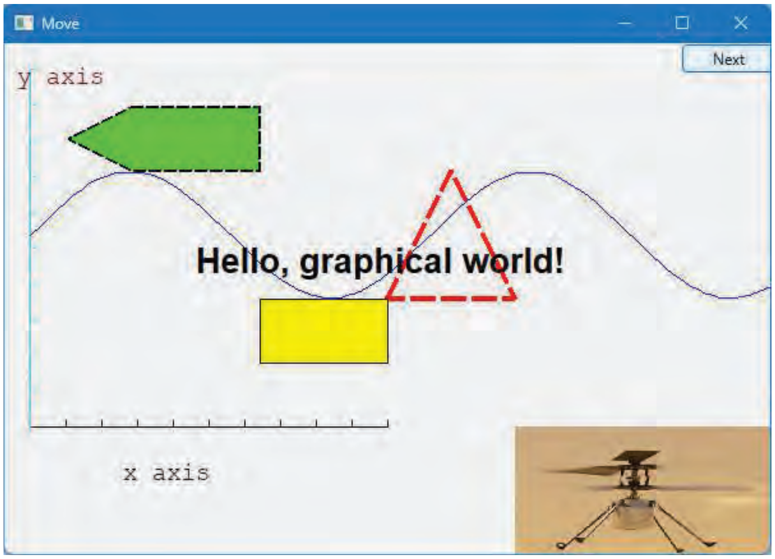


This was done by:

```
Image copter {Point{100,50},"mars_copter.jpg"};
win.attach(copter);
win.set_label("Mars copter");
win.wait_for_button();
```

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:

```
copter.move(100,250);
win.set_label("Move");
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.

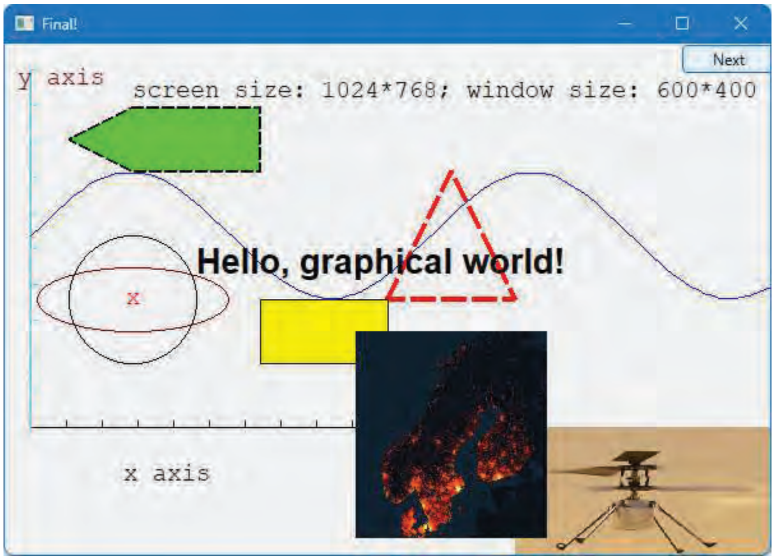
10.7.8 And much more

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'};
m.set_color(Color::red);
```

```
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 scan{ Point{275,225},"scandinavia.jfif" };
scan.scale(150,200);

win.attach(c);
win.attach(m);
win.attach(e);

win.attach(sizes);
win.attach(scan);
win.set_label("Final!");
win.wait_for_button();
```

Can you guess what this code does? Is it obvious?



AA 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** (§9.11) to format the text object displaying sizes. The string composed in **oss** is referred to as **oss.str()**.

10.8 Getting the first example 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 GUI system we use is installed and correctly linked to ours. Previous editions of the PPP code used the FLTK library; the current version uses the more modern Qt library. Both work over a wide range of systems.

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 Qt library
- The C++ standard library

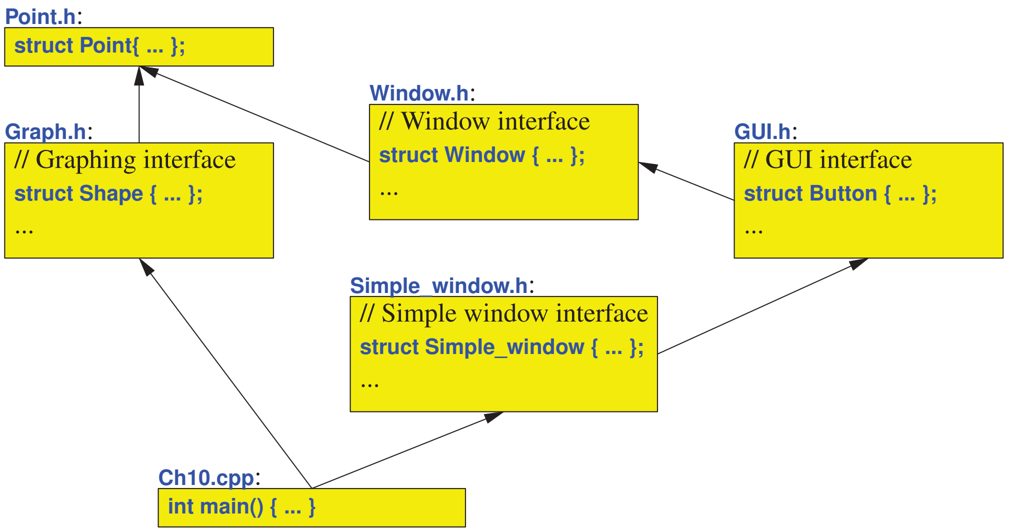
Indirectly, we also use the operating system.

10.8.1 Source files

Our graphics and GUI interface library consists of just five header files:

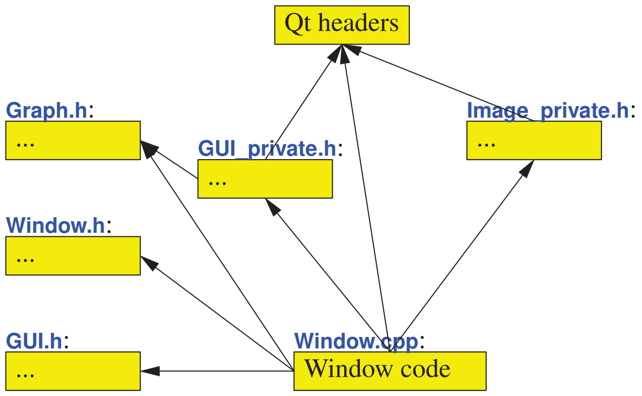
- Headers meant for users (aka "user-facing headers"):
  - **Point.h**
  - **Window.h**
  - **Simple_window.h**
  - **Graph.h**
  - **GUI.h**
- To implement the facilities offered by those headers, a few more files are used. Implementation headers:
  - Qt headers
  - **GUI_private.h**
  - **Image_private.h**
  - **Colormap.h**
- Code files:
  - **Window.cpp**
  - **Graph.cpp**
  - **GUI.cpp**
  - **GUI_private.cpp**
  - **Image_private.cpp**
  - **Colormap.cpp**
  - Qt code

We can represent the user-facing headers like this:



An arrow represents a **#include**. Until Chapter 14 you can ignore the GUI header.

A code file implementing a user-facing header **#include**s that header plus any headers needed for its code. For example, we can represent **Window.cpp** like this



In this way, we use files to separate what a user sees (the user-facing headers, such as **Window.h**) and what the implementation of such headers uses (e.g., Qt headers and **GUI_private.h**. In modules, that distinction is controlled by **export** specifiers (§7.7.1).

This "mess of files" is *tiny* compared to industrial systems, where many thousands of files are common, not uncommonly tens of thousands of files. That's one reason we prefer modules; they help organize code. Fortunately, we don't have to think about more than a few files at a time to get work done. This is what we have done here: the many files of the operating system, the C++ standard library, and Qt are invisible to us as users of our graphics interface library.

10.8.2 Putting it all together

Different systems (such as Windows, Mac, and Linux) have different ways of installing a library (such as Qt) and compiling and linking a program (such as ours). Worse, such set-up procedures change over time. Therefore, we place the instructions on the Web: www.stroustrup.com/programming.html and try to keep those descriptions up to date. When setting up your first project, be careful and be prepared for possible frustration. Setting up a relatively complex system like this can be very simple, but there are usually "things" that are not obvious to a novice. If you are part of a course, your teacher or teaching assistant can help, and might even have found an easier way to get you started. In any case, installing a new system or library is exactly where a more experienced person can be of significant help.

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 Qt library, `#include Graph.h` and `Simple_window.h` in your code, and compile and link `Graph.cpp` and `Window.cpp` into your program.

[2] Now add the examples from §10.7 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] What library do we use to implement our graphics/GUI interface library? Why don't we use the operating system directly?

Terms

color	graphic	JPEG	coordinates
GUI	line style	display	**PPP_graphics**
library	software layer	fill	**Shape**
color	HTML	window	Qt
image	XML	**Simple_window**	

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 source file diagram from §10.8.1.

[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>** and module **std** (PPP2.§24.8).

[12] A superellipse is a two-dimensional shape defined by the equation

$$\left|\frac{x}{a}\right|^m + \left|\frac{y}{b}\right|^n = 1;\ \text{where } m > 0 \text{ and } n > 0.$$

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

AA

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/GUI is just an illustration 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.

11 Graphics Classes

A language that doesn´t
change the way you think
isn´t worth learning.
– Traditional

Chapter 10 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.

11.1 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 expressions as needed. You can do so either by adding to the facilities described here or by adopting a full-scale C++ graphics/GUI library.

The key interface classes are:

Graphics interface classes (in Graph.h)	
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
Mark	a point marked by a character (such as x or o)
Line	a line as we see it on the screen, defined by its two end Points
Lines	a set of Lines defined by pairs of Points
Open_polyline	a sequence of connected Lines defined by a sequence of Points
Closed_polyline	like an Open_polyline, except that a Line connects the last Point to the first
Marks	a sequence of points indicated by marks (such as x and o)
Marked_polyline	an Open_polyline with its points indicated by marks
Polygon	a Closed_polyline where no two Lines intersect
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
Text	a string of characters
Image	the contents of an image file

Chapter 13 examines Function and Axis. Chapter 14 presents the main GUI interface classes:

Window interface classes	
Window	an area of the screen in which we display our graphics objects. In Window.h.
Simple_window	a window with a "Next" button. In Simple_window.h.
Application	the class that provides our interface to Qt

Every GUI/graphics program needs to start by defining an Application object.

GUI interface classes (in GUI.h)	
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

This Graphics/GUI library is presented as module PPP_graphics. At the time of writing not every C++ implementation has excellent module support, so we also make the library source code available as source files organized as described in §10.8.1.

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. In Graph.h.	
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, television, and phone 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 the code 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.

11.2 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 §10.5, *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).

A 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); }
```

Whatever appears in a Window is a Shape, which we describe in detail in Chapter 12. So a Line is a Shape that connects two Points with a line:

```
struct Line : Shape {                  // a Line is a Shape defined by two Points
      Line(Point p1, Point p2);        // construct a Line from two Points
};
```

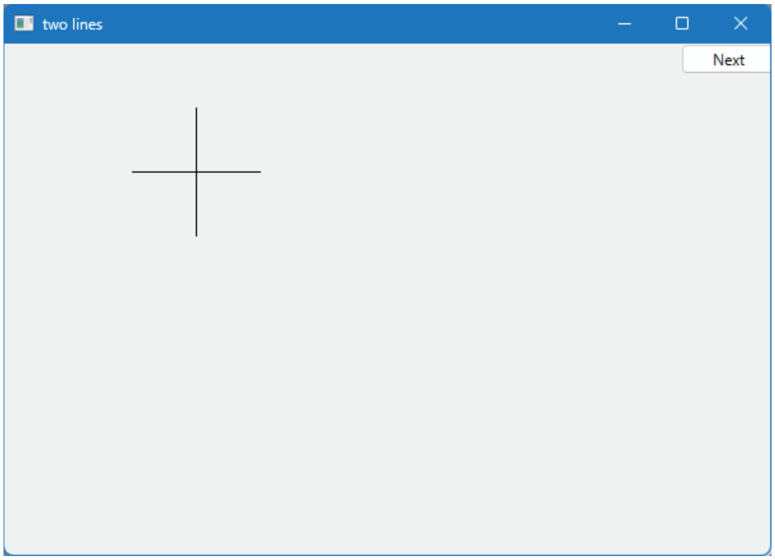
A Line is a kind of Shape. That's what : Shape means. Shape is called a *base class* for Line or simply a *base* of Line. Shape provides the facilities needed to make the definition of Line simple. Once we have a feel for the particular shapes, such as Line and Open_polyline, we'll explain what that implies (§12.2).

A Line is defined by two Points. We can create lines and cause them to be drawn like this:

```
#include "PPP.h"
#include "PPP/Simple_window.h"
#include "PPP/Graph.h"

using namespace Graph_lib;

int main()
      // draw two lines
{
      constexpr Point x {100,100};

      Simple_window win {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

      win.attach(horizontal);                                // attach the lines to the window
      win.attach(vertical);

      win.wait_for_button();                                 // display!
}
catch (...) {
      cout << "something went wrong\n";
}
```

As a reminder, we left in the "scaffolding" (#includes, etc., as described in §10.3).

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 12, 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 also leave out the definition of the Simple_window (§14.3) and the calls of attach() and set_label(). Those are just more "scaffolding" that we need for a complete program but that adds little to the discussion of specific Shapes.

11.3 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**:

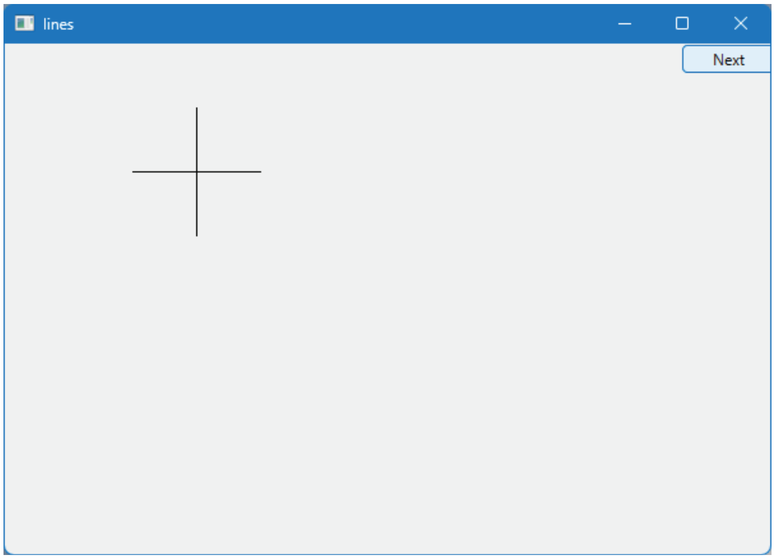
```
struct Lines : Shape {          // related lines
        Lines(initializer_list<Point> lst = {});               // by default, an empty list
        void draw_specific(Painter& painter) const override;
        void add(Point p1, Point p2);
};
```

The **override** means “use this **draw_specific** rather than **Shape**’s for **Lines**” (§12.3.3).

A **Lines** object is simply a **Shape** (§12.2) that consists of a collection of lines, each defined by a pair of **Point**s. For example, had we considered the two lines from the **Line** example in §11.2 as part of a single graphical object, we could have defined them like this:

```
Lines y;
y.add(Point{100,100}, Point{200,100});          // first line: horizontal
y.add(Point{150,50}, Point{150,150});           // second line: vertical
```

This gives output that is indistinguishable (to the last pixel) from the **Line** version:

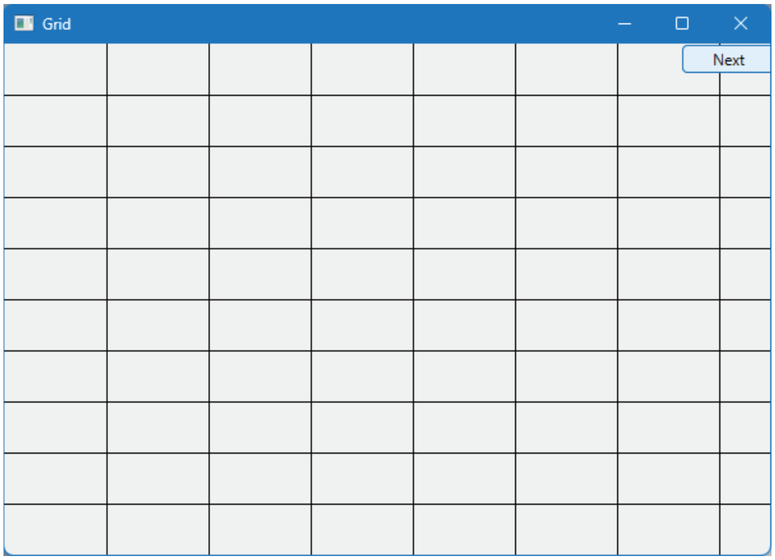


The only way we can tell that this is a different window is that we labeled them differently.

CC 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:

```
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 dimensions 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, and asks for the modified object to be redrawn in the **Window**:

```
void Lines::add(Point p1, Point p2)
{
	Shape::add(p1);
	Shape::add(p2);
	redraw();
}
```

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's add().

The draw_specific() function draws the lines defined using add():

```
void Lines::draw_specific(Painter& painter) const
{
	if (color().visibility())
		for (int i=1; i<number_of_points(); i+=2)
			painter.draw_line(point(i−1),point(i));
}
```

That is, Lines::draw_specific() 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 (Painter::draw_line()). A Painter is an object that holds the information about how an object is to be displayed on a screen. It defines the mapping from the concepts (such as Color and Line_style) in our interface library to the Qt versions of such concepts represented as a QPainter. QPainter is a complex and highly optimized class that we will not describe. Efficient quality rendering of information is a non-trivial art and here we will just use painters in simple ways, such as to draw a straight line.

For this example, we simply used the default color (black). Visibility is a property of the Lines' Color object (§11.4), so we have to check that the lines are meant to be visible before drawing them. 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 (§12.2) and have their obvious meaning.

As we explain in §12.2.3, draw_specific() is called by draw() that in turn is called "the system" when a Shape needs to appear. These two functions provide read-only access to a Shape's points. The member function draw_specific() is defined to be const (see §8.7.4) because it doesn't modify the shape.

11.3.1 Initialization

AA The Lines constructor takes an initializer_list of pairs of Points, each defining a line. Given that initializer-list constructor (§17.3), 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; just check that the number of points is even and let Shape's list constructor do the work:

```
void Lines::Lines(initializer_list<pair<Point,Point>> lst)
        : Shape{lst}
{
        if (lst.size() % 2)
                error("odd number of points for Lines");
}
```

The initializer_list type is defined in the standard library (§17.3, §20.2.2).

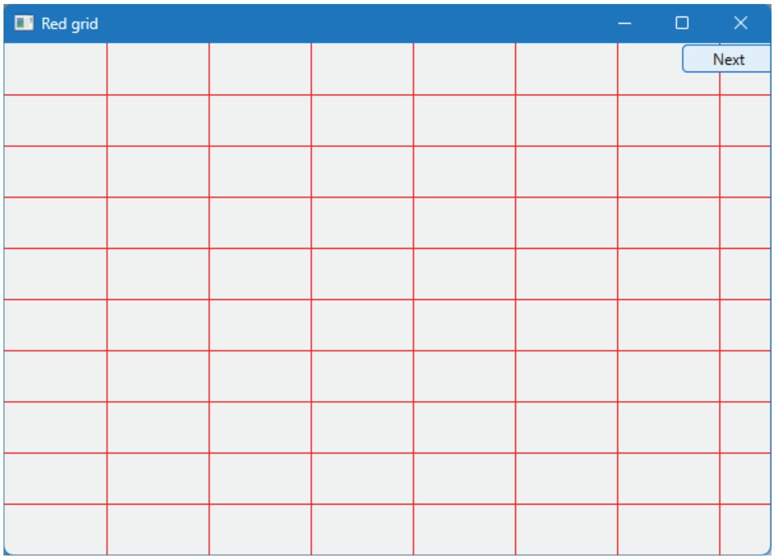
The default constructor, Lines{}, simply sees an empty list {} and creates an empty object (containing no lines): the model of starting out with no points and then add()ing pairs of points as needed is more flexible than any constructor could be. In particular, it allows us to add lines later.

11.4 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, blue, green,
                yellow, white, black,
                magenta, cyan, dark_red,
                dark_green, dark_yellow, dark_blue,
                dark_magenta, dark_cyan,
                palette_index,
                rgb
        };
        enum Transparency { invisible = 0, visible=255 };

        Color(Color_type cc) :c{cc}, ct{cc}, v{visible} { }                    // named colors
        Color(Color_type cc, Transparency vv) :c{cc}, ct{cc}, v{vv} { }
        Color(int cc)                        // choose from palette of 256 popular colors
                :c{cc}, ct{Color_type::palette_index}, v{visible} { }
        Color(Transparency vv) :c{}, ct{Color_type::black}, v{vv} { }
        Color(int r, int g, int b) :c{}, ct{Color_type::rgb}, rgb_color{r,g,b}, v{visible} {}   // RGB

        int as_int() const { return c; }
        int red_component() const { return rgb_color.r; }
        int green_component() const { return rgb_color.g; }
        int blue_component() const { return rgb_color.b; }
        Color_type type() const { return ct; }

        char visibility() const { return v; }
        void set_visibility(Transparency vv) { v=vv; }
private:
        int c = 0;
        Color_type ct = black;
        struct Rgb { int r; int g; int b; };
        Rgb rgb_color = {0,0,0};
        Transparency v;
};
```

The purpose of Color is

- To hide the implementation's notion of color
- To map between integer encodings of colors and the implementation's notion of color
- To give the color constants a scope
- To provide a simple version of transparency (visible and invisible)

As ever, when a class represents something in the real world that people can see and care about, complexity and alternatives creep in.

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 (§11.7.3).

- 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. See also exercise 6.

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 a plain int which we use appropriately for the color_type. We supplied readout functions, such as as_int() and red_component() to allow users to determine what color a Color object represents. Such functions don't change the Color object that they are used for so we declared them const. AA

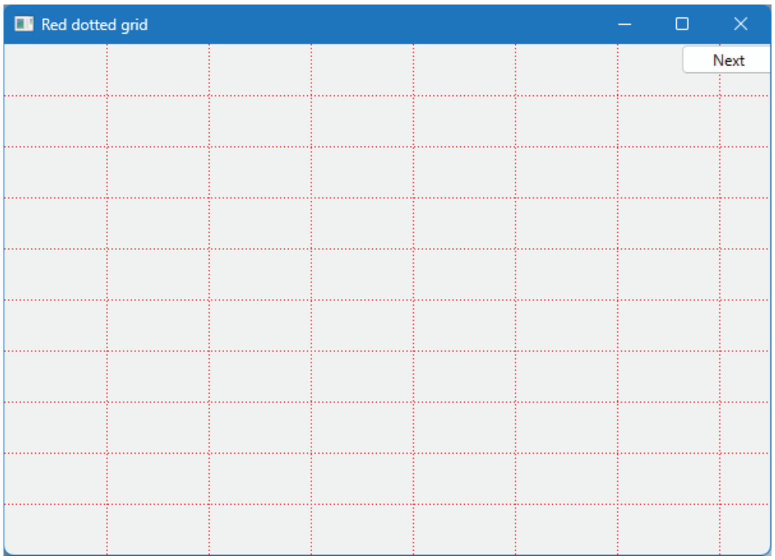
The transparency is represented by the member v which can hold the values Color::visible and Color::invisible, with their obvious meaning. It may surprise you that an "invisible color" can be useful, but it can be most useful to have part of a composite shape invisible. This design allows us to extend the design to support many degrees of transparency, should we find the need to.

11.5 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. Like set_color(), set_style() applies to all lines of a shape. 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.

The Line_style type looks like this:

```
struct Line_style {
        enum Line_style_type {
                solid,                          // -------
                dash,                           // - - - -
                dot,                            // .......
                dashdot,                        // - . - .
                dashdotdot                      // -..-..
        };
        Line_style(Line_style_type ss) :s{ss} { }
        Line_style(Line_style_type ss, int ww) :s{ss}, w(ww) { }
        Line_style() {}

        int width() const { return w; }
        int style() const { return s; }
private:
        int s = solid;
        int w = 1;
};
```

Line_style has two “components”:

- The style proper (e.g., use dashed or solid lines).
- The width (the thickness of the line used). The default width is 1, meaning one pixel.

TRY THIS

Replicate the grid example as above but use a different color and a different line style.

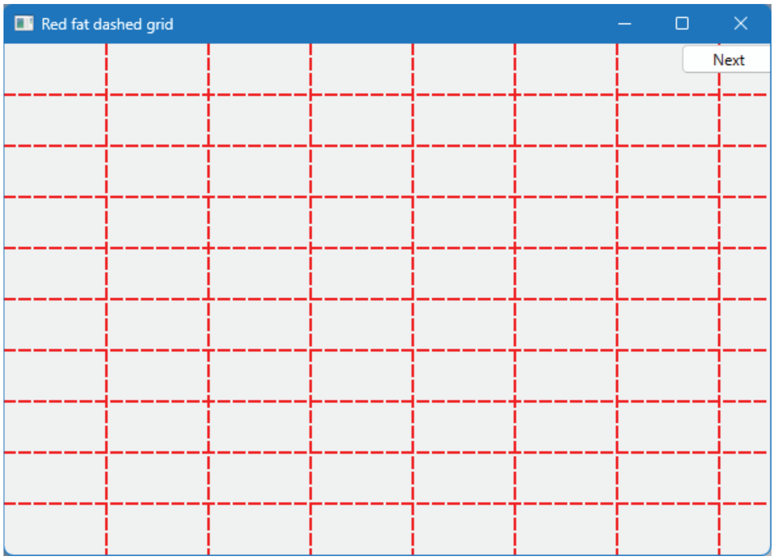
The programming techniques for defining Line_style are exactly the same as the ones we used for Color. Here, we hide the fact that Qt uses its own QFont type to represent line styles. Why is something like that worth hiding? Because it is exactly such a detail that might change as a library evolves or if we change our underlying graphics library. In particular, earlier editions of this book used FLTK where plain ints are used to represent fonts, so hiding that detail saved us from updating user code. In real-world software, such stability is of immense importance.

AA 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.

We can request a fat dashed line like this:

```
grid.set_style(Line_style{Line_style::dash,2});
```

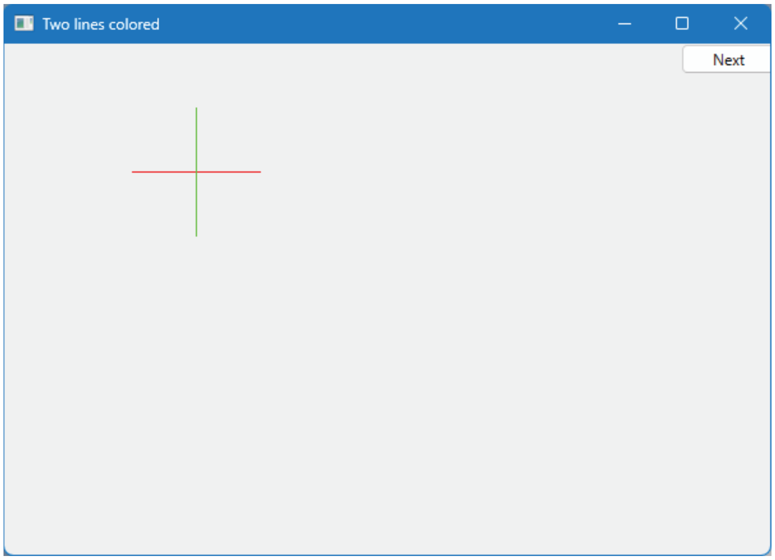
This produces:



When we want to control the color or style for lines separately, we must define them as separate **Lines** as we did in §11.2. For example:

```
horizontal.set_color(Color::red);
vertical.set_color(Color::green);
```

This gives us



11.6 Polylines

A polyline is a sequence of connected lines. *Poly* is the Greek word for "many," and *polyline* is a fairly conventional name for a shape composed of many lines. Polylines are the basis for many shapes and especially for graphs.

In this shape library, we support:

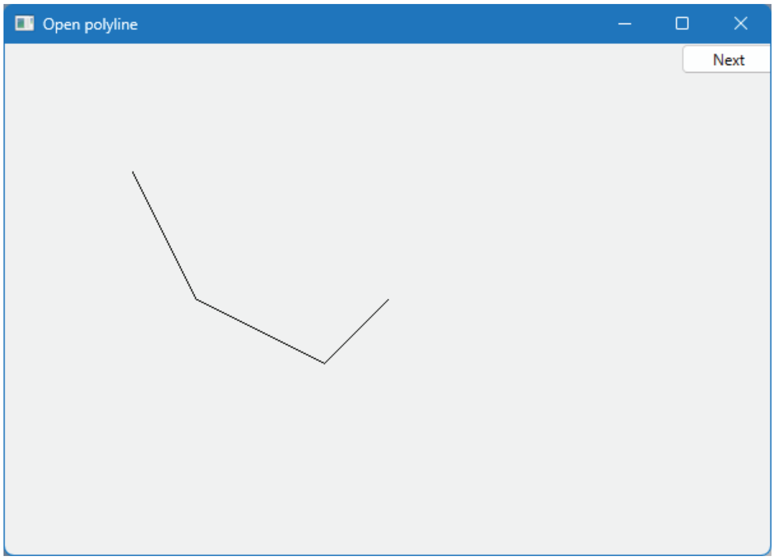
- Open_polyline
- Closed_polyline
- Marked_polyline
- Marks

11.6.1 Open_polyline

An Open_polyline is a shape that is composed of a series of connected Lines defined by a series of points. 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
        Open_polyline(initializer_list<Point> lst = {}) : Shape(lst) {}
        void add(Point p) { Shape::add(p); redraw(); }
        void draw_specific(Painter& painter) const override;
};
```

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()). For Open_polyline, draw_specific() is the function that connects the dots with Lines:

```
void Open_polyline::draw_specific()(Painter& painter) const
{
        if (color().visibility())
                for (int int i=1; i<number_of_points(); ++i)
                        painter.draw_line(point(i−1),point(i));
}
```

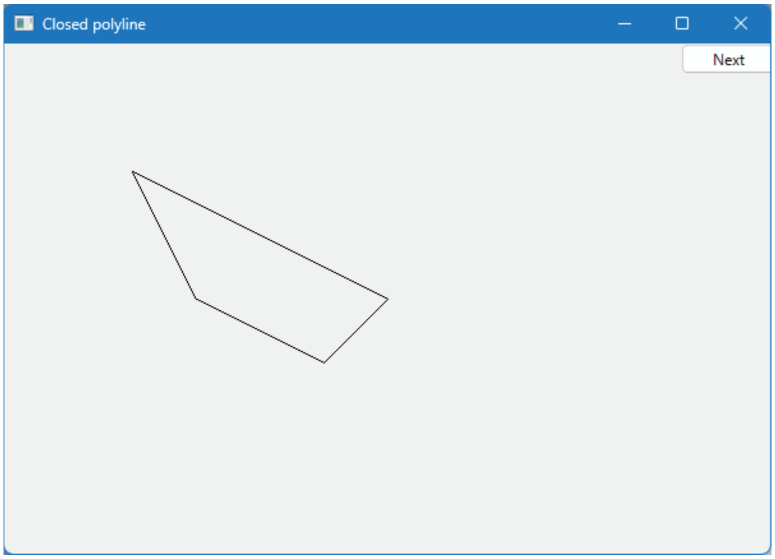
Painter is the class that maps from our drawing functions to Qt's functions for "painting" the screen. It is never used directly by the users of our Graphics/GUI classes.

11.6.2 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 §11.6.1 for a Closed_polyline:

```
Closed_polyline cpl = {    {100,100}, {150,200}, {250,250}, {300,200} };
```

The result is (of course) identical to that of §11.6.1 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;
        void draw_specific(Painter& painter) const override;
};

void Closed_polyline::draw_specific(Painter& painter) const
{
        painter.draw_polygon(*this);
}
```

The using declaration (§7.6.1) says that Closed_polyline has the same constructors as Open_polyline. Closed_polyline needs its own draw_specific() to draw that closing line connecting the last point to the first. It turns out that Qt has an optimized function for that, so we use it.

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 don't. The Qt library QPainter knows how to draw polygons. So, our Painter simply invokes this underlying graphics library. However, as in every other case, the mention of Qt is kept within the implementation of our class rather than being exposed to our users. No user code needs to use painter. If we wanted to, we could replace Qt with another GUI library with very little impact on our users' code.

11.6.3 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", {{100,100}, {150,200}, {250,250}, {300,200}}};
```

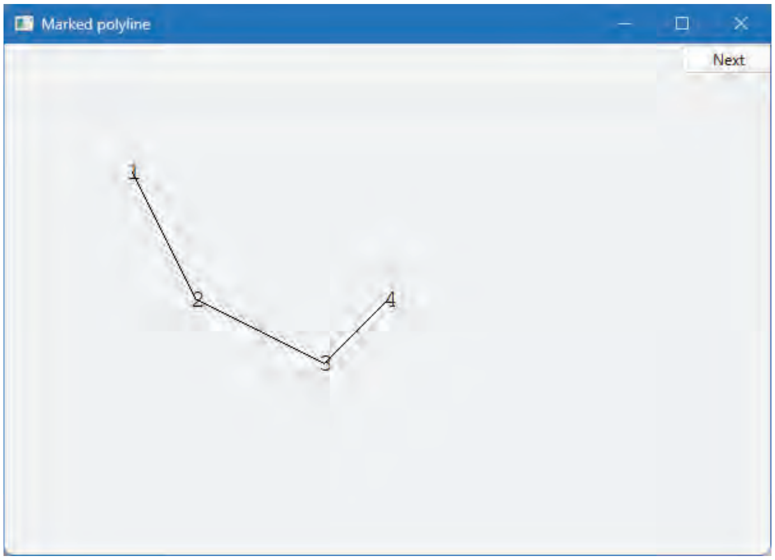
The "1234" is the characters we use to labe the points and the second initlizer is a list of points.

We could have defined mpl like this:

```
Marked_polyline mpl{ "1234" };
mpl.add(Point{100,100});
mpl.add(Point{150,200});
mpl.add(Point{250,250});
mpl.add(Point{300,200});
```

That's verbose, but it shows how we can add points after the initial definition of a Marked_polyline object.

Either way, this produces:



The definition of Marked_polyline is

```
struct Marked_polyline : Open_polyline {
        Marked_polyline(const string& m, initializer_list<Point> lst = {});

        void set_font(Font f) { fnt = f; redraw(); }
        Font font() const { return fnt; }

        void set_font_size(int s) { fnt_sz = s; redraw(); }
        int font_size() const { return fnt_sz; }

        void set_color(Color col) { Shape::set_color(col); set_mark_color(col); }
        void set_mark_color(Color c) { m_color = c; redraw();}
        Color mark_color() const { return m_color;}

        void draw_specifics(Painter& painter) const override;
protected:
        void hide_lines(bool hide = true);    // make the lines invisible or visible
private:
        string mark;
        Font fnt = Font::courier;
        int fnt_sz = 14;          // at least 14 point
        Color m_color;
};
```

By deriving from Open_polyline, we get the handling of Points "for free"; all we have to do is to deal with the marks. We wanted to control the appearance of the marks and that meant adding functions to manipulate the font, color, and size. The alternative would have been to leave those data members public. However, fonts and colors are exactly the kinds of properties for which implementations are likely to change over time, so we hide them behind a functional interface. In particular, draw_specific() becomes

```
void Marked_polyline::draw_specific(Painter& painter) const
{
        Open_polyline::draw_specifics(painter);

        painter.set_line_style(style());
        painter.set_color(m_color);
        painter.set_font(font());
        painter.set_font_size(font_size());

        for (int i=0; i<number_of_points(); ++i)
                draw_mark(painter, point(i),mark[i%mark.size()]);
}
```

The call Open_polyline::draw_specific() takes care of the style 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 marks[i%marks.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_specific() uses a little helper function draw_mark() to actually output a letter at a given point:

```
void draw_mark(Painter& painter, Point xy, char c)
{
        string m(1,c);
        painter.draw_centered_text(xy, m);
}
```

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::Marked_polyline(const string& m, initializer_list<Point> lst)
        : Open_polyline{lst}, mark{(m=="") ? "*" : m}
{
}
```

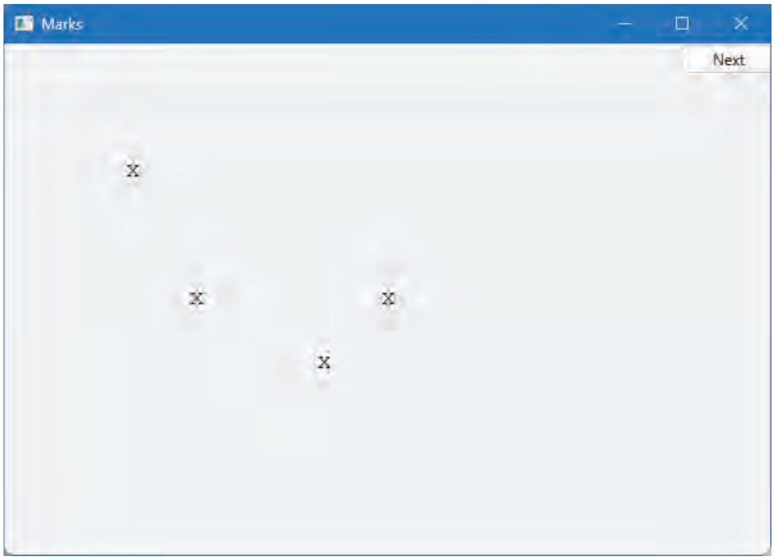
The ?: test for the empty string is needed to avoid draw_specific() trying to access a character that isn't there.

11.6.4 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 where the lines are invisible:

```
struct Marks : Marked_polyline {
        Marks(const string& m, initializer_list<Point> lst = {})
                : Marked_polyline{ m,lst }
        {
                Color orig = color();
                Marked_polyline::set_color(Color::invisible); // hide the lines
                set_mark_color(orig);
        }

        void set_color(Color col) { set_mark_color(col); }
};
```

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 (§8.4.4).

All we need to do is to make the lines invisible and to provide a way for the user to set the color of the characters used as marks.

11.7 Closed shapes

A closed shape differs from a polyline by having a well-defined inside and outside. We can color the inside of a closed shape, but a polyline – even a closed_polyline – may not have an inside to color. In this shape library, we support:

- Polygon
- Rectangle
- Circle
- Ellipse

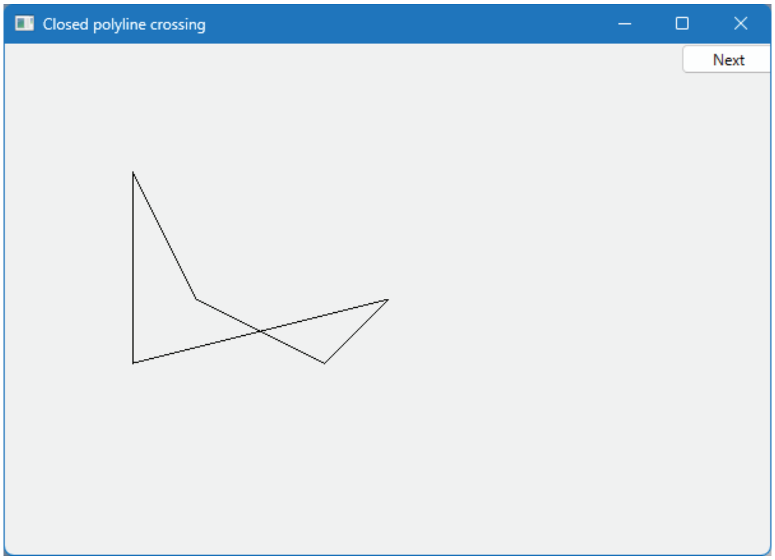
These Shapes support the notion of *fill color* (§12.2).

11.7.1 Polygon

The difference between Polygons and Closed_polylines is that Polygons don't allow lines to cross. For example, the Closed_polyline from §11.6.2 looks like 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 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_specific(Painter& painter) const override;
};
```

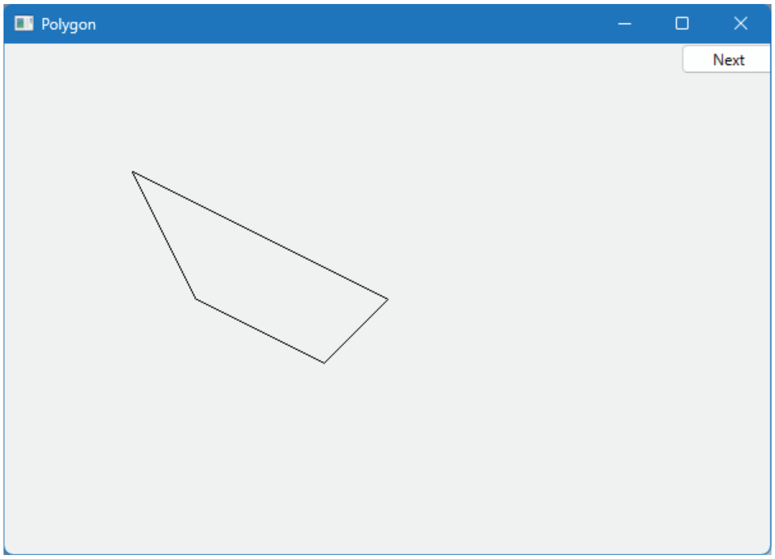
```
void Polygon::add(Point p)
{
        // ... check that the new line doesn't intersect existing lines (code not shown) ...
        Closed_polyline::add(p);
}
```

Here we inherit Closed_polyline's definition of draw_specific(), 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-squared*) 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 (§17.3), 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.

AA 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 always 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.

11.7.2 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_specific(Painter& painter) const override;

        int height() const { return h; }
        int width() const { return w; }
private:
        int w;                  // width
        int h;                  // height
};
```

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);
}
```

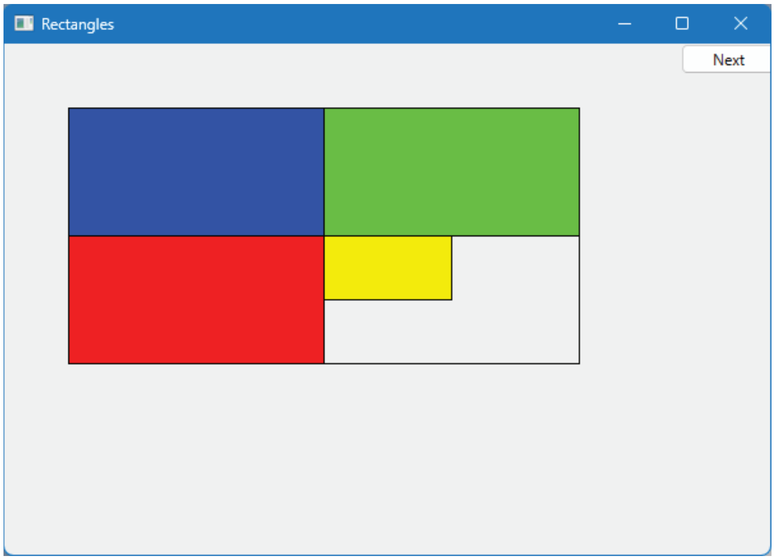
Each constructor initializes the members h and w appropriately (using the member initialization syntax; see §8.2) and stores away the top left corner point in the Rectangle's base Shape (using add()). In addition, it does a simple sanity check: we don't really want Rectangles with non-positive width or height.

CC 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 **Polygon**s and **Circle**s.

The notion of fill color is common to all closed **Shape**s so it is provided by **Shape**. We can set the fill color in a constructor or by the operation **set_fill_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:

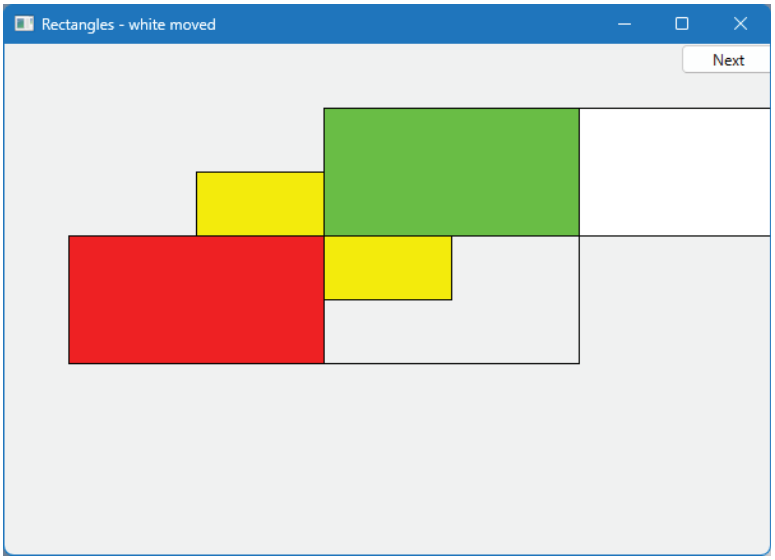


When you don't have a fill color, the rectangle is transparent; that's how you can see a corner of the yellow **rect00**.

We can move shapes around in a window (§12.2.3). For example:

```
rect11.move(400,0);              // to the right of rect21
rect11.set_fill_color(Color::white);
```

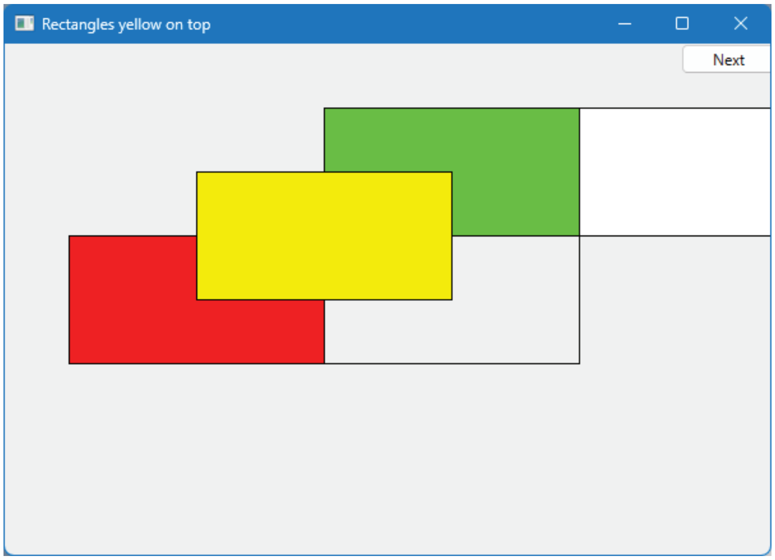
This produces:



CC 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. For example:

```
win12.put_on_top(rect00);
```

This produces:

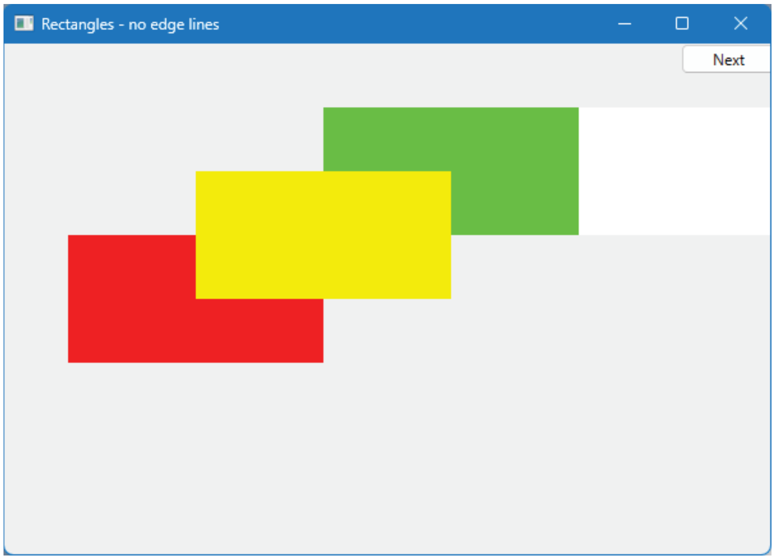


Note how only part of the white rect11 fits in the window. What doesn't fit is "clipped"; that is, it is not shown anywhere on the screen.

Note 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.

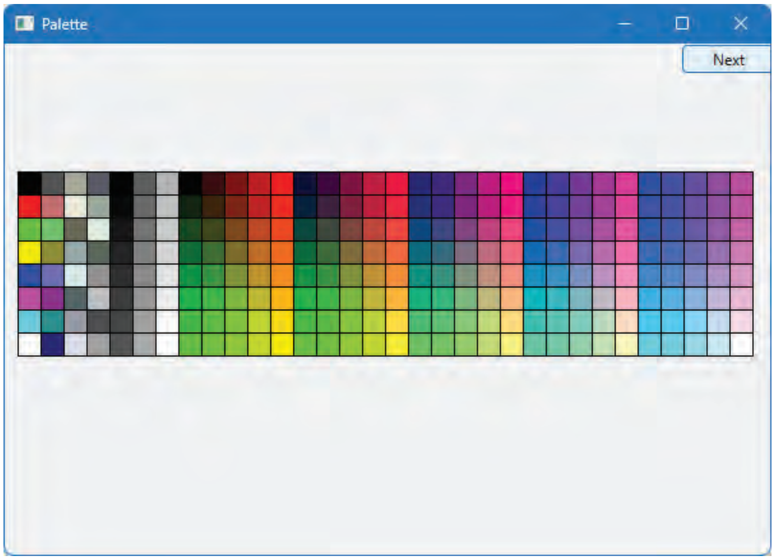
Because it has to deal with both line color and fill color, Rectangle's draw_specific() is a bit messy:

```
void Rectangle::draw_specific(Painter& painter) const
{
        painter.draw_rectangle(point(0), w, h);
}
```

As you can see, Qt's Painter provides functions for drawing rectangles. By default, we draw the lines/outline on top of the fill.

11.7.3 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 a palette; that is, let's make 256 colored squares and draw them in a 32-by-8 matrix with a set of popular colors and ranges of colors. First, here is the result:



AA 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 §12.3. Here, we'll just present our solution: a **vector** type that can hold named and unnamed objects:

AA

```
template<class T>
class Vector_ref {
        // ...
public:
        Vector_ref() {}
        Vector_ref(T& a);
        Vector_ref(unique_ptr<T> x);

        void push_back(T& s);                    // add a named variable
        void push_back(unique_ptr<T> x);         // add an unnamed object

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

```
          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 a named variable

     rect.push_back(make_unique<Rectangle>(Point{50,60},Point{80,90}));      // add an unnamed object

     for (int i=0; i<rect.size(); ++i)
          rect[i].move(10,10);                   // use rect
```

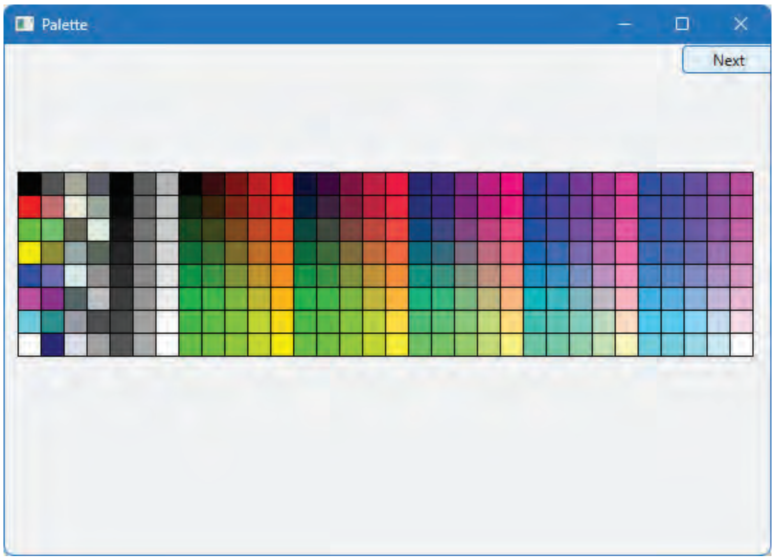
We explain the standard-library make_unique() in §15.5.2 and §18.5.2. For now, it is sufficient to know that we can use it to hold unnamed objects. AA

Experienced programmers will be relieved to hear that we did not introduce a memory leak (§15.4.5) in this example. Also, Vector_ref offers the support needed for range-for loops (§17.6).

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;

          const int max = 32;          //number of columns
          const int side = 18;         // size of color rectangle
          const int left = 10;         // left edge
          const int top = 100;         // top edge
          int color_index = 0;

          for (int i = 0; i < max; ++i) {              // all columns
               for (int j = 0; j < 8; ++j) {           // 8 rows in each column
                    vr.push_back(make_unique<Rectangle>(Point{ i*side+left,j*side+top }, side, side));
                    vr[vr.size()−1].set_fill_color(color_index);
                    ++color_index;                     // move to the next color
                    win.attach(vr[vr.size()−1]);
               }
          }
```

We make a Vector_ref of 256 Rectangles, organized graphically in the Window as a 32-by-8 matrix, reflecting its possible use. 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:



The colors in the palette were chosen to provide easy access to popular colors and ranges of popular colors.

11.7.4 Circle

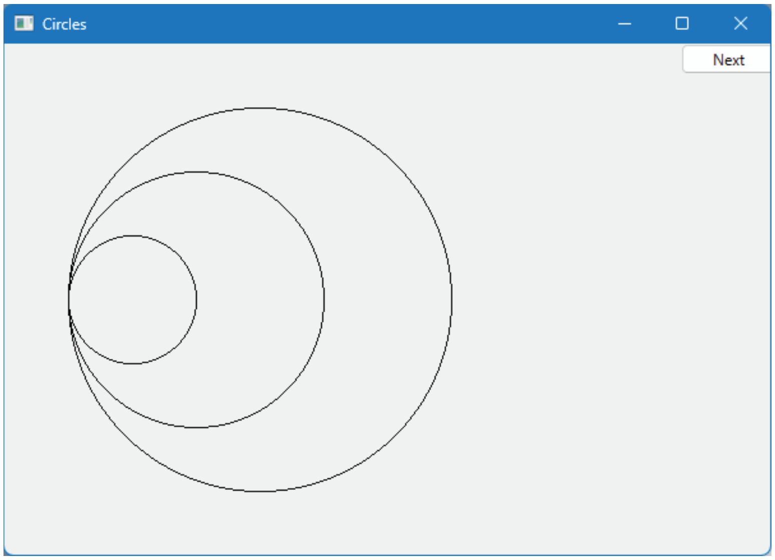
Just to show that the world isn't completely rectangular, we provide class **Circle** and class **Ellipse**. A **Circle** is defined by a center and a radius:

```
struct Circle : Shape {
        Circle(Point p, int rr)         // center and radius
                :r{ rr }
        {
                add(Point{ p.x – r, p.y – r });
        }

        void draw_specific(Painter& painter) const override;

        Point center() const { return { point(0).x + r, point(0).y + r }; }

        void set_radius(int rr) { r=rr; redraw(); }
        int radius() const { return r; }
private:
        int r;
};
```

We can use **Circle** like this:

```
Circle c1 {Point{100,200},50};
Circle c2 {Point{150,200},100};
c2.set_fill_color(Color::red);
win.c1.put_on_top();
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. 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_specific(Painter& painter) const
{
        painter.draw_ellipse(center(), r, r);
}
```

Note the use of the Painter to draw the circle. Qt offers an optimized function for drawing ellipses so we use that.

11.7.5 Ellipse

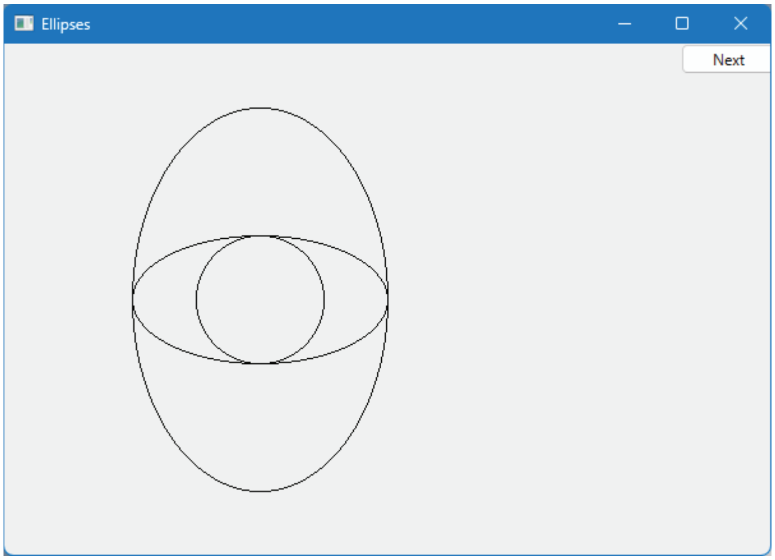
An ellipse is similar to Circle but is defined with a major and a minor axis, instead of a radius; that is, to define an ellipse, we give the center's coordinates, the distance from the center to a point on the *x* axis, and the distance from the center to a point on the *y* axis:

```
struct Ellipse : Shape {
        Ellipse(Point p, int ww, int hh) // center, min, and max distance from center
        :w{ ww }, h{ hh } {
                add(Point{ p.x – ww, p.y – hh });
        }

        void draw_specific(Painter& painter) const override;

        Point center() const { return{ point(0).x+w, point(0).y+h }; }
        Point focus1() const;
        Point focus2() const;

        void set_major(int ww) { w=ww; redraw(); }
        int major() const { return w; }
        void set_minor(int hh) { h=hh; redraw(); }
        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.

For an ellipse there are two points on the long axis, so that the sum of distances from any point on the ellipse to those two points, called focus points, is the same. Given an **Ellipse**, we can compute a focus. For example:

```
Point focus1() const
{
        return{ center().x + round_to<int>(sqrt(w*w−h*h)), center().y };
}

Point focus2() const
{
        return{ center().x − round_to<int>(sqrt(w*w−h*h)), center().y };
}
```

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. **CC**

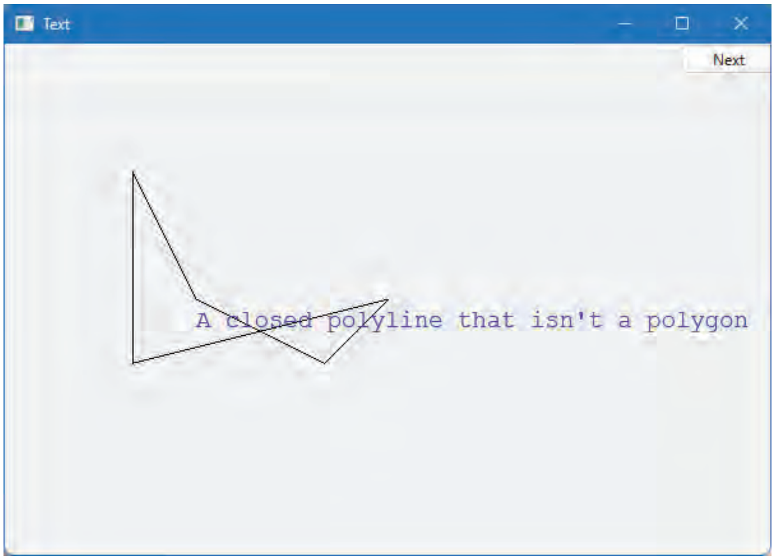
AA When we design classes, we have to be careful not to be too clever and not to be deceived by our "intuition" into defining classes that don't make sense as classes in our code. Conversely, we have to take care that our classes represent some coherent concept and are not just a collection of data and 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.

11.8 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 §11.6.2:

```
Text t {Point{150,200},"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 (§9.11) are useful for composing **string**s for display in **Text** objects (examples in §10.7.8, §14.5). **Text** is defined like this:

```
struct Text : Shape {
    Text(Point x, const string& s) : lab{ s } { add(x); }    // the point is the bottom left of the first letter
```

```
        void draw_specific(Painter& painter) const override;

        void set_label(const string& s) { lab = s; redraw(); }
        string label() const { return lab; }

        void set_font(Font f) { fnt = f; redraw(); }
        Font font() const { return Font(fnt); }

        void set_font_size(int s) { fnt_sz = s; redraw(); }
        int font_size() const { return fnt_sz; }
private:
        string lab;                         // label
        Font fnt = Font::courier;
        int fnt_sz = 14;                    // font size in the conventional "point" unit
};
```

If you want the font character size to be different from the default (14), 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, in an earlier version of our interface library, an update of the FLTK library that we used then changed its default in a way that broke existing programs by making the characters tiny. We decided to prevent that problem.

We provide the initializers for fnt and fnt_sz 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_specific() because only the Text class knows how its string is stored:

```
void Text::draw_specific(Painter& painter) const
{
        painter.set_font(font());
        painter.set_font_size(font_size());
        painter.draw_text(point(0), lab);
}
```

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 for them 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:

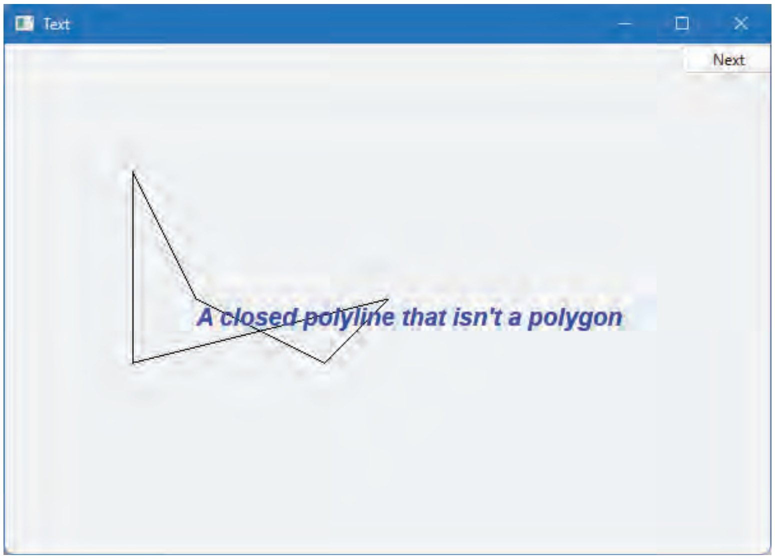
```
struct Font {
        enum Font_type {
                helvetica, helvetica_bold, helvetica_italic, helvetica_bold_italic,
                courier, courier_bold, courier_italic, courier_bold_italic,
                times, times_bold, times_italic, times_bold_italic,
                symbol,
                screen, screen_bold,
                zapf_dingbats
        };
        Font(Font_type ff) :f(ff) { }

        int as_int() const { return f; }
private:
        int f = courier;
};
```

The style of class definition used to define **Font** is the same as we used to define **Color** (§11.4) and **Line_style** (§11.5).

We can use that to make our text more prominent:

```
t.set_font(Font::helvetica_bold_italic);
t.set_color(Color::blue);
```

This produces:



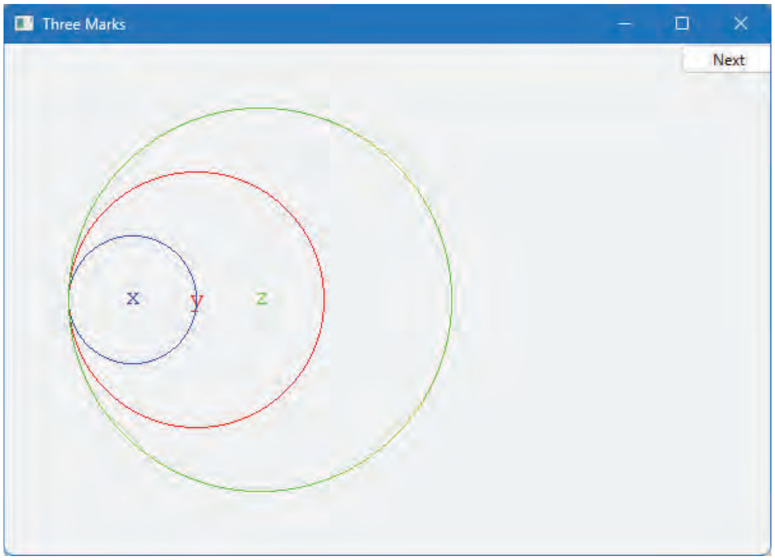
11.9 Mark

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 §11.2 or by using **Marks** (§11.6.4). 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 §11.7.4 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);

m1.set_mark_color(Color::blue);
m2.set_mark_color(Color::red);
m3.set_mark_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);
        }
};
```

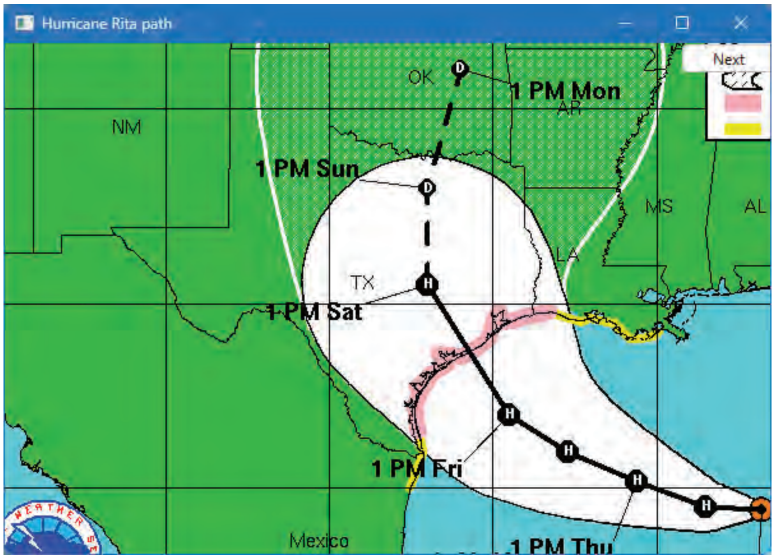
The string{1,c} is a constructor for string, initializing the string to contain the single character c.

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.

11.10 Image

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:



In our graphics interface library, we represent an image in memory as an object of class **Image**:

```
class ImagePrivate; // Implementation class that provides the interface to Qt classes

struct Image : Shape {
        Image(Point xy, string s);
        ˜Image() {};

        void set_mask(Point xy, int ww, int hh) { w=ww; h=hh; cx=xy.x; cy=xy.y; redraw(); }
        void move(int dx,int dy) override { Shape::move(dx,dy); redraw(); }
        void scale(int ww, int hh, bool keep_aspect_ratio = true);

        ImagePrivate& get_impl() const { return *impl; }
        void draw_specific(Painter& painter) const override;
private:
        int w,h,cx,cy;          // define "masking box" within image relative to position (cx,cy)
        Text fn;
        std::unique_ptr<ImagePrivate> impl;
};
```

The **Image** constructor tries to open a file with the name given to it. 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:

```
Image::Image(Point xy, string s)
        :w(0), h(0), fn(xy,""), impl(std::make_unique<ImagePrivate>())
{
        add(xy);

        if (!can_open(s)) {
                fn.set_label("cannot open \"" + s + '\"');
                return;
        }
        impl–>load(s);
}
```

If the image cannot be displayed (e.g., because the file wasn't found), the Image displays a Text including the name of the file it failed to open. That way, we can see the problem right there on the screen, rather than trying to find the problem in the code. This is a common technique in GUI libraries.

Images can be encoded in a bewildering variety of formats, such as JPEG and GIF. We use the file's suffix, e.g. jpg or gif, to pick the kind of object we create to hold the image.

That mapping from our interface library to Qt's facilities is done by ImagePrivate. We access an ImagePrivate using a unique_ptr that takes care of its proper deletion (§18.5)

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.is_open();
}
```

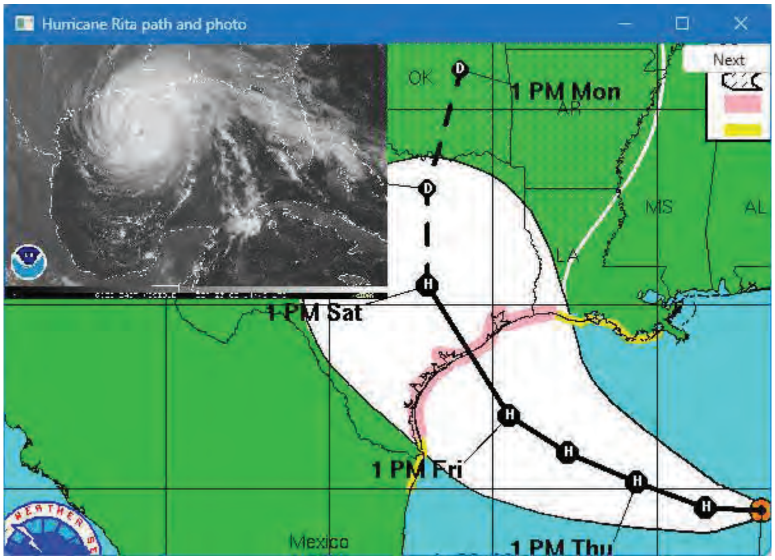
Opening a file and then closing it again is a fairly clumsy way of portably separating errors related to "can't open the file" from errors related to the format of the data in the file.

TRY THIS

Write, compile, and run a simple program displaying a picture of your choice. Did it fit in your window? If not, what happened?

We selected part of that image file and added a photo of Rita as seen from space (rita.jpg):

```
Image path{ Point{0,0},"rita_path.gif" };
path.set_mask(Point{ 50,250 }, 600, 400);    // select likely landfall

Image rita{ Point{0,0},"rita.jpg" };
rita.scale(300, 200);
```



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.

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 **Line**s 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 **Point**s 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**?
[26] How do you scale an **Image**?

Terms

closed shape	**Image**	**Point**	**Shape**
Color	image encoding	**Polygon**	**Text**
Ellipse	invisible	polyline	**Open_polyline**
fill	JPEG	unnamed object	**Closed_polyline**
Font	**Line**	**Vector_ref**	**Lines**
font size	**Line_style**	visible	**Marked_polyline**
GIF	open shape	**Rectangle**	**Mark**

Exercises

For each "define a class" exercise, display a couple of objects of the class to demonstrate that they work.

[1] Define a class **Arrow**, which draws a line with an arrowhead.

[2] 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**.

[3] Define the functions from exercise 2 for a **Circle** and an **Ellipse**. Place the connection points on or outside the shape but not outside the bounding rectangle.

[4] Write a program that draws a class diagram like the one in §10.6. It will simplify matters if you start by defining a **Box** class that is a rectangle with a text label.

[5] Make an RGB color chart (e.g., search the Web for "RGB color chart").

[6] 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.

[7] Tile a part of a window with **Regular_hexagons** (use at least eight hexagons).

[8] 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.

[9] 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.

[10] Draw a circle. Move a mark around on the circle (let it move a bit each time you hit the "Next" button).

[11] Draw the color matrix from §11.7.3, but without lines around each color.

[12] Define a right triangle class. Make an octagonal shape out of eight right triangles of different colors.

[13] "Tile" a window with small right triangles.

[14] Do the previous exercise, but with hexagons.

[15] Do the previous exercise, but using hexagons of a few different colors.

[16] 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.

[17] 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.

[18] There is an **Arc** class in **Graph.h**. Find it and use it to define a box with rounded corner.

Postscript

Chapter 10 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.

12

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. Ypu could think of this chapter as "Class Design and Implementation" or even "Object-Oriented Programming."

12.1 Design principles

This chapter focuses on the techniques usually referred to as *Object-Oriented Programming*. To complement, we have concrete classes (Chapter 8, Chapter 17) and parameterized classes (Chapter 18) and their associated techniques.

12.1.1 Types

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?

CC 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.

CC 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.

CC

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 Qt details in our implementation for you to get an idea of how to use that (or a similar "full-blown" industrial graphics/GUI library) directly, should you so desire. However, if you decide to go that route, wait until you have absorbed Chapter 15, Chapter 16, and Chapter 17. Those chapters contain information about pointers and memory management that you need for successful direct use of most graphics/GUI libraries.

AA

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.
- A single class providing "everything" would leave the user messing with data and options without a framework to help understanding, debugging, and performance.
- Large classes and large functions are harder to get correct than a collection of smaller ones.
- The "many small classes" model simplifies adding new classes to a library.

For example, imagine what it would take to add a **Spline** or a **Clock_face** in the two alternative ways to representing **Shape**s.

12.1.2 Operations

CC

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.

AA

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);                 // an 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.

AA Logically identical operations have the same name. For example, every function that adds points, lines, etc. to any kind of shape is called add(). 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 Chapter 19, Chapter 20, and Chapter 21.

12.1.3 Naming

AA 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:

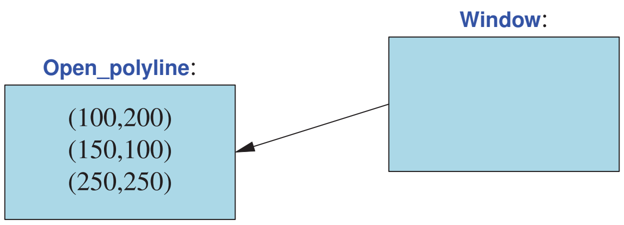
```
Open_polyline opl;
opl.add(Point{100,100});
opl.add(Point{150,200});
opl.add(Point{250,250});
```

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. 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();
}
```

XX

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 15, 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 (§11.7.3) to help with that.

Note that had we declared f() to take its Window as a const reference argument (as recommended in §7.4.6), the compiler would have prevented our mistake. However, 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.

12.1.4 Mutability

CC

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
```

AA As you might have noticed in Chapter 11, 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 don't take full advantage of this opportunity to check against errors, 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.

12.2 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:

```
struct Shape  {               // deals with color and style, and holds sequence of lines
        virtual ˜Shape() { }                                     // destructor: see §15.5.2

        Shape(const Shape&) = delete;                            // prevent copying: see §12.4.1
        Shape& operator=(const Shape&) = delete;

        virtual void move(int dx, int dy);                       // move the shape +=dx and +=dy

        void set_color(Color col) { lcolor = col; redraw(); }    // write
        Color color() const { return lcolor; }                   // read
        void set_style(Line_style sty) { ls = sty; redraw(); }
        Line_style style() const { return ls; }

        void set_fill_color(Color col) { fcolor = col; redraw(); }
        Color fill_color() const { return fcolor; }

        Point point(int i) const { return points[i]; }
        int number_of_points() const { return naro<int>(points.size()); }

        void set_window(Window∗ win) { parent_window = win; }

        void draw(Painter& painter) const;                   // deal with color and draw_specifics
    protected:
        Shape(std::initializer_list<Point> lst = {});        // add() the Points to this Shape

        void add(Point p){ points.push_back(p); redraw(); }
        void set_point(int i, Point p) { points[i] = p; redraw(); }

        void redraw();
    private:
        virtual void draw_specifics(Painter& painter) const = 0; // draw this specific shape

        Window∗ parent_window = nullptr;                     // The window in which the Shape appears
        vector<Point> points;                                // not used by all shapes
        Color lcolor = Color::black;
        Line_style ls;                                       // use the default line style
        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 5 data members and 19 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.

12.2.1 An abstract class

Consider first `Shape`'s constructor:

```
    protected:
        Shape(initializer_list<Point> lst = {});          // add() the Points to this Shape
```

The constructor `add()`s the elements of its argument list to the `Shape`'s `vector<Point>`:

```
    Shape::Shape(initializer_list<Point> lst)
    {
        for (Point p : lst)
            add(p);
    }
```

If we don't provide a set of **Points**, we default to the empty **initializer_list**.

The constructor is **protected**. That means that it can only be used directly from classes derived from **Shape** using the **:Shape** notation. That is, for specific shapes, such as **Circle** and **Closed_polyline**, rather than for the general notion of a shape. Classes defined using the **:Shape** notation are called *derived classes* and **Shape** is called their *base class* (§12.3). The purpose of that **protected:** is to ensure that we don't make **Shape** objects directly. For example:

```
Shape ss;        // error: cannot construct Shape
```

CC 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.

CC 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* (§12.3.5). 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 type of camera? 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 of that class.

12.2.2 Access control

Class **Shape** declares all data members **private**:

```
private:
        Window* parent_window = nullptr;        // The window in which the Shape appears
        vector<Point> points;                   // not used by all shapes
        Color lcolor = Color::black;            // color for lines and characters (with default)
        Line_style ls;                          // use the default line style
        Color fcolor = Color::invisible;        // fill color (default: no color)
```

The initializers for the data members don't depend on constructor arguments, so we specified them in the data member declarations. As ever, the default value for a vector is "empty" so we didn't have to be explicit about that. The constructor will apply those default values.

AA 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 (§8.7.4).

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? (§10.7.4, §11.7.5).
- Lines allows only pairs of points to be added (and not an individual point; §11.3).
- 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 (§11.7.1).

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. AA

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 (§8.4.3). 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." 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_specifics(Painter& painter) const           // draw lines connecting pairs of points
{
        if (color().visibility())
                for (int i=1; i<number_of_points(); i+=2)
                        painter.draw_line(point(i−1),point(i));
}
```

CC 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. For example, 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(initializer_list<Point> = {});
        void draw() const;                                   // deal with color and call draw_specifics
        virtual void draw_specifics(Painter& painter) const;  // draw this specific shape
        virtual void move(int dx, int dy);                    // move the shape +=dx and +=dy
        virtual ˜Shape();

        Window∗ parent_window;
        vector<Point> points;                                 // not used by all shapes
        Color lcolor;
        Line_style ls;
        Color fcolor;
};
```

CC What value did we add by those extra 14 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" (§8.4.3). 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;       // earlier: use FLTK's color type
int line_style;        // earlier: use a plain integer to represent a 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). AA

We could simplify Shape even further by removing everything not needed by all shapes. For example, a Line doesn't use fill color and a Circle uses only a single Point, rather than a vector of Points. That would make Shape a pure interface as described in §12.3.5, but would also force us to repeat a lot of code in implementations of specific shapes.

12.2.3 Drawing shapes

We have now described almost all but the real heart of class Shape:

```
void draw(Painter& painter) const;                    // deal with color and style and call draw_specifics
virtual void draw_specifics(Painter& painter) const;  // draw this specific shape 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 (§12.5), but drawing is Shape's essential business. It does so using Qt and the operating system's basic machinery, but from a user's point of view, it provides just two functions:

- draw() chooses color and style; then calls draw_specifics(); then restores color and style.
- draw_specifics() puts pixels on the screen for a specific Shape.

The draw() function doesn't use any novel techniques. It simply calls Qt functions to set the color and style to what is specified in the Shape, calls draw_specifics() to do the actual drawing on the screen, and then restores color and style to what they were before the call:

```
void Shape::draw(Painter& painter) const
{
        painter.save();
        painter.set_line_style(style());
        painter.set_color(color());
        painter.set_fill_color(fill_color());
        draw_specifics(painter);
        painter.restore();
}
```

Shape::draw() doesn't directly handle fill color or the visibility of lines. Those are handled by the individual draw_specifics() functions that have a better idea of how to interpret them. For example set_fill_color() is a "no op", does nothing, for shapes that are not closed (§11.7). In principle, all color and style handling could be delegated to the individual draw_specifics() functions, but that would be quite repetitive.

CC Now consider how we might handle draw_specifics(). 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.

CC 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_specifics() which we want to call instead of Shape's draw_specifics(). That's what the virtual in the declaration of Shape::draw_specifics() means:

```
struct Shape {
        // ...
        virtual void draw_specifics(Painter& painter) const;      // draw this specific shape appropriately
        // ...
};

struct Circle : Shape {
        // ...
        void draw_specifics(Painter& painter) const override;
        // ...
};
```

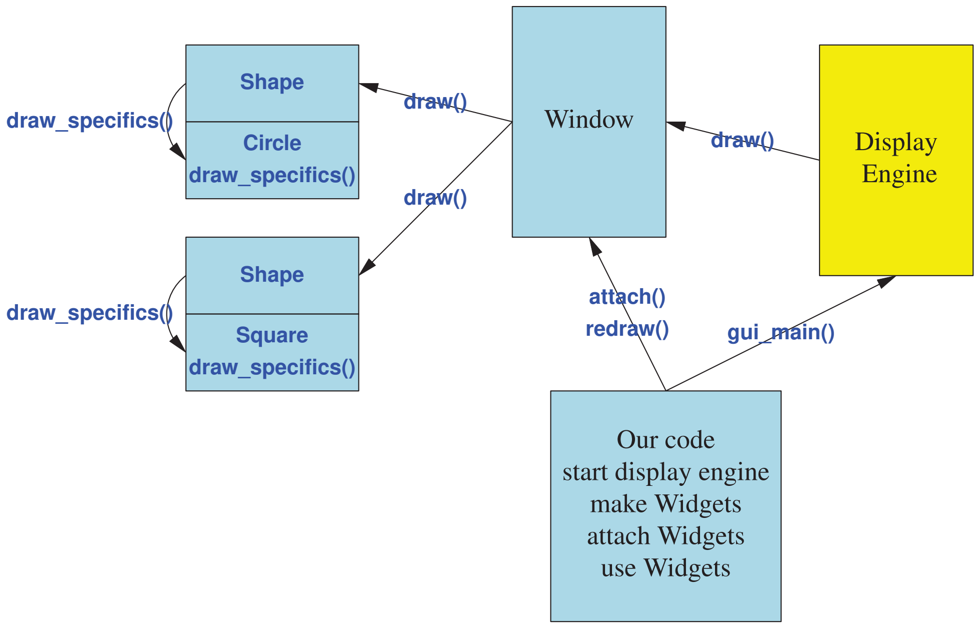
So, Shape's draw_specifics() must somehow invoke Circle's draw_specifics() if the Shape is a Circle and Rectangle's draw_specifics() if the Shape is a Rectangle. That's what the word virtual in the draw_specifics() declaration ensures: Circle has defined its own draw_specifics() (with the same type as Shape's draw_specifics()), so that Circle::draw_specifics() will be called. 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_specifics() 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 §10.2. The system that drives the screen knows about Window. Window knows about Shape and can call Shape's draw(). Finally, draw() invokes the draw_specifics() for the particular kind of shape.

If we make a change to a Shape, we call redraw() to inform the Window that it must refresh its image on the screen accordingly.

A call of Application's 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 (§14.7). Instead we use wait_for_button(), which implicitly invokes the display engine in a more simple-minded manner.

12.3 Base and derived classes

CC

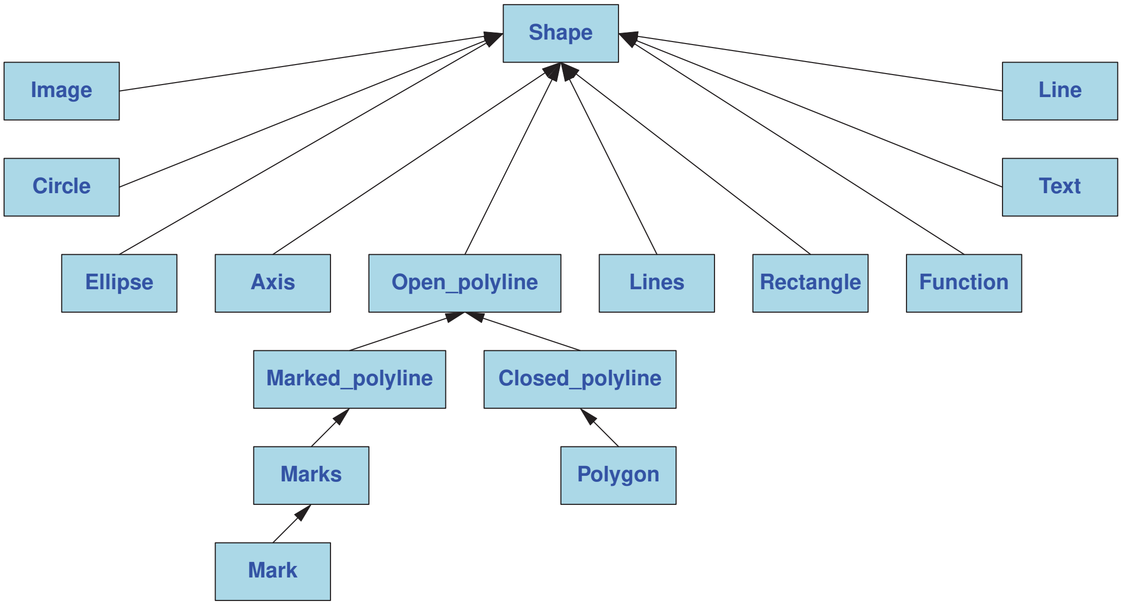
Let's take a more technical view of base and derived classes; that is, let us for this section (only) change the focus of discussion from programming, application design, and graphics to programming language features. When designing our graphics interface library, we relied on three key language mechanisms:

- *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_specifics() (through draw()) for a Shape that is a Circle, it is the Circle's draw_specifics() that is executed, rather than Shape's own draw_specifics(). 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 (§12.2.2). 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 Chapter 20 and Chapter 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 (PPP2.Ch22).

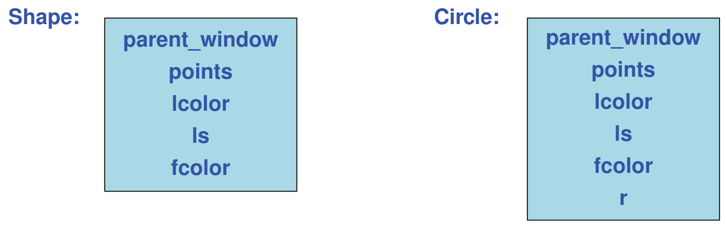
That was a lot of technical terminology! But what does it all mean? And how does it actually work on our computers? Let's first draw a simple diagram of our graphics interface classes showing their inheritance relationships:



The arrows point from a derived class to its base. Such diagrams help visualize class relationships and often decorate the whiteboards 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.

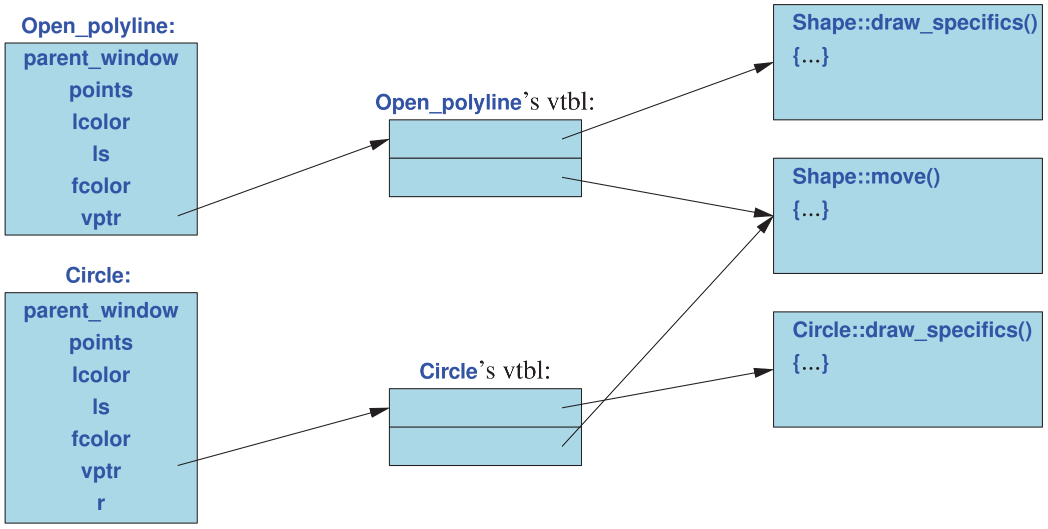
12.3.1 Object layout

CC How are objects laid out in memory? As we saw in §8.4.1, 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.

CC 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_specifics(). 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 Chapter 17 and Chapter 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_specifics() 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_specifics(), the compiler generates a call to the function found in the draw_specifics() slot in the vtbl for x. Basically, the code just follows the arrows on the diagram. So if x is a Circle, Circle::draw_specifics() will be called. If x is of a type, say Open_polyline, that uses the vtbl exactly as Shape defined it, Shape::draw_specifics() 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_specifics()) 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_specifics() overrides Shape::draw_specifics().

AA 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. When we need to select among an unknown set of alternatives at run time, we can't code the functionality to be any faster or to use less memory by using other language features than a virtual function call. You can see that for yourself: measure before making statements about efficiency (§20.4).

12.3.2 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 { /* ... */ };
```

CC By default, the members of a struct are public (§8.3), 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 the keyword virtual is neither required nor allowed outside the class. For example:

```
struct Shape {
        // ...
        virtual void draw_specifics(Painter& painter) const;
        virtual void move();
        // ...
};

virtual void Shape::draw_specifics(Painter& painter) const { /* ... */ }    // error: "virtual" outside class
void Shape::move() { /* ... */ }                                            // OK
```

12.3.3 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: XX

```
struct Circle : Shape {
        void draw_specifics(int) const;       // probably a mistake (int argument?)
        void drawlines() const;               // probably a mistake (misspelled name?)
        void draw_specifics();                // probably a mistake (const missing?)
        void draw_specifics() const;          // probably a mistake (Painter& argument missing?)

        void draw_specifics(Painter&) const;                // OK: implicit override
        void draw_specifics(Painter&) const override;       // OK: explicit override

        //  ...
};
```

Here, the compiler will see four functions that are independent of Shape::draw_specifics() (because they have a different name or a different type) and won't override them. A good compiler will warn about these likely mistakes.

The draw_specifics() 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::g B::f B::g D::f D::g DD::f DD::g
```

When you understand why, you 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 is not const
        void g() const override { cout << "DD::g "; }          // error: no virtual D::g to override
};
```

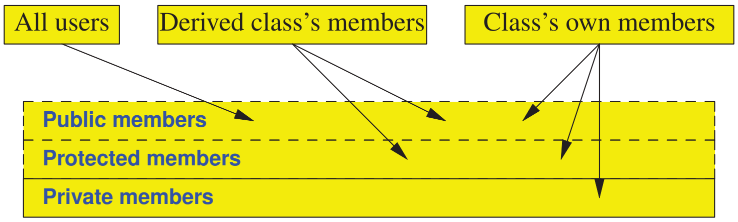
Explicit use of **override** is particularly useful in large, complicated class hierarchies.

12.3.4 Access

C++ provides a simple model of access to members of a class. A member of a class can be **CC**

- *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++* [DnE], the C++ *History of Programming Languages* papers [HOPL-4], and *The C++ Programming Language*. The official definition of C++ is *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.

12.3.5 Pure virtual functions

CC 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*).

CC In §12.2.1, 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 must 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
```

AA

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).

12.4 Other Shape functions

To complete Shape, we need to deal with copying Shapes. We also present how to move a Shape on the screen.

12.4.1 Copy

CC

The Shape class declared its copy constructor and the copy assignment deleted:

```
Shape(const Shape&) =delete;                    // prevent copying
Shape& operator=(const Shape&) =delete;
```

This eliminates the default copy operations for Shapes and for every class derived from Shape that hasn't defined its own copy operations (and we won't do that). Consider:

```
void copy_to(Circle& c, Rectangle& r)
{
        c = r;          // this doesn't look innocent at all (and fortunately it is an error)
        // ...
}
```

Assigning a Rectangle to a Circle doesn't make sense:
- A Rectangle is represented by two Points.
- A Circle is represented by a Point and an int.

Those two objects aren't even of the same size! If allowed, what space would the Rectangle be copied into (§12.3.1)? Also, Circle and Rectangle have completely different draw_specifics() functions.

So why would c=r have worked if we hadn't prohibited it? Had copying of Shapes been allowed, Shape's copy assignment would have been called, treating the Circle and the Rectangle as Shapes:

```
Shape& operator=(const Shape&);         // copy assignment
```

Implementing that so that it makes sense is impossible in general. For example, assigning a Circle to a Rectangle simply doesn't make sense. For pairs of Shapes, where an assignment does make sense, we can define functions to handle those special cases.

XX

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 (i.e., has at least one virtual function), disable its copy constructor and copy assignment using =delete as was done for Shape. See also §12.3.

12.4.2 Moving Shapes

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 (auto& xy : points) {
                xy.x += dx;
                xy.y += dy;
        }
        redraw();
}
```

Like draw_specifics(), 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 (§10.7.1, §13.4). Every class derived from Shape that doesn't store all its data in Shape must define its own move().

Note the call of redraw(). After a change to a Shape, we must tell the Window to refresh that Shape's image on the screen (§12.2.3).

12.5 Benefits of object-oriented programming

CC 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.

XX 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_specifics() 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!

AA 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).

AA 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_specifics() 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" (for now, Qt 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). 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.

CC

In other words, we can add new **Shape**s 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, Chapter 15 – Chapter 18 where we define a **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.

XX

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 §12.3.5.

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 *not* make a member function virtual?
[20] Why would you make a virtual member function pure?
[21] What does overriding mean?
[22] Why should you always suppress copy operations for a class in a class hierarchy?
[23] How does interface inheritance differ from implementation inheritance?
[24] What is object-oriented programming?

Terms

abstract class	mutability	**public**	access control
object layout	pure virtual function	base class	object-oriented
subclass	derived class	**override**	superclass
dispatch	polymorphism	**virtual** function	encapsulation
private	virtual function call	inheritance	**protected**
virtual function table	**=0**	OOP	**=delete**

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 class Rounded that is like a Rectangle, except that it has rounded corners. Use class Arc that you can find in the PPP support code on www.stroustrup.com/programming.html. Test it.

[10] Define a class Box that is a closed shape like a Rectangle (so it has fill color), except that it has rounded corners. Use class Pie that you can find in the PPP support code on www.stroustrup.com/programming.html.

[11] 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.

[12] 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.

[13] 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 conventionally grow downward from a top node (amusingly, but logically, often called the root).

[14] 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).

[15] 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.

[16] 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 "lrrlr" 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).

[17] 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.

[18] The exceptions defined in the C++ standard library, such as exception, runtime_error, and out_of_range (§4.6.3), are organized into a class hierarchy (with a 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.

13

Graphing Functions and Data

The best is the enemy of the good.
– Voltaire

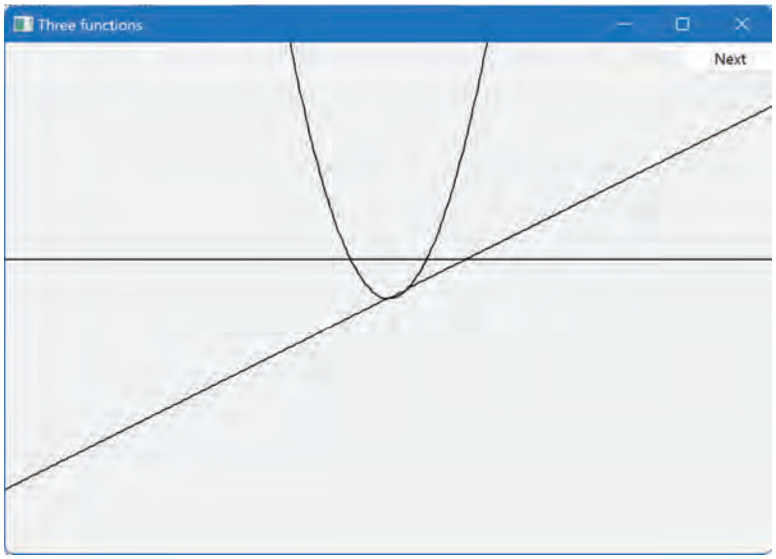
If you are in any empirical field, you need to graph data. If you are in any field that uses math to model phenomena, you need to graph functions. This chapter discusses basic mechanisms for such graphics. As usual, we show the use of the mechanisms and also discuss their design. The key examples are graphing a function of one argument and displaying values read from a file.

13.1 Introduction

AA 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.

13.2 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 0.5*x; }          // the slope is 0.5
```

This is the function that generated the sloping line. For every x, we get the y value 0.5*x. In other words, (x,y)==(x,0.5*x). The point where the two lines cross is (2,1).

Now we can try something more slightly 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,"Three functions"};

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();
```

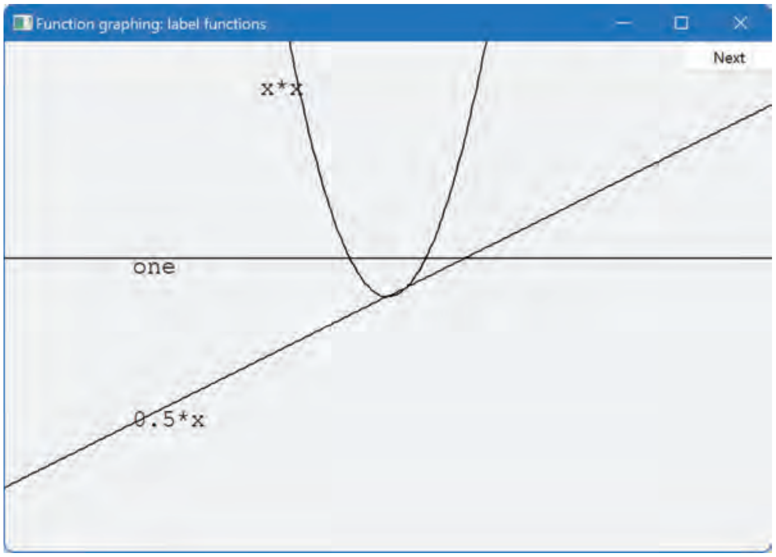
First, we define a bunch of constants so that we won't have to litter our code with "magic constants." Then, we make a window, define the functions, attach them to the window, and finally give control to the graphics system to do the actual drawing.

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): [r_min:r_max). The fourth argument (here, orig) tells the Function where the origin (0,0) is to be located within the window.

XX 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 (§13.3). First, however, let's label our graphs:



AA 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 (§11.8):

```
Text ts {Point{100,y_orig−40},"one"};
Text ts2 {Point{100,y_orig+y_orig/2−20},"0.5∗x"};
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 note that **0.5∗x** touches **x∗x** at **(0,0)** and that **one** crosses **0.5∗x** 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 (§13.4): AA

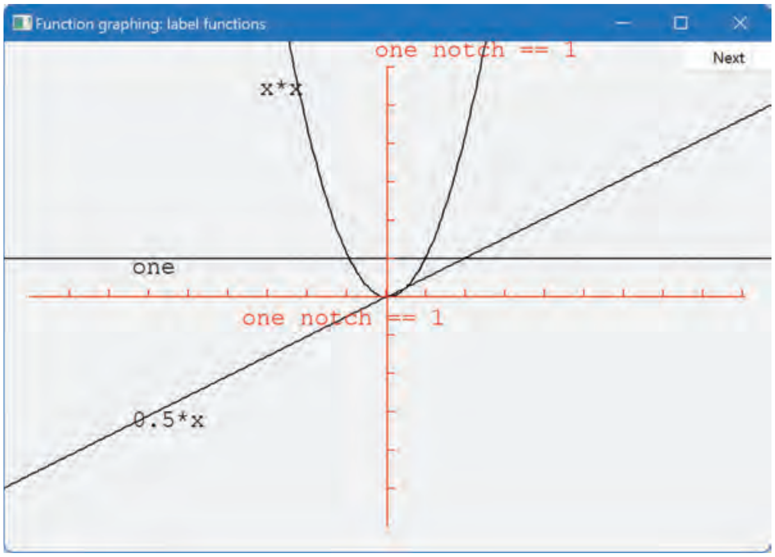
```
constexpr int xlength = xmax−40;  // make the axis a bit smaller than the window
constexpr int ylength = ymax−40;

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 (§13.6), you can of course do that instead.

To distinguish the axes from the data, we 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." AA

13.3 Function

The **Function** graphics interface class is defined like this:

```
using Fct = std::function<double(double)>;    // function taking a double argument and returning a double

struct Function : Open_polyline {
        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+round_to<int>(r∗xscale),xy.y−round_to<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 constructors, 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).

When trying to use **Fct** with one of the standard-library mathematical functions, say **cos**, we get an unpleasant surprise:

```
Function f3{ cos,r_min,r_max,orig,200,30,30 };          // error: can't deduce type of argument "cos"
```

The problem is that the standard library offers several cosine functions called **cos** so the compiler can't know which one we wanted. There are several ways to resolve this problem, but the simplest is to define a function that specifically does **cos** for the type we want, here **double**:

```
double dcos(double d) { return cos(d); }        // dcos() chooses cos(double)
// ...
Function f3{ dcos,r_min,r_max,orig,200,30,30 };
```

13.3.1 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 {dsqrt, 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 {dsqrt, 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. That would have been more work, 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: CC

```
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:

```
Function(Fct f, double r1, double r2, Point orig,
                    int count = 100, double xscale=25, double yscale=25);
```

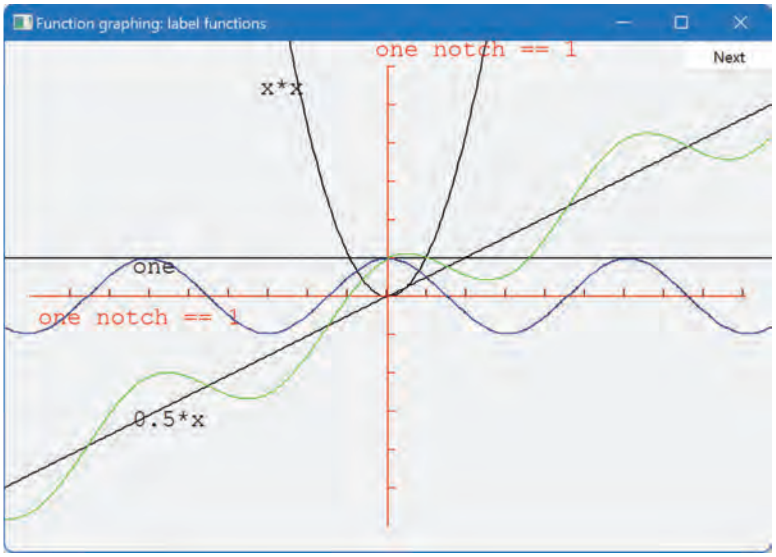
Sometimes, picking good default arguments is easy. Examples of that are the default for string (the empty string) and the default for vector (the empty vector). In other cases, such as Function, choosing a default is less easy; we found the ones we used after a bit of experimentation and a failed attempt. 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.

13.3.2 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 0.5∗x slope:

```
double sloping_cos(double x) { return cos(x)+slope(x); }
```

Here is the result:



The code is

```
Function s4 {dcos,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};
s5.set_color(Color::green);

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.

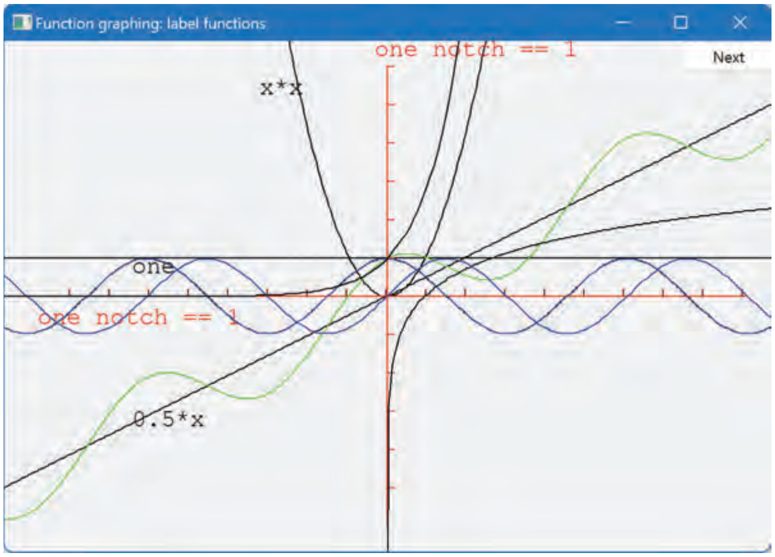
Finally, we graph a log, an exponential, a sine, and a cosine:

```
Function f1 {dlog,0.000001,r_max,orig,200,30,30};          // log() logarithm, base e
Function f2 {dsin,r_min,r_max,orig,200,30,30};             // sin()
f2.set_color(Color::blue);

Function f3 {dcos,r_min,r_max,orig,200,30,30};             // cos()
Function f4 {dexp,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.

This is messy and only an example of what we can do, rather than a recommendation of style. Rather than labeling those functions we used color.

The result is



13.3.3 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 that can be 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 using the suffix return type notation (§7.4.10):

```
Function s5 {[](double x) -> double { return cos(x)+slope(x); },r_min,r_max,orig,400,30,30};
```

In our `Function` example, we could have used lambdas, rather than named functions, to resolve the overloading problem (§13.3). For example:

```
Function f3{[](double d) { return cos(d); },r_min,r_max,orig,200,30,30 };        // use cos(double)
```

That would make sense if this was the only place we needed `cos(double)` but not if we needed it repeatedly.

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 AA 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 (§21.2.3).

13.4 Axis

We use Axis wherever we present data (e.g., §13.6.4) 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 {                    // representation left public
        enum Orientation { x, y, z };
        Axis(Orientation d, Point xy, int length, int nummber_of_notches=0, string label = "");

        void draw_specifics(Painter& painter) const override;
        void move(int dx, int dy) override;

        void set_color(Color c);

        Text label;
        Line line;
        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}, line{xy, (d==x) ? Point{xy.x+length,xy.y} : Point{xy.x,xy.y–length}}
{
        if (length<0) error("bad axis length");
        switch (d){
        case Axis::x:
                if (1<n) {                                  // add notches
                        int dist = length/n;
                        int x = xy.x+dist;
                        for (int i = 0; i<n; ++i) {
                                notches.add(Point{x,xy.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:
                if (1<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_specific(Painter& painter) const
{
        line.draw_specific(painter);    // the line
        notches.draw(painter);          // the notches may have a different color from the line
        label.draw(painter);            // the label may have a different color from the line
}
```

We use **draw()** rather than **draw_specific()** 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)
{
        line.set_color(c);
        notches.set_color(c);
        label.set_color(c);
        redraw();
}
```

Similarly, **Axis::move()** moves all the parts of the **Axis** together:

```
void Axis::move(int dx, int dy)
{
	line.move(dx,dy);
	notches.move(dx,dy);
	label.move(dx,dy);
	redraw();
}
```

13.5 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

$$e^x \equiv 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

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 e^x than the one before it. Here, `pow(x,n)` is the standard-library function that returns x^n. 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 exp_n(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 exp_n() is:

```
Function real_exp {exp,r_min,r_max,orig,200,x_scale,y_scale};
real_exp.set_color(Color::blue);
```

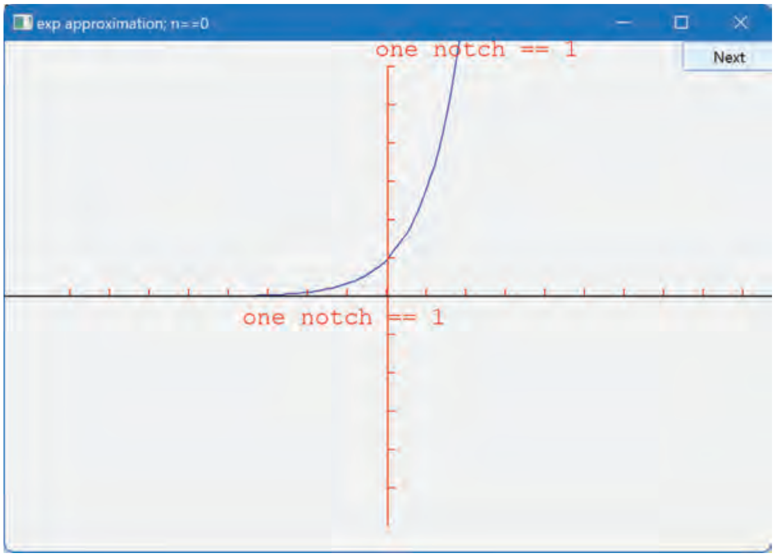
But how can we use exp_n()? From a programming point of view, the difficulty is that our graphing class, Function, takes a function of one argument and exp_n() needs two arguments. Given C++, as we have seen it so far, there is no really elegant solution to this problem. However, lambda expressions provide a way (§21.2.3). 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 exp_n(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 exp_n(x,n) gets its n when its Function is created and its x from each call from within the Function.

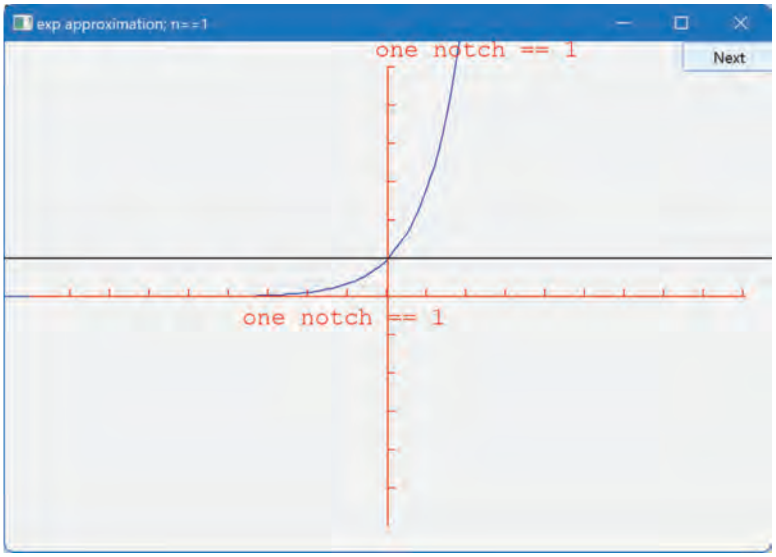
Note the final detach(e) in that loop. The scope of the Function object e is the block of the for-statement. Each time we enter that block we get a new Function called e, and each time we exit the block that e goes away, to be replaced by the next. The window must not remember the old e because it will 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:

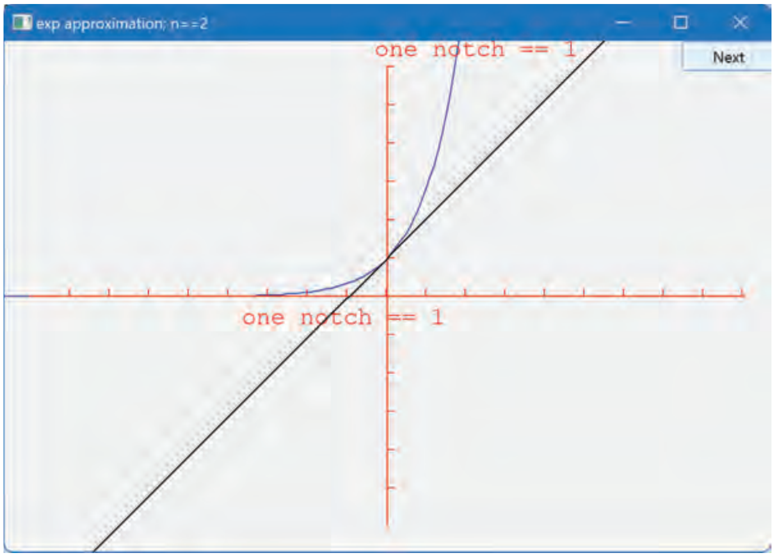


We see that **exp(0)** is **1** so that our blue "real exponential" crosses the *y* axis at **(0,1)**.

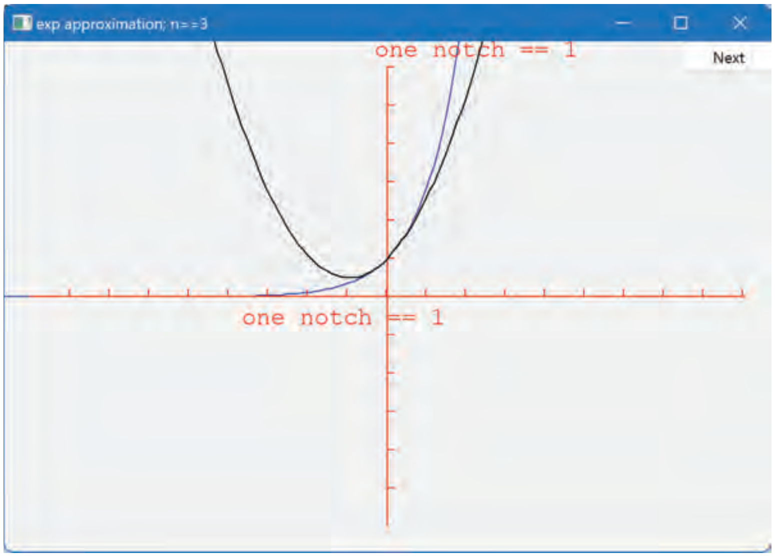
If you look carefully, you'll see that we actually drew the zero term approximation (**exp_n(x,0)==0**) as a black line right on top of the *x* axis. Hitting "Next," we get the approximation using just one term. Note that we display the number of terms used in the approximation in the window label:



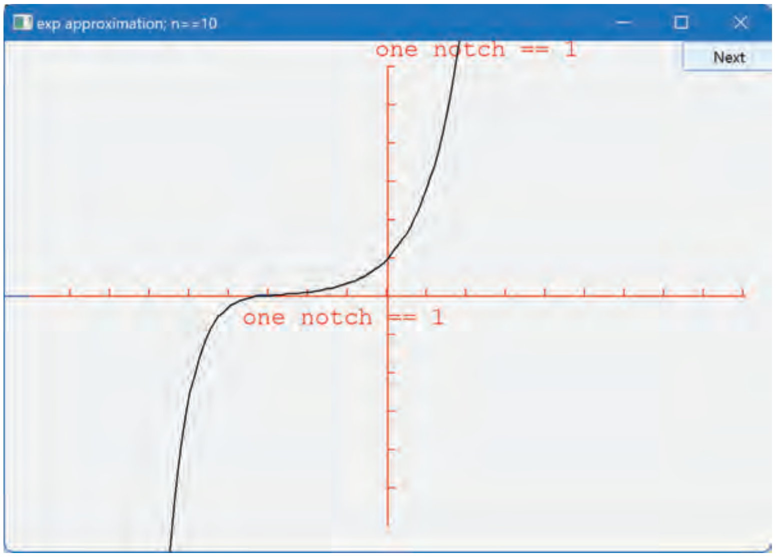
That's the function exp_n(x,1)==1, the approximation using just one term of the sequence. It matches the exponential perfectly at (0,1), but we can do better:



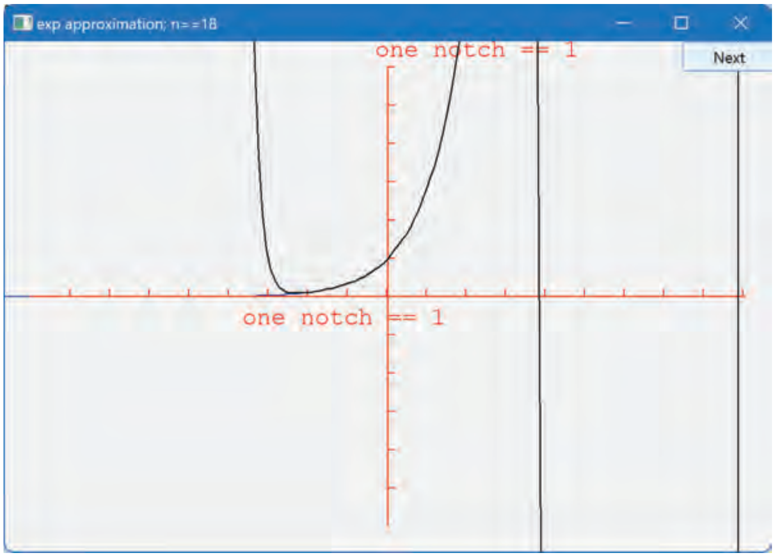
With two terms (1+x), we get the diagonal crossing the *y* axis at (0,1). With three terms (1+x+pow(x,2)/fac(2)), we can see the beginning of a convergence:



With ten terms we are doing rather well 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:



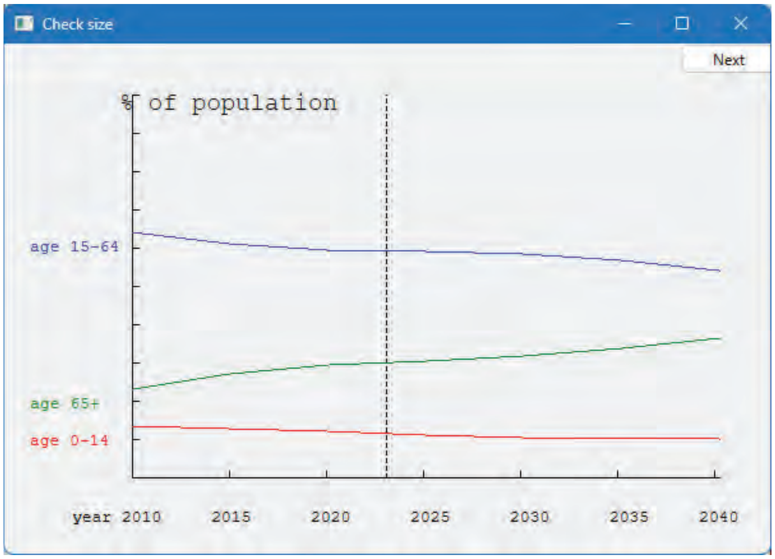
XX The computer's arithmetic is not pure math. Floating-point numbers are simply an approximation to real numbers for a fixed number of bits. An int overflows if you try to place a too-large integer in it, whereas a **double** stores an approximation. When I saw the strange output for larger numbers of terms, I first suspected that our calculation started to produce values that couldn't be represented as **double**s, 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 10 of Chapter 6.

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.

13.6 Graphing data

AA

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 some decades. The data to the right of the 2023 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 individual 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)`.

AA 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` (§11.6.1) 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` (§11.6.4) can be used to plot unconnected points.

Now, back to our Japanese age distribution example.

13.6.1 Reading a file

The file of age distributions consists of lines like this:

```
{ 2010 : 13.1 63.8 23.0 }
{2015 : 12.5 60.7 26.8}
{2020 : 11.7 59.2 29.1}
```

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;
        double 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 != '}')   // format error
                              is.clear(ios_base::failbit);
                    else
                              d = dd;
          }
          return is;
}
```

This is a straightforward application of the ideas from Chapter 9. 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.

First we need to open the file:

```
string file_name = "japanese-age-data.txt";
ifstream ifs {file_name};
if (!ifs) {
          Text err_label {Point{20,20},"Can't open file"};
          win.attach(err_label);
          win.wait_for_button();
          error("can't open ", file_name);
}
```

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 error message turned out to be less helpful as the window disappeared, so we added a helpful text before exiting. Really, we needed a better error function for graphics programs; see also §11.10.

Given **Distribution**, the read loop becomes

```
for (Distribution d; ifs >> d; ) {
        if (d.year < base_year || end_year < d.year)
                error("year out of range");

        double all = d.young + d.middle + d.old;
        if (all−100 > 1.5 || 100−all>1.5 )          // take rounding errors into account
                error("percentages don't add up");

        // ... use the data ...
}
```

The read loop checks that the year read is in the expected range and that the percentages add up to about 100. That's a basic sanity check for the data. Date is often "dirty" so we always need to check. Since `>>` checks the format of each individual data item, we didn't bother with further checks in the main loop.

13.6.2 General layout

So what do we want to appear on the screen? You can see our answer at the beginning of §13.6. 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.

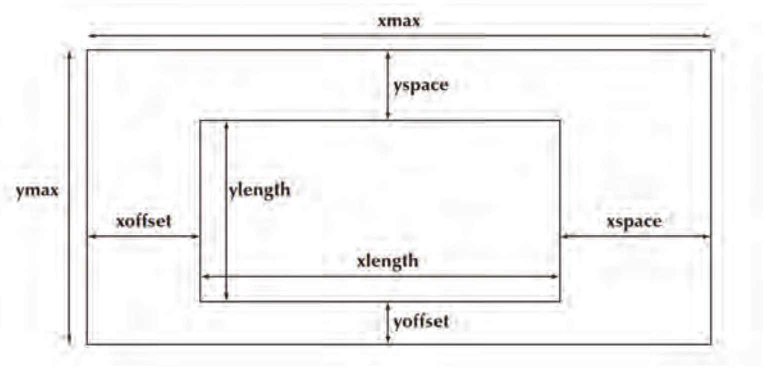
We want to label the *x* axis with the years. The vertical line through the year 2023 indicates where the graph goes from hard data to projected data.

We decided to just use the window's label as the title for our graph.

AA 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:



Without such a "schematic view" of where things are in our window and the symbolic constants that define it, we tend to get lost and become frustrated when our output doesn't reflect our wishes. **AA**

13.6.3 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 = 2010;
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 (§7.4.7).

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()(double v) const { return cbase+(v−vbase)∗scale; }          // see §21.2
};
```

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.

13.6.4 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:

```
Simple_window win {Point{100,100},xmax,ymax,"Aging Japan"};

Axis x { Axis::x, Point{xoffset,ymax − yoffset}, xlength, (end_year−base_year)/5,    // one notch per 5 years
             "year "
             "2010     2015     2020     2025     "
             "2030     2035     2040"
      };
x.label.move(−100,0);
x.label.set_font_size(10);

Axis y {Axis::y, Point{xoffset,ymax−yoffset}, ylength, 10,"% of population"};

int now = 2023;
Line current_year {Point{xs(now),ys(0)},Point{xs(now),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 five years, and the exact number of notches is calculated from **base_year** and **end_year** so that if we change that range, the axis will 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 "
"2010     2015     2020     2025     "
"2030     2035     2040"
```

Adjacent string literals are concatenated by the compiler, so that's equivalent to

```
"year 2010     2015     2020     2025     2030     2035     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_polylines and fill them in the read loop:

```
Open_polyline children;
Open_polyline adults;
Open_polyline aged;

for (Distribution d; ifs>>d; ) {
        // ... data validation ...
        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:

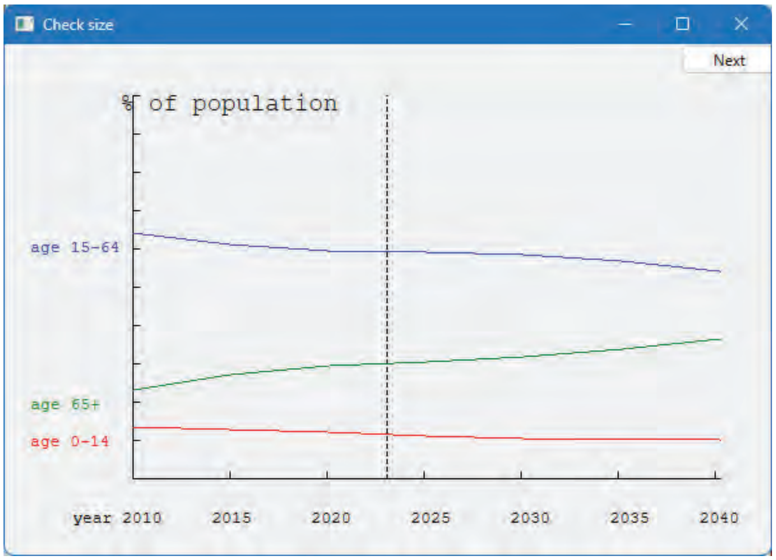
```
Text children_label {Point{20,children.point(0).y},"age 0–14"};
children.set_color(Color::red);
children_label.set_color(Color::red);
children_label.set_font_size(10);
children_label.set_style(Line_style::dash);

Text adults_label {Point{20,adults.point(0).y},"age 15–64"};
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);
adults_label.set_font_size(10);
Text aged_label {Point{20,aged.point(0).y},"age 65+"};
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
aged_label.set_font_size(10);
aged_label.set_style(Line_style::dashdotdot);
```

Finally, we need to attach the various Shapes to the Window and start the GUI system (§12.2.3):

```
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);

win.wait_for_button();
```

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

Graphing drill:

[1] Make an empty 600-by-600 Window labeled "Function graphs."

[2] 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).

[3] Make both axes red.

[4] 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).

[5] Change it to use *x* scale 20 and *y* scale 20.

[6] From now on use that range, scale, etc. for all graphs.

[7] Add double slope(double x) { return 0.5*x; } to the window.

[8] Label the slope with a Text "0.5x" at a point just above its bottom left end point.

[9] Add double square(double x) { return x*x; } to the window.

[10] Add a cosine to the window (don't write a new function).

[11] Make the cosine blue.

[12] Write a function sloping_cos() that adds a cosine to slope() (as defined above) and add it to the window.

Class definition drill:

[13] Define a **struct Person** containing a **string** name and an **int** age.

[14] Define a variable of type **Person**, initialize it with "Goofy" and 63, and write it to the screen (**cout**).

[15] 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**).

[16] Give **Person** a constructor initializing **name** and **age**.

[17] Make the representation of **Person** private, and provide **const** member functions **name()** and **age()** to read the name and age.

[18] Modify **>>** and **<<** to work with the redefined **Person**.

[19] 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.

[20] Read a sequence of **Person**s from input (**cin**) into a **vector<Person>**; write them out again to the screen (**cout**). Test with correct and erroneous input.

[21] 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 expression	screen layout	**Axis**	overflow

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 §13.5 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 §13.5) the series 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.

14

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 Web sites. 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 the code available on the Web, which uses features and techniques presented in Chapter 15 and Chapter 16. Here we focus on usage.

14.1 User-interface alternatives

CC 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** (Chapter 9) provide suitable and convenient mechanisms for this. If graphical output is needed, we can use a graphics display library (as shown in Chapter 10 – Chapter 14) 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 or phone 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 webbrowser interface*: For that, we need to use a markup (layout) language, such as HTML, and usually a scripting language, such as JavaScript, Python, and PHP. 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.

AA 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. Our Graphics/GUI library can run directly on a phone or a computer as well as in a browser.

14.2 The “Next” button

How did we provide that “Next” button that we used to drive the graphics examples in Chapter 10? 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 we could 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 Chapter 10 – Chapter 13 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
```

CC

The implementations of 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 stop and wait for the user to respond; 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. This is called *control inversion* (see §14.5.1).

For starters, we just want to say, “Please wake me up when someone clicks my button”; that is, “Please stop executing my program until someone clicks the mouse button and the cursor is in the rectangular area where the image of my button is displayed. Then wake me up” This is just about the simplest action we could imagine. However, such an operation isn’t directly provided by “the system'' so we wrote one ourselves. Seeing how that is done is the first step in understanding GUI programming.

14.3 A simple window

CC 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 : Window {
        Simple_window(Point xy, int w, int h, const string& title );
        ˜Simple_window() {}
        void wait_for_button();
private:
        Button next_button;
};
```

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 Qt) connects our notion of a window with the system’s window implementation. For details of Window’s implementation, see Window.h on www.stroustrup.com/programming.html.

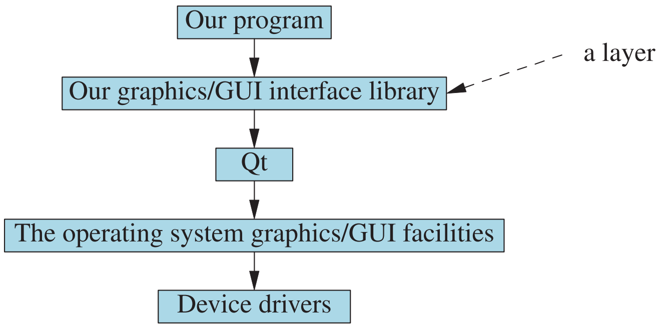
Our button is initialized in Simple_window’s constructor:

```
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", []{})
{
        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 an action []{}. The first four parameters are exactly parallel to what we do for a Window: we place a rectangular shape on the screen and label it.

The []{} is the minimal action. It is a lambda (§13.3.3, §21.2.3) indicating nothing is to be done. That implies that the processing simply continues after the button has been pressed (‘‘clicked’’). An action defined in our program that is invoked by the system in response to some user-action (e.g., “clicking” a button) is call a *callback*. That is the system calls back into our program. Here, **[]()** indicates “ do nothing and proceed.” Most callbacks do more.

Before showing that code, let’s consider what is going on here:


Our program
Our graphics/GUI interface library
a layer
Qt
The operating system graphics/GUI facilities
Device drivers

Our program runs on top of several *layers* of software. It directly uses our graphics/GUI library that we implemented using Qt. Qt in turn is implemented using operating system interfaces. In the operating system there are more layers until we reach the device drivers that put pixels on our screen and keep track of what our mouse is doing. Such layering helps us manage our complex system by letting us focus on one – or only a few – layers at a time.

14.3.1 A wait loop

So, in this – our simplest – case, what do we want done by **Simple_window** each time the button is “pressed”? We want to stop the execution of our program to give us a chance to see what we have done so far. Then, the program should wait for us to press the button to proceed:

```
// ... 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 because we rely on Qt facilities, rather than the raw system:

```
void Simple_window::wait_for_button()          // wait for button to be pushed
{
     get_impl().wait_for_button(&next_button);      // pass &next_button to the Window’s implementation
}
```

CC

This looks simple, but there is significant complexity hidden here. For example, the code is portable across different operating systems, including MacOS, iOS, Linux, Android, and Windows. In our interface library, the Qt specific code is “hidden” in a class **WidgetPrivate** that is part of the implementation of our **Window** class.

Like most GUI systems, Qt provides a function that suspends (stops) a program until something happens. The Qt version is called **exec()** and it wakes up our program whenever anything that our program has expressed interest in happens. In this case, by **attaching** **next_button** to the **window** we expressed interest in “clicks” on the button. So, when someone clicks our “Next” button, **exec()** calls our **[]()** action and returns (to wait for more events).

14.4 Button and other Widgets

We define a Button like this:

```
using Callback = std::function<void()>;          // a callback takes no argument and returning nothing

struct Button : Widget {
        Button(Point xy, int w, int h, const string& label, Callback cb);
        void attach(Window&);
};
```

CC 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.

14.4.1 Widget

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 a QWidget - it is *not* a QWidget
        // We keep our interface classes at arm's length from Qt
public:
        Widget(Point xy, int w, int h, const string& s, Callback cb);

        Widget& operator=(const Widget&) = delete;          // don't copy Widgets
        Widget(const Widget&) = delete;

        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;                                      // the action

        virtual ~Widget();

        WidgetPrivate& get_impl() const { return *impl; }
private:
        std::unique_ptr<WidgetPrivate> impl;
};
```

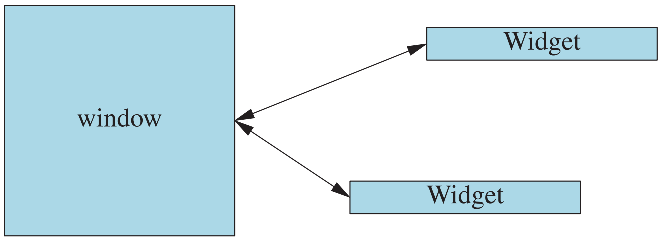
A Widget is similar to a Shape but differs in being able to perform actions involving users. It is thus a somewhat more complex mechanism with deeper integration with the GUI engine. Therefore, despite similarities in interface and use, a Widget is not a Shape or vice versa.

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 §14.5.2):

- hide() makes the Widget invisible.
- show() makes the Widget visible.

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 (§12.3.5): every class derived from Widget must define what it means for it to be attached to a Window. In fact, it is in Window::attach() that the system-level widgets are created. The Widget::attach() and Shape::attach() functions are 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 §12.5, 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. However, the alternative would have been to add 10 access functions with essentially no semantic benefits. The impl members are a Widget's link to its Qt implementation. Users of user-interface class Widget have no business messing with its implementation, so we keep it private. A Widget is an interface to a unique object that cannot be copied – it represents an area on the screen and the action associated with it – so we delete its copy operations (§12.4.1).

14.4.2 Button

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);
     void attach(Window&) {}
};
```

```
Button::Button(Point xy, int w, int h, const string& label, Callback cb)
        :Widget(xy,w,h,label,cb)
{
        WidgetPrivate& w_impl = get_impl();
        QPushButton* button = new QPushButton();
        w_impl.widget = button;
        button->setText(QString::fromStdString(label));
        QObject::connect(button, &QPushButton::clicked, [this]{ do_it(); });
}
```

That's all. The constructor contains all the (relatively) messy code connecting to Qt. For now, please just note that defining a simple **Widget** isn't particularly difficult. The call of **connect()** it the most interesting: It says that when someone clicks the button the callback **do_it()** is invoked.

AA 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.

14.4.3 In_box and Out_box

We provide a **Widget** for getting text into our program:

```
struct In_box : Widget {
        In_box(Point xy, int w, int h, const string& s, Callback cb);

        int get_int();
        string get_string();

        int get_int_keep_open();
        string get_string_keep_open();

        void attach(Window& win) override;

        void dismiss();
        void hide() override;
        void show() override;
        void hide_buttons();
        void show_buttons();

        enum State {idle, accepted, rejected};
        State last_result();
        void clear_last_result();
        string last_string_value();
        int last_int_value();
        struct ResultData {
                State state = idle;
                string last_string;
                int last_int = 0;
        };
private:
        ResultData result;
        bool waiting = false;
};
```

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(). We can indicate that we are finished typing into the In_box by hitting keyboard return/enter key. Alternatively, we can bring up a button for the user to click when finished (using show_buttons()). A box that accepts input and generates output is often called a *dialog box*.

Our In_box is a bit complicated – apparently unnecessarily so – because it is written to support variations of dialog boxes that are beyond the scope of this book, such as pop-up dialog boxes. That's a common phenomenon in widely used libraries: they are written to serve many users, and many of those users have needs that differ from our current needs. We should be tolerant of such complexity because we might be among those "other users" next year or later.

An Out_box is used to present some message to a user. In analogy to In_box's get functions, we can put either integers or strings into an Out_box:

```
struct Out_box : Shape {
        enum Kind { horizontal, vertical };
        Out_box(Point p, const string& s, Kind k = horizontal);

        void set_parent_window(Window* win) override;

        void put(int);
        void put(const string&);
        void draw_specific(Painter& painter) const override;

        Text label;
        Text data;
        Kind orientation;               // does the Label come before or above the data?
};
```

An Out_box is far simpler than an In_box because it doesn't require interaction with the user, it simply displays a value. Therefore, an Out_box is just a Shape, whereas an In_box is a Widget.

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 (§9.11). AA

§14.5 gives examples of the use of In_box and Out_box.

14.4.4 Menu

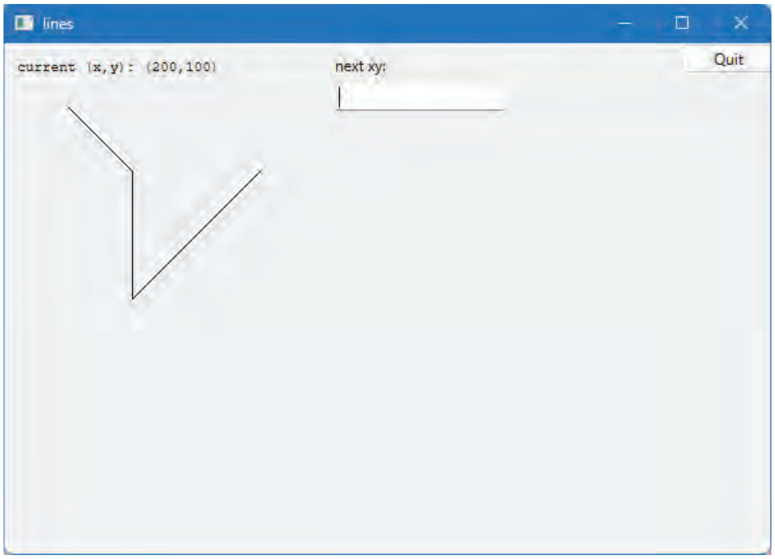
We offer a very simple notion of a menu:

```
struct Menu : Widget {
        enum Kind { horizontal, vertical };
        Menu(Point xy, int w, int h, Kind kk, const string& label);

        using Window::attach;                   // attach the menu to the window

        int attach(Button& b);                  // attach a named buton to the menu
        int attach(unique_ptr<Button> p);       // attach an unnamed button to the menu

        void show()                             // show all buttons
        {
                for (auto&& x : selection)
                        x–>show();
        }

        void hide()                             // hide all buttons
        {
                for (auto&& x : selection)
                        x–>hide();
        }

        void move(int dx, int dy)               // move all buttons
        {
                for (auto&& x : selection)
                        x–>move(dx,dy);
        }

private:
        Vector_ref<Button> selection;
        Kind k;
        int offset;

        void layoutButtons(Button& b);          // horizontal or vertical
        void layoutMenu();                      // tells the Widget where the Menu is in the Window
};
```

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 §14.5 and §14.5.2. 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** (§11.7.3). If you want a “pop-up” menu, you have to make it yourself; see §14.5.2.

14.5 An example: drawing lines

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; §11.6.1) specified as a sequence of coordinate pairs. The idea is that the user repeatedly enters (*x*, *y*) coordinates in the "next_xy" box For example, `100,200`. Note that the comma is required and no parentheses are accepted. After each coordinate pair, the user hits return/enter keyboard key. The window above shows the result after entering four coordinate pairs: *(50,50)*, *(100,100)*, *(100,200)*, *(200,100)*.

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.

Let's define a class for representing such windows. It is pretty straightforward:

```
struct Lines_window : Window {
        Lines_window(Application& application, Point xy, int w, int h, const string& title);
        Open_polyline lines;
        void wait_for_button();
private:
        Application* app = nullptr;
        Button quit_button;               // a Widget
        In_box next_xy;                   // a Widget
        Out_box xy_out;                   // a Shape
        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(Application& application, Point xy, int w, int h, const string & title)
        : Window{ xy,w,h,title },
                app(application),
                quit_button{ Point{x_max() − 70,0}, 70, 20, "Quit", [this]() { quit(); } },
                next_xy{ Point{250,0}, 50, 20, "next xy:", [this]() { next(); } },
                xy_out{ Point{10,10}, "current (x,y): " }
{
        attach(lines);
        attach(quit_button);

        next_xy.hide_buttons();           // a Qt input box comes with buttons; we decided to hide them
        attach(next_xy);
        next_xy.show();                   // but we do want the box itself to show

        xy_out.label.set_font_size(8);    // use a smaller than default font
        xy_out.data.set_font_size(8);
        attach(xy_out);
}
```

AA That is, each **Widget** and **Shape** is constructed and then attached to the window. 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.

The "Quit" button deletes the **Window** using Qt facilities:

```
void Lines_window::quit()
{
        end_button_wait();        // don't wait anymore
        next_xy.dismiss();        // clean up
        app−>quit();
}
```

All the real work is done in the **next_xy** button's **next()**: it reads a pair of coordinates, updates the **Open_polyline**, and updates the position readout:

```
void Lines_window::next()           // the action performed by next_xy when woken up
{
        if (next_xy.last_result() == In_box::accepted) {      // check if the value has changed
                string s = next_xy.last_string_value();        // read coordinate pair
                istringstream iss{ s };
                int x = 0;
                char ch = 0;
                int y = 0;
                iss >> x >> ch >> y;
                lines.add(Point{ x,y });

                ostringstream oss;                             // update current position readout
                oss << '(' << x << ',' << y << ')';
                xy_out.put(oss.str());
        }
        next_xy.clear_last_result();                           // clear the box
}
```

We use an **istringstream** (§9.11) to read the integer coordinates from input and an **ostringstream** (§9.11) to format the string to be put into the **Out_box**. If a user enters something that isn't an integer coordinate value, we just default to zero. The test is there to handle the case where a user hits ESC rather than enter/return.

So what's odd and different about this program? Let's see its **main()**:

```
#include "GUI.h"

int main()
try {
        Application app;                                              // create a GUI application
        Lines_window win {app,Point{100,100},600,400,"lines"};        // our window
        return app.gui_main();                                        // start the GUI application
}
catch(exception& e) {
         cerr << "exception: " << e.what() << '\n';
         return 1;
}
catch ( ...) {
         cerr << "Some exception\n";
         return 2;
}
```

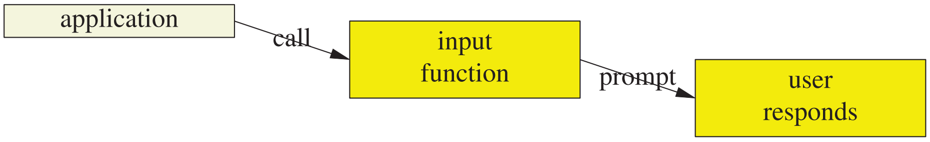
There is basically nothing there! The body of **main()** is just the definition of our window, **win**, preceded with the request to create a Qt GUI application and succeeded by a call to give control to the GUI system. There is not another function, **if**, **switch**, or loop – nothing of the kind of code we saw in Chapter 5 and Chapter 6 – just a definition of a variable and a call to the GUI system's main loop, **gui_main()**, which is simply the infinite loop.

Except for a few implementation details we have seen all of the code that makes our "lines" program run. We have seen all of the fundamental logic. So what happens?

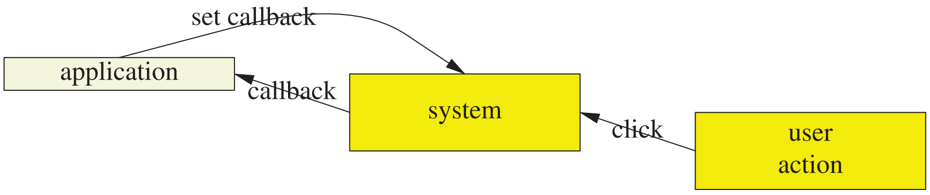
14.5.1 Control inversion

CC 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, a call app.exec() tells "the system" to look out for the widgets and invoke the appropriate callbacks. In theory, app.exec() could tell you, the programmer, which widget requested attention and leave it to you to call the appropriate function. However, in Qt and most other GUI systems, app.exec() 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:



XX 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. The techniques for dealing with GUI testing 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 §14.5. There, the Lines_window's next() function, invoked by pressing the "Next point" button, reads data from the In_box (next_xy) 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.

14.5.2 Adding a menu

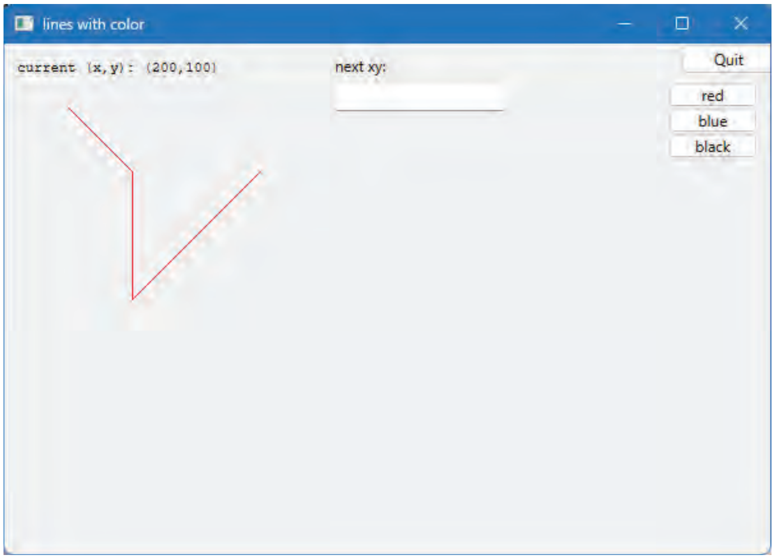
Let's explore the control and communication issues raised by "control inversion" by providing a menu for our "lines" program. First, we'll simply provide a menu that allows the user to change the color of all lines in the lines member variable. We add the menu color_menu and its callbacks:

```
struct Color_window : Lines_window {
        Color_window(Application& app, Point xy, int w, int h, const string& title);
        void change(Color c) { lines.set_color(c); }
private:
        Button menu_button;
};
```

Having defined the color_menu member, we need to initialize it:

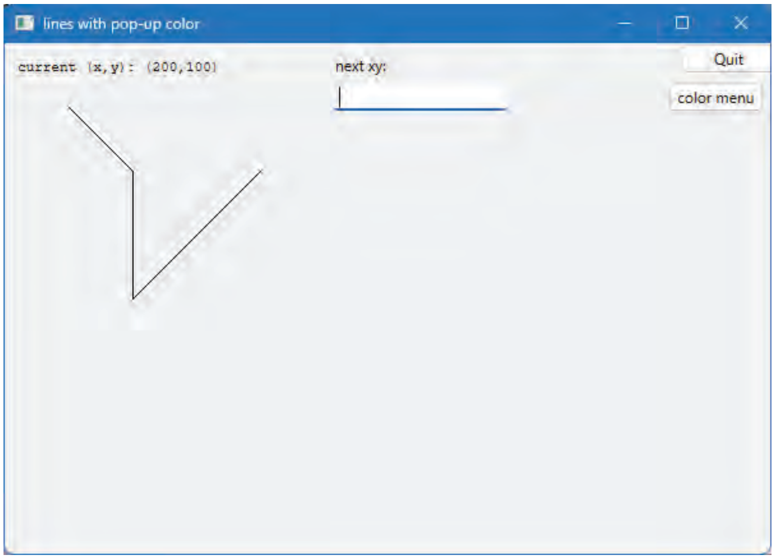
```
Color_window::Color_window(Application& app, Point xy, int w, int h, const string& title)
        : Lines_window{ app,xy,w,h,title },
                color_menu{ Point{x_max() – 70,40},70,20,Menu::vertical,"color" }
{
        color_menu.attach(make_unique<Button>( Point{0,0},0,0,"red", [&] { change(Color::red); } ));
        color_menu.attach(make_unique<Button>( Point{0,0},0,0,"blue",[&] { change(Color::blue); } ));
        color_menu.attach(make_unique<Button>( Point{0,0},0,0,"black",[&] { change(Color::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. After entering a few coordinate pairs and pressing "red", we get:

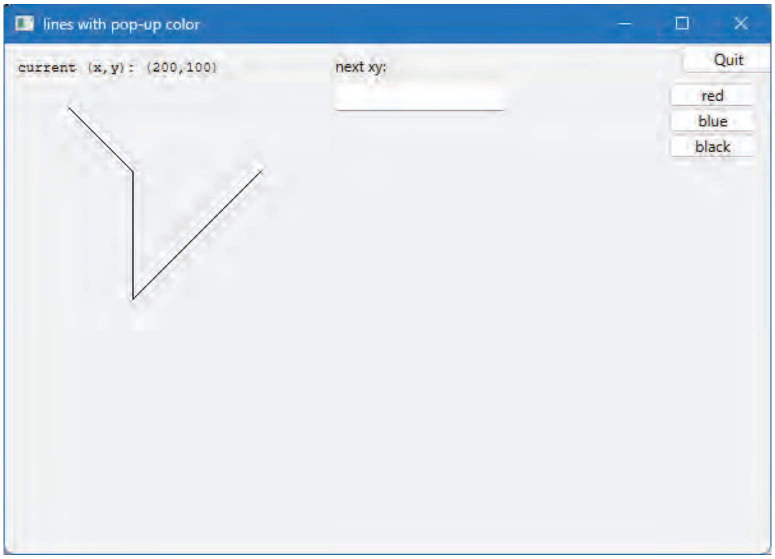


Having played with this for a while, we decided that what we really wanted was a “pop-up menu”; that is, we didn’t want to spend precious screen space on a menu except when we are using it. So, we added a “color menu” button; press that, and up pops the color menu. When we have made a selection, the menu is again hidden, and the button appears.

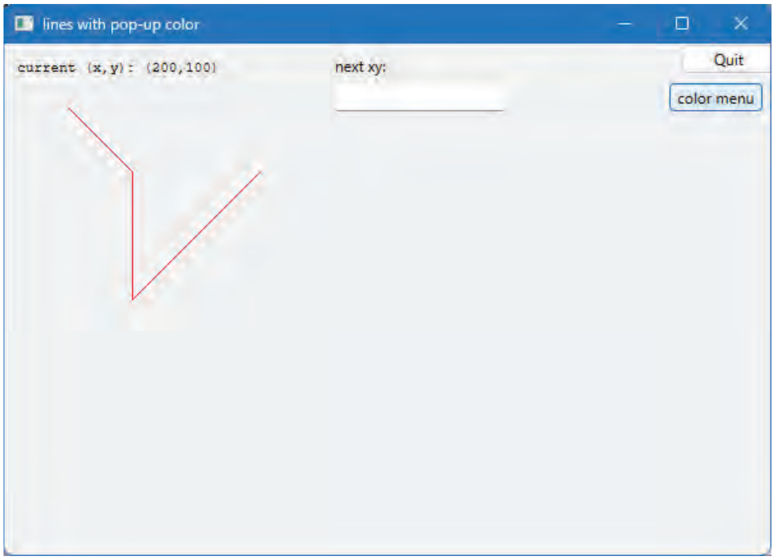
Here first is the window after we have added a few lines:



We see the new “color menu” button and some 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 "red" and we get



The lines are now red 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 `Color_window` after all of our modifications:

```
struct Color_window : Lines_window {
        Color_window(Application& app, Point xy, int w, int h, const string& title);
private:
        void change(Color c) { lines.set_color(c); }
        void hide_menu() { color_menu.hide(); menu_button.show(); }

        Button menu_button;
        Menu color_menu;
};

Color_window::Color_window(Point xy, int w, int h, const string& title)
        :Lines_window{ xy,w,h,title },
        menu_button{ Point{x_max() – 80,30}, 80, 20, "color menu",
                [&] { menu_button.hide(); color_menu.show(); } },
        color_menu{ Point{x_max() – 70,40},70,20,Menu::vertical,"color" }
{
        attach(color_menu);
        color_menu.attach(make_unique< Button>(Point{0,0},0,0,"red",
                [&] { change(Color::red); hide_menu(); }));
        color_menu.attach(make_unique< Button>(Point{0,0},0,0,"blue",
                [&] { change(Color::blue); hide_menu(); }));
        color_menu.attach(make_unique< Button>(Point{0,0},0,0,"black",
                [&] { change(Color::black); hide_menu(); }));
        attach(menu_button);
        hide_menu();
    }
```

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.

14.6 Simple animation

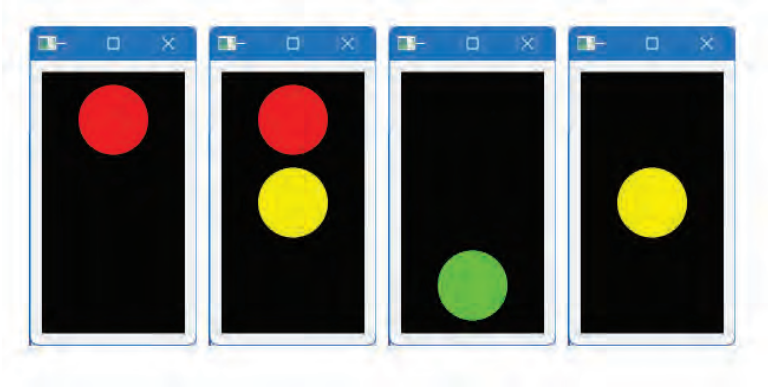
Buttons, Texts, and Menus are all very good but they are static: nothing moves! Well, we can load images that have some movement, but with the facilities presented so far, the best we can do is for something to move whenever we click a button. However, Window offers a mechanism to wait for a while and then redraw. If we take the opportunity to change what's on that window before that redraw, we can have movement. Here is a simple traffic light that changes every two seconds. First we define our window:

```
Window w{140, 240, "Traffic light"};

Rectangle r{{10, 10}, 120, 220};
r.set_fill_color(Color::black);

Circle red {{70, 50}, 30};
Circle amber {{70, 120}, 30};
Circle green {{70, 190}, 30};
```

Attach the four Shapes and we have a static Window. However, we can now add actions:

```
const int second = 1000;          // 1000 milliseconds == 1 second; the timer counts milliseconds
const int yellow_delay = 10*second;
const int red_green_delay = 120*second;

while (true)
    for (int i = 0; i < 3; ++i) {
        red.set_fill_color(Color::red);
        w.timer_wait(red_green_delay);

        amber.set_fill_color(Color::yellow);
        w.timer_wait(yellow_delay);

        red.set_fill_color(Color::black);
        amber.set_fill_color(Color::black);
        green.set_fill_color(Color::green);
        w.timer_wait(red_green_delay);
        amber.set_fill_color(Color::yellow);
        green.set_fill_color(Color::black);
        w.timer_wait(yellow_delay);

        amber.set_fill_color(Color::black);
    }
```

Now the traffic light slowly circles from red, to red and yellow, to green, to yellow:



The w.timer_wait(yellow_delay) simply waits for 10,000 milliseconds (ten seconds):

```
void Window::timer_wait(int milliseconds)
{
        impl–>timer_wait(milliseconds);
}
```

You can also add an action ("callback") as an argument. That action is invoked after the delay.

```
void Window::timer_wait(int milliseconds, std::function<void()> cb)
{
        impl–>timer_wait(milliseconds, cb);
}
```

One of the nice things about the callback model is that "the system" still looks out for other user actions while the animation runs. That way, Buttons, Menus, and such still work.

The traffic light example was the simplest we could think of but we are confident that you can think of more interesting ones. Remember that an application can have many windows, so we can have several independent animations running simultaneously. Also, between timer_wait()s, we can update several different "animated objects."

14.7 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()
{
        Application app;
        Lines_window {app,Point{100,100},600,400,"lines"};
        app.gui_main();
}
```

AA 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

XX 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 §14.5):

```
int main()
{
        Application app;
        Lines_window win {app,Point{100,100},600,400,"lines"};
        app.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.

XX 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.

XX 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++).

AA Common problems found during debugging include **Shape**s and **Widget**s 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:

```
void load_disaster_menu(Menu& m)      // helper function for loading buttons into a menu
{
        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. 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. Compilers can and should catch such errors, but unfortunately not all do. The solution is to use unnamed objects created by make_unique instead of named local objects:

```
void load_disaster_menu(Menu& m)
        // helper function for loading buttons into a menu
{
        Point orig {0,0};
        m.attach(make_unique<Button>(orig,0,0,"flood",cb_flood));
        m.attach(make_unique<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 Qt (www.stroustrup.com/programming.html).
[2] Using the facilities of Graph_lib, type in the line-drawing program from §14.5 and get it to run.
[3] Modify the program to use a pop-up menu as described in §14.5.2 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 a callback?
[6] What is a widget?
[7] Is Qt an acronym?
[8] How do you pronounce Qt?
[9] What other GUI toolkits have you heard of?
[10] Which systems use the term *widget* and which prefer *control*?
[11] What are examples of widgets?
[12] When would you use an inbox?
[13] What is the type of the value stored in an inbox?
[14] When would you use a button?
[15] When would you use a menu?
[16] What is control inversion?
[17] What is the basic strategy for debugging a GUI program?
[18] Why is debugging a GUI program harder than debugging an “ordinary program using streams for I/O”?
[19] How do you animate a widget?

Terms

Button	dialog box	visible/hidden	callback
GUI	waiting for input	console I/O	GUI I/O
menu	wait loop	control	software
layer	widget	control inversion	user interface
browser I/O	**Application**	Qt	**In_box**
Out_box	animation	**timer_wait()**	

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 **PPP__support** to pick a new location for the “image button”:

```
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. Hint: `chrono::now()`, `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 6 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.

AA

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

Part III focuses on the C++ standard library's containers and algorithms framework (often referred to as the STL). It shows how containers (such as **vector**, **list**, and **map**) are implemented and used. In doing so, we introduce pointers, arrays, dynamic memory, exceptions, and templates. As part of that, we introduce the fundamental principles and techniques of generic programming. We also demonstrate the design and use of standard-library algorithms (such as **sort**, **find**, and **inner_product**).

15

Vector and Free Store

Use vector *as the default!*
– Alex Stepanov

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

15.1 Introduction

CC The most useful container in the C++ standard library is vector. A vector provides a sequence of elements of a given type. You can refer to an element by its index (subscript), extend the vector by using push_back() or resize(), ask a vector for the number of its elements using size(), and have access to the vector checked against attempts to access out-of-range elements. The standard library vector is a convenient, flexible, efficient (in time and space), statically type-safe container of elements. The standard string has similar properties, as have other useful standard container types, such as list and map, which we describe in Chapter 20. However, a computer's memory doesn't directly support such useful types. All that the hardware *directly* supports is sequences of bytes. For example, for a vector<double>v, the operation v.push_back(2.3) adds 2.3 to a sequence of 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 three chapters, we show how to build a Vector from the basic language facilities available to every programmer. Gradually, our Vector approximates the standard-library vector. This approach allows us to illustrate useful concepts and programming techniques, and to see how they are expressed using C++ language features. The language facilities and programming techniques we encounter in our Vector implementation are generally useful and very widely used.

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

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

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

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

- *Chapter 18*: How can we have **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. We present a **Vector** that approximates the standard-library **vector** and also the resource-management pointers **unique_ptr** and **shared_ptr**.

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

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

CC

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

XX

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

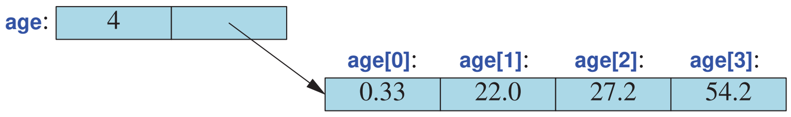
15.2 vector basics

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

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

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

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

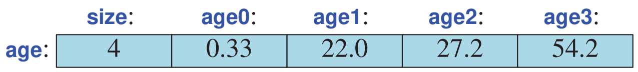


How do we make this "graphical design" real in a computer's memory? How do we get the values stored and accessed like that? Obviously, we have to define a class and we want to call this class **Vector**. Furthermore, it needs a data member to hold its size and one to hold its elements. But how do we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

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

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

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



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

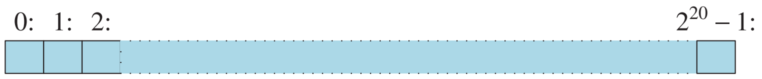
```
class Vector {          // a very simplified vector of doubles (like vector<double>)
        int sz;                         // the size
        double* elem;                   // pointer to the first element (of type double)
public:
        Vector(int s);                  // constructor: allocate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

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

15.3 Memory, addresses, and pointers

CC

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



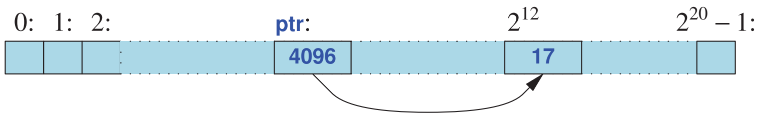
Everything we put in memory has an address. For example:

```
int var = 17;
```

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

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

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



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

Each type has a corresponding pointer type. For example:

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

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

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

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

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

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

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

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

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

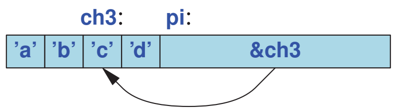
Similarly, a pointer to char (a char*) is not a pointer to int (an int*). For example:

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

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

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

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



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

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

15.3.1 The sizeof operator

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

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

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

TRY THIS

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

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

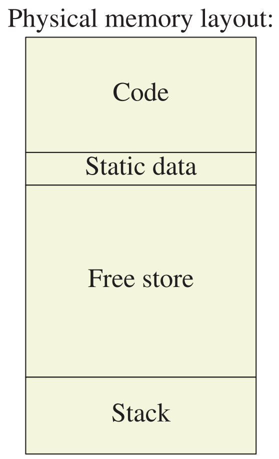
The output will be something like

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

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

15.4 Free store and pointers

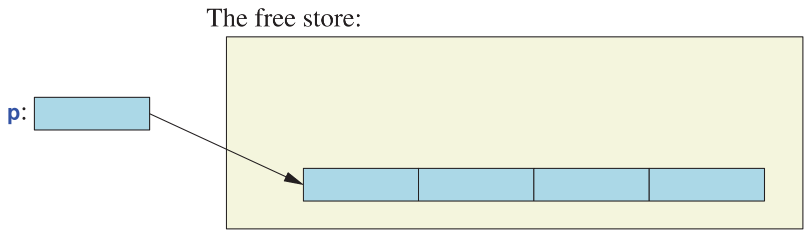
CC Consider the implementation of **Vector** from the end of §15.2. From where does the **Vector** get the space for the elements? How do we get the pointer **elem** to point to them? When you start a C++ program, the compiler sets aside memory for your code (sometimes called *code storage* or *text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



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

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

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



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

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

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

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

15.4.1 Free-store allocation

CC

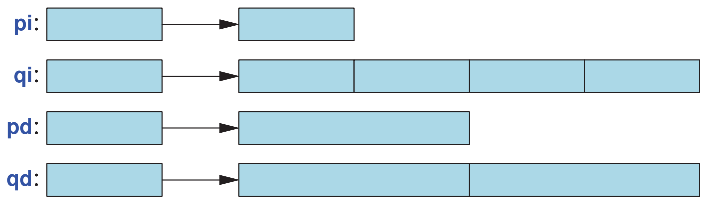
We request memory to be *allocated* on the *free store* by the **new** operator:

- 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 (this is the root cause of many problems).

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

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

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

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



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

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

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

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

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

15.4.2 Access through pointers

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

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

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

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

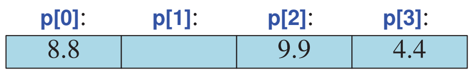
A pointer points to an object in memory. The "contents of" operator (also called the *dereference* operator) allows us to read and write the object pointed to by a pointer **p**:

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

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

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

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



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

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

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

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

15.4.3 Initialization

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

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

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

```
*p0 = 7.0;
```

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

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

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

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

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

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

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

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

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

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

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

15.4.4 The null pointer

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

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

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

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

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

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

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

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

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

15.4.5 Free-store deallocation

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

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

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

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

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

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

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

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

15.5 Destructors

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

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

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

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

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

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

};

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


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

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

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

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

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

15.5.1 Generated destructors

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

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

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

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

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

15.5.2 Virtual destructors

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

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

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

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

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

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

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

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

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

TRY THIS

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

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

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

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

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

XX Also, please remember that a "naked" **new** outside a constructor is an opportunity to forget to **delete** the object that **new** created, thus creating a potentially serious resource leak. "Naked" **new**s and “naked” **deletes** are sources of serious errors. Instead, use resource-management classes, such as **vector**, **unique_ptr** (§18.5.2), or **Vector_ref** (§11.7.3). Keep **news** in constructors and **deletes** in destructors (and in related resource-management functions (§17.4)).

15.6 Access to elements

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

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

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

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

Both **get()** and **set()** access the elements using the **[ ]** operator on the **elem** pointer: **elem[n]**.
Now we can make a **Vector** of **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';
}
```

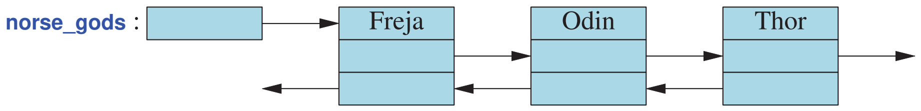
This will output

```
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 start small and simple and then grow our programs step by step, testing along the way. This strategy of growth and repeated testing minimizes errors and debugging.

15.7 An example: lists

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



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

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

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

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

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

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

This works provided that p really points to a Link and that the Link pointed to by p really has a predecessor. Please convince yourself that this really is so. When thinking about pointers and linked structures, such as a list made out of 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{"Freja"});
```

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

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

15.7.1 List operations

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

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

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

We could write these operations like this:

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

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

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

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

We could have considered an attempt to **advance()** outside the list an error and thrown an exception, but we chose to return the **nullptr** thereby forcing the user to deal with that possibility. Why? Primarily because at this level of pointer use, the user has to be constantly aware of the possibility of nulltpr anyway. Yes, that's error prone, so we use higher-level constructs (such as the standard-library list; §20.6) whenever we can.

15.7.2 List use

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

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

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

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

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

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

Similarly, we can move Zeus into his correct Pantheon:

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

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

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

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

Finally, let’s print out those lists:

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

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

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

This should give

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

15.8 The this pointer

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

```
class Link {
public:
	string value;

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

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

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

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

This looks promising. We defined the operations that don't change the state of a Link into const member functions. We added (nonmodifying) next() and previous() functions so that users could iterate over lists (of 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 global insert() and modifying it:

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

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

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

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

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

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

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

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

15.8.1 More link use

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

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

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

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

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

Move Zeus into his correct Pantheon:

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

Finally, let's print out those lists:

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

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

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

This should again give

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

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

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

Drill

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

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

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

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

Review

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

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

Terms

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

Exercises

[1] What is the output format of pointer values on your implementation? Hint: Don't read the documentation.

[2] How many bytes are there in an **int**? In a **double**? In a **bool**? Do not use **sizeof** except to verify your answer.

[3] List two ways that a pointer can be misused in potentially disastrous ways. Give examples.

[4] Consider the memory layout in §15.4. Write a program that tells the order in which static storage, the stack, and the free store are laid out in memory. In which direction does the stack grow: upward toward higher addresses or downward toward lower addresses? In an array on the free store, are elements with higher indices allocated at higher or lower addresses?

[5] We have not said what happens when you run out of memory using **new**. That's called *memory exhaustion*. Find out what happens. You have two obvious alternatives: look for documentation, or write a program with an infinite loop that allocates but never deallocates. Try both. Approximately how much memory did you manage to allocate before failing?

[6] Write a program that reads characters from **cin** into an array that you allocate on the free store. Read individual characters until an exclamation mark (**!**) is entered. Do not use a **std::string**. Do not worry about memory exhaustion.

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

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

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

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

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

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

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

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

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

Postscript

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

16

Arrays, Pointers, and References

Caveat emptor!
– Good advice

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

16.1 Arrays

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

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

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

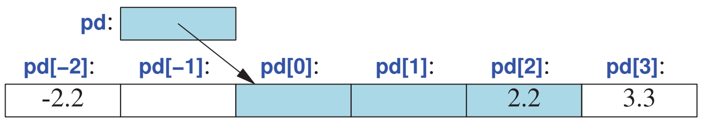
C++ uses the conventional subscript notation **[ ]** to indicate "array." For most uses, **vector<T>** is a better choice than **T[]** for representing a contiguous sequence of elements of a type **T**. "Contiguous" means that there are no gaps between the elements.

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

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

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

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

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



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

We have no idea what the memory locations marked pd[−2] and pd[3] are used for. However, we do know that they weren't meant to be used as part of our array of three 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. CC

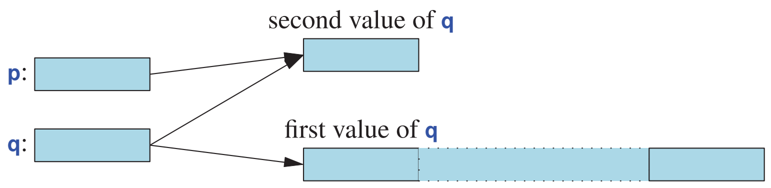
Out-of-range access is often called *range error* or *buffer overflow*. Such errors are particularly nasty because apparently unrelated parts of a program are affected. An out-of-range read gives us a "random" value that may depend on some completely unrelated computation. An out-of-range write can put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

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. AA

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

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

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

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



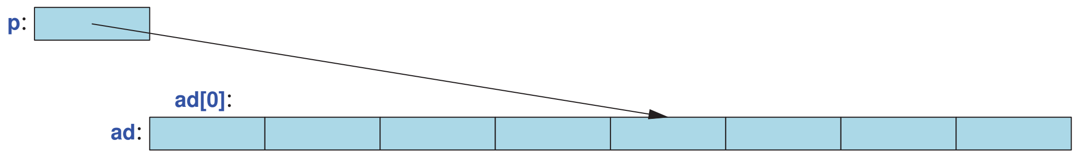
By now, we hope that you are asking, "But why can't pointers remember the size?" Obviously, we could design a "pointer" that did exactly that – a vector is almost that, and if you look through the C++ literature and libraries, you'll find many "smart pointers" that compensate for weaknesses of the low-level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

16.1.1 Pointer arithmetic

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

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

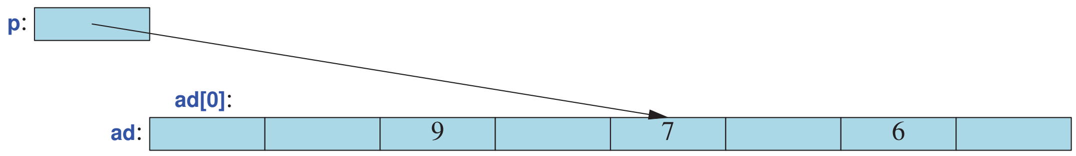
We now have a pointer **p** to the **double** known as **ad[4]**:



We can subscript and dereference that pointer:

```
*p =7;
p[2] = 6;
p[-2] = 9;
```

We get

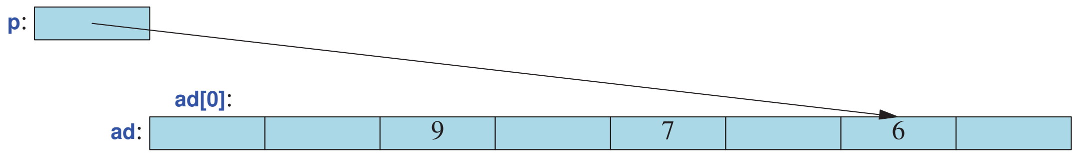


That is, we can subscript the pointer with both positive and negative numbers. As long as the resulting element is in range, all is well. However, access outside the range of the array pointed into is illegal (as with free-store-allocated arrays; see §16.1). Typically, access outside an array is not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

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

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

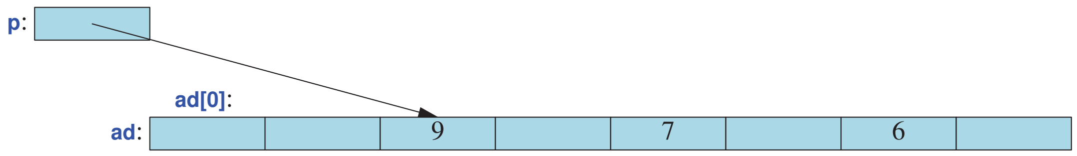
We get



We can also move backwards:

```
p -= 4;          // move p 4 elements to the left
```

We get



Using **+**, **−**, **+=**, and **−=** to move pointers around is called *pointer arithmetic*. Obviously, if we do that, we have to take great care to ensure that the result doesn't point to memory outside the array: CC

```
p += 1000;          // insane: p points into an array with just 8 elements
double d = *p;      // illegal: probably a bad value
*p = 12.34;         // illegal: probably scrambles some unknown data
```

Unfortunately, not all bad bugs involving pointer arithmetic are that easy to spot. The best policy is simply to avoid pointer arithmetic except when implementing facilities for which there is no reasonable alternative.

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

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

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

Or backward:

```
for (double* p = &ad[max−1]; p>=&ad[0]; −−p)
        cout << *p << '\n';
```

This use of pointer arithmetic is not uncommon. However, we find the last ("backward") example quite easy to get wrong. Why **&ad[max−1]** and not **&ad[max]**? Why **>=** and not **>**? These examples could equally well (and equally efficiently) be done using subscripting. Such examples could be done equally well using subscripting into a **vector** or a **span** (§16.4.1), which is more easily range checked. Also, many examples can be done using a range-**for** (§3.6.1) rather than by subscripting.

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

Note that most real-world uses of pointer arithmetic involve a pointer passed as a function argument (like the **use()** example in §16.1). In that case, the compiler doesn't have a clue how many elements are in the array pointed into: you are on your own. That is a situation we prefer to stay away from whenever we can.

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

```
double* pd = &ad[0];
double d1 = *(pd+7);
double d2 = &pd[7];
if (d2 != d3)
        cout << "impossible!\n";
```

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

16.2 Pointers and references

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

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

For example:

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

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

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

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

16.2.1 Pointer and reference parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16.2.2 Pointers as parameters

XX Pointers are popular as parameters. For example:

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

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

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

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

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

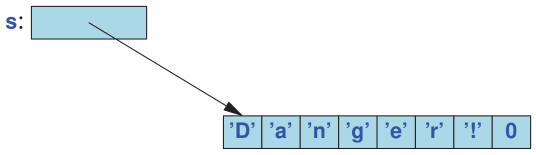
```
void print_n(span<int> s)       // a span points to an array and “remembers” its number of elements (§16.4.1)
{
        for (int x : s)
                cout << x << ' ';
}

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

16.3 C-style strings

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

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

We can represent **s** like this:



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16.4 Alternatives to pointer use

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

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

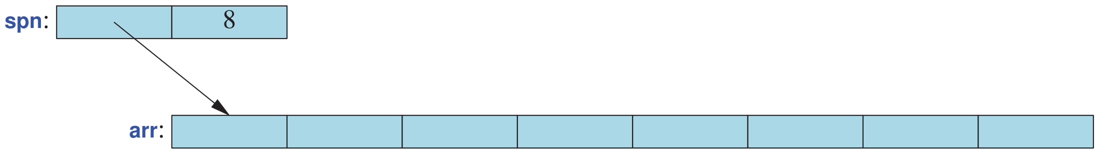
Often, there are alternatives to the standard-library components, but make the standard library as your default choice.

16.4.1 Span

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

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

or graphically:



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

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

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

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

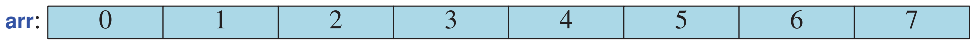
Unfortunately, std::span is not guaranteed to range check, though the version in PPP_support does.

16.4.2 array

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

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

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



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

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

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

16.4.3 not_null

Consider a possible implementation of strlen():

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

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

TRY THIS

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

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

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

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

16.5 An example: palindromes

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

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

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

16.5.1 Palindromes using string

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

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

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

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

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

16.5.2 Palindromes using arrays

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

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

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

```
istream& read_word(istream& is, char* buffer, int max)
        // read at most max-1 characters from is into buffer
{
        is.width(max);          // read at most max-1 characters in the next >>
        is >> buffer;           // read whitespace-terminated word and add zero after the last character read
        return is;
}
```

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

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

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

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

16.5.3 Palindromes using pointers

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

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

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

```
int main()
{
        const int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {
                cout << s << " is";
```

```
            if (!is_palindrome(&s[0],&s[strlen(s)−1]))
                  cout << " not";
            cout << " a palindrome\n";
      }
}
```

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

```
bool is_palindrome(const char∗ first, const char∗ last)
      // first points to the first letter, last to the last letter
{
      if (first<last)
            return (∗first==∗last) ? is_palindrome(first+1,last−1) : false;
      return true;
}
```

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

16.5.4 Palindromes using span

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

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

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

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

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

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

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

Drill

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

Array drill:

[1] Define a global int array ga of ten ints initialized to 1, 2, 4, 8, 16, etc.

[2] Define a function f() taking an int array argument and an int argument indicating the number of elements in the array.

[3] In f():
- Define a local int array la of ten ints.
- Copy the values from ga into la.
- Print out the elements of la.
- Define a pointer p to int and initialize it with an array allocated on the free store with the same number of elements as the argument array.
- Copy the values from the argument array into the free-store array.
- Print out the elements of the free-store array.
- Deallocate the free-store array.

[4] In main():
- Call f() with ga as its argument.
- Define an array aa with ten elements and initialize it with the first ten factorial values (1, 2∗1, 3∗2∗1, 4∗3∗2∗1, etc.).
- Call f() with aa as its argument.

Standard-library vector drill:

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

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

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

[4] In main():
- Call f() with gv as its argument.
- Define a vector<int> vv and initialize it with the first ten factorial values (1, 2∗1, 3∗2∗1, 4∗3∗2∗1, etc.).
- Call f() with vv as its argument.

Review

[1] What does “Caveat emptor!” mean?

[2] What is an array?

[3] How do you copy an array?

[4] How do you initialize an array?

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

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

Terms

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

Exercises

[1] Write a function, void to_lower(char* s), that replaces all uppercase characters in the C-style string s with their lowercase equivalents. For example, Hello, World! becomes hello, world!. Do not use any standard-library function. A C-style string is a zero-terminated array of characters, so if you find a char with the value 0 you are at the end.

[2] Write a function, char* str_dup(const char*), that copies a C-style string into memory it allocates on the free store. Do not use any standard-library function.

[3] Write a function, char* find_x(const char* s, const char* x), that finds the first occurrence of the C-style string x in s.

[4] Write a function, int str_cmp(const char* s1, const char* s2), that compares C-style strings. Let it return a negative number if s1 is lexicographically before s2, zero if s1 equals s2, and a positive number if s1 is lexicographically after s2. Do not use any standard-library functions. Do not use subscripting; use the dereference operator * instead.

[5] Consider what happens if you give your str_dup(), find_x(), and str_cmp() a pointer argument that is not a C-style string. Try it! First figure out how to get a char* that doesn't point to a zero-terminated array of characters and then use it (never do this in real – non-experimental – code; it can create havoc). Try it with free-store-allocated and stack-allocated "fake C-style strings." If the results still look reasonable, turn off debug mode. Redesign and re-implement those three functions so that they take another argument giving the maximum number of elements allowed in argument strings. Then, test that with correct C-style strings and "bad" strings.

[6] See what happens if you give the standard-library function strcmp() a pointer argument that is not a C-style string.

[7] Write a function, string cat_dot(const char* s1, const char* s2), that concatenates two strings with a dot in between. For example, cat_dot("Niels", "Bohr") will return a string containing Niels.Bohr.

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

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

[10] Write versions of the `cat_dot()`s from the previous exercises to take C-style strings as arguments and return a free-store-allocated C-style string as the result. Do not use standard-library functions or types in the implementation. Test these functions with several strings. Be sure to free (using `delete`) all the memory you allocated from free store (using `new`). Compare the effort involved in this exercise with the effort involved for exercises 5 and 6.

[11] Rewrite all the functions in §16.5 (palindromes) to use the approach of making a backward copy of the string and then comparing; for example, take `"home"`, generate `"emoh"`, and compare those two strings to see that they are different, so *home* isn't a palindrome.

[12] Look at the "array solution" to the palindrome problem in §16.5.2. Fix it to deal with long strings by (a) reporting if an input string was too long and (b) allowing an arbitrarily long string. Comment on the complexity of the two versions.

[13] Implement a version of the game "Hunt the Wumpus." "Hunt the Wumpus" (or just "Wump") is a simple (non-graphical) computer game originally invented by Gregory Yob. The basic premise is that a rather smelly monster lives in a dark cave consisting of connected rooms. Your job is to slay the wumpus using bow and arrow. In addition to the wumpus, the cave has two hazards: bottomless pits and giant bats. If you enter a room with a bottomless pit, it's the end of the game for you. If you enter a room with a bat, the bat picks you up and drops you into another room. If you enter the room with the wumpus or he enters yours, he eats you. When you enter a room, you will be told if a hazard is nearby:

- "I smell the wumpus": It's in an adjoining room.
- "I feel a breeze": One of the adjoining rooms is a bottomless pit.
- "I hear a bat": A giant bat is in an adjoining room.

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

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

Postscript

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

17

Essential Operations

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

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

17.1 Introduction

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

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

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

Basically, those are the constraints on the built-in types and operations of C++ (as inherited through C from hardware; see PPP2.§22.2.5 and PPP2.Ch27). In Chapter 15, we saw the beginnings of a **Vector** type that controls all access to its elements and provides us with operations that seem "natural" from the point of view of a user, rather than from the point of view of hardware.

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

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

17.2 Access to elements

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

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

We can do better. The way to get the usual **v[i]** notation is to define a member function called **operator[]**. Here is our first (naive) try:

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

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

```
Vector v(10);
double x = v[2];               // fine
v[3] = x;                      // error: v[3] is not an lvalue (§3.3)
```

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

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

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

```
class Vector {
        // ...
        double* operator[](int n) { return &elem[n]; }        // return pointer
};
```

Given that definition, we can write

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

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

```
class Vector {
        // ...
        double& operator[](int n) { return elem[n]; }                   // return a reference
        const double& operator[](int n) const { return elem[n]; }  // return a const& for a const (§8.7.4)
};
```

References are not just for function arguments.

Now we can write

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

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

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

17.3 List initialization

Consider yet again our Vector:

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

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

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

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

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

So how do we write a constructor that accepts an initializer list as its argument? A { }-delimited list of elements of type T is presented to the programmer as an object of the standard-library type initializer_list<T>, a list of Ts. Its first element is identified by begin() and the end of the list by end(), so we can write:

```
class Vector {
        int sz;                // the size
        double∗ elem;          // a pointer to the elements
public:
        Vector(int s)                          // constructor (s is the element count)
                :sz{s}, elem{new double[s]}    // uninitialized memory for elements
        {
                for (int i = 0; i<sz; ++i)
                        elem[i] = 0.0;         // initialize to a default value
        }
        Vector(initializer_list<double> lst)           // initializer-list constructor
                :sz{lst.end()−lst.begin()},
                 elem{new double[sz]}                  // uninitialized memory for elements
        {
                copy(lst.begin(),lst.end(),elem);      // initialize using std::copy()
        }

        // ...
};
```

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

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

Now we can write

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

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

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

CC

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

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

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

The difference is purely one of style.

Note that we pass `initializer_list<double>` by value. That was deliberate and required by the language rules: an `initializer_list` is simply a handle to elements allocated "elsewhere."

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

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

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

17.4 Copying and moving

Consider again our incomplete **Vector**:

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

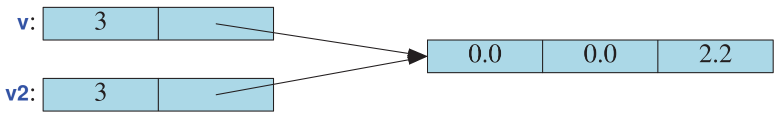
Let's try to copy one of these vectors:

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

Ideally, **v2** becomes a copy of **v** (that is, **=** makes copies); that is, **v2.size()==v.size()** and **v2[i]==v[i]** for all **i**s in the range [**0**:**v.size()**). Furthermore, all memory is returned to the free store upon exit from **f()**. That's what the standard-library **vector** does (of course), but it's not what happens for our still-far-too-simple **Vector**. If you are lucky, your compiler will warn you.

Our task is to improve our **Vector** to get it to handle such examples correctly, but first let's figure out what our current version actually does. Exactly what does it do wrong? How? And why? Once we know that, we can probably fix the problems. More importantly, we have a chance to recognize and avoid similar problems when we see them in other contexts.

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



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

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

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

```
delete[] elem;
```

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

17.4.1 Copy constructors

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

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

```
Vector(const Vector&);          // copy a Vector
```

This constructor will be called when we try to initialize one Vector with another. We pass by reference because we (obviously) don't want to copy the argument of the constructor that defines copying. We pass by const reference because we don't want to modify our argument (§7.4.4). So we refine Vector like this:

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

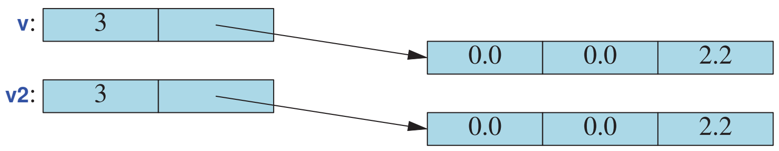
The copy constructor sets the number of elements (sz) and allocates memory for the elements (initializing elem) before copying element values from the argument Vector:

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

Given this copy constructor, consider again our example:

```
Vector v2 = v;
```

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



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

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

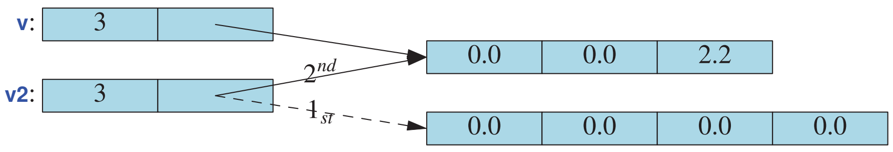
This will output **0 0**.

17.4.2 Copy assignments

CC We handle copy construction (initialization), but we can also copy **Vector**s by assignment. As with copy initialization, the default meaning of copy assignment is memberwise copy, so with **Vector** as defined so far, assignment will cause a double deletion (exactly as shown for copy constructors in §17.4) plus a memory leak. For example:

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

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



When we leave **f2()**, we have the same disaster as we had when leaving **f()** in §17.4 before we added the copy constructor: the elements pointed to by both **v** and **v2** are freed twice (implicitly using **delete[]**). In addition, we have leaked the memory initially allocated for **v2**'s four elements. We "forgot" to free those. If you ever make that mistake, hope for a compiler warning.

The remedy for this flawed copy assignment is fundamentally the same as for the flawed copy initialization: we define an assignment that copies properly:

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

Vector& Vector::operator=(const Vector& a)   // make this Vector a copy of a
{
        double* p = new double[a.sz];        // allocate new space
        copy(a.elem,a.elem+a.sz,p);          // copy elements [0:sz) from a.elem into p
        delete[] elem;                       // deallocate old space
        elem = p;                            // now we can reset elem
        sz = a.sz;
        return *this;                        // return a self-reference (§15.8)
}
```

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

```
double* p = new double[a.sz];        // allocate new space
copy(a.elem,a.elem+a.sz,p);          // copy elements [0:sz) from a.elem into p
```

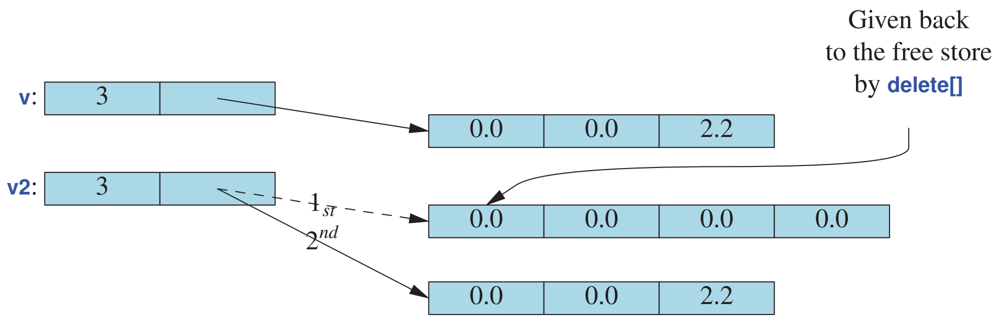
Then we free the old elements from the target **Vector**:

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

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

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

We can represent the result graphically like this:



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

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

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

Please check that our implementation handles that case correctly (if not optimally efficient).

17.4.3 Copy terminology

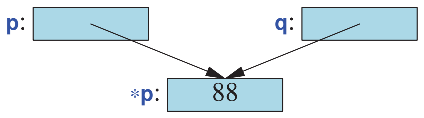
CC Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer (or reference) or copy the information pointed to (referred to):

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

Here is an example of shallow copy:

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

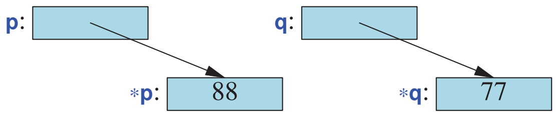
We can illustrate that like this:



In contrast, we can do a deep copy:

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

We can illustrate that like this:



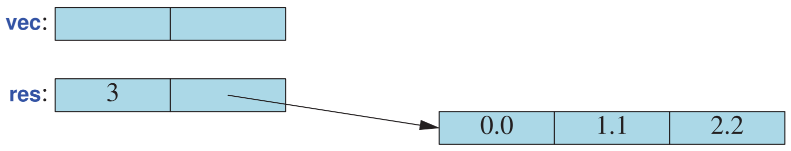
AA Using this terminology, we can say that the problem with our original **Vector** was that it did a shallow copy, rather than copying the elements pointed to by its **elem** pointer. Our improved **Vector**, like **std::vector**, does a deep copy by allocating new space for the elements and copying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

17.4.4 Moving

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

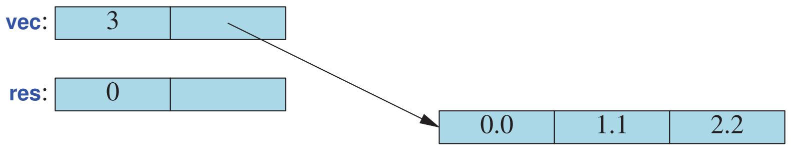
```
Vector fill(istream& is)
{
        Vector res;                    // assuming we have a default constructor (§17.5)
        for (double x; is>>x; )        // assuming support for range-for (§17.6)
                res.push_back(x);      // assuming we have a Vector::push_back (§17.8.4)
        return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

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

Here, we fill the local **Vector**, **res**, from the input stream and return it to **use()**. Copying **res** out of **fill()** and into **vec** could be expensive. But why copy? We don't want a copy! We can never use the original (**res**) after the return. In fact, **res** is destroyed as part of the return from **fill()**. So how can we avoid the copy? Consider how **res** would be represented in memory if we had entered **0.0 1.1 2.2**:



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

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



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

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

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

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

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

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

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

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

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

XX 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 = fill2(cin);
        // ... use vec ...
        delete vec;
}
```

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

17.5 Essential operations

CC 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 (§17.5)
- Copy constructor (copy object of same type; §17.4.1)
- Copy assignment (copy object of same type; §17.4.2)
- Move constructor (move object of same type; §17.4.4)
- Move assignment (move object of same type; §17.4.4)
- Destructor (§15.5)

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

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

The meaning/use of an initializer is completely up to the constructor. The standard **string**'s constructor uses a character string as an initial value, whereas **Image**'s constructor uses the string as the name of a file to open. Usually, we use a constructor to establish an invariant (§8.4.3). If we can't define a good invariant for a class that its constructors can establish, we probably have a poorly designed class or a plain data structure.

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

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

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

So, having a default constructor is often useful. The question then becomes: "When does it make sense to have a default constructor?" An answer is: "When we can establish the invariant for the class with a meaningful and obvious default value." For every type **T**, **T{}** is the default value, if a default exists. For example, **double{}** is **0.0**, **string{}** is **""**, **vector<int>{}** is the empty **vector** of **int**s, and in §8.4.2 we made our **Date{}** January 1, 2001. For our **Vector** that would be:

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

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

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

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

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

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

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

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

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

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

17.5.1 Explicit constructors

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

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

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

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

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

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

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

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

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

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

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

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

17.5.2 Debugging constructors and destructors

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

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

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

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

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

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

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

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

```
X glob {2};                          // a global variable

X copy(X a) { cout << "copy()\n"; return a; }
X copy2(X a) { cout << "copy2()\n"; X aa = a; return aa; }
X& ref_to(X & a) { cout << "ref_to()\n"; return a; }
X* make(int i) { cout << "make()\n";  X a(i); return new X(a); }

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

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

TRY THIS

Try executing that. Then remove the move operations and run it again. The compilers can be very clever at avoiding unnecessary copies. We really mean it: do run this example and try to explain the result. If you do, you'll understand most of what there is to know about construction and destruction of objects.

Depending on the quality of your compiler, you may note some "missing copies" relating to our calls of copy() and copy2(). We (humans) can see that those functions do nothing: they just copy a value unmodified from input to output. A compiler is allowed to assume that a copy operation copies and does nothing else but copy. Based on that, it can eliminate redundant copies (copy elision; §17.4.4). Similarly, a compiler assumes that move operations do nothing but move. AA

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

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

17.6 Other useful operations

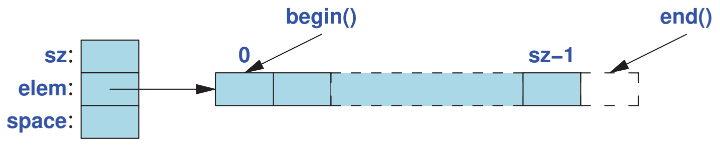
In addition to the essential operations, there are quite a few that are often important for common uses:

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

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

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

Or graphically:



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

17.6.1 Comparison operators

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

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

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

17.6.2 Related operators

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

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

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

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

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

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

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

Other operators that you can define for your own types include

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

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

17.7 Remaining Vector problems

Our Vector class has reached the point where we can

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

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

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

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

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

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

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

- The number of elements
- The type of elements

Those kinds of variability are useful in rather fundamental ways. We always collect data. Looking around my desk, I see piles of bank statements, credit card bills, and phone bills. Each of those is basically a list of lines of information of various types: strings of letters and numeric values. In front of me lies a phone; it keeps lists of phone numbers and names. In the bookcases across the room, there are shelf after shelf of books. Our programs tend to be similar: we have containers of elements of various types. We have many different kinds of containers (vector is just the most widely useful), and they contain information such as phone numbers, names, transaction amounts, and documents. Essentially all the examples from my desk and my room originated in some computer program or another. The obvious exception is the phone: it *is* a computer, and when I look at the numbers on it, I'm looking at the output of a program just like the ones we're writing. In fact, those numbers may very well be stored in a **std::vector<Number>**.

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

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

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

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

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

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

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

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

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

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

17.8 Changing size

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

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

we can change its size in three ways:

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

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

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

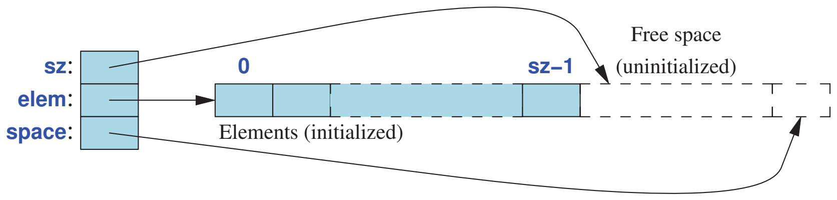
The standard-library **vector** offers more operations that can change a **vector**'s size, such as **erase()** and **insert()**, but here we will just see how we can implement **resize()**, **push_back()**, and **operator=()** for our **Vector**.

17.8.1 Representation

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

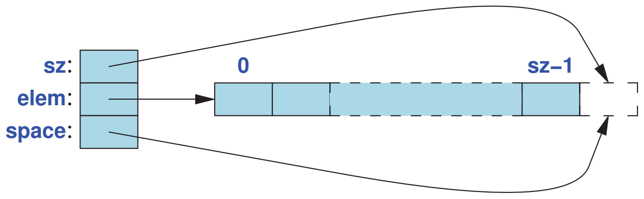
```
class Vector {
        // ...
private:
        int sz;                 // number of elements
        double* elem;           // address of first element
        int space;              // number of elements plus "free space"/"slots" for new elements
};
```

We can represent this graphically like this:



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

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

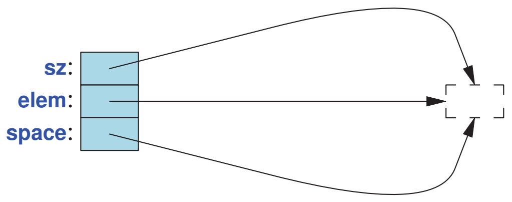


We don't start allocating extra slots until we begin changing the number of elements. Typically, space==sz, so there is no memory overhead unless we use push_back().

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

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

That gives



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

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

17.8.2 reserve() and capacity()

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

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

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

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

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

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

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

17.8.3 resize()

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

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

Let's see what we get:

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

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

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

TRY THIS

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

17.8.4 push_back()

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

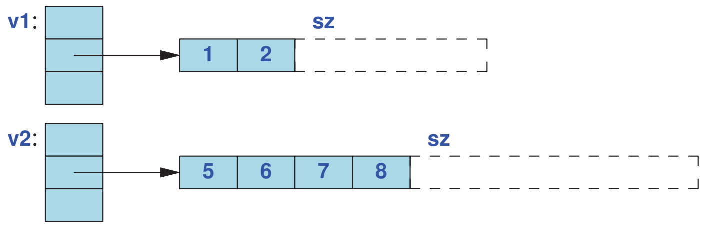
```
void Vector::push_back(double d)
         // increase vector size by one; initialize the new element with d
{
        if (space==0)                   // start with space for 8 elements
                reserve(8);
        else if (sz==space)
                reserve(2*space);       // get more space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

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

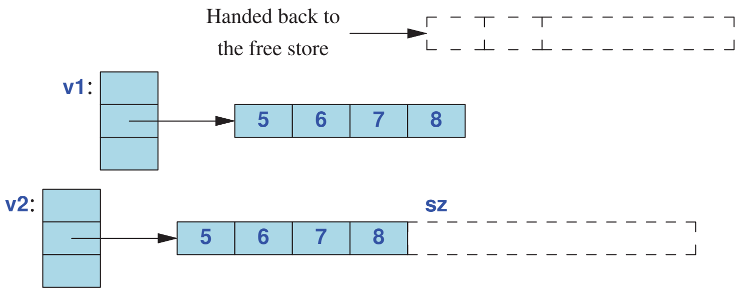
17.8.5 Assignment

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

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



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



The simplest implementation of that is:

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

Like this:

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

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

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

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

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

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

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

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

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

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

17.9 Our Vector so far

Now we have an almost real Vctor of doubles:

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

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

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

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

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

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

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

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

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

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

Note how it has the essential operations (§17.5): constructor, default constructor, copy operations, move operations, and destructor. It has an operation for accessing data (subscripting: **[ ]**) and for providing information about that data (size() and capacity()) and for controlling growth (resize(), push_back(), and reserve()).

Drill

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

Review

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

Terms

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

Exercises

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

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

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

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

Postscript

The standard-library vector is built from lower-level memory management facilities, such as pointers and arrays, and its primary role is to help us avoid the complexities of those facilities. Whenever we design a class, we must consider the essential operations: initialization, copying, moving, and destruction. In addition, we should consider what further operations are needed for that kind of type. For most types: == and !=. For containers: list initialization and assignment, begin(), end(), and swap(). The ultimate aim of such operations is to give the type a coherent and familiar semantics. Finally, we dramatically increase the flexibility of our Vector by adding operations that allow us to change a Vector's size: push_back(), resize(), and reserve().

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

18

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. We show 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. 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. In this context, we discuss the general resource-management technique called ‘‘Resource Acquisition is Initialization’’ (RAII), resource-management guarantees, and the standard-library resource-management pointers unique_ptr and shared_ptr.

18.1 Templates

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. We want to freely specify the element type for our Vectors like this:

```
Vector<double>
Vector<int>
Vector<Month>
Vector<Window*>                // Vector of pointers to Windows
Vector<Vector<Record>>         // Vector of Vectors of Records
Vector<char>
```

AA 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 must examine how the designers and implementers of the standard library provide facilities such as the vector type and the sort() function (§21.5). 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 Chapter 19 to Chapter 21, we show how templates can be used to implement the standard-library containers and algorithms.

CC 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.

18.1.1 Types as template parameters

CC We want to make the element type a parameter to our Vector. So we take 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:

```
template<typename T>          // for all types T (just like in math)
class Vector {
        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(initializer_list<T>);                          // list constructor
        Vector& operator=(initializer_list<T>);               // list assignment
        Vector(const Vector&);                          // copy constructor
        Vector& operator=(const Vector&);               // copy assignment

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

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

        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; }          // the current capacity

        void resize(int newsize);                       // growth
        void push_back(const T& d);
        void reserve(int newalloc);

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

template<typename T>
bool operator==(const Vector<T>&, const Vector<T>&);

template<typename T>
bool operator!=(const Vector<T>&, const Vector<T>&);
```

That's just our Vector of doubles from §18.1 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<T>, 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: **AA**

```
class Vector_char {
        int sz;         // the size
        char* elem;     // a pointer to the elements
        int space;      // size + free space
public:
        Vector_char() :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(initializer_list<T>);                         // list constructor
        Vector& operator=(initializer_list<T>);              // list assignment

        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;                                // the current capacity

        void resize(int newsize);                            // growth
        void push_back(const char& d);
        void reserve(int newalloc);

        char∗ begin() const { return elem; }                 // iteration support
        char∗ end() const { return elem+sz; }
};

bool operator==(const Vector_char&,  const Vector_char&);
bool operator!=(const Vector_char&,  const Vector_char&);
```

Similarly, for Vector<double>, the compiler generates roughly the Vector (of double) from §17.9 (using a suitable internal name meaning Vector<double>).

CC 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<Open_polyline∗> 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("Ada");
        // ...
}
```

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("Ada"), 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("Ada") 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.

18.1.2 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 CC

- *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 and move. In Chapter 19 to Chapter 21, we will see templates that require arithmetic operations on their arguments. When what we parameterize is a class, we get a *class template* (often called a *parameterized type* or a *parameterized class*). When what we parameterize is a function, we get a *function template*, (often called a *parameterized function* and sometimes also 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. CC

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-* CC *oriented programming* (§12.3). 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 Chapter 10 to Chapter 13 we accessed many shapes (such as **Text**, **Circle**, and **Polygon**) through the interface defined by **Shape**. When we use **vector**s, 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 §12.3.1). 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).

CC To sum up:

- *Generic programming*: supported by templates, relying on compile-time resolution
- *Object-oriented programming*: supported by class hierarchies with virtual functions, allowing run-time resolution

Combinations of the two are possible and useful. For example:

```
void draw_all(vector<Shape*>& v)
{
        for (auto x : v)
                x−>draw();
}
```

Here, we call a virtual function (**draw()**) on a base class (**Shape**) – 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.

AA 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 PPP2.Ch24 and PPP2.Ch25).
- Use templates where flexibility in combining information from several types is essential (e.g., the C++ standard library; see Chapter 19 to Chapter 21).

18.1.3 Concepts

XX Templates with unconstrained parameters, **template<typename T>**, have many useful properties, such as great flexibility and near-optimal performance. In addition, we need a precise specification of what a template requires of its template arguments. For example, in much older C++ code, we find something like this:

```
template<typename T>          // for all types T
class Vector {
        // ...
};
```

This doesn't precisely state what is expected of an argument type T.

We call a set of requirements on a set of template arguments a *concept*. For example, a vector requires that its elements can be copied and 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. 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 notation 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 {
        // ...
};
```

The mathematical vocabulary we use to describe concepts reflects the mathematical roots of concepts. Concepts represent a form of predicate logic.

The standard library provides many useful concepts, some of which we use in Chapter 19 to Chapter 21. Then, their meaning and utility will become clear.

- range<C>(): C can hold Elements and be accessed as a [begin():end()) sequence.
- input_iterator<In>(): In can be used to read a sequence [b:e) once only (like an input stream).
- output_iterator<Out>(): Out can be used to write to output (like an output stream). We cannot read using Out.
- forward_iterator<For>(): For can be used to traverse a sequence [b:e) (like a linked list, a vector, or an array). We can traverse [b:e) repeatedly using For.
- random_access_iterator<Ran>(): Ran can be used to read and write a sequence [b:e) repeatedly and supports subscripting using [ ].
- random_access_range<Ran>(): range with random_access_iterators.
- equality_comparable<T>(): We can compare two Ts for equality using == to get a Boolean result.
- equality_comparable_with<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 a set of arguments of the N specified types T1, T2, ... to get a Boolean result.

- indirect_unary_predicate<P,I>(): We can call P with an iterator argument of type I to get a Boolean result.
- invocable<F,T...>(): We can call F with a set of arguments of the N specified types T1, T2, ...
- totally_ordered<T>(): We can compare two Ts with ==, !=, <, <=, >, and >= to get a Boolean result representing a total order.
- totally_ordered_with<T,U>(): We can compare a T and a U with ==, !=, <, <=, >, and >=, to get a Boolean result representing a total order.
- binary_operation<B,T,U>(): We can use B to do an operation on two Ts.
- binary_operation<B,T,U>(): We can use B to do an operation on a T and a U.
- derived_from<D,B>(): D is publicly derived from B.
- convertible_to<F,T>(): An F can be converted to a T.
- integral<T>(): A T is an integral type (like int).
- floating_point<T>(): A T is a floating point type (like double).
- copyable<T>(): A T can be copied.
- moveable<T>(): A T can be moved.
- semiregular<T>(): A T can be copied, moved, and swapped.
- regular<T>(): T is semiregular and equality_comparable.
- sortable<I>(): I is a random_access_iterator with value_type elements that can be compared using <.
- sortable<I,C>(): I is a random_access_iterator with value_type elements that can be compared using C.

There are many more to serve a variety of needs. In addition, we define a few that we need here:

- Element<E>(): E can be an element in a container. Roughly, that means that E is semiregular. Individual operations on a vector<E> can impose stricter requirements.
- Boolean<T>: T can be used as a Boolean (like bool).
- Number<N>(): N behaves like a number, supporting +, –, *, and /.
- Allocator<A>(): A can be used to acquire and release memory (like the free store).

For standard-library containers and algorithms, these concepts (and many more) are specified in excruciating detail. Here, especially in Chapter 19 to Chapter 21, we use them to specify our containers and algorithms.

The type of the elements of a container or iterator type, T, is called its *Value type* and is often defined as a member type T::value_type; see vector and list (§19.3).

18.1.4 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 (§12.4.1). 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 (§12.3.1). What would have happened – had we allowed it – would have been what is called *slicing*; that is, only the Shape part of the Circle would have been copied and the result would have been an incomplete Shape that would have caused run-time errors if used. It is the class object equivalent of integer truncation (§2.9). XX

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)
{
        Shape s = new Rectangle{Point{0,0},Point{100,100}}
        v.push_back(s);           // put a Rectangle* into a vector<Shape*>
}
```

Obviously, we can put a Rectangle∗ into a vector<Shape∗>, but not into a Circle∗. After all, the Rectangle∗ doesn't point to a Circle. However, had the type system allowed f(vpc) that would have been exactly what it did. 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<B>" for an arbitrary template C – and we should value that as a protection against accidental type violations. See also PPP2.§25.4.4. XX

18.1.5 Value 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 a detailed description of *value template parameters* is beyond the scope of this book, so we will show just one example. Consider a buffer type: CC

```
template<typename T, int sz>
class Buffer {
pblic:
        using value_type = T;
        const int size() { return sz; }
        // ... useful operations ...
private:
        T elem[sz];
};
```

We can use this to place fixed-sized buffers where they are needed. For example:

```
Buffer <int,1024> global;

void use()
{
        Buffer<Message,12> local;
        // ...
}
```

A type like Buffer is useful when we don't want to use the free store.

18.2 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 §17.8.2, §17.8.3, and §17.8.4 they make assumptions that are true for doubles, but not true for all types that we'd like to use as Vector element types, such as strings and Dates:

- How do we handle a Vector<X> where X doesn't have a default value?
- How do we cope with element types that have copy operators that may throw exceptions?
- How do we ensure that elements are destroyed when we are finished with them?

AA 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 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<Element 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

class No_default {
        No_default(int);               // the only constructor
        // ...
};

Vector<No_default> v2(10);         // error: tries to make 10 No_default()s
Vector<No_default> v3;             // OK: makes no elements
```

```
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 in the presence of destructors and exceptions. So far, we have gone out of our 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.

18.2.1 Allocators

First, we need to find a way to get construction and destructions done correctly when 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
};
```

Unsurprisingly, an **allocator** is exactly what we need for implementing **Vector<T>::reserve()**. We start by giving **Vector** an allocator parameter:

```
template<Element T, typename A = allocator<T>>
class Vector {
        A alloc;          // use allocate to handle memory for elements
        // ...
};
```

The standard-library allocator model is based on a type **pmr** (*polymorphic memory resource*) that is a generalization of our **allocator**. Look it up if you feel the need to use something more advanced than the default allocator used by operator **new**.

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** member functions that directly deal with memory, such as **Vector<T>::reserve()**:

```
template<Element T, Allocator A>
void Vector<T,A>::reserve(int newalloc)
{
        if (newalloc<=space)                     // never decrease allocation
                return;
        T* p = alloc.allocate(newalloc);             // allocate new space
        uninitialized_move(elem,&elem[sz],p);        // move elements into uninitialized space
        destroy(elem,space);                         // destroy old elements
        alloc.deallocate(elem,capacity());           // 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. We use the standard-library function uninitialized_move() to move the elements into the newly allocated space; uninitialized_move(b,e,p) moves from the range [b:e) into the range [p:p+(e−b)). We use the standard-library function destroy() to call the destructors for the old copies; destroy(b,e) invokes the destructors for elements in the range [b:e). In Chapter 21, we show many more algorithms operating on half-open ranges, such as [b:e).

Given reserve(), Vector<T,A>::push_back() is simple:

```
template<Element T, Allocator A>
void Vector<T,A>::push_back(const T& val)
{
        reserve((space==0) ? 8 : 2*space);        // get more space
        construct_at(&elem[sz],val);              // add val at end
        ++sz;                                     // increase the size
}
```

This uses the standard-library function construct_at() to construct the new element in its correct position in the uninitialized space.

Similarly, Vector<T,A>::resize() is not too difficult:

```
template<Element T, Allocator A>
void Vector<T,A>::resize(int newsize, T val = T())
{
        reserve(newsize);
        if (sz<newsize)
                uninitialized_fill(&elem[sz],&elem[newsize],val);    // initialize added elements to val
        if (newsize<sz)
                destroy(&elem[newsize],&elem[sz]);                   // destroy surplus original elements
        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 uninitialized_fill() is a cousin of uninitialized_move() that repeatedly assigns a single value, rather than moving from a sequence of elements.

CC 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 back into "raw memory." Conversely, a constructor turns "raw memory" into a typed object.

XX "Messing with allocators" is pretty advanced stuff, and tricky. Leave it alone until you are ready to become an expert.

18.3 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<Element T,Allocator A>
T& Vector<T,A>::operator[](int n)
{
        return elem[n];
}
```

So, consider:

```
Vector<int> v(100);
v[−200] = v[200];           // oops!
for (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<Element T, Allocator 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<Element T, Allocator A>
T& Vector<T,A>::at(int n)
{
        if (n<0 || sz<=n)
                throw out_of_range();
        return elem[n];
}

template<Element T, Allocator 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.

Obviously, these range checks make sense only if the size() of a Vector is reasonable. What about Vector(−100)? That doesn't make sense and we "forgot" to check the size argument to the constructor. However, our constructor used new T[s] and new tests its size argument, and −100 would make it throw a bad_array_new_length exception. The standard-library vector has its own opinion about what's reasonable and would throw a length_error. We can do the same for Vector:

```
int reasonable_size = std::numeric_limit<int>::max;      // largest int (possibly 2’147’483’647).

template<Element T>
explicit Vector(int s) :sz{s}, elem{new T[s]}, space{s}
{
        if (!0<s && s<reasonable_size))
                throw std::length_error{"size too large for Vector"};
        for (int i=0; i<sz; ++i)
                elem[i]=T{};                    // elements are initialized
}
```

18.3.1 An aside: design considerations

CC So far, so good, but why didn't we just add the range check to operator[]()? Well, std::vector provides checked at() and potentially 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, such as the one from PPP_support but preferably one supported by your standard-library provider.

18.3.1.1 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 are skeptical about that argument, but again nobody who hasn't had to make such decisions about real code should not 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 **vector**s that (being pre-standard) didn't use exceptions. Much of that code was later modified to use the standard.

18.3.1.2 Efficiency

AA 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. In general, it is a mistake to try to optimize every detail of a program. That leads to brittle code and distracts from optimizing the program as a whole.

18.3.1.3 Constraints

The arguments against using exceptions hold 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 (PPP2.Ch25.2.1), prefer exception-based error handling and range-checked **vector**s.

18.3.1.4 Optional checking

AA 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.

18.3.2 Module PPP_support

Like our **Vector**, most implementations of **std::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 §18.3.1: 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 version from **PPP_support**. It cuts down on errors and debug time at little cost to performance.

Module PPP_support has the following outline:

```
export module PPP_support;

export import std;

export namespace PPP {
        using namespace std;                    // make all of std available

        // except for what we want to provide ourselves for PPP:

        template <Element T>
        class Checked_vector : std::vector<T> { /* ... */ };   // range-checked vector

        class Checked_string : std::string { /* ... */ };          // range-checked string

        template<Element T>
        class Checked_span : std::span { /* ... */ };                    // range-checked span

        // added features:

        // ...

} // namespace PPP
```

This demonstrates the power of modules, namespaces, and inheritance.

PPP_support::Checked_vector is defined like this:

```
namespace PPP {
        template<Element T>                          // constrain element types
        struct Checked_vector : public std::vector<T> {
                using size_type = typename std::vector<T>::size_type;
                using value_type = T;
                using vector<T>::vector;             // use vector<T>'s constructors

                T& operator[](size_type i)           // rather than return at(i);
                {
                        return std::vector<T>::at(i);
                }

                const T& operator[](size_type i) const
                {
                        return std::vector<T>::at(i);
                }
        }; // Checked_vector
} // namespace PPP
```

Deriving from std::vector gives us all of vector's member functions for PPP_support::Checked_vector. The first using introduces a convenient synonym for std::vector's size_type. The second using gives a name to the element type. The third using gives us all of std::vector's constructors for PPP_support::Checked_vector.

AA This PPP_support::Checked_vector has been useful in debugging nontrivial programs. The alternative is to use a systematically checked implementation of the complete standard-library vector. Unfortunately, there is no standard, portable, complete, and clean way of getting range checking from an implementation of std::vector's [ ]. All major implementations of the standard library have one, but at the time of writing there is no standard way of installing them. We find module PPP_support a significant help, especially when we use a C++ Core Guidelines rules checker to keep us out of nasty "dark corners" of the language. In general, avoid data structures that don't have sufficient information to do range checks.

18.4 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 4). 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 many systems. For example, many embedded systems cannot just stop and a financial system cannot just abort in the middle of a transaction. Often, we need to do a bit of clean-up before terminating or to re-start the system in a known good state. For some kinds of errors, we know enough about the state of the program to be able to recover. For example, failing to compute an answer, we might try again using a different algorithm or a different set of resources. For some kinds of errors, we might even decide not to recover but to produce an answer that's not the desired one but considered harmless. For example, failing to correctly display an image, we might decide to display an image signifying "we couldn't produce the image you asked for."

CC 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

CC 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 §15.4.5, it's not always easy to remember to release memory. 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. To write reliable software, we must make the release of resources, such as memory, implicit. Doing so also makes writing the code easier.

CC 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.

18.4.1 Potential resource-management problems

XX 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 of **suspicious()** 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.

Consider:

```
void suspicious(int s, int x)
{
    int* p = new int[s];            // acquire memory
    // ...
    if (x<0)
        p = q;                      // make p point to another object
    if (x==0)
        return;                     // we may return
    if (0<x)
        p[x] = v.at(x);             // subscripting may throw
    // ...
    delete[] p;                     // release memory
}
```

This code will never delete **p**; it leaks memory. The reason for that varies depending on the value of **x**. Obviously, we'd never deliberately write such messy code, but the three ways of messing up illustrated do turn up in real-world code where functions are longer and the control structures more complicated. Often, the problem wasn't in the original code, but was inserted by mistake years later by someone maintaining the code.

When people first encounter the exception throw version of this problem, they tend to consider it a problem with exceptions rather than a general 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);    // subscripting may throw
                // ...
        }
        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. In other words, this solution is ugly; worse, it doesn't generalize well. Consider acquiring more resources:

```
void suspicious(int s)
{
        int* p = new T[s];
        // ...
        int* q = new T[s];
        // ...
        delete[] p;
        // ...
        delete[] q;
}
```

The try-catch technique works for this example also, but we will 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. In particular, if we try to handle resource leaks with try-catch, we have to remember to use try-catch consistently. That's not a good idea. Experience shows that we sometimes forget. It is easy to forget to release a resource; just ask any librarian! Release must be implicit and guaranteed. XX

Having a lot of try-blocks is a sign of poor design and an indicator of likely problems with error handling. We must – and can – do better.

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.

We use the new/delete example just because it's easy to understand and experiment with. However, most examples of resource leaks are less easy to spot. Consider an old-style example that can still be found in many programs:

```
void old_style()
{
        FILE∗ output = fopen("myfile","r");             // acquire
        fprintf(output,"Hello, old world!\n");          // print to file
        // ...
        fclose(output);
}
```

Here use the (C and C++) standard-library functions fopen() and fclose() to open and close a file. Again, it is easy for code in the ... part to exit the function without closing the file. If so, the output buffer may never be flushed, leaving us without a hint of why no output was produced.

18.4.2 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 a combination of suspicious() and old_style():

```
void user(int s)
{
        vector<T> p(s);
        vector<T> q(s);
        ifstream in {"myfile"};
        // ...
}
```

CC This is better. More importantly, it is *obviously* better. The resources (here, free-store memory and a file handle) are acquired by constructors and released by the matching destructors. 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 releases it again in the matching destructor. Examples of resources that are usually best dealt with in this way include character strings, database transactions, 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 RAII.

Consider the example above. Whichever way we leave user(), the destructors for p, q, and in are invoked appropriately. 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.

AA In particular, use vector rather than explicit new and delete when you need a nonconstant amount of storage within a scope.

Finally, consider an artificial example concocted to test error handling:

```
void test()
{
        string name;
        cin>>name;
        ifstream in {name};                                // §9.3
        if (!in)
              error("couldn't open ",name);

        vector<string> v {"hello"};
        for (string s; in>>s; )
              v.push_back(s);
        v[3] += "odd";

        Simple_window win;                                 // Chapter 10
        auto ps = make_unique<Shape>(read_shape(cin));     // §18.5.2, §12.2
        Smiley_face face {Point{0,0},20};
        win.attach(face);
        // ...
  }
```

Real-world code can be far more complex than that:

- Imagine what it would take to re-write this test() with types that did not have destructor.
- Imagine what it would take to re-write this test() and catch and correctly report all possible errors without using exceptions.

Note that we didn't rely on exceptions when opening the file. We could have done so, but there is nothing exceptional about not being able to open a file, so we dealt with that possibility locally.

18.4.3 Resource-management guarantees

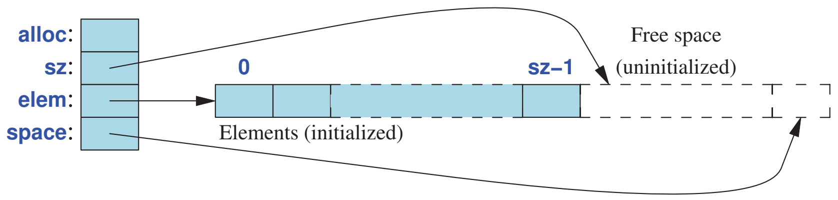
When we leave a function – or just a scope – what kinds of guarantees can we offer to make resource management comprehensible? To be able to think rationally about such problems, consider these basic concepts: CC

- *The basic guarantee*: A function either succeeds or throws an exception without having leaked any resources. All code that is part of a program that we expect to recover from an exception throw (or any other form of error affecting resources) should provide the basic guarantee. All standard-library code provides the basic guarantee.
- *The strong guarantee*: A function provides the basic guarantee and 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. 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 or throwing an exception, we would not be able to write code to meet the basic guarantee or the strong guarantee. Fortunately, essentially all built-in facilities in the C++ language provide the no-throw guarantee: they simply can't throw. To avoid throwing, simply avoid throw itself, new, and dynamic_cast of reference types. The main danger is a simple return that leaves a required resource unreleased.

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.

18.4.4 RAII for Vector

As an example of resource management, let's look at how we can provide guarantees for Vector assignment. As we saw in §17.8, the key to memory management and element initialization for Vector is the reserve() operation, so we must start there. That reserve() was written without thought about exceptions. However, the call to allocate more memory may fail (throwing an exception) and so might the move of the elements into the newly allocated space. Ouch! We could try adding a try-block, but in that direction lies complexity (§18.4.1). 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_rep 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 = allocator<T>>
struct Vector_rep {
        A alloc;                    // allocator
        int sz;                     // number of elements
        T* elem;                    // start of allocation
        int space;                  // amount of allocated space

        Vector_rep(const A& a, int n)
                : alloc{ a }, sz{ n }, elem{ alloc.allocate(n) }, space{ n } { }
        ~Vector_rep() { alloc.deallocate(elem, space); }
};
```

That Vector_rep deals with memory rather than (typed) objects. Our Vector implementation can use that to hold objects of the desired element type. Basically, Vector (dealing with typed elements) is simply a convenient interface to Vector_rep (dealing with "raw" memory):

```
template<typename T, typename A = allocator<T>>
class Vector {
        Vector_rep<T,A> r;
public:
        Vector() : r{A{},0} { }
        explicit Vector(int s) :r{A{},s}
        {
                for (int i = 0; i < r.sz; ++i)
                        r.elem[i] = 0;          // elements are initialized
        }
        // ...
}
```

We must revise all member functions to use the **Vector_rep**. In particular, we can rewrite **reserve()** from §18.2.1 to something simpler and more correct:

```
template<typename T, typename A>
void Vector<T, A>::reserve(int newalloc)
{
        if (newalloc <= r.space)                                    // never decrease allocation
                return;
        Vector_rep<T, A> b{ r.alloc,newalloc };                     // allocate new space
        uninitialized_move(r.elem, r.elem+r.sz, b.elem);            // move
        destroy(r.elem, r.elem + r.sz);                             // destroy the old elements
        swap(r, b);                                                 // swap representations
}
```

We use the standard-library functions **uninitialized_move()** and **destroy()** to do most of the work.

When we exit **reserve()**, the old allocation is automatically freed by **Vector_rep**'s destructor if the copy operation succeeded. If instead that exit is caused by a move operation throwing an exception, the new allocation is freed. The **swap()** function is a standard-library algorithm that exchanges the value of two objects.

AA Wait! "A move operation throwing an exception!" Is that possible? Is that allowed? Move operations are simple and have no reason to throw. They don't acquire new resources. Make sure that that no move you write ever throws. That's not hard. However, it turns out that there are quite a few types from before the introduction of move operations for which the move is synthesized by copying into the target followed by destructing the source.

So how does **uninitialized_move()** cope with throwing move operations? To offer the basic guarantee, it mustn't leak any resource. In my experience, throwing move operations are very rare, but **uninitialized_move()** must still handle the cases like the one where three objects are successfully moved and then the fourth move throws. It does so by keeping track of which objects have been constructed and then destroying those (and only those) before re-throwing the exception to signal that the **uninitialized_move()** failed. Yes, writing serious foundational software is challenging, but also important and at times exciting!

With **reserve()** coping with exceptions, **push_back()** and **resize()** work as written in §18.2.1, so we can look at assignments again:

```
template<typename T, typename A>
Vector<T>& Vector::operator=(const Vector<T>& arg)
{
        auto tmp = arg;             // copy all elements
        swap(*this,arg);            // then swap (Vector handles): strong guarantee
        return *this;
}
```

We first copy and then swap. That means that for v1=v2, v1 gets v2's elements and the old elements from v1 (now in tmp) are destroyed upon exit from operator=(). We never get to the swap, if the copy throws. So an exception thrown from the copy operation will have no effect on either v1 or v2. That is, our operator=() offers the strong guarantee.

Vector's swap() just moves the pointers in the Vector objects (§7.4.5), so it provides the no-throw guarantee. If it wasn't for pointers and moves, we wouldn't be able to provide any useful guarantees.

We can even write that operator=() even simpler:

```
template<typename T, typename A>
Vector<T>& Vector::operator=(const Vector<T> arg)
{
        swap(*this,arg);            // then swap (Vector handles): strong guarantee
        return *this;
}
```

The pass-by value (§7.4.3) makes the copy for us.

But what about the optimized version of Vector::operator=() that copies elements straight into its target if there is sufficient room (§17.8.5):

```
template<typename T, typename A>
Vector<T>& Vector::operator=(const Vector<T>& arg)
{
        if (arg.size()<=size()) {                          // enough space; copy directly
                move(arg.r.elem,arg.r.elem+arg.size(),r.elem);
                destroy(r.elem+arg.size(),r.elem+size());  // destroy surplus elements
        }

        auto tmp = arg;             // copy all elements
        swap(*this,arg);            // then swap (Vector handles): strong guarantee
        return *this;
}
```

We move, so we can only offer the basic guarantee: elements may have been overwritten before an exception was thrown. Also this optimized operator=() is twice the size of the elegant, "pure swap()" version. Is the optimization worth the effort and the loss of the strong guarantee? In the case of vector, the answer is yes: the difference in performance can be very significant and there are many important performance-sensitive applications where vectors of the same or smaller size are assigned from. For example, think of matrix computations where matrix elements are stored in vectors. So, std::vector's = is optimized. If you need the strong guarantee, you can easily write a function to give that:

```
template<typename T>
strong_assign(Vector<T>& target, const Vector<T> arg)
{
        swap(target,arg);           // then swap (vector handles): strong guarantee
}
```

Don't optimize without reason, without being reasonably sure that an optimization is worthwhile in important code.

18.5 Resource-management pointers

There are many common uses of pointers, such as:

- Returning large objects from functions
- Passing interfaces to polynomic objects as arguments (OOP)
- Passing the position of elements of a container to an algorithm (Chapter 21)
- Implementing high-level (easy to use, efficient, RAII, etc.) types, such as **vector**

To cope, we need to distinguish different possible uses of pointers.

- Where are pointers necessary? Often we have superior alternatives (§16.4, §18.5.1)
- Where does a pointer just identify an object?
- Where does a pointer identify a sequence of objects? Don't use built-in pointers to identify sequences of elements; they don't know how many elements they point to (§16.1). Use a container (Chapter 19) or a **span** (§16.4.1) instead.
- Where does a pointer represent ownership (i.e. needs to be deleted)? (§18.5.2, §18.5.3)

Here, we will consider the first and the last of these cases.

18.5.1 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 **delete**s 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 **delete**d in case of an exception? Unless we are systematic about the management of pointers the answer will be something like "Well, we think so," and that's not good enough:

```
Vector<int>* make_vec()      // make a filled vector
{
         auto res = new Vector<int> res;
        // ... fill the Vector with data; this may throw an exception ...
        return res;       // return a pointer
}

auto p = make_vec();
// ...
delete p;
```

Someone, some day, will forget to **delete** some result returned by this **make_vec()**.

Fortunately, when we added move operations to **Vector**, we solved that problem for **Vector**s: 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
}

auto v = make_vec();
```

This version of make_vec() is far simpler 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 (we can move an fstream but not copy it). 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.

18.5.2 unique_ptr

The technique of returning an object is ideal in many cases, but what if we really need a pointer? Consider:

```
Shape* read_shape(istream& is)
{
        // ... read a variety of shapes ...
}
```

This is OK if "someone" keeps track of the returned pointer and deletes the object pointed to when it is no longer needed. However, that's not common. We need to indicate that the returned pointer represents ownership; that is, it must be deleted. Here is a way:

```
unique_ptr<Shape> read_shape(istream& is)        // pseudo code
{
        // ... read a variety of shapes ...
        // ... if we read a Circle ...
                return make_unique<Circle>(center,radius);
}
```

A unique_ptr is an object that holds a pointer and represents ownership of what that pointer points to. We create one using std::make_unique(). Because unique_ptr is a kind of pointer, inheritance works correctly: we can return a unique_ptr<Circle> as a unique_ptr<Shape> and our OOP techniques are still useful.

You can use -> and * on a unique_ptr exactly like a built-in pointer. However, the unique_ptr owns the object pointed to: when the unique_ptr is destroyed, it deletes the object it points to.

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_ptrs 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 = make_unique<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 (§18.5.3). However, that is a more heavyweight solution that involves a use count to ensure that the last copy destroyed destroys the object referred to.

In contrast, unique_ptr has no overhead compared to an ordinary pointer.

In addition to being used as a way of transferring ownership, a unique_ptr can be used to guarantee deletion in case of an exception being thrown:

```
void trouble(int n)
{
        int∗ naked = new int[n];
        auto covered = make_unique<int[]>(n);
        // ... a return or a throw somewhere here ...
        delete[] naked;
}
```

The make_unique protects against the likely leak but the ‘‘naked new’’ does not.

Finally, a unique_ptr can be used instead of a ‘‘naked’’ pointer to represent ownership. Consider §15.5. There, we use a destructor to handle the deletion of the elements and the memory holding them. With a unique_ptr we can make that implicit and thus eliminate the possibility of getting the destructor wrong: CC

```
template<Elem T>
class Vector {
public:
        Vector(int s) :sz{s}, elem{std::make_unique<T[]>(s)} { }        // construct a Vector
        T& operator[](int i) { return elem[i]; }                         // element access: subscripting
        int size() { return sz; }
        // ...
private:
        int sz;                           // the number of elements
        std::unique_ptr<T[]> elem;        // pointer to the elements
        int alloc;
};
```

The generated destructor for Vector (§15.5.1) will call elem’s destructor.

Why didn’t we use that simplification from the very beginning? We wanted to show how resource-management classes were constructed, and (of course?) unique_ptr is constructed with exactly the same techniques and basic language features as we showed for Vector.

18.5.3 Shared_ptr

What if you need many pointers to an object and it is not clear which pointer should be deleted? We can try to avoid this by having an object allocated in such a way that it ‘‘outlives’’ all pointers to it. That’s often ideal and can be achieved by having the object in an outer scope to all pointers to it. However, there are significant uses where that’s hard to achieve simply and reliably. In that case, we can use the standard-library shared_ptr. It is used much like unique_ptr:

```
shared_ptr<Shape> read_shape(istream& is)        // pseudo code
{
        // ... read a variety of shapes ...
        // ... if we read a Circle ...
                return make_shared<Circle>(center,radius);
}
```

However, shared_ptr differs from unique_ptr by allowing many copies. It keeps a use count, that is a count of shared_ptrs to a given object and when that use count goes to zero, the shared_ptr's destructor deletes the object pointed to. That's sometimes colloquially referred to as "last person out switch off the lights."

Please note that shared_ptr is still a pointer: you need to use * or -> to access its object and you can create circular references:

```
struct Slink {
        string name;
        shared_ptr<Slink> next;
};

auto p = make_shared<Slink>("Friday",nullptr);
auto q =make_shared<Slink>("viernes",p);
p->next = q;
```

Those two Slinks will live "forever." To break such loops, you need a standard-library weak_ptr, but if you can, it is best simply not to create such loops.

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] Make val private.
[6] Add a member function access() that returns a reference to val.
[7] Put the definition of access() outside the class.
[8] Do 4 again using access().
[9] Add a S<T>::operator=(const T&). Hint: Much simpler than §17.8.5.
[10] Provide const and non-const versions of access().
[11] Define a function template<typename T> read_val(T& v) that reads from cin into v.
[12] Use read_val() to read into each of the variables from exercise 3 except the S<vector<int>> variable.
[13] 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 have different element types for different vectors?
[2] What is a template?
[3] What is generic programming?

[4] How does generic programming differ from object-oriented programming?
[5] What is a concept?
[6] What benefits do we get from the use of concepts?
[7] Name four standard-library concepts.
[8] How does resize() differ from reserve()?
[9] What is a resource? Define and give examples.
[10] What is a resource leak?
[11] List the three resource-management guarantees.
[12] How can the use of a built-in pointer lead to a resource leak? Give examples.
[13] What is RAII? What problem does it address?
[14] What is unique_ptr good for?
[15] What is shared_ptr good for?

Terms

constraint	owner	specialization	at()
push_back()	strong guarantee	basic guarantee	RAII
template	exception	resize()	template parameter
guarantees	resource	handle	re-throw
throw;	instantiation	unique_ptr	concept
range checking	shared_ptr	regular	equality_comparable
sortable	requires	throw;	

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 add() that adds the elements of one vector<T> to the elements of another; for example, add(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 (§6.8).
[4] Modify class Link from §15.7 to be a template with the type of value as the template argument. Then redo exercise 13 from Chapter 16 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 (§18.2) using the most basic standard-library allocation functions malloc() and free(). Get Vector as defined by the end of §18.3 to work for a few simple test cases.
[9] Re-implement Vector::operator=() (§17.8.5) using an allocator (§18.2) for memory management.
[10] Implement a simple unique_ptr supporting only a constructor, destructor, –>, *, and release(). Delete the assignment and copy constructors.
[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 16. 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.
[17] Define a function finally() that takes a function object as its arguments and returns an object with a destructor that invokes that function object. Thus

```
auto x = finally([](){ cout<< "Bye!\n"; });
```

will "say" Buy! whenever we exit the scope of x. For what might finally() be useful?

19

Containers and Iterators

Any problem in computer science can be solved
with another layer of indirection.
Except, of course,
the problem of too many indirections.
– David J. Wheeler

This chapter and the next two 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 Chapter 21. As an example, we present a framework for text editing as a sample application.

19.1 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 and uses pointers, whereas Jill stores hers in a vector and relies on move semantics (§17.4.4). 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 Chapter 9 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 fills an array and puts the number of elements in *count
vector<double> get_from_jill();         // Jill fills a 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;
}
```

Assume that we can't rewrite Jack's and Jill's code or wouldn't want to.

19.1.1 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 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 (double& x : jill_data)
        if (h<x) {
                jill_high = &x;         // save address of largest element
                h = x;                  // update "largest element"
        }

cout << "Jill's max: " << *jill_high
        << "; Jack's max: " << *jack_high;
```

19.1.2 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. CC

Obviously, what we do for Jack's data strongly resembles what we do for Jill's. However, there are annoying differences: Jack's traditional **for**-loop and subscripting vs. Jill's range-**for**. We could eliminate the latter difference by using a traditional **for**-loop for Jill's data:

```
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);
double* jill_high = high(&jill_data[0],&jill_data[0]+jill_data.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:

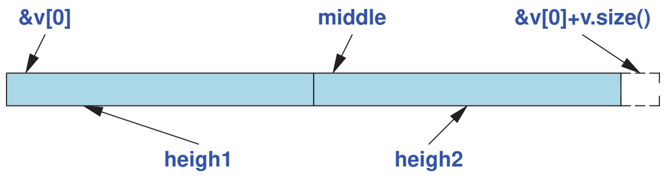
- It works for arrays only. We rely on the elements of a vector being stored in an array, but there are many more ways of storing data, such as lists (§19.3) and maps (§20.2).
- It can be used for vectors and arrays of doubles, but not for arrays or vectors with other element types, such as vector<double*> and char[10].
- It finds the element with the highest value, but there are many more 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 = nullptr;
        for (int& x : v)
                if (h<x) {
                        high = &x;
                        h = x;
                }
        return high;
}
```

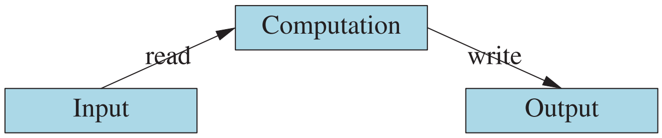
However, that wouldn't give us the flexibility we "accidentally" obtained from high() – we can't use find_highest() to find the element with the highest value in part of a vector. We actually achieved a practical benefit from writing a function that could be used for both arrays and vectors by "messing with pointers." We will remember that: generalization can lead to functions that are useful for more problems.

19.1.3 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 9) and C-style string functions (PPP2.Ch23), 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.

CC

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 ideally has to elegantly interact with the "data part."



AA 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 Chapter 8, §9.3.2, and §11.7.4.

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 lexicographically 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 handcraft 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 42*nd* 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. **AA**

- 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

CC 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, ...

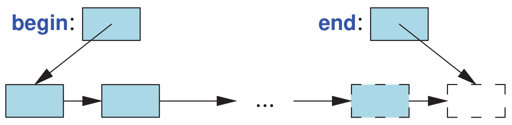
The STL provides that, and more. We will look at it not just as a very useful set of facilities, but also as an example of a library designed for maximal flexibility and performance. The STL was designed by Alex Stepanov to provide a framework for general, correct, and efficient algorithms operating on data structures. The ideal was the simplicity, generality, and elegance of mathematics.

XX 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 designers, 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.

19.2 Sequences and iterators

CC The central concept of the STL is the *sequence*, also called a *range*. 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.

CC What is an iterator? An iterator is a rather abstract notion. For example, if p and q are iterators to elements of the same sequence:

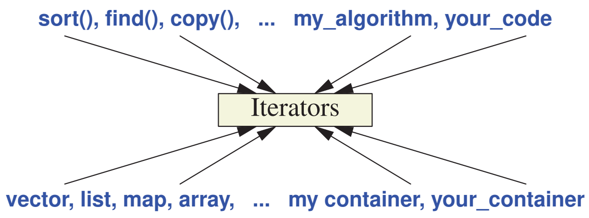
Basic standard iterator operations	
p==q	true if and only if p and q point to the same element or both point to one beyond the last element
p!=q	!(p==q)
*p	refers to the element pointed to by p
*p=val	writes to the element pointed to by p
val=*p	reads from the element pointed to by p
++p	makes p refer 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 (§15.4). In fact, a pointer to an element of an array is an iterator. However, many iterators are not just pointers; for example, we can define a range-checked iterator that throws an exception (§20.7.2) 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(), copy(), ... my_algorithm, your_code
Iterators
vector, list, map, array, ... my container, your_container

In other words, 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, data providers no longer have to write code to serve a bewildering variety of users; they just have to implement an iterator for their kind of data. At the most basic level, an iterator is defined by just the *, ++, ==, and != operators. That makes them simple and fast. To further increase flexibility and their range of uses, all STL containers are parameterized with their element type (§18.1.1) and an allocator (§18.2.1).

The STL framework consists of about a dozen containers and about 125 algorithms connected by iterators (Chapter 21). That's about 1,500 combinations represented by about 150 pieces of code. In addition, many organizations and individuals provide containers and algorithms in the style of the STL. The STL is probably the best-known and most widely used example of generic programming (§18.1.2, §21.1.2). If you know the basic concepts and a few examples, you can use the rest.

19.2.1 Back to the Jack-and-Jill 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<forward_iterator Iter>
Iter high(Iter first, Iter last)
        // return an iterator to the element in [first:last) that has the highest value
{
        Iter high = first;
        for (Iter 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 is 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 (§3.3.1, §6.6.1, §9.9.1, 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 fills an array and puts 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);
        double* jill_high = high(jill_data.begin(),jill_data.end());

        cout << "Jill's high " << *jill_high << "; Jack's high " << *jack_high;
        delete[] jack_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.

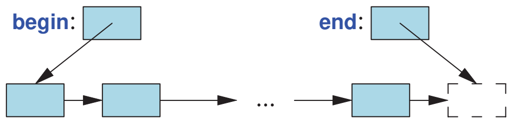
TRY THIS

We again left a serious error in that program. Find it, fix it, and suggest a general remedy for that kind of problem.

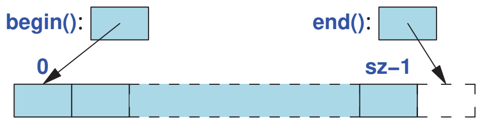
The process of analyzing pieces of code for similarities and generalizing them into a single piece of code is often called *lifting*. We can lift the Jack-and-Jill example further using ideas from the STL and the STL itself; see the exercises.

19.3 Linked lists

Consider again the graphical representation of the notion of a sequence: CC



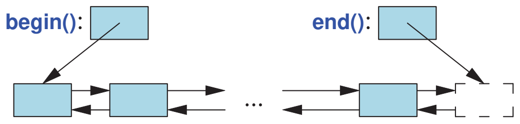
Compare it to the way we visualize a **vector** in memory:



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.

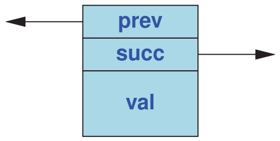
The data structure most directly suggested by the STL sequence diagram is called a *linked list*. The arrows in the abstract model are usually implemented as pointers. An element of a linked list is part of a "link" consisting of the element and one or more pointers. A linked list where a link has just 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<Element T>
struct Link {
        Link∗ prev;            // previous link
        Link∗ succ;            // successor (next) link
        T val;                 // the value
};

template<Element T> struct List {
        Link<T>∗ first;
        Link<T>∗ last;         // one beyond the last link
};
```

We can present the layout of a Link like this



There are many ways of implementing linked lists and presenting them to users. 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. See also the list example in §15.7.

19.3.1 List operations

CC

What operations do we need for a list?

- The operations we have for vector (constructors, size, etc.) (§18.1.1), except subscripting.
- Operations insert() (add an element) and erase() (remove an element); vector also has those (see §19.4.2).
- 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<Element T>
class List {
        // ... representation and implementation details ...
public:
        // ... constructors, destructor, etc. ...

        class iterator;              // member type: iterator

        iterator begin();            // iterator to first element
        iterator end( );             // iterator to one beyond last element

        iterator insert(iterator p, const T& v);    // insert v into list after p
        iterator erase(iterator p);                 // remove p from the list

        void push_back(const T& v);                 // insert v at end
        void push_front(const T& v);                // insert v at front
        void pop_front();                           // remove the first element
        void pop_back();                            // remove the last element

        T& front();       // the first element
        T& 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 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 §16.1.1 and §17.6. This style of iterators is the key to the standard-library algorithms (§21.1).

XX

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 §19.5.2). Consequently, std::list doesn't provide the innocuous-looking subscript operator, [ ].

We made List's iterator type a member (a nested class) because there was no reason for it to be global. 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.

19.3.2 Iteration

The List iterator must provide *, ++, ==, and !=. Since std::list is a doubly-linked list, it also provides -- for iterating "backward" toward the front of the list:

```
template<Element T>
class List<Elem>::iterator {
        LinkT>* curr;                   // current link
public:
        iterator(Link<T>* p) :curr{p} { }

        iterator& operator++() {curr = curr->succ; return *this; }             // forward
        iterator& operator--() { curr = curr->prev; return *this; }            // backward
        T& 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<T>::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<input_iterator Iter>
Iter high(Iter first, Iter last)
        // return an iterator to the element in [first,last) that has the highest value
{
        Iter high = first;
        for (Iter p = first; p!=last; ++p)
                if (*high<*p)
                        high = p;
        return high;
}
```

We can use it for a list or 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';
}
```

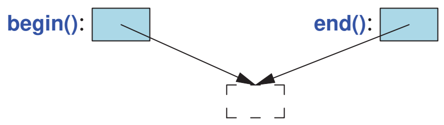
Here, the "value" of high()'s Iter argument is list<int>::Iterator, and the implementation of ++, *, and != has changed dramatically from the array case, but the meaning is still the same. The template function high() still traverses the data (here a list) and finds the highest value. We can insert an 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.

CC 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 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).

19.4 Generalizing Vector yet again

The standard-library vector has an iterator member type and begin() and end() member functions (just like std::list). However, those we provided for our Vector in §18.1.1 just returned pointers. What does it really take for different containers to be used more or less interchangeably in the STL generic programming style? First, we'll outline the solution and then explain it:

```
template<Element T, Allocator A = allocator<T>>
class Vector {
public:
        using size_type = int;
        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();

        // ...
};
```

CC 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';
```

To write that, we don't actually have to know what types are named by iterator and size_type. In particular, because the code above is expressed in terms of iterator and size_type, it will work with a variety of standard library containers and on a variety of systems with different representations of container types and iterator types. For example, we can provide a range-checked iterator type (§20.7.2). The size_type is usually an unsigned integer type (PPP2.§25.5.3) but here we have used int for simplicity. To safely convert between signed and unsigned types, we use narrow (§7.4.7).

The standard defines list and the other standard containers similarly. For example:

```
template<Element T, Allocator A = allocator<T>>
class List {
public:
        using size_type = int;
        using value_type = T;
        class Link;
        class iterator;                // see §19.3
        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();
        // ...
};
```

19.4.1 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';
}
```

However, this doesn't work for lists because list does not provide subscripting. However, we can traverse a standard-library vector and list using iterators. For example: CC

```
void print2(const vector<double>& v, const list<double>& lst)
{
        for (vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
                cout << *p << '\n';

        for (list<T>::iterator p = v.begin(); p!=v.end(); ++p)
                cout << *p << '\n';
}
```

This works for both the standard-library containers and for our Vector and List. CC

In generic code, type names can get unpleasantly long. We don't really like to type names like vector<T>::iterator repeatedly and we know that the value returned by begin() is an iterator of the appropriate type. So we typically use auto to simplify such code:

```
void print2(const vector<double>& v, const list<double>& lst)
{
        for (auto p = v.begin(); p!=v.end(); ++p)
                cout << *p << '\n';

        for (auto p = v.begin(); p!=v.end(); ++p)
                cout << *p << '\n';
}
```

When we don't need the position of an element, we can do better still. The range-for-loop is simply "syntactic sugar" for a loop over a sequence using iterators, so we get:

```
void print3(const vector<double>& v, const list<double>& lst)
{
        for (double x : v)
                cout << x << '\n';

        for (double x : lst)
                cout << x << '\n';
}
```

The techniques used for print2() and print3() can be used for all standard-library containers (§20.6).

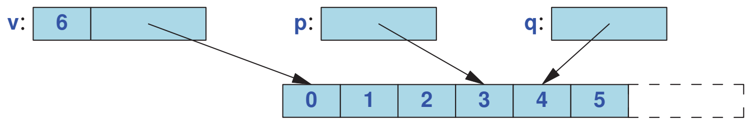
19.4.2 insert() and erase()

AA 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.

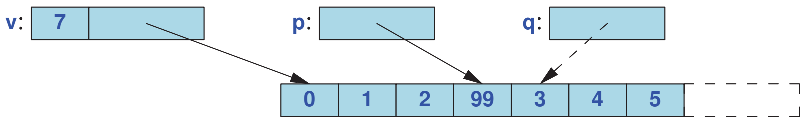
XX Moving elements also implies a logical constraint: don't hold iterators or pointers to elements of a vector when you do list operations (such as insert(), erase(), and push_back()): if an element moves, your iterator or pointer will point to the wrong element or to no element at all. This is the principal advantage of lists (and maps (§19.4.2) 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
vector<int>::iterator q = p;
++q;                                        // point to its 5th element
```

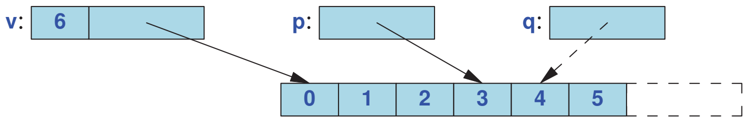


```
p = v.insert(p,99);                    // p points to the inserted element
```



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.

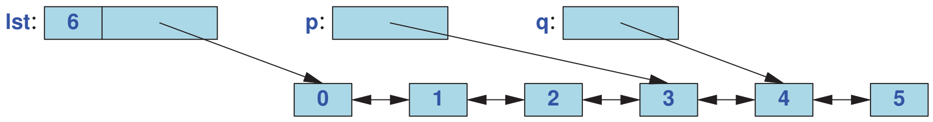
```
p = v.erase(p);                        // p points to the element after the erased one
```



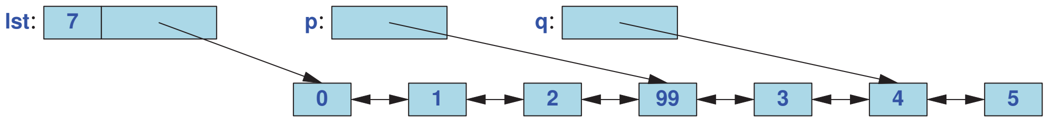
That is, an **insert()** followed by an **erase()** of the inserted element leaves us back where we started, but with **q** invalidated. However, in between, we moved all the elements after the insertion point, and maybe all elements were relocated as **v** grew.

To compare, we'll do exactly the same with a **list**:

```
list<int>::iterator p = lst.begin();   // take a list
++p; ++p; ++p;                         // point to its 4th element
list<int>::iterator q = p;
++q;                                   // point to its 5th element
```

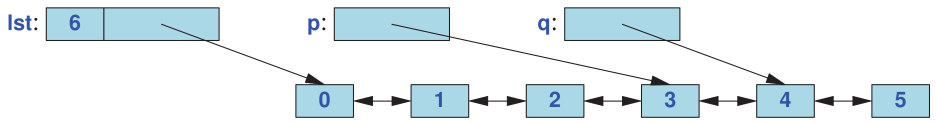


```
p = v.insert(p,99);                    // p points to the inserted element
```



Note that **q** still points to the element with the value **4**.

```
p = v.erase(p);                  // p points to the element after the erased one
```



Again, we find ourselves back where we started. However, for **list** as opposed to for **vector**, we didn't move any elements and **q** was valid at all times.

19.4.3 Adapting our vector to the STL

After adding **begin()**, **end()**, and the type aliases in §19.4, our **Vector** now just lacks **insert()** and **erase()** to be as close an approximation of **std::vector** as we need it to be:

```
template<Element T, Allocator A = allocator<T>>          // §18.4.4
class Vector {
        Vector_rep<A> r;
public:
        // ...
        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).

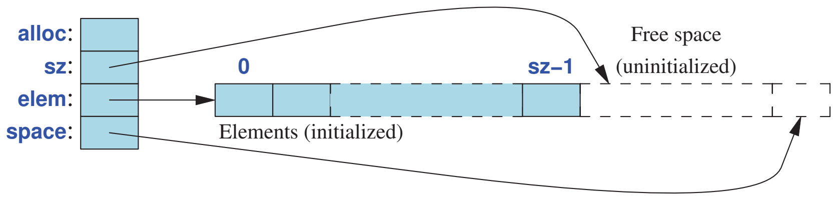
AA Typically, people don't provide list operations, such as **insert()** and **erase()**, for data types that keep their elements in contiguous storage, such as **vector**. However, list operations, such as **insert()** and **erase()**, are immensely useful and surprisingly efficient for **vector**s with small elements 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 §18.1.1 with the additions from §18.4.4 and §19.4, we get

```
template<Element T, Allocator A>
Vector<T,A>::iterator Vector<T,A>::erase(iterator p)
{
        if (p==end())
                return p;
        move(p+1,r.sz,p);                     // move each element one position to the left
        destroy_at(r.elem()+r.sz−1));         // destroy surplus last element
        −−r.sz;
        return p;
}
```

We use standard-library functions **move()** and **destroy_at** to avoid messier lower-level facilities.

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<Element T, Allocator A>
Vector<T,A>::iterator Vector<T,A>::insert(iterator p, const T& val)
{
        int index = p–begin();                      // save index in case of relocation
        if (size()==capacity())
                reserve(size()==0?8:2*size());      // make sure we have space
        p = begin()+i;                              // p now points into the current allocation
        move_backward(p,r.sz–1,p+1);                // move each element one position to the right
        *(begin()+index) = val;                     // "insert" val
        ++r.sz;
        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.
- We use the standard-library function move_backward() instead of std::move() to make sure that the last element is the first to be moved. Plain move() would move the first element first, thus overwriting the second before that was moved.
- It is subtleties like these that make us avoid dealing with low-level memory issues whenever we can. Naturally, std::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 usually use insert() and erase() in the middle of a 100,000-element vector; lists and maps are designed for that (§19.4.2). 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) `list`s for representing a list of a few small elements.

19.5 An example: a simple text editor

CC 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

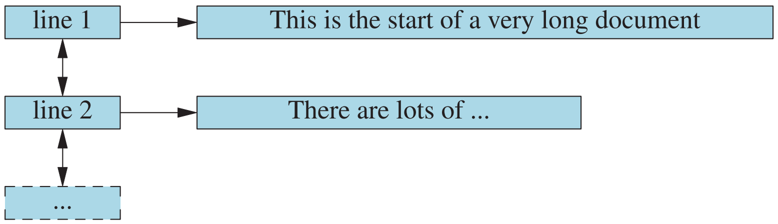
The simplest representation would be a `vector<char>`. However, to add or delete a character we would have to move every following character in the document. Consider:

```
This is he 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 ...
```

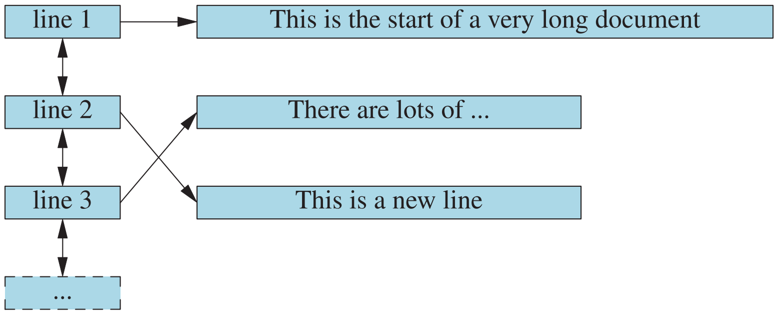
However, if those characters were stored in a single `vector<char>`, we'd have to move every character from `h` onward one position to the right. That could be a lot of copying. In fact for a 70,000-character-long document (such as this chapter, counting spaces), we would, on average, have to move 35,000 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:



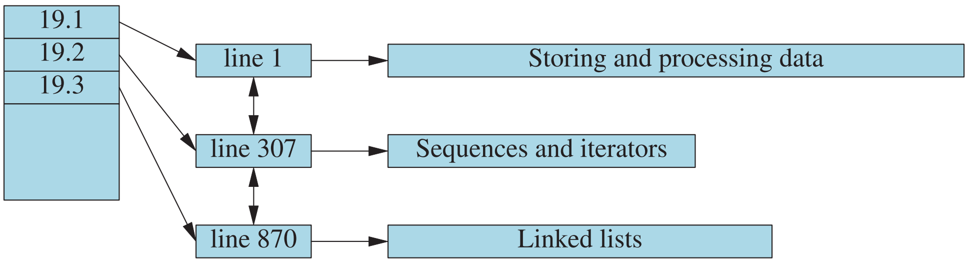
Now, when we inserted that **t**, we only had to move the rest of the characters on that line. Furthermore, when we need to, we can add a new line without moving any characters. For example, we could insert **This is a new line.** after **document.** to get

```
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 19.2” without invalidating the iterator to “paragraph 19.3.”

AA 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 §19.6. 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.

19.5.1 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., . (dot)).
[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).

19.5.2 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:
        Text_iterator(list<Line>::iterator ll, Line::iterator pp)
                // line ll's character position 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 can now iterate over the characters of a document like this:

```
void print(Document& d)
{
        for (auto p : d)
                cout << p;
}
```

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 (!(0<=n && n<d.line.size()))
                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<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 **n**th 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**.

AA 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**, **std::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 (e.g., **milkshake** and **Gavin**)
- for sequences of letters that can't easily be considered words. (e.g., **secret\nhomestead** – i.e., a line ending with **secret** followed by a line starting with **homestead**)
- for regular expressions (e.g., **[bB]\w*ne** – i.e., an upper- or lowercase **B** followed by zero or more letters followed by **ne**). See PPP2.§23.5-9.
- and more.

Let's see 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)
        // find s in [first:last)
{
        if (s.size()==0)                // can't find an empty string
                return last;
        char first_char = s[0];

        auto p = last;
        for (p = find(first,last,first_char); !(p==last || match(p,last,s)), ++p)
                // do nothing
                ;
        return p;
}
```

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 (§21.1.1). 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.

19.6 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 five ways to store a sequence of characters:

- char[N] (array of N characters)
- array<char,N> (std::array of N characters; §20.6.2)
- vector<char>
- string
- list<char>

CC 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?

- T[N]: Doesn't know its own size. Doesn't have begin(), end(), or any of the other useful container member functions. Instead, we can use begin(a) and end(a) for an array, though (obviously) not for a pointer. 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.
- array<T,N>: Like T[N], but with no implicit conversion to a pointer (§20.6.2). Has the usual value semantics for copying and comparison.
- vector<T>: 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: Elements are characters. 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<T>: 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. It can be expensive to iterate over the elements. A list is expandable. Comparison operators (==, !=, <, <=, >, and >=) compare elements.

As we have seen (§16.1), arrays are useful and necessary for dealing with memory at the lowest level and for interfacing with code written in C (PPP2.§27.1.2, PPP2.§27.5). Apart from that, vector is preferred because it is easier to use, more flexible, and safer. AA

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 to be optimized for 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 erase(). 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. AA

TRY THIS

What does that list of differences mean in real code? For each 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 }.

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<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 `std::copy()`. 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 `std::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?

Terms

algorithm	empty sequence	singly-linked list	list
array container	end()	size_type	auto
erase()	STL	begin()	insert()
traversal	container	iteration	using
contiguous	iterator	type alias	doubly-linked list
range	linked list	value_type	element
sequence	lifting	generalize	

Exercises

[1] If you haven't already, do all **TRY THIS** exercises in the chapter.

[2] Get the Jack-and-Jill example from §19.2.1 to work. Use input from a couple of small files to test it.

[3] Look at the palindrome examples (§16.5); redo the Jack-and-Jill example from §19.2.1 using that variety of techniques.

[4] Find and fix the errors in the Jack-and-Jill example from §19.2.1 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 §19.5.

[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 white-space 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 §19.3 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 nullptr 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 holds 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 11.

[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 §20.4. 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.

[21] Define and test a Checked_iterator; that is, an iterator that "knows" the size of its range and throws an exception if we try to access outside that. Hint: §20.7.2.

Postscript

CC 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 about 12 containers and about 125 algorithms (not even counting the range algorithms), the brute-force approach would require 1,500 functions, whereas the STL strategy requires only 125 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. copy()). In addition, the STL provides conventions for defining algorithms that simplify writing correct code and composable code, so the saving is greater still.

20

Maps and Sets

Write programs that do one thing
and do it well.
Write programs to work together.
– Doug McIlroy

The C++ standard library offers more containers than `vector` and `list`. In this chapter, we introduce the associative containers `map` and `unordered_map` in which we can look up a value by giving a "key." For example, looking for a record by quoting a name. Such types are pervasive in applications because they simplify and speed up common tasks. Some types are not standard-library containers, but can be seen as a sequence of elements. Examples are built-in arrays and I/O streams. The notion of iterators is so flexible that such types can be fitted into the framework. To benefit from that flexibility, we generalize ranges.

20.1 Associative containers

CC After vector, the most useful standard-library container is probably the map. A map (§20.2) 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 Nicholas' phone number. The only potential competitor to map in a popularity contest is unordered_map (§20.3), 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*. In the standard library, we collectively call such data structures *associative containers*. Popular and useful concepts always seem to have many names.

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

20.2 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 pair<string,int>& 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"`. CC

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` (§20.2.2):

```
for (const pair<string,int>& 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* [TC++PL] 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 1.

20.2.1 Structured binding

Looking at `map` nodes is an example of using functions returning more than one value. In fact, dereferencing a `map` iterator gives us a `pair` with members `first` and `second`. Why `first` and `second`? Well, why not? The implementer of `pair` had to call them something. On the other hand, as users of this particular `pair<string,int>`, we think of those values as `key` and `value`. A mechanism called *structured binding* lets us name members of structures returned from functions. For example:

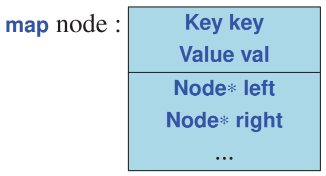
```
for (const auto& [key,value] : words)
        cout << key << ": " << value << '\n';
```

The iteration over `words` returns references to `pair<key,value>` objects. The structured binding simply allows us to use our preferred names `key` and `value` for those `pair`'s `first` and `second`. No additional copying is done.

Like all powerful mechanisms, structured binding can be overused and lead to obscure code, but in this case, as in many others, we think it makes the code more readable.

20.2.2 map overview

CC So what is a `map`? There are 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 made out of nodes:

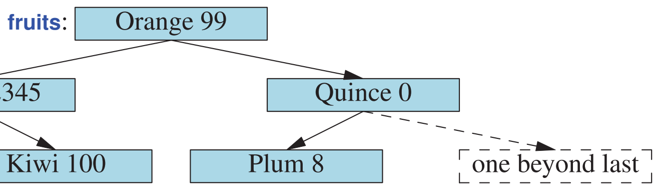


A **Node** holds a key, its corresponding value, and pointers to two descendant **Node**s.

Assuming a type **Fruit** with a suitable constructor, we can write

```
map<Fruit,int> fruits = { {Kiwi,100}, {Quince,0}, {Plum,8}, {Apple,7}, {Grape,2345}, {Orange,99} };
```

This generates a data structure like this:



Calling the key member of a **map<Fruit,int>** node **key**, the basic rule of a binary search tree is

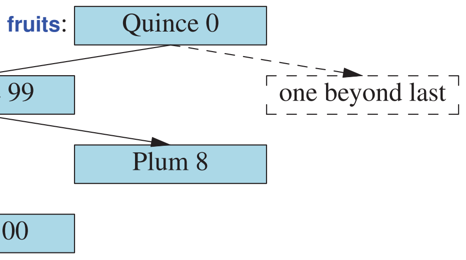
```
left–>key<key && key<right–>key
```

That is, for every node,

- Its left sub-node's key is less than the node's key, and
- The node's key is less than its right sub-node's key

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. **AA**

A balanced-tree **Node** typically also holds some data which the map will use to keep its tree of nodes balanced. If a tree with *N* nodes is balanced, we have to at most look at *log2(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 §20.3.) 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–>key<key && key<right–>key
```

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, binary_operation<Value> Cmp = less<Key>>
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?

        iterator erase(iterator p);                         // remove element pointed to by p
        pair<iterator, bool> insert(const value_type&);     // insert a (key,value) pair
        // ...
};
```

CC 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** (§19.4) is obvious. The main difference is that when you iterate over a **map**, the elements pointed to by the iterators 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
        T1 first;
        T2 second;
        // ...
};
```

CC Note that when you iterate over a **map**, the elements will come in the order defined by the key. For example, using structured binding (§20.2.1) to get nicer names than **first** and **second**:

```
for (const auto& [key,value] : fruits)
        cout << '(' << key << ',' << value << ") ";
```

gives:

```
(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.

CC 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 §21.5. By default the order is defined by less<Key>, meaning "less than."

20.2.3 Another map example

To better appreciate the utility of map, consider a stock market index. The way that works is to take a set of companies and assign each a "weight." For example, when last we looked in the Dow Jones Industrial Index, Alcoa had the weight of 2.4808. 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. 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 = 0;
for (size_t i = 0; i<dow_price.size(); ++i)
        dji_index += dow_price[i]*dow_weight[i];

cout << "DJI value " << dji_index << '\n';
```

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 an 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",104.48},
        {"AAPL",165.02},
        {"MSFT",285.76},
        // ...
};
```

The (symbol,weight) map:

```
map<string,double> dow_weight = {       // Dow (symbol,weight)
        {"MMM", 2.41},
        {"AAPL",2.84},
        {"MSFT",4.88},
        // ...
};
```

The (symbol,name) map:

```
map<string,string> dow_name = {  // Dow (symbol,name)
        {"MMM","3M"},
        {"AAPL","Apple"},
        {"MSFT","Microsoft"},
        // ...
};
```

Given those maps, we can conveniently extract all kinds of information. For example:

```
double caterpillar = dow_price ["CAT"];                 // 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:

```
for (const auto& [symbol,price] : dow_price)
        cout << symbol << '\t' << price << '\t' << dow_name[symbol] << '\n';
```

AA 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.

20.3 unordered_map

CC 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 (*N); we call that cost O(N).*

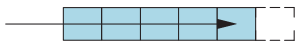
To find an element in a map, the subscript operator needs to examine all the elements of the tree from the root to the element with the right value or to a leaf. On average, the cost is proportional to the depth of the tree. A balanced binary tree holding *N* elements has a maximum depth of *log2(N)*, so the lookup cost is *O(log2(N));* that is, cost proportional to *log2(N)*, and that is actually pretty good compared to *O(N)*:

N	16	128	1024	16∗1024	1024∗1024	1024∗1024∗1024
log2(N)	4	7	10	14	20	30

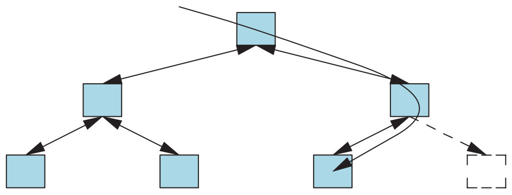
The actual cost will depend on how soon in our search we find our values and how expensive comparisons and iterations are. It is usually somewhat more expensive to chase pointers (as the map lookup does) than to increment a pointer (as find() does in a vector).

CC 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*, the function that computes it is called a *hash function*, 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 URLs. 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”). The standard library provides reasonable hash functions for the built-in types and strings. Graphically:

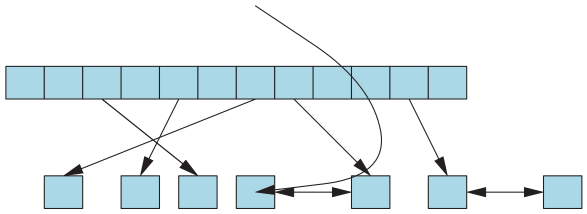
- Lookup in unsorted vector:



- Lookup in map (balanced binary tree):



- Lookup in **unordered_map** (hash table):



AA 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 §20.2.3 like this:

```
unordered_map<string,double> dow_price;

for (const auto& [symbol,price] : dow_price)
        cout << symbol << '\t' << price << '\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.

20.4 Timing

We have mentioned efficiency and the speed of operations. But how fast do these containers really operate? We consider talking about "efficiency" without concrete measurement suspect: Don't make claims about efficiency without backing up those claims with measurements. The big-O complexity measures (§20.3) are all very good and one – often excellent – thing to consider, but if performance might matter, we would like to measure how much time a piece of code really takes to run. Consider:

```
using namespace chrono;                // that's where the timing support is

auto t0 = system_clock::now();         // the point of time of the call
auto x = do_something();
auto t1 = system_clock::now();
cout << "res: " << x <<'\n';
cout << t1−t0 << '\n';                 // that's how long it took
```

The basic technique is that simple, but modern computers are fast, so this style of measurements requires a fair amount of computation to get meaningful results. It also requires that we run our tests repeatedly, say 3 or 5 times, to make it likely that our result isn't corrupted by something else going on on the computer.

A clock's now() function returns a point in time, a time point, so subtracting two time_points gives a period of time, a duration.

Why did we write res to output? Well, if we didn't use a result from do_something(), the optimizer would decide that we didn't need to run the code and eliminate it. That is, the time measured would be 0.

That said, let's try to measure the cost of lookup in a vector and a map using the random number generators from §4.7.5:

```
using namespace chrono;                                          // that's where the timing support is

vector<pair<string,int>> v = generate(1'000'000);               // generate some data (§4.7.5):
string x = v[v.size()/2].first;                                  // pick a string to search for (§20.2.2)

auto t0 = system_clock::now();                                   // the point of time of the call
auto pv = ranges::find_if(v, [&x](const auto& s) { return s.first == x; });         // linear search
auto t1 = system_clock::now();
cout << "vector: " << pv−>second << '\n';
cout <<  duration_cast<microseconds>(t1−t0).count() << "us\n\n";      // count() is a number of "clock ticks"

map<string, int> m {v.begin(), v.end()}; ;
auto t2 = system_clock::now();
auto vm = m[x];                                                  // tree search; may add an element
auto t3 = system_clock::now();
cout << "map[]: " << vm << '\n';
cout << duration_cast<microseconds>(t3−t2).count() << "us\n\n";

auto t22 = system_clock::now();
auto pm = ranges::find_if(m, [&x](const auto& s) { return s.first == x; });         // linear search
auto t32 = system_clock::now();
cout << "map find_if: " << pm−>second << '\n';
cout << duration_cast<microseconds>(t32−t22).count() << "us\n";
```

We got:

```
vector: 665618
3085us

map[]: 665618
2us

map find_if: 665618
81749us
```

The exact results will be different on your machine.

We could have output the results with the simpler statement:

```
cout << t1−t0 << "\n";
```

However, we preferred to be explicit about the unit of measurement and duration_cast does that; us is a common abbreviation for microsecond because it looks a bit like the Greek letter μ.

We find simple timing measurements immensely useful for getting a feel for run-time costs. They are not perfect, but far better than guessing. It is seriously hard to guess about the performance of real-world code on modern computers.

TRY THIS

Run your own version of that timing experiment. Run it at least three times and with at least three different numbers of elements. Explain your results.

In §4.7.5, we promised to give some details about how our random_int() functions worked. Here is the implementation:

```
namespace Random {
        using engine = default_random_engine;
        using distribution = uniform_int_distribution;

        engine ran;

        int random_int(int min, int max) { return distribution {min, max } (ran); }
        int random_int(int max) { return random_int(0, max); }
        void seed(int s) { Random::ran.seed(s); }
        void seed() { Random::ran.seed(random_device{}()); }
}
```

The PPP::random_int() and other functions are redundant; they just call their Random equivalents. In professional code, we'd use Random::random_int().

The reason for the namespace Random is to have a common engine for all uses, rather than a bunch of different ones, while still avoiding polluting the global namespace. Having a single engine improves randomness. The definition of seed() is a bit of black magic. It digs deep into the guts of the operating environment to gain almost perfect randomness, but it is still part of the ISO C++ standard library. The using-declarations allow us to quickly change our engine and distribution. We could have parameterized these functions with an integer type but decided to keep it all simple by sticking to int.

20.4.1 Dates

Dates has nothing to do with **unordered_map**s or containers in general, but after talking about timing and the notions of **time_point** and **duration**, we inevitably think about days, weeks, and dates.

Here is a simple example:

```
using namespace chrono;

auto now = system_clock::now();                    // clock's precision, probably microseconds or less
auto today = floor<days>(now);                     // round the time to days
cout << weekday(today) << '\n';                    // Tue
cout << format("{:%A}\n", weekday(today));         // Tuesday
```

First we get the time, then we round it down to the precision of days (rather than some much smaller unit of time), and then we write out the weekday. The default output was **Tue**, but if we don't like the abbreviation, we can get **Tuesday** by using **format()** (§9.10.6).

We can use the facilities in **chrono** to make calendars and operations on those. For example:

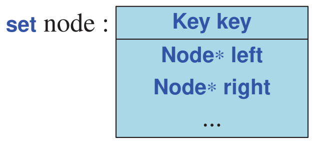
```
sys_days xmas = December/24/2023;
cout << "New Year " << xmas + days{7} << '\n';
```

In the likely event that you have to write code that deals with time on a human scale, read up on the standard-library **chrono** facilities for dates and time zones. This library component is far more likely to give correct results than someone's "homebrew" code. A standard-library component is tested and documented. Also, its designers are likely to have thought of many more real-world needs and constraints than we are likely to while working on a problem we need to solve in a reasonable time. For example, **chrono** handles time zones and leap seconds correctly. If your reaction to that nugget of information is "but time zones change in irrational ways every year!" and "what is a leap second?" you have made our point.

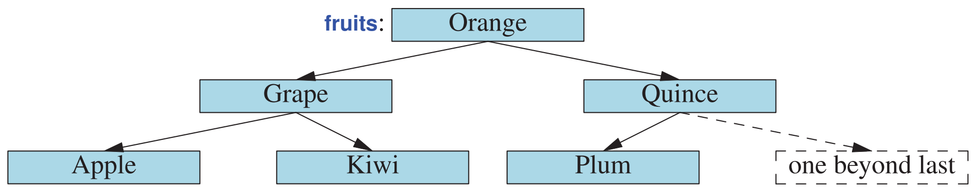
On top of that, the **chrono** library is blindingly fast.

20.5 Set

We can think of a **set** as a **map** where we are not interested in the values, or rather as a **map** without separate values. We can visualize a **set** node like this: CC



We can represent the **set** of fruits used in the **map** example (§20.2.2) 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 (§21.2) can significantly increase the range of problems for which an STL component is useful.

CC 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: 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';
```

20.6 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 when you need to insert and delete elements without moving existing elements
forward_list	a singly-linked list; use for lists that are mostly empty
deque	a cross between a list and a vector; don't use 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 (§20.2)
multimap	a balanced ordered tree where there can be multiple copies of a key; use it when you need to access elements by value
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 (§20.3)
unordered_multimap	a hash table where there can be multiple copies of a key; an optimized version of multimap; use for large maps when you need high performance and can devise a good hash function
set	a balanced ordered tree; use it when you need to keep track of individual values (§20.5)
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
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

You can look up incredible amounts of additional information on these containers and their use in books and online documentation.

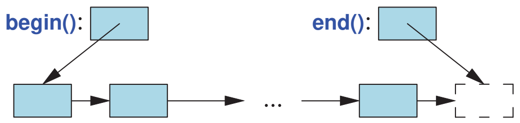
AA Do you feel cheated? Do you think we should explain all about containers and their use to you? That's just not possible. There are too many standard facilities, too many useful techniques, and too many useful libraries for you to absorb them all at once. Programming is too rich a field for anyone 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 when you need it. 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.

CC 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

- Has a sequence of elements [begin():end()).
- Provides copy operations that copy all elements.
- Provides move operations that move all elements.
- 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* (§20.7.1).
- Provides insert() and erase(), front() and back(), push_back() and pop_back(), size(), swap(), 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 their elements in order starting with the first.

The forward_list is an example of a container that offers a forward iterator, but no more. That is, we can traverse a forward_list from its beginning to its end, but not the other way.



For example:

```
forward_list<double> lst = {0.0,1.1,2.2,3.3};
for (auto p = lst.begin(); p!=lst::end(); ++p)          // OK (but verbose)
        cout << *p << '\n';
for (auto p = lst.end(); p!=lst::begin(); --p)          // error: can't go backwards (no --)
        cout << *p << '\n';
for (const int x : lst)                                 // OK
        cout << *p << '\n';
size_t sz = lst.size();                                 // error: a forward_list doesn't have size()
size_t sz2 = 0;
for (auto p = lst.begin(); p!=lst::end(); ++p)          // OK
        ++sz2;
```

Why no size()? By *not* storing the size, a forward_list can be represented as a single pointer to its first node, if any. That minimal size is important because a common use of forward_lists is where many thousands of usually empty containers are needed. If you want to know the number of elements of a forward_list, just count them.

Note that in the example we traversed lst three times. The difference between an input iterator and a forward iterator is that you can traverse a sequence repeatedly using a forward iterator.

20.6.1 Almost containers

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"	
T[n]	a built-in array; no size() or other member functions; prefer a container, such as vector, string, or array, over a built-in array when you have a choice.
array	a fixed-size array that doesn't suffer most of the problems related to the built-in arrays (§20.6.2).
string	holds only characters but provides operations useful for text manipulation, such as concatenation (+ and +=); prefer the standard string to other strings.
valarray	a numerical vector with mathematical 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. AA

20.6.2 array

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, etc. (§16.1). We have also pointed out their main strength: they model physical memory almost perfectly. To get the best of both worlds, the standard library offers array<T,N> that "knows" that its size is N (§16.4.2). As an example, we can write an array version of the high() example from §19.3.2:

```
void f()
{
        array<double,6> arr = { 0.0, 1.1, 3.3, 5.5, 2.2, 4.4 };
        auto p = high(arr.begin(), arr.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.

Needing to find the largest element in a sequence is common, so the standard library has a version. We could even have said:

```
auto p = ranges::max_element(arr);        // p is an iterator to the highest value element in arr
```

Given that array<T,N> knows its number of elements, we can (and do) provide assignment, ==, !=, etc. just as for vector.

20.6.3 Adapting built-in arrays to the STL

When we have the definition of a built-in array in scope, we can actually see how many elements it has. For example:

```
void f(int[] a1)        // int[] looks like an array, but really is just a pointer: int*
{
        int a2[10];     // obviously 10 elements
        // ...
}
```

We have all the information needed to refer to **a2** as a [begin:end) sequence, we just need a convenient way of referring to that. The standard library provides that **begin()** and **end()**:

```
void f(int[] a1)  // int[] looks like an array argument, but really is just a pointer: int*
{
        int a2[10];                     // obviously 10 elements
        // ...
        sort(begin(a2),end(a2));        // OK
        sort(begin(a1),end(a1));        // error: we don't know where the end of a1 is
        // ...
}
```

The definitions of **begin()** and **end()** use a general mechanism called *traits*. Look it up if you really need to, but most programmers don't need to. It also allows us to say **begin(v)** for a **vector** instead of **v.begin()**.

20.6.4 Adapting I/O streams to the STL

CC 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 the standard-library **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 **∗**

Now sequence is a very general concept, so we can fit input and output streams into that model. The standard library provides **ostream_iterator**, an iterator that you can use to write values of type **T**. For example:

```
ostream_iterator<string> oo {cout};        // assigning to *oo is to write to cout

*oo = "Hello, ";                           // meaning cout << "Hello, "
++oo;                                      // "get ready for next output operation"
*oo = "World!\n";                          // meaning cout << "World!\n"
```

You can imagine how this could be implemented.

Similarly, the standard library provides **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 XX

```
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 often 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
```

AA 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
```

20.6.5 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
}
```

CC 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.

Naturally, we could even have used `ranges::copy()`:

```
ranges::copy(b, ostream_iterator<string>{os," "});        // copy buffer to output
```

20.7 Ranges and iterators

We have now used ranges for a long while. They are our preferred way of using sequences and algorithms over sequences because they are easier to write and don't offer opportunities for mistakes that direct use of iterators does. Consider:

```
sort(v.end(),v.start());          // Oops!
sort(v1.start(),v2.end());        // Oops!
```

Ranges are less flexible than explicit use of iterators and that implies less opportunities for errors:

```
ranges::sort(v);                  // if this doesn't make sense the compiler will tell us
```

So far, we have considered a range to be defined by a pair of iterators referring to the first element (if any) and one-beyond-the-last element. However, the stream iterators show that this view is a simplification. In fact, we can define a range in three ways:

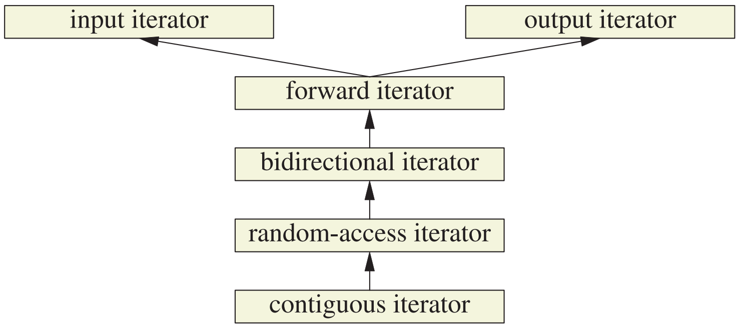
- `{begin,end}`: a pair of iterators
- `{begin,length}`: an iterator and a number of elements
- `{begin,predicate}`: an iterator and predicate to determine if the end has been reached

20.7.1 Iterator categories

We have mostly 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	Can iterate forward using `++` and read element values using `*`. This is the kind of iterator that `istream` offers; see §20.6.4. If `(*p).m` is valid, `p->m` can be used as a shorthand.
output iterator	Can iterate forward using `++` and write element values using `*`. This is the kind of iterator that `ostream` offers; see §20.6.4.
forward iterator	An input iterator that can iterate repeatedly over a sequence and repeatedly read from or write to an element. This is the kind of iterator that `forward_list` offers §20.6.
bidirectional iterator	A forward iterator that can move backward (using `--`). This is the kind of iterator that `list`, `map`, and `set` offer.
random-access iterator	A bidirectional iterator that can move forward and backwards `n` positions using `+=n`, `+`, `-=n`, and `-`.
contiguous iterator	A random-access iterator where the elements are allocated contiguously. This is the kind of iterator that `vector` offers.

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. There are concepts (§18.1.3) representing these iterator categories.

For each iterator category, there is a corresponding range category. For example:

```
template<input_iterator In>           // simplified
In find(In b, In e);

template<ranges::input_range R>
ranges::iterator_t<R> find(R r);
```

Naturally, since `R` is an `input_range`, its iterator, referred to as `iterator_t<R>`, is an `input_iterator`. You may wonder how we can have an input range when we can't have a pointer to the end of input, but the notion of iterators is sufficiently flexible to cope with the idea of "from the start of input to the end of input" (§20.6.4).

20.7.2 Output ranges

The standard library offers output iterators, but unfortunately not output ranges. That's a pity because an output range can be used to catch range errors. An output range is like an output iterator that kept track of which element in the range we are writing to. Consider a simple implementation of this idea:

```
template<ranges::range R>
class Output_range {
public:
        using value_type = ranges::range_value_t<R>;
        using difference_type = int;

        Output_range(R r) : b{ r.begin() }, e{ r.end() }, p{ b } {}
        Output_range& operator++() { check_end(); ++p; return *this; }
        Output_range operator++(int) { check_end(); auto t{ *this }; ++p; return t; }

        value_type& operator*() const { check_end();  return *p; }
private:
        void check_end() const { if (p == e) throw Overflow{}; }

        ranges::iterator_t<R> b;
        ranges::iterator_t<R> e;
        ranges::iterator_t<R> p;
};
```

Obviously, there is a cost compared to using an ordinary iterator, but then there is often a cost imposed by range errors.

This is not “industrial strength” or “ISO C++ standard-library quality”, but it actually works for a range of uses:

```
vector<int> v = {0,1,2,3,4,5};
vector<int> v1(10);
vector<int> v2(5);

ranges::copy(v,Output_range{v1});        // copies v into v1
ranges::copy(v,Output_range{v2});        // throws Range_error
```

The STL is an extensive framework. Sometimes we must extend it to get what we want. However, that is rarely a task for beginners.

Output_range is really an iterator that keeps track of the range it is iterating over, so we could have called it Checked_iterator.

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**.

Review

[1] What is a container?
[2] What containers does the STL provide?
[3] What is an associative container? Give at least three examples.
[4] Is **list** an associative container? Why not?
[5] What is a hash function?
[6] What is the basic ordering property of binary tree?
[7] What (roughly) does it mean for a tree to be balanced?
[8] How much space per element does a **map** take up?
[9] How much space per element does a **vector** take up?
[10] Why would anyone use an **unordered_map** when an (ordered) **map** is available?
[11] What is a big-O measure?
[12] How does a **set** differ from a **map**?
[13] How does a **multimap** differ from a **map**?
[14] How does an associative container differ from other containers?
[15] Why is an array not a container?
[16] Why bother to time code?
[17] How do you time a piece of code?
[18] When does structured binding clarify code? When does it obscure it?
[19] What containers does the STL offer?
[20] What is an iterator category?
[21] What kinds of iterators does the STL offer? Mention at least eight.
[22] What operations are provided by a random-access iterator, but not a bidirectional iterator?

Terms

associative container	**map**	associative array	structured binding
balanced tree	**unordered_map**	big-O	**duration**
now()	**time_point**	clock	date
set	container	almost container	**array**
istream_iterator	**ostream_iterator**	range	iterator category
random-access iterator	output range	hash function	**ranges::**

Exercises

[1] Rewrite the word-counting program from §20.2 to not distinguish between uppercase and lowercase letters and to treat punctuation characters as whitespace.

[2] In the Fruit example (§20.5), 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.

[3] Take the word-frequency example from §20.2 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.

[4] Read a file of integers into a vector. Measure how long the reading took. Use chrono (§20.4). Sort the vector, and measure how long the sorting took. You will need a large file to get any useful data. Generate the file of at least a million integers using random (§20.4).

[5] Read the file of integers from the previous exercise into a set. Measure how long the reading and creation of the set took.

[6] Do the previous exercise with an unsorted_set. Did the timings from this and the previous two exercises match your expectations?

[7] Repeat the previous three exercises with a file of random character strings. Each string should contain only letters and digits. Each string should have between 4 and 24 characters.

[8] Read a sequence of (name,height) pairs into a map. Names should be represented as strings and heights by ints. Output the pairs lexicographically sorted by name. Print a name : height per line.

[9] Output the pairs from the previous exercise sorted by height in decreasing order (tallest first).

[10] Read the lines from a file of text and output the unique lines (once only). Hint: read the lines into a map and output a line only if it hasn't been seen before. This is the AWK program (!a[$0]++).

[11] Build a map of (English word, Spanish word) pairs of words with equivalent meaning. Each word is a string. Read the pairs from a file, and display them on the screen. A couple of dozen pairs will do. Give a user the choice of displaying the words in English lexicographical order or Spanish lexicographical order.

[12] Repeat the previous exercise, but handle cases where a word has more than one meaning. For example, (sheet,hoja) and (leaf,hoja). Hint: multimap.

Postscript

Choosing an appropriate data structure to hold an application's data is key to simplifying the code and to maximizing performance. Thick books have been written about this. Here, we focus on a few container types that are often good choices: map, unordered_map, set, array, string, and of course vector. These are, in fact, relatively simple and therefore compact and fast. The extensive parameterization makes them extremely flexible. Their design and implementation techniques provide a useful pattern for the development of more specific containers. CC

21

Algorithms

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 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.

21.1 Standard-library algorithms

The standard library offers about 125 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	
p=find(b,e,v)	p points to the first occurrence of v in [b:e).
p=find_if(b,e,p)	p points to the first element x in [b:e) so that p(x) is true.
x=count(b,e,v)	x is the number of occurrences of v in [b:e).
x=count_if(b,e,p)	x is the number of elements in [b:e) so that p(x) is true.
sort(b,e)	Sort [b:e) using <.
sort(b,e,p)	Sort [b:e) using p.
x=is_sorted(b,e)	If [b:e) is sorted x is true.
b2=copy(b,e,b2)	Copy [b:e) to [b2:b2+(e−b)); there had better be enough elements after b2 to hold [b:e).
b2=move(b,e,b2)	move [b:e) to [b2:b2+(e−b)); there had better be enough elements after b2 to hold [b:e).
b2=uninitialized_copy(b,e,b2)	Copy [b:e) to an uninitialized [b2:b2+(e−b)); there had better be enough elements after b2 to hold [b:e).
b2=unique_copy(b,e,b2)	Copy [b:e) to [b2:b2+(e−b)); don't copy adjacent duplicates.
e2=merge(b,e,b2,e2,r)	Merge two sorted sequences [b:e) and [b2:e2) into [r:r+(e−b)+(e2−b2)).
[p,q]=equal_range(b,e,v)	[p:q) is the subsequence of the sorted range [b:e) with the value v, basically, a binary search for v.
equal(b,e,b2)	Do all elements of [b:e) and [b2:b2+(e−b)) compare equal? The sequences must be sorted.
x=accumulate(b,e,i)	x is the sum of i and the elements of [b:e).
r=max(x,y)	r is a reference to the larger of x and y.
p=max_element(b,e)	p points to the largest element in [b:e).
iota(b,e,i)	[b:e) becomes the sequence i, i+i, i+2, ...

CC By default, comparison for equality is done using == and ordering is done based on < (less than). 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.

A sequence can also be represented as a *range* (§20.7), so instead of find(b,e,v) we can write ranges::find(r,v). This is the case for most algorithms.

Also, the standard library provides parallel and vectorized versions of key algorithms to allow people to take better advantage of hardware facilities. Look them up if you need more performance than is provided by the default implementations of the algorithms.

The STL algorithms, containers, and iterators offer a model that we can use to write our own versions for needs beyond what the standard directly addresses.

21.1.1 The simplest algorithm: find()

Arguably, the simplest useful algorithm is find(). It finds an element with a given value in a sequence:

```
template<input_iterator In, equality_comparable<In::value_type> T>     // §18.1.3
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 this definition of find(). It is slightly simplified compared to what you can find looking at a std::find() implementation, e.g., it doesn't handle plain pointers such as int* because those don't have a value_type member. The technique for that is beyond this book (if you are seriously curious, look up *iterator traits*). Naturally, you can use find() without knowing exactly how it is implemented – in fact, we have used it already (e.g., §19.5.2). 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: CC

```
void f(vector<int>& v, int x)
{
        auto p = find(v.begin(),v.end(),x);
        if (p!=v.end()) {
                // ... we found x in v; we can use *p ...
        }
        else {
                // ... no x in v; don't dereference p ...
        }
        // ...
}
```

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 by find(), we used auto (§2.10). 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<input_iterator In, equality_comparable<In::value_type> T>
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<input_iterator In, equality_comparable<In::value_type> T>
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.

21.1.2 Some generic uses

CC 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 (that can be read from)
- Any element type (that can be compared to the iterator's value type)

Here are some examples (consult the diagrams in §19.3 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. CC

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. CC

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 §19.5:

```
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.

The most common use of an algorithm like find() is to look at a complete container. In such case, the compiler can find the relevant begin() and end() for us and we can write

```
template<class T>
void use(vector<T>& v, T& x)
{
        auto p = ranges::find(v,x);
        if (p!=v.end()) {
                // ... we found x ...
        }
        // ...
}
```

Having to write that **ranges::** is most unfortunate. The reason has to do with problems related to having both old-style unconstrained versions of algorithms (for compatibility reasons) and properly constrained (type checked) versions.

21.1.3 Generalizing 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. Then, we can get a much more useful **find** operation when we can define our search criteria ourselves. Maybe we want to find a value larger than 42. Maybe we want to compare strings without taking case (uppercase vs. lowercase) 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<input_iterator In, predicate<In::value_type> Pred>
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.

CC 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 **int**s 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! And, of course, there is. XX

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.

21.2 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

- can be called as a predicate, e.g., pred(∗first)
- can hold a value, such as 31 or x, for use when called

CC 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.

CC 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[ ].

21.2.1 An abstract view of function objects

CC 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.

CC 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 (§21.2), for defining sorting criteria (§21.2.2), for specifying arithmetic operations in numerical algorithms (§21.3), for defining what it means for values to be equal (§21.1.3), and for much more. The use of function objects is a major source of flexibility and generality.

AA 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 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.

21.2.2 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 many 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

```
// ...
ranges::sort(vr, Cmp_by_name{});        // sort by name
// ...
ranges::sort(vr, Cmp_by_addr{});        // sort by addr
// ...
```

CC **Cmp_by_name** is a function object that compares two **Record**s by comparing their **name** members. **Cmp_by_addr** is a function object that compares two **Record**s 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 **Record**s. 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; }        // Huh?
};
```

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 C (and C++) standard-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 if you ever need such an obscure comparison.

21.2.3 Lambda expressions

Defining a function object (or a function) in one place in a program and then using it in another can be 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 (§13.3.3). 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

```
// ...
ranges::sort(vr, [](const Record& a, const Record& b) { return a.name < b.name; });
// ...
ranges::sort(vr, [](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 §21.1.3. 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 = ranges::find_if(v, [](double a) { return a>31; });
        if (p!=v.end()) {
                // ... we found a value > 31 ...
        }

        auto q = ranges::find_if(v, [&](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. In particular, it can be a great help that you don’t have to look in two places to understand what’s going on. This can be significant when the unique operation doesn’t have an obvious simple name.

The `[&]` and `[]` are called *lambda captures*:

`[]`*:* If there is nothing between `[` and `]`, the lambda is just like an ordinary function: it can access its arguments, its own local variables, and names in the global (namespace) scope (§7.3).

`[&]`*:* If we use `[&]`, the lambda can also use names from the scope in which it is defined, its enclosing scope. Now it acts like a local function.

`[=]`*:* You can even ask to access copies of variables in the enclosing scope: `[=]`.

There are more subtle ways of capturing, but those are beyond the scope of this book.

21.3 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 a few STL-style standard-library numerical algorithms:

Numerical algorithms	
x=accumulate(b,e,i)	Add a sequence of values; e.g., for {a,b,c,d} produce i+a+b+c+d. The type of the result x is the type of the initial value i.
x=inner_product(b,e,b2,i)	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 x is the type of the initial value i.
r=partial_sum(b,e,r)	Produce the sequence of sums of the first n elements of [b:e); e.g., for {a,b,c,d} produce {a, a+b, a+b+c, a+b+c+d}.
r=adjacent_difference(b,e,b2,r)	Produce the sequence of differences between elements of [b:e); e.g., for {a,b,c,d} produce {a,b-a,c-b,d-c}.
iota(b,e,v)	Fill the range [b:e) with the values v, v+1, v+2, ...; The values are computers using prefix ++.
x=midpoint(a,b)	Compute the midpoint between a and b; roughly (a+b)/2 without overflow
x=gcd(a,b)	x is the greatest common divisor of a and b
x=lcm(a,b)	x is the least common multiple of a and b

We'll describe the first two here and leave it for you to explore the rest if you feel the need. There are also inclusive_scan(), exclusive_scan(), transform_inclusive_scan(), and transform_exclusive_scan() that are beyond the scope of this book.

21.3.1 Accumulate

The simplest and most useful numerical algorithm is accumulate(). In its simplest form, it adds a sequence of values:

```
template<input_iterator In, 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>& v, int* p, int n)
{
      double sum = accumulate(v.begin(), v.end(), 0.0);
      int sum2 = accumulate(p,p+n,0);
}
```

For mysterious reasons, the ranges versions of the numerical algorithms didn't make it into C++20, but they are not hard to define. For example:

```
template<input_range R, output_iterator Out, typename T>
T accumulate(R r, Out oo, T init)
{
      return accumulate(begin(r),end(r),oo,init);
}
```

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(vector<int>& v)
{
      int s1 = accumulate(v, 0);                  // sum into an int
      long sl = accumulate(v, long{0});           // sum the ints into a long
      double s2 = accumulate(v, 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.

Using the variable in which you want the result as the initializer is a popular idiom for specifying the type of the accumulator: AA

```
void f(vector<double>& v)
{
      double s1 = 0;
      s1 = accumulate(v, s1);
      int s2 = accumulate(v, s2);         // oops
      float s3 = 0;
      accumulate(v, s3);                  // oops
}
```

Do remember to initialize the accumulator and to assign the result of accumulate() to the variable. XX 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 §7.4.3), but the result is never assigned anywhere; that compilation is just a waste of time.

21.3.2 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<input_iterator In, typename T, invocable<T,In::value_type> BinOp>
[[nodiscard]]    // warn if the return value isn't used by a caller
T accumulate(In first, In last, T init, BinOp op)
{
        while (first!=last) {
                init = op(init, *first);
                ++first;
        }
        return init;
}
```

The invocable concept is from the standard library (§18.1.3). Any binary operation that accepts two arguments of the accumulator's type and the iterator's value_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).

Note that for products of floating-point numbers, the obvious initial value is 1.0.

CC As in the sort() example (§21.2.2), 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 (including lambdas):

- 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.

21.3.3 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). If you prefer code to words, here is the STL version:

```
template<input_iterator In, input_iterator In2, typename T>
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 the stock market index example from §20.2.3. The way that works is to take a set of companies and assign each a "weight." We could replace the loop calculating the DJII from the vector version of dow_price and dow_weightwith a single call of inner_product:

```
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. XX

TRY THIS

Define an inner_product() that takes two input ranges. Then try the examples above with your version.

21.3.4 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<input_iterator In, input_iterator In2, typename T, typename BinOp, typename BinOp2>
        requires invocable<BinOp,T,In::value_type>
                && invocable<BinOp2,T,In2::value_type>
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;
}
```

We can even do some computation directly using map version of dow_price and dow_weight. In particular, we can calculate the index, using the standard-library algorithm inner_product() (§21.3.3). 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;   // using the pairs' second member
}
```

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>(),                          // add (as usual)
                                weighted_value                           // extract values and weights, then multiply
                                        );
```

A ranges version of inner_product() would make that code even nicer.

21.4 Copying

In §21.1.1, 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. But 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	
b2=copy(b,e,b2)	Copy [b:e) to [b2:b2+(e–b)).
b2=unique_copy(b,e,b2)	Copy [b:e) to [b2:b2+(e–b)); suppress adjacent copies.
b2=copy_if(b,e,b2,p)	Copy [b:e) to [b2:b2+(e–b)), but only elements that meet the predicate p.

21.4.1 The simplest copy: copy()

The basic copy algorithm is defined like this:

```
template<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.

21.4.2 Generalizing copy: 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<input_iterator In, output_operator Out, predicate<In::value_type> Pred>
Out copy_if(In first, In last, Out res, Pred p)
        // copy elements that fulfill the predicate p into res
{
        while (first!=last) {
                if (p(*first)) {
                        *res = *first;
                        ++res;
                }
                ++first;
        }
        return res;
}
```

For example, using the ranges version of copy_if(), we can copy 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 into v2
{
        vector<int> v2(v.size());
        ranges::copy_if(v, v2.begin(), [](int x){ return x>6;});
        // ...
}
```

21.5 Sorting and searching

CC 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<random_access_iterator Ran>
void sort(Ran first, Ran last);

template<random_access_iterator Ran, less_than_comparable<Ran::value_type> Cmp>
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 {
        bool operator()(const string& x, const string& y) const          // is lowercase(x) < lowercase(y)?
        {
                for (int i = 0; i<x.length(); ++i) {
                        if (i == y.length())                // y<x
                                return false;
                        char xx = tolower(x[i]);
                        char yy = tolower(y[i]);
                        if (xx<yy)                          // x<y
                                return true;
                        if (yy<xx)                          // y<x
                                return false;
                }
                if (x.length()==y.length())                 // x==y
                        return false;
                return true;                                // x<y (fewer characters in x)
        }
};

void sort_and_print(vector<string>& vc)
{
        ranges::sort(vc,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: CC

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(log2(N));* see §20.3. AA

The binary_search algorithm comes in two variants:

```
template<random_access_range Ran, typename T>
bool binary_search(Ran r, const T& val);

template<random_access_range Ran, typename T, predicate<Ran::value_type,Ran::value_type> Cmp>
bool binary_search(Ran r, const T& val, Cmp cmp);
```

XX Obviously, there are also versions that take a pair of iterators. 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
        }
        // ...
}
```

AA 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() (PPP2.§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. For example:

```
template<class T>
void print same(const vector<T>& v, const T& x)
{
        for (const auto& x : ranges::equal_range(v,x))        // equal_range() returns a sub-range
                cout << x << '\n';
}
```

Drill

[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] Why would you use a function or function object rather than a lambda as an argument?
[12] What does accumulate() do?
[13] What does inner_product() do?
[14] Why use a copy() algorithm when we could ‘‘just write a simple loop’’?
[15] What is a binary search?

Terms

accumulate()	find_if()	searching	algorithm
function object	sequence	application: ()	generic
set	sort()	iota()	find()
inner_product()	sorting	binary_search()	upper_bound()
lambda	copy()	lower_bound()	predicate
unique_copy()	copy_if()	equal_range()	invocable

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] 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?
[7] 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().

[8] 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.

[9] Provide a GUI interface for entering Orders into files.

[10] 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.

[11] 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 lowercase, 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).

[12] 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."

[13] Provide a GUI for the program from the previous exercise.

Postscript

AA 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.