2

Objects Without Inheritance

2.1 Pass a Structure to a Function

A calendar computation

An "object" lets us package together a group of variables and the functions that operate on them. We will present objects with a program that performs the calendar computation in the box on p. 108. The three versions of the program will perform the same computation and produce the same output. Version 1 will have individual variables; Version 2, a structure; and Version 3, our first object.

Employing an object in such a simple program is like using a sledgehammer to kill ants. Even Version 1 has more machinery than is needed. A computation this simple would normally be done entirely in the `main` function; it is split into four separate functions only to foreshadow the object that will appear in Version 3.

At this point we do not wish to burden the reader with a program complicated enough to show what an object is good for. For now, we will just show what an object is. To avoid issues that have nothing to do with objects, we will make three simplifying assumptions about our calendar system.

(1) There are no leap years, and never were.

(2) There was a Year Zero between 1 B.C. and 1 A.D. We will refer to B.C. years as negative years: 1 B.C. will be the year −1.

(3) The calendar has always been Gregorian, never Julian. The switch-over in September, 1752 never happened.

September 1752						
S	M	Tu	W	Th	F	S
		1	2	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

2.1.1 Version 1: Pass Individual Variables to a Function

To do the computation, the program must know the lengths of the twelve months. We embody this knowledge as the array of twelve integers in lines 5–19. (Thirteen, actually. The subscripts start at zero, so we added a dummy element to let January be subscript 1.) Since an initial value was provided for each element, there was no need to write the number of elements in the square brackets in line 5.

The main character of the program is the trio of integer variables `year`, `month`, `day` in lines 27–29, and the three functions in lines 21–23 that operate on them. When we get to Version 3, all six will be packaged as one unit. The language will help us think of them as a single entity.

The simplest function is `date_print`, called in line 32 and defined in line 67. There was no need to pass the trio to `date_print` by reference. Pass-by-value would have been simpler than the pointer arguments in line 67.

```
void date_print(int year, int month, int day)   //arguments passed by value
{
    cout << month << "/" << day << "/" << year;
}
```

But it will be easier for Version 3 to introduce objects if all the functions take arguments of the same type, and the arguments of the other functions will have to be pointers.

To ensure that `date_print` does not change the values of the trio, line 67 declares the pointer arguments to be read-only. We have to write the keyword `const` three times; six times, if we include the declaration in line 23.

Line 69 outputs the month before the day, separated by slashes, to follow the American convention; we will internationalize on pp. 1031−1057.

The function `date_next` is called in line 38 and defined in 45. This time, the arguments must be passed by reference to let the function change the values of our trio of variables. It makes repeated calls to another function in line 51 to do the real work, passing along the first three arguments unchanged. The two functions have the same name; we can get away with this because their numbers of arguments are different.

The loop in line 50 will work correctly only if `count` is initially non-negative; this will be fixed on pp. 128−129. The increment in line 62 will work correctly only if the year does not already have the maximum value for an integer. We will check for this when we have "exceptions", on pp. 599−600.

We could have let the trio be global variables, making them accessible to the functions without the need for pointer arguments. But globals are cursed with immortality. They cannot be created, destroyed, and re-created while the program is running. See p. 464.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/version/version1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 const int date_length[] = {
 6      0,   //dummy element so that January will have subscript 1
 7     31,   //January
 8     28,   //February, ignoring for now the possibility of leap year
 9     31,   //March
10     30,   //April
11     31,   //May
12     30,   //June
13     31,   //July
14     31,   //August
15     30,   //September
16     31,   //October
17     30,   //November
18     31    //December
19 };
20
21 void date_next(int *pyear, int *pmonth, int *pday, int count);
22 void date_next(int *pyear, int *pmonth, int *pday);
23 void date_print(const int *pyear, const int *pmonth, const int *pday);
24
25 int main()
```

```
26 {
27     int year = 2014;
28     int month = 1;   //1 to 12 inclusive
29     int day = 1;     //1 to date_length[month] inclusive
30
31     cout << "How many days forward from ";
32     date_print(&year, &month, &day);
33     cout << " do you want to go? ";
34
35     int count;       //uninitialized variable
36     cin >> count;
37
38     date_next(&year, &month, &day, count);
39     cout << "The new date is ";
40     date_print(&year, &month, &day);
41     cout << ".\n";
42     return EXIT_SUCCESS;
43 }
44
45 void date_next(int *pyear, int *pmonth, int *pday, int count)
46 {
47     //Call the three-argument date_next (line 55) count times.
48     //Pass along the three pointers we received.
49
50     while (--count >= 0) {
51         date_next(pyear, pmonth, pday);
52     }
53 }
54
55 void date_next(int *pyear, int *pmonth, int *pday)
56 {
57     //Move to the next date.
58     if (++*pday > date_length[*pmonth]) {
59         *pday = 1;
60         if (++*pmonth > 12) {
61             *pmonth = 1;
62             ++*pyear;
63         }
64     }
65 }
66
67 void date_print(const int *pyear, const int *pmonth, const int *pday)
68 {
69     cout << *pmonth << "/" << *pday << "/" << *pyear;
70 }
```

Let's look at a representative expression in the body of one of the functions, the `++*pday` in the above line 58. The value of the subexpression `*pday` is the variable `day` in line 29, so the ++ adds 1 to `day`.

It is good that the ++ does not add 1 to `pday`, which is a pointer to `day`. If we added 1 to `pday`, it would point somewhere else, probably to garbage. Fortunately, there is no way that the ++ could possibly access `pday`. Because of their equal precedence and right-to-left associativity, the subexpression `*pday` is evaluated before the ++ is executed. We express this graphically by surrounding the subexpression `*pday` with a box. An operator outside a box cannot reach into the box and single out a sub-subexpression such as the `pday` in `*pday`. From the `++`'s point of view, the subexpression `*pday` is a monolithic whole.

$$\boxed{\texttt{++}\ \boxed{\texttt{*}\ \boxed{\texttt{pday}}}}$$

Line 58 therefore does the work of the following two lines. Think of them as an exploded view of line 58.

```
71      *pday = *pday + 1;
72      if (*pday > date_length[*pmonth]) {
```

To sum up, our major problem is how to make the trio of variables available to the functions that work on them. The main possibilities are pass-by-value, pass-by-reference (in read/write and read-only flavors), and global variables. We have settled on pass-by-reference, leaving our function bodies bristling with asterisks.

```
How many days forward from 1/1/2014 do you want to go? 280
The new date is 10/8/2014.
```

2.1.2 Version 2: Pass A Structure to a Function

Each `date` consists of a trio of integers: `year`, `month`, `day`. A thousand `date`'s would be three thousand separate integers. We could keep track of them more easily by clumping each trio into a structure. The main character of the program is now the single variable `d` in line 33 and the three functions in lines 27−29 that operate on it.

Lines 21−25 are the definition for a new data type named `date`. Despite being called a ''definition'', they do not create any variable of this type. They are merely the blueprint, describing what one of these variables would contain if we went ahead and created one. This is done in line 33.

Of course, our little program has only one date. Writing it as a structure is overkill, as was the division of the program into separate functions. The structure and functions are introduced only to prepare the way for the object in Version 3.

In C a structure data type was a second-class citizen, and line 33 would have needed the keyword `struct`. This is unnecessary in C++.

```
1      struct date d = {2014, 1, 1};   /* line 33 written in C */
```

The simplest function is `print`, called in line 36 and defined in line 71. To use the pointer argument `p` in line 73, we apply the dereferencing operator `*`, retrieving the pointed-to variable. Since this variable turns out to be a structure, we apply the dot operator and the name of a field. To execute the `*` before the dot, line 73 needs the parentheses in the expression `(*p).month`; without them, the dot would have gone first because of its higher precedence. We saw these parentheses on p. 48.

Line 73 will work, but is commented out because 74 is a simpler way to do the same thing. The arrow operator −> does the work of the star and dot. And now that there is only one operator, the parentheses are no longer needed.

To ensure that `print` does not change the values of the fields of the structure, line 71 declares the pointer argument to be read-only. This time, we have to write the keyword `const` only once.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/version/version2.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 const int date_length[] = {
```

```
 6       0,   //dummy element so that January will have subscript 1
 7      31,   //January
 8      28,   //February
 9      31,   //March
10      30,   //April
11      31,   //May
12      30,   //June
13      31,   //July
14      31,   //August
15      30,   //September
16      31,   //October
17      30,   //November
18      31    //December
19 };
20
21 struct date {
22     int year;
23     int month;   //1 to 12 inclusive
24     int day;     //1 to date_length[month] inclusive
25 };             //Version 3 will need this semicolon, too.
26
27 void next(date *p, int count);
28 void next(date *p);
29 void print(const date *p);
30
31 int main()
32 {
33     date d = {2014, 1, 1};   //curly braces around initial values
34
35     cout << "How many days forward from ";
36     print(&d);
37     cout << " do you want to go? ";
38
39     int count;               //uninitialized variable
40     cin >> count;
41
42     next(&d, count);
43     cout << "The new date is ";
44     print(&d);
45     cout << ".\n";
46     return EXIT_SUCCESS;
47 }
48
49 void next(date *p, int count)
50 {
51     //Call the one-argument next (line 59) count times.
52     //Pass along the pointer we received.
53
54     while (--count >= 0) {
55         next(p);
56     }
57 }
58
59 void next(date *p)
```

```
60 {
61     //Move to the next date.
62     if (++p->day > date_length[p->month]) {
63         p->day = 1;
64         if (++p->month > 12) {
65             p->month = 1;
66             ++p->year;
67         }
68     }
69 }
70
71 void print(const date *p)
72 {
73     //cout << (*p).month << "/" << (*p).day << "/" << (*p).year;
74     cout << p->month << "/" << p->day << "/" << p->year;
75 }
```

Let's look at the representative expression in this version, the `++p->day` in the above line 62. The value of the subexpression `p->day` is the field `d.day` in line 33, so the `++` adds 1 to `d.day`.

It is good that the `++` does not add 1 to `p`, which is a pointer to `d`. If we added 1 to `p`, it would point somewhere else, probably to garbage. Fortunately, there is no way that the `++` could possibly access `p`. Because the precedence of `->` is higher than that of prefix `++`, the subexpression `p->day` is evaluated before the `++` is executed. We express this graphically by surrounding the subexpression `p->day` with a box. An operator outside a box cannot reach into the box and single out a sub-subexpression such as the `p` in `p->day`. From the `++`'s point of view, the subexpression `p->day` is a monolithic whole.

$$\boxed{\texttt{++}\ \boxed{\boxed{\texttt{p}}\ \texttt{->}\ \boxed{\texttt{day}}}}$$

Line 62 therefore does the work of the following two lines. Think of them as an exploded view of line 62.

```
76     p->day = p->day + 1;
77     if (p->day > date_length[p->month]) {
```

Version 2 collected the trio of disparate variables into one structure. Version 2 also runs faster because it passes fewer arguments to the functions. But the bodies of the functions have become more complicated: the arrows in Version 2 are more baroque than the stars in Version 1. What we want is the speed of Version 2 with a notation as simple as Version 1. This is what Version 3 will deliver, and more.

As a footnote, the function names in Version 2 have been simplified. The Version 1 function arguments were of a plain vanilla data type: pointer to `int`. We therefore had to add `date_` to the name of each function to allow for the possibility of other functions taking arguments of the same type.

```
78 void date_print(const int *pday,  const int *pmonth,  const int *pyear);
79 void time_print(const int *phour, const int *pminute, const int *psecond);
```

But the Version 2 function arguments are of a distinctive data type: pointers to a very specific type of structure. We are now in a position to overload the function name.

```
80 void print(const date *p);
81 void print(const time *p);
```

In Version 1, we could easily have passed a date to the function that prints a time:

```
82     time_print(&day, &month, &year);//bug not caught by compiler
```

In Version 2, the same error would require an explicit cast.

```
83      print(reinterpret_cast<const time *>(&d));
```

In addition, we can no longer accidentally pass a date to the function that prints a time, or a time to the function that prints a date. If we try to do this, the program will not compile.

The name of the array will also be reduced, from `date_length` to `length` when we do "static data members" on pp. 238–239. In fact, any compound name that we invent is for temporary use only. Eventually, we will use the language itself, rather than a compound name with an underscore, to indicate what goes with what.

```
How many days forward from 1/1/2014 do you want to go? 280
The new date is 10/8/2014.
```

2.1.3 Version 3: Call Member Functions of an Object

An object is a structure passed to a function

I dislike inventing new terminology . . .

—Bjarne Stroustrup, *The Design and Evolution of C++*, p. 31

Our program has only three major variables, `year`, `month`, and `day`. It is simple enough to do all its work in `main`. But with more variables and greater complexity, we would want to clump the variables into structures and divide the program into functions. Our problem would then be to pass these structures down to the functions where the work is done.

Objects will eventually be a new "paradigm" for programming (pp. 163–179). But we will begin with a much more mundane definition. An *object* is a structure that can be passed to a function as quickly as the structures in Version 2, with a notation even simpler than that of Version 1.

The *class* of an object is its data type. The object `d` is of the class `date` defined in line 37 of `version3.C` on p. 115. As in the previous example, the class definition does not create any variables. It is merely a blueprint.

We say "class of an object" rather than "data type of a structure" because the terminology of C++ is borrowed from the language Simula67 rather than C.

Note that the word "structure" can mean two things in C: a certain kind of data type, or a variable of that type. In C++, these ideas have separate words: "class" and "object".

Members of an object

In C, a structure has *fields,* all of which must be variables. In C++, a class has *members,* which can be variables or functions. From this difference will proceed all of object-oriented programming.

The first three members of class `date`, declared in lines 22–24, are variables, like the fields of a C structure. They are called *data members.* The last four members, declared in lines 26–29, are functions, and have no direct counterpart in C. They are called *member functions.* Our convention will be to declare the data members before the member functions; page 119 will explain why the reverse order would sometimes upset the human reader. The `public:` in line 25 will be explained on p. 114.

A data member is located physically inside of the object to which it belongs, as a field is inside a structure in C. But a member function has an entirely different relationship to the object to which it belongs. When we say that we are calling a member function that "belongs" to a particular object, we mean only that we are passing the address of that object to the function, using the special notation described below. A member function is not located inside of an object, as a data member is. A member function is shared by all the objects of its class. We could just as easily pass it the address of some other object of the class.

Line 41 shows the special notation for calling a member function of an object, i.e., for passing the address of the object to the member function. The "dot" operator has the same operands it had in C: a structure or object on the left, and a field or member on the right. When the right operand is a data member, the dot has the same meaning it had in C. (Line 40 is an example, although it will not compile.) But when the right operand is a member function, it calls the member function and passes it the address of the object which is the left operand of the dot. Line 41 calls the `print` member function of the object `d`, i.e., it passes the address of the object `d` to the member function `print`.

Let's walk through the order in which the subexpressions of line 41 are evaluated. To use the object `d` we apply the dot operator and the name of a member, which delves into the object and accesses the member. Since the member turns out to be a member function, we then apply the function call operator (the parentheses that surround the argument list). To execute the dot before the function call operator, no parentheses are needed. They have equal precedence and left-to-right associativity, so the dot goes first. See p. 48 for a similar sequence of subexpressions.

$$\boxed{\boxed{\boxed{\texttt{d}}\ \texttt{.}\ \boxed{\texttt{print}}}\ \texttt{()}}$$

The function `print` in line 41 receives no arguments other than the address of `d`. The function `next` in line 47 receives the argument `count` as well as the address of `d`.

The definition of the member functions

The `print` member function in line 99 has the simplest definition. It must name the class to which the function belongs, since there could be a function with the same name belonging to another class.

```
void date::print() const   //first line of definition of our function
void time::print() const   //first line of definition of another function
```

The name of the class and the member function are pasted together with the scope operator `::`. Its operands are always a last name and a first name. In the expression `std::cout` on p. 20, the operands were a namespace and one of its members. In the `date::print` in line 99, they are a class and one of its members.

A member function always receives the address of its object as an *implicit* (invisible) argument. The argument is not declared in the parentheses in line 99, and is usually never mentioned at all. If its value must be used explicitly, however, it is available as a pointer named `this`. When `print` is called from line 41, for example, the value of `this` in lines 101–106 will be the address of the object `d`. `this` can be mentioned only in the body of a member function.

There is nothing wrong with lines 101 and 102. I commented them out only to make all three versions of the program produce the same output. When `print` is called from line 41, the pointer `this` in line 102 is the address of the object `d`, and `*this` is the value of `d`. Applying the dereferencing operator `*` to any pointer in C or C++ will get us the value of the pointed-to variable.

Lines 104 and 105 are the same as lines 73 and 74 of the above `version2.C`, with the pointer `p` now named `this`. They are commented out because line 106 is a simpler way to do the same thing. In the body of a member function of a class, a member of that class with no dot or arrow in front of it is always a member of the object to which the member function belongs, i.e., the object to which the member function has received the implicit pointer. After all, the simplest notation is—no notation at all. When `print` is called from line 41, the `month` in line 106 will be the `month` data member of the object `d` in line 41.

A member function may also receive *explicit* (visible) arguments. In the `next` function called in line 47 and defined in 74, we have one implicit and one explicit argument.

Lines 81–83 correspond to lines 104–106, but with a member function instead of a data member. They demonstrate that a member function of an object can easily call another member function of the same object. Lines 104 and 105 are commented out because 106 is a simpler way to do the same thing; 81 and 82 are commented out because 83 is a simpler way to do the same thing. In the body of a member function of a class, a member of that class with no dot or arrow in front of it is always a member of the object to which the member function belongs, i.e., the object to which the member function has received the implicit pointer. When the one-explicit-argument `next` is called from line 47, the no-explicit-argument `next` in line 83 will be the no-explicit-argument `next` member function of the object `d` in line 47.

The member function names `next`, `next`, and `print` are only provisional. When we do operator overloading, we will give them their proper C++ names: `operator+=` (pp. 282–285), `operator++`, (pp. 288–289), and `operator<<` (pp. 337–340). Some adjustment of their arguments and return values will also be necessary.

const member functions

A `const` member function is one that cannot change the value of any data member of its object. We declared `print` to be `const` in lines 29 and 99. A non-`const` member function is one that can change the value of the data members of its object; that in fact is the *raison d'être* of the `next` functions.

We have already seen that one member function can call another member function of the same object (line 83). A non-`const` member function can call any member function of its object. But a `const` member function can call only the `const` member functions of its object.

If a member function is `const`, the pointer passed to it is read-only. In `date::print`, the pointer is implicitly declared as

```
const date *const this;
```

But in the `date::next` functions, it is implicitly declared as

```
date *const this;
```

Note that in both cases the pointer is `*const`.

Member functions and free functions

For the time being, a member function is one that receives an implicit pointer argument. A *free* function is one that receives no implicit pointer argument. Every C function is free; and in both languages the `main` function is always free.

It sounds like every function is either a member or free, but not both. We will see later that there is one exceptional kind of member function that receives no implicit pointer. It is a free member and is called a "static" member function. See pp. 242–247.

A constructor

The most important member functions of a class are its *constructors*. One of these functions is always called when we create a new object of the same class. In fact, we usualy speak of *constructing* an object rather than creating it. A constructor has the same name as the class to which it belongs. Ours is declared in line 26 and defined in 55.

The job of a constructor is to make a newborn object ready to assume its responsibilities. The responsibilities of our object are to hold a valid date, move it forward, and print it. A constructor must leave the object in a state in which its other member functions will operate correctly.

For the moment, we will assume that a constructor must put valid values into all the data members of the newborn object. Later, we will see constructors that do less (p. 149) and more (p. 195). For the moment, we also assume that the number of arguments of the constructor will be the same as the number of data members of the newborn object. Later, we will see that this is not always so. (p. 125.)

Our constructor is called in line 37. As usual, the arguments are in parentheses; recall that the corresponding line 33 of `version2.C` had `{`curly braces`}`. The object `d` is the main character of our program.

When a `date` is constructed, the data members are created in the order in which they are declared in lines 22–24. For the present, however, this is only of academic interest. No one cares what order integers are created in, because nothing observable happens when an integer is born.

But the order may become important in the future. It is possible for the data members of an object to be little objects, just as the fields of a structure can be little structures (pp. 257–265). If we let `year`, `month`, and `day` be objects in their own right, each will have its own constructor. When that happens, the error checking for the `month` data member now in lines 57–61, and the error checking for the `day` data member now in lines 63−67, will be moved to the constructors for these month and day objects. Observe that lines 57−61 must be executed before 63−67, because the `month` value must be validated before it is used as an array subscript in line 63. When they become objects, we will therefore have to construct the `month` before the `day`. Lines 22−24 do this now even though it is not currently necessary. It will be one less thing to change when our three data members change from integers to objects.

A constructor returns the object that it constructs; we will take advantage of this on pp. 137−138. But the object is always returned implicitly. Do not declare a return type for the constructor, not even `void`. And do not write a `return` statement with an expression inside a constructor.

```
    return;                              //okay
    return something;                    //won't compile
```

Public and private members

The members of a class fall into two groups, *public* and *private.* The private ones are those declared at the start of the class declaration (`year`, `month`, `day`); the public ones are those declared after the label `public:` in line 25 (`date`, `next`, `next`, `print`). We could have inserted the label `private:` at line 21½, but this would have been redundant.

The public members of a class can be mentioned by any function. The `main` function mentions two of them: `print` in lines 41 and 49, and the one-explicit-argument `next` in line 47. But the private members can be mentioned only in the bodies of the member functions of that class (and in the class declaration itself, lines 21−30). The members `month`, `day`, `year` can be mentioned in line 106 by the `print` member function, but not in line 40 by `main`. Uncomment line 40 and see what the error message is.

It takes more effort to plan a C++ class than a C structure. Any function can access any field of any C structure. But we have to decide in advance which functions will be able to access the private members of a C++ class; they will have to be member functions of the class. We will also have to decide which functions will have read/write access to the members; they will have to be non-`const` member functions.

For the time being, let the data members of a class be private to make it easier to debug and modify. A data member should certainly be private if not every value is legal for it, or if the legal range of values depends on the value of another data member. For example, a value of 31 is legal for `day` only for certain values of `month`.

There are two other places where a private member can be mentioned. We bring them up now only for completeness; don't worry about them yet. A private member can be mentioned in the initial value for a "static" data member, pp. 236−242. A private member that belongs to no object can be mentioned between its declaration and the curly brace that ends the class declaration.

```
class exotic {
    static size_t s;        //a "static" data member
    char a[sizeof s];       //Can mention s here even though it's private.
};                          //end of class declaration

size_t exotic::s = sizeof a; //Can mention a here even though it's private.
```

A variable that is not a data member (yet)

The `date_length` array in lines 5−19 is intended for use only by the member functions of class `date`. For safety, it should be inaccessible to every other function. We already know how to do this: by making the array a private data member of class `date`.

But doing this now would waste space if we had more than one `date` object. There is no need for each `date` to contain its own copy of the array. We will have to wait until we talk about a different kind of data member, the "static" ones on pp. 238−239. For the present, we merely observe that the array has yet to receive its final disposition. Somewhat unsatisfactorily, it remains a separate variable, floating near its associated class.

To indicate that the array has some kind of connection to class `date`, we temporarily gave it a name starting with `date_`. When it becomes a member of the class, the `date_` will be removed.

How vulnerable is the array in the meantime? Not very. A `const` global variable in C++ can be mentioned only in the `.C` file in which it is defined. (To change this, we could declare the array with the keyword `extern` at the start of line 5.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/version/version3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 const int date_length[] = {
 6      0,   //dummy element so that January will have subscript 1
 7     31,   //January
 8     28,   //February
 9     31,   //March
10     30,   //April
11     31,   //May
12     30,   //June
13     31,   //July
14     31,   //August
15     30,   //September
16     31,   //October
17     30,   //November
18     31    //December
19 };
20
21 class date {
22     int year;
23     int month;                //1 to 12 inclusive
24     int day;                  //1 to date_length[month] inclusive
25 public:
26     date(int initial_month, int initial_day, int initial_year);
27     void next(int count);     //Go count days forward.
28     void next();              //Go one day forward.
29     void print() const;       //Output date to cout (function declaration).
30 };                            //Don’t forget ; at end of class declaration.
31
32 int main()
33 {
34     //Construct d by passing three arguments to the constructor for d.
35     //The constructor will initialize d’s data members.
36
37     date d(1, 1, 2014);       //parentheses around function arguments
38
39     cout << "How many days forward from ";
40     //cout << d.month << "/" << d.day << "/" << d.year;   //won’t compile
41     d.print();   //Pass the address of d to the print member function.
42     cout << " do you want to go? ";
43
44     int count;                //uninitialized variable
45     cin >> count;
46
```

```
47     d.next(count);
48     cout << "The new date is ";
49     d.print();
50     cout << ".\n";
51
52     return EXIT_SUCCESS;
53 }
54
55 date::date(int initial_month, int initial_day, int initial_year)
56 {
57     if (initial_month < 1 || initial_month > 12) {
58         cerr << "bad month " << initial_month << "/" << initial_day
59             << "/" << initial_year << "\n";
60         exit(EXIT_FAILURE);
61     }
62
63     if (initial_day < 1 || initial_day > date_length[initial_month]) {
64         cerr << "bad day " << initial_month << "/" << initial_day
65             << "/" << initial_year << "\n";
66         exit(EXIT_FAILURE);
67     }
68
69     year = initial_year;
70     month = initial_month;
71     day = initial_day;
72 }
73
74 void date::next(int count)
75 {
76     //Call the no-explicit-argument next (line 87) count times.
77     //Pass along the implicit pointer we received.
78
79     while (--count >= 0) {
80         //call another member function of same object
81         //(*this).next();
82         //this->next();
83         next();
84     }
85 }
86
87 void date::next()
88 {
89     //Move to the next date.
90     if (++day > date_length[month]) {
91         day = 1;
92         if (++month > 12) {
93             month = 1;
94             ++year;
95         }
96     }
97 }
98
99 void date::print() const   //This is a function definition.
100 {
```

```
101     //cout << "The address of this object is " << this << ".\n";
102     //cout << "Size in bytes of this object is " << sizeof *this << ".\n";
103
104     //cout << (*this).month << "/" << (*this).day << "/" << (*this).year;
105     //cout << this->month << "/" << this->day << "/" << this->year;
106     cout << month << "/" << day << "/" << year;
107 }
```

The expression in above line 90 corresponds to the ones we examined in Versions 1 and 2. It is now simple enough to need no exploded view.

```
How many days forward from 1/1/2014 do you want to go? 280
The new date is 10/8/2014.
```

The above lines 101–102 would produce the following extra output on my platform.

```
The address of this object is 0xffbff03c.
Size in bytes of this object is 12.            3 int's of 4 bytes each
```

The above line 40 would cause the following trio of error messages on my platform.

```
version3.C: In function 'int main()':
version3.C:23:6: error: 'int date::month' is private
version3.C:40:12: error: within this context
version3.C:24:6: error: 'int date::day' is private
version3.C:40:30: error: within this context
version3.C:22:6: error: 'int date::year' is private
version3.C:40:46: error: within this context
```

▼ Homework 2.1.3a: deliberately introduce compilation errors

What is the error message on your platform when you try to violate each of the following rules?

(1) We can mention a private member of a class only in the body of the class declaration (lines 21–30 above) or in the body of a member function of the class. Uncomment line 40 of the above `version3.C` and see what happens.

(2) `this` is a `*const` pointer, so it always points to the same place. Try to make it point elsewhere.

```
    ++this;
```

(3) Since the `date::print` member function is `const`, there are three things it cannot do to the object to which it belongs. We would never want to do them in a "print" function, but let's try them anyway.

(3a) `date::print` cannot change the value of a data member of the object.

```
    ++day;
```

(3b) `date::print` cannot call a non-`const` member function of the object, because that could change the value of a data member.

```
    next();
```

(3c) `date::print` cannot return a pointer granting read/write access to the object. You will also have to change the `void` to `date *` in the above line 29.

```
date *date::print() const
{
    cout << month << "/" << day << "/" << year;
```

```
 7     return this;
 8 }
```

If we could get away with the above lines 4−8, we could use the returned pointer to change the value of a data member:

```
 9     const date d(1, 1, 2014); //should not be allowed to change d
10     date *p = d.print();      //p is a pointer to a date. Don't let this compile!
11     p->next();                //change the value of a data member of d
```

Verify that `date::print` could return a pointer that is read-only. You will have to change the `void` to `const date *` in the above line 29.

```
12 const date *date::print() const
13 {
14     cout << month << "/" << day << "/" << year;
15     return this;
16 }
```

Test the new `date::print` like this:

```
17     date d(1, 1, 2014);
18     const date *p = d.print();
19     cout << "\n";
20
21     cout << p << "\n"                //should output the address of d
22         << &d << "\n";               //should output the same address
```

▲

2.2 Notational Conveniences

2.2.1 Inline Member Functions

The member function `print` is small enough to be inline. We already know one way to do this.

(1) Add the keyword `inline` to the start of the function definition in line 99 of the above `version3.C`.

(2) Move the definition in lines 99−107 up to line 31, because the definition of a function must be seen before we can make an inline call to it.

If the function is a member function, there is a more compact notation for making it inline.

(1) Remove the function's definition in the above lines 99−107.

(2) Change the declaration in the above line 29 to the following line 9, which is both a declaration and definition. Write no semicolon after the closing curly brace, just as there was no semicolon after the closing curly brace in the above line 107. After the `year`, however, write a semicolon as in the above line 106.

```
1     class date {
2         int year;
3         int month;                    //1 to 12 inclusive
4         int day;                      //1 to date_length[month] inclusive
5     public:
6         date(int initial_month, int initial_day, int initial_year);
7         void next(int count);         //Go count days forward.
8         void next();                  //Go one day forward.
9         void print() const {cout << month << "/" << day << "/" << year;}
```

```
10     };
```

Surprisingly, an inline member function defined this way can mention another member before that member has been declared. We could actually have written the following line 16, mentioning the `month`, `day`, and `year` before their declarations in lines 18–20. For another example, see p. 214.

```
11 class date {
12 public:
13     date(int initial_month, int initial_day, int initial_year);
14     void next(int count);        //Go count days forward.
15     void next();                 //Go one day forward.
16     void print() const {cout << month << "/" << day << "/" << year;}
17 private:
18     int year;
19     int month;                   //1 to 12 inclusive
20     int day;                     //1 to date_length[month] inclusive
21 };
```

But moving the definition to the above line 16 would just get people upset. Keep it in line 9. With only one other exception, a C or C++ variable or function can never be mentioned before its declaration.*

Given the compact notation, why would we ever want to use the other one? Sometimes we have no choice. If `month_before_day` had to be declared in line 33 for some reason, and if `print` mentioned `month_before_day`, then the definition of `print` must come after line 33.

```
22 class date {
23     int year;
24     int month;                   //1 to 12 inclusive
25     int day;                     //1 to date_length[month] inclusive
26 public:
27     date(int initial_month, int initial_day, int initial_year);
28     void next(int count);        //go count days forward
29     void next();                 //go one day forward
30     void print() const;
31 };
32
33 bool month_before_day = true;
34
35 inline void date::print() const
36 {
37     if (month_before_day) {
38         cout << month << "/" << day << "/" << year;
39     } else {
40         cout << day << "/" << month << "/" << year;
41     }
42 }
```

2.2.2 A Header File for a Class Declaration

To use the same class in many different C++ programs without having to copy and paste, write the declaration for the class (including the definitions of its inline member functions) in a separate *header* file named after the class. Write the definitions of the non-inline member functions of the class in a `.C` file, also named after the class. This file is sometimes called the class's *implementation* file.

* The other exception involves "templates". See p. 751.

If a header file contained the definition for a variable (not merely the declaration), and if it were included in more than one `.C` file of the same program, we would have more than one copy of the variable and would be wasting memory. For this reason, the definition of the array `date_length` is written in the implementation file `date.C`.

In C++, a constant global variable is static by default: it can be used only in the file in which it is defined. The array can therefore be used only by the functions in the file `date.C`. In C, the array would have needed the keyword `static` to make it static. In C++, the array would have needed the keyword `extern` to make it non-static.

For the `ifndef` machinery in lines 1, 2, and 16, see pp. 81–82. `date.h` must include `iostream` and use namespace `std` because line 14 mentions `cout` and `<<`. `date.C` includes `date.h`, which makes it redundant to include `iostream` and use `std` in lines 1 and 4 of `date.C`. We keep lines 1 and 4 in `date.C` anyway, just in case someone removes them from `date.h`. This could happen if the `print` function was made non-inline and moved back to `date.C`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/date/date.h`

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class date {
 7     int year;
 8     int month;                 //1 to 12 inclusive
 9     int day;                   //1 to date_length[month] inclusive
10 public:
11     date(int initial_month, int initial_day, int initial_year);
12     void next(int count);      //Go count days forward.
13     void next();               //Go one day forward.
14     void print() const {cout << month << "/" << day << "/" << year;}
15 };
16 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/date/date.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 const int date_length[] = {
 7      0,   //dummy element so that January will have subscript 1
 8     31,   //January
 9     28,   //February
10     31,   //March
11     30,   //April
12     31,   //May
13     30,   //June
14     31,   //July
15     31,   //August
16     30,   //September
17     31,   //October
18     30,   //November
19     31    //December
```

```
};

date::date(int initial_month, int initial_day, int initial_year)
{
    if (initial_month < 1 || initial_month > 12) {
        cerr << "bad month " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    if (initial_day < 1 || initial_day > date_length[initial_month]) {
        cerr << "bad day " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    year = initial_year;
    month = initial_month;
    day = initial_day;
}

void date::next(int count)
{
    //Call the other next count times.
    while (--count >= 0) {
        next();
    }
}

void date::next()
{
    //Move to the next date.
    if (++day > date_length[month]) {
        day = 1;
        if (++month > 12) {
            month = 1;
            ++year;
        }
    }
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/date/main.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
    date d(1, 1, 2014);

    cout << "How many days forward from ";
    d.print();
```

```
    cout << " do you want to go? ";

    int count;   //uninitialized variable
    cin >> count;

    d.next(count);
    cout << "The new date is ";
    d.print();
    cout << ".\n";

    return EXIT_SUCCESS;
}
```

When compiling on Unix we mention only the names of the `.C` files, not the names of the `.h` files.

```
1$ g++ -o ~/bin/prog main.C date.C                    Create ~/bin/prog.
2$ ls -l ~/bin/prog
3$ prog
```

```
How many days forward from 1/1/2014 do you want to go? 280
The new date is 10/8/2014.
```

2.3 Scoping Rules

Two groups of variables in scope in a free function

Suppose that the name of a variable is encountered in the body of a member function, and the name is not immediately preceded by any of the operators `.`, `->`, or `::`, or the more exotic ones `.*` or `.->`. The computer will first check if there is a local variable (one declared earlier in the body of the function) with the same name. If so, the computer decides that this variable is the local one. If not, the computer will then check if there is a global variable with the same name. If so, the computer decides that this variable is the global one. If not, the computer gives up and issues an error message.

A variable whose name can be mentioned at a certain point in a program is said to be *in scope* at that point. In the body of a free function two groups of variables are in scope. We repeat the order in which the computer checks them.

(1) The variables declared in the body of the function. These variables are called *local* because they are in scope only in the body of the function in which they were declared.

(2) The variables that are not local. These variables must be declared earlier in the file, and are called *global* because they are in scope throughout the file.

A local variable and a global variable can have the same name (lines 5 and 1). Since the computer checks the local names before the global ones, the local will hide the global (line 7). We would need the unary scope operator `::` in line 8 to access the eclipsed global when there is a local with the same name.

```
1 int i = 10;
2
3 void f()
4 {
5     int i = 20;
6
7     cout << i << "\n";              //the local i in line 5
8     cout << ::i << "\n";            //the global i in line 1
9 }
```

In practice, you should never have a local and a global with the same name. The names of local variable should be short and conventional: `i` for a loop counter, `p` for a pointer. Global names should have more individuality: `max_users`, `current_window`.

Three groups of variables in scope in a member function

In the body of a member function of a class, three groups of variables are in scope.

(1) The local variables.

(2) The members of the class.

(3) The variables that are neither local nor members, which we will now call the globals.

If the name of a variable is encountered in the body of a member function of a class, and if the name is not immediately preceded by one of the operators `.`, `->`, or `::`, or the more exotic ones `.*` or `.->`, the computer first considers the locals, then the members of the class, and finally the globals.

If a local and a member have the same name (`day` in lines 7 and 3), the local will hide the member (line 9). We would need the binary scope operator `::` in line 10 or the `this->` in line 11 to access the eclipsed member when there is a local with the same name. If a member and a global have the same name (`month` in lines 3 and 1), the member will hide the global (line 13). We would need the unary scope operator `::` in line 14 to access the eclipsed global when there is a member with the same name. And as before, a local will hide a global.

```
 1 int month = 10;
 2
 3 //Class date has data members named year, month, day
 4
 5 void date::print() const
 6 {
 7     int day = 20;
 8
 9     cout << day << "\n"             //the local day in line 7
10         << date::day << "\n"        //the day member of class date (binary ::)
11         << this->day << "\n";       //the day member of class date
12
13     cout << month << "\n"           //the month member of class date
14         << ::month << "\n";         //the global month in line 1 (unary ::)
15 }
```

When we do inheritance, four or more groups of variables will be in scope in the body of a member function of a “derived class”. See pp. 479–480.

The above rules apply not only to variables, but also to anything that can have a name: functions, typedefs, enumerations, etc. (Keywords such as `main`, `this`, and `sizeof` do not count as names.) A local (in this case, an enumeration) will hide a member with the same name (line 24), and a member (in this case, a member function) will hide a global with the same name (line 28).

```
16 void print();    //Declaration for a function that is not a member function.
17
18 //Class date has a data member named day and a member function named print.
19
20 void date::next() const
21 {
22     enum {day, night};
23
24     cout << day << "\n"             //the local day in line 22
25         << date::day << "\n"        //the day member of class date
26         << this->day << "\n";       //the day member of class date
27
```

```
28     print();                            //the print member of class date
29     ::print();                          //the global print declared in line 16
30 }
```

2.4 Structures vs. Objects

Why a C++ object is better than a C structure

(1) A member function of an object has a simpler body than a function that takes an explicit pointer. Here are corresponding lines from each body.

```
60         if (++*pmonth > 12) {                 //Version 1, p. 107
64         if (++p->month > 12) {                //Version 2, p. 110
92         if (++month > 12) {                   //Version 3, p. 116
```

Does the ++ in Version 1 increment the `pmonth` or the `*pmonth`? Does the ++ in Version 2 increment the `p` or the `p->month`? These questions disappear in Version 3.

(2) A member function of an object has fewer explicit arguments than a function that takes pointers. Here are corresponding function calls.

```
38     date_next(&day, &month, &year, count);   //Version 1, p. 107
42     next(&d, count);                         //Version 2, p. 109
47     d.next(count);                           //Version 3, p. 116
```

(3) A C structure can easily be created without being initialized, leaving it full of garbage. But a C++ object cannot be created without being initialized, at least if we write a constructor for its class.

(4) If a field of a C structure receives the wrong value, any function in the program might be guilty. But if a data member of a C++ object receives the wrong value, it's easy to make a list of every possible suspect: it must be one of the non-`const` member functions of that object.

(5) If we change the name or data type of a field of a structure, there's no easy way to list all the functions that would have to be rewritten. But if we change the `private` members of an object, only the member functions of the object's class would have to be rewritten. No other function in the program would need to be changed.

For example, here are the private data members of Version 3.

```
22     int year;
23     int month;          //1 to 12 inclusive
24     int day;            //1 to date_length[month] inclusive
```

If we change them to

```
22     int year;
23     int julian;         //1 to 365 inclusive (we're ignoring leap years)
```

then only the member functions of class `date` would have to change, and not even all of them.

Let's walk through a call to the constructor of a two-data-member `date` object, passing it the three arguments 10, 8, and 2014 for October 8, 2014. In line 15 the data member `year` will receive the value 2014, and in line 17 the data member `julian` will receive 8.

(1) The first time we decrement `initial_month` in line 17, it becomes 9 (for September) and `julian` is 8.

(2) The second time we decrement `initial_month` in line 17, it becomes 8 (for August) and `julian` is 8 + 30 = 38.

(3) The third time we decrement `initial_month` in line 17, it becomes 7 (for July) and `julian` is 8 + 30 + 31 = 69.

(4) The tenth (and last) time we decrement `initial_month` in line 17, it becomes 0 (for no month at all) and `julian` is 8 + 30 + 31 + 31 + 30 + 31 + 30 + 31 + 28 + 31 = 281. We then leave the loop.

Conversely, let's walk through a call to the `print` member function of a `date` of this type that contains October 8, 2014. In line 40, the data member `julian` contains 281.

(1) The first time we do the > comparison in line 43, `m` is 1 (for January) and `d` is 281, which represents the outrageous date January 281, 2014.

(2) The second time we do the comparison in line 43, `m` is 2 (for February) and `d` has been reduced to 250, which represents the slightly less outrageous date February 250, 2014.

(3) The third time we do the comparison in line 43, `m` is 3 (for March) and `d` has been reduced to 222, which represents the even less outrageous date March 222, 2014.

(4) The tenth (and last) time we do the comparison in line 43, `m` is 10 and `d` (for October) has been reduced to 8, which represents the legitimate date October 8, 2014. We then leave the loop.

If the `m` in line 41 was declared in the loop in lines 43–45, we would not be able to mention it in line 47 outside the loop.

```
 1 date::date(int initial_month, int initial_day, int initial_year)
 2 {
 3     if (initial_month < 1 || initial_month > 12) {
 4         cerr << "bad month " << initial_month << "/" << initial_day
 5             << "/" << initial_year << "\n";
 6         exit(EXIT_FAILURE);
 7     }
 8
 9     if (initial_day < 1 || initial_day > date_length[initial_month]) {
10         cerr << "bad day " << initial_month << "/" << initial_day
11             << "/" << initial_year << "\n";
12         exit(EXIT_FAILURE);
13     }
14
15     year = initial_year;
16
17     for (julian = initial_day; --initial_month > 0;
18         julian += date_length[initial_month]) {
19     }
20 }
21
22 void date::next(int count)          //same as 3-data-member version
23 {
24     while (--count >= 0) {
25         next();
26     }
27 }
28
29 void date::next()                   //simpler than 3-data-member-version
30 {
31     //Change to the next date.
32     if (++julian > 365) {
33         julian = 1;
34         ++year;
35     }
36 }
37
38 void date::print() const            //more complicated than 3-data-member version
```

```
{
    int d = julian;                  //compute month and day of month
    int m = 1;

    for (; d > date_length[m]; ++m) {
        d -= date_length[m];
    }

    cout << m << "/" << d << "/" << year;
}
```

Are there any other functions that could possibly have to be rewritten as a result of the above change to the private data members of class `date`? Only if our functions try to "x-ray" class `date`, which they have no business doing.

```
    if (sizeof (date) == 3 * sizeof (int)) {    //no longer true
```

```
    date d(12, 31, 2014);
    //no longer gets the year, month, day
    int year  = reinterpret_cast<int *>(&d)[0];
    int month = reinterpret_cast<int *>(&d)[1];
    int day   = reinterpret_cast<int *>(&d)[2];
```

▼ Homework 2.4a: modify the member functions of class date

Make the following changes to the member functions of the class `date` with the three `int` data members `year`, `month`, `day`.

Throughout these changes, the `next` function that takes one explicit argument should remain a public, non-`const`, non-inline member function of class `date`. It must continue to take exactly one explicit argument (an `int`) and return `void`. As before, it should change the contents of the `date` (at least, when its argument is non-zero), and it must produce no output. Do not remove the `print` member function or the constructor. `print` should remain `const`. Do not add any data members to class `date`. Do not use the value of the dummy array element `date_length[0]`: your program should still work even if the element contains garbage. Assume there are no leap years.

To demonstrate that your new class `date` is still correct, run a program consisting of your class `date` and the `main` function and other code in `http://i5.nyu.edu/~mm64/book/src/date/test1/main.C`. Make no changes to this file. The `main` function will create many `date` objects and test their member functions. It examines the standard output of your `print` member function and would become confused by any debugging output directed to the standard output. You should therefore send any debugging output to the standard error output. Hand in your new `date.h`, `date.C`, and the output of the program, in that order. Do not hand in `main.C`.

(1) A *Julian date* is an integer in the range 1 to 365 inclusive giving the day of the year for a particular date. For any year A.D. or B.C.,

(a) The Julian date of January 1 is 1.

(b) The Julian date of January 31 is 31.

(c) The Julian date of February 1 is 32.

(d) The Julian date of December 31 is 365, since we are still ignoring leap years.

Write a member function named `julian` that will return the `date`'s Julian date. Recall how the `print` function required no explicit arguments because the date it printed came from the data members of the `date` object to which the `print` function belonged. In the same way, the `julian` function will require no explicit arguments because the date whose Julian date it returns will come from the data members of the `date` object to which the `julian` function belongs.

`julian` must be a public `const` member function of class `date`. It must output nothing and return an `int`.

If you create any local variables inside the body of `julian`, destroy them as soon as you are done with them. For example, a variable that is used only within a `for` should be declared after the `(` of the `for` loop.

(2) A member function can easily call another member function of the same object. For example, the one-argument `next` function (one explicit argument, that is) called the no-argument `next` function (no explicit arguments) in the `while` loop in lines 79–84 of `version3.C` on p. 116.

But the one-argument `next` would be faster if it did all its work itself, without calling the no-argument `next`. Transplant the body of the no-argument `next` (lines 89–96 of `version3.C` on p. 116) into the body of the `while` loop in the one-argument `next`.

```
 1 void date::next(int count)
 2 {
 3     while (--count >= 0) {
 4         //Move to the next date.
 5         if (++day > date_length[month]) {
 6             day = 1;
 7             if (++month > 12) {
 8                 month = 1;
 9                 ++year;
10             }
11         }
12     }
13 }
```

For the time being, continue to assume (pray) that the `count` argument will be non-negative.

(3) Remove the no-argument `next` function, now that it is no longer called by the one-argument `next` function. But we still want to be able to say

```
14     date d(1, 1, 2014)
15     d.next();           //with no argument
```

To permit this, provide a default value of 1 for the argument of the one-argument `next`. Remember that a default value is specified only in the function declaration, not in the function definition; see p. 95.

(4) The one-argument `next` function (which is now our only `next` function) advances one day toward the answer with each iteration of its `while` loop. But we can do better. Recall that in the two-data-member class `date` on pp. 124–126, the constructor and the `print` member function strode *one full month* toward the answer with each iteration of their loops. In the same way, let the `next` function of the three-data-member class `date` advance one full month with each iteration of its loop. Continue to assume that the `count` argument will be non-negative.

(5) To make our `next` function even faster, lines 18–19 break the `count` into a quotient and remainder. The addition in line 21 will get us to within one year of the target date in a single bound. The `while` loop in lines 23–24 will therefore never need to loop more than 11 times. Continue to assume that the `count` argument is non-negative.

```
16 void date::next(int count)
17 {
18     const int quotient  = count / 365;
19     const int remainder = count % 365;   //in range 0 - 364 inclusive
20
21     add quotient to year;
22
23     the while loop you wrote in ¶ (4) to move year, month, and day
24         remainder additional days forward;
```

```
25 }
```

(6) Deep in most machines, the `/` in the above line 18 actually computes both the quotient and the remainder. The quotient is stored in a variable and the remainder is discarded. The `%` in line 19 also computes both values. This time, the remainder is stored and the quotient is discarded.

To get the quotient and remainder with no wasted effort, call the C Standard Library function `div`. It returns a `div_t` structure containing two integer fields, `quot` and `rem`. Continue to assume that the `count` argument is non-negative.

```
26 #include <cstdlib>      //for div and div_t
27 using namespace std;    //because div, like cout, belongs to namespace std
28
29 void date::next(int count)
30 {
31     const div_t d = div(count, 365);
32     //quotient is d.quot, remainder is d.rem
```

(7) So far, our `next` function works only if its argument is non-negative. Make it work for a negative argument as well:

```
33     date d(10, 8, 2014);
34
35     cout << "280 days before ";
36     d.print();
37     cout << " is ";
38     d.next(-280);
39     d.print();
40     cout << ".\n";
```

We will be able to write the above more legibly when we have ‘‘operator overloading’’.

```
41     date d(10, 8, 2014);
42
43     cout << "280 days before " << d << " is ";
44     d -= 280;                      //d = d - 280;
45     cout << d << ".\n";
```

```
280 days before 10/8/2014 is 1/1/2014.
```

For a non-negative `count` argument, the `div` function will give us a remainder in the range 0 to 364 inclusive. For example, a `count` of 368 will give us a quotient of 1 and a remainder of 3. Unfortunately, a negative `count` will give us a remainder in the range –364 to 0 inclusive. A `count` of –368 will give us a quotient of −1 and a remainder of –3.*

In the above ¶ (4) we wrote a `while` loop that advances a non-negative number of days. To continue to use this loop, without modification, our remainder will have to be non-negative. A `count` of –368 will have to give us a quotient of –2 and a remainder of 362. The `if` in lines 50–53 will perform this adjust-

* At least the `div` function is portable. The `/` and `%` operators are not. On some platforms, we have

```
-368 / 365 ==  -1
-368 % 365 ==  -3
```

On other platforms, we have

```
-368 / 365 ==  -2
-368 % 365 == 362
```

ment, yielding a remainder that is always non-negative. A `count` of −368 will now move us 2 years back and 362 days forward.

```
46 void date::next(int count)
47 {
48     div_t d = div(count, 365);   //no longer const
49
50     if (the remainder is negative) {
51         make the remainder non-negative by adding 365 to it;
52         subtract 1 from the quotient to compensate for the addition;
53     }
54
55     add quotient to year;
56
57     the while loop you wrote in ¶ (4) to move year, month, and day
58         remainder additional days forward;
59 }
```

▲

▼ Homework 2.4b: modify the data members of class date

On pp. 124–126, we changed the three data members of class `date`

```
    int year;
    int month;       //1 to 12 inclusive
    int day;         //1 to date_length[month] inclusive
```

to two data members.

```
    int year;
    int julian;      //1 to 365 inclusive
```

Now change them to one data member

```
    int day;         //number of days before or after Jan 1, 0 A.D.
```

Demonstrate that your new class `date` is still correct by handing in the output of `http://i5.nyu.edu/~mm64/book/src/date/test1/main.C`. Hand in your new `date.h`, `date.C`, and the output, in that order. Do not hand in `main.C`.

Your class will be the most simple if your data member contains the number of days before or after January 1, 0 A.D. (We are pretending that there was a year 0 A.D.) Your class will be unnecessarily complicated if your data member contains the number of days before or after December 31, −1 A.D. Here are some examples of correct values.
`day` will contain 0 when the object contains January 1, 0 A.D.
`day` will contain 1 when the object contains January 2, 0 A.D.
`day` will contain 30 when the object contains January 31, 0 A.D.
`day` will contain 31 when the object contains February 1, 0 A.D.
`day` will contain $59 = 31 + 28$ when the object contains March 1, 0 A.D.
`day` will contain 364 when the object contains December 31, 0 A.D.
`day` will contain 365 when the object contains January 1, 1 A.D.
`day` will contain $365 \times 2014 = 735{,}110$ when the object contains January 1, 2014 A.D.
`day` will contain −1 when the object contains December 31, −1 A.D. (i.e., 1 B.C.)
`day` will contain −365 when the object contains January 1, −1 A.D. (i.e., 1 B.C.)

After dividing the data member by 365, make the remainder non-negative by writing an `if` like the one in lines 50–53 of the previous Homework. The quotient will then give the year. The remainder will be in the range 0 to 364 inclusive, giving the day of the year. Add 1 to get the Julian date.

If a function needs both the quotient and the remainder, call the `div` function. If only the remainder is needed, use the `%` operator.

Class `date` must have no data member other than the `int day`. Create no global variables or static local variables. Do not use the value of the dummy array element `date_length[0]`. Assume there are no leap years.

Make no change from the previous Homework in the names, argument types, return values, or `const`'ness of the public member functions of class `date`. For example, the constructor for `date` must still take three arguments (month, day, year) even though class `date` now has only one data member. And the `print` member function must still output the `date` in the format `m/d/y`. The `next` member function must still accept an argument that is non-negative or negative. The default value of this argument must still be `1`. The `julian` member function will still take no arguments and return an `int` in the range 1 to 365 inclusive.

If any of the member functions are now short enough, make them inline. If your `date.h` file no longer needs to include `iostream` or use `namespace std`, remove these lines. If the lines are now needed in `date.C`, move them there.

▲

2.4.1 Constant Objects and Pointers Thereto

Which member functions can we call?

We can call any member function of a non-`const` object.

```
1       date d(12, 31, 2014);
2       d.print();                      //a const member function
3       d.next();                       //a non-const member function
```

But we can call only the `const` member functions of a `const` object.

```
4       const date e(12, 31, 2014);
5       e.print();                  //will compile because print is const
6       e.next();                   //won't compile because next isn't const
```

Any member function that can be `const` should be `const` so that it can be called for a `const` object such as `e`. A member function can be `const` if it changes no data member of its object and calls no non-`const` member function of its object.

Why can the above line 4 call the constructor, which is not a `const` member function? A `const` object becomes constant when we return from its constructor, either by a `return` statement or simply by reaching the `}` at the end of the constructor's body. Until then, we can still modify the object's data members and call its non-`const` member functions. (On p. 268, we will see the moment when a `const` object ceases to be `const`.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_obj/const_obj.C`

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

class obj {
    int i;
public:
    obj(int initial_i) {i = initial_i; f();}
    void f() {cout << "I am not currently a const object.\n";}
};

```

```
int main()
{
    const obj o(10);   //will compile, even though constructor calls f
    //o.f();           //won't compile: too late to call f
    return EXIT_SUCCESS;
}
```

```
I am not currently a const object.
```

Pointers and references to objects

A pointer and reference to an object have the same syntax as a pointer and reference to a structure. In the following program, the structure is in column 1 and the object in column 2.

Line 22 applies the dereferencing operator * to the pointer to get the pointed-to variable. Since the variable turns out to be a structure or object, we then apply the dot operator and the name of a field or member. Parentheses are necessary to execute the * operator before the dot operator. When the member is a member function, there is one more step: we apply the function call operator. It comes last because it and the dot have left-to-right associativity.

But don't write line 22. Line 26 is a simpler way to do the same thing. The arrow operator -> does the work of the two operators * and dot. And now that there is only one operator, we no longer need the parentheses.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/const_obj/pointer.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 struct mystruct {
 7     int field1;
 8     int field2;
 9 };
10
11 int main()
12 {
13     mystruct s = {10, 20};          date d(12, 31, 2014);
14
15     cout << s.field1 << "\n";       d.print();
16                                     cout << "\n\n";
17
18     //A pointer to a structure and a pointer to an object.
19     mystruct *p1 = &s;              date *p2 = &d;
20
21     //Don't write this.
22     cout << (*p1).field1 << "\n";   (*p2).print();
23                                     cout << "\n\n";
24
25     //Write this instead.
26     cout << p1->field1 << "\n";     p2->print();
27                                     cout << "\n\n";
28
29     //A reference to a structure and a reference to an object
30     mystruct& r1 = s;               date& r2 = d;
```

```
31
32     cout << r1.field1 << "\n";       r2.print();
33                                      cout << "\n";
34
35     return EXIT_SUCCESS;
36 }
```

```
10            lines 15–16
12/31/2014

10            lines 21–23
12/31/2014

10            lines 25–27
12/31/2014

10            lines 32–33
12/31/2014
```

What types of pointers can point at a const object?

Only a read-only pointer can point to a `const` object; only a read-only reference can refer to a `const` object. The pointers are in column 1; the references, in column 2.

```
1      const date d(12, 31, 2014);
2
3      //won't compile
4      date *p = &d;                    date& r = d;
5
6      //will compile
7      const date *p = &d;              const date& r = d;
```

If the above line 4 were legal, we could change a `const` object by saying `p->next()` or `r.next()`.

Which member functions can we call with a read-only pointer or reference?

Once again, the pointers are in column 1; the references, in column 2.

The object in line 1 is not `const`. But the pointer and reference to it in line 4 are read-only. Using them, we can call only the `const` member functions of the object. Of course, by going directly to the object in line 12 we can still call any member function.

```
1      date d(12, 31, 2014);
2
3      //read-only pointer and reference
4      const date *p = &d;              const date& r = d;
5
6      //will compile because print is const
7      p->print();                      r.print();
8
9      //won't compile because next isn't const
10     p->next();                       r.next();
11
12     d.next();
```

An object can be broken into

> Hiding [making the data members `private`] is for the prevention of accidents, not the prevention of fraud.
>
> —Bjarne Stroustrup in Booch, *Object-Oriented Analysis and Design, 2nd ed.*, p. 54

The values of the private data members of an object can be changed by a determined intruder, even if it is not a member function of the object. For example, here is a different `main` function for Version 3.

Line 9 performs “type punning”. The expression `&d` is a pointer to a `date`, not a pointer to a `char`, so we need a `reinterpret_cast` to store it into `p`. The value of `p` is now the address of the first byte of `d`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_obj/broken.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date d(12, 31, 2014);
 9     char *const p = reinterpret_cast<char *>(&d);
10
11     for (size_t i = 0; i < sizeof d; ++i) {
12         p[i] = '\0';   //Overwrite d with zeroes.
13     }
14
15     d.print();
16     cout << "\n";
17     return EXIT_SUCCESS;
18 }
```

```
0/0/0
```

2.5 Constructors

Declare and define a constructor

A constructor has four distinguishing features. Compare the rules for a “destructor” on p. 154.

(1) A constructor’s name is the same as that of its class. For example, in `version3.C` the class is named `date` (p. 115, line 21) and the class’s constructor is also named `date` (declared in line 26, called in line 37, defined in lines 55–72).

(2) Do not declare a return type for the constructor. Don’t even declare the constructor’s return type to be `void`: just write nothing at all for the return type as in lines 26 and 55 of `version3.C`. Do not write a `return` statement that returns any value from the constructor. The constructor will implicitly return the newly-constructed object.

(3) Don’t declare the constructor to be `const` (as `print` was in `version3.C` on pp. 115–116, lines 29 and 99), even if it changes the value of no data member and calls no non-`const` member function of the same object.

(4) For the time being, a constructor must always be public, not private. (Our first private constructor will be on p. 295; examples in the standard library will be in the stream classes on pp. 324–326 and the `type_info` class on p. 1017.) If the constructor for a class were `private`, it could be called only by the

member functions of another object of that class. That other object could have been constructed only by a third object. We would have an infinite regress (a ‘‘chicken and egg’’ situation).

My convention is to name each argument of the constructor after the data member that it initializes, with a leading `initial_`. The same name could be used for the argument and the data member, if we write a `::` or `->` in front of it when it refers to the data member. But using the same name for two different variables is always confusing. Line 3 uses the scope operator we saw on p. 123, line 10. Line 9 uses the arrow we saw on p. 123, line 11; and on p. 117, line 105 of `version3.C`. For brevity, we do not show the error checking.

```
 1 date::date(int month, int day, int year)
 2 {
 3     date::year = year;
 4     date::month = month;
 5     date::day = day;
 6 }

 7 date::date(int month, int day, int year)
 8 {
 9     this->year = year;
10     this->month = month;
11     this->day = day;
12 }
```

For the time being, a constructor should initialize every member of the object. (Our first exception will be the constructor for class `stack` on p. 149.) The number of arguments of the constructor does not necessarily have to be the same as the number of data members. An example was the constructor for the above class `date` with a `julian` data member.

Syntax for calling a constructor in a declaration

When calling a constructor with two or more arguments (line 26), we must surround them with parentheses. We saw this syntax in line 37 of `version3.C` on p. 115.

When calling a constructor with exactly one argument, we have a choice of notation. We can surround the argument with the parentheses in line 29, or precede it with the equal sign in line 30. Since both do the same thing (subject to the caveat on p. 137), the choice serves only as documentation. Write the parentheses to emphasize that a function is called to initialize the object. Write the equal sign to make the user think of the object as merely a holder for a value.

The equal sign notation was provided so that we could use the same syntax when declaring variables of all data types. A variable of a built-in type is initialized with an equal sign (line 36), and now we can do the same for an object (line 30), at least when its constructor has exactly one argument. Conversely, an object is initialized with parentheses (lines 26 and 29), and we can also do the same for a built-in (line 37). We can pretend that each built-in data type has a constructor that takes one argument of the same type. Using the same syntax for objects and built-ins will make a feature called ‘‘templates’’ applicable to all these types (p. 634).

The default constructor

Class `zero` has no data members. It would be unheard-of for a C structure to have no fields, but it is quite reasonable for a C++ class to have no data members. Examples are on pp. 590, 625, and 842.

Class `zero` has a *default constructor:* one that can be called with no arguments, either because it has no arguments at all, or because every argument has a default value.

We would expect that line 33 would be the syntax for calling a default constructor. But in C and C++, this syntax is already used for declaring a function.

```
    int f();        //declare a function that takes no arguments and returns an int
    zero z1();      //declare a function that takes no arguments and returns a zero
```

To call the default constructor, we must write line 34. (When the object constructed by a default constructor is anonymous, the parentheses have to be written; see p. 137. Yet another syntax for calling a deafult constructor, for use only within a template, will be given on p. 660.)

The copy constructor

A *copy constructor* creates a copy of an existing object. It takes exactly one argument, a read-only reference to another object of the same class as the one being constructed. The copy constructor for class `mono` is declared in line 16, defined in lines 57–61, and called in line 39 (and maybe called also in line 30; see the caveat on p. 137). Class `mono` can have two constructors (lines 15 and 16) because their arguments are different; this is an example of function name overloading. On the other hand, a class can have at most one default constructor and at most one copy constructor.

Our convention is to use the name `another` for the argument of a copy constructor. It must *always* be passed by reference. Were it passed by value, it would have to be copied before the function call (pp. 69–70). But `another` is an object, and the only way to copy an object is to call its copy constructor. Therefore every call to the copy constructor would have to be preceded by another call to the same copy constructor, and we would go into an infinite loop.

A member function can always access the private members of its own object. This copy constructor is our first example of a member function that can also access the private members of another object of the same class (p. 201). When called from line 39, for example, the `another.member1` in line 60 is a member of the object `m1`, and the plain `member1` in line 60 is a member of the object `m3`.

The copy constructor must not change the value of `m1` in line 39. To ensure this, we declare its reference argument to be `const` in lines 16 and 57.

Had we not defined a copy constructor for class `mono`, line 39 would have worked anyway. The computer would have behaved as if we had defined the following one.

```
mono::mono(const mono& another)
{
	//"memberwise" copy: copy each data member of the other object
	//into the corresponding member of the new object.

	member1 = another.member1;
}
```

(We'll see on p. 261 that the computer actually behaves as if we had written a copy constructor with a colon.

```
mono::mono (const mono& another)
	: member1(another.member1)
{
}
```

With the colon, it *initializes* rather than *assigns to* each data member of the newborn object.)

While learning C++, the author often put an output statement into a constructor just to verify that the constructor is called. We wrote our own copy constructor only because the one provided implicitly would not have had the output statement in line 59. Sometimes, however, there is a non-trivial reason to write our own copy constructor. This will happen on pp. 153 and 306–307, when we have a data member that is a pointer.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/constructor/duo.C`

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

class duo {
```

```
    int member1;
    int member2;
public:
    duo(int initial_member1, int initial_member2);
};

class mono {
    int member1;
public:
    mono(int initial_member1);
    mono(const mono& another);   //copy constructor
};

class zero {
public:
    zero();
};

int main()
{
    duo d1(10, 20);
    //duo d2;         //won't compile: constructor needs arguments

    mono m1(10);
    mono m2 = 10;     //another way to do the same thing when the
                      //constructor has exactly one argument

    //zero z1();      //creates no object, calls no constructor
    zero z2;          //call a constructor that has no arguments

    int i = 10;
    int j(10);        //another way to do same thing

    mono m3 = m1;      //call copy constructor: m3.mono(m1)

    return EXIT_SUCCESS;
}

duo::duo(int initial_member1, int initial_member2)
{
    cout << "constructor for duo\n";
    member1 = initial_member1;
    member2 = initial_member2;
}

mono::mono(int initial_member1)
{
    cout << "constructor for mono\n";
    member1 = initial_member1;
}

mono::mono(const mono& another)   //copy constructor
{
    cout << "copy constructor for mono\n";
```

```
60     member1 = another.member1;
61 }
62
63 zero::zero()
64 {
65     cout << "constructor for zero\n";
66 }
```

One final caveat. Most modern compilers will create `m2` in the above line 30 by calling the constructor that takes an integer.

`constructor for duo`	*line 26*
`constructor for mono`	*line 29*
`constructor for mono`	*line 30 constructs* `m2`
`constructor for zero`	*line 34*
`copy constructor for mono`	*line 39*

But the compiler would be within its rights if it created `m2` in a more roundabout way. An older compiler, or the `g++` compiler with the option `-fno-elide-constructors`, could create an *anonymous temporary* object—one that has no name—when it sees the `= 10` in the above line 30. It would do this by calling the constructor that takes an integer (line 51).* Then `m2` could be created by calling the copy constructor (line 57).

`constructor for duo`	*line 26*
`constructor for mono`	*line 29*
`constructor for mono`	*line 30 constructs an anonymous temporary* `mono`
`copy constructor for mono`	*line 30 copies the anonymous temporary into* `m2`
`constructor for zero`	*line 34*
`copy constructor for mono`	*line 39*

As we saw in the first box of output, a newer compiler will *elide* the anonymous temporary in line 30. For other examples of elision, see pp. 190–191, 234–236, and 660.

Construct an anonymous object

We have made it sound as if a constructor can be called only in a declaration for an object. But it can also be called to create an anonymous temporary object. A variable needs no name if it would be mentioned only once. Our examples will be a `double` and a `date`.

(1) To print the square root of 2, all we have to do is to print the `double` returned by the `sqrt` function in line 1. There is no need to declare the unnecessary variable in line 5 to hold the square root and print it in line 6.

Similarly, to print a `date`, all we have to do is to print the `date` returned by the constructor function in line 1. A constructor returns the newly-constructed object, even though we return no explicit value when defining the constructor. When constructing an anonymous object, the name of the constructor must always be followed by parentheses, even if the parentheses enclose no arguments. There is no need to declare the unnecessary variable in line 5 to hold the `date` and print it in line 6.

The << operator that outputs the `date` in line 1 will not compile yet; we'll make it work on p. 335 when we do operator overloading. In the meantime, line 2 can call the `print` member function of the anonymous temporary object returned by the constructor. At least this is better than an unnecessary declaration. (This is our first example of calling a member function of an anonymous temporary object returned by a function. Others will be line 11 of `return_obj.C` on p. 190 and line 11 of `return.C` on p. 191;

* This is legal because the constructor in line 51 was declared without the keyword `explicit`.

pp. 203−204 and 209; p. 292; line 20 of `path.C` on p. 322; line 8 on p. 326.)

```
 1     cout << sqrt(2.0) << "\n";               //cout << date(12, 31, 2014) << "\n";
 2                                              date(12, 31, 2014).print();
 3                                              cout << "\n";
 4
 5     double unnecessary_double = sqrt(2.0); date unnecessary_date(12, 31, 2014);
 6     cout << unnecessary_double << "\n";    cout << unnecessary_date << "\n";
```

(2) To pass the square root of 2 to a function `f`, all we have to do is to take the `double` returned by the `sqrt` function and pass it to `f` in line 7. There is no need to declare the unnecessary variable in line 9 to hold the square root and pass it to `f` in line 10.

Similarly, to pass a `date` to a function `f`, all we have to do is to take the `date` returned by the constructor function and pass it to `f` in line 7. There is no need to declare the unnecessary variable in line 9 to hold the `date` and pass it to `f` in line 10. (Thanks to function name overloading, the `f` that receives the `double` is not the same function as the one that receives the `date`.)

```
 7     f(sqrt(2.0));                            f(date(12, 31, 2014));
 8
 9     double unnecessary_double = sqrt(2.0); date unnecessary_date(12, 31, 2014);
10     f(unnecessary_double);                   f(unnecessary_date);
```

(3) Line 11 creates the variable `x`. To change the value of `x` to the square root of 2, all we have to do is to take the `double` returned by the `sqrt` function and assign it to `x` in line 12. There is no need to declare the unnecessary variable in line 14 to hold the square root and assign it to `x` in line 15.

Similarly, line 11 creates the variable `d`. To change the value of `d` to a new `date`, all we have to do is to take `date` returned by the constructor function and assign it to `d` in line 12. There is no need to declare the unnecessary variable in line 14 to hold the new `date` and assign it to `d` in line 15.

```
11     double x = 10.0;                         date d(1, 1, 2014);
12     x = sqrt(2.0);                           d = date(12, 31, 2014);
13
14     double unnecessary_double = sqrt(2.0); date unnecessary_date(12, 31, 2014);
15     x = unnecessary_double;                  d = unnecessary_date;
```

(4) To return the square root of 2 from a function, all we have to do is to return the `double` returned by the `sqrt` function in line 16. There is no need to declare the unnecessary variable in lines 18−19 to hold the square root and return it in line 20.

Similarly, to return a `date` from a function, all we have to do is to return the `date` returned by the constructor function in line 16. There is no need to declare the unnecessary variable in lines 18−19 to hold the `date` and return it in line 20. (The unnecessary variables in lines 19−19 could be `const`'s because they will never be changed. They are destructed in the very next line.)

```
16     return sqrt(2.0);                        return date(12, 31, 2014);
17
18     const double                             const date
19         unnecessary_double = sqrt(2.0);          unnecessary_date(12, 31, 2014);
20     return unnecessary_double;               return unnecessary_date;
```

If the constructor has exactly one argument, we can omit the name of the constructor and the parentheses around the argument in the above line 16.

```
21     return duo(10, 20);                      //Must write "duo" & parentheses.
22     return mono(10);                         //Could write "mono" & parentheses,
23     return 10;                               //but don't have to.
```

This will work only if the constructor was declared without the keyword `explicit`.

(5) A final warning. Never write the following call to the `sqrt` function in a statement all by itself in C or C++. It would return an anonymous `double`, which would then be discarded without having been used. In other words, it would be a dead value; see p. 37. Similarly, never write the following call to the constructor function for class `date` in a statement all by itself. It would construct and return an anonymous `date` object, which would also be a dead value.

```
24     sqrt(2.0);                      date(12, 31, 2014);
```

Can one constructor call another one for the same object?

The following class `myobj` has the two constructors called in lines 26 and 27. Let's call them the "ID constructor" and the "DI constructor" respectively. Since they do the same work, we want to write it only once. We'll do the work in the ID constructor, and the DI constructor will merely be a call-through to the ID constructor.

The DI constructor attempts to call the ID constructor in line 13. We hope this will work because the syntax of line 13 imitates that of line 10, which successfully calls another member function of the same object. We saw one member function calling another member function of the same object as early as lines 80–83 of Version 3 on p. 116.

But this syntax will work only if the other function is not a constructor. If the other function *is* a constructor, as in line 13, we will be committing the blunder in the above line 24. Line 13 does not call another member function of the same object. It constructs and discards a separate, anonymous object, which has no effect on the object that the DI constructor is trying to construct.

Lines 14 and 16 are vain attempts to make it work, but 14 has the same bug and 16 will not even compile. Line 18 does work, but only at the price of constructing a separate, anonymous object, and copying it into the object `*this` that the DI constructor is constructing. The DI constructor has therefore constructed a total of two objects. That's too expensive for us.

```
 1 class myobj {
 2     int i;
 3     double d;
 4 public:
 5     myobj(int initial_i, double initial_d) {i = initial_i; d = initial_d;}
 6
 7     myobj(double initial_d, int initial_i) {
 8
 9         //Call another member function of the same object:
10         f(initial_i, initial_d);
11
12         //2 unsuccessful attempts to call another mem func of same object:
13         myobj(initial_i, initial_d);
14         myobj::myobj(initial_i, initial_d);
15
16         this->myobj(initial_i, initial_d);      //won't compile
17
18         *this = myobj(initial_i, initial_d);
19     }
20
21     void f(int i, double d) const {}
22 };
23
24 int main()
25 {
26     myobj m1(10, 3.14159);          //ID: int and double
27     myobj m2(3.14159, 10);          //DI: double and int
28
```

```
    return EXIT_SUCCESS;
}
```

How to do it

If two constructors have the same work to do, they should call a common private member function. Do not attempt to have one constructor call another constructor for the same object. Every object should have exactly *one* constructor called for it.

```
class myobj {
    int i;
    double d;
    void init(int initial_i, double initial_d) {i = initial_i; d = initial_d;}
public:
    myobj(int initial_i, double initial_d) {init(initial_i, initial_d);}
    myobj(double initial_d, int initial_i) {init(initial_i, initial_d);}
};
```

Get the date and time in C

We will define another constructor for class `date`, one that initializes the newborn object to today's date. Here is the code in C and C++ to get the current date from the operating system.

```
/* Excerpt from <time.h>, showing some of the fields of struct tm. */

typedef long time_t;    /* may not be long on all platforms */

struct tm {
    int tm_mday;        /* 1 to 31 inclusive */
    int tm_mon;         /* 0 to 11 inclusive */
    int tm_year;        /* year minus 1900 */
    /* etc. */
};
```

In versions of C prior to C99, the declarations in a block must always come before the other statements; see pp. 32–33. This forces line 8 to leave `p` uninitialized: the assignment to `p` in line 15 must come *after* the `if`, while the declaration of `p` must come *before* the `if`. C also needs the keyword `struct` in line 8.

What could go wrong without the cast in line 10? On my platform, `time_t` is another name for the data type `long`. Imagine, however, that it was another name for `unsigned short`, and that an `unsigned short` was narrower than an `int` (16 and 32 bits respectively). If the call to `time` in line 7 failed, the `t` in line 10 would hold the value $2^{16} - 1 = 65,535$, the 16-bit unsigned equivalent of −1. This value would be promoted to `int` to match the other operand of the equality in line 10, the `-1`. The promotion would be by zero-extension (p. 61), resulting in an `int` value of 65,535. But 65,535 does not equal −1, and the `if` would not detect the failure of the `time` function. Of course, a `time_t` would never be 16 bits. But in some future implementation, `time_t` and `int` might be 32 and 64 bits, causing the same bug.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/constructor/time.c`

```
1 #include <stdio.h>    /* C example */
2 #include <stdlib.h>
3 #include <time.h>     /* for time_t, time, localtime */
4
5 int main(int argc, char **argv)
6 {
7     const time_t t = time(NULL);
```

```
 8     const struct tm *p;   /* uninitialized and not *const */
 9
10     if (t == (time_t)-1) {
11         fprintf(stderr, "%s: time failed\n", argv[0]);
12         return EXIT_FAILURE;
13     }
14
15     p = localtime(&t);
16
17     printf(  "day == %d\n", p->tm_mday);
18     printf("month == %d\n", p->tm_mon + 1);
19     printf( "year == %d\n", p->tm_year + 1900);
20
21     return EXIT_SUCCESS;
22 }
```

```
day == 8
month == 4
year == 2014
```

Get the date and time in C++

In C++, we write zero instead of `NULL` in line 8; see p. 68. Line 15 does not need the keyword `struct`; see line 33 of `version2.C` on p. 109. Even better, `p` can now be a `*const` pointer since it is initialized in its declaration.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/constructor/time.C`

```
 1 #include <iostream>   //C++ example
 2 #include <cstdlib>
 3 #include <ctime>
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     const time_t t = time(0);
 9
10     if (t == static_cast<time_t>(-1)) {
11         cerr << argv[0] << ": time failed\n";
12         return EXIT_FAILURE;
13     }
14
15     const tm *const p = localtime(&t);   //initialized and *const
16
17     cout << "day == " << p->tm_mday << "\n"
18        << "month == " << p->tm_mon + 1 << "\n"
19         << "year == " << p->tm_year + 1900 << "\n";
20
21     return EXIT_SUCCESS;
22 }
```

```
day == 8
month == 4
year == 2014
```

▼ Homework 2.5a: give a class another constructor

Class `date` already has the three-argument constructor declared in line 26 of `version3.C` on p. 115. Keep it, but also provide a default constructor declared as follows.

```
public:
    date(int initial_month, int initial_day, int initial_year); //3-arg const.
    date();  //default constructor
```

The default constructor will put today's date into the data member(s) of the newborn `date` object.

You can use the version of class `date` with either one, two, or three data members. If your version of class `date` has a `month` data member whose value is a number in the range 1−12 inclusive, store `tm_mon` plus 1 into it. If your version of class `date` has a `year` data member whose value is the year, store `tm_year` plus 1900 into it.

Do not remove the `next` and `julian` member functions.

In the following test program, the objects in lines 8 and 12 have names; the one in line 16 is an anonymous temporary. The temporary is constructed by the default constructor for class `date`.

There is also a default constructor for each built-in data type, which puts a zero into the newborn variable. Line 19 calls the default constructor for the data type `int` and outputs the value of the resulting anonymous temporary. These constructors are intended for use only in a "template" (p. 660) where it is advantageous to having the same syntax for all data types (p. 634). An example is line 13 on p. 796. Elsewhere, we should simply write `0` instead of `int()`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/constructor/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date d1(12, 31, 2014); //parentheses for more than one argument
 9     d1.print();
10     cout << "\n";
11
12     date d2;               //no parentheses for no arguments
13     d2.print();
14     cout << "\n";
15
16     date().print();        //Construct an anonymous date; call its print function.
17     cout << "\n";
18
19     cout << int() << "\n"; //Construct an anonymous int with the value 0.
20     return EXIT_SUCCESS;
21 }
```

12/31/2014	*lines 8–10*
4/8/2014	*lines 12–14*
4/8/2014	*lines 16–17*
0	*line 19*

▲

Review of typedef

The next Homework will require a typedef statement, so we review it now.

Despite its name, a typedef does not create a new data type. It merely creates a one-word alternative name for an existing data type. For example, the typedef in line 4 creates the nickname `number_t` for the data type `long unsigmed`.

A typedef makes the code easier to change. For example, instead of writing the old name `long unsigned` three times in lines 1–3, we can write it once in line 4. The variable in line 6 will have exactly the same data type as the one in line 1.

```
1       long unsigned n1 = 10;
2       long unsigned n2 = 20;
3       long unsigned n3 = 30;

4       typedef long unsigned number_t; //From now on, number_t means long unsigned.
5
6       number_t n1 = 10;
7       number_t n2 = 20;
8       number_t n3 = 30;
```

Instead of the typedef in the above line 4, `number_t` could have been a macro.

```
9   #define number_t long unsigned
```

But there is no practical way to use a macro for a data type whose name consists of two separate parts, such as the `char [4]` ("array of four characters") in lines 10–12. Instead of writing this name three times, we should have written it only once in line 13.

```
10      char   kennedy[4] = "JFK";
11      char laguardia[4] = "LGA";
12      char    newark[4] = "EWR";

13      typedef char airport_t[4];      //An airport_t is an array of 4 char's.
14
15      airport_t kennedy   = "JFK";
16      airport_t laguardia = "LGA";
17      airport_t newark    = "EWR";
```

The name created by a typedef conventionally ends in `_t`, at least in the C Standard Library. The examples we have used with class `date` are `div_t` and `time_t`; the most important ones are `size_t` and `ptrdiff_t`. The C++ Standard Library contains all the C typedefs except `wchar_t`, which is now a keyword in C++. For many of its own typedefs, the C++ library has abandoned the `_t` suffix in favor of `_type`: `size_type`, `difference_type`, and `value_type`.

One common use of typedef in C is now unnecessary in C++. A C structure was a second-class citizen, needing the helping word `struct` in line 23.

```
18  struct mystruct {                    /* C example */
19      int field1;
20      int field2;
21  };
22
```

```
23 struct mystruct m = {10, 20}; /* name of data type is "struct mystruct" */
```

A typedef was often used to create a one-word name for the data type.

```
24 typedef struct {                        /* C example */
25     int field1;
26     int field2;
27 } mystruct;
28
29 mystruct m = {10, 20};    /* name of data type is now only one word */
```

But the C++ structure in line 35 does not need the keyword `struct`, just as the C++ object in 36 does not need the keyword `class`. The typedef in the above line 24 is therefore no longer necessary.

```
30 struct {                                //C++ example
31     int field1;
32     int field2;
33 } mystruct;
34
35 mystruct m = {10, 20};
36 date d(4, 8, 2014);
```

▼ Homework 2.5b: do the work in the member functions of a class

We translated the game of life into C++ on pp. 42–44, but not the way the language should really have been used. This Homework will be another step in the right direction.

Package the answer to the Game of Life homework as a class named `life`. One benefit of doing so will appear on pp. 170–172. For the present, declare the class in a header file named `life.h`. Define the three non-inline member functions in a file named `life.C` (or `life.cpp`, or whatever the filename extension is on your platform).

Test the class with the following `main` function. Lines 22–25 call the four member functions of class `life`, starting with the one-argument constructor for the object named `glider` in line 22.

I try to use consistent names for the member functions of all my classes. Classes `date` and `life` have member functions named `next` and `print`. When we do operator overloading, we'll give these functions their proper C++ names: `operator++` (pp. 288–289), and `operator<<` (pp. 337–340).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/life/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>   //for strcmp
 4 #include "life.h"    //for class life, life_xmax, and life_ymax
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const bool glider_matrix[life_ymax][life_xmax] = { //sorry y before x
10         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
11         {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
12         {0, 0, 1, 1, 0, 0, 0, 0, 0, 0},
13         {0, 1, 1, 0, 0, 0, 0, 0, 0, 0},
14         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
15         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
16         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
17         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
18         {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };

    for (life glider = glider_matrix;; glider.next()) {
        glider.print();

        cout << glider.generation()
            << ": Press c to continue, q to quit, and RETURN.\n";

        char buffer[256];
        cin >> buffer;
        if (strcmp(buffer, "c") != 0) {
            break;
        }
    }

    return EXIT_SUCCESS;
}
```

The nested `for` loops that print the matrix data member will now be in the `print` member function of class `life`. The code that updates the matrix data member (and the `++generation`) will be in the `next` member function of class `life`. The code that initializes the two data members will be in the constructor for class `life`. Remember to initialize the cells along the edge of the array to `false`.

The matrix originally named `old` is used continuously during the lifetime of the game. It should be enshrined as a data member of class `life`, private, as any data member should be. The constructor will fill in its initial value, including the `false`'s along the edges.

On the other hand, the matrix originally named `new` is used only by the code that updates the `old` matrix. It is then discarded, and created anew the next time we update the `old` matrix. Since it is used only intermittently, the `new` matrix should be a local variable in the `next` member function of class `life`.

The `life_ymax` and `life_xmax` in lines 5–6 of `life.h` are not data members of class `life`. They merely float somewhat unsatisfactorily near it as global variables. See the similar disposition of the array `date_length` in `version3.C` on pp. 114–115.

As in C, a *global* variable is one that is declared and defined outside the body of any function. A *static* global variable is one that can be mentioned in only one `.C` file. A global variable is often declared in a header file, but rarely defined in one.

To see why, assume that the header were included in more than one `.C` file of the same program. If a static global variable were defined in the header, we could be wasting memory: each `.C` file that included the header could get its own private copy of the variable. Worse, if a non-static global variable were defined in the header, the program would not even link. The global variable would be "multiply defined", causing an error message.

But in lines 5–6, our header defines two global variables anyway. We can get away with this because in C++, a `const` global variable is static by default. (To make it non-static, we would need the keyword `extern` in front of the `const`.) Our C++ program will link, wasting a bit of memory when we include the header in more than one `.C` file. But this is a necessary evil. `life_ymax` and `life_xmax` are used as array dimensions in line 10, so they must be defined, not merely declared, earlier in this header file.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/life/life.h`

```
1 #ifndef LIFEH
2 #define LIFEH
3 #include <cstddef>   //for size_t
4
5 const size_t life_ymax = 10;
```

```
 6 const size_t life_xmax = 10;
 7
 8 class life {
 9     int g;        //generation number
10     bool matrix[life_ymax + 2][life_xmax + 2];
11 public:
12     life(const bool initial_matrix[life_ymax][life_xmax]);
13     int generation() const {return g;}
14     void next();  //Advance the matrix data member 1 generation; add 1 to g.
15     void print() const; //Print the matrix data member.
16 };
17 #endif
```

Also make the following changes.

(1) Instead of hardwiring the `'X'`s and `'.'`s into the first `cout` statement in the `print` member function, give the function two arguments:

```
18     void print(char filled, char empty) const;
```

Let the arguments have the default values `'X'` and `'.'` respectively. See pp. 94–97 for default values.

(2) At line 7 of `life.h`, write a typedef to declare that `life_matrix_t` is another name for the data type “`life_ymax` × `life_xmax` array of `bool`’s”. Then change the name of the data type of the following two variables to `life_matrix_t`.

(a) the local variable `glider_matrix` in the `main` function (which should also be `const`);

(b) the `initial_matrix` argument of the constructor (which should also be `const`).

(3) At line 7½ of `life.h`, write a typedef to declare that `_life_matrix_t` (with a leading underscore) is another name for the data type “(`life_ymax` + 2) × (`life_xmax` + 2) array of `bool`’s”. This typedef is only for internal use by class `life`. Then change the name of the data type of the following two variables to `_life_matrix_t`.

(a) the `matrix` data member of class `life`;

(b) the local variable `newmatrix` in the `next` member function.

(4) Define a new public member function of class `life` to display the game by means of the `term_` functions written in C:

```
19     void put(char filled, char empty) const;
```

`life::put` will put the `matrix` data member, or as much of it as will fit, upper-left justified onto the screen. See the extra credit part of the Game of Life homework.

Let the arguments have the default values `'X'` and `'.'` respectively. Assume that `term_construct` has already been called before the first call to `life::put`, and that `term_destruct` will be called after the last call to `life::put`. Call the two-argument `min` function (pp. 43–44) to compute the minimums in lines 23–24.

```
20 void life::put(char filled, char empty) const
21 {
22     //How much of the game will fit on the screen?
23     const size_t ymax = minimum of term_ymax() and life_ymax;
24     const size_t xmax = minimum of term_xmax() and life_xmax;
25
26     for (a pair of...
27         for (...classic nested "for" loops
28             const char c = either filled or empty, depending on the
```

```
                contents of the matrix data member;

            if (term_get() says c is not already at this place on the screen) {
                term_put(c at this place);
            }
        }
    }
}
```

Test `life::put` without using `cin` or `cout`:

```
    term_construct();         //before constructing any life objects

    for (life glider = glider_matrix;; glider.next()) {
        glider.put();

        //Don't bother to display the generation number.
        term_puts(0, minimum of life_ymax and term_ymax()-1,
            "Press c to continue, q to quit.");

        char c;               //uninitialized variable
        use term_key to wait until the user has pressed a key;

        if (the key is not 'c') {
            break;
        }
    }

    term_destruct();
```

▲

2.6 Destructors

A class with a constructor and a destructor

We have seen several classes with a constructor. We now introduce the matching member function, the destructor. Our example will be a stack, first in C as a lot of variables and functions, and then in C++ as a class.

A *stack* is what a copyright lawyer would call an “information storage and retrieval system”. In a stack, the order in which the values are stored dictates the order in which they will be retrieved: last in, first out. Accountants would call it a LIFO list; the rest of us would say “last hired, first fired.”

The values stored and retrieved by our stack will be integers. Each value is stored by the function `push` and retrieved by `pop`. Retrieving a value removes it from the stack, so it can be retrieved only once.

The stack contains an array `a` big enough to hold 100 elements. We also have a variable `n` to hold the number of elements currently in the stack. Initially the stack is empty, so `n` is initialized to zero in line 7 of `stack.c`.

In both languages, the variables `a` and `n` will be accessible only to the functions `push` and `pop`. To accomplish this in C we declare the variables to be static, and define no functions other than `push` and `pop` in the same file as `a` and `n`. In other words, the source code has to be sliced into separate files to express which variables can be accessed by which functions.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackc/stack.h`

```
#ifndef STACKH   /* C example */
```

```
#define STACKH

void push(int i);
int pop(void);   /* C needs the keyword void */
#endif
```

Warning: the `%u` in lines 14 and 27 is not portable. `size_t` will not always be another name for `unsigend int`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackc/stack.c`

```
 1 #include <stdio.h>    /* C example */
 2 #include <stdlib.h>
 3 #include "stack.h"
 4 #define STACK_MAX_SIZE 100
 5
 6 static int a[STACK_MAX_SIZE];
 7 static size_t n = 0;  /* number of values in the stack; stack initially empty */
 8
 9 /* Push a value onto the stack. */
10
11 void push(int i)
12 {
13     if (n == STACK_MAX_SIZE) {   /* overflow */
14         fprintf(stderr, "Can't push when size %u == capacity %u.\n",
15             n, STACK_MAX_SIZE);
16         exit(EXIT_FAILURE);
17     }
18
19     a[n++] = i;
20 }
21
22 /* Pop a value off the stack. */
23
24 int pop(void)
25 {
26     if (n == 0) {                /* underflow */
27         fprintf(stderr, "Can't pop when size %u == 0.\n", n);
28         exit(EXIT_FAILURE);
29     }
30
31     return a[--n];
32 }
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackc/main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main()
{
    printf("To hire a person, type their social security number.\n"
        "To fire the most recently hired person, type a zero.\n"
        "To quit, type a negative number.\n");
```

```

    for (;;) {
        int ss;             /* uninitialized variable */
        scanf("%d", &ss);

        if (ss < 0) {       /* quit */
            break;
        }

        if (ss > 0) {       /* hire */
            push(ss);
        } else {            /* fire */
            printf("Firing number %d.\n", pop());
        }
    }

    return EXIT_SUCCESS;
}
```

```
To hire a person, type their social security number.
To fire the most recently hired person, type a zero.
To quit, type a negative number.
10                        You type the numbers in italics.
20
30
0
Firing number 30.
0
Firing number 20.
40
0
Firing number 40.
0
Firing number 10.
-1
```

Package the stack as a C++ class.

As above, the variables a and n will be accessible only to the functions push and pop. But in C++ we can express this in the language itself rather than by the crude device of slicing the source code into separate files. We let a and n be private data members of a class, and push and pop be member functions.

Of course, the code is still divided into separate files. But now this is for an entirely different reason: to reuse class stack in many C++ programs without having to copy and paste it into other files. We can simply add stack.h and stack.C to the list of files that constitute a program.

The stack_max_size in line 5 of stack.h is not a data member of class stack (yet). It merely floats somewhat unsatisfactorily near it. See the similar disposition of the global array date_length on pp. 114–115, and the global variables life_ymax and life_xmax on p. 145. Eventually it will be renamed max_size, the conventional C++ name for the maximum size of a data structure. A C++ data structure is called a *container*. The container classes vector, map, and string all have a max_size.

We already know that a constructor is the member function called at the start of an object's life, and the name of the constructor is the same as the name of the object's class. Incidentally, this is our first constructor that does not put a value into every data member of the newborn object. Since n is zero, there is no need for the constructor to put anything into the array a.

A *destructor* is the member function called at the end of an object's life. The name of the destructor is the name of the object's class with a tilde in front of it. Our constructor, defined in line 11, is inline; our destructor, declared in line 12, is not.

Until now, none of our classes has needed a destructor. If a class has no destructor, then, for the time being, nothing happens when an object of that class expires. But if the dying object requires any kind of cleanup or funeral, a postmortem examination, a puff of smoke, an aria, or if it needs to inform its neighbors of its demise, we can place these last rites in a destructor to ensure that they are never forgotten.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack/stack.h`

```
 1 #ifndef STACKH        //C++ example
 2 #define STACKH
 3 #include <cstddef>    //for size_t
 4 
 5 const size_t stack_max_size = 100;
 6 
 7 class stack {
 8     int a[stack_max_size];
 9     size_t n;           //number of values currently in the stack
10 public:
11     stack() {n = 0;}    //constructor: start with the stack empty
12     ~stack();           //destructor
13 
14     void push(int i);
15     int pop();          //C++ doesn't need the keyword void
16 };
17 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack/stack.C`

```
#include <iostream>
#include <cstdlib>
#include "stack.h"
using namespace std;

stack::~stack()   //destructor
{
    //cout << "destructor for stack\n";

    if (n != 0) {
        cerr << "Warning: stack still contains " << n << " value(s).\n";
    }
}

//Push a value onto the stack.

void stack::push(int i)
{
    if (n == stack_max_size) {   //overflow
        cerr << "Can't push when size " << n << " == capacity "
            << stack_max_size << ".\n";
        exit(EXIT_FAILURE);
    }

```

```
    a[n++] = i;
}

//Pop a value off the stack.

int stack::pop()
{
    if (n == 0) {                //underflow
        cerr << "Can’t pop when size " << n << " == 0.\n";
        exit(EXIT_FAILURE);
    }

    return a[--n];
}
```

An object’s destructor is always called when the object reaches the end of its lifespan. For example, the object `s` is local to the `main` function, so its life extends from its declaration in line 12 of `main.C` to the return from `main` in line 28. The constructor for `s` is called in line 12; the destructor is called in 28. If we delete the `return` in line 28, the destructor would be called when we return from `main` at the closing curly brace in line 29.

Our class `stack` belongs to no namespace. Another class `stack`, belonging to namespace `std`, is declared in the header file `<stack>`. We did not include this header directly, but it might have been included by one of the headers that we did include.

The double colon in line 12 ensures that the `stack` is the one that belongs to no namespace; the double colon is needed only if the header file `<stack>` was included (either directly or by another header file). Assuming that `<stack>` was included, a `std::stack` in line 12 would have been the class `stack` belonging to namespace `std`, and an unadorned `stack` would not have compiled.

Is there any down side to rewriting the code as a class? Well, the C function `push` took only one argument, but the member function `stack::push` takes two arguments, one explicit and one implicit. As long as there is only one stack, this is a disadvantage.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4 #include "stack.h"
 5
 6 int main()
 7 {
 8     cout << "To hire a person, type their social security number.\n"
 9         "To fire the most recently hired person, type a zero.\n"
10         "To quit, type a negative number.\n";
11
12     ::stack s;   //Call the constructor for s with no arguments.
13
14     for (;;) {
15         int ss;            //uninitialized variable
16         cin >> ss;
17         if (ss < 0) {      //quit
18             break;
19         }
20
21         if (ss > 0) {      //hire
```

```
22              s.push(ss);
23          } else {            //fire
24              cout << "Firing number " << s.pop() << ".\n";
25          }
26      }
27
28      return EXIT_SUCCESS;   //Call the destructor for s.
29  }
```

A C++ object should live no longer than it needs to. If we change the above lines 12–14 to

```
30      for (::stack s;;) {
```

the stack will be destructed as soon as we leave the loop.

▼ Homework 2.6a: add new member functions to class stack (analogous to Homework 2.4a)

Add three new member functions to class `stack`. All three have return values; none of them should produce any output. Do not add any data members to class `stack` in this homework.

To demonstrate that your new member functions work, hand in the output of the three programs `main12.C`, `main3.C`, and `main4.C` in the directory `http://i5.nyu.edu/~mm64/book/src/stack/test`.

(1) Add an inline member function declared as

```
bool empty() const;
```

that will return `true` if the stack is empty, `false` otherwise. The stack is empty if `n` is equal to zero. Since the value of a comparison operator such as == is `true` or `false`, you do not need to write an `if` or the operator `?:` in `empty`. `empty` must produce no output.

The destructor and the `pop` member function should now see if the stack is empty by calling `empty`. It is no sin for one member function to call another.

(2) Add an inline member function declared as

```
bool full() const;
```

that will return `true` if the `stack` is full, `false` otherwise. The stack is full if `n` is equal to `stack_max_size`. Since the value of a comparison operator such as == is `true` or `false`, you do not need to write an `if` or the operator `?:` in `full`. `full` must produce no output.

The `push` member function should now see if the stack is full by calling `full`.

(3) Add an inline member function declared as

```
size_t size() const;
```

that will return the current number of elements in the stack: the number of values that have been pushed but have not yet been popped. It will always return the value of `n`, even if it is greater than `stack_max_size`. This should never happen.

`size` must produce no output. The destructor and the `push` and `pop` member functions should display the return value of `size` as part of their error message.

▲

▼ Homework 2.6b: change a data member of class stack (analogous to Homework 2.4b)

To demonstrate that your class `stack` still works after you do this homework, hand in the output of the three programs `main12.C`, `main3.C`, and `main4.C` in the directory `http://i5.nyu.edu/~mm64/book/src/stack/test`.

Change the n data member of class stack from a size_t to a read/write pointer to an int, and rename it p. At any given moment, p will point to the "next free element" of a: the element with the smallest subscript that is not currently occupied by a pushed value. For example, the constructor for class stack will store the address of a[0] into p because no element of a is yet occupied by a pushed value. push will store a value into the array element to which p is pointing, and will then make p point to the next array element. pop will make p point to the previous array element, and will then return the value to which p is now pointing.

The only data members of the new class stack will be a and p. Do not create any global variables. Do not declare any local variable inside a function to be static. Do not remove the empty, full, and size member functions from class stack. empty, full, size, the constructor, and the destructor will continue to produce no output. push and pop will produce no output other than error messages.

Make whatever changes are necessary in the existing member functions of class stack. Since n and p are private, no changes will be needed in any function that is not a member of class stack.

We will also have to write our own copy constructor for class stack to accommodate the change from n to p. The following line 2 calls the copy constructor for class stack.

```
1     ::stack s1;                     //call the default constructor
2     ::stack s2 = s1;                //call the copy constructor
```

Since we have not written the copy constructor, the computer will behave as if we had written the following. It blindly copies the values from the data members of the other object (s1) into the data members of this object (s2). Line 9 is wrong: it leaves the p data member of this object pointing at a data member of the other object. (There is also a performance bug: the for loop usually will not need to copy the entire array.)

```
 3 stack::stack(const stack& another)
 4 {
 5     for (size_t i = 0; i < stack_max_size; ++i) {
 6         a[i] = another.a[i];
 7     }
 8
 9     p = another.p;   //bug
10 }
```

You will therefore have to write your own copy constructor for the new class stack.

When we have operator overloading, we will have to write one more member function for class stack to accommodate the change from n to p. See p. 311.

▲

The conventional name for the data type of each element

To keep the exposition simple, we wrote class stack in terms of data type int. But the C++ convention is to create a typedef named value_type for the data type of the values stored in a data structure. The members of class stack should therefore have been declared as follows.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/stack/stack2.h

```
#ifndef STACKH
#define STACKH
#include <cstddef>        //for size_t
using namespace std;

typedef int value_type;   //data type of each value contained in the stack
const size_t stack_max_size = 100;

class stack {
```

```
10      value_type a[stack_max_size];
11      size_t n;              //number of values currently in the stack
12  public:
13      stack() {n = 0;}
14      ~stack();
15
16      void push(value_type i);
17      value_type pop();
18
19      bool empty() const;
20      bool full() const;
21
22      size_t size() const;
23  };
24  #endif
```

A destructor is declared and defined like a constructor.

Compare the four constructor rules on pp. 133−134.

(1) A destructor's name is the same as that of its class with a leading tilde `~`, chosen because the tilde means "not" in C and C++. In the above C++ program, the class is named `stack` (line 7 of `stack.h`) and the class's destructor is named `~stack` (declared in line 12 of `stack.h`, defined in lines 6−13 of `stack.C`).

(2) Since a constructor can take arguments, a class can have more than one constructor thanks to function name overloading. But a destructor takes no arguments, so a class can never have more than one destructor.

(3) A destructor returns no value. But do not declare its return type to be `void`: just write nothing at all as in line 12 of `stack.h` and line 6 of `stack.C`.

(4) Do not declare a constructor or destructor to be `const` (as `print` was on pp. 115−116, Version 3, lines 29 and 99), even if they change the value of no data member and call no non-`const` member function of the same object.

(5) For the time being, a constructor and the destructor must always be `public`, not `private`. If the destructor for a class was `private`, it could be called only by the member functions of another object of that class. The other object could be destructed only by a third object. We would have an infinite regress (a "chicken and egg" situation). (A non-public destructor will appear on p. 1045.)

(6) Additional rules will be promulgated later. A destructor should never call `exit`; see p. 184. A destructor must be "virtual" if any other member function is; see pp. 493−494. An "exception" should never escape from a destructor if there is currently another exception at large; see pp. 614−616.

Don't write an explicit call to a destructor.

> [T]he aged star still continues meticulously to fulfill its part in the dance. . . . Finally its light is extinguished and its tissues disintegrate in death. Henceforth it continues to sweep through space, but it does so unconsciously, and in a manner repugnant to its still conscious fellows
>
> —Olaf Stapledon, *Star Maker* (1937), chapter XI, §3

After its destructor is called, an object is no more than a cadaver. By "cadaver", we mean the memory occupied by the destructed object. For example, the object `s` declared in line 8 is local to the `main` function, so it occupies in memory until the return from `main` in line 15. But the call to the destructor in line 13 turns it into a cadaver.

There is no guarantee that a cadaver's member functions will still work or that its data members will hold their values. Properly speaking, a cadaver has no members. Since `s` became a cadaver in line 13, the call to the member function in line 14 might fail.

To ensure that a cadaver is disposed of immediately, the memory occupied by an object must never outlive the object. In other words, we must ensure that every object stays alive until the last moment before its memory is deallocated. This means that we must never call a destructor explicitly. And there is no reason we would ever want to. Just before deallocating the memory occupied by an object, the destructor is always called automatically. There is no reason to call it ourselves.

But what if you really want to destruct the object at the above line 13? In that case, the object should have been allocated dynamically, not automatically. A dynamic object can be destructed at any point. See `new` and `delete` in Chapter 4.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack/cadaver.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "stack.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     ::stack s;                //Call the constructor.
 9     s.push(10);
10     s.push(20);
11     cout << s.pop() << "\n"; //This pop will work correctly.
12
13     s.~stack();              //Call the destructor explicitly.  Never do this.
14     cout << s.pop() << "\n"; //This pop may not work.
15     return EXIT_SUCCESS;     //Call the destructor implicitly; may not work.
16 }
```

Incidentally, the above program has another bug. Every C++ object should be constructed and destructed exactly once. But line 15 calls the destructor for `s`, even though it has already been called in line 13. And there's a third bug. The call to the destructor in line 15 might fail for the same reason as the call to `pop` in line 14.

On rare occasions, mostly related to the "placement" operator `new`, we will have to make an explicit call to a destructor. An example will be when we pass different arguments to the constructor for each object in a dynamically allocated array of objects. We'll see this on p. 406 when we do `new` and `delete`.

Class `stack` in the C++ Standard Library

A class `stack` has already been written for us in the C++ Standard Library. In fact, it is getting hard to think of simple, general classes that we would have to write for ourselves; most of them are already in the library. Here, for example, is the prewritten class `stack`.

A class whose name contains <angle brackets>, such as the class `stack<int>` in line 8, is called a *template class*. Although we do not yet know how to create a template class, we can easily use one: just plug the name of another data type into the angle brackets. For the standard library class `stack`, we plug in the name of the data type of the values to store and retrieve. The `stack` in line 8 will store and retrieve `int`'s.

The `top` function of the standard library `stack` in line 18 returns the most recently pushed element, but without removing it from the stack. To remove it, we must call the `pop` in line 19.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack/stdstack.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <stack>
4 using namespace std;
```

```
 5
 6 int main()
 7 {
 8     stack<int> s;   //Born empty because we gave no args to constructor.
 9
10     cout << "empty == " << s.empty() << ", size == " << s.size() << "\n";
11
12     s.push(10);
13     s.push(20);
14     s.push(30);
15
16     cout << "empty == " << s.empty() << ", size == " << s.size() << "\n";
17
18     cout << s.top() << "\n";
19     s.pop();   //returns void, unlike the stack::pop we wrote
20
21     cout << s.top() << "\n";
22     s.pop();
23
24     cout << s.top() << "\n";
25     s.pop();
26
27     cout << "empty == " << s.empty() << ", size == " << s.size() << "\n";
28     return EXIT_SUCCESS;
29 }
```

To print the `bool` return values as the words `true` or `false`, see p. 354.

```
empty == 1, size == 0       line 10; bool prints as 1 or 0
empty == 0, size == 3       line 16
30                          line 18
20                          line 21
10                          line 24
empty == 1, size == 0       line 27
```

Why was the functionality of our `pop` function split into two separate functions in the class `stack` in the standard library? The obvious reason is to allow us to peek at the top element of the `stack` without removing it. But even if every call to `top` is followed by a call to `pop`, there would still be a reason for the split.

Consider the following fragment, in which a value is popped from a stack of the class that we wrote.

```
    stack s;
    s.push(10);
    cout << s.pop() << "\n";
```

Our `pop` function returns an integer by value. But it could have gotten away with a return by reference, since it returns an element of an array that does not evaporate as we return.

Now imagine a more sophisticated stack, one that stores its elements into a dynamically expanding and contracting block of memory. Its `pop` function would have to return by value if it deallocated the memory for the most recently pushed element and then returned the element. Return by reference could not be used because the element's memory has been deallocated.

Such may be the case with the class `stack` in the standard library. Its `top` function can return the most recently pushed element by reference, because the element is still in memory. But its `pop` function can not return the element by reference, because the element is no longer there. Rather than returning each element by value from a call to `pop`, the standard library `stack` returns each element by reference from a call to `top`. But don't try to use the reference after the element to which it refers has been popped (lines 17−18, 22−23).

```
 4 #include <stack>
 5 using namespace std;
 6
 7     stack<int> s;                    //the standard library stack
 8     s.push(10);
 9     s.push(20);
10     s.push(30);
11
12     cout << s.top() << "\n";         //outputs 30
13     s.pop();
14
15     int *p = &s.top();
16     cout << *p << "\n";              //outputs 20
17     s.pop();
18     cout << *p << "\n";              //may no longer output 20
19
20     int& r = s.top();
21     cout << r << "\n";               //outputs 10
22     s.pop();
23     cout << r << "\n";               //may no longer output 10
```

See another example at p. 802.

All the values in a stack must be of the same data type. But because the standard library `stack` is a template class, each stack can hold a different type of value.

The stack of stacks in line 33 needs a space between the two >'s. See line 17 in ¶ (2b) on p. 101. Something we could push onto the stack of stacks is the stack in line 28.

```
24 #include <stack>
25 #include "date.h"
26 using namespace std;
27
28     stack<int> s1;                   //a stack of int's
29     stack<double> s2;                //a stack of double's
30     stack<int *> s3;                 //a stack of pointers
31     stack<date> s4;                  //a stack of objects
32
33     stack<stack<int> > s5;           //a stack of stacks
34     s5.push(s1);                     //Push the stack in line 28.
```

2.7 An Interface Class for the Terminal

▼ Homework 2.7a:

Another way to write on the screen

The major classes we have seen are `date`, `life`, and `stack`. We will introduce one more, class `terminal`, before talking about classes in general.

Instead of doing our special effects by calling the ten `term_` C functions in pp. 85−89, we will now construct an object of class `terminal` and call its ten member functions. Compare the following test program with the `main.C` back in pp. 87−88. It does exactly the same demo by calling a different set of functions. For each C function, there is now a member function that does the same job.

For convenience, we also introduce two member functions which have no counterparts among the `term_` functions. The function `next` in lines 18 and 23 takes a pair of coördinates, `x` and `y`, and advances them to the next location. It would change (0, 0) to (1, 0), one location to the right. And on a screen with 80 columns, it would change (79, 0) to (0, 1), the first location on the next line. Finally, with 24 rows it would change (79, 23) to (0, 24), one step below the bottom row of the screen, but would refuse to advance it any farther.

We asked you to make every reference argument read-only on pp. 72−74. The call to `next` in line 18 shows the danger of violating this rule. Although there's no way to see it by inspecting that line, `next` changes the values of `x` and `y`. We will clean this up when we introduce an "iterator" for class `terminal` on p. 966.

The function `in_range` in line 23 returns `true` if the pair of coördinates is on the screen. It was named after the `out_of_range` "exception" on pp. 622−623.

We need two variables, `x` and `y`, to loop across the screen in line 23. When we have iterators we will be able to do the loop with only one, even though the screen is two-dimensional. In anticipation of that day, we changed the pair of nested loops in the original test program (lines 23−24 of `main.C` on p. 87) to the single loop in line 23. This change is premature: if there are two variables, there should be two loops. But try to think of the `x` and `y` as a single object with two data members. In time they will be.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/terminal/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "terminal.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const terminal term('.'); //The constructor for term calls term_construct.
 9
10     const unsigned xmax = term.xmax();
11     const unsigned ymax = term.ymax();
12
13     unsigned x = xmax / 2;       //center of screen
14     unsigned y = ymax / 2;
15
16     term.put(x, y, 'X');
17     char c = term.get(x, y);
18     term.next(x, y);
19     term.put(x, y, c);
20
21     term.put(0, 0, "Please type printable characters ending with a q.");
22
23     for (x = 0, y = 1; term.in_range(x, y); term.next(x, y)) {
24         while ((c = term.key()) == '\0') {
25         }
26
27         if (c == 'q') {          //quit
28             break;
29         }
30
31         term.put(x, y, c);
32     }
33
34     term.wait(1000);
```

```
    term.beep();
    return EXIT_SUCCESS;      //The destructor for term calls term_destruct.
}
```

An interface class

Let's read the definition for class `terminal`, starting with the simplest member function.

The `beep` function in line 29 is merely a call-through (p. 95): it simply calls the corresponding C function `term_beep`. Other examples are in lines 27–29. Since this class does almost no work on its own, we call it an *interface class*. It merely delivers the results of another piece of software, the ten `term_` functions.

Now let's look at the data members. The constructor takes a `char` argument and stores it in the data member `_background` in line 9 of `terminal.C`. It has an underscore because a class can't have a data member and a member function with the same name. To keep the name of the public member short and simple, the burden of the underscore is placed on the private member. (See p. 241 for another example.) The constructor also initializes the screen in line 11 of `terminal.C`, and then stores the dimensions of the screen into the other two data members `_xmax` and `_ymax` in lines 13 and 14.

The member functions `background`, `xmax`, and `ymax` in lines 18–20 of `terminal.h` grant the public read-only access to the private data members. (See p. 242 for another example.)

The two-argument `put` in line 23 of `terminal.h` passes the `_background` data member to the three-argument `put` in line 22. This has the effect of letting `_background` be the default value for the third argument. I wish we could combine the two functions into one with a default value for its third argument:

```
    void put(unsigned x, unsigned y, char c = _background) const;
```

But the language just does not let us do this. A data member of an object can be mentioned inside the *body* of a member function of the same object; it cannot be mentioned inside the *argument list* of a member function of the same object.

The `in_range` in line 31 has no need to check if `x` and `y` are negative. They never can be, because they are unsigned.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/terminal/terminal.h`

```
1 #ifndef TERMINALH
2 #define TERMINALH
3
4 extern "C" {
5 #include "term.h"
6 }
7
8 class terminal {
9     char _background;   //default value for third argument of put
10    unsigned _xmax;     //number of columns of characters
11    unsigned _ymax;     //number of rows of characters
12
13    void check(unsigned x, unsigned y) const;
14 public:
15    terminal(char initial_background = ' ');
16    ~terminal();
17
18    char background() const {return _background;}
19    unsigned   xmax() const {return _xmax;}
20    unsigned   ymax() const {return _ymax;}
```

```

    void put(unsigned x, unsigned y, char c) const;
    void put(unsigned x, unsigned y) const {put(x, y, _background);}
    void put(unsigned x, unsigned y, const char *s) const;
    char get(unsigned x, unsigned y) const {check(x, y); return term_get(x, y);}

    char key() const {return term_key();}
    void wait(int milliseconds) const {term_wait(milliseconds);}
    void beep() const {term_beep();}

    bool in_range(unsigned x, unsigned y) const {return x < _xmax && y < _ymax;}
    void next(unsigned& x, unsigned& y) const;
};
#endif
```

Every character is ultimately put on the screen by the three-argument `put` in line 36, which calls the C Standard Library function `isprint` to check that the character is printable. If it is not, we cast the character to print its ASCII code. See line 14 of `static_cast.C` on p. 65.

The `initial_background` argument of the constructor is checked when line 19 fills the screen with the background character.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/terminal/terminal.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cctype>   //for isprint
 4 #include "terminal.h"
 5 using namespace std;
 6
 7 terminal::terminal(char initial_background)
 8 {
 9     _background = initial_background;
10
11     term_construct();
12
13     _xmax = term_xmax();
14     _ymax = term_ymax();
15
16     if (_background != ' ') {
17         for (unsigned y = 0; y < _ymax; ++y) {
18             for (unsigned x = 0; x < _xmax; ++x) {
19                 put(x, y);
20             }
21         }
22     }
23 }
24
25 terminal::~terminal()
26 {
27     for (unsigned y = 0; y < _ymax; ++y) {
28         for (unsigned x = 0; x < _xmax; ++x) {
29             put(x, y, ' ');
30         }
31     }
32
```

```
    term_destruct();
}

void terminal::put(unsigned x, unsigned y, char c) const
{
    if (isprint(static_cast<unsigned char>(c)) == 0) {
        cerr << "unprintable character "
            << static_cast<unsigned>(static_cast<unsigned char>(c))
            << ".\n";
        exit(EXIT_FAILURE);
    }

    check(x, y);
    term_put(x, y, c);
}

void terminal::put(unsigned x, unsigned y, const char *s) const
{
    for (; *s != '\0'; ++s) {
        put(x, y, *s);
        next(x, y);
    }
}

//Move to the next (x, y) position: left to right, top to bottom.
//Warning: will change the values of the arguments.

void terminal::next(unsigned& x, unsigned& y) const
{
    check(x, y);

    if (++x >= _xmax) {
        x = 0;
        if (++y >= _ymax) {
            cerr << "can't go to or beyond row " << _ymax << "\n";
            exit(EXIT_FAILURE);
        }
    }
}

void terminal::check(unsigned x, unsigned y) const
{
    if (!in_range(x, y)) {
        cerr << "coordinates (" << x << ", " << y
            << ") must be >= (0, 0) and < ("
            << _xmax << ", " << _ymax << ")\n";
        exit(EXIT_FAILURE);
    }
}
```

List of the five source files that constitute the test program

(1) `term.h` and `term.c` (pp. 85–89). `term.c` is written in the language C; `term.h` is acceptable to both languages. The remaning files are in C++.

(2) `terminal.h` and `terminal.C` (pp. 159–161)

(3) `main.C` (pp. 157–159)

Compile the test under Unix

```
1$ gcc -I. -DUNIX= -c term.c          minus uppercase I
2$ ls -l term.o                       minus lowercase L

3$ g++ -I. -o ~/bin/tester main.C terminal.C term.o -lcurses
4$ ls -l ~/bin/tester
5$ tester                             Run it.
```

It's just as fast to call the member functions of class terminal.

Instead of calling the C functions directly, we are now calling them through the member functions of a `terminal` object. In a moment we will see the benefits of this extra layer of software. But first we must consider if the extra layer has slowed the program down.

When we write a call to an inline function, the computer behaves as if we had written the body of the inline function in place of the call. When we write line 2, for example, the computer behaves as if we had written line 3. Calling the member function `beep` in line 2 is therefore just as fast as calling the C function `term_beep` in line 3.

```
1   const terminal term('.');
2   term.beep();              //When we write this,
3   term_beep();              //the computer behaves as if we had written this.
```

Sometimes the member functions of class `terminal` are even faster. When we write line 5, the computer behaves as if we had written line 6. But line 6 calls no function; it simply uses the value of a data member. Calling the member function `xmax` in line 5 is therefore faster than calling the C function `term_xmax` in line 7.

```
4   const terminal term('.');
5   unsigned x = term.xmax();       //No function is called.
6   unsigned x = term._xmax;
7   unsigned x = term_xmax();       //A function is called.
```

Why bother with an interface class?

> The B words were in all cases compound words.
>
> —George Orwell, *1984*, Appendix: The Principals of Newspeak

Class `terminal` does not slow down the program, and in a few cases it makes it faster. But the real reason we introduced this extra layer is for æsthetics. Here is how calling the member functions of an object is more convenient than calling naked C functions.

(1) The C function names had to be compound words because we might have several devices to manipulate. If there are n devices and m functions for each device, the number of different function names will be $n \times m$.

```
term_beep();              /* C: number of names increases geometrically */
modem_beep();
pager_beep();
```

But the C++ member function names can be shorter because the member functions belong to an object. If there are n devices and m functions for each device, the number of different names will be only $n + m$.

```
term.beep();              //C++: number of names increases arithmetically
modem.beep();
pager.beep();
```

Our first example of shortening the names was on pp. 110–111.

(2) The C++ member functions also have fewer and simpler names thanks to function name overloading.

```
 7     /* C: every function must have a different name. */
 8     term_put(x, y, c);        /* display a character */
 9     term_puts(x, y, s);       /* display a string */

10     //C++: can use same name for similar functions
11     term.put(x, y, c);        //display a character
12     term.put(x, y, s);        //display a string
```

(3) The most frequently used value for an argument can be made the default in C++. For example, the most frequently displayed character is the background character.

```
13     term_put(x, y, '.');      /* C */
14     term.put(x, y);           //C++: display term's background character
```

(4) Instead of two widely separated function calls

```
15     term_construct();
16     //the whole game
17     term_destruct();
```

we now write only a single declaration:

```
18     terminal term('.');       //This declaration calls the constructor.
19     //the whole game
20     return from main;         //The return from main calls the destructor.
```

If the call to `term_construct` in the above line 15 was missing in C, a call to `term_put` at line 16 would still compile but would execute incorrectly. But if the declaration for `term` in the above line 18 was missing in C++, a call to `term.put` at line 19 would not even compile. This is better than executing incorrectly.

(5) Packaging the ten C functions as a class would also make it easier to have a program with more than one terminal. As we will see, this is one of the main reasons for making a class.

▲

2.8 What are Objects For?

I wish I could give you a single, overarching statement about what an object is, or what an object is for, or how to recognize when an object should be used. The best I could do was to come up with four general ways of thinking about objects.

(1) An object is a `struct` with better security:
   - (a) Thanks to its constructor, we can't put garbage into a newborn object.
   - (b) If garbage appears later in the private data members, it's easy to round up all the possible suspects. Only the member functions of the object's class could have put the garbage there.

(2) An object can trigger a pair of events; multiple objects can trigger nested pairs. More radically, an object *is* the pair of events, or the interval between the events.

(3) An object is something that we might want to make more than one of.

(4) An object is a group of one or more variables whose values are used and changed by a series of function calls, and which persist until the last call of the series.

Let's take them one at a time. Later, there will be two more (p. 473 and pp. 734–735).

2.8.1 A Structure with Better Security

In C we wrote a structure with fields, and an array and function floating somewhat unsatisfactorily nearby. There is no connection between the structure and the floaters, except for the `date_` prefix on their names.

```
1 struct date {
2     int year;
3     int month;
4     int day;
5 };
6
7 const int date_length[] = /* etc. */
8 void date_print(const date *p);
```

In C++ we write a class with data members and member functions. The floating function in the above line 8 is now an integral part of the class in the following line 16. To ensure that the data members are initialized to legal values, there is also a constructor in line 15. No other functions in any program can read or write the private data members of a `date` object.

The floating array in the above line 7 will also become a member of the class in line 10. We will do this on pp. 238–239 when we have "static" data members. Now that the floaters are members, their names have been shortened.

```
 9 class date {
10     static const int length[];
11     int year;
12     int month;
13     int day;
14 public:
15     date(int initial_year, int initial_month, int initial_day);
16     void print() const;
17 };
```

2.8.2 A trigger for a Pair of Events.

The mere existence of an object triggers a pair of events: a call to one of its constructors and a call to its destructor. Here are examples of what a constructor/destructor pair can do.

(1) Error checking. The constructor installs a legal initial value into a newborn object; the destructor certifies that the value at death is still legal. A destructor can print a warning if the object is unhealthy at the end of its life or has not been properly drained (class `stack`, pp. 149–154; also pp. 923, 295–299.)

(2) Make something unforgettable. To ensure that an event will happen at some future time, we can construct an object whose destructor performs the event. For example, we might have to print a message whenever we are about to return from a function. If the function has many `return` statements, we simply construct an (automatically allocated) object upon entry to the function. When any `return` is executed, the object's destructor will be called and the message will be printed.

(3) Bookkeeping. Imagine a program that constructs and destructs many objects of the same class. To keep count of how many objects exist at any given moment, we can define the following variable (a global, not a data member):

```
int count = 0;
```

Every constructor for the class will say

```
++count;
```

and the destructor will say

```
--count;
```

The count will then be maintained automatically.

To keep all the objects of a class on a linked list, every constructor can insert the object into the list and the destructor can remove it. To keep all objects visible on the screen while they exist, every constructor can draw the object and the destructor can erase it. Et cetera.

(4) Resource management. Many resources may have to be allocated at an object's birth and deallocated at the object's death: memory, files, locks, windows, network connections, etc. Every constructor for the object's class can allocate the resources, and the destructor can deallocate them.

Events come in pairs

In the world of computer programming, events happen in pairs. Not all of them, of course; but when they must be paired, failure to do so usually results in disaster.

Each event often centers around a call to a function. In C, we might call the two functions in the wrong order, or forget to call one of them, or call one of them more than once. In C++, the first function can be called by an object's constructor and the second by the object's destructor. We can trigger the correct pair of calls simply by creating an object and letting it live out its life.

Generally the first event creates one or more variables that must be saved for later use. Often they lapse into irrelevance after the second event. If so, there is now a natural place to store these values: in the object's data members. It is therefore common to have a C++ class with only one data member, while it would be quite unusual to have a C structure with only one field. (A C++ class can even have no members at all; see pp. 590 and 842.)

The first two examples of pairs of events are taken from the C Standard Library. In C++, they could be packaged as objects.

(1) The functions `malloc` and `free` allocate and free dynamic memory. (C++ will replace them with `new` and `delete`.)

The first event is the allocation in lines 6–10, which creates the variable `p`. (The `perror` in line 8 is the error-printing function from the C Standard Library.) The second event is the deallocation in 15, after which the value of `p` is useless. (The `free` in line 15 returns `void`, so we can't test it for failure.) Between them, `p` is used by the series of function calls in lines 12–13.

```
 1 /* C example */
 2 #include <stdio.h>         /* for perror and printf */
 3 #include <stdlib.h>        /* for malloc, free, exit, EXIT_FAILURE, and NULL */
 4 #include <string.h>        /* for strcpy */
 5
 6     char *const p = malloc(6);
 7     if (p == NULL) {
 8         perror(argv[0]);
 9         exit(EXIT_FAILURE);
10     }
11
12     strcpy(p, "hello");
13     printf("%s\n", p);
14
15     free(p);
```

(2) `fopen` and `fclose` open and close a file. C++ will use the constructor and destructor for classes `ofstream` or `ifstream`.

The first event is the opening in lines 20–24, which creates the variable `fp`. The second event is the closing in 29–32, after which the value of `fp` is useless. Between them, `fp` is used by the series of function calls in lines 26–27.

```
16 /* C example */
17 #include <stdio.h>         /* for fopen, NULL, fprintf, fflush, fclose, perror */
18 #include <stdlib.h>        /* for exit and EXIT_FAILURE */
19
20     FILE *const fp = fopen("outfile", "w");
21     if (fp == NULL) {
22         perror(argv[0]);
23         exit(EXIT_FAILURE);
24     }
25
26     fprintf(fp, "hello\n");
27     fprintf(fp, "goodbye\n");
28
29     if (fclose(fp) != 0) {
30         perror(argv[0]);
31         exit(EXIT_FAILURE);
32     }
```

In each case, the entire scenario could be packaged as a C++ class. The first event could be the constructor; the second event, the destructor. The data that exists from the first event to the second could reside in the data members. The intervening functions that use the data could be member functions.

Here are other pairs of events suitable for this treatment.

(3) Create and destroy a file or directory.

(4) Compress and decompress a file.

(5) Encrypt and decrypt a string.

(6) Copy a group of files into or out of an archive file such as a `tar` archive.

(7) Lock and unlock a record in a file or database.

(8) Pop up a menu and make it disappear.

(9) Open and close a network connection.

(10) An obscure example for the C Standard Library: `va_start` and `va_end`.

When should we end one function and start another?

The local objects declared in a function will be destructed when we return from the function. For example, the local variable `ost` constructed in line 6 will be destructed when we reach the the closing curly brace at the end of the function in line 10. (As in C, however, variables that are static follow a different rule.)

The amount of code in a C++ function is often determined by how long we want its local objects to stay alive. When it is time for them to be destructed, it is time to cap off the function with a `}`. In fact, we often think of a function as primarily a framework over which we stretch the lifespan of the local objects.

For example, here is a function that writes to a file. We will see that the two-event scenario in the above lines 20–32 can be performed by constructing an object of class `ofstream` and letting it live out its life. The constructor for this object opens an output file, the destructor closes it, and in between, this object can write to the file with the same << syntax used by `cout` and `cerr`. The output file stays open for the lifespan of this object. The function `write_to_file` keeps the object alive for as long as we want to keep the file open.

```
1 #include <fstream>      //for ofstream
2 using namespace std;
3
4 void write_to_file()
5 {
6     ofstream ost("outfile");   //The constructor opens an output file.
```

```

    ost << "hello\n";          //Write to the file.
    ost << "goodbye\n";
}                              //The destructor closes the file.
```

There is a way to destruct an object in the middle of a function, without returning from the function. It would require the C++ equivalents of `malloc` and `free`. Let's not think about this yet.

An example of program reorganization

Here is the outline of a video game in C. We initialize the screen and keyboard before the game begins, and restore them to their prior state when the game ends. Initializing the screen consists of creating a graphics window or putting the screen into graphics mode. Initializing the keyboard will give the program access to each keystroke as it is typed, without having to wait for the RETURN key. In other words, it makes the keyboard live.

```
 1 /* C example */
 2
 3 int main()
 4 {
 5     initialize the screen (i.e., create a graphics window);
 6     initialize the keyboard (i.e., so you don't have to press RETURN);
 7     play_game();
 8     restore the keyboard to the way it was before;
 9     restore the screen to the way it was before;
10
11     return EXIT_SUCCESS;
12 }
```

When I learned structured programming (circa 1980), they taught us to reorganize a program by putting all the initialization code into one big subroutine, and all the restoration code into another.

```
13 int main()
14 {
15     initialize();
16     play_game();
17     restore();
18
19     return EXIT_SUCCESS;
20 }
21
22 void initialize(void)
23 {
24     initialize the screen;
25     initialize the keyboard;
26 }
27
28 void restore(void)
29 {
30     restore the keyboard to the way it was before;
31     restore the screen to the way it was before;
32 }
```

But the chunks of code for initialization in the above lines 24 and 25 probably have nothing in common: no shared variables, constants, typedef's, etc. The only way they could communicate with each other would be through global variables. Ditto for the restoration chunks in lines 30 and 31.

Is there a better pairing for these bedfellows? The ones that belong together are the functions in 24 and 31: they probably share more variables than 24 and 25. In fact, line 24 probably stores the values that are used to restore the screen in line 31. Similarly, the keyboard functions in 25 and 30 belong together.

We will pair them by making them constructors and destructors. They will communicate via the data members of the object to which they belong. Incidentally, this will halve the number of function names we have to invent: a constructor and destructor share the same name, with a tilde. See pp. 133, 154. The `main` function will now contain declarations for two variables (lines 49–50) instead of two pairs of widely separated function calls (in the above lines 5 and 9, 6 and 8).

```
33 class screen {              //C++ example
34     declare variables shared by constructor and destructor here
35 public:
36     screen();               //constructor: initialize the screen
37     ~screen();              //destructor: restore the screen
38 };
39
40 class keyboard {
41     declare variables shared by constructor and destructor here
42 public:
43     keyboard();             //constructor: initialize the keyboard
44     ~keyboard();            //destructor: restore the keyboard
45 };
46
47 int main()
48 {
49     screen s;               //construct (i.e., initialize) the screen
50     keyboard k;             //construct (i.e., initialize) the keyboard
51
52     play_game(s, k);
53
54     return EXIT_SUCCESS;    //destruct (i.e., restore) keyboard & screen,
55                             //in that order
56 }
```

To add extra screens and keyboards, we can simply declare extra objects. Local objects are destructed in the opposite order from that in which they were constructed: last hired, first fired.

```
57 int main()
58 {
59     screen s1;
60     keyboard k1;
61
62     screen s2;
63     keyboard k2;
64
65     play_game(s1, k1, s2, k2);
66
67     return EXIT_SUCCESS;    //destruct k2, s2, k1, and s1, in that order
68 }
```

Many operations must be undone in reverse order

Many operations have to be done in pairs. Furthermore, the pairs must often be nested. Our first example creates a window and puts icons into it, does some work, and then destroys (or hides) the icons and window. We must destroy the icons *before* we destroy the window. If we destroyed the window first, the icons would (momentarily) land on the desktop. The nested indentation shows how the pairs of events nest.

Create a window.
    Create icons in the window.
        Do the work.
    Destroy the icons.
Destroy the window.

A second example creates a directory and puts files into it, does some work, and then destroys the files and directory.

Create a directory (or "folder").
    Create files (or "documents") in the directory.
        Do the work.
    Remove the files.
Remove the directory.

A third example copies many individual files into a `.tar` archive, compresses it, does some work, and then decompresses the archive and extracts the original files from it.

Create a `.tar` file, and copy many individual files into it.
    Compress the `.tar` file into a `.tar.gz` file.
        Do the work.
    Decompress the `.tar.gz` file back into a `.tar` file.
Extract the individual files from the `.tar` file, and remove the `.tar` file.

To nest these events, we can simply declare objects whose lifespans are nested. For example, imagine a `tempdir` object whose constructor and destructor create and destroy a temporary directory, and a `tempfile` object whose constructor and destructor create and destroy a temporary file. The constructor for `tempfile` takes an argument giving the directory in which to create the file. To create and destroy the directory and files in the correct order, we need only declare the `tempdir` before the `tempfile`'s and then let them live out their lives. See pp. 180–181.

```
void f()
{
    tempdir td;                     //Create a directory.

    tempfile f1(td);                //Create files in the directory.
    tempfile f2(td);
    tempfile f3(td);

    do the work;
}   //Destroy the files f3, f2, f1; then destroy the directory td.
```

The above pairings were temporal; the following pairings are spatial. In a table in HTML ("Hypertext Markup Language" on the web), each box is surrounded by a pair of `TD` tags ("table data"). The opening and closing tags of each pair have the same name, but closing tag also has a diagonal slash. Similarly, each row of boxes is surrounded by a pair of `TR` tags ("table row"), and the entire table is surrounded by a pair of `TABLE` tags.

A web browser converts the pairings from spatial to temporal by reading the table from top to bottom. Whenever it encounters the opening tag of each pair, it can construct an object for that pair. When it encounters a closing tag, it can destruct the object for that pair. If there is no such object, we must have encountered a closing tag with no prior matching opening tag. If we pass the end of the table and there are leftover objects, we must have encountered an opening tag with no subsequent matching closing tag.

row 1, col 1	row 1, col 2
row 2, col 1	row 2, col 2

```
<TABLE BORDER>
    <TR>
        <TD>
            row 1, col 1
        </TD>
        <TD>
            row 1, col 2
        </TD>
    </TR>
    <TR>
        <TD>
            row 2, col 1
        </TD>
        <TD>
            row 2, col 2
        </TD>
    </TR>
</TABLE>
```

2.8.3 Something to Make More Than One Of.

It would be hard to modify the answer to Homework 1.5a (pp. 42–44) to run more than one game simultaneously. But the answer to Homework 2.5b (pp. 144–147) would require no modification:

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/life/main2.C`

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "life.h"
using namespace std;

int main()
{
    const life_matrix_t glider_matrix = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 0, 0, 0, 0, 0, 0},
        {0, 1, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };
    life glider = glider_matrix;

    const life_matrix_t blinker_matrix = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };
    life blinker = blinker_matrix;

    for (;;) {
        glider.print();
        cout << "\n";
        blinker.print();

        cout << glider.generation()
            << ": Press c RETURN to continue, q RETURN to quit.\n";
        char buffer[256];
        cin >> buffer;
        if (strcmp(buffer, "c") != 0) {
            break;
        }

        glider.next();
        blinker.next();
    }

    return EXIT_SUCCESS;
}
```

I printed the pictures side-by-side to save paper. They actually appear one above another.

```
..........  ..........
.X........  ..X.......
..XX......  ..X.......
.XX.......  ..X.......
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
0: Press c RETURN to continue, q RETURN to quit: c

..........  ..........
..X.......  ..........
...X......  .XXX......
.XXX......  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
1: Press c RETURN to continue, q RETURN to quit: c
```

```
..........  ..........
..........  ..X.......
.X.X......  ..X.......
..XX......  ..X.......
..X.......  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
2: Press c RETURN to continue, q RETURN to quit: c

..........  ..........
..........  ..........
...X......  .XXX......
.X.X......  ..........
..XX......  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
3: Press c RETURN to continue, q RETURN to quit: c

..........  ..........
..........  ..X.......
..X.......  ..X.......
...XX.....  ..X.......
..XX......  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
..........  ..........
4: Press c RETURN to continue, q RETURN to quit: c
```

There's no good way to do it without objects.

Our C stack used a crude method to make the variables `a` and `n` accessible only to `push` and `pop`: we sliced the C program into `.c` files and barricaded the variables and functions together in a separate file. In C++, we can protect `a` and `n` by using the language instead of a pair of scissors. We simply let them be private members of class `stack`.

Suppose we needed three stacks, each with its `a` and `n` accessible to no one else. There is no good way to do this in C. One bad way is to make three `.c` files (and three `.h` files, not shown). Error checking omitted for brevity.

```
/* This file is stack0.c (C example). */
#include <stddef.h>     //for size_t

static int a[STACK_MAX_SIZE];
static size_t n = 0;

void push0(int i) {a[n++] = i;}
int pop0(void) {return a[--n];}
```



```
/* This file is stack1.c (C example). */
#include <stddef.h>

static int a[STACK_MAX_SIZE];
static size_t n = 0;

void push1(int i) {a[n++] = i;}
int pop1(void) {return a[--n];}
```

```
/* This file is stack2.c (C example). */
#include <stddef.h>

static int a[STACK_MAX_SIZE];
static size_t n = 0;

void push2(int i) {a[n++] = i;}
int pop2(void) {return a[--n];}
```

A more sophisticated bad way to have three stacks would be to add an argument to `push` and `pop` specifying which stack we want to use. For example,

```
    /* C example */
    int i;

    push(1, 10);                    /* Push 10 onto stack number 1. */
    push(2, 20);                    /* Push 20 onto stack number 2. */
    i = pop(1);                     /* Pop stack number 1. */
```

But implementing this will make the code more than twice as complicated. (Error checking omitted for brevity.)

```
/* This file is stack.c (C example). */
static int a[3][STACK_MAX_SIZE];
static size_t n[3] = {0, 0, 0};

void push(size_t which_stack, int i)
{
    /* More than twice as complicated as line 25 of stack.C on p. 151. */
    a[which_stack][n[which_stack]++] = i;
}

int pop(size_t which_stack)
{
    /* More than twice as complicated as line 37 of stack.C on p. 151. */
    return a[which_stack][--n[which_stack]];
}
```

Even with all this work, we can't create and destroy stacks as the program runs, or make a stack accessible to only one function.

In C++ we can simply declare three objects of our class `::stack`. They will last as long as the call to function `f`, and will be accessible to only that one function.

```
void f()
{
    ::stack s0;             //local to the function f
    ::stack s1;
    ::stack s2;
```

```

    s1.push(10);
    s2.push(20);
    int i = s1.pop();
}                            //Destruct the stacks in the order s2, s1, s0.
```

Eventually we will make an array of objects.

```
    ::stack a[3];            //Call the constructor for class stack 3 times.

    a[1].push(10);           //Push 10 onto stack a[1].
    a[2].push(20);
    int i = a[1].pop();
```

Another example of something we might want to make more than one of

Here's an example from K&R, pp. 46–47. Let's assume that an `unsigned long` is four bytes, with the bits numbered from right to left starting at bit 0. The most significant bit is number 31.

Line 6 scrambles the value of `next`. The first time it is executed, it will change the value of `next` to 1,103,527,590. Mathematicians have determined that the most random part of the resulting value consists of bits 30 through 16 inclusive. Here is 1,103,527,590 in binary with these bits underlined:

`01000001110001100111111010100110`

In line 7, the `/ 65536` chops off the bottom 16 bits of `next` and the `(unsigned)` and the `% 32768` chop off the top bit. What remains is bits 16 through 30. The first time it is executed, line 7 will returns 16,838, which is the number we underlined:

`100000111000110`

(Where did the mysterious number 1,103,515,245 = $3^5 \times 5 \times 7 \times 129{,}749$ come from? Look up "Linear Congruential Sequences" in Donald Knuth's *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.*)

```
 1 /* return pseudo-random integer in 0..32767 (C example) */
 2
 3 int rand(void)
 4 {
 5     static unsigned long next = 1;
 6     next = next * 1103515245 + 12345;
 7     return (unsigned)(next / 65536) % 32768;
 8 }
```

Unfortunately, every program that calls the above function always receives exactly the same series of random numbers, starting with 16838, 5758, 10113, etc. The example in K&R therefore lets us give an initial value of our own choosing (the *seed*) to the variable `next`. For example, the initial value 2014 gives us a different series of random numbers starting with 30237, 3862, 1078, etc. The variable `next` had to be made global in line 11 to be visible to both functions:

```
 9 /* C example */
10
11 static unsigned long next = 1;
12
13 void srand(unsigned seed)          /* set seed for rand() */
14 {
15     next = seed;
16 }
17
```

```
int rand(void)                  /* return pseudo-random integer in 0..32767 */
{
    next = next * 1103515245 + 12345;
    return (unsigned)(next / 65536) % 32768;
}
```

To get two series of random numbers, with the seeds 1 and 2014, we can easily generate them one after the other:

```
    /* C example */
    int i;

    /* Start with the default seed. */
    for (i = 0; i < 3; ++i) {
        printf("%d\n", rand());
    }

    /* Start with the seed 2014. */
    srand(2014);
    for (i = 0; i < 3; ++i) {
        printf("%d\n", rand());
    }
```

The output is

16838	*series that started with the seed 1*
5758	
10113	
30237	*series that started with the seed 2014*
3862	
1078	

But to interlace the two series, we would first have to precompute them and store them in two arrays:

```
    /* C example */
    int i;
    int r1[3];
    int r2[3];

    for (i = 0; i < 3; ++i) {
        r1[i] = rand();
    }

    srand(2014);
    for (i = 0; i < 3; ++i) {
        r2[i] = rand();
    }

    for (i = 0; i < 3; ++i) {
        printf("%d\t%d\n", r1[i], r2[i]);
    }
```

To right-justify the columns of numbers, see p. 353.

16838	30237	*first number in each series*
5758	3862	*second number in each series*
10113	1078	*third number in each series*

The inline constructor in line 9 has a default value for its argument.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/myrandom/myrandom.h

```
 1 #ifndef MYRANDOMH
 2 #define MYRANDOMH
 3
 4 //An object of this class generates a series of random numbers.
 5
 6 class myrandom {
 7     unsigned long next;
 8 public:
 9     myrandom(unsigned initial_next = 1) {next = initial_next;}
10     int rand();
11 };
12 #endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/myrandom/myrandom.C

```
#include "myrandom.h"

int myrandom::rand()
{
    //As in C, but next is now a data member instead of a static variable.
    next = next * 1103515245 + 12345;
    return static_cast<unsigned>(next / 65536) % 32768;
}
```

Instead of the above lines 36–52, we can now output the two interlaced series without the arrays:

—On the Web at
http://i5.nyu.edu/~mm64/book/src/myrandom/main.C

```
#include <iostream>
#include <cstdlib>
#include "myrandom.h"
using namespace std;

int main()
{
    myrandom r1;
    myrandom r2(2014);

    for (int i = 0; i < 3; ++i) {
        cout << r1.rand() << "\t"
             << r2.rand() << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
16838    30237        first number in each series
5758     3862         second number in each series
10113    1078         third number in each series
```

2.8.4 A Set of Variables Used by a Series of Function Calls

Often we notice that the same variable, or the same group of variables, is used by a series of function calls. Perhaps we are calling different functions; or perhaps the same function over and over. The functions might change the values of the variables, but the variables survive from one call to the next.

Here are three of the ways this can happen.

(1) The variable is passed as an argument to two or more function calls. For example, on p. 165, ¶ (1), the variable `p` was passed to `strcpy` and `printf`. And on pp. 165−166, ¶ (2), the variable `fp` was passed to a series of calls to `fprintf`.

(2) The variable is a static local variable in a function that is called two or more times. For example, on p. 174, line 5, the variable `next` will be used by each call to the function `rand`. Since it is static, the variable retains its value from one call to the next.

(3) The variable is a global variable that can be used by several functions. For example, on p. 174, line 11, the variable `next` can be used by the functions `srand` in lines 13−16 and `rand` in 18−22. As in the previous paragraph, the variable retains its value from one call to the next.

When a group of variables is used by a series of function calls, the variables and functions should become the data members and member functions of an object. A good candidate for objecthood was the pair of variables `x` and `y`, and the functions `term_put` and `term_get` to which they were passed, in the `term_` function test program `main.C` on pp. 87−88. A another candidate was the pair `x` and `y`, and the functions `put`, `get`, `next`, and `in_range` to which they were passed, in the terminal test program `main.C` on pp. 157−159. These functions are currently members of class `terminal`; later, they will become members of an object whose data members are the `x` and `y`. Such an object, which keeps track of our location in a data structure, will be called an *iterator.*

A third candidate for objecthood will be the trio of points `A`, `B`, and `C`, and the functions `area` and `contains` to which they will be passed, in `main.C` on pp. 208−209. We will pull them together into an object of a class named `triangle`. A fourth candidate will be on p. 727.

2.8.5 Other Uses of Objects

Certain kinds of classes occur so frequently that it is worthwhile to have names for them.

Container classes

An object of a *container class* contains other objects, pointers thereto, or at least values of a built-in data type. Our first example, the class `stack` on pp. 149−154, was hardwired to store and retrieve only integers. The more sophisticated container classes in the C++ Standard Library—`vector`, `list`, `map`, `queue`—are "templates" that let us plug in our choice of the data type to be held. For a preview, see the standard library `stack` on pp. 155−157.

To qualify as a container, the class must do much more than just hold values. It must let us access the values through "iterators" (Chapters 4 and 8) and manipulate the iterators through "algorithms" (Chapters 7 and 8). Class `terminal` will eventally acquire with all of these features (Chapter 9).

Series and streams

Some objects are the source or destination (or both) of a stream of data. Our first example, the class `myrandom` on p. 176, generated a series of random integers.

Often the stream of data provided by an object comes from the outside world. Examples are class `istream`, which can read from the standard input, and class `ifstream`, which reads from a file. Or the stream can go *to* the outside world: class `ostream`, which can write to the standard output, and class `ofstream`, which writes to a file. (The most famous objects of classes `istream` and `ostream` are `cin` and `cout` respectively.)

A stream object can be dressed up to look like a container (pp. 850–855), allowing it to be manipulated by an algorithm.

Miscellaneous

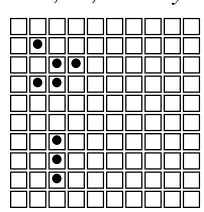
Any visible component of a GUI should be an object. If the component has a color, a symbol, or a location on the screen, the natural place to store this information is in the data members of an object. Our examples will be classes `rabbit` and `wolf` (pp. 194–197 and 197–199), and the class `terminal` itself.

Class `terminal` is our example of an *interface* class. Although it caches a little bit of data for us, its member functions do almost no work. They simply call other functions to get the job done. An interface class shields us from dealing directly with these other, presumably distasteful, functions.

Divide a program into objects

> A word of caution: beginners often find it hard to ''find the classes,'' but that problem is usually soon overcome without long-term ill effects. Next, however, often follows a phase in which classes . . . seem to multiply uncontrollably. . . . Not every minute detail needs to be represented by a distinct class . . .
>
> —Bjarne Stroustrup, *The C++ Programming Language, Special Edition, p. 734*

> An architect's first work is apt to be spare and clean. He knows he doesn't know what he's doing, so he does it carefully and with great restraint.
>
> As he designs the first work, frill after frill and embellishment after embellishment occur to him. These get stored away to be used ''next time.'' Sooner or later the first system is finished, and the architect, with firm confidence and a demonstrated mastery of that class of systems, is ready to build the second system.
>
> This second is the most dangerous system a man ever designs. . . .
>
> —Frederick P. Brooks, Jr., *The Mythical Man-Month, Anniversary Edition*, p. 55.



There are always many ways to divide a program into objects. Let's glance at some of the possibilities for the Game of Life.

(1) In Homework 2.5b, the whole game of life was one big object with a 10×10 array of `bool`'s.

(2) We could let each cell be a separate object. In this case there would be a 10×10 array of ''cell'' objects, each having a `bool` data member.

The most interesting possibilities lie between these two extremes.

(3) As the game runs, individual cells turn on and turn off. Should this pair of events be packaged as a constructor and destructor? In this case, only the *occupied* cells would be objects. The other cells would be nothing at all. The above diagram would have eight ''occupied cell'' objects, and their number would change as the game runs. The game would have one linked list containing all the occupied cell objects, each having `x` and `y` data members. The objects would no longer need the `bool` data member in the previous paragraph, since the mere fact of an object's existence indicates that its cell is occupied.

I find the occupied cell objects attractive because they make the pairing of events explicit. This approach might also save memory. Most cells will be empty most of the time because of the Law of Death. Having a permanent object for every cell would be wasteful. Finally, we would no longer have to worry about falling off the edge of the playing board—there would be no more board. The `x` and `y` data members of each object could range through all possible `int` values.

(4) To move to the next generation, we have to accumulate information about the occupied cells and the cells adjacent to the occupied cells. Perhaps all of these should be the objects, each with two `int` data members, `x` and `y`, and another `int` data member to count how many neighbors are occupied. The above diagram would have 37 objects.

(5) The above diagram has two ‘‘blobs’’ or ‘‘islands’’. Should *they* be the objects? The blobs are the active, organic entities, while the individual cells are merely machines. The above diagram would have two ‘‘blob’’ objects, each containing a linked list of ‘‘occupied cell’’ objects.

Letting each blob be a separate object would make it easier to let the user pick up a blob and move it. In addition, this approach might make the game run faster. Instead of considering the interaction between *every* pair of occupied cells, we have partitioned them into ‘‘island universes’’. We might also want to enter the shape of each blob into a hash table.

What should happen to a blob object when its blob splits into two or more blobs? How do we tell when two or more blobs come close enough to interact or merge? And when this happens, which blob object should absorb the other(s)? Or to be fair, should the merging blobs die and be replaced by one new object?

(6) The above objects are visible to the user. Now let’s consider objects that are part of the implementation. A game has several 10×10 arrays. Should each of them be an object? Let’s call the hypothetical object a `matrix`. The nested loops that print an array could then be a member function named `print`; the ones that copy an array could be a member function named `matrix::copy` (eventually to be renamed `matrix::operator=`). `matrix::copy` would be called by the the constructor for class `life` and at the end of `life::next`.

(7) Suppose we wanted to save each picture so we could run through them again, either forwards or backwards. Should each generation be an object? If so, should it be type of object in ¶ (5)?

(8) Are the above possibilities mutually exclusive?

2.9 Objects as Function Arguments and Return Values

A class of objects that announce their own birth and death

> You . . . will be told that we musicians of the eighteenth century were no better than servants. . . . But we were learned servants! . . . We took unremarkable men . . . and sacramentalized their mediocrity. Trumpets sounded when they entered the world, and trombones groaned when they left it!
>
> —Peter Shaffer, *Amadeus*, Act I, Scene 3

An object of the following class `obj` will announce its own birth and death. These objects will show us exactly when and in what order they are created, copied, and destroyed. They will also show us *how* they are created: each constructor in lines 9−11 prints a different message.

The member functions are short enough to be inline, so there is no `obj.C` file. The `operator` functions in lines 16−19 and 22 will be covered in Chapter 3.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/obj.h`

```
1 #ifndef OBJH
2 #define OBJH
3 #include <iostream>   //for <<, >>, ostream, istream
4 using namespace std;
5
6 class obj {
7     int i;
```

```
public:
    obj(int initial_i)      {i = initial_i; cout << "construct " << i << "\n";}
    obj(const obj& another) {i = another.i; cout << "copy construct " << i << "\n";}
    obj()                   {i = 0; cout << "default construct " << i << "\n";}

    ~obj() {cout << "destruct " << i << "\n";}
    void print() const {cout << i;}

    obj& operator++() {++i; return *this;}   //prefix
    operator int() const {return i;}
    friend ostream& operator<<(ostream& ostr, const obj& ob) {return ostr << ob.i;}
    friend istream& operator>>(istream& istr,       obj& ob) {return istr >> ob.i;}
};

inline const obj operator++(obj& ob, int) {      //postfix
    const obj old = ob;
    ++ob;   //ob.operator();
    return old;
}
#endif
```

The lifespan of an object

To talk about the lifespan of an object or any other variable in C or C++, we need four definitions.

A *declaration* announces the name and data type of a variable. A declaration may also be a *definition,* a statement that actually creates the variable.

```
    extern int i;   //This declaration is not a definition.
    int i = 10;     //This declaration is also a definition.
    int j;          //This declaration is also a definition.
```

A group of statements in {curly braces} is a *block;* see p. 32. A variable defined outside of any block is a *global.* A global can be mentioned in all the functions of its source file, and possibly also in other source files of the same program.

A variable has one of three possible lifespans. We say that it belongs to one of three *storage classes:* static, automatic, or dynamic.

(1) A global variable, or one defined with the keyword `static` inside a block, is said to be *static* or *statically allocated.* A static variable is constructed exactly once, and then lives without interruption until it is destructed exactly once when the program ends. "Exactly once" means no more and no less than once.

The statics fall into two groups.

(1a) Global variables are constructed before the start of `main` and destructed after the end of `main`. As usual, they are constructed in the order in which they are declared. `cin`, `cout`, and `cerr` are globals.

(1b) Variables defined with the keyword `static` inside the body of a function are constructed the first time their definition is executed.

(2) A variable defined in a block, without the keyword `static`, is said to be *automatic* or *automatically allocated.* It is constructed when we execute its definition, and destructed when we leave the block. If the block has more than one automatic variable, they are destructed in order of increasing age. We took advantage of this on p. 169.

There are several ways of leaving a block. We can always leave it by reaching the closing curly brace at the end of the block, or by executing a `return` statement. If the block is the body of a loop, we can also leave it with a `break` or `continue`. If the block is a `switch` statement, we can leave it with a `break`. See below for leaving a block by calling `exit`.

An automatic variable will be reincarnated each time we re-enter its block and re-execute its definition. For example, a variable defined in the body of a function will be re-constructed and re-destructed each time the function is called. A variable defined in the body of a loop will be re-constructed and re-destructed each time the loop is repeated.

(3) A variable that is created and destroyed with `malloc` and `free`, or with their C++ counterparts `new` and `delete`, is said to be *dynamic* or *dynamically allocated* or *on the heap*. Static and automatic variables always have names (unless they are anonymous temporaries), but dynamically allocated variables never do. See pp. 386–389 for why we would want to allocate a variable dynamically.

▼ Homework 2.9a: statically allocated objects

If a C++ program consists of more than one `.C` file, there is no way to predict which one will construct its globals first. What order do you get on your platform?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/global/global.h`

```
#ifndef GLOBALH
#define GLOBALH
#include "obj.h"

extern obj obj4;
extern obj obj5;

void f();
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/global/main.C`

```
#include <iostream>
#include <cstdlib>
#include "obj.h"
#include "global.h"
using namespace std;

obj obj1 = 10;
obj obj2 = 20;

int main()
{
    static obj obj3 = 30;
    f();
    return EXIT_SUCCESS;
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/global/other.C`

```
#include "global.h"

obj obj4 = 40;
obj obj5 = 50;

void f()
{
    obj obj6 = 60;
```

```
}
```

My platform gave me different output depending on the command line that ran the compiler.

`g++ main.C other.C`

`construct 40`	*Global objects in* `other.C` *constructed first.*
`construct 50`	
`construct 10`	*Global objects in* `main.C` *constructed last.*
`construct 20`	
`construct 30`	*local static in* `main` *function*
`construct 60`	*local static in* `f` *function*
`destruct 60`	
`destruct 30`	
`destruct 20`	*Global objects in* `main.C` *destructed first.*
`destruct 10`	
`destruct 50`	*Global objects in* `other.C` *destructed last.*
`destruct 40`	

`g++ other.C main.C`

`construct 10`	*Global objects in* `main.C` *constructed first.*
`construct 20`	
`construct 40`	*Global objects in* `other.C` *constructed last.*
`construct 50`	
`construct 30`	*local static in* `main` *function*
`construct 60`	*local static in* `f` *function*
`destruct 60`	
`destruct 30`	
`destruct 50`	*Global objects in* `other.C` *destructed first.*
`destruct 40`	
`destruct 20`	*Global objects in* `main.C` *destructed last.*
`destruct 10`	

▲

▼ Homework 2.9b: automatically allocated objects

Each time we call the function `f` in the following line 15, the automatic variable `ob` is created. The variable stays alive for as long as we stay within the curly braces in lines 16 and 19.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/infinite.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 void f(int i);
 7
 8 int main()
 9 {
10     f(1);
11     f(2);
12     return EXIT_SUCCESS;
13 }
```

```
14
15 void f(int i)
16 {
17     obj ob = i;
18     //f(i + 1);
19 }
```

```
construct 1
destruct 1
construct 2
destruct 2
```

Uncomment line 18, and comment out line 11. `f` will call itself repeatedly, creating a new variable each time. How many variables can it create before you run out of memory?

▲

▼ Homework 2.9c: automatically allocated objects

The body of this loop has curly braces, which make it a block. The object `i` is defined outside the block and will be constructed and destructed exactly once. The object `j` is define inside the block and will be constructed and destructed duing each iteration.

In newer versions of C++, `i` will be destructed before `k` is destructed. Does this happen on your platform? We detected this indirectly on pp. 34–35, but now we can see it in the output.

Define the following public member function at line 15 of `obj.h` on p. 180. Its name is only provisional. When we do operator overloading, it will be replaced by the `operator++` in line 17. See pp. 288–289.

```
20     void next() {++i;}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/automatic.C`

```
#include <iostream>
#include <cstdlib>
#include "obj.h"
using namespace std;

int main()
{
    for (obj i = 1; i <= 3; i.next()) {
        obj j = i + 10;
        j.print();
        cout << "\n";
    }

    obj k = 20;
    return EXIT_SUCCESS;
}
```

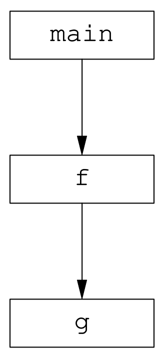
I must have a newer version of C++: my `i` was destructed before `k` was constructed.

`construct 1`	*Line 8 constructs* `i`.
`construct 11`	*Line 9 constructs the first* `j`.
`11`	
`destruct 11`	*Line 12 destructs the first* `j`.
`construct 12`	*Line 9 constructs the second* `j`.
`12`	
`destruct 12`	*Line 12 destructs the second* `j`.
`construct 13`	*Line 12 constructs the third* `j`.
`13`	
`destruct 13`	*Line 12 destructs the third* `j`.
`destruct 4`	*Line 12 destructs* `i` *after the loop is over.*
`construct 20`	*Line 14 constructs* `k`.
`destruct 20`	*Line 15 destructs* `k`.

▲

Don't deprive an object of its last rites

We have claimed that the C++ language guarantees that every constructed object will eventually be destructed. But there are three functions in the standard library that may prevent this: `exit`, `terminate`, and `abort`.



In the following program, `main` calls `f`, `f` calls `g`, and on the way down we construct static and automatic variables. The more elaborate program `unwind.C` on pp. 608−611 will have dynamic variables as well.

If we call `exit`, the program will call the destructors for the static objects* but not for the automatic and dynamic ones. For example, the output shows that the `exit` in line 25 triggered the destruction of only the three static objects. But the three other objects still in existence—`auto3`, `auto4`, and `auto5`—vanished without being destructed. If we want all of our objects to be destructed, we must never call `exit` when automatic or dynamic objects still exist.

If we call `terminate` or `abort`, the situation is even worse: the program will end without calling the destructors for any objects at all. We must never call these functions when objects still exist.

But what if we really do want to end the program without destructing the objects that stil exist? For the present, we will call `exit` anyway and just pray that nothing goes wrong. Eventually, however, we will "throw exceptions" to ensure that all the objects are properly destructed (pp. 608−611).

Line 17 is *inaccessible:* it will never be executed because of the `exit` in line 25. Some compilers require it, however, because line 12 declares that `main` returns an `int`.

* We might therefore go into an infinite loop if the destructor for a static object calls `exit`. And since there is no portable way for an object to tell if it is static, a destructor should never call `exit`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/exit.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 void f();
 7 void g();
 8
 9 obj static1 = 10;
10 obj static2 = 20;
11
12 int main()
13 {
14     obj auto3 = 30;
15     obj auto4 = 40;
16     f();
17     return EXIT_SUCCESS;
18 }
19
20 void f()
21 {
22     obj auto5 = 50;
23     static obj static6 = 60;
24     g();
25     exit(EXIT_SUCCESS);
26 }
27
28 void g()
29 {
30     obj auto7 = 70;
31     obj auto8 = 80;
32 }
```

`construct 10`	*Line 9−10 construct two global static objects.*
`construct 20`	
`construct 30`	*Lines 14−15 construct two automatic objects local to* `main`.
`construct 40`	
`construct 50`	*Line 22 constructs an automatic object local to* `f`.
`construct 60`	*Line 23 constructs a static object local to* `f`.
`construct 70`	*Lines 30−31 construct two automatic objects local to* `g`.
`construct 80`	
`destruct 80`	*Line 32 starts our journey back up.*
`destruct 70`	
`destruct 60`	*Line 25 calls* `exit`, *destructing the three static objects.*
`destruct 20`	
`destruct 10`	

Pass an object as an explicit argument to a function

> When beggars die there are no comets seen;
> The heavens themselves blaze forth the death of princes.
>
> —*Julius Cæsar*, II ii 30–31

There's an easy way to pass an object to a function in C++. Let the function be a member function of the object's class, and then the object can be passed implicitly (invisibly):

```
    date d;
    d.print();                  //d passed implicitly to print
```

But only one object can be passed implicitly. Additional objects must be passed explicitly (visibly). Here's how we would have to call a version of `print` that takes three objects.

```
    date d1, d2, d3;
    d1.print(d2, d3);           //d1 passed implicitly, d2 and d3 explicitly
```

An object will also have to be passed explicitly to a function that is not a member of the object's class. Sometimes, in fact, the function can't be a member of the object's class. For example, a function can be a member of only at most one class. If it already is a member of another class, it can't also be a member of the first one.

```
    time t;     //Imagine that there was a class time.
    date d;

    t.print(d); //d must be passed explicitly,
                //because this print is already a member function of class time
```

Even if the function could be a member function of the object's class, we might not want it to be. To minimize the number of functions that have to be debugged when a bad value shows up in a private data member, a class should have no unnecessary member functions. If the function uses no private members of the class, it does not need to be, and therefore should not be, a member function of the class.

For these reasons, an object passed to a function may have to be passed explicitly. Fortunately, the rules for passing objects explicitly are the same as those for passing arguments of the built-in types in C and C++. First we'll do it with an `int`, and then we'll do it with an object.

It looks like line 10 is passing the variable `i` to the function `f`. But when we pass an argument by value, we actually make a copy of the argument and give the copy to the function. Normally, no one is aware that a copy is created—there are no observable side effects when an `int` is copied—but it explains why `f` cannot change the value of `i`. `f` never receives `i`. It receives only a copy, and can change only the copy.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/pass_int.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f(int copy);
 6
 7 int main()
 8 {
 9     int i = 10;
10     f(i);
11     return EXIT_SUCCESS;
12 }
```

```

void f(int copy)
{
    cout << copy << "\n";
}
```

When we pass an object by value, we make a copy of the object and give the copy to the function. Of course, the copy is constructed by a copy constructor, and destructed by a destructor; the evidence is underlined in the output. This time, however, we are very much aware that a copy is created. The copy constructor and the destructor have the side effect of producing output.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/pass_obj.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 void f(obj copy);
 7
 8 int main()
 9 {
10     obj ob = 10;
11
12     cout << "about to call f\n";
13     f(ob);
14     cout << "just returned from f\n";
15
16     return EXIT_SUCCESS;
17 }
18
19 void f(obj copy)
20 {
21     cout << "start of f\n";
22     copy.print();          //just to make sure that f received the obj
23     cout << "\n";
24     cout << "end of f\n";
25 }
```

`construct 10`	*line 10 constructs* `ob`
`about to call f`	*line 12*
`copy construct 10`	*lines 13 and 19 construct* `copy`
`start of f`	*line 21*
`10`	*line 22*
`end of f`	*line 24*
`destruct 10`	*line 25 destructs* `copy`
`just returned from f`	*line 14*
`destruct 10`	*line 16 destructs* `ob`

Pass the address of the object to avoid constructing and destructing a copy

To pass an argument without copying it, we pass it by reference. The argument is now a pointer to an `obj` in lines 6 and 19. To accommodate this change, we must introduce a `&` in line 13 and a `->` in line 22.

Since the argument is merely a pointer, no copy is constructed of the object `ob`. And since the argument is a read-only pointer, the function `f` cannot change the value of `ob`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/pointer.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 void f(const obj *p);
 7
 8 int main()
 9 {
10     obj ob = 10;
11
12     cout << "about to call f\n";
13     f(&ob);
14     cout << "just returned from f\n";
15
16     return EXIT_SUCCESS;
17 }
18
19 void f(const obj *p)
20 {
21     cout << "start of f\n";
22     p->print();             //just to make sure that f received the obj
23     cout << "\n";
24     cout << "end of f\n";
25 }
```

`construct 10`	*line 10 constructs* `ob`
`about to call f`	*line 12*
`start of f`	*line 21*
`10`	*line 22*
`end of f`	*line 24*
`just returned from f`	*line 14*
`destruct 10`	*line 16 destructs* `ob`

Here's a simpler notation for passing the address of an object to a function. Lines 13 and 22 revert to their original operators.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/reference.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 void f(const obj& a);
 7
 8 int main()
 9 {
10     obj ob = 10;
11
```

```
12     cout << "about to call f\n";
13     f(ob);
14     cout << "just returned from f\n";
15
16     return EXIT_SUCCESS;
17 }
18
19 void f(const obj& a)
20 {
21     cout << "start of f\n";
22     a.print();              //just to make sure that f received the obj
23     cout << "\n";
24     cout << "end of f\n";
25 }
```

`construct 10`	*line 10 constructs* `ob`
`about to call f`	*line 12*
`start of f`	*line 21*
`10`	*line 22*
`end of f`	*line 24*
`just returned from f`	*line 14*
`destruct 10`	*line 16 destructs* `ob`

A function that returns an object

It looks like line 16 has a bug. The variable `i` in line 15 is local to the function `f`, so `i` dies when we return from `f` in line 16. Aren't we therefore returning a value that evaporates as we return it?

There is no bug, because `i` is returned via pass-by-value. Line 16 actually creates a copy of `i`, and the copy is what is returned as the original evaporates. Normally, no one is aware that a copy is created—there are no noticeable side effects when an `int` is copied—but it explains why there is no bug.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/return_int.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int f();
 6
 7 int main()
 8 {
 9     cout << f() << "\n";
10     return EXIT_SUCCESS;
11 }
12
13 int f()
14 {
15     int i = 10;
16     return i;
17 }
```

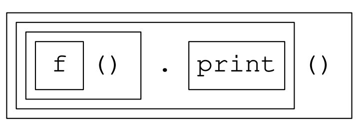
```
10
```

When returning an object via pass-by-value, a function is within its rights if it constructs and returns a copy of the object. Of course, the copy is constructed by a copy constructor and destructed by a destructor; the evidence is underlined in the output.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/return_obj.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 obj f();
 7
 8 int main()
 9 {
10     cout << "start of main\n";
11     f().print();
12     cout << "\n";
13     cout << "end of main\n";
14
15     return EXIT_SUCCESS;
16 }
17
18 obj f()
19 {
20     cout << "start of f\n";
21     obj ob = 10;
22     cout << "about to return from f\n";
23     return ob;
24 }
```

The above line 11 calls the `print` member function of the anonymous object returned by `f`. We have already seen an example of a call to a member function of an anonymous object returned by a function, in line 2 on pp. 137–138.



`start of main`	*line 10*
`start of f`	*line 20*
`construct 10`	*line 21 constructs an anonymous temporary* `obj`
`copy construct 10`	*line 21 copies the temporary into* `ob`
`destruct 10`	*line 21 destructs the anonymous temporary*
`about to return from f`	*line 22*
`copy construct 10`	*line 23 constructs the anonymous temporary* `obj` *printed in line 11*
`destruct 10`	*line 23 destructs* `ob`
`10destruct 10`	*line 11 prints & destructs anonymous temporary constructed in 23*
	line 12 outputs a newline
`end of main`	*line 13*

With a newer compiler, the above line 21 creates only one object; we saw an example of this on p. 137. In addition, this single object `ob` in line 21 is now merely another name for the (anonymous) object whose `print` function is called in line 11. The copying we just did in line 23 is gone. The two temporary objects that we no longer create—the ones that were in the above lines 21 and 23—show the two ways that temporaries can be *elided.*

`start of main`	*line 10*
`start of f`	*line 20*
`construct 10`	*line 21 constructs* `ob`
`about to return from f`	*line 22*
`10destruct 10`	*line 11 prints and destructs* `ob`
	line 12 outputs a newline
`end of main`	*line 13*

(Both of the above outputs were actually produced by the same compiler [`g++`]; I merely gave it the option `-fno-elide-constructors` when creating the executable that produced the first output.)

It is disquieting that we can legitimately get two different outputs from the same program. A program should produce the *same* output no matter what compiler was used. An object should therefore be returned via pass-by-value only if its copy constructor and destructor cause no output or other observable side effects.

In the above line 11, the anonymous temporary object in the expression `f().print()` will always be destructed after the *entire* expression has been evaluated. This is good news. If the temporary had been destructed after the `f()` but before the `.print()`, we would be printing a corpse. It is a minor annoyance that the destructor for the anonymous object emits a line of output (`"destruct 10\n"`) that appears awkwardly between the `.print()` in line 11 and the newline in line 12. We will fix this on p. 338 when the member function `print` becomes a "friend" function named `operator<<`, allowing us to print and destruct the object, and print the newline, all in the same expression.

If the anonymous temporary makes you uncomfortable, you can save the return value of `f` in a variable with a name. The above line 11 could be changed to

```
25      obj ob = f();
26      ob.print();
```

Making an unnecessary copy of a return value can sometimes be avoided by returning the variable's address (pass-by-reference). But we can't do this in the above line 21. The variable `ob` is automatically allocated, so we would be returning the address of a value that turns to garbage as we return.

Call a constructor explicitly in a return statement

If the returned object is mentioned only in the return statement (the above line 23), it's easier to construct it in the `return` statement itself. We saw earlier that a declaration is not the only place where we can construct an object. See p. 138, ¶ (4).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/objarg/return.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 obj f();
 7
 8 int main()
 9 {
10     cout << "start of main\n";
11     f().print();
12     cout << "\n";
13     cout << "end of main\n";
14
```

```
15      return EXIT_SUCCESS;
16  }
17
18  obj f()
19  {
20      cout << "start of f\n";
21      cout << "about to return from f\n";
22      return obj(10);                //Don't bother to give the object a name.
23  }
```

Since the constructor in the above line 22 takes only one argument, and since it was declared in `obj.h` without the keyword `explicit`, we may change it to

```
24      return 10;
```

See p. 138.

On my platform, the object construced in the above line 22 (or 24) is the same object as the one pinted in line 11.

```
start of main              line 10
start of f                 line 20
about to return from f     line 21
construct 10               line 22 constructs an anonymous object
10destruct 10              line 11 prints and destructs the anonymous object
                           line 12 outputs a newline
end of main                line 13
```

But there is no guarantee that they will be the same object. With an older compiler, or with the `-fno-elide-constructors` option of `g++`, they are two separate objects.

```
start of main              line 10
start of f                 line 20
about to return from f     line 21
construct 10               line 22 constructs an anonymous object
copy construct 10          line 22 copies the anonymous object
destruct 10                line 22 destructs the original anonymous object
10destruct 10              line 11 outputs and destructs the copy
                           line 12 outputs a newline
end of main                line 13
```

2.10 The Rabbit Game

▼ Homework 2.10a:
Version 1.0 of the Rabbit Game: initial version of the game

```
................................
................................
................................
..........W.........r...........
................................
................................
................................
```

C++ has many powerful features with subtle interactions, such as dynamic memory allocation, inheritance (single and multiple, public and private), virtual functions, templates, and the Standard Template Library (STL). The syntax of each feature could be presented in an example of only one or two pages. But it would take a fairly substantial program before there would be any benefit from using these constructs. Instead of burdening the student many large programs, we will deploy all of these features in one evolving program. It will become simpler, less repetitious, more orthogonal, and easier to maintain and expand. Later incarnations will deliver more functionality. And the kludges for special cases will disappear.

The program is a video game with moving animals. Carnivores will be uppercase, herbivores lowercase. The `r`, for example, is a rabbit. It hops randomly around the terminal, one step at a time. It knows that it can't move off the screen or occupy the same place at the same time as another animal.

The `W` is the wolf. It is under manual control: you have to press keys to move it. To avoid the complexity of making the arrow keys work on all platforms, we use four letters:

`h`	*left*
`j`	*down*
`k`	*up*
`l`	*right (lowercase L)*

These four letters are in a row on a QWERTY keyboard. (They are also the motion keys in the Unix editor `vi`.) I readily concede that it is counterintuitive for L to mean ‘‘right’’.

You win the game by making the wolf stomp on the rabbit. You can also win merely by launching the game and going out to lunch. The rabbit, moving randomly around the screen, will eventually blunder into the wolf and be eaten.

The game has three objects: the `terminal`, `wolf`, and `rabbit`. The calls to their constructors will be visible: the `terminal` will fill the screen with its background character, and the two animals will draw themselves. Eventually, the calls to their destructors will also be visible: the rabbit and wolf will erase themselves, and the `terminal` will blank itself out.

Each animal will have `x` and `y` data members giving its current location. We will see the data members changing: whenever this happens, the animal will move.

The main function

Line 12 of `main.C` ‘‘seeds’’ the random number generator (p. 174), ensuring that the subsequent calls to `rand` in lines 42–43 of `rabbit.C` on p. 196 will return a different series of random numbers during each run of the game. We will use the current time as our seed number. The zero used to be `NULL` in C; see p. 68. The cast suppresses the warning we would get on machines where the return type of `time` (`time_t`) is wider than the argument type of `srand` (`unsigned`).

Lines 18–19 construct the wolf and rabbit one-third of the screen apart, at middle height. The main loop in line 21 will then call the `move` member function of each animal four times per second. These functions return `true` if the rabbit is still alive, `false` if it has been eaten. In the latter case, we break out of the main loop and the game is over.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/game1/main.C`

```
 1 #include <cstdlib>   //for the srand function and EXIT_SUCCESS
 2 #include <ctime>     //for the time function
 3
 4 #include "terminal.h"
 5 #include "wolf.h"
 6 #include "rabbit.h"
 7
 8 using namespace std;
 9
10 int main()
11 {
```

```
12     srand(static_cast<unsigned>(time(0)));
13     const terminal term('.');
14
15     const unsigned xmax = term.xmax();
16     const unsigned ymax = term.ymax();
17
18     wolf   w(term, xmax * 1 / 3, ymax / 2);
19     rabbit r(term, xmax * 2 / 3, ymax / 2);
20
21     for (;; term.wait(250)) {   //250 milliseconds equals .25 seconds
22         if (!w.move()) {
23             break;
24         }
25         if (!r.move()) {
26             break;
27         }
28     }
29
30     term.put(0, 0, "You killed the rabbit!");
31     term.wait(3000);     //Give user three seconds to read the message.
32     return EXIT_SUCCESS; //Destruct rabbit, wolf, & terminal, in that order.
33 }
```

The above lines 21–28 may be combined to

```
34     for (; w.move() && r.move(); term.wait(250)) {
35     }
```

But don't do it. We would just have to change it back in a later version of the game.

Class rabbit

The game is played on a `terminal` object shared by the animals. The animals call the member functions of the terminal. One way to make the terminal accessible to the animals would be to make it a global variable.

```
const terminal term('.');

int main()
{
```

But if we did this, we would be locking ourselves into having exactly one terminal and exactly one game. It would be impossible to turn our program into a server that runs many games simultaneously.

To keep our options open, we made the terminal accessible to the animals by giving each animal a pointer to the terminal it inhabits. This pointer `t` in line 6 is read-only to make it impossible for an animal to change the size or background character of its terminal. To ensure that `rabbit.h` can mention the name of class `terminal`, it must include `terminal.h`.

The data members in lines 6–8 of `rabbit.h` are of the built-in data types: integers, characters, pointers. They are not objects, they have no constructors, and nothing happens when they are constructed. As long as they are constructed before being used in lines 13 and 29 of `rabbit.C`, it doesn't matter what order they are constructed in.

This being the case, there is no reason at present to declare `t` before the other three data members. But perhaps there will be a reason in the future. The four data members might become objects, each initialized by its own constructor. When that happens, the error checking now performed in lines 13 and 29 of `rabbit.C` will be done in the constructors for `x`, `y`, and `c`. The data member `t` is used by this error checking code, so `t` will have to be constructed first. To prepare for this eventuality, `t` is constructed first by being declared first in lines 6–8 of `rabbit.h`, although we do not need this right now. It will be one less thing to change should the data members ever become objects.

We consistently use an unsigned number to represent a position in a space whose coördinates start at zero. Earlier examples were the unsigned data type `size_t` for an array subscript (p. 66); the unsigned arguments and return value of the C functions `term_put` and `term_xmax` on p. 86; the unsigned arguments and return value of the `put` and `xmax` member functions of class `terminal` on pp. 159–160. In keeping with this practice, the coördinates of an animal are unsigned to keep them from becoming negative. These include the data members `x` and `y` in line 7 of `rabbit.h` and the local variables `newx` and `newy` in lines 50–51 of `rabbit.C`.

On the other hand, we use a signed number to represent a direction and distance of motion. Earlier examples were the signed argument of the function `date::next` and the local variables `dx` and `dy` in `life::next`. In keeping with this practice, horizontal or vertical motions are signed to let them be positive or negative. These include the offsets `dx` and `dy` in lines 43–44 of `rabbit.C`. The unsigned/signed distinction appeared in the C Standard Library as `size_t` vs. `ptrdiff_t`, and will reappear in the containers in the C++ Standard Library as `size_type` vs. `difference_type`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rabbit1/rabbit.h`

```
 1 #ifndef RABBITH
 2 #define RABBITH
 3 #include "terminal.h"
 4
 5 class rabbit {
 6     const terminal *t;
 7     unsigned x, y;
 8     char c;
 9 public:
10     rabbit(const terminal& initial_t, unsigned initial_x, unsigned initial_y);
11     bool move();
12 };
13 #endif
```

We saw back on pp. 184–185 what will go wrong when calling `exit` in line 17: the objects that are not statically allocated will never be destructed. We will fix this bug when we cover "exceptions". For now, let's hope it never happens. Line 23 disallows two animals in the same location at the same time. Line 29 disallows an invisible rabbit: one whose "color" (character) is the same as the terminal's background.

The value of the expression `rand() % 3` in line 43 is either 0, 1, or 2. The value of the larger expression `rand() % 3 - 1` is therefore –1, 0, or 1, to indicate left, no motion, or right.

Will line 53 really be able to detect an out-of-range location? Let's say that in line 50, `x` is zero and `dx` is –1. Since `x` is `unsigned` and `dx` is `int`, the sum will be `unsigned`. An `unsigned` sum, and the `unsigned` variable `newx`, cannot possibly hold the out-of-range value –1. But the –1 will be stored in `newx` as the maximum possible `unsigned` value, which line 53 will recognize as out-of-range.

Line 35 "registers" the newborn `rabbit` with its `terminal`: it informs the `terminal` that the `rabbit` exists. This is a clear demonstration that an object's constructor must sometimes do more than just put values into the object's data members. It also notifies other objects about the birth of the new one.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rabbit1/rabbit.C`

```
1 #include <iostream>
2 #include <cstdlib>   //for rand and exit functions
3 #include "rabbit.h"
4 using namespace std;
5
```

```
rabbit::rabbit(const terminal& initial_t, unsigned initial_x, unsigned initial_y)
{
    t = &initial_t;
    x = initial_x;
    y = initial_y;
    c = ’r’;

    if (!t->in_range(x, y)) {
        cerr << "Initial rabbit position (" << x << ", " << y
            << ") off " << t->xmax() << " by " << t->ymax()
            << " terminal.\n";
        exit(EXIT_FAILURE);
    }

    const char other = t->get(x, y);
    const char background = t->background();

    if (other != background) {
        cerr << "Initial rabbit position (" << x << ", " << y
            << ") already occupied by ’" << other << "’.\n";
        exit(EXIT_FAILURE);
    }

    if (c == background) {
        cerr << "Rabbit character ’" << c << "’ can’t be the same as "
            "the terminal’s background character.\n";
        exit(EXIT_FAILURE);
    }

    t->put(x, y, c);
}

//Return false if this rabbit was eaten, true otherwise.

bool rabbit::move()
{
    //The values of dx and dy are either -1, 0, or 1.
    const int dx = rand() % 3 - 1;
    const int dy = rand() % 3 - 1;

    if (dx == 0 && dy == 0) {
        return true;   //This rabbit had no desire to move.
    }

    const unsigned newx = x + dx;
    const unsigned newy = y + dy;

    if (!t->in_range(newx, newy)) {
        return true;   //Can’t move off the screen.
    }

    const char other = t->get(newx, newy);

    if (other != t->background()) {
```

```
60         if (other == c) {
61             //This rabbit collided with another rabbit.
62             return true;
63         } else {
64             //This rabbit blundered into the wolf and was eaten.
65             return false;
66         }
67     }
68
69     t->put(x, y);       //Erase this rabbit from its old location.
70     x = newx;
71     y = newy;
72     t->put(x, y, c);    //Redraw this rabbit at its new location.
73
74     return true;
75 }
```

The above lines 60–66 may be combined to the single statement

```
76         return other == c;
```

But don't do it. It's clearer the way it is now.

The above lines 70–72 may be combined to

```
77     t->put(x = newx, y = newy, c);
```

But don't do it. C++ does not share C's rage to cram as much code as possible into a single expression.

The above lines 69 and 72 seem to form a pair. Should they be rewritten as calls to a constructor and destructor for some new kind of object? Closer inspection reveals that the constructor would be called at line 72 and the destructor at 69. Then should 69 be paired with 35, and 72 with a line you will write in the destructor for class `rabbit` in Homework 2.10b?

I decided to leave lines 69 and 72 as they stand because the new object would be too trivial to be of much use. What would we call this kind of object: an `apparition`? A `quantum`? See pp. 178–179.

In conclusion, we make two criticisms of class `rabbit`. A `rabbit` interacts with a `terminal` in a very sophisticated way: it can call member functions of the `terminal` by means of the umbilical cord `t`. But a `rabbit` interacts with other animals in a very crude way: it sees only the `char` of the other animal. This is sufficient to identify the species of the other animal, but is not enough for any meaningful communication with it. When we have several other `rabbit`'s, all with the same character `'r'`, we will not be able to tell which `rabbit` we have collided with. We will remedy this on p. 467.

The second problem is more pervasive. Every line of class `rabbit` betrays the fact that our terminal is Cartesian and two-dimensional, from the data members `x` and `y` to the double-barreled arithmetic in the above lines 43–44 and 50–51. We will remedy this when we have "iterators", allowing us to rewrite the game without any mention of `x` and `y`, `dx` and `dy` (p. 966). We will then be able to port the game to a terminal with a different topology: polar coördinates, three dimensions, etc.

Class wolf

The data members, and the declarations for the member functions, are the same in classes `wolf` and `rabbit`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/wolf1/wolf.h`

```
#ifndef WOLFH
#define WOLFH
#include "terminal.h"

```

```
class wolf {
    const terminal *t;
    unsigned x, y;
    char c;
public:
    wolf(const terminal& initial_t, unsigned initial_x, unsigned initial_y);
    bool move();
};
#endif
```

An array of structures is the easiest way for a C or C++ program to store information in rows and columns (lines 27–32). In both languages, we use the data type `size_t` for the number of elements in an array (line 33).

The declaration for `p` is tucked in the left parentheses of the `for` loop in line 36; see pp. 33–34. Similarly, the declaration for `k` is tucked in the left parentheses of the `if` in line 35. The `if` will be true if the initial value of `k` is non-zero, which will happen if the user pressed a key. `k` will be destructed when we reach the end of the `if`, marked by the `}` in line 59.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/wolf1/wolf.C`

```
 1 #include <iostream>
 2 #include <cstdlib>   //for exit function
 3 #include "wolf.h"
 4 using namespace std;
 5
 6 wolf::wolf(const terminal& initial_t, unsigned initial_x, unsigned initial_y)
 7 {
 8     t = &initial_t;
 9     x = initial_x;
10     y = initial_y;
11     c = 'W';
12
13     //Copy lines 13-35 of the above rabbit.C here,
14     //changing the word "rabbit" to "wolf".
15 }
16
17 //Return false if this wolf ate another animal, true otherwise.
18
19 bool wolf::move()
20 {
21     struct keystroke {
22         char c;
23         int dx;   //horizontal difference
24         int dy;   //vertical difference
25     };
26
27     static const keystroke a[] = {
28         {'h',   -1,    0},   //left
29         {'j',    0,    1},   //down
30         {'k',    0,   -1},   //up
31         {'l',    1,    0}    //right
32     };
33     static const size_t n = sizeof a / sizeof a[0];
34
35     if (const char k = t->key()) {
```

```
36          for (const keystroke *p = a; p < a + n; ++p) {
37              if (k == p->c) {
38                  const unsigned newx = x + p->dx;
39                  const unsigned newy = y + p->dy;
40
41                  if (!t->in_range(newx, newy)) {
42                      break;          //Go to line 57.
43                  }
44
45                  const bool I_ate_him =
46                      t->get(newx, newy) != t->background();
47
48                  t->put(x, y);       //Erase this wolf from its old location.
49                  x = newx;
50                  y = newy;
51                  t->put(x, y, c);    //Redraw this wolf at its new location.
52
53                  return !I_ate_him;
54              }
55          }
56
57          //Punish user who pressed an illegal key or tried to move off screen.
58          t->beep();
59      }
60
61      return true;
62  }
```

Up to one quarter of a second may elapse between a keystroke and the next call to the wolf's `move` function, causing the wolf to respond sluggishly. This could be fixed by making the input ‘‘interrupt driven’’, but we will not pursue it for now.

For the present, there is an asymmetry in the behavior of colliding animals. When a wolf stomps on a rabbit, the rabbit disappears. But when a rabbit blunders into a wolf, the rabbit merely freezes because its move is never carried out. We'll fix this on p. 469 when we introduce ‘‘dynamic memory allocation’’, which will give us greater control over the exact moments of an object's birth and death.

Classes `wolf` and `rabbit` are identical in their data members, almost identical in their constructors, and similar in their remaining member functions. We will eventually consolidate this duplication by means of *inheritance* from a common base class.

List of the nine source files that constitute the game

(1) `term.h` and `term.c` (pp. 85–89). These are the only two written in C; the rest are C++.

(2) `terminal.h` and `terminal.C` (pp. 157–163)

(3) `main.C` (pp. 193–194)

(4) `rabbit.h` and `rabbit.C` (pp. 194–197)

(5) `wolf.h` and `wolf.C` (pp. 197–199)

Compile the game on Unix

```
1$ gcc -I. -DUNIX= -c term.c
2$ ls -l term.o
```

```
3$ g++ -I. -o ~/bin/game main.C wolf.C rabbit.C terminal.C term.o -lcurses
4$ ls -l ~/bin/game

5$ game                                    Run the game.
6$ echo $?                                 See the game's exit status.
```

▲

▼ Homework 2.10b:
Version 1.1 of the Rabbit Game: destructors for classes `wolf` and `rabbit`

Write a destructor for class `wolf`, even though there currently is no animal that could eat a `wolf`, and a destructor for class `rabbit`. Each destructor should do three things in the following order.

(1) Beep the terminal on which the dying animal is displayed.

(2) Pause for one second, so the dying animal stands "frozen in the headlights".

(3) Call the `get` member function of the animal's terminal to see if the animal's location on the screen is occupied by the animal's character. If so, call the `put` member function of the animal's terminal to wipe the animal's character off the screen by displaying the terminal's background character there, as in line 48 of the above `wolf.C`. Otherwise, do not call `put` and do not change the character at that location on the screen, because the location is already occupied by another animal. Remember, there is one occasion when two animals are momentarily at the same place at the same time: right after the wolf stomps on the rabbit. This anomalous situation will be removed on p. 470 when we have "dynamic memory allocation", but for now we have to handle it.

The destructor should not change the value of any of the dying animal's data members. There would be no point in doing so, since the animal is about to evaporate. Changing its data members would be like rearranging the deck chairs on the *Titanic*.

▲

▼ Homework 2.10c:
Version 1.2 of the Rabbit Game: make the animals impossible to copy

Make the wolves and rabbits impossible to copy by depriving them of their copy constructors. A C++ object can be copied only by its copy constructor.

Even though we defined no copy constructor for classes `wolf` and `rabbit`, the computer behaves as if we had (p. 135). To prevent the computer from doing this, declare a private copy constructor for each class but do not define it. In other words, do not write a function body. If a member function of one of these classes tries to call the copy constructor for that class, the copy constructor will be undefined and the program will not link. And if any other function tries to call the copy constructor, the copy constructor will be private and the program will not even compile. In either case, it will be impossible to copy the animal.

```
class rabbit {
    const terminal *t;
    unsigned x, y;
    char c;
    rabbit(const rabbit& another);   //deliberately undefined
public:
    //etc.
```

▲

2.11 Friend Functions

The unit of protection is a class, not an object.

We have said that the private members of an object can be mentioned only by the member functions of that object. But under certain circumstances they can also be mentioned by the member functions of other objects *of the same class*. We have even seen two examples: the copy constructor for class `mono`, in line 57 of `duo.C` on pp. 136–137, and the copy constructor for class `stack`, on p. 153. Each copy constuctor was able to mention the private members of two different objects: the object of which it was a member, and `another` object which it received as an explicit argument.

More examples are in the following class `point`, which represents a point in a two-dimensional space with Cartesian coördinates *x, y.*

Four examples in this class show that it is quite natural for the member functions of one object to use the private members of other objects of the same class. In fact, a `point` whose member functions couldn't do this would be useless. It would be stuck in its own solipsistic universe, with no way to interact with other `point`'s.

(1) The member function `dist` in lines 17–25 of `point.C` mentions the private members of two objects of class `point`: the object of which it is a member, and `another` object which it receives as an explicit argument.

(2) The member function `dist` in lines 7–15 of `point.C` mentions the private members of two objects of class `point`: the object of which it is a member, and the global object `point_origin` in line 5 of `point.C`.

(3) The member function `midpoint` in lines 19–21 of `point.h` mentions the private members of two objects of class `point`: the object of which it is a member, and `another` object which it receives as an explicit argument.

(4) The member function `area` in lines 27–33 of `point.C` mentions the private members of three objects of class `point`: the object of which it is a member, and two more which it receives as explicit arguments.

The members are named `dist` to avoid conflict or confusion with the `distance` function in the C++ Standard Library.

Line 24 of `point.C` constructs and returns an anonymous `double`. Similarly, line 20 of `point.h` constructs and returns an anonymous object. See p. 138, ¶ (4). It can construct the object even before the entire declaration for the class (lines 6–24) has been seen, since `midpoint` is inline.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/point1/point.h`

```
 1 #ifndef POINTH
 2 #define POINTH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class point {
 7     double x, y;
 8 public:
 9     point(double initial_x = 0.0, double initial_y = 0.0) {
10         x = initial_x;
11         y = initial_y;
12     }
13
14     void print() const {cout << "(" << x << ", " << y << ")";}
15
16     double dist() const;
17     double dist(const point& another) const;
18
19     point midpoint(const point& another) const {
```

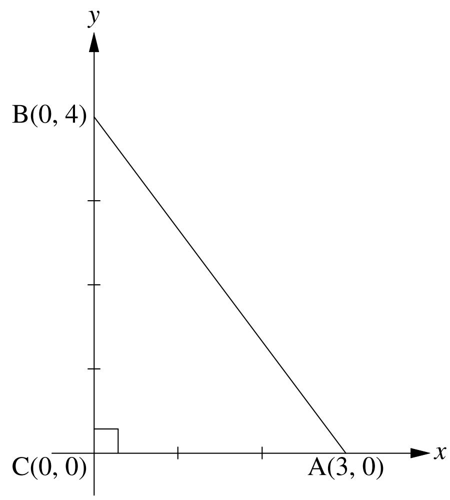
```
        return point((x + another.x) / 2, (y + another.y) / 2);
    }

    double area(const point& A, const point& B) const;
};
#endif
```

The `point_origin` in line 5 of `point.C` is not a data member of class `point`. It merely floats somewhat unsatisfactorily near it. See the similar disposition of the array `date_length` on pp. 114–115.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/point1/point.C`

```
 1 #include <cmath>   //for sqrt
 2 #include "point.h"
 3 using namespace std;
 4
 5 const point point_origin;   //give arguments 0.0, 0.0 to constructor
 6
 7 //Return the distance between this point and the origin.
 8
 9 double point::dist() const
10 {
11     const double dx = point_origin.x - x;
12     const double dy = point_origin.y - y;
13
14     return sqrt(dx * dx + dy * dy);               //Pythagorean theorem
15 }
16
17 //Return the distance between this point and another.
18
19 double point::dist(const point& another) const
20 {
21     const double dx = another.x - x;
22     const double dy = another.y - y;
23
24     return sqrt(dx * dx + dy * dy);
25 }
26
27 //Return the area of the triangle whose vertices are points *this, B, and C.
28
29 double point::area(const point& B, const point& C) const
30 {
31     return abs(x * B.y + B.x * C.y + C.x * y
32          - y * B.x - B.y * C.x - C.y * x) / 2;
33 }
```

The `main` function constructs these three points:



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/point1/main.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "point.h"
4 using namespace std;
5
6 int main()
7 {
8     //A 3-4-5 right triangle with its right angle at the origin.
9     const point A(3, 0);
10    const point B(0, 4);
11    const point C;
12
13    cout << "A's distance from origin is " << A.dist() << ".\n"
14        << "The distance between A and B is " << A.dist(B) << ".\n"
15        << "The area of triangle ABC is " << C.area(A, B) << ".\n";
16
17    cout << "The midpoint of A and B is ";
18    const point M = A.midpoint(B);
19    M.print();
20    cout << ".\n";
21
22    return EXIT_SUCCESS;
23 }
```

A C or C++ variable that is used only once can be replaced with an anonymous temporary. The object `M` in the above line 18, for example, is used only in line 19. These two lines may therefore be combined to the following statement, calling the `print` member function of the anonymous object returned by the `midpoint` member function of `A`.

```
24    A.midpoint(B).print();
```

```
A . midpoint ( B ) . print ()
```

Our first example of a call to a member function of an anonymous object returned by a function was in line 2 on pp. 137–138.

The above line 14 output a `double` with the << operator. In the same way, we will eventually be able to output a `date` with the << operator instead of the `.print()` function, Lines 17–20 will be combined to

```
25	cout << "The midpoint of A and B is " << A.midpoint(B) << ".\n";
```

which will then be adjoined to the `cout` statement that begins in line 13.

```
A's distance from origin is 3.
The distance between A and B is 5.
The area of triangle ABC is 6.
The midpoint of A and B is (1.5, 2).
```

Symmetry as a motivation for friend functions

The member function `dist` in line 19 of the above `point.C` on p. 202 deals evenhandedly with its two `point`'s. In fact, it would return the same value even if the two `point`'s were interchanged.

But the function is written in a lopsided notation that arbitrarily favors one of the objects. It seems unfair that one `point` is anonymous, while the other has a name ("`another`"). In line 21, for example, the two `x`'s are called plain old `x` and `another.x`. A more balanced notation would provide names for both objects. In line 29, the formula in the `area` function would be easier to read if all three `point`'s had names.

The calls to these functions are also unfair. Why should line 14 of the above `main.C` say `A.dist(B)`? What entitles `A` to a place in the sun while `B` is cowers in the parentheses?

Our criticisms are purely aesthetic—so far. But aesthetic deficiency causes bugs. Consider the Pythagorean theorem in lines 21–24 of the above `point.C`. This formula is so well known that we were able to write the preliminary subtractions correctly even with a lopsided notation. But the formula for the area of a triangle in lines 31–32 is more arcane. We will soon see that it is a perfectly balanced "determinant" from Linear Algebra. But its symmetry is obscured by the lopsided notation: only two of the three objects have names. It took me several tries before I was able to transcribe the formula correctly.

Now why don't all three objects in the `area` function have names? Well, a member function receives an implicit argument pointing to an object. To access the members of that object, the member function offers us the simplest possible notation: no notation at all. Think back to the body of the first member function of our first class, `date::print`. We wrote nothing there to identify the object to which the three members belong. They belong to the object targeted by the implicit pointer:

```
1	cout << month << "/" << day << "/" << year;
```

This minimal notation has always been the glory of a member function: when concentrating on one object, the object needs no name. But now that we're writing functions dealing with the private members of two or more objects passed as arguments, it's awkward that we have names for all of them except one. It would be simpler if every argument had a name.

Friend functions

This is where *friend functions* come in. A friend of a class is the same as a member function of the class, except that it does not receive an implicit argument pointing to an object of that class. All of the friend's arguments must be explicit: declared within the parentheses of the argument list. Like a member function, a friend of a class can mention the private members of that class.

The member function `dist` in lines 17–25 of the above `point.C` shambles along with one implicit argument and one explicit argument. The friend function `dist` in lines 17–25 of the following `point.C` has both arguments explicit. The difference between the member function and the friend is only a matter of notation: the source code of the friend is more balanced. Deep in the machine, both functions take the same two arguments passed the same way (by reference). The functions do the same work and are equally fast.

Since it has no implicit argument, a friend must always say which object it is accessing the members of. For example, within the bodies of the following friends we must always say `A.x` or `B.x`; we can never say plain old `x`.

Write the keyword `friend` only inside the definition of the class that the friend is a friend of. See the declaration in line 24 of `point.h` and the definition in line 19 of `point.C`. A friend of a class cannot be a member function of the same class, so the customary `point::` is not written in front of the function names in lines 19 and 29 of `point.C`. But even though they are not members, we still define the functions in the `point.C` file.

A friend is a free function (p. 113): it takes no implicit pointer argument. It is called with the same syntax as any free function; see line 14 of `main.C`.

The categories `const` vs. non-`const` apply only to member functions, not to friends. A `const` member function cannot modify the object to which it receives an implicit pointer (lines 20 and 23 of `point.h`). But a friend receives no implicit pointer, so a `const` friend would be meaningless. Instead, each argument passed by reference to a friend can be declared `const` or non-`const`; see the two in line 24 of `point.h`.

When do we have no choice between a member function and a friend?

A function that does not mention the private members of a class should be neither a member function nor a friend of that class. This will keep the number of suspects as small as possible in case an incorrect value appears in a private data member.

A function that needs to mention the private members of a class will have to be either a member function or a friend of the class. In three cases, we have no choice.

(1) If there are a constructor and destructor, they must be member functions.

(2) If the function must be private, it must be a member function. There is no such thing as a private friend. In fact, the terms ''public'' and ''private'' do not apply to friends at all, only to members. In lines 24, 26, and 30 of the following `point.h`, we just happened to declare the friends in the public section of the class, but it would have made no difference had we declared them in the private section. It makes more sense to declare them in the public section, though.

(3) If a function needs to mention the private members of two or more classes, it can be a friend of all of them. The function can even be a member of one class and a friend of one or more other classes. But a function cannot be a member function of more than once class: a function cannot receieve more than one implicit pointer.

This means that if a function needs to mention the private members of two or more classes, it can be a member function of at most one of them and must be a friend of all the rest. Do we ever need to mention the private members of two or more classes? The textbook example would be a function that multiplies a matrix and a vector, returning the product. Our three examples are the `operator>>` that takes a `scale` on p. 373 (a friend of classes `scale` and `point`); the `operator/` on p. 296 (a member of class `timebomb` and a friend of class `date`); and the `get` on p. 467 (a member of class `game` and a friend of class `rabbit`).

What should we do when we have a choice?

Other than the above cases, a function that needs to mention the private members of a class can be eithe a member function or a friend. Here are our recommendations.

(1) A function that mentions the private members of exactly one object should be a member function of the class of that object. See the `print` and `dist` member functions in lines 20 and 23 of `point.h`.

(2) A function that mentions the private members of two or more objects should still be a member function of one of them if that object plays the starring rôle. In the `assign` member function in line 21 of `point.h`, for example, one of the objects acquires a new value while the other remains completely untouched. (When we do "operator overloading", we'll give this member function its conventional name: `operator=`.)

(3) A function that uses the private members of two or more objects playing equal rôles should be a friend of the class of those objects. See the `dist`, `midpoint`, and `area` friends in lines 24, 26, and 30 of `point.h`. The area formula in lines 31–32 of `point.C` now has names for all three `point`'s. Its derivation remains a mystery, but at least its symmetry is now apparent.

The functions `collinear` and `contains` are neither member functions nor friends of class `point`. But they can be declared in `point.h` anyway, since they will be called by most of the `.C` files that use the class. One function is inline, the other not.

I considered making `contains` a member function, since one of its four objects plays a special rôle. But I decided it was more important to have no unnecessary member functions or friends. We might have to introduce a fudge factor into `collinear`.

Instead of repeating the trio of arguments `ABC` in lines 15 and 31, we'll eventually collect them into one bigger object of class `triangle` (pp. 264–265). As we shall see, a big object can contain smaller objects as its data members.

The `point_error` in line 7 of `point.h` is not a data member of class `point`. It merely floats somewhat unsatisfactorily near it. See the similar disposition of the array `date_length` on pp. 114–115.

A smothered friend

A `friend` must always be declared in the class definition. For example, `dist` and `midpoint` are declared at lines 24 and 26 in the following definition for class `point`.

`dist` was too big to be inline, so we defined it in `point.C`. But `midpoint` is inline, and defined in the class definition. Do we also need a declaration for `midpoint` outside of the class definition?

Surprisingly, the answer depends on the arguments of the friend. Given the arguments `A` and `B` of class `point` in line 18 of the following `main.C`, the computer will look for a friend named `midpoint` in the definition for class `point`. The same would be true for arguments of any type compounded from class `point`: "pointer to `point`", "array of `point`", etc. The declaration of `midpoint` in line 26 of `point.h` therefore suffices for the call to `midpoint` in line 18 of `main.C`.

Line 23 of `main.C` prints the address of `midpoint`. (For the double cast, see line 24 of `reinterpret_cast.C` on p. 67.) But line 23 passes no arguments of type `point` to `midpoint`. The computer will therefore not look for a friend named `midpoint` in the class definition for `point`, even though it did this as recently as line 18. To get line 23 to compile, we must also declare `midpoint` outside of the class definition, at line 33 or line 10 of `point.h`. In the latter case, line 10 would require the sneak preview of the name `point` in line 9. See p. 363 for another example.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/point2/point.h`

```
 1 #ifndef POINTH
 2 #define POINTH
 3 #include <iostream>
 4 #include <cmath>   //for abs
 5 using namespace std;
 6
 7 const double point_error = .0001;   //floating point roundoff error
 8
 9 //class point;
10 //point midpoint(const point& A, const point& B);
11
12 class point {
```

```
    double x, y;
public:
    point(double initial_x = 0.0, double initial_y = 0.0) {
        x = initial_x;
        y = initial_y;
    }

    void print() const {cout << "(" << x << ", " << y << ")";}
    void assign(const point& another) {x = another.x; y = another.y;}

    double dist() const;
    friend double dist(const point& A, const point& B);

    friend point midpoint(const point& A, const point& B) {
        return point((A.x + B.x) / 2, (A.y + B.y) / 2);
    }

    friend double area(const point& A, const point& B, const point& C);
};

point midpoint(const point& A, const point& B);

//Return true if all three points lie along the same line.

inline bool collinear(const point& A, const point& B, const point& C) {
    return abs(area(A, B, C)) < point_error;
}

bool contains(const point& A, const point& B, const point& C, const point& D);
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/point2/point.C`

```
#include <cmath>   //for sqrt, abs
#include "point.h"
using namespace std;

const point point_origin;

//Return the distance between this point and the origin.

double point::dist() const
{
    const double dx = point_origin.x - x;
    const double dy = point_origin.y - y;

    return sqrt(dx * dx + dy * dy);               //Pythagorean theorem
}

//Return the distance between points A and B.

double dist(const point& A, const point& B)
{
    const double dx = A.x - B.x;
```

```
    const double dy = A.y - B.y;

    return sqrt(dx * dx + dy * dy);
}

//Return the area of triangle ABC.

double area(const point& A, const point& B, const point& C)
{
        return abs(A.x * B.y + B.x * C.y + C.x * A.y
                 - A.y * B.x - B.y * C.x - C.y * A.x) / 2;
}

/*
Return true if the triangle ABC contains the point D, or if D is on the
perimeter.  Ideally area(A, B, C) would equal sum exactly, but floating point
arithmetic is never exact.  We return true if area(A, B, C) is within
point_error of sum.
*/

bool contains(const point& A, const point& B, const point& C, const point& D)
{
    const double sum = area(A, B, D) + area(B, C, D) + area(C, A, D);
    return abs(area(A, B, C) - sum) < point_error;
}
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/point2/main.C

```
#include <iostream>
#include <cstdlib>
#include "point.h"
using namespace std;

int main()
{
    //A 3-4-5 right triangle with right angle at the origin.
    const point A(3, 0);
    const point B(0, 4);
    const point C;

    cout << "A's distance from origin is " << A.dist() << ".\n"
        << "The distance between A and B is " << dist(A, B) << ".\n"
        << "The area of triangle ABC is " << area(A, B, C) << ".\n";

    cout << "The midpoint of A and B is ";
    const point M = midpoint(A, B);
    M.print();
    cout << ".\n";

    cout << "The address of midpoint is "
        << reinterpret_cast<void *>(reinterpret_cast<size_t>(midpoint))
        << ".\n";

    cout << "A, B, and M are " << (collinear(A, B, M) ? "" : "not ")
```

```
27         << "collinear.\n";
28
29     //Borderline case: M is on on the perimeter.
30
31     cout << "Triangle ABC "
32         << (contains(A, B, C, M) ? "contains" : "does not contain")
33         << " M.\n";
34
35     return EXIT_SUCCESS;
36 }
```

A C or C++ variable that is used only once can be replaced with an anonymous temporary. Had `M` been used only in line 19, for example, we could have combined 18–19 to the following line. It calls the `print` member function of the anonymous object constructed and returned by the `midpoint` friend. See pp. 203–204.

```
37     midpoint(A, B).print();
```

```
A's distance from origin is 3.
The distance between A and B is 5.
The area of triangle ABC is 6.
The midpoint of A and B is (1.5, 2).
The address of midpoint is 0x1154c.
A, B, and M are collinear.
Triangle ABC contains M.
```

▼ Homework 2.11a: consolidate duplicate code

The member function `dist` in lines 7–15 of the above `point.C` can be reduced to a single statement:

```
1 //Return the distance between this point and the origin.
2
3 double point::dist() const
4 {
5     return ::dist(*this, point_origin);
6 }
```

The double colon makes the above line 5 call the `dist` that is a free function, i.e., the one in lines 17–25 of the above `point.C`. This function happens to be a friend, but the scoping rules give no special treatment to friends. The only distinction they care about is member function vs. free function.

Without the double colon in line 5, the computer would try three possibilities when it sees the word `dist` in that line. See p. 123.

(1) Is `dist` the name of a locally declared item (the name of a variable, function, typedef, enumeration, etc., declared within the `{`curly braces`}` of the block)? In our case, no. This function has no local declarations at all.

(2) Is `dist` the name of a member of class `point`? In our case, yes, and the computer would stop here.

(3) Is `dist` the name of a global item? In our case, the computer would never get this far.

So without the double colon, line 5 would try to call the member function `dist`. We would then get an error message because the number of explicit arguments is wrong.

The arguments of the `dist` friend must be `point`'s, not pointers to `point`'s. That's why the `this` has a star. `this` is merely a pointer to a `point`; the actual `point` is `*this`.

The above function is now short enough to be made inline by moving its definition to `point.h`. But for the time being it has to stay in `point.C`, because `point_origin` is in scope only there. We'll fix this on p. 239 when we have "static" data members.

▲

Class point in the C++ Standard Library

The standard library has no class named `point`. But it has a similar one, class `complex`, with the same two data members. In fact, we can choose the type of the data members because class `complex` is a template class, like the standard library class `stack` on pp. 155−157. The reasonable choices are `float`, `double`, and `long double`.

We can perform arithmetic on complex numbers, such as the subtraction in line 17. There are also functions whose argument is a complex and whose return type is the data type of the data members of the complex number. For example, the `abs` and `norm` functions in lines 15 and 16 return `double` because `b` is a `complex<double>`. Other functions (`sin`, `sinh`) take and return a complex.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/point2/complex.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <complex>
 4 #include <cmath>   //for sqrt
 5 using namespace std;
 6
 7 int main()
 8 {
 9     complex<double> a(3);         //last argument defaults to 0.0
10     complex<double> b(0, 4);
11     complex<double> c;            //arguments default to 0.0, 0.0
12
13     cout << b << "\n"
14         << "x, y coordinates: " << b.real() << ", " << b.imag() << "\n"
15         << "r, theta coordinates: " << abs(b) << ", " << arg(b) << "\n"
16         << "distance from origin: " << sqrt(norm(b)) << "\n"
17         << "distance from a: " << sqrt(norm(b - a)) << "\n";
18
19     return EXIT_SUCCESS;
20 }
```

```
(0,4)
x, y coordinates: 0, 4
r, theta coordinates: 4, 1.5708      1.5708 = π/2 radians is 180°
distance from origin: 4
distance from a: 5
```

▼ Homework 2.11b: define four free functions

A *free* function is one that receives no invisible arguments. Declare the following free functions in `date.h`. Define the small ones to be inline in `date.h`; define the big ones in `date.C`.

If a function needs to mention a private member of class date, make the function a friend. If it does not need to mention any private member, do not make the function a friend or a member function.

A friend must be declared inside the class definition. To make the friend mentionable in the absence of arguments of class `date`, the friend must also be declared outside the class definition. An example was the `midpoint` friend of the above class `point`.

```
    bool     equals(const date& d1, const date& d2);
    int        dist(const date& d1, const date& d2);
    const date& min(const date& d1, const date& d2);
    date   midpoint(const date& d1, const date& d2);
```

You can use the version of class date with either one, two, or three data members. But do not remove any of the member functions you added to class date in previous homeworks. Your four new functions must produce no output. Demonstrate that they are correct by handing in the output of http://i5.nyu.edu/~mm64/book/src/date/test2/main.C. And make sure that the test program http://i5.nyu.edu/~mm64/book/src/date/test1/main.C. still works.

(1) equals will return true if d1 and d2 are the same date. When we do "operator overloading", we'll give this function its conventional name on p. 278: operator==. For now, the name equals was chosen to avoid conflict with the equal_to and equal functions in the C++ Standard Library.

(2) dist will return the distance in days between the two dates. The distance between two point's is always non-negative, but the distance between two date's may be positive, negative, or zero. The return value should be positive if d1 is later than d2, negative if d1 is earlier than d2, and zero if d1 and d2 are the same date. For example, if d1 is January 3, 2014 and d2 is January 1, 2014, the return value would be 2. And if d1 is January 1, 2014 and d2 is January 3, 2014 the return value would be −2.

When we do operator overloading, we'll give this function its conventional name: operator-. For now, the name dist was chosen to avoid conflict with the distance function in the C++ Standard Library.

(3) min will return a read-only reference to the earlier of the two dates. For compatibility with the min in the standard library, min should return a reference to the date on the left if the two arguments are equal. min does not have to construct a new date; it should return a reference to one of the existing date's passed in as arguments. Since these date's will not be destructed as we return from min, we can get away with return by reference.

Your min function will belong to no namespace. Another min function, belonging to namespace std, is declared in the header file <algorithm>. We did not include this header directly, but it might have been included by one of the headers that we did include.

When you call your min function (but not when you declare or define it) please write its name as ::min. This will ensure that you call the min that belongs to no namespace; the double colon is needed only if the header <algorithm> was included. Assuming that <algorithm> was included, a call to std::min would have been the min function belonging to namespace std, and a call to an unadorned min would not have compiled.

We will remove this min function on p. 634.

(4) midpoint will construct and return the date that is halfway between d1 and d2. For example, if d1 is January 3, 2014 and d2 is January 1, 2014 (or vice versa), midpoint will construct and return a date whose value is January 2, 2014. You can measure the distance between d1 and d2 by calling your dist function. Then divide the distance by 2, and add the quotient to d2.

If the distance between the two date's is an odd number of days, construct and return the date that is immediately *before* the midpoint. For example, if d1 is January 4, 2014 and d2 is January 1, 2014 (or vice versa), construct and return a date whose value is January 2, 2014.

When dividing an odd distance by 2, we must therefore make sure that the division truncates *downwards* (towards negative infinity). For example, a quotient of 1½ should be rounded to 1, and a quotient of −1½ should be rounded to −2. The div function in the C Standard Library always gives us a quotient truncated towards zero (−1½ becomes −1), but a follow-up if can easily change the quotient to one truncated downwards:

```
#include <cstdlib>            //for div and div_t
using namespace std;

    div_t d = div(dividend, 2);
```

```

	if (d.rem < 0) {		//if truncation was in wrong direction
		--d.quot;		//truncate downwards
	}

	cout << "The quotient truncated downwards is == " << d.quot << "\n";
```

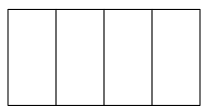
Unlike `min`, `midpoint` will construct and return a new `date`. This `date` will be a local variable inside of `midpoint`. Like all local variables, it will be destructed as we return from `midpoint`. `midpoint` must therefore return by value, not by reference. We can never return the address of a variable that is destructed as we return. See pp. 76–77.

▲

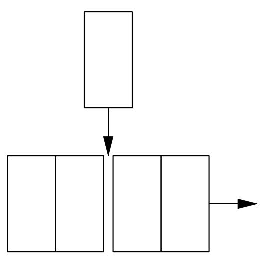
An array vs. a linked list

Here's another example in which (a member function of) one object needs to access the private members of other objects of the same class. Our examples are the `n->prev` and `n->next` in lines 45 and 54 of `node.C` on pp. 215–216. The data member `prev` is currently public, but it will become private when we have "iterators".

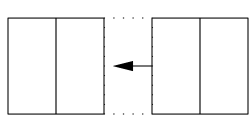
The elements in an array (and later, in a `vector`) are stored shoulder to shoulder in memory. There is no wasted space:



But this virtue becomes a liability if we want to insert a new element in the middle of the array. All subsequent elements will have to be moved one space to the right to make room for the new one. And there might be millions of them.



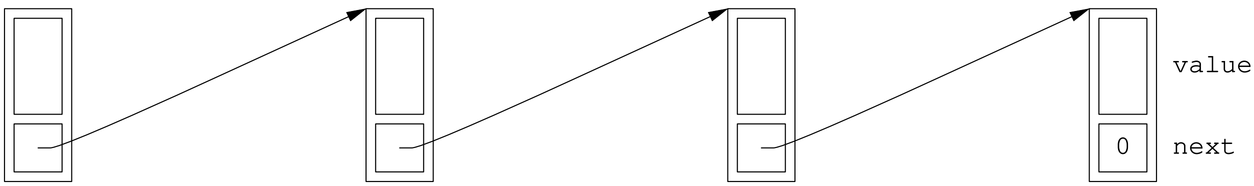
The same problem will happen if we delete an element from the middle of the array. All subsequent elements will have to be moved one space to the left to close the gap.



If we anticipate many insertions and deletions, it would be faster to store the information in a *linked list.* We drop the requirement that the elements be stored consecutively. They can be spaced far apart in memory, leaving plenty of room for insertions:

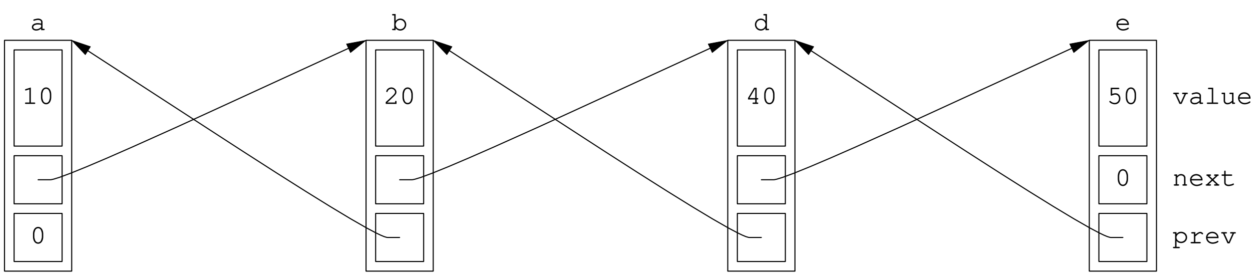
But now we have a new problem. How can a loop navigate from one list element to the next? This wasn't a problem in an array, where each element is located immediately after its predecessor. But the elements of a linked list may be far apart, at irregular intervals.

Each element of a linked list is therefore provided with a pointer giving the address of the next element. In other words, each element is now an object with two data members. These objects are called *nodes*. The payload data member will be named `value` (to agree with the `value_type` typedef below); the pointer data member will be named `next`. To loop from left to right along the linked list, we follow the `next` pointers. The last node has a `next` whose value is is zero.

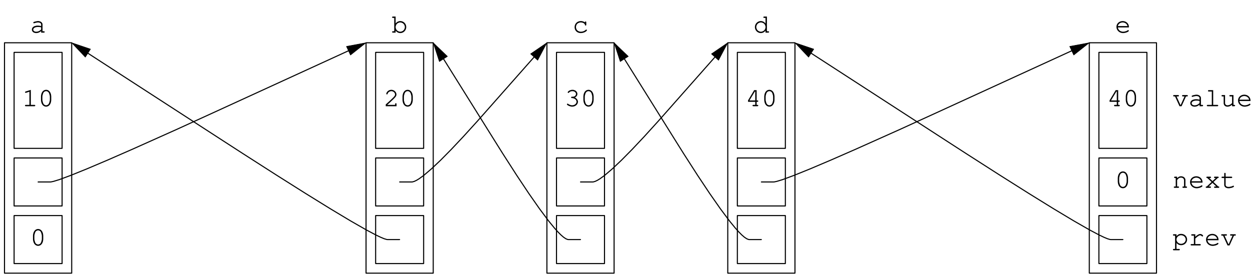


If we also want to loop from right to left, each node can have another pointer named `prev`, pointing to the previous node. The first node has a `prev` whose value is is zero. The list is now said to be *doubly-linked.* (The code looks neater when the two names, `next` and `prev`, have the same number of characters.)

I've named the two center nodes `b` and `d`. Imagine that they are linked together in the middle of a list, holding the values `20` and `40`:



We can insert a new node `c` between `b` and `d` without moving any existing node.



The `insert_this_before` function in line 21 of `node.h` deals with two nodes. But one of them gets inserted while the other is merely a bystander, so we made `insert_this_before` a member function of the object that is inserted. Similarly, the `link` function in line 18 mentions the private members of two nodes. But they play equal rôles, so we made `link` a friend. Finally, the `detach` function in line 19 of `node.h` does not mention the private members of any node. It therefore does not need to be a member function or a friend. But we made it a member function anyway, so we could call it in line 27 of `main.C` with the same syntax as the `insert_this_before` in line 19 of `main.C`.

Copying a node might be useful for splicing genes into trees and networks, but it would corrupt our simple lists. To make sure that no node can be copied, we let the copy constructor in line 10 of `node.h` be private and undefined as we did with classes `wolf` and `rabbit`. This made it impossible to pass the arguments of `link` and `insert_this_before` by value.

It is impossible for a node to contain a node—it would blow up to infinite size—but it is okay for a node to contain a pointer to a node such as the data members `prev` and `next`. We assign values to these data members in lines 10, 25, 27, 34, and 36 of `node.C`, which is why they had to be pointers, not references. A reference always refers to the same object.

Code that constructs a `node`, or that otherwise requires us to know the size of a `node`, will have to wait until after the end of the class definition in line 26 of `node.h`. But code that merely mentions class `node`, without constructing an actual `node` object, can appear any time after line 8. Simple examples are the pointers in lines 12 and 18. A more complicated one is on p. 716.

`prev` and `next` are public to make it possible for the `main` function to loop through the list. (This is what is meant by "fast and dirty" programming.) We will make them private when we do iterators.

The C++ convention is to make a typedef named `value_type` for the type of data stored in a container (lines 6, 9, and 25 of `node.h`). An example was back on pp. 153–154.

A final curiosity. An inline member function or friend, defined within the {curly braces} of a class declaration, can mention a member or friend that has not yet been declared. This allows the destructor in line 16 of `node.h` to mention the `detach` function in line 19. It is the only place where anything in C or C++ can be mentioned before it is declared. See p. 119.

The `operator value_type` member function in line 25 will be explained on pp. 315–316 and used in line 52 of `linked.C` on p. 399.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node/node.h`

```
 1 #ifndef NODEH
 2 #define NODEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 typedef int value_type;
 7
 8 class node {                        //A node on a doubly-linked list.
 9     value_type value;
10     node(const node& another);   //deliberately undefined
11 public:
12     node *prev;
13     node *next;
14
15     node(const value_type& initial_value);
16     ~node() {detach();}
17
18     friend void link(node *n1, node *n2);
19     void detach() {link(prev, next);}
20
21     void insert_this_before(node *n);
22     void insert_this_after(node *n);
23
24     void print() const {cout << value;}
25     operator value_type() const {return value;}   //explained in Chapter 3
26 };
27 #endif
```

The keyword `this` is used in lines 45, 48, 54, and 57. We saw it back in lines 101–105 of Version 3 on p. 117.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node/node.C`

```
#include <iostream>
#include <cstdlib>    //for exit
#include <cassert>    //for assert
#include "node.h"
using namespace std;

node::node(const value_type& initial_value)
{
    value = initial_value;
    next = prev = 0;
}

//Link the two nodes together.
//Sever n1 from its successor, if any.  Let n2 be the successor of n1.
//Sever n2 from its predecessor, if any.  Let n1 be the predecessor of n2.

void link(node *n1, node *n2)
{
    assert(n1 != n2 || n1 == 0);

    if (n1) {
        if (n1->next) {
            //Make sure n1->next->prev is correct before blowing it away.
            assert(n1->next->prev == n1);
            n1->next->prev = 0;
        }
        n1->next = n2;
    }

    if (n2) {
        if (n2->prev) {
            //Make sure n2->prev->next is correct before blowing it away.
            assert(n2->prev->next == n2);
            n2->prev->next = 0;
        }
        n2->prev = n1;
    }
}

//Insert this node into the list that contains n, immediately before or after n.

void node::insert_this_before(node *n)
{
    if (n) {
        link(n->prev, this);
    }

    link(this, n);
}

void node::insert_this_after(node *n)
{
```

```
    if (n) {
        link(this, n->next);
    }

    link(n, this);
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node/main.C`

```
#include <iostream>
#include <cstdlib>
#include "node.h"
using namespace std;

int main()
{
    node a(10);
    node b(20);
    node d(40);
    node e(50);

    link(&a, &b);
    link(&b, &d);
    link(&d, &e);

    //Insert c between b and d.
    node c(30);
    c.insert_this_before(&d);

    for (const node *p = &a; p; p = p->next) {
        p->print();
        cout << "\n";
    }

    cout << "\n";
    c.detach();

    for (const node *p = &a; p; p = p->next) {
        p->print();
        cout << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
10          lines 21–24
20
30
40
50

10          lines 29–32
20
40
50
```

A linked list is faster than an array for making insertions and deletions. Instead of moving many elements, an insertion merely had to make the two links in lines 45 and 48 of the above `node.C`. The price we pay for this speed is the two-pointer overhead attached to each list element. But nowadays we have memory to burn, don't we?

Now that we've made a linked list, we can reveal that a doubly-linked list class has already been written for us in the C++ Standard Library, just as a `stack` class has been provided. See it on pp. 443–450.

2.12 Enumerations as Members of a Class

Macros

Does line 8 construct "April 7" or "July 4"?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/macro/nomacro.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date d1( 7,  4, 1776);
 9     date d2(10, 29, 1929);
10     date d3(12,  7, 1941);
11     date d4( 7, 20, 1969);
12     date d5( 9, 11, 2001);
13
14     d1.print();
15     cout << "\n";
16
17     d2.print();
18     cout << "\n";
19
20     d3.print();
21     cout << "\n";
22
23     d4.print();
24     cout << "\n";
25
26     d5.print();
27     cout << "\n";
28
```

```
    return EXIT_SUCCESS;
}
```

```
7/4/1776
10/29/1929
12/7/1941
7/20/1969
9/11/2001
```

We can make it clearer with the 12 macros in lines 6–17. We use them not only as constructor arguments, but also in the comments and wherever we mention a month number.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/macro/date.h

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 #define JANUARY    1
 7 #define FEBRUARY   2
 8 #define MARCH      3
 9 #define APRIL      4
10 #define MAY        5
11 #define JUNE       6
12 #define JULY       7
13 #define AUGUST     8
14 #define SEPTEMBER  9
15 #define OCTOBER   10
16 #define NOVEMBER  11
17 #define DECEMBER  12
18
19 class date {
20     int year;
21     int month;   //JANUARY to DECEMBER inclusive
22     int day;     //1 to date_length[month] inclusive
23 public:
24     date(int initial_month, int initial_day, int initial_year);
25     void next(int count = 1);   //Go count days forward.
26     void print() const {cout << month << "/" << day << "/" << year;}
27 };
28 #endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/macro/date.C

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

const int date_length[] = {
     0,   //dummy element so that JANUARY will have subscript 1
    31,   //JANUARY
    28,   //FEBRUARY
```

```
    31,   //MARCH
    30,   //APRIL
    31,   //MAY
    30,   //JUNE
    31,   //JULY
    31,   //AUGUST
    30,   //SEPTEMBER
    31,   //OCTOBER
    30,   //NOVEMBER
    31    //DECEMBER
};

date::date(int initial_month, int initial_day, int initial_year)
{
    if (initial_month < JANUARY || initial_month > DECEMBER) {
        cerr << "bad month " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    if (initial_day < 1 || initial_day > date_length[initial_month]) {
        cerr << "bad day " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    year = initial_year;
    month = initial_month;
    day = initial_day;
}

void date::next(int count)
{
    div_t divide = div(count, 365);
    if (divide.rem < 0) {
        divide.rem += 365;
        --divide.quot;
    }

    year += divide.quot;
    day  += divide.rem;

    while (day > date_length[month]) {
        day -= date_length[month];
        if (++month > DECEMBER) {
            month = JANUARY;
            ++year;
        }
    }
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/macro/main.C`

```
#include <iostream>
```

```
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
    date d1(JULY,       4, 1776);
    date d2(OCTOBER,   29, 1929);
    date d3(DECEMBER,   7, 1941);
    date d4(JULY,      20, 1969);
    date d5(SEPTEMBER, 11, 2001);

    d1.print();
    cout << "\n";

    d2.print();
    cout << "\n";

    d3.print();
    cout << "\n";

    d4.print();
    cout << "\n";

    d5.print();
    cout << "\n";

    return EXIT_SUCCESS;
}
```

```
7/4/1776
10/29/1929
12/7/1941
7/20/1969
9/11/2001
```

Enumerations

Consecutively numbered macros are practical only if there are a small number of them. If we had many more, or if they often had to be inserted, deleted, or reordered, we would not want to renumber them by hand.

The *enumeration values* in lines 7–18 of `date.h` are like mass-produced, consecutively numbered macros. If we insert, delete, or reorder, they will automatically renumber themselves the next time we compile.

By default, the enumerations are numbered starting at zero; the `= 1` in line 7 starts the numbering at 1.

The `month_type` in line 6 is the name for a data type which can hold any of these enumeration values.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/enum/date.h`

```
1 #ifndef DATEH
2 #define DATEH
```

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

enum month_type {
    january = 1,
    february,
    march,
    april,
    may,
    june,
    july,
    august,
    september,
    october,
    november,
    december
};

class date {
    int year;
    int month;   //january to december inclusive
    int day;     //1 to date_length[month] inclusive
public:
    date(int initial_month, int initial_day, int initial_year);
    void next(int count = 1);   //Go count days forward.
    void print() const {cout << month << "/" << day << "/" << year;}
};
#endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/enum/date.C

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

const int date_length[] = {
     0,   //dummy element so that january will have subscript 1
    31,   //january
    28,   //february
    31,   //march
    30,   //april
    31,   //may
    30,   //june
    31,   //july
    31,   //august
    30,   //september
    31,   //october
    30,   //november
    31    //december
};

date::date(int initial_month, int initial_day, int initial_year)
{
```

```
    if (initial_month < january || initial_month > december) {
        cerr << "bad month " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    if (initial_day < 1 || initial_day > date_length[initial_month]) {
        cerr << "bad day " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    year = initial_year;
    month = initial_month;
    day = initial_day;
}

void date::next(int count)
{
    div_t divide = div(count, 365);
    if (divide.rem < 0) {
        divide.rem += 365;
        --divide.quot;
    }

    year += divide.quot;
    day  += divide.rem;

    while (day > date_length[month]) {
        day -= date_length[month];
        if (++month > december) {
            month = january;
            ++year;
        }
    }
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/enum/main.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
    date d1(july,       4, 1776);
    date d2(october,   29, 1929);
    date d3(december,   7, 1941);
    date d4(july,      20, 1969);
    date d5(september, 11, 2001);

    d1.print();
    cout << "\n";
```

```

    d2.print();
    cout << "\n";

    d3.print();
    cout << "\n";

    d4.print();
    cout << "\n";

    d5.print();
    cout << "\n";

    return EXIT_SUCCESS;
}
```

Enumeration arithmetic

We can convert an enumeration to an `int` implicitly, but in the opposite direction we must write a cast.

```
1     int i = 10;
2     month_type m = january;
3
4     i = m;                          //convert enum to int
5     m = static_cast<month_type>(i); //convert int to enum
```

The cast makes the conversion easy to find if anything goes wrong. And something could potentially go wrong: the `int` could easily be too big to fit in the enumeration. On the other hand, an enumeration will (almost always) fit in an `int`.

Incidentally, there's a simpler way to write the above line 5. Although an enumeration is not an object, we can pretend it has a constructor taking one argument. Instead of creating the enumeration with a cast, we can create it with a constructor.

```
6     m = month_type(i);              //convert int to enum
```

To avoid the enumeration cast or constructor, we kept the data type `int` for the first argument of the constructor for the above class `date`. This permits both of the following.

```
7     date d1(7,    4, 1776);         //first argument is an int
8     date d2(july, 4, 1776);         //first argument is an enumeration
```

The data member `month` also remained an `int` so that line 54 of the above `date.C` could still say `++month`. We won't be able to increment an enumeration until we do operator overloading.

Enumeration members of a class

If the name `july` were already taken by the enumeration in line 13 of the above `date.h`, we could not have the global variable `july` in line 7 of the following `main.C`. This would be an example of *name space pollution.*

To avoid pollution, our enumerations will now be members of class `date`. This will let us have a global variable or a global function named `july`. In the same way, we also have two `print` functions: the member function `print` and the global function `print` in line 6 of `main.C`.

Inside the body of a member function of class `date`, we don't write anything in front of a member of class `date`. We can therefore keep the `date.C` file from the previous example. In it, for example, the `month` and `january` in line 55 will now be members of class `date`.

But outside the body of a member function of class `date`, a member of class `date` must always be preceded by something to tell the computer which object it belongs to. For example, `july` and `print` are members of the object `d1` in lines 17 and 23 of `main.C`.

If the member has the same value for every object of the class, we can write the name of the class with a double colon in front of the member, rather than the name of an object with a dot. For example, the enumeration member `july` in lines 17 and 18 of `main.C` will always have the value 7 in every object of class `date`, so we write line 19 to avoid accusations of favoritism. Another advantage of 19 is that it can be executed even when no `date` objects exist. (A static data member will also have the same value for every object of the class. See p. 241.) On the other hand, the data member `month` can have a different value in different objects of class `date`, and the data members used in the member function `print` can have different values in different objects of class `date`.

If there is no object name or class name in front of an identifier outside the body of a member function, the compiler assumes that it is the name of something that is not a member of any class. Examples are in lines 20 and 22 of `main.C`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/enummember/date.h`

```
#ifndef DATEH
#define DATEH
#include <iostream>
using namespace std;

class date {
    int year;
    int month;   //date::january to date::december inclusive
    int day;     //1 to date_length[month] inclusive
public:
    enum month_type {
        january = 1,
        february,
        march,
        april,
        may,
        june,
        july,
        august,
        september,
        october,
        november,
        december
    };

    date(int initial_month, int initial_day, int initial_year);
    void next(int count = 1);   //Go count days forward.
    void print() const {cout << month << "/" << day << "/" << year;}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/enummember/date.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;
```

```
const int date_length[] = {
     0,   //dummy element so that january will have subscript 1
    31,   //january
    28,   //february
    31,   //march
    30,   //april
    31,   //may
    30,   //june
    31,   //july
    31,   //august
    30,   //september
    31,   //october
    30,   //november
    31    //december
};

date::date(int initial_month, int initial_day, int initial_year)
{
    if (initial_month < january || initial_month > december) {
        cerr << "bad month " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    if (initial_day < 1 || initial_day > date_length[initial_month]) {
        cerr << "bad day " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    year = initial_year;
    month = initial_month;
    day = initial_day;
}

void date::next(int count)
{
    div_t divide = div(count, 365);
    if (divide.rem < 0) {
        divide.rem += 365;
        --divide.quot;
    }

    year += divide.quot;
    day  += divide.rem;

    while (day > date_length[month]) {
        day -= date_length[month];
        if (++month > december) {
            month = january;
            ++year;
        }
    }
```

```
59 }
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/enummember/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 inline void print() {cout << "the global print\n";}
 7 const double july = 80.5;   //Fahrenheit
 8
 9 int main()
10 {
11     date d1(date::july,       4, 1776);
12     date d2(date::october,   29, 1929);
13     date d3(date::december,   7, 1941);
14     date d4(date::july,      20, 1969);
15     date d5(date::september, 11, 2001);
16
17     cout << d1.july << "\n"     //the member july
18         << d2.july << "\n"      //the member july
19         << date::july << "\n"   //better way to say the member july
20         << july << "\n";        //the non-member july in line 7
21
22     print();      //the non-member print in line 6
23     d1.print();   //the member print
24     cout << "\n";
25
26     d2.print();
27     cout << "\n";
28
29     d3.print();
30     cout << "\n";
31
32     d4.print();
33     cout << "\n";
34
35     d5.print();
36     cout << "\n";
37
38     return EXIT_SUCCESS;
39 }
```

```
7               line 17
7               line 18
7               line 19
80.5            line 20
the global print
7/4/1776
10/29/1929
12/7/1941
7/20/1969
9/11/2001       line 38
```

Now that `march`, `may`, and `august` are members of class `date`, we can create members named `march`, `may`, and `august` of other classes. (See pp. 1024−1025 for a better way to give them last names.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/enummember/main2.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

class music {
public:
    enum {
        waltz,
        march,
        tango
    };
};

class verb {
public:
    enum {
        will,
        shall,
        can,
        may
    };
};

class cape {
public:
    enum {
        canaveral,   //Florida
        cod,         //Massachusetts
        may          //New Jersey
    };
};

class adjective {
public:
    enum {
        majestic,
        sublime,
        august
```

```
    };
};

int main()
{
    cout << date::march << "\n"
        << music::march << "\n\n"

        << date::may << "\n"
        << verb::may << "\n"
        << cape::may << "\n\n"

        << date::august << "\n"
        << adjective::august << "\n";

    return EXIT_SUCCESS;
}
```

```
3
1

5
3
2

8
2
```

2.13 Arrays of Objects

An array of integers

An *array* is a group of variables at equally spaced addresses. The variables are called the *elements* of the array. An array element must therefore be a variable that has a memory address. This includes objects and pointers, but not references: a reference has no address of its own. See p. 80.

It is only fair to warn you that arrays will shortly be superseded by vectors. A vector will have the look and feel of an array (e.g., the `[`square brackets`]`) but without its drawbacks. Vectors will have to wait, however, until we do "operator overloading".

Before constructing an array of objects, we will make an array of integers and an array of structures. The definition in lines 7–20 creates an array of 12 integers. It is meaningless to ask what order they are created in: nothing happens when an integer is born, so there is no experiment we could perform to produce an observable effect showing us the order. The question belongs to the realm of metaphysics.

Use the data type `size_t` for a variable that holds an array subscript (line 23) or the number of elements in an array (line 21). See p. 66. The `sizeof / sizeof` idiom in line 21 was last seen in line 33 of `wolf.C` on p. 198.

But don't use the `size_t i` loop in lines 23–25. On some platforms, the pointer `p` loop in lines 29–31 will run faster. It is also safer. If the `const` were removed from line 7, the array could be damaged by an expression such as `++a[i]` but could not be damaged by `++*p`. The latter expression would not compile because `p` is a read-only pointer.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/array/int.C`

```
#include <iostream>   //includes cstddef, where size_t is defined
#include <cstdlib>
using namespace std;

int main()
{
    const int a[] = {
        31,   //January
        28,   //February, ignoring for now the possibility of leap year
        31,   //March
        30,   //April
        31,   //May
        30,   //June
        31,   //July
        31,   //August
        30,   //September
        31,   //October
        30,   //November
        31    //December
    };
    const size_t n = sizeof a / sizeof a[0];

    for (size_t i = 0; i < n; ++i) {
        cout << a[i] << "\n";
    }

    cout << "\n";

    for (const int *p = a; p < a + n; ++p) {
        cout << *p << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
31
28
31          etc.
```

An array of structures

The definition in lines 12–25 creates an array of 12 structures, each containing an integer and a pointer. Again, it is meaningless to ask what order the structures are created in. Nothing happens when a structure containing an integer and pointer is born.

In the expression `a[i].length` in line 29, the two operators have equal precedence and left-to-right associativity. The `[]` executes before the dot:

$$\boxed{\boxed{\boxed{\texttt{a}}\ \texttt{[}\ \boxed{\texttt{i}}\ \texttt{]}}\ .\ \boxed{\texttt{length}}}$$

This is exactly the order we need for an array of structures.

(1) Since `a` is an array, we apply a `[]` operator to it to delve into it and get the element we want.

(2) Since the element is a structure, we apply a dot operator to it to delve into it and get the field we want.

But don't write the `size_t i` loop in lines 28−30. The pointer `p` loop in lines 34−36 executes faster on some platforms, and is safer because the pointer is read-only. Most importantly, the `p->length` in line 35 is simpler than the `a[i].length` in 29.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/array/structure.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 struct month {
 6     int length;
 7     const char *name;
 8 };
 9
10 int main()
11 {
12     const month a[] = {          //C++ doesn't need the keyword struct here.
13         {31,   "January"},
14         {28,   "February"},
15         {31,   "March"},
16         {30,   "April"},
17         {31,   "May"},
18         {30,   "June"},
19         {31,   "July"},
20         {31,   "August"},
21         {30,   "September"},
22         {31,   "October"},
23         {30,   "November"},
24         {31,   "December"}
25     };
26     const size_t n = sizeof a / sizeof a[0];
27
28     for (size_t i = 0; i < n; ++i) {
29         cout << a[i].length << "\t" << a[i].name << "\n";
30     }
31
32     cout << "\n";
33
34     for (const month *p = a; p < a + n; ++p) {
35         cout << p->length << "\t" << p->name << "\n";
36     }
37
38     return EXIT_SUCCESS;
39 }
```

```
31  January
28  February
31  March                etc.
```

An array of objects

The definition in lines 8−16 creates an array of seven objects. When an object is born a constructor is called, possibly with an observable effect such as the production of output. When we have an array of objects, it therefore becomes meaningful to ask what order the elements are created in. The array definition calls the constructors for the seven objects `a[0]` through `a[6]`, in that order. Line 31 calls their destructors in the reverse order, or at least it would have if class `date` had a destructor.

It's too bad that we have to write the constructor's name (`date`), and the parentheses around its argument list, in lines 9−15. From now on, we will assume that class `date` has the 12 public enumerations, so will feel free to use them as arguments of the constructors. Lines 14−15 demonstrate that we don't have to call the same constructor for each element of the array.

In the expression `a[i].print()` in line 20, the three operators have equal precedence and left-to-right associativity:

$$\boxed{\boxed{\boxed{\boxed{\texttt{a}}\ [\ \boxed{\texttt{i}}\ ]}\ .\ \boxed{\texttt{print}}}\ ()}$$

This is exactly the order we need for an array of objects.

(1) Since `a` is an array, we apply a `[]` operator to it to delve into it and get the element we want.

(2) Since the element is an object, we apply a dot operator to it to delve into it and get the member we want.

(3) Since the member is a function, we apply the `()` operator to it to call it.

But don't write the `size_t i` loop in lines 19−22. The pointer `p` loop in lines 26−29 executes faster on some machines, and is safer because the pointer is read-only. Most importantly, the `p->print()` in line 27 is simpler than the `a[i].print()` in 20.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/array/object.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const date a[] = {
 9         date(date::july,       4, 1776),   //Call the 3-argument constructor.
10         date(date::october,   29, 1929),
11         date(date::december,   7, 1941),
12         date(date::july,      20, 1969),
13         date(date::september, 11, 2001),
14         date(),                            //Call the default constructor.
15         date(a[5])                         //Call the copy constructor.
16     };
17     const size_t n = sizeof a / sizeof a[0];
18
19     for (size_t i = 0; i < n; ++i) {
20         a[i].print();
21         cout << "\n";
22     }
23
24     cout << "\n";
```

```
25
26     for (const date *p = a; p < a + n; ++p) {
27         p->print();
28         cout << "\n";
29     }
30
31     return EXIT_SUCCESS;
32 }
```

```
7/4/1776        lines 19–22
10/29/1929
12/7/1941
7/20/1969
9/11/2001
4/8/2014
4/8/2014

7/4/1776        lines 26–29
10/29/1929
12/7/1941
7/20/1969
9/11/2001
4/8/2014
4/8/2014
```

Call a one-argument constructor for an array element

When calling a one-argument constructor for an array element, we don't have to write the name of the constructor and the parentheses around the object. Class `obj` was on pp. 179–180.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/array/one_arg.C`

```
#include <iostream>
#include <cstdlib>
#include "obj.h"
using namespace std;

int main()
{
    obj a[] = {
        obj(10),   //Can write name of constructor and parentheses...
        obj(20),
        obj(30)
    };
    const size_t na = sizeof a / sizeof a[0];

    obj b[] = {
        40,        //...but don't have to when constructor has 1 arg.
        50,
        60
    };
    const size_t nb = sizeof b / sizeof b[0];

    for (size_t i = 0; i < nb; ++i) {
```

```
23         b[i].print();
24         cout << "\n";
25     }
26
27     return EXIT_SUCCESS;
28 }
```

```
construct 10
construct 20
construct 30
construct 40
construct 50
construct 60
40
50
60
destruct 60
destruct 50
destruct 40
destruct 30
destruct 20
destruct 10
```

Line 15 of the above `object.C` may therefore be written as

```
29         a[5]
```

Call the default constructor for every array element

A *default constructor* is one that can be called with no arguments, either because it has no arguments at all or because it has a default value for every argument. See pp. 134−135.

Here are constructors with no arguments at all:

(1) the constructor for class `zero` on pp. 134−135

(2) the constructor for class `date` on pp. 142−143

(3) the constructor for class `stack` on pp. 149−154

Here are constructors with a default value for every argument:

(4) the constructor for class `terminal` on pp. 157−163

(5) the constructor for class `point` on pp. 201−204

All of the above are default constructors.

The definition in line 9 calls the default constructor for each array element. It will compile only if there is a default constructor. When providing no explicit initial values, we must write the number of elements in the `[`square brackets`]` in line 9.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/array/no_arg.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "date.h"
4 using namespace std;
5
6 int main()
7 {
```

```
    const size_t n = 3;
    const date a[n];   //Call the default constructor n times.

    for (const date *p = a; p < a + n; ++p) {
        p->print();
        cout << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
4/8/2014                 three copies of today's date
4/8/2014
4/8/2014
```

C does not allow us to use the value of any variable in an expression declaring the number of elements in an array.

```
18     /* C example */
19     const size_t n = 10;
20     int a[n];    /* won't compile, even though n was const */
```

In C, the `n` in the above line 20 would have to be changed to a literal `10`, or to an expression whose value does not depend on the value of any variable or on the return value of any function. A macro for a legal expression could also be used.

C++ lets us write an array dimension as an expression whose value can depend on the value of a variable, provided that the expression can be evaluated when the program is compiled. This is called a *constant expression.* If the value of a constant expression depends on the value of a variable, the variable must be `const` and its initial value must in turn be a constant expression.

```
21     //C++ example
22     size_t n1 = 10;
23     const size_t n2 = n1;
24     const size_t n3 = 10 + 20;
25     inline size_t f() {return 10;}
26
27     int a1[n1];  //won't compile: n1 was not const
28     int a2[n2];  //won't compile: initial val of n2 was not a const expr
29     int a3[n3];  //will compile
30     int a4[f()]; //won't compile: can't use return value of function
```

The above line 30 just happens to compile with the GNU compiler `g++`, but it shouldn't.

Explicit initial values are copied into an array

We have already seen an array of objects with explicit initial values.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/array/copy.C`

```
#include <iostream>
#include <cstdlib>
#include "obj.h"
using namespace std;

int main()
{
```

```
 8     obj a[] = {10, 20, 30};
 9     return EXIT_SUCCESS;
10 }
```

`construct 10`	*Did not call copy constructor.*
`construct 20`	
`construct 30`	
`destruct 30`	*Line 9 destructs* `a`.
`destruct 20`	
`destruct 10`	

But the computer is within its rights if it creates objects outside of the array, and then copies them into the array by calling the copy constructor for each element. With an older compiler, or with the `-fno-elide-constructors` option of the `g++` compiler, the output is the following. See p. 137 for a non-array example of the compiler exercising this right.

`construct 10`	*Construct an object outside the array.*
`copy construct 10`	*Copy it into the first array element.*
`destruct 10`	*Destruct the object outside the array.*
`construct 20`	
`copy construct 20`	
`destruct 20`	
`construct 30`	
`copy construct 30`	
`destruct 30`	
`destruct 30`	*Line 9 destructs* `a`.
`destruct 20`	
`destruct 10`	

We must therefore be allowed to call the copy constructor for the objects in an array with explicit initial values. This is true even if the compiler is smart enough to avoid the actual calls to the copy constructor.

▼ Homework 2.13a:
Version 1.3 of the Rabbit Game: array of rabbits

Remove the rabbit `r` in line 19 of `main.C` on p. 194. In its place, define an explicitly initialized array of at least three non-`const` rabbits like the explicitly initialized array of `date`'s in lines 8–16 of `object.C` on p. 231. Name the array `a`. Initialize each rabbit to a different position, making sure that the `x`, `y` arguments of the constructor of each `rabbit` are on the screen. After defining the array, use the `sizeof / sizeof` idiom to count the array elements. Store this number in a constant named `n`.

Remove the declaration for the undefined copy constructor for class `rabbit` you wrote on p. 200. The computer will now provide us with a copy constructor, allowing the array definition to compile. But let's hope that the compiler is smart enough to avoid calling this copy constructor. We don't want to duplicate the `rabbit`'s.

Change the main loop in lines 21–28 of `main.C` on p. 194 to the following. The game will now end as soon as the `wolf` kills any `rabbit`.

```
    for (;; term.wait(250)) {
        if (!w.move()) {
            goto done;
        }

        for (let p be a read/write pointer to each rabbit in the array) {
            if (!p->move()) {
```

```
 8                 goto done;
 9             }
10         }
11     }
12
13     done:;
14     continue with lines 30ff. on p. 194
```

Change the message to "You killed *a* rabbit!", since there is now more than one of them.

It's too bad that the above lines 2–4 duplicate lines 7–9. But for the time being, the duplication has to remain: the wolf can't be an element in an array of rabbits, so it has to be moved separately. When we do inheritance, a `wolf` and a `rabbit` will become, in some sense, the same species of animal (a "`wabbit`"), and will then be able to share the same array.

▲

▼ Homework 2.13b: platform dependent output

In how many ways can a C++ program legally produce different output on different platforms? Consider expressions whose operators can be executed in different orders; global objects in separate source files; temporaries that can be elided when initializing an object or an array of objects; and objects returned from a function via pass-by-value.

Is the data type `char` signed or unsigned on your machine? Can you write a program where this affects the output? What happens if you try to store a value into a signed integral variable that is too small to hold it? Does a pointer print in hex or octal on your machine?

```
int i = 10;
cout << &i << "\n";
```

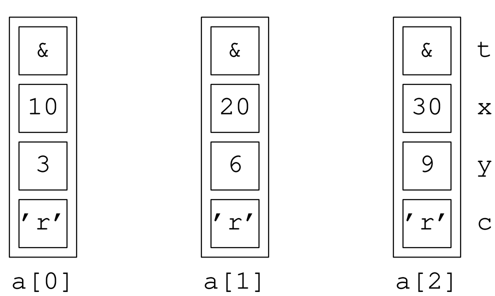
▲

2.14 Static Members

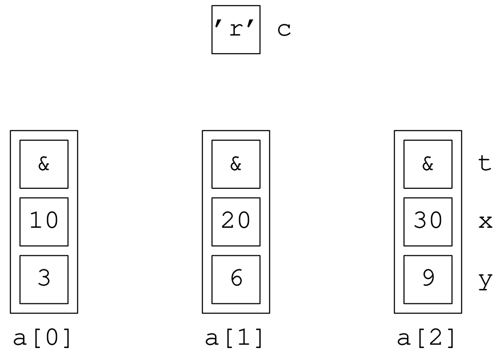
2.14.1 Static Data Members

Here is a picture of three rabbits in an array. Each rabbit has different values in its `x` and `y` data members: we do not allow two rabbits in the same place at the same time. An ampersand represents a pointer value. Although the rabbits have pointers to the same terminal now, they will have different pointers when we have multiple terminals.

But every rabbit has the *same* value (`'r'`) in its `c` data member. This was a non-issue when there was only one rabbit. But now that we have an array of them, we are wasting memory:



We can save memory by making all the rabbits share just one copy of `c`. It will not be physically inside of any rabbit, but will still be a private data member of the class, entitled to all the privileges and protection pertaining thereto. This kind of member is called a *static* data member.



No matter how many rabbits exist at a given moment, there will always be exactly one copy of the static data member `c`. Even after all of the rabbits have been destructed, `c` will persist until the end of the program. `c` will also be there before the first rabbit is constructed, and, even weirder, will exist even if no rabbit is *ever* constructed. A static data member is as immortal as a global variable.

This means that the constructor for class rabbit is relieved of its normal responsibility for initializing `c`, so we will have to remove the statement `c = 'r';` from line 11 of `rabbit.C` on p. 196.

Line 7 declares a static data member. The declaration is the only place where we write the keyword `static` when creating this kind of member. This keyword means "static member" only when written within the `{`curly braces`}` of a class declaration. Outside of a class declaration, it has the same meanings it had in C.

It does not matter whether the declaration comes before or after the non-static data members. In either case, the static data member will be created long before the others. But declare it first because it is created first. (If there were two or more static data members, they would be created in the order they were declared in.)

```
1 //Excerpt from the file rabbit.h.
2 #ifndef RABBITH
3 #define RABBITH
4 #include "terminal.h"
5
6 class rabbit {
7     static const char c;   //declaration
8     const terminal *t;
9     unsigned x, y;
```

But we must do more than just declare `c`. We must also define it, i.e., create and initialize it. Do this in line 16, above the function definitions in the `rabbit.C` file. Do not repeat the keyword `static` here.

```
10 //Excerpt from the file rabbit.C.
11 #include <iostream>
12 #include <cstdlib>
13 #include "rabbit.h"
14 using namespace std;
15
16 const char rabbit::c = 'r';   //definition
17
18 rabbit::rabbit(const terminal& initial_t, unsigned initial_x, unsigned initial_y)
19 {
20     t = &initial_t;
21     x = initial_x;
22     y = initial_y;
23     //Do not initialize c here; c has already been initialized.
```

The definition in the above line 16 consists of the same three parts as a simpler definition such as `int i = 10;`.

name of data type	*name of variable*	*initial value*
`int`	`i`	`= 10;`
`const char`	`rabbit::c`	`= 'r';`

Newer versions of C++ let us declare and define a static data member in a single line. But this notation can be used only if the member is also constant, and integral (p. 61) or an enumeration, and if its initial value is a constant expression (p. 234).

Since our data member `c` and its initial value meet all the requirements, we can declare and define it in one statement in line 30, removing the above line 16.

```
24 //Excerpt from the file rabbit.h.
25 #ifndef RABBITH
26 #define RABBITH
27 #include "terminal.h"
28
29 class rabbit {
30     static const char c = 'r';      //declaration and definition
31     const terminal *t;
32     unsigned x, y;
```

▼ Homework 2.14.1a:
Version 1.4 of the Rabbit Game: static data member for classes `wolf` and `rabbit`

Let the `c` data member of class `rabbit` be static. Do the same for the `c` data member of class `wolf`, even though there is currently only one `wolf`. Use the above line 30 if you can; lines 7 and 16 if you must. Now that `c` is static, declare it before the other data members.

We currently have only one terminal, so we could make the `t` data members static too. But don't do this. We would only have to undo it when we have multiple terminals.

▲

▼ Homework 2.14.1b: another way to think of a static data member

We've seen several variables that we haven't known where to put. Each one was floating, somewhat unsatisfactorily, near an associated class.

(1) `date_length` floating near class `date` in lines 5–19 of `version3.C` on p. 115;

(2) `life_ymax` and `life_xmax` floating near class `life` in lines 5–6 of `life.h` on pp. 145–146;

(3) `stack_max_size` floating near class `stack` in line 5 of `stack.h` on p. 148;

(4) `point_error` and `point_origin` floating near class `point`, in line 7 of `point.h` on p. 206 and line 5 of `point.C` on p. 202.

Each variable had no official connection with its class. It was not a data member; it merely had the name of the class and an underscore prefixed to its own name.

For security, we would like the variable to be private data member of its associated class. And now we have a way to do this. We can let the variable be a static private data member.

Make the following changes, except for class `life`.

(1) Let `date_length` be a private static data member of class `date` and rename it `length`. You'll have to write the number of elements in the `[square brackets]` of the declaration. Write this number as `12 + 1`, rather than `13`, to make the magic number `12` visible.

Since an array is not an integral data type, you won't be able to define it in the class declaration in the `.h` file. Define it in the `date.C` file.

Now that `length` is a private data member, observe that we can change it with no repercussions beyond the member functions of the class. For example, we could remove the dummy element from the start of the array. But just observe this—don't do it.

(2) We will eventually let `life_xmax` and `life_ymax` be private static data members of class `life`, renaming them `xmax` and `ymax`. They must be initialized before being used by the array declaration in line 4. This means that they can be static data members only in versions of C++ that permit the initialization in line 30 on p. 238.

```
 1 class life {
 2     static const size_t xmax = 10;
 3     static const size_t ymax = 10;
 4     bool matrix[ymax][xmax];
 5     //etc.
```

But even if the initializations are permitted, we can't let `xmax` and `ymax` be static data members yet. The problem is that they must also be initialized before being used by the typedefs for `life_matrix_t` and `_life_matrix_t`, which currently must be written *before* the declaration for class `life`.

```
 6 const size_t life_xmax = 10;
 7 const size_t life_ymax = 10;
 8
 9 typedef bool  life_matrix_t[life_ymax][life_xmax];
10 typedef bool _life_matrix_t[life_ymax + 2][life_xmax + 2];  //internal use only
11
12 class life {
13     _life_matrix_t matrix;
14     //etc.
```

The eventual solution will be to move the typedef into the class itself (pp. 423–424).

```
15 class life {
16     static const size_t xmax = 10;
17     static const size_t ymax = 10;
18     typedef bool _life_matrix_t[ymax + 2][xmax + 2];    //private typedef
19     _life_matrix_t matrix;
20 public:
21     typedef bool life_matrix_t[ymax][xmax];             //public typedef
22     life(const life_matrix_t initial_matrix);
```

But don't do this yet.

(3) Let `stack_max_size` be a private static data member of class `stack`, and rename it `max_size`. But `stack_max_size` is used as the dimension of an array, so `stack_max_size` must have a value *before* the array is declared. Therefore `stack_max_size` can be a static data member only in versions of C++ that permit the initialization in line 30 on p. 238

```
23 class stack {
24     static const size_t max_size = 100;
25     value_type a[max_size];
```

(4) Let `point_error` and `point_origin` be private static data members of class `point`, and rename them `error` and `origin`. The member function `dist` with no explicit arguments can now become inline; see p. 210. Note that it would be impossible for class `point` to have a `point` as a non-static data member—a `point` object would blow up to infinite size—but class `point` can have a `point` as a static data member.

▲

▼ Homework 2.14.1c: another static data member for class date

Let the following array be a private static data member of class `date`. In the initialization for a static data member, we can mention the names of other static data members, even private ones.

```
//Excerpt from date.C.

//Number of days in the year before each month.  For example, pre[3] is
//59 == 31 + 28, because there are 59 days in the year before March 1.

const int date::pre[] = {
    0,                      //dummy element to give january subscript 1
    0,                      //january
    pre[ 1] + length[ 1],   //february
    pre[ 2] + length[ 2],   //march
    pre[ 3] + length[ 3],   //april
    pre[ 4] + length[ 4],   //may
    pre[ 5] + length[ 5],   //june
    pre[ 6] + length[ 6],   //july
    pre[ 7] + length[ 7],   //august
    pre[ 8] + length[ 8],   //september
    pre[ 9] + length[ 9],   //october
    pre[10] + length[10],   //november
    pre[11] + length[11]    //december
};
```

If your class `date` is implemented with three data members, let the `julian` member function be a single statement.

```
    return pre[month] + day;
```

The function can now be inline for even more speed.

If your `date` is implemented with one data member, get rid of the loop in the constructor for the one-data-member class `date`.

```
    day = 365 * initial_year + pre[initial_month] + initial_day - 1;
```

▲

▼ Homework 2.14.1d: another static data member for class date

It's expensive to get the current date and time from the operating system whenever we call the default constructor for class `date`. Remedy tis by declaring the the following private member for class `date`.

```
1     static const date today;
```

Initialize it as follows in the `date.C` file. The keyword `static` in lines 2 and 3 is the C keyword that means "visible only in this file". The static data member in line 4 should not have the keyword `static`.

```
2 static const time_t t = time(0);
3 static const tm *const p = localtime(&t);
4 const date date::today(p->tm_mon + 1, p->tm_mday, p->tm_year + 1900);
```

Then the default constructor can simply copy the data member(s) of `today` into the newborn `date`. Let's hope the program doesn't keep running across a midnight, though.

▲

▼ Homework 2.14.1e: find the bug

Let's say we have a program that constructs and destructs objects of the following class. At any point in time, the static data member `_count` in line 7 should be the number of `counted` objects that exist. After all, it's initialized to zero in line 3 of `counted.C`, incremented in the constructor in line 9 of `counted.h`, and decremented in the destructor in line 10 of `counted.h`.

`_count` has an underscore because a public member function named `count` will be introduced in the next example, and a class cannot have a data member and a member function with the same name (p. 159). The burden of the underscore is placed on the private member.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/counted1/counted.h`

```
 1 #ifndef COUNTEDH
 2 #define COUNTEDH
 3
 4 class counted {
 5     int i;
 6 public:
 7     static unsigned _count;  //data member temporarily public for simplicity
 8
 9     counted(int initial_i) {i = initial_i; ++_count;}
10     ~counted() {--_count;}
11 };
12 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/counted1/counted.C`

```
1 #include "counted.h"
2
3 unsigned counted::_count = 0;
```

Within the {curly braces} of the class declaration in lines 4–11 of the above `counted.h`, and within the body of a member function of class `counted`, we're on a first-name basis with all the members of class `counted`, including the static data member `_count`. But elsewhere we must write something in front of `_count` to indicate which class it is a member of. Lines 21–23 are three ways to do this, since `_counted` has the same value for any object of class `counted`. But these lines are annoyingly arbitrary. Why select the object `a` in line 21? What's wrong with `b`?

We therefore prefer line 24. Write the name of the class and a double colon instead of the name of an arbitrarily selected object of the class and a dot. Use this notation for any member that has the same value for every object of the class. The static member `counted::_count` is one example; another is the enumeration member `date::july` in line 19 of `main.C` on p. 226.

Another advantage of line 24 is that it will work even if no objects of the class are ever constructed. Lines 21–23 will work only if objects `a`, `b`, and `c` exist at that point.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/counted1/main.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "counted.h"
4 using namespace std;
5
6 void f();
7
8 int main()
9 {
```

```
10     f();
11     cout << "There are " << counted::_count << " objects.\n";
12     return EXIT_SUCCESS;
13 }
14
15 void f()
16 {
17     counted a = 10;
18     counted b = 20;
19     counted c = b;
20
21     cout << "There are " << a._count << " objects.\n"
22          << "There are " << b._count << " objects.\n"
23          << "There are " << c._count << " objects.\n"
24          << "There are " << counted::_count << " objects.\n";
25 }
```

`There are 2 objects.`	*Lines 21−24: there are actually 3 objects at this point.*
`There are 2 objects.`	
`There are 2 objects.`	
`There are 2 objects.`	
`There are 4294967295 objects.`	*Line 11: there are actually 0 objects at this point.*

The above line 19 calls a copy constructor for class `counted`, but it's not any copy constructor that we wrote. The computer behaves as if we had written the following copy constructor in `counted.h` and called it in line 19. See p. 135.

```
26 public:
27     counted(const counted& another) {i = another.i;}
```

But the copy constructor in the above line 27 isn't good enough: it forgot to increment `_count`. We therefore have to write the copy constructor ourselves:

```
28 public:
29     counted(const counted& another) {i = another.i; ++_count;}
```

▲

2.14.2 Static Member Functions

The data member `_count` in the previous example should never have been public. The following line 5 now makes it private. The public is granted read-only access to it via the public member function `count` in line 12. (See p. 159 for another example.) A class can't have a data member and a member function with the same name, so we gave an underscore to the member that is never mentioned in the outside world.

Until now, every member function has received an implicit argument pointing to the object to which the member function belongs. This invisible pointer lets the member function access a member of the object simply by mentioning its name. Our original example was the `print` member function of the class `date` in lines 99−107 of `version3.C` on pp. 116−117.

But there is one kind of member function, called a *static member function* that receives no implicit pointer argument. The member function `count` in line 12 is static to avoid burdening it with an implicit pointer that it does not need. Since the data member `_count` in line 12 is not in any object, the function needs no pointer to access it. Although it is a member function, a static member function is a free function (p. 113) because it receives no implicit pointer argument.

Line 12 demonstrates that a static member function can access a static member simply by mentioning its name. But a static member function, like a friend function, would not compile if it tried to access a non-static member simply by mentioning its name. We would have to write something in front of the member to indicate which object it belonged to.

A `const` member function cannot change the data members of the object it belongs to, i.e., the object to which it receives an invisible pointer. Once again, our original example was `date::print`. But a `static` member function does not belong to any object. It would therefore be meaningless (and illegal) to make it `const`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/counted2/counted.h`

```
 1 #ifndef COUNTEDH
 2 #define COUNTEDH
 3
 4 class counted {
 5     static unsigned _count;
 6     int i;
 7 public:
 8     counted(int initial_i) {i = initial_i; ++_count;}
 9     counted(const counted& another) {i = another.i; ++_count;} //copy constructor
10     ~counted() {--_count;}
11
12     static unsigned count() {return _count;}
13 };
14 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/counted2/counted.C`

```
#include "counted.h"

unsigned counted::_count = 0;
```

As in the above `main.C`, we prefer the following line 24 to the three above it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/counted2/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "counted.h"
 4 using namespace std;
 5
 6 void f();
 7
 8 int main()
 9 {
10     f();
11     cout << "There are " << counted::count() << " objects.\n";
12     return EXIT_SUCCESS;
13 }
14
15 void f()
16 {
17     counted a = 10;
18     counted b = 20;
```

```
19     counted c = b;
20
21     cout << "There are " << a.count() << " objects.\n"
22          << "There are " << b.count() << " objects.\n"
23          << "There are " << c.count() << " objects.\n"
24          << "There are " << counted::count() << " objects.\n";
25 }
```

```
There are 3 objects.     lines 21−24
There are 3 objects.
There are 3 objects.
There are 3 objects.
There are 0 objects.     line 12
```

▼ Homework 2.14.2a:
Version 1.5 of the Rabbit Game: static member functions for class `terminal`

The `beep` member function (line 29 of `terminal.h` on p. 160) receives an implicit pointer to an object of class `terminal`. But the pointer is never used: `beep` does not mention any non-static member of the object. We can therefore let `beep` be static, getting rid of the pointer and making the function faster.

Here is how to identify the member functions that must be non-static.

(1) A constructor and destructor must be non-static.

(2) A member function must be non-static if it uses the pointer `this`, explicitly or implicitly. Two examples are in in `version3.C` on p. 117. Line 106 accesses the non-static data members of the object to which the implicit `this` points; line 105 does the same thing with an explicit `this`. The member function `print` in line 99 must therefore be non-static. In the same program, line 83 calls a non-static member function of the object to which the implicit `this` points; line 82 does the same thing with an explicit `this`. The member function `next` in line 74 must therefore be non-static.

(3) All the other member functions can, and therefore should, be static.

Let every possible member function of class `terminal` be static. Note that a static member function cannot be `const`. `const` would mean that the function receives an implicit pointer that is read-only. But a static member function receives no implicit pointer at all. You must therefore let your newly static member functions be non-`const`.

▲

The choice between a friend and a static member function

I gave the same definition for a friend and a static member function: both are just like a normal (non-static) member function, except that they receive no invisible pointer. The next section will show that there actually is a slight difference between them. But for now, any friend could easily be rewritten as a static member function and vice versa. They are equal in space and speed.

The choice between them is important, however, because it shows the *intent* of the function.

(1) If the function operates on more than one object, write it as a friend; see p. 206, ¶ (3). An example is the function `average` declared in line 20 and defined in 33−36. Line 34 shows that a friend can mention a static member, but must indicate which class the member belongs to. In a friend defined in the class declaration (lines 15−18), this indication is not necessary.

(2) If the function does not operate on objects at all, write it as a static member function. The functions `count` and `exist` in lines 22 and 25 access only members that are static, i.e., not in any object of the class.

(3) If the function operates on exactly one object, or if one object plays the starring rôle, write it as a non-static member function. An example is the function `print` in lines 27−30. Line 28 shows that a non-static member function can mention a static member, but we already knew this from lines 11−13. A

`_count` needs no preliminary `counted::` inside the body of a member function (line 28) or anywhere inside the curly braces of the class declaration in lines 7 and 31 (line 16). But outside of these places, the `counted::` is needed (line 34).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/friend_vs_static/counted.h`

```
 1 #ifndef COUNTEDH
 2 #define COUNTEDH
 3 #include <iostream>
 4 #include <cassert>
 5 using namespace std;
 6
 7 class counted {
 8     static unsigned _count;
 9     int i;
10 public:
11     counted(int initial_i) {i = initial_i; ++_count;}
12     counted(const counted& another) {i = another.i; ++_count;}
13     ~counted() {--_count;}
14
15     friend bool equal(const counted& c1, const counted& c2) {
16         assert(_count > 0);
17         return c1.i == c2.i;
18     }
19
20     friend int average(const counted& c1, const counted& c2);
21
22     static unsigned count() {return _count;}
23
24     //Return true if any counted objects currently exist.
25     static bool exist() {return _count > 0;}
26
27     void print() const {
28         assert(_count > 0);
29         cout << i;
30     }
31 };
32
33 inline int average(const counted& c1, const counted& c2) {
34     assert(counted::_count > 0);
35     return (c1.i + c2.i) / 2;
36 }
37 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/friend_vs_static/counted.C`

```
#include "counted.h"

unsigned counted::_count = 0;
```

A static member function and a friend with the same name

There is only a trivial difference between a static member function and a friend: the former is a member while the latter is not. But this tautology is relevant because the scoping rules on pp. 122–124 treat members and non-members differently.

The function `g` in line 12 is a member of class `myclass`. On the other hand, the global functions `f` and `g` in lines 5–6 are non-members, despite the additional declarations in lines 10–11. Even if they were declared outside the class (lines 32–33) and defined inside (37–38), they would still be global.

The `main` function is also global. In a global function, two groups of names are in scope: the locals and the globals. Line 24 therefore calls the global `g`, not the member `g`. Thus, in the body of a free function, a friend eclipses a member with the same name. To call the member function `g`, line 25 needs the binary scope operator `::` and the name of `g`'s class.

The `h` in line 14 is a member function. In a member function, three groups of names are in scope: locals, members, and globals. Line 16 therefore calls the member `g`, not the global `g`. Thus, in the body of a member function, a member eclipses a friend with the same name. To call the friend `g`, line 17 needs the unary scope operator `::`.

The `::` in line 17 was necessary only because there was a member with the same name. Line 15 does not need it. Earlier examples of the `::` were on p. 123, line 14; and, in class `point`, on p. 209, line 5.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/friend_vs_static/eclipse.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 inline void f() {cout << "friend f\n";}
 6 inline void g() {cout << "friend g\n";}
 7
 8 class myclass {
 9 public:
10     friend void f();    //the function in line 5
11     friend void g();    //the function in line 6
12     static void g() {cout << "static member function g\n";}
13
14     void h() const {
15         f();            //the friend in lines 5 and 10
16         g();            //the static member function in line 12
17         ::g();          //the friend in lines 6 and 11
18     }
19 };
20
21 int main()
22 {
23     f();                //the friend in lines 5 and 10
24     g();                //the friend in lines 6 and 11
25     myclass::g();       //the static member function in line 12
26     cout << "\n";
27
28     myclass x;
29     x.h();
30     return EXIT_SUCCESS;
31 }
```

`friend f`	*line 23*
`friend g`	*line 24*
`static member function g`	*line 25*
`friend f`	*line 29 calls 14, which calls 15, 16, and 17*
`static member function g`	
`friend g`	

We have already seen a class with a member and a friend having the same name: class `point` and its `dist` functions. It is actually quite natural and convenient.

2.15 Pointers to Non-Static Members

A "pointer to a member" of a class is a variable whose value answers the question "which member of the class are we talking about?". It is not really a pointer at all; it is more like an enumeration. For a class with three members, all of the same data type, a pointer to a member would have one of three possible non-zero values.

There are two types of pointers to members: pointers to data members and pointers to member functions. We'll begin with pointers to data members, since they are simpler. In each case we'll show the syntax followed by a motivating example.

Before we introduce any new type of pointer, however, we will review a type we already have in C: a pointer to a free function.

2.15.1 A Pointer to a Free Function

A free function is one that takes no invisible pointer; see p. 113. The name of an array, without the subscripting operator `[]` after it, stands for the address of the array. Similarly, the name of a free function, without the function call operator `()` after it, stands for the address of the function. An example is the name of the `sqrt` function in line 8; we could also have written `&sqrt`. Line 8 stores this address into a very specific kind of pointer: one that can point only to a function taking a `double` and returning a `double`. The pointer is `*const`: it will always point to the same function.

The standard library has three square root functions.

```
//Excerpt from <cmath>

float sqrt(float);
double sqrt(double);
long double sqrt(long double);
```

The `sqrt` in line 8 is the address of the `double` function because it is stored into the pointer `p`. It is a rare example of an expression whose value depends on the surrounding context.

Line 11 prints the value of this pointer. The C++ Standard Library has << operators that can print a pointer to `void` or a pointer to a variable, but none that can print a pointer to a function of the type of `sqrt`. Neither a `static_cast` nor a `reinterpret_cast` were willing to convert a pointer-to-function directly into a pointer-to-nonfunction such as a pointer-to-`void`. Line 11 therefore casts the pointer to a `size_t` and then to a `void *`. Since a `size_t` can hold the number of bytes in any block of memory, it should be able to hold all the bits in a pointer with no loss of information.

If we tried to output `p` without any casting, it would be converted to a `bool` and printed as such. `bool` is the only data type for which the library has an << operator and to which this type of pointer may be converted without a cast.

Of course, if all we want to do is to print the address of a function, we don't need to store it into the pointer variable `p` in line 8. We can just print the address directly in lines 13−15. But this requires even more elaborate casting. The cast in line 15 specifies which of the three square root functions we want;

those in 14 and 13 are the ones we saw in 11.

The expression `(*p)(2.0)` in line 18 calls the function to which `p` points. It has two operators. First it applies the dereferencing operator `*` to the expression `p`, retrieving what `p` points to. Since this is a function, it then applies the function call operator `()` (with a `2.0` inside it) to the function. The parentheses around the `*p` are not an operator. They merely force the `*` to execute before the function call operator `()`.

$$\boxed{\boxed{(\ast\ \boxed{\texttt{p}})}\,(\,\boxed{\texttt{2.0}}\,)}$$

Finally, line 18 stores the return value into `d`.

Of course, if all we want to do is to call the function and print its return value, we don't need to store the value into the variable `d` in line 18. We can just print it directly in line 20.

The use of `p` in the expression `(*p)(2.0)` in lines 18 and 20, and the declaration for `p` in line 8, have the same operators in the same relative order. I usually figure out how to write the use of the pointer first, and then paste it into the declaration.

In the expression `(*p)(2.0)` in lines 18 and 20, the dereferencing operator `*` is optional in C and C++. Lines 22 and 24 repeat this expression without the `*`. And now that the `*` is gone, we no longer need the parentheses that forced it to go first.

In C, making the dereferencing operator optional was merely a convenience. In C++, it will make it possible for a ''template function'' to use the same syntax for two different types of arguments: a pointer to a function, and a ''function object''. See pp. 764–770.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_function/free.C`

```
 1 #include <iostream>   //C++ example
 2 #include <cstdlib>
 3 #include <cmath>      //for sqrt
 4 using namespace std;
 5
 6 int main()
 7 {
 8     double (*const p)(double) = sqrt;
 9
10     cout << "p == "
11         << reinterpret_cast<void *>(reinterpret_cast<size_t>(p)) << "\n"
12         << "sqrt == "
13         << reinterpret_cast<void *>(
14            reinterpret_cast<size_t>(
15            static_cast<double (*)(double)>(sqrt)))
16         << "\n\n";
17
18     double d = (*p)(2.0);
19     cout << "d == " << d << "\n"
20         << "sqrt(2.0) == " << (*p)(2.0) << "\n\n";
21
22     d = p(2.0);
23     cout << "d == " << d << "\n"
24         << "sqrt(2.0) == " << p(2.0) << "\n";
25
26     return EXIT_SUCCESS;
27 }
```

The above line 8 could be split into

```
28     double (*p)(double);
29     p = sqrt;
```

But why would we want to? To permit the assignment in line 29, we would have to remove the `const` from line 12.

`p == 0x2112c`	*lines 10−11: base 16 on my platform*
`sqrt == 0x2112c`	*lines 12−13*
`d == 1.41421`	*line 19*
`sqrt(2.0) == 1.41421`	*line 20*
`d == 1.41421`	*line 23*
`sqrt(2.0) == 1.41421`	*line 24*

Why would we want a pointer to a function?

Here's a `main` function that decides which of four other functions to call. It has four `if-else`'s.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_function/before4.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f0();
 6 void f1();
 7 void f2();
 8 void f3();
 9
10 int main()
11 {
12     int i;   //uninitialized variable
13     cin >> i;
14
15     if (i == 0) {
16         f0();
17     } else if (i == 1) {
18         f1();
19     } else if (i == 2) {
20         f2();
21     } else if (i == 3) {
22         f3();
23     } else {
24         cerr << "input " << i
25             << " must be in the range 0 to 3 inclusive\n";
26         return EXIT_FAILURE;
27     }
28
29     return EXIT_SUCCESS;
30 }
```

A simpler way to do the same job is with the array of pointers to functions in line 12. The declaration for the array is rather mysterious. An easier way to see the data type of the array is by looking at the initial values in lines 13−16. They show that `a` an array of pointers to functions.

The expression `(*a[i])()` in the comment in line 29 calls the function to which `a[i]` points. It has three operators. First it applies the subscripting operator `[]` to the array, retrieving an array element. Since this element is a pointer, it applies the dereferencing operator `*` to the pointer, retrieving what the pointer points to. Since this is a function, it applies the function call operator `()` to the function. The parentheses around the `*a[i]` are not an operator. They merely force the `*` to execute before the function call operator `()`.

$$\boxed{\boxed{(\;*\;\boxed{\boxed{\texttt{a}}\;[\;\boxed{\texttt{i}}\;]}\;)}\,()}$$

The use of `a` in the expression `(*a[i])()` in the comment in line 29, and the declaration for `a` in line 12, have the same operators in the same relative order. I usually figure out how to write the use of the array first, and then paste it into the declaration.

In the expression `(*a[i])()` in the comment in line 29, the dereferencing operator `*` is optional in C and C++. Line 29 has this expression without the `*`. And now that the `*` is gone, we no longer need the parentheses that forced it to go first.

The expression `a[i]()` in line 29 now does all the work of the above `before4.C`, except for the error checking which has been neatly isolated in lines 23–27. It is also faster than the list of `if`'s and `else`'s, since it leaps directly to the correct function.

The `int i` in line 12 of `before4.C` has become a `size_t` in line 20 of `after4.C`. This is the data type that should be used for any array subscript in C or C++; see p. 66. It should also be used for the number of bytes in a block of memory, as in the `offsetof` macro on pp. 254–255. Since `size_t` is unsigned, line 23 does not need to check if `i` is less than zero. A negative number stored in `i` would become a large positive number.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_function/after4.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f0();
 6 void f1();
 7 void f2();
 8 void f3();
 9
10 int main()
11 {
12     void (*const a[])() = {
13         f0,   //or &f0, with explicit "address of" operator
14         f1,
15         f2,
16         f3
17     };
18     const size_t n = sizeof a / sizeof a[0];
19
20     size_t i;   //uninitialized variable
21     cin >> i;
22
23     if (i >= n) {
24         cerr << "input " << i << " must be in the range 0 to " << n - 1
```

```
            << " inclusive\n";
        return EXIT_FAILURE;
    }

    a[i]();   //or (*a[i])() with explicit dereferencing operator
    return EXIT_SUCCESS;
}
```

Not convinced yet? Let’s scale up the example. Here the `main` function decides which of six functions to call, arranged in a 2×3 matrix. Unfortunately, the `if` statements are starting to nest and multiply.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_function/before2by3.C`

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

void f00();   void f01();   void f02();
void f10();   void f11();   void f12();

int main()
{
    int x;   //uninitialzied variable
    cin >> x;

    int y;   //uninitialzied variable
    cin >> y;

    if (y == 0) {
        if (x == 0) {
            f00();
        } else if (x == 1) {
            f01();
        } else if (x == 2) {
            f02();
        } else {
            cerr << "x value " << x
                << " must be in the range 0 to 2 inclusive\n";
            return EXIT_FAILURE;
        }
    } else if (y == 1) {
        if (x == 0) {
            f10();
        } else if (x == 1) {
            f11();
        } else if (x == 2) {
            f12();
        } else {
            cerr << "x value " << x
                << " must be in the range 0 to 2 inclusive\n";
            return EXIT_FAILURE;
        }
    } else {
        cerr << "y value " << y
            << " must be in the range 0 to 1 inclusive\n";
```

```
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

A simpler way to do the same job is with the 2 × 3 array of pointers to functions in line 12. This time, the declaration for the array is even more mysterious. But the initial values in lines 13−14 show that it's a 2 × 3 an array of pointers to functions. Since we provided an initial value for every element, the leftmost dimension in line 12 can be left blank. But all subsequent dimensions must be specified.

The expression `a[y][x]()` in line 36 now does all the work of the above `before2by3.C`, except for the error checking which has been neatly isolated in lines 21–25 and 30–34. I'm sorry that the `y` comes before the `x` in the `a[y][x]()`, but it has to be this way. In a two-dimensional array in C or C++, the first subscript is the row and the second is the column.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_function/after2by3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f00();   void f01();   void f02();
 6 void f10();   void f11();   void f12();
 7
 8 int main()
 9 {
10     const size_t xmax = 3;
11
12     void (*const a[][xmax])() = {
13         {f00, f01, f02},   //or &f00 with explicit "address of" operator
14         {f10, f11, f12}
15     };
16     const size_t ymax = sizeof a / sizeof a[0];
17
18     size_t x;   //uninitialized variable
19     cin >> x;
20
21     if (x >= xmax) {
22         cerr << "x value " << x << " must be in the range 0 to "
23             << xmax - 1 << " inclusive\n";
24             return EXIT_FAILURE;
25     }
26
27     size_t y;   //uninitialized variable
28     cin >> y;
29
30     if (y >= ymax) {
31         cerr << "y value " << y << " must be in the range 0 to "
32             << ymax - 1 << " inclusive\n";
33             return EXIT_FAILURE;
34     }
35
36     a[y][x]();   //or (*a[y][x])() with explicit dereferencing operator
37     return EXIT_SUCCESS;
38 }
```

2.15.2 A Pointer to a Static Member

A “pointer to a member” is always assumed to be a pointer to a non-static member. If the member is static, we can use the normal kind of pointer.

The static data member `j` is initialized in its declaration in line 8. Normally, therefore, we would not need a separate definition for it; see p. 238. But because we take its address in line 18, it must be defined in line 13.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_member/static.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class myclass {
 6 public:
 7     static int i;
 8     static const int j = 20;
 9     static void f() {cout << "myclass::f\n";}
10 };
11
12 int myclass::i = 10;
13 const int myclass::j;   //needed because we're taking address of j
14
15 int main()
16 {
17     int *p1 = &myclass::i;
18     const int *p2 = &myclass::j;
19
20     cout << "myclass::i == " << *p1 << "\n"
21         << "myclass::j == " << *p2 << "\n";
22
23     void (*p3)() = myclass::f;   //or &myclass::f with explicit "address of"
24     p3();                    //or (*p3)() with explicit dereferencing operator
25
26     return EXIT_SUCCESS;
27 }
```

```
myclass::i == 10
myclass::j == 20
myclass::f
```

2.15.3 A Pointer to a Data Member

We’ll do pointers to data members before pointers to member functions, because pointers to data members have fewer parentheses. In each case we’ll show the syntax followed by a motivating example. We reviewed pointers to free functions because we will need thir syntax when we do pointers to member functions.

Assume that class `date` has the following members. For simplicity, we’ll make them public:

```
    int year, month, day;
    void print() const;
```

A pointer to an object is a variable that tells us which object we're interested in. For example, the pointer `p1` in line 5 points to the object `d`. `p1` can point only to a `date` object.

A pointer to a data member is a variable that tells us which data member we're interested in. For example, the pointer `p2` in line 3 "points to" the data member `year` of class `date`. `p2` can point only to an `int` data member of class `date`.

A pointer to a data member really isn't a pointer, and `p2` certainly does not contain the address of the `year` data member of any object. To emphasize this, we initialized `p2` before we constructed any `date` object at all. We'll return to this issue of what a pointer to a data member really is.

Lines 10–11 form a little table with two rows and two columns, showing four binary operators. The ones in the bottom row must be used when dereferencing a pointer to a member.

```
.      ->
.*     ->*
```

In column 1 of lines 10 and 11, our choice of the object `d` is hard-coded in. In column 2, our choice of object is indicated by the variable `p1`.

Similarly, in line 10, our choice of the data member `year` is hard-coded in. In line 11, our choice of data member is indicated by the variable `p2`.

```
 1 #include "date.h"
 2
 3     int date::*p2 = &date::year;        //ampersand required
 4     date d;                             //today's date
 5     date *p1 = &d;
 6
 7     //Four ways to output the year data member of the object d:
 8
 9     //the object d      //the object pointed to by p1
10     cout << d.year;     cout << p1->year;   //the data member year
11     cout << d.*p2;      cout << p1->*p2;    //the data member pointed to by p2
```

```
 2014              2014
 2014              2014
```

See line 33 of `point.h` on p. 725 for an exampe of a pointer to a data member under actual combat conditions.

What does a pointer to a member really hold?

A pointer to a member tells us what member we're interested in. But it does not identify the member by holding the member's address. It identifies the member by holding the member's *offset,* i.e. its distance in bytes from the start of its object.

To show how a pointer to a data member actually works, here is how a C program would make a variable indicating which field of a structure we're interested in. The C Standard Library has a macro named `offsetof` that returns the offset of a field from the start of its structure. Line 13 stores the value of the macro into the variable `offset` to indicate that we are interested in `field2`. Line 19 uses `offset` to access `field2` of the structure `s`. When an integer is added to a pointer in C and C++, the integer is implicitly multiplied by the number of bytes in each pointed-to object. Line 19 casts the `&s` to a pointer to a `char`, so that the implicit multiplication will be a multiplication by 1. The sum is cast to a pointer to an `int`, so that we can retrieve an `int` field from the structure. Finally, the pointer to an `int` is dereferenced with a leading `*`, the cherry on the sundae.

All of this virtuoso pointer arithmetic is hidden in the expressions `d.*p2` and `p1->*p2` in the above line 11.

Incidentally, the `%u` in line 16 is not portable. On my machine `size_t` is another name for `unsigned`, but on your machine it might be different.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_member/offsetof.c`

```
1 #include <stdio.h>    /* C example */
2 #include <stdlib.h>
3 #include <stddef.h>   /* for offsetof */
4
5 struct mystruct {
6     int field1;
7     int field2;
8 };
9
10 int main()
11 {
12     /* We're interested in field2. */
13     const size_t offset = offsetof(struct mystruct, field2);
14     struct mystruct s = {10, 20};
15
16     printf("The field starts %u bytes from the start of mystruct.\n", offset);
17
18     printf("The field of mystr that we're interested in has the value %d.\n",
19         *(int *)((char *)&s + offset));
20
21     return EXIT_SUCCESS;
22 }
```

```
The field starts 4 bytes from the start of mystruct.
The field of mystr that we're interested in has the value 20.
```

2.15.4 A Pointer to a Member Function

A pointer to a member function is a variable that tells us which member function we're interested in. For example, the pointer `p2` in line 3 "points to" the `print` member function of class `date`. `p2` can point only to a `const` member function of class `date` that takes no arguments and returns no value.

Lines 10–11 form a little table with two rows and two columns. In column 1 of lines 10 and 11, our choice of the object `d` is hard-coded in. In column 2, our choice of object is indicated by the variable `p1`.

Similarly, in line 10, our choice of the `print` member function is hard-coded in. In line 11, our choice of member function is indicated by the variable `p2`.

The dot and arrow operators in line 10 have higher precedence than the function call operator `()`, so they need no surrounding parentheses to make them execute first. But the `.*` and `->*` operators in line 11 have lower precedence than the function call operator, so they must be surrounded with parentheses to make them execute first. See p. 873, line 11, for an example of the `->*` operator.

```
1 #include "date.h"
2
3     void (date::*p2)() const = &date::print;    //ampersand required
4     date d;                                      //today's date
5     date *p1 = &d;
6
7     //Four ways to call the print member function of the object d:
8
```

```
    //the object d      //the object pointed to by p1
     d.print();          p1->print();       //the member function print
    (d.*p2)();          (p1->*p2)();        //the member function pointed to by p2
```

```
     4/8/2014            4/8/2014
     4/8/2014            4/8/2014
```

The `.*` operator cannot be overloaded. Neither, by the way, can the `.` (dot) operator.

The above pointer to a member function is needed only to point to a non-static member function.

What are pointers to member functions for?

Lines 9–13 of `main.C` are a menu with many possible actions: cut, copy, paste, etc. Each action can be selected with a key pressed down: the shift key, the control key, or no key at all.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_member/action.h`

```
#ifndef ACTIONH
#define ACTIONH
#include <iostream>
using namespace std;

class action {   //a term from jurisprudence
    const char *p;
public:
    action(const char *initial_p) {p = initial_p;}

    void      plain() const {cout << p << " plain\n";}
    void    shifted() const {cout << p << " shifted\n";}
    void controlled() const {cout << p << " controlled\n";}
};
#endif
```

To avoid the complexity of displaying the menu and getting the mouse clicks and keystrokes, lines 27 and 28 pick the action and accompanying key at random.

Line 29 demonstrates the elegance of arrays and pointers to member functions. It would be instructive to rewrite this line without using pointers to member functions.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer_to_member/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>    //for srand, rand, exit, EXIT_SUCCESS
 3 #include <ctime>      //for time
 4 #include "action.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const action menu[] = {   //array of objects
10         "cut",
11         "copy",
12         "paste"
13     };
14     const size_t menu_size = sizeof menu / sizeof menu[0];
15
```

```
    //array of pointers to the member functions of the above objects
    void (action::*const key[])() const = {
        &action::plain,
        &action::shifted,
        &action::controlled
    };
    const size_t key_size = sizeof key / sizeof key[0];

    srand(static_cast<unsigned>(time(0)));

    for (int i = 0; i < 3; ++i) {
        const size_t m = rand() % menu_size; //which object of class action
        const size_t k = rand() % key_size;  //which member function of class action
        (menu[m].*key[k])();                 //Call member function k of object m.
    }

    return EXIT_SUCCESS;
}
```

```
copy controlled
paste plain
cut plain
```

2.16 Aggregation

Three ways to build a bigger class out of smaller classes or other data types

Programming has always been the craft of building bigger things out of smaller ones. Before there were objects, we had bigger functions calling smaller ones. A hierarchical structure was imposed on the program by the diagram, real or imagined, of who called whom. This organization was proudly called ‘‘structured programming’’, but is now disparaged as ‘‘procedural programming’’.

Nowadays most of our variables will be data members, and most of our functions will be member functions. The bigger things that we build out of smaller ones will therefore be classes. C++ has three ways of building bigger classes out of smaller classes or other data types.

style of programming	*other names for it*	*what it means*
Object-based programming	aggregation, containment, composition	The big object *has a* little object.
Object-oriented programming	inheritance, derivation	The big object *is a* little object.
Generic programming	templates, instantiation	The little data type is *plugged into* the big class.

Aggregation is the use of little objects as the data members of big objects. *Inheritance* is the creation of a new class with a head start: the new class will have all the members that an old class had, plus more. A *template* lets us write a blank that will be filled in later, perhaps by another person, with the name of a data type. For example, ‘‘an object of the new class contains objects of the old class _____’’, or ‘‘the new class is derived from the old class _____’’.

Objects as data members

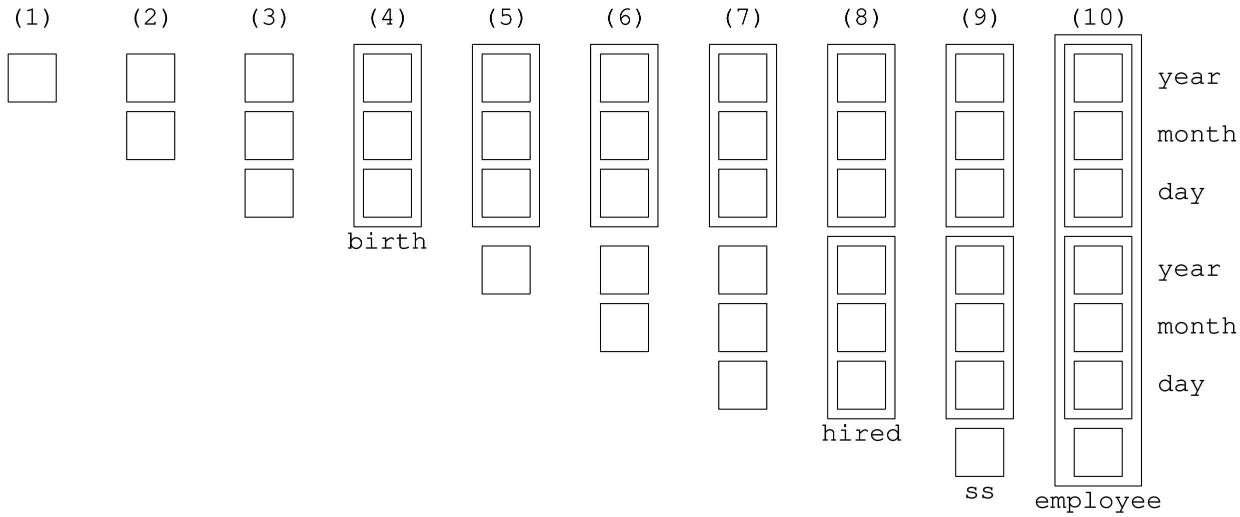
Our original class `date` contained three `int`’s: `year`, `month`, and `day`. In fact, every data member of every class has hitherto been of a built-in type (p. 27) or pointer thereto, or array thereof.

It is also possible for the data members to be objects. An object of the following class `employee` contains two `date` objects. DSince the big object has two little objects inside it, we say that there is a *has-*

a relationship between the big object and the little objects. There is also a *has-a* relationship between the data type of the big object and the data types of its data members.

The header file for the containing class always begins by including the header files for the contained classes (line 3 of `employee.h`). For rthe same reason, the header file for class `rabbit` began by including the header file for class `terminal` on p. 195.

The data members are always constructed before, and destructed after, the object that contains them. Here is a motion picture showing the order in which an `employee`, and the two objects inside of it, are constructed. The three objects are rectangles, the seven integers are squares.



To explain why the inner objects are constructed first, we introduce some terminology. The `{`curly braces`}` of a function and the enclosed statements) are the *body* of the function. When we return from the body of a constructor, either by reaching the closing `}` or by executing a `return` statement, the object being constructed is said to be *completely constructed*.

Suppose that we were somehow interrupted between steps (9) and (10) in the above diagram. Then the two objects `birth` and `hired` would be completely constructed, and the `employee` object would not be. This black-and-white view of the world is very clean: every object is either completely constructed or it isn't.

Now suppose that the outer object was constructed first. If we were then interrupted before constructing the `birth` and `hired` inside of it, the outer object would be a hollow shell. We would have to classify it as a "completely constructed object with missing guts". No one wants this shade of gray.

There's another reason why the data members must be constructed before the object that contains them. A `date` has no idea whether it is part of a surrounding `employee`: the files `date.h` and `date.C` make no mention of class `employee`. The error check in lines 9 and 23 of `employee.C` therefore can not be performed by a constructor for class `date`. It must be performed by a constructor for class `employee`, which must be executed after the constructors for the `date`'s.

On my platform, an `int` is big enough to hold a nine-digit social security number (line 5). On other platforms we might have to use a `long`; if the government adds extra digits or letters, we might have to use an object. For the time being, we assume that a `ss_t` is fast enough to pass by value.

When we have operator overloading, we will simplify the constructors for class `employee` (p. 340) and its `print` and `retire` member functions (pp. 340 and 286).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/employee/employee.h`

```
1 #ifndef EMPLOYEEH
2 #define EMPLOYEEH
```

```
#include "date.h"

typedef int ss_t;   //social security number; not portable

class employee {
    date birth;
    date hired;
    ss_t ss;
public:
    employee(const date& initial_birth, ss_t initial_ss);
    employee(int initial_month, int initial_day, int initial_year,
        ss_t initial_ss);
    ~employee() {cout << "Employee # " << ss << " gets a pink slip.\n";}

    date retire() const {date d = birth; d.next(65 * 365); return d;}
    void print() const;
};
#endif
```

I wish the error messages in the following lines 10–12 and 24–26 could be written to `cerr`. But the `print` function in lines 11 and 25 is hardwired to write to `cout`. We'll fix this on p. 340 when `print` is replaced with an `operator<<` function.

Line 35 would not compile if we changed it to

```
    cout << birth.month << "/" << birth.day << "/" << birth.year << "\n";
```

Class `employee` does contain a `date`, but that doesn't give `employee` permission to mention the private members of `date`. Anyway, we don't even know if there is a member named `month` in this class `date`. It might be the class `date` with only one data member.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/employee/employee.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "employee.h"
 4 using namespace std;
 5
 6 employee::employee(const date& initial_birth, ss_t initial_ss)
 7     : birth(initial_birth), hired()
 8 {
 9     if (dist(hired, birth) < 16 * 365) {
10         cout << "employee born on ";
11         birth.print();
12         cout << " << too young to hire\n";
13         exit(EXIT_FAILURE);
14     }
15
16     ss = initial_ss;
17 }
18
19 employee::employee(int initial_month, int initial_day, int initial_year,
20      ss_t initial_ss)
21     : birth(initial_month, initial_day, initial_year), hired()
22 {
23     if (dist(hired, birth) < 16 * 365) {
24         cout << "employee born on ";
```

```
        birth.print();
        cout << " << too young to hire\n";
    }

    ss = initial_ss;
}

void employee::print() const
{
    cout << "birth date: ";
    birth.print();
    cout << "\thired on: ";   //\t is the tab character
    hired.print();
    cout << "\tss #: " << ss;
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/employee/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "employee.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const date birthday(date::july, 12, 1955);   //the author's birthday
 9     const employee e1(birthday, 123456789);
10     e1.print();
11     cout << "\n";
12
13     const employee e2(date::may, 1, 1957, 987654321);
14     e2.print();
15     cout << "\n\n";
16
17     cout << "The second employee will retire on ";
18     const date r = e2.retire();
19     r.print();
20     cout << ".\n";
21
22     cout << "The second employee will retire on ";
23     e2.retire().print();   //and then destruct the anonymous date
24     cout << ".\n\n";
25     return EXIT_SUCCESS;   //destruct r, e2, e1, birthday, in that order
26 }
```

The `birthday` in the above line 8 is used only once, in line 9. We may therefore reduce it to an anonymous temporary and combine lines 8–9 to the following. Our first example of an anonymous object passed to a function was in line 6 on p. 138.

```
    const employee e1(date(date::july, 12, 1955), 123456789);
```

Be sure to tell the computer the names of all the non-header files that constitute the program.

```
1$ g++ -o ~/bin/employee main.C employee.C date.C
```

```
birth date: 7/12/1955  hired on: 4/8/2014   ss #: 123456789
birth date: 5/1/1957   hired on: 4/8/2014   ss #: 987654321

The second employee will retire on 5/1/2022.
The second employee will retire on 5/1/2022.

Employee # 987654321 gets a pink slip.
Employee # 123456789 gets a pink slip.
```

The colon notation for calling a constructor for a data member

> Let us tentatively suppose . . . that '$x + y = y + x$' does hold as a genuine identity; i.e., that the order of summands is wholly immaterial. A notation of addition more suggestive than '$x + y$', then, would consist in simply superimposing 'x' and 'y' in the manner of a monogram.
>
> —W. V. Quine, in *The Philosophy of Alfred North Whitehead*, p. 128

The constructors for the data members are executed *before* the body of the constructor for the object that contains the data members. Similarly, the destructors for the data members are executed *after* the body of the destructor for the containing object.

For example, line 9 of `main.C` calls the constructor with two arguments for class `employee`. Normally we would go straight from this line to line 9 of `employee.C`, which is the first line of the body of the constructor. But we make two detours along the way. From line 9 of `main.C` we go to line 7 of `employee.C`. This "colon line" calls the constructors for the data members `birth` and `hired` and gives them their arguments. The copy constructor is called for `birth` and the default constructor for `hired`. Only after these two constructors have executed do we proceed to line 9 of `employee.C`.

The `birth` data member is constructed before `hired` because of the order of the declarations in lines 8–9 of `employee.h`. It has nothing to do with the order in which the data members are mentioned in line 7 of `employee.C`. `birth` would still be constructed before `hired` even if line 7 of `employee.C` had said

```
	: hired(), birth(initial_birth)
```

But there's no reason to write the names in a misleading order.

Line 7 of `employee.C` passes no arguments to the constructor for `hired`. The line would therefore usually be written as follows. The default constructor for `hired` would still be called after the copy constructor for `birth`.

```
	: birth(initial_birth)
```

The `, hired()` can also be removed from line 21 of `employee.C`.

Even if there is no colon line at all, a constructor for each data member will still be called. In this case it will be the default constructor, and the program will compile only if every data member has one. But even if it does compile, the program may be wasting time. For example, let's remove the colon line from our constructor.

```
employee::employee(const date& initial_birth, ss_t initial_ss)
{
	birth = initial_birth;

	if (dist(hired, birth) < 16 * 365) {
		cout << "employee born on ";
		birth.print();
		cout << " << too young to hire\n";
		exit(EXIT_FAILURE);
	}
```

```
13
14     ss = initial_ss;
15 }
```

The default constructors for `birth` and `hired` will still be called at the above line 3½. This will initialize `birth` to today's date, and will then wipe it out in line 5.

The best way to define the two-argument constructor for class `employee` is the following. Line 17 calls the copy constructor for `birth` and the default constructor for `hired`. It also calls the "constructor" for the data member `ss`. This member is merely a built-in (p. 27), not an object. But we are allowed, and in fact encouraged, to use a syntax that makes it look as if every member were an object. If we program in the same style with all our variables, built-ins and objects, we will be able to convert our code into "templates" more easily. See p. 634.

```
16 employee::employee(const date& initial_birth, ss_t initial_ss)
17     : birth(initial_birth), ss(initial_ss)
18 {
19     if (dist(hired, birth) < 16 * 365) {
20         cout << "employee born on ";
21         birth.print();
22         cout << " << too young to hire\n";
23         exit(EXIT_FAILURE);
24     }
25 }
```

Initialization vs. assignment

Initialization puts the first value into a new variable. *Assignment* puts a new value into an existing variable. Although we have always written them with the same symbol (=), we can readily tell them apart.

```
    int i = 10;                //Initialize i.
    i = 20;                    //Assign to i.
```

For the built-in data types (p. 27), pointers, and enumerations, initialization and assignment differ in only one way: we can initialize a `const` but cannot assign to it.

```
    const int i = 10;          //can initialize a const
    i = 20;                    //can't assign to a const: won't compile
```

If the variable is not a `const`, initialization and assignment are the same operation.

The original `ss = initial_ss` in line 16 of `employee.C` is the notation for assigning a value to any variable. The `: ss(initial_ss)` in the above line 17 is the notation for initializing a variable that is a data member. Since `ss` is a non-`const` built-in, our preference for initialization over assignment is merely a matter of style. But if `ss` ever becomes a `const`, we will be forced to initialize it (p. 266). And if `ss` becomes an object, we have seen that it will be initialized whether we write the colon line or not. Therefore the above line 17 initializes `ss` now, so we won't have to change it later.

An object is always initialized by calling its constructor. For an object, *initialization* and *construction* are two names for the same operation. But pp. 302−303 will show that for an object, initialization and assignment may be very different operations, initialization often being cheaper. For these reasons, always put the initial value into any object by initialization, not assignment. In fact, do this when possible for any variable.

Destruction

Each `employee`'s destructor will be called automatically in line 25 of the above `main.C`. After executing the body of this destructor (line 15 of `employee.h`), we automatically call the destructors for the data members `hired` and `birth` (in the order opposite to that of lines 8–9 of `employee.h`). To see the order in which the three objects are destructed, read the above diagram from right to left.

▼ Homework 2.16a: initialize the data members

Wherever possible, go back and make the constructors initialize the data members, rather than assign to them. It will make a difference if the data members ever become objects in their own right. This could happen without our even knowing it, when we switch to templates.

(1) On pp. 145−146, the constructor currently begins like this:

```
life::life(const matrix_t& initial_matrix)
{
    g = 0;
```

Change it to

```
life::life(const matrix_t& initial_matrix)
    : g(0)
{
```

(2) In line 11 of `stack.h` on p. 150, class `stack` has an inline constructor:

```
    stack() {n = 0;}
```

Change it to

```
    stack(): n(0) {}
```

(3) In Homework 2.6b on pp. 152−153, the new class `stack` has a similar constructor:

```
    stack() {p = a;}
```

Change it to

```
    stack(): p(a) {}
```

The data member `p` should also be initialized by the copy constructor for the new class `stack`. The data member `a`, however, is an array, and there is no syntax for initializing an array with a colon. The copy constructor for the new class `stack` will have to continue to assign to `a` with the `for` loop.

(4) In lines 9−12 of `point.h` on p. 201, class `point` has an inline constructor with default values for both its arguments:

```
    point(double initial_x = 0.0, double initial_y = 0.0) {
        x = initial_x;
        y = initial_y;
    }
```

Change it to

```
    point(double initial_x = 0.0, double initial_y = 0.0)
        : x(initial_x), y(initial_y) {}
```

(5) Make this change to classes `duo` and `mono` on pp. 135−137.

(6) Make this change to class `terminal` in lines 7−23 of `terminal.C` on p. 160. It's easy to initialize the data member `_background`. To initialize the other two data members, we must call the C functions `term_xmax` and `term_ymax`. But before we call them, we must call `term_construct`. Where is there room to do that in the colon line?

C and C++ have a trick for writing two expressions where the syntax permits only one. We can build a big expression out of two smaller ones with the *comma operator.* The two smaller expressions are executed from left to right (p. 13). Its most common use is to let us change the values of two variables in a loop.

```
    //The righmost comma is a comma operator.
    for (int i = 0, j = 1; i < 10; ++i, j *= 2) {
        cout << "2 to the power " << i << " is " << j << ".\n";
```

```
    }
```

When a comma operator is written in an argument list, its two expressions must be surrounded by parentheses. That's how the computer knows it's the comma operator, rather than the comma that separates arguments.

```
void f(int a, int b);          //function declarations
void f(int a, int b, int c);

    f((++i, --j), k);         //call 2-arg f; leftmost , is the comma operator
    f(++i, --j, k);           //call 3-arg f; none of these is the comma operator
```

Since we are writing the comma operator in the argument list of the constructor for `_xmax`, its two operands must be enclosed in parentheses.

```
terminal::terminal(char initial_background)
    : _background(initial_background),
    _xmax((term_construct(), term_xmax())),
    _ymax(term_ymax())
{
```

A similar use of parentheses is to enclose the > operator in a template preamble; see p. 693.

I'm not happy about this bizarre syntax, but I want the data members of class `terminal` to be initialized. Soon they will have to be (p. 269).

(7) Make this change to class `random` in line 9 of `myrandom.h` on p. 176.

(8) Make this change to all three constructors for class `obj` in lines 9–11 of `obj.h` on p. 180.

(9) Make this change to the constructor for class `node` in line 15 of `node.h` on p. 214.

(10) Make this change to the constructor for class `action` in line 10 of `action.h` on p. 256.

▲

▼ Homework 2.16b:
Version 1.6 of the Rabbit Game: initialize the data members of classes `wolf` and `rabbit`

The constructors for classes `rabbit` and `wolf` currently assign values to their three non-static data members. Let them initialize the data members instead.

▲

▼ Homework 2.16c: create a class whose data members are objects

Before we had objects, we had to pass the addresses of many individual variables to a function. An example was in line 32 of `version1.C` on p. 107.

```
    date_print(&year, &month, &day);
```

After we had objects, all we had to pass was one invisible address. See line 41 of `version3.C` on p. 109.

```
    d.print();
```

Now look at the calls to the `area` and `contains` functions in lines 15 and 32 of `main.C` on pp. 208–209. Like the original `date_print`, we have to pass them three separate variables.

```
    cout << area(A, B, C) << "\n"
        << contains(A, B, C, D) << "\n";
```

Define a new class, `triangle`, whose data members will be three `point`'s named `A`, `B`, and `C`. Remove the existing `area` and `contains` functions and reincarnate them as member functions of class `triangle`. The `contains` function will construct three `triangle` objects as anonymous temporaries, and compare the sum of their areas with the area of the `triangle` of which the `contains` is a member.

```
    triangle t(A, B, C);
```

```

    cout << t.area() << "\n"
        << t.contains(D) << "\n";
```

You can invent additional member functions: `perimiter`, `center`, `is_right`, etc. Should triangle `ABC` be equal to triangle `ACB`?

The header file `triangle.h` will have to include `point.h`. To compute the area without mentioning the private members of class `point`, `triangle::area` can use Heron's formula. Let a, b, and c be the lengths of the sides. Let

$$s = \frac{a+b+c}{2}$$

Then the area of the triangle will be

$$\sqrt{s(s-a)(s-b)(s-c)}$$

Better yet, give class `point` the following public member functions. Call them from `triangle::area` and let `triangle::area` use the original formula for the area.

```
    double get_x() const {return x;}
    double get_y() const {return y;}
```

▲

▼ Homework 2.16d: should this class have data members that are objects?

Let's assume we have a class `point` whose data members are

```
    double x;
    double y;
```

We can easily create a class `line` by giving it the data members

```
    point A;                         //must be two different points!
    point B;
```

Let's assume that our class `line` is to represent an infinitely long line, not a line segment. The two points would therefore contain too much information, a total of four `double`'s. A smaller way to represent the line would be

```
    point p; //any point on the line
    double slope;
```

A `line` now contains a total of three `double`'s, but one of them could be infinity.*

Is there an even more compact represenation for a `line`? Should a `triangle` contain three `line`'s instead of three `point`'s?

▲

2.17 Constant Non-static Data Members

* There may be a way to store ∞ into a `double` without attempting to divide by zero. See the `has_infinity` data member and the `infinity` member function of the template class `numeric_limits`.

Initialize a constant non-static data member

We have already seen a `const` static data member, in line 7 on p. 237. We can also have a `const` non-static data member, in the following line 8. Line 12 will therefore not compile: we cannot assign to a `const`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_data_member/badinterval.h`

```
 1 #ifndef INTERVALH
 2 #define INTERVALH
 3 #include <iostream>
 4 #include "date.h"
 5 using namespace std;
 6
 7 class interval {
 8     const date begin;
 9     date end;
10 public:
11     interval(const date& initial_begin, const date& initial_end) {
12         begin = initial_begin;
13         end = initial_end;
14     }
15
16     void change_end(const date& new_end) {end = new_end;}
17     void print() const {cout << "(" << begin << ", " << end << ")";}
18 };
19 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_data_member/badincluder.C`

```
#include "badinterval.h"

int main()
{
}
```

The “discard qualifiers” message means you’re trying to do something with a `const` object that can only be done with a non-`const` object.

```
In file included from badincluder.C:1:0:
badinterval.h: In constructor ’interval::interval(const date&, const date&)’:
badinterval.h:12:11: error: passing ’const date’ as ’this’ argument of ’date&
date::operator=(const date&)’ discards qualifiers
```

Instead of the assignment in above line 12, the constructor for class `interval` will have to initialize `begin` in line 24:

```
23     interval(const date& initial_begin, const date& initial_end)
24         : begin(initial_begin) {
25         end = initial_end;
26     }
```

The constructor in the above lines 23–26 still has a bug, although it is only a performance bug. To see it, observe that the above line 24 really does the same thing as line 28:

```
27     interval(const date& initial_begin, const date& initial_end)
28         : begin(initial_begin), end() {
```

```
29          end = initial_end;
30      }
```

Now we can see that line 24 uselessly initializes `end` to today's date, and then the next line assigns `initial_end` to `end`. To avoid this waste of time, `end` should be initialized to the correct value:

```
31      interval(const date& initial_begin, const date& initial_end)
32          : begin(initial_begin), end(initial_end) {}
```

We can't assign to an object that contains a const data member.

The equal signs in lines 9 and 14 perform the initialization operation. They call copy constructors. The equal signs in lines 10 and 15 perform the assignment operation. They assign the value of each data member in their right operand to the corresponding data member in their left operand. Line 15 is therefore a shorthand for

```
    i1.begin = i2.begin;            //won't compile
    i1.end = i2.end;                //will compile
```

which is why line 15 will not compile.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_data_member/main.C`

```
 1 #include <cstdlib>
 2 #include "date.h"
 3 #include "interval.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date d1;             //Initialization: call the default constructor.
 9     date d2 = d1;        //Initialization: call the copy constructor.
10     d2 = d1;             //Assignment.
11
12     d2.next(10);
13     interval i1(d1, d2);//Initialization: call the two-arg constructor.
14     interval i2 = i1;    //Initialization: call the copy constructor.
15     i1 = i2;             //This assignment won't compile.
16
17     return EXIT_SUCCESS;
18 }
```

```
In file included from main.C:3:0:
interval.h: In member function 'interval& interval::operator=(const interval&)':
interval.h:7:16: error: non-static const member 'const date interval::begin',
can't use default assignment operator
main.C: In function 'int main()':
main.C:15:7: note: synthesized method 'interval& interval::operator=(const
interval&)' first required here
```

Two ways to make the data members constant

The class `const_members` in lines 4–19 shows one way of making a data member constant. We have chosen to make all of them `const`, but we didn't have to. The data members are constant throughout its lifetime, so the increments and decrements in lines 9 and 14 would not compile.

Although its data members are `const`, the object `o1` in line 36 is not `const`. The calls to the non-`const` member function `f` in line 37 will therefore compile.

Another way of making constant data members is by declaring the whole object to be `const` in line 39. This time, every data member will always be constant. But the object does not become `const` until it reaches the closing curly brace at the end of the constructor in line 27. Until then the data members can still be modified, and we can still call non-`const` member functions. The object ceases to be `const` when it reaches the opening curly brace at the start of the destructor in line 29. After that the data members can again be modified, and non-`const` member functions can be called.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/const_data_member/const_members.C`

```
 1 #include <cstdlib>
 2 using namespace std;
 3
 4 class const_members {
 5     const int i;
 6     const int j;
 7 public:
 8     const_members(int initial_i, int initial_j): i(initial_i), j(initial_j){
 9         //++i or ++j would not compile here
10         f();
11     }
12
13     ~const_members() {
14         //--j or --i would not compile here
15         f();
16     }
17
18     void f() {}   //a non-const member function
19 };
20
21 class obj {
22     int i;
23     int j;
24 public:
25     obj(int initial_i, int initial_j): i(initial_i), j(initial_j) {
26         ++i; ++j; f();
27     }
28
29     ~obj() {f(); --j; --i;}
30
31     void f() {}   //a non-const member function
32 };
33
34 int main()
35 {
36     const_members o1(30, 40);
37     o1.f();                  //will compile: o1 is not const object
38
39     const obj o2(10, 20);
40     //o2.f();                //won't compile: o2 is a const object
41
42     return EXIT_SUCCESS;
43 }
```

▼ Homework 2.17a:
Version 1.7 of the Rabbit Game: constant data member for classes `wolf` and `rabbit`

Let the `t` data member of classes `wolf` and `rabbit` be constant. They will now have to be initialized with a colon like `interval::begin`. (You have already done this if your homework is up-to-date.)

The data member `t` is a pointer. Recall from pp. 50–52 that there are two ways of making a pointer constant. `t` is already constant in the sense of being a "read-only" ponter. Keep it constant that way, but also make it constant in the sense of "always pointing to the same `terminal`". This will keep the animal tethered to the same `terminal` throughout its life.

Do not change the data types of the arguments of the constructors of classes `wolf` and `rabbit`.

▲

▼ Homework 2.17b:
Version 1.8 of the Rabbit Game: constant data members for class `terminal`

No member function or friend of class `terminal` changes the three data members of that class. Enforce this by making them `const`. See pp. 263–264 for instructions on how to initialize the data members.

▲