4

Operations Expressed by Overloaded Operators

4.1 Input and Output

4.1.1 Formatted I/O with Manipulators

Each data type is output or input in a different format. An `int` appears as a series of digits; a `char` as a single character; a `double` has a decimal point.

In C, the format of each value had to be specified as a conversion character after each `%` given to `printf` and `scanf`. In C++, the format is determined by the data type of the value. We saw this on pp. 27−28 and 30−31.

In both languages, the format can be fine-tuned. In C, the `printf` function can print an integer in three different bases, round a `double` to a desired number of digits, and justify a string to the left or right. In C++ we do the same formatting, but with very different machinery: function name overloading and i/o manipulators.

int and char output

Integer and character output is produced by calling two functions with the same name. In line 8, the expression `i` is of type `int`. When we write

```
cout << i
```

the computer behaves as if we had written a call to the `operator<<` function whose argument is an `int`.

```
cout.operator<<(i)
```

This function outputs the `int` in decimal, like the `%d` format of `printf`.

In the next line, the expression `static_cast<char>(i)` is of type `char`. We call a different `operator<<`, one whose argument is a `char`. This function outputs the `char` as one ASCII character like the `%c` format of `printf`.

Lines 8−9 output the integer `i` in both formats; lines 12−13 do the same for the character `c`. For the double cast in line 13, see line 14 of `static_cast.C` on p. 65.

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

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 int main()
6 {
```

```
 7      int i = 65;
 8      cout << i << "\n"                         //printf("%d", i);
 9          << static_cast<char>(i) << "\n";      //printf("%c", i);
10
11      char c = 'A';
12      cout << c << "\n"                         //printf("%c", c);
13          << static_cast<unsigned>(static_cast<unsigned char>(c))
14          << "\n";   //printf("%u", (unsigned)(unsigned char)c);
15
16      return EXIT_SUCCESS;
17  }
```

`65`	*line 8*
`A`	*line 9*
`A`	*line 12*
`65`	*line 13*

In C, the `printf` function decides at runtime which format to use; see p. 29. In C++, the compiler decides at compile time which `operator<<` function to call. (In the jargon, an overloaded function name is *resolved* at compile time.)

Here are simplified definitions for the `operator<<` functions that take an integer and a character. The latter happens not to be a member function because it can do its work by calling a member function of class `ostream`, the `put` on pp. 329−330.

```
class ostream {
    //etc.
public:
    ostream& operator<<(int i) {output i in decimal; return *this;}
    //etc.
};

inline ostream& operator<<(ostream& ost, char c) {return ost.put(c);}
```

Bases, manipulators, and format flags

Most changes of format in C++ are performed by ‘‘outputting’’ or ‘‘inputting’’ invisible things called *i/o manipulators* to a stream such as `cout`. The simplest examples are the `oct`, `hex`, and `dec` in lines 12−14. No characters are output when we ‘‘output’’ the `oct`. But outputting the `oct` makes a change to the stream, causing all subsequent integers output there to be written in octal. In other words, a C++ stream can ‘‘remember’’ a format for output, and we can even copy this format into another stream object (line 17). In C, on the other hand, a C file pointer such as `stdout` has no memory. It must to be given a format every time we call `printf`.

There is also a `setbase` manipulator in line 15, but its only arguments are 8, 10, or 16. A manipulator with an argument needs the header file `<iomanip>`; those without arguments do not.

Lines 9 and 18 save and restore the base of a stream. Saving the base is unnecessary here, because the initial base of a stream is always 10. Restoring it is also unnecessary, because the program is about to end. But code in the middle of a larger program might want to restore a base it had changed.

A stream object has an integer whose bits are flags describing its current format, including three for octal, hex, and decimal. A variable that holds format flags must be of data type `fmtflags`, a typedef for the appropriate type of integer (line 9). This data type has the last name `ios_base` (`ios` in older versions of the C++ Standard), just as the variable `cout` had the last name `std` on p. 20. (Pages 419−422 will show what it means for a data type to have a last name; for now, don't worry about it.)

The `setf` function in line 18 restores only the three flags that govern the base. (To restore all the flags, see line 42 of the next program.) The other flags of the stream remain unchanged because of the `ios_base::basefield` argument. This is an enumeration that belongs to a class, like our `date::january` on pp. 223−228. We'll look at it more closely in the next section.

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

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     int i = 10;
 9     ios_base::fmtflags save = cout.flags();   //Save all the format flags.
10
11     cout << i << "\n"     //Decimal by default: printf("%d", i)
12         << oct << i << "\n"                   //printf("%o", i);
13         << hex << i << "\n"                   //printf("%x", i);
14         << dec << i << "\n"                   //printf("%d", i);
15         << setbase(16) << i << "\n";          //printf("%x", i);
16
17     cerr.copyfmt(cout);   //Copy the entire format of cout into cerr.
18     cout.setf(save, ios_base::basefield);     //Restore the base.
19     cout << i << "\n";                        //same base as line 11
20     return EXIT_SUCCESS;
21 }
```

`10`	*line 11: decimal*
`12`	*line 12: octal after* `oct`
`a`	*line 13: hexadecimal after* `hex`
`10`	*line 14: decimal after* `dec`
`a`	*line 15: hexadecimal after* `setbase(16)`
`10`	*line 19: same as line 11*

▼ Homework 4.1.1a: inconsistent format flags

To see the format flags, call the `flags` function with no arguments in line 8. To see the meaning of each flag within the integer of flags, print the enumerations in lines 12−14. (The actual values may be different on each platform.) Like the `basefield` in the previous program, these enumerations are members of class `ios_base`. `basefield`, by the way, is a combination of the flags for all three bases (line 15).

Lines 17−19 use "bitwise and" to print the value of an individual flag.

The `oct`, `hex`, and `dec` manipulators turn the flags on and off. We can also do this directly by calling the member functions and manipulators in lines 21−40, taking arguments of type `ios_base::fmtflags`. If line 24 is too drastic for you, do 28 instead.

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

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 using namespace std;
 5
```

```
 6 int main()
 7 {
 8     ios_base::fmtflags save = cout.flags();
 9
10     cout << hex
11         << "cout.flags() == " << save << "\n"
12         << "ios_base::dec == " << ios_base::dec << "\n"
13         << "ios_base::hex == " << ios_base::hex << "\n"
14         << "ios_base::oct == " << ios_base::oct << "\n"
15         << "ios_base::basefield == " << ios_base::basefield << "\n";
16
17     if (cout.flags() & ios_base::hex) {
18         cout << "The hex flag is set.\n";
19     }
20
21     ios_base::fmtflags myflags = ios_base::dec | ios_base::hex;
22
23     //Turn on myflags; turn off all others.
24     cout.flags(myflags);
25
26     //Outside of the basefield, leave all flags unchanged.
27     //Within the basefield, turn on myflags and turn off the others.
28     cout.setf(myflags, ios_base::basefield);
29
30     //Turn on myflags; leave the others unchanged.
31     cout.setf(myflags);
32
33     //Turn on myflags; leave the others unchanged.
34     cout << setiosflags(myflags);
35
36     //Turn off myflags; leave the others unchanged.
37     cout.unsetf(myflags);
38
39     //Turn off myflags; leave the others unchanged.
40     cout << resetiosflags(myflags);
41
42     cout.flags(save); //restore all the format flags, not just 3 base flags
43     return EXIT_SUCCESS;
44 }
```

`cout.flags() == 1002`	*line 11: binary* `001000000000010`
`ios_base::dec == 2`	*line 12: binary* `000000000000010`
`ios_base::hex == 8`	*line 13: binary* `000000000001000`
`ios_base::oct == 40`	*line 14: binary* `000000001000000`
`ios_base::basefield == 4a`	*line 15: binary* `000000001001010`
`The hex flag is set.`	*line 20: same as line 11*

What will `setbase` do to the three flags if its argument is neither 8, 10, or 16? Can you turn on more than one of the three base flags by saying

```
45     cout << dec << hex;
```

or would you have to resort to `setf` or `setiosflags`? If more or less than one of the three base flags are set, in what base will the output be?

▲

Three trivial manipulators

Negative numbers have a negative sign. Positive numbers will have a positive sign if we output the `showpos` manipulator in line 11. This works only in base 10.

The `showbase` manipulator in line 12 will output a `0` (zero) before an octal integer, and a `0x` before a hexadecimal integer. (This works only if the integer is non-zero.) It will also output the currency symbol in certain locales (p. 1040). If you’re showing the base and padding a hex integer with zeroes, specify the `internal` padding on p. 357. `uppercase` makes the numbers uppercase.

The three manipulators in line 15 turn these features off. This is unnecessary since lines 8 and 17 save and restore the base and the three trivial flags. This in turn is unnecessary since the program is about to end.

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

```
 1 #include <iostream>   //don’t need <iomanip>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int i = 10;
 8     ios_base::fmtflags save = cout.flags();
 9
10     cout << i << "\n"
11         << showpos << i << "\n"      //printf("%+d", i)
12         << hex << i << "\n"          //printf("%x", i)
13         << showbase << i << "\n"     //printf("%#x", i)
14         << uppercase << i << "\n"    //printf("%#X", i)
15         << nouppercase << noshowbase << noshowpos;
16
17     cout.setf(save, ios_base::showpos | ios_base::showbase |
18         ios_base::uppercase | ios_base::basefield);
19
20     cout << i << "\n";   //same base and format as line 10
21     return EXIT_SUCCESS;
22 }
```

`10`	*line 10: decimal*
`+10`	*line 11: after* `showpos`*, positive sign*
`a`	*line 12: after* `hex`*, hexadecimal*
`0xa`	*line 13: after* `showbase`*, shows the base prefix* `0x`
`0XA`	*line 14: after* `uppercase`
`10`	*line 20: same as line 10*

The width evaporates after one use

The “set width” manipulator `setw`, with the argument `3` in line 11, causes the next item to be output with at least three characters. That item, `bond`, is only a single-digit number, so it will be padded with two blanks for a total of three characters. The comment shows the equivalent `printf`.

Unlike the other manipulators, `setw` evaporates after one use. (See p. 1048 for how this is implemented.) No padding is applied to the items after the `bond`: the `"\n"` at the end of line 11, the next `bond` in line 12, etc. To pad another item, we would have to output another `setw`.

In C, the only padding characters are blank and zero, in the `printf`’s in lines 11 and 15 respectively. In C++, the `setfill` manipulator in line 15 will let us request any padding character. Lines 14 and 16 save and restore the padding character, even though it is unnecessary here.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iomanip>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     int bond = 7;
 9
10     cout << bond << "\n"             //printf("%d")
11         << setw(3) << bond << "\n"   //printf("%3d", i);
12         << bond << "\n";             //printf("%d", i)
13
14     char save = cout.fill();
15     cout << setw(3) << setfill('0') << bond << "\n"; //printf("%03d", bond);
16     cout.fill(save);                 //or cout << setfill(save);
17
18     cout << setw(3) << bond << "\n"; //same padding character as line 11
19     return EXIT_SUCCESS;
20 }
```

`7`	*line 10*
`  7`	*line 11: padded with two spaces*
`7`	*line 12:* `setw` *evaporated after one use*
`007`	*line 15: padded with two zeroes*
`  7`	*line 18: same as line 11*

Output a bool

By default, a `bool` is output as the number `1` or `0`. Lines 11 and 12 turn verbal output on and off. Lines 8 and 14 save and restore the `bool` format, even though it is unnecessary here.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     bool b = true;
 8     ios_base::fmtflags save = cout.flags();
 9
10     cout << b << " " << !b << "\n"
11         << boolalpha << b << " " << !b << "\n"
12         << noboolalpha << b << " " << !b << "\n";
13
14     cout.setf(save, ios_base::boolalpha);   //Restore only the bool format.
15     cout << b << " " << !b << "\n";         //same format as line 10
16     return EXIT_SUCCESS;
17 }
```

1 0	*line 10*
true false	*line 11: after* boolalpha
1 0	*line 12: after* noboolalpha
1 0	*line 15*

Output a double

As in C, the default precision for a double is six digits. Line 9 outputs the double rounded to this number of digits, just like the %g format of printf. The precision is the *total* number of digits, some to the left of the decimal point and some to the right.

To change the precision, call the manipulator setprecision in line 14. The double is now rounded to three significant digits; the equivalent format of printf is in the comment.

The maximum precision is DBL_DIG, a macro defined the header file <cfloat>. On my platform it is 15 digits, and line 8 takes full advantage of it. We will eventually discard this macro in favor of the “template” numeric_limits<double>::digits10. See pp. 745−747.

To use the precision as the number of digits to the right of the decimal point, rather than the number of significant digits, switch to the fixed or scientific formats in lines 15−16. As usual, the value is rounded, not truncated. fixed will display every digit to the left of the decimal point; see line 17 of max.C on p. 748 for an example.

Unfortunately, there are no manipulators to turn off fixed and scientific. To reset the two flags, line 17 must use the resetiosflags manipulator on p. 352.

Lines 10 and 18 save and restore the precision. A variable that holds the precision must be of data type streamsize, the type for counting characters that are output or input.

Lines 9 and 20 save and restore the format: fixed, scientific, or neither.

If a double value happens to be a whole number, it normally does not display a decimal point and fractional digits. You can change this with the showpoint manipulator. Also applicable to double output are showpos and, if the format is scientific, uppercase.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iomanip>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     double d = 6.66666666666666; //15 digits, including 1 to left of decimal
 9     ios_base::fmtflags save = cout.flags();
10     streamsize prec = cout.precision();
11     cout << "The default precision is " << prec << ".\n";
12
13     cout << d << "\n"                          //printf("%g", d);
14         << setprecision(3) << d << "\n"     //printf("%.3g", d);
15         << fixed << d << "\n"               //printf("%.3f", d);
16         << scientific << d << "\n"          //printf("%.3e", d);
17         << resetiosflags(ios_base::floatfield)<<d<<"\n" //printf("%.3g",d);
18         << setprecision(prec) << d << "\n"; //printf("%g", d);
19
20     cout.setf(save, ios_base::floatfield);
21     cout << d << "\n";                         //same format as line 13
22     return EXIT_SUCCESS;
23 }
```

`The default precision is 6.`	
`6.66667`	*line 13: total of six digits*
`6.67`	*line 14: total of three digits*
`6.667`	*line 15: fixed format, three digits to the right of the decimal point*
`6.667e+00`	*line 16: scientific format, three digits to the right of the decimal point*
`6.67`	*line 17: back to non-fixed, non-scientific format, still a total of three digits*
`6.66667`	*line 18: back to default precision*
`6.66667`	*line 21: same as line 13*

Output an array of characters

The `s` in line 8 is an eight-character string, not counting its terminating `'\0'`. Lines 14−15 output it with a width of ten, padding it with two characters (asterisks for visibility).

By default, the padding characters are output before the string, right-justifying it within its ten-character field. The manipulators `left` and `right` in lines 17 and 20 let us control the justification. We can even specify `internal` padding in line 23, which inserts the padding character between the sign and the rest of the number.

Lines 10 and 26 save and restore the three justification flags, `left`, `right`, and `internal`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <iomanip>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     char s[] = "John Doe";
 9     double d = 10.00;
10     ios_base::fmtflags save = cout.flags();
11
12     cout << "Pay to the order of " << s << " the amount of\n"
13
14         << "Pay to the order of " << setfill('*') << setw(10) << s
15             << " the amount of\n"
16
17         << "Pay to the order of " << left << setw(10) << s
18             << " the amount of\n"
19
20         << "Pay to the order of " << right << setw(10) << s
21             << " the amount of\n"
22
23         << internal << fixed << showpos << setprecision(2)
24             << setfill(' ') << setw(7) << d << "\n";
25
26     cout.setf(save, ios_base::adjustfield);   //restore only the 3 flags
27     cout << "Pay to the order of " << setw(10) << s << " the amount of\n";
28     return EXIT_SUCCESS;
29 }
```

`Pay to the order of John Doe the amount of`	*line 12*
`Pay to the order of **John Doe the amount of`	*lines 14−15: right justified by default*
`Pay to the order of John Doe** the amount of`	*lines 17−18: after* `left`
`Pay to the order of **John Doe the amount of`	*lines 20−21: after* `right`
`+ 10.00`	*lines 23−24: after* `internal`
`Pay to the order of   John Doe the amount of`	*line 27: same as lines 14−15*

Another use of `internal` padding is to insert the padding character between a base indicator and a number.

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

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     int i = 10;
 9
10     cout << hex << showbase << setfill('0')
11         << setw(10) << i << "\n"
12         << internal << setw(10) << i << "\n";
13
14     return EXIT_SUCCESS;
15 }
```

`00000000xa`	*line 11: wrong*
`0x0000000a`	*line 12: correct*

Input manipulators

`istream` objects such as `cin` have the same format flags as `ostream` objects. `istream`'s have input manipulators, which are "input" with the `>>` operator.

By default, integers are input in decimal because an `istream` is born with the `ios_base::dec` flag on. Line 12 will accept a number with a leading `0`, but the zero will be ignored. Line 12 will reject a number with a leading `0x`, but we didn't bother with error checking. We should have.

To permit octal input, the familiar `oct` appears in lines 15−17 as an input manipulator. When we "input" the `oct` from an `istream`, no characters are actually input. But inputting the `oct` makes a change to `cin`, causing all subsequent integers input from that stream to be read in octal. (Line 16 will reject a number with a leading `0x`.)

Line 20 does hex input (it will accept and ignore a leading zero), and 24 goes back to decimal. Line 29 resets the flags for the three bases. With all three turned off, we can now accept integer input in any base.

As before, lines 9 and 34 save and reset the three base flags. Input error checking omitted for brevity.

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

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <iomanip>
4 using namespace std;
5
```

```
 6 int main()
 7 {
 8     int i, j, k;   //uninitialized variables
 9     ios_base::fmtflags save = cin.flags();
10
11     cout << "Input an integer in decimal: ";
12     cin >> i;
13     cout << "In decimal, the integer is " << i << ".\n\n";
14
15     cout << "Input two integers in octal; leading 0 optional: ";
16     cin >> oct >> i >> j;
17     cout << "In decimal, the integers are " << i << ", " << j << ".\n\n";
18
19     cout << "Input two integers in hexadecimal; leading 0x optional: ";
20     cin >> hex >> i >> j;
21     cout << "In decimal, the integers are " << i << ", " << j << ".\n\n";
22
23     cout << "Input an integer in decimal: ";
24     cin >> dec >> i;
25     cout << "In decimal, the integer is " << i << ".\n\n";
26
27     cout << "Input 3 integers in any base.\n"
28         "Leading 0 for octal, 0x for hex, are now mandatory: ";
29     cin >> resetiosflags(ios_base::basefield) >> i >> j >> k;
30
31     cout << "In decimal, the integers are "
32         << i << ", " << j << ", " << k <<".\n";
33
34     cin.setf(save, ios_base::basefield);
35     return EXIT_SUCCESS;
36 }
```

```
Input an integer in decimal: 10
In decimal, the integer is 10.

Input two integers in octal; leading 0 optional: 10 010
In decimal, the integers are 8, 8.

Input two integers in hexadecimal; leading 0x optional: 10 0x10
In decimal, the integers are 16, 16.

Input an integer in decimal: 10
In decimal, the integer is 10.

Input 3 integers in any base.
Leading 0 for octal, 0x for hex, are now mandatory: 10 010 0x10
In decimal, the integers are 10, 8, 16.
```

Skip white space

By default, the `>>` operators discard any leading whitespace encounted before the value they are looking for. For example, the character that line 11 inputs into `c` is the first non-whitespace character. To get a fresh start we then ignore the rest of the input line: the next newline or 1000 characters, whichever comes first. (What if the line is longer than 1000? We will fix this with `numeric_limits<streamsize>::max()` on pp. 747−748.)

The `noskipws` manipulator in line 18 will prevent us from skipping white space. In this case `c` will be the very next character read. (`noskipws` works only if the value to be input is a character or string, not a number. White space is always skipped before numerical input.) Line 24 turns skipping back on.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     ios_base::fmtflags save = cin.flags();
 8
 9     cout << "Input a line:";   //no space after colon
10     char c;                    //uninitialized variable
11     cin >> c;
12     cin.ignore(1000, '\n');
13
14     cout << "The first non-whitespace character was '" << c
15         << "'; ignoring rest of line.\n\n";
16
17     cout << "Input another line:";
18     cin >> noskipws >> c;
19     cin.ignore(1000, '\n');
20     cout << "The first character was '" << c
21         << "'; ignoring rest of line.\n\n";
22
23     cout << "Input yet another line:";
24     cin >> skipws >> c;
25     cin.ignore(1000, '\n');
26     cout << "The first non-whitespace character was '" << c
27         << "'; ignoring rest of line.\n";
28
29     cin.setf(save, ios_base::skipws);
30     return EXIT_SUCCESS;
31 }
```

```
Input a line:   This line begins with three spaces.
The first non-whitespace character was 'T'; ignoring rest of line.

Input another line:   This line begins with three spaces.
The first character was ' '; ignoring rest of line.

Input yet another line:   This line begins with three spaces.
The first non-whitespace character was 'T'; ignoring rest of line.
```

Output a pointer

A pointer to any type of variable can be implicitly converted to a pointer to `void`. Therefore there is only one `operator<<`, taking a `const void *`, for printing a pointer. The `p` in line 13 and the `&i` in line 14 are passed to this `operator<<`. The pointer is output in the platform's conventional format, hexadecimal on mine.

When line 16 tries to print the address of a function, we get a nasty surprise: it prints as the number `1`. Line 17 shows where the `1` come from: it is actually the representation of the `bool` value `true`. Why was the pointer converted to a `bool`? A pointer to any *variable* can be implicitly converted into a pointer to `void`, but a pointer to a function cannot be. The only type to which a pointer to a function can be implicitly converted, and for which there is an `operator<<`, is `bool`. Since the pointer was non-zero it was converted to `true`, which prints out as the digit `1` or the word `true`.

We could print the address of `f` if we could convert it to a `void *`, but neither `static_cast` nor `reinterpret_cast` will convert a pointer to a function into a pointer to a non-function. Line 18 will not compile. Paradoxically, we can convert a pointer to a function into a non-pointer (line 19). I selected the data type `size_t` because it should be as wide as a pointer. Since `size_t` is an integer, it prints in decimal. We convert it into a pointer in line 20 to print it in hex.

We already saw this double cast in line 24 of `reinterpret_cast.C` on p. 67. We could avoid it by writing the primitive C cast in line 22. But don't succumb to this temptation. There is no way to search the program to find all the C casts.

Two types of pointers have their own `operator<<` function. The pointer `q` in line 25 is a pointer to `const char`, so we call the `operator<<` whose argument is a pointer to `char`. This function outputs the characters to which the pointer points, not the value of the pointer. A pointer to a `signed` or `unsigned char` is treated the same way.

To output the actual value of the pointer (the address of the pointed-to character), line 29 casts the pointer into a pointer to a different type of variable. `void *` is the only non-arbitrary choice.

The other type of pointer that has its own `operator<<` is a pointer to the specific type of function shown in line 6: one that takes and returns a reference to an `ostream`. This `operator<<` does not output the value of the pointer. It calls the function that the pointer points to. We will see the reason for this oddity on pp. 361−362.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 inline void f() {cout << "typical function\n";}
 6 inline ostream& g(ostream& ost) {return ost << "g is called, not printed\n.";}
 7
 8 int main()
 9 {
10     int i = 10;
11     int *p = &i;
12
13     cout << "The value of p is " << p << ".\n"     //printf("%p", p)
14         "The address of i is " << &i << ".\n\n";
15
16     cout << f << "\n"
17         << boolalpha << f << "\n"
18         //<< reinterpret_cast<const void *>(f) << "\n"   //won't compile
19         << reinterpret_cast<size_t>(f) << "\n"
20         << reinterpret_cast<const void *>(reinterpret_cast<size_t>(f))
21             << "\n"
22         << (const void *)f   //depricated
23         << "\n\n";
24
25     const char *q = "hello";
26
```

```
27     cout << "q points at the characters \"" << q   //printf("%s", q)
28         << "\".\n"
29         "The value of q is " << static_cast<const void *>(q) << ".\n\n";
30
31     cout << g;
32
33     return EXIT_SUCCESS;
34 }
```

`The value of p is 0xffbff174.`	*line 13: may be octal or decimal on other platforms*
`The address of i is 0xffbff174.`	*line 14: the same address*
`1`	*line 16: the address of* `f`*, converted to* `bool`
`true`	*line 17: the address of* `f`*, converted to* `bool`
`70288`	*line 19: the address of* `f`*, converted to* `size_t`
`0x11290`	*line 20: the same address, formatted as a* `void *`
`0x11290`	*line 22: the same address, produced by a C-style cast*
`q points at the characters "hello".`	*line 27: a pointer to a character*
`The value of q is 0x113f0.`	*line 29: the address of the* `h` *in* `hello`
`g is called, not printed`	*line 31*

Here are simplified definitions for the three `operator<<` functions that take pointers. The one that takes a pointer to a function (line 5) is short enough to be inline. The one that takes a pointer to a `char` (line 9) happens not to be a member function because it can do its work by calling a member function (`write`) of class `ostream`.

```
 1 class ostream {
 2     //etc.
 3 public:
 4     ostream& operator<<(const void *) {output the value of p; return *this;}
 5     ostream& operator<<(ostream& (*p)(ostream&)) {return p(*this);}
 6     //etc.
 7 };
 8
 9 inline ostream& operator<<(ostream& ost, const char *p)
10 {
11     ost.write(p, strlen(p));   //output the characters to which p points
12     return ost;
13 }
```

How a manipulator works

The `hex` output manipulator is actually a function declared in the header file `<iostream>`. Like an `operator<<` function, its argument and return value is an `ostream`—the same `ostream`, since it is passed and returned by reference. Recall that the `ostream` argument of an `operator<<` is implicit, since the `operator<<` is member function of class `ostream`. The `ostream` argument of `hex` is explicit, since `hex` is not a member of any class.

```
ostream& hex(ostream& ost)
{
    ost.setf(ios_base::hex, ios_base::basefield);
    return ost;
```

```
}
```

The name of a function, with no argument list after it, stands for the address of that function. The expression `hex` is a therefore a pointer to a function that takes and returns an `ostream`. This function is like a tiny, time-release capsule: it lies dormant until it is fed to the `operator<<` that "outputs" it. When we write

```
    cout << hex
```

we are therefore calling the `operator<<` function that takes a pointer to this particular type of function.

```
    cout.operator<<(hex)
```

As we saw on pp. 360−361, this `operator<<` does not output the value of the pointer. It calls the function to which the pointer points, in this case the `hex` function.

We could also define a `hex` input manipulator as follows.

```
istream& hex(istream& ist)
{
    ist.setf(ios_base::hex, ios_base::basefield);
    return ist;
}
```

If we wrote

```
    cin >> hex
```

the computer would behave as if we have said

```
    cin.operator>>(hex)
```

calling the function

```
class istream {
    //etc.
public:
    istream& operator>>(istream& (*p)(istream&)) {return p(*this);}
    //etc.
};
```

But when we have inheritance it will be unnecessary to define the same manipulator twice. See pp. 484−485.

Our first example of an i/o manipulator was the `endl` on p. 26. Here is a simplified definition for it.

```
ostream& endl(ostream& ost)
{
    ost << '\n';
    ost.flush();
    return ost;
}
```

Extend the format of an ostream

The class `point` in pp. 201−204 had a `print` member function for output only to `cout`. We will replace it by an `operator<<` friend for output to any `ostream`. We will also invent two i/o manipulators, `cartesian` and `polar`, to output a `point` in these two coördinate systems. A demonstration is in lines 10−12 of `main.C` on p. 365; the default in line 10 is Cartesian.

Each `ostream` object already contains data members to keep track of whether it should output in decimal or hex, justified left or right, padded with blanks or zeroes, etc. Can we add another data member to hold its choice of coördinate system? No. The number of data members of a class is fixed, once and for all, by the declaration for that class.

But class `ostream` has a special feature that gives us the effect of an extra data member. Each `ostream` object contains an expandable array of `long`'s. Although each object has its own array, the arrays are all the same length. To add a new element to the array of every `ostream` object simultaneously, we call the `xalloc` static member function of class `ostream` in line 6 of `point.C`. In each array the new element has the same subscript, which is returned by `xalloc`.

The new element must be added to the array of each `ostream` before any `point` object is output to any `ostream`. To ensure this, the call to `xalloc` is used to initialize a static data member of class `point`. The static data members of a class are always initialized before any object of that class is ever constructed, let alone output. (This static data member cannot be initialized in its declaration in line 10 of `point.h`, even though it is integral and constant, because its initial value is not a constant expression. See p. 238.)

The new element of each array is initialized to zero, which is why we chose zero to represent the default format, Cartesian, in the element. The arrays have no name. To access an element of an `ostream`'s array, we pass its subscript to the `ostream`'s member function `iword` in line 10 of `point.C`. Note that `xalloc` is a static member function that affects all the `ostream`'s simultaneously, while `iword` is a non-static member function that accesses the array of the `ostream` of which it is a member function. The call to `iword` is in an `operator<<` with familiar arguments and return value, but also with a `switch` statement and a light dusting of trigonometry.

The manipulators `cartesian` and `polar` in `main.C` are actually two functions, the friends of class `point` in lines 16 and 21 of `point.h`. Like the `hex` function, they take an `ostream` object and return the same object. Along the way, they assign a value to the new element of the `ostream`. We can use the return value of `iword` as an lvalue in lines 17 and 22 because it is a read/write reference to the element. See pp. 12−13.

`cartesian` and `polar` mention the private member `subscript` of class `point`, so they must be member functions or friends of that class. If they were member functions, they could be static because they need no implicit pointer argument. In fact, they would have to be static because the pointer argument `p` of the `operator<<` in line 5 on p. 361 can point only to a free function. A different type of pointer would be needed to point to a non-free function. See p. 113 for free and non-free functions; p. 242 for static member functions as free functions; pp. 255−257 for pointers to non-free functions.

Had `cartesian` and `polar` been static member functions, we would have had to write `point::cartesian` and `point::polar` in lines 11−14 of `main.C`. We therefore define them as friends, to eliminate the last name. Since they are defined in the class definition and have no arguments of type point or compounded therefrom, they must also be declared outside the class at lines 6−7. See p. 206.

We are on a first-name basis with the members of a class within the curly braces of the class declaration. That's why lines 17 and 22 of `point.h` can mention the `subscript`. We are also on a first-name basis with the members of a class within the definition of a member function of the class. But the `operator<<` in line 8 of `point.C` is not a member function, so its line 10 must say `point::subscript`.

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

```
 1 #ifndef POINTH
 2 #define POINTH
 3 #include <iostream>
 4 using namespace std;
 5
 6 ostream& cartesian(ostream& ost);
 7 ostream& polar(ostream& ost);
 8
 9 class point {
10     static const int subscript; //subscript of new element in iword "array"
11     double x, y;
```

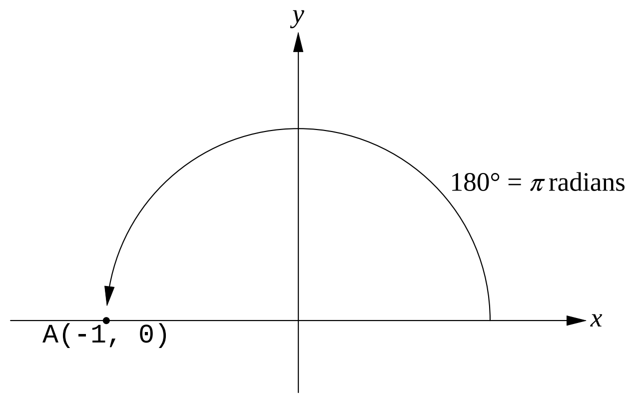
```
public:
    point(double initial_x = 0, double initial_y = 0)
        : x(initial_x), y(initial_y) {}

    friend ostream& cartesian(ostream& ost) {
        ost.iword(subscript) = 0;   //Cartesian coordinates
        return ost;
    }

    friend ostream& polar(ostream& ost) {
        ost.iword(subscript) = 1;   //polar coordinates
        return ost;
    }

    friend ostream& operator<<(ostream& ost, const point& p);
};
#endif
```

On my platform, the `atan2` function might set the ‘‘error number’’ variable `errno` if both of its arguments are zero. To avoid this, we call `atan2` only if at least one argument is not zero. Line 19 must output its zeroes as numbers, rather than as the string `"(0, 0)"`, to respond to the `fixed`, `scientific`, and `setprecision` manipulators,

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

```
 1 #include <cstdlib>   //for exit, EXIT_FAILURE
 2 #include <cmath>     //for sqrt and atan2
 3 #include "point.h"
 4 using namespace std;
 5
 6 const int point::subscript = ostream::xalloc();
 7
 8 ostream& operator<<(ostream& ost, const point& p)
 9 {
10     switch (ost.iword(point::subscript)) {
11
12     case 0:
13         //Cartesian coordinates.
14         return ost << "(" << p.x << ", " << p.y << ")";
15
16     case 1:
17         //Polar coordinates.
18         if (p.x == 0.0 && p.y == 0.0) {
19             return ost << "(" << 0.0 << ", " << 0.0 << ")";
20         } else {
21             return ost << "(" << sqrt(p.x * p.x + p.y * p.y) << ", "
22                 << atan2(p.y, p.x) << ")";
23         }
24
25     default:
26         cerr << "iword(" << point::subscript << ") == "
27             << ost.iword(point::subscript)
28             << " is neither 0 (Cartesian) nor 1 (polar).\n";
29         exit(EXIT_FAILURE);
30     }
```

```
}
```



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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "point.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const point A(-1, 0);
 9
10     cout << A << "\n"               //operator<<(cout, A)
11         << polar << A << "\n"
12         << cartesian << A << "\n";
13
14     cerr << polar << A << "\n";
15
16     return EXIT_SUCCESS;
17 }
```

When we write the above line 14, the computer behaves as if we had written

```
18     operator<<(operator<<(operator<<(cerr, polar), A), "\n");
```

When we write the above lines 10−12, the computer behaves as if we had written

```
19     operator<<(
20         operator<<(
21             operator<<(
22                 operator<<(
23                     operator<<(
24                         operator<<(
25                             operator<<(
26                                 operator<<(
27                                     cout,
28                                 A),
29                             "\n"),
30                         polar),
31                     A),
32                 "\n"),
33             cartesian),
```

```
34          A),
35      "\n");
```

`(-1, 0)`	*line 10: cartesian by default*
`(1, 3.14159)`	*line 11: radius == 1, θ == π radians*
`(-1, 0)`	*line 12: back to cartesian*
`(1, 3.14159)`	*line 14: standard error output*

A more general version of `cartesian` and `polar`, applicable to input as well as output, will be presented on pp. 485−486 after we have inheritance.

```
36      point A;
37      cin >> polar >> A >> cartesian;
```

For another pair of user-defined i/o manipulators, see p. 989.

To output a `point` in different formats was easy: we simply wrote a smart `operator<<` for class `point`. To output an `int` as a Roman or Arabic numeral would be harder: the `operator<<` for type `int` has already been written and engraved in granite in the Standard Library. We will need a different approach; see pp. 1047−1050.

▼ Homework 4.1.1b: output a date in French Revolutionary format

Define two manipulators to switch the output of a `date` to and from French Revolutionary format. This artificial calendar is simpler than any traditional calendar. Also define two public, static member functions of class `date` to save and restore the political format of an `ostream`.

```
date first(date::september, 22, 1792);      //Republic proclaimed

bool save = date::get_format(cout);

cout << first << "\n"
    << revolutionary << first << "\n"
    << norevolutionary << first << "\n\n";

date last(date::july, 27, 1794);            //Robespierre arrested

cout << last << "\n"
    << revolutionary << last << norevolutionary << "\n";

date::set_format(cout, save);
```

```
9/22/1792
1 Vendémiaire de l'Année I de la Républic
9/22/1792

7/27/1794
9 Thermidor de l'Année II de la Républic
```

Each month in this calendar is the same length, 30 days. The first month, *Vendémiaire,* begins on September 22. (That date in 1792 was the autumnal equinox and the day after the proclamation of the Republic.) The Year I of this calendar therefore begins

$$1791 \times 365 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 22 - 1$$

days after year 1 of the normal calendar. Ignore leap years. Don't bother with the accent marks or Roman numerals until we get to p. 1050.

	name	*translation*	*equivalent*
1	*Vendémiaire*	Vintage	September−October
2	*Brumaire*	Mist	October−November
3	*Frimaire*	Frost	November−December
4	*Nivôse*	Snow	December−January
5	*Pluviôse*	Rain	January−February
6	*Ventôse*	Wind	February−March
7	*Germinal*	Seed	March−April
8	*Floréal*	Blossom	April−May
9	*Prairal*	Meadow	May−June
10	*Messidor*	Harvest	June−July
11	*Thermidor*	Heat	July−August
12	*Fructidor*	Fruits	August−September

The twelve months total 360 days. The last five days of the year (the *sans-culottides,* those without knee breeches) have special names.

name	*translation*	*equivalent*
Fete de la vertu	Festival of Virtue	September 17
Fete du génie	Festival of Talent	September 18
Fete du travail	Festival of Industry	September 19
Fete de l'opinion	Festival of Ideas	September 20
Fete des recompenses	Festival of Rewards	September 21

Aux armes, citoyens! Formez vos bataillons!

▲

Define a manipulator with an argument

We have seen a number of manipulators that take arguments: `setbase`, `setw`, `setfill`, and `setprecision`. We now create one of our own.

The `set_life_foreground` manipulator in line 24 of `main.C` on p. 370 will change the format in which a `life` object is output. It takes an argument giving the character with which each occupied location should be drawn. This regains half of the functionality lost when the `print` member function of class `life` became an `operator<<` friend on p. 341.

To concentrate on the new features of this class, we have stripped away most of the overloaded operators. The Three Laws have been compressed into the single expression in line 42 of `life.C`.

For convenience, the same header file contains classes `set_life_foreground` and `life`; we would never use the former without the latter. The expression `set_life_foreground('O')` in line 24 of `main.C` calls the constructor for an anonymous object of this class and passes it an argument. The object stores the argument in its `c` data member (line 13 of `life.h`) and then lies dormant.

The `set_life_foreground` objects awakens when it is fed to its `operator<<` in line 57 of `life.C`. Long before this happens, however, line 4 of `life.C` has called `xalloc` to add a new element to the expandable array in every `ostream` object. The awakened `set_life_foreground` object stores its character data member into the new element in line 59 and plays no further rôle. Some time later, when a `life` object is fed to *its* `operator<<` in line 63 of `life.C`, the character is fetched from the array element and is used to display the `life` object.

There is no guarantee that a `set_life_foreground` object will be "output" before a `life` object is. Line 66 therefore defaults to `'X'` if no foreground character has been established.

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

```
#ifndef LIFEH
#define LIFEH
#include <iostream>   //defines size_t
using namespace std;

const size_t life_ymax = 10;
const size_t life_xmax = 10;

typedef bool life_matrix_t[life_ymax][life_xmax];
typedef bool _life_matrix_t[life_ymax + 2][life_xmax + 2];

class set_life_foreground {
    const char c;
public:
    set_life_foreground(char initial_c): c(initial_c) {}
    friend ostream& operator<<(ostream& ost, const set_life_foreground& f);
};

class life {
    static const int subscript;   //subscript of new element in iword array

    int g;                        //generation number
    _life_matrix_t matrix;
public:
    life(const life_matrix_t initial_matrix);
    int generation() const {return g;}
    life& operator++();

    friend ostream& operator<<(ostream& ost, const life& li);
    friend ostream& operator<<(ostream& ost, const set_life_foreground& f);
};
#endif
```

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

```
#include "life.h"
using namespace std;

const int life::subscript = ostream::xalloc();

life::life(const life_matrix_t initial_matrix)
    : g(0)
{
    //Copy initial_matrix into matrix.
    for (size_t y = 1; y <= life_ymax; ++y) {
        for (size_t x = 1; x <= life_xmax; ++x) {
            matrix[y][x] = initial_matrix[y - 1][x - 1];
        }

        //left and right edges
        matrix[y][0] = matrix[y][life_xmax + 1] = false;
    }

    //top and bottom edges
```

```
    for (size_t x = 0; x < life_xmax + 2; ++x) {
        matrix[0][x] = matrix[life_ymax + 1][x] = false;
    }
}

life& life::operator++()
{
    _life_matrix_t newmatrix;   //uninitialized variable

    for (size_t y = 1; y <= life_ymax; ++y) {
        for (size_t x = 1; x <= life_xmax; ++x) {

            //How many of the 8 neighbors of element x, y are on?
            int count = -matrix[y][x];

            for (size_t y1 = y - 1; y1 <= y + 1; ++y1) {
                for (size_t x1 = x - 1; x1 <= x + 1; ++x1) {
                    count += matrix[y1][x1];
                }
            }

            // Laws of Survival, Birth, and Death
            newmatrix[y][x] = count==2 ? matrix[y][x] : count == 3;
        }
    }

    //Copy newmatrix into matrix.
    for (size_t y = 1; y <= life_ymax + 1; ++y) {
        for (size_t x = 1; x <= life_xmax + 1; ++x) {
            matrix[y][x] = newmatrix[y][x];
        }
    }

    ++g;
    return *this;
}

ostream& operator<<(ostream& ost, const set_life_foreground& f)
{
    ost.iword(life::subscript) = f.c;
    return ost;
}

ostream& operator<<(ostream& ost, const life& li)
{
    const long character = ost.iword(life::subscript);
    const char full = character == 0 ? 'X' : character;

    for (size_t y = 1; y <= life_ymax; ++y) {
        for (size_t x = 1; x <= life_xmax; ++x) {
            cout << (li.matrix[y][x] ? full : '.');
        }
        cout << "\n";
    }
```

```

    return ost;
}
```

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

```
#include <iostream>
#include <cstdlib>
#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;

    cout << glider << "\n"
        << set_life_foreground('O') << ++glider;   //uppercase letter O

    return EXIT_SUCCESS;
}
```

```
..........          line 23: foreground character defaults to X
.X........
..XX......
.XX.......
..........
..........
..........
..........
..........
..........

..........          line 24
..O.......
...O......
.OOO......
..........
..........
..........
..........
..........
..........
```

▼ **Homework 4.1.1c: improvements to set_life_format**

(1) What happens if we say

```
cout << set_life_foreground('\0');
```

Have the constructor for class `set_life_foreground` disallow this by checking that its argument is a printable character. (Call the `isprint` function in the standard library and include the header file `<cctype>`.) On the other hand, should we allow `'\0'` as a special value that sets the foreground character back to its default? In this case, we would still disallow all other nonprintable characters.

(2) Make it possible for the user to save the foreground character for later restoration.

```
//a public static member function of class life
char save = life::get_foreground(cout);

...

cout << set_life_foreground(save);   //restore the previous foreground
```

(3) We could create another array element to hold the background character. Should we have separate manipulators for the foreground and background characters,

```
cout << set_life_foreground('F') << set_life_background('b');
```

or a single constructor whose manipulator takes two arguments? The second argument could be optional.

```
cout << set_life_format('F', 'b');
```

▲

An argument that will not fit into a long integer

Let's remove the `cartesian` and `polar` manipulators from the class `point` on pp. 362−366, and invent a new manipulator for scaling a `point`. The `scale` in lines 11−12 of `main.C` on p. 374 will take an argument giving the factor by which the coördinates of each `point` should be multiplied.

Long before any `point` object is output or even constructed, line 6 of `point.C` will call `xalloc` to add a new element to the expandable array in every `ostream` object. Each `ostream` actually has a pair of arrays, always agreeing in their number of elements. We have seen the array of `long`'s accessed by the `iword` member function of the `ostream`; a parallel array of `void *`'s is accessed by the `pword` member function. Like `iword`, `pword` returns a read/write reference to an array element. This lets us store a new value into the element. If a format value is integral or an enumeration, it can be stored in an element of the `iword` array. Otherwise, the address of the value can be stored in an element of the `pword` array.

The expression `scale(2.54)` in line 11 of `main.C` calls the constructor for an anonymous object of class `scale`, passing it one argument. The `scale` object stores the argument in its `factor` data member and then lies dormant. It awakens when it is fed to the `operator<<` function in line 15 of `point.C`. Line 27 of this function copies the data member into the new element of the `pword` array; more precisely, into the block of memory to which the new element points.

Where did this block come from? For convenience, line 17 of `point.C` creates a reference `p` to the new element. The element is a pointer to `void`; `p` is a reference to a pointer to `void`. The initial value of each element in the `pword` array is a zero pointer, just as the initial value of each element in the `iword` array was a zero `long`. If line 19 finds that we have never assigned a value to the new element, line 20 will store the address of a block of memory there. To permit this assignment to be made through `p`, p had to be a read/write reference to the array element. If we removed the & from line 17, `p` would be merely a copy of the array element, not a reference thereto. The assignment to `p` in line 20 would then put a value only into `p`, leaving the array element unchanged. Line 24 is discussed below.

Could the `malloc` somehow go up in the static initialization in line 6? No. Line 6 is performed only once, but the `malloc` must be called for each `ostream` object to which we output a `scale`. The `malloc`, by the way, is only temporary. It will be superseded by the C++ operator `new`.

To output a `point`, we call the `operator<<` function in line 8 of `point.C`. For convenience, line 10 creates a copy of the new element of the `pword` array. This `p` can be a copy, not a reference, because this `operator<<` has no interest in changing the value of the new array element. If line 11 finds that we have never assigned a value to the new element, it means that no `scale` object has been output to this `ostream` yet. In that case, line 11 assumes a default scale of `1.0`; otherwise, it fetches the factor stored by line 27. Finally, the `x` and `y` data members are output with a light dusting of multiplication.

If line 20 successfully allocates a block of memory, line 24 arranges to have it deallocated when the `ostream` is destructed. The first argument of the `register_callback` function in line 24 is the address of a *callback* function to be called at some future event. The second argument of `register_callback` will be passed to the callback function when the callback function is called.

The callback function will be called on three types of occasions, represented by the three enumerations in lines 37, 41, and 53. If we recieve an illegal enumeration (lines 56−57), we do not attempt to output an error message because the streams are messed up so badly. Our concern here is with line 37, the case in which the `ostream` is destructed. Line 38 frees the block of memory that was allocated in line 20.

The callback function is also called after all the formatting information of a stream is copied into another stream, including the pointers in the `pword` array. We do not want two different streams to have pointers to the same block of memory. Line 43 saves a pointer to the block, line 44 creates a new block for this `ostream`, and lines 48−49 copy the contents of the old block into the new one.

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

```
#ifndef POINTH
#define POINTH
#include <iostream>
#include <cstdlib>
using namespace std;

class scale {
    const double factor;
```

```
public:
    scale(double initial_factor): factor(initial_factor) {}
    friend ostream& operator<<(ostream& ost, const scale& s);
};

class point {
    static const int subscript;   //subscript of new element in pword array
    static void callback(ios_base::event e, ios_base& ost, int i);

    double x, y;
public:
    point(double initial_x = 0, double initial_y = 0)
        : x(initial_x), y(initial_y) {}

    friend ostream& operator<<(ostream& ost, const point& pt);
    friend ostream& operator<<(ostream& ost, const scale& s);
};
#endif
```

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

```
#include <iostream>
#include <cstdlib>   //for malloc, exit, EXIT_FAILURE
#include "point.h"
using namespace std;

const int point::subscript = ostream::xalloc();

ostream& operator<<(ostream& ost, const point& pt)
{
    const void *const p = ost.pword(point::subscript);
    const double factor = p == 0 ? 1.0 : *static_cast<const double *>(p);
    return ost << "(" << factor * pt.x << ", " << factor * pt.y << ")";
}

ostream& operator<<(ostream& ost, const scale& s) //friend of two classes
{
    void *& p = ost.pword(point::subscript);      //a reference to a pointer

    if (p == 0) {                                 //if the pointer is 0
        if ((p = malloc(sizeof (double))) == 0) {
            cerr << "scale operator<< out of store\n";
            exit(EXIT_FAILURE);
        }
        ost.register_callback(point::callback, point::subscript);
    }

    *static_cast<double *>(p) = s.factor;
    return ost;
}

void point::callback(ios_base::event e, ios_base& ost, int i)
{
    void *& p = ost.pword(i);
```

```

    switch (e) {

    case ios_base::erase_event:
        free(p);
        break;

    case ios_base::copyfmt_event:
        if (p != 0) {
            const void *const q = p;
            if ((p = malloc(sizeof (double))) == 0) {
                cerr << "point::callback out of store\n";
                exit(EXIT_FAILURE);
            }
            *static_cast<double *>(p) =
                *static_cast<const double *>(q);
        }
        break;

    case ios_base::imbue_event:
        break;

    default:
        exit(EXIT_FAILURE);
    }
}
```

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "point.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const point A(1, 2);             //in inches
 9
10     cout << A << "\n"
11         << scale(2.54) << A << "\n"  //1 inch == 2.54 centimeters
12         << scale(1) << A << "\n";
13
14     return EXIT_SUCCESS;
15 }
```

`(1, 2)`	*line 10: display the* `point` *in inches*
`(2.54, 5.08)`	*line 11: centimeters*
`(1, 2)`	*line 12: back to inches*

▼ Homework 4.1.1d: make a typedef

The data type `double` appears many times throught classes `point` and `scale`. Make a typedef for `double` named `value_type` at line 6 of the above `point.h`.
▲

▼ **Homework 4.1.1e: move the multiplication to the correct place**

We should never have buried the double-barreled multiplication in a place like line 12 of the above `point.C`. This multiplication should be written once and for all in an `operator*=` for class `point`.

```
class point {
    //etc.
public:
    point& operator*=(double d) {x *= d; y *= d; return *this;}
```

Now that we can multiply a `point` by a `double`, we should change the `operator<<` function to the following.

```
ostream& operator<<(ostream& ost, point pt)  //point now passed by value
{
    if (const void *const p = ost.pword(point::subscript)) {
        pt *= *static_cast<const double *>(p);
    }
    return ost << "(" << pt.x << ", " << pt.y << ")";
}
```

It may be objected that we are now constructing a new object, since the `point` must be passed to the `operator<<` by value. But the same object was constructed piecemeal in line 12 of the above `point.C`. Each multiplication there created a `double` anonymous temporary to hold the product, so we were constructing the equivalent of a two-data-member object. It's clearer to make the object official.

▲

▼ **Homework 4.1.1f: copyfmt**

We can copy the format (base, justification, precision, etc.) of one stream to another:

```
    cout << scale(2.54);
    cerr.copyfmt(cout);   //Copy the format of cout to cerr.
                          //Now cerr has scale 2.54 too.
```

When this happens, the `iword` and `pword` arrays are copied from `cout` to `cerr`, and then the callback function of `cout` is called with the argument `ios_base::copyfmt_event`.

What does the following fragment output?

```
    const point A(1, 2);
    cout << scale(2.54);
    cerr.copyfmt(cout);
    cout << scale(1);
    cerr << A << "\n";   //should output with scale 2.54
```

How does the output change when we remove lines 41−51 of the above `point.C`?

▲

4.1.2 File I/O with Classes `ostream` and `istream`

Class ofstream is derived from class ostream

The C function `printf` is quite capable of outputting to a file: just run the program from the command line using the file output symbol `>`.

```
prog > outfile
```

Why, then, did they invent the trio of functions `fopen`, `fprintf`, and `fclose`? For two reasons:

(1) All the `printf`'s and `putchar`'s in a C program send their output to the *same* destination, which may be a file. But to send output to two or more different destinations, e.g., two output files, we must use the `fprintf` trio. The following program is an example.

(2) Even if there is only one output file, we might still want to use the `fprintf` trio. `printf` gives the program no control over the name of the output file, the name of the directory that will hold the file, or whether the file will be opened in overwrite or append mode. All of these things can be specified with the `fprintf` trio.

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

```
#include <stdio.h>   /* C example */
#include <stdlib.h>

int main(int argc, char **argv)
{
    FILE *out1;
    FILE *out2;

    /* Open two output files.
    Clobber them if they already exist; create them if they don't. */

    out1 = fopen("outfile1", "w");
    if (out1 == NULL) {
        fprintf(stderr, "can't open outfile1.\n");
        return EXIT_FAILURE;
    }

    out2 = fopen("outfile2", "w");
    if (out2 == NULL) {
        fprintf(stderr, "can't open outfile2.\n");
        return EXIT_FAILURE;
    }

    fprintf(out1, "hello\n");   /* Output 6 char's.  Do not output '\0'. */
    fprintf(out2, "goodbye\n");

    fclose(out1);
    fclose(out2);
    return EXIT_SUCCESS;
}
```

`hello`	*This file is* `outfile1`.

`goodbye`	*This file is* `outfile2`.

We saw the above scenario in pp. 164−166: a pair of events, with data (the variable `out1`) that persists from the first event to the second. In C++, we tie this all together by constructing and destructing an object of class `ofstream`, for "output file stream". The constructor opens an output file, and the destructor closes it. As usual, the destructors are called implicitly.

Construct an object of class `ofstream` to perform file output. An object of class `ofstream` (such as `out1` and `out2`) can do everything that an object of class `ostream` (such as `cout` and `cerr`) can do: `<<`, `hex`, `precision`, etc.; line 33 demonstrates `setw`. This is because `ofstream` is derived from `ostream`. Furthermore, an object of class `ofstream` can do *more* than an object of class `ostream`: it lets us specify the name and directory of the destination file, and whether we're overwriting or appending to it.

See p. 327 for the use of `!` in the tests in lines 21 and 27. The `if` in line 21 is true when the constructor called in line 20 failed to open `outfile1` successfully. In this case, `out1` requires no destruction.

But our program still has a bug. Suppose that line 20 constructed the object `out1`, but line 26 failed to construct the object `out2`. In this case `out2` requires no destruction, but `out1` does. Unfortunately, the `exit` in line 29 will terminate the program without destructing `out1`. We'll fix this bug when we do exceptions.

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

```
 1 #include <iostream>   //C++ example
 2 #include <fstream>    //for ofstream
 3 #include <iomanip>    //for setw
 4 #include <cstdlib>
 5 using namespace std;
 6
 7 void f();
 8
 9 int main(int argc, char **argv)
10 {
11     f();
12     return EXIT_SUCCESS;
13 }
14
15 void f()
16 {
17     //The constructors called in lines 20 and 26 open two output files.
18     //Clobber the files if they already exist; create them if they don't.
19
20     ofstream out1("outfile1");
21     if (!out1) {                     //if (out1.operator!()) {
22         cerr << "can't open outfile1.\n";
23         exit(EXIT_FAILURE);
24     }
25
26     ofstream out2("outfile2");
27     if (!out2) {
28         cerr << "can't open outfile2.\n";
29         exit(EXIT_FAILURE);
30     }
31
32     out1 << "hello\n";               //Output 6 char's.  Do not output '\0'.
33     out2 << setw(8) << "goodbye" << "\n";
34 }   //Call destructors for out2 and out1.
```

`hello`	*This file is* `outfile1`.

`  goodbye`	*This file is* `outfile2`.

A constructor with a default argument

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

```
#include <stdio.h>    /* C example: K&R C book, pp. 160-161, 242 */
#include <stdlib.h>

int main()
{
    FILE *out1;
    FILE *out2;

    out1 = fopen("outfile1", "w");   /* overwrite */
    if (out1 == NULL) {
        fprintf(stderr, "can't open outfile1\n");
        return EXIT_FAILURE;
    }

    out2 = fopen("outfile2", "a");   /* append */
    if (out2 == NULL) {
        fprintf(stderr, "can't open outfile2\n");
        return EXIT_FAILURE;
    }

    fprintf(out1, "hello\n");
    fprintf(out2, "goodbye\n");

    fclose(out1);
    fclose(out2);
    return EXIT_SUCCESS;
}
```

The constructor for class `ofstream` has an optional second argument, which is an integer whose bits specify in greater detail how to open the file. Each bit has an enumeration that provides a convenient name for it; the value of the enumeration is a number with that bit turned on and the rest turned off.

For example, the default value for the second argument is the enumeration `ios_base::out`, causing the constructor to open the file as an output file. When no other bits in the argument are turned on, this also truncates the file as it is opened. Another possible argument is the `ios_base::app` in line 14, which appends to the file instead of truncating it.

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

```
 1 #include <iostream>   //C++ example
 2 #include <fstream>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     ofstream out1("outfile1");                  //overwrite
 9     if (!out1) {                                //if (out1.operator!()) {
10         cerr << "can't open outfile1\n";
11         return EXIT_FAILURE;
12     }
```

```

    ofstream out2("outfile2", ios_base::app);   //append
    if (!out2) {
        cerr << "can’t open outfile2\n";
        return EXIT_FAILURE;
    }

    out1 << "hello\n";
    out2 << "goodbye\n";
    return EXIT_SUCCESS;   //Call destructors for out2 and out1.
}
```

Class ifstream is derived from class istream

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

```
#include <stdio.h>   /* C example */
#include <stdlib.h>

int main(int argc, char **argv)
{
    FILE *in1;
    FILE *in2;
    int i;
    int j;

    /* Open two input files. */

    in1 = fopen("infile1", "r");
    if (in1 == NULL) {
        fprintf(stderr, "can’t open infile1.\n");
        return EXIT_FAILURE;
    }

    in2 = fopen("infile2", "r");
    if (in2 == NULL) {
        fprintf(stderr, "can’t open infile2.\n");
        return EXIT_FAILURE;
    }

    fscanf(in1, "%d", &i);
    fscanf(in2, "%d", &j);

    printf("%d %d\n", i, j);

    fclose(in1);
    fclose(in2);
    return EXIT_SUCCESS;
}
```

In C++, we open and close two input files by making two objects of class `ifstream`, for ‘‘input file stream’’. An object of class `ifstream` (such as `in1` and `in2`) can do everything that an object of class `istream` (such as `cin`) can do (plus more): `>>`, `!`, etc. This is because `ifstream` is derived from `istream`.

`ostream` and `istream` are both derived from `ios_base`.

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

```
 1 #include <iostream>   //C++ example
 2 #include <fstream>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     //The constructors called in lines 10 and 16 open two input files.
 9
10     ifstream in1("infile1");
11     if (!in1) {                   //if (in1.operator!()) {
12         cerr << "can't open infile1.\n";
13         return EXIT_FAILURE;
14     }
15
16     ifstream in2("infile2");
17     if (!in2) {
18         cerr << "can't open infile2.\n";
19         return EXIT_FAILURE;
20     }
21
22     int i;                        //uninitialized variable
23     in1 >> i;
24
25     int j;                        //uninitialized variable
26     in2 >> j;
27
28     cout << i << " " << j << "\n";
29     return EXIT_SUCCESS;
30 }
```

Class fstream is derived from both ifstream and ofstream

To open a file for both reading and writing in C, the second argument of the `fopen` in line 12 must be either `"r+"` and `"w+"`. `"w+"` destroys the file's previous contents, if any; `"r+"` doesn't.

The `fprintf` in line 18 writes the word `hello` at the beginning of the file; the `fscanf` in line 43 reads the word from the file. Between these two lines, we need the `fseek` in line 31 to rewind the file back to the beginning. The calls to `ftell` before and after the `fseek`, in lines 24 and 36, display our current position in the file.

The `long` variable `position` in line 9 holds our current position in the file. `ftell` gets the position and stores it into this variable; `fseek` sets the position from this variable. If the number of bytes in the file is too big to store in a `long`, we will have to upgrade to a variable of data type `fpos_t` and the pair of functions `fgetpos` and `fsetpos`.

The third argument of `fseek` in line 31 must be one of the following macros, defined in `stdio.h`:

`SEEK_SET` *offset from start of file*
`SEEK_CUR` *offset from current position*
`SEEK_END` *offset from end of file*

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

```
#include <stdio.h>   /* C example */
#include <stdlib.h>
#include <errno.h>   /* for errno */
#include <string.h>  /* for strerror */

int main(int argc, char **argv)
{
    FILE *fp;
    long position;
    char buffer[256];

    fp = fopen("file", "w+");
    if (fp == NULL) {
        fprintf(stderr, "can’t open file: %s\n", strerror(errno));
        return EXIT_FAILURE;
    }

    fprintf(fp, "hello\n");
    if (fflush(fp) != 0) {
        fprintf(stderr, "can’t fflush: %s\n", strerror(errno));
        return EXIT_FAILURE;
    }

    position = ftell(fp);
    if (position == -1) {
        fprintf(stderr, "can’t ftell: %s.\n", strerror(errno));
        return EXIT_FAILURE;
    }
    printf("position %ld\n", position);

    if (fseek(fp, 0, SEEK_SET) != 0) {   /* rewind file back to beginning */
        fprintf(stderr, "can’t fseek: %s.\n", strerror(errno));
        return EXIT_FAILURE;
    }

    position = ftell(fp);
    if (position == -1) {
        fprintf(stderr, "can’t ftell: %s.\n", strerror(errno));
        return EXIT_FAILURE;
    }
    printf("position %ld\n", position);

    if (fscanf(fp, "%s", buffer) != 1) {
        fprintf(stderr, "can’t fscanf\n");
        if (ferror(fp)) {
            fprintf(stderr, ": %s", strerror(errno));
        }
        fprintf(stderr, ".\n");
    }

    printf("%s\n", buffer);

    if (fclose(fp) != 0) {
        fprintf(stderr, "can’t fclose: %s.\n", strerror(errno));
```

```
55         return EXIT_FAILURE;
56     }
57
58     return EXIT_SUCCESS;
59 }
```

The standard output is

```
position 6        line 29
position 0        line 41
hello             line 51
```

The file `file` will contain

```
hello
```

To open a file for both reading and writing in C++, construct an object of class `fstream`. Its constructor, like those for classes `ofstream` and `ifstream`, has an optional second argument which is an integer whose bits specify in greater detail how to open the file. The second argument in line 10 is the value 19. But don't think of it as nineteen—think of it as `10011`: "yes, no, no yes, yes". It contains the answers to several independent yes/no questions:

name of enum	*value in binary*
`ios_base::in`	`00000000 00000001`
`ios_base::out`	`00000000 00000010`
`| ios_base::trunc`	`00000000 00010000`
	`00000000 00010011`

If `in` is specified, the file will be truncated only if we also specify `trunc`. If `in` is not specified, the file will be truncated even without the `trunc`. For example, the default value is `ios_base::in | ios_base::out`, which would not truncate the file.

C has only one pair of tell and seek functions, but C++ has two. Call `tellg` (line 18) and `seekg` (line 24) to get and set the position for reading; the `g` stands for "get". Call `tellp` and `seekp` to get and set the position for writing; the `p` stands for "put".

The optional second argument of `seekg` or `seekp` must be one of the enumerations

`ios_base::beg`	*offset from start of file (the default)*
`ios_base::cur`	*offset from current position*
`ios_base::end`	*offset from end of file*

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

```
 1 #include <iostream>   //C++ example
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include <cerrno>     //for errno
 5 #include <cstring>    //for strerror
 6 using namespace std;
 7
 8 int main(int argc, char **argv)
 9 {
10     fstream fstr("file", ios_base::in | ios_base::out | ios_base::trunc);
11     if (!fstr) {        //if (fstr.operator!()) {
12         cerr << "can’t open file: "<< strerror(errno) << ".\n";
13         return EXIT_FAILURE;
14     }
```

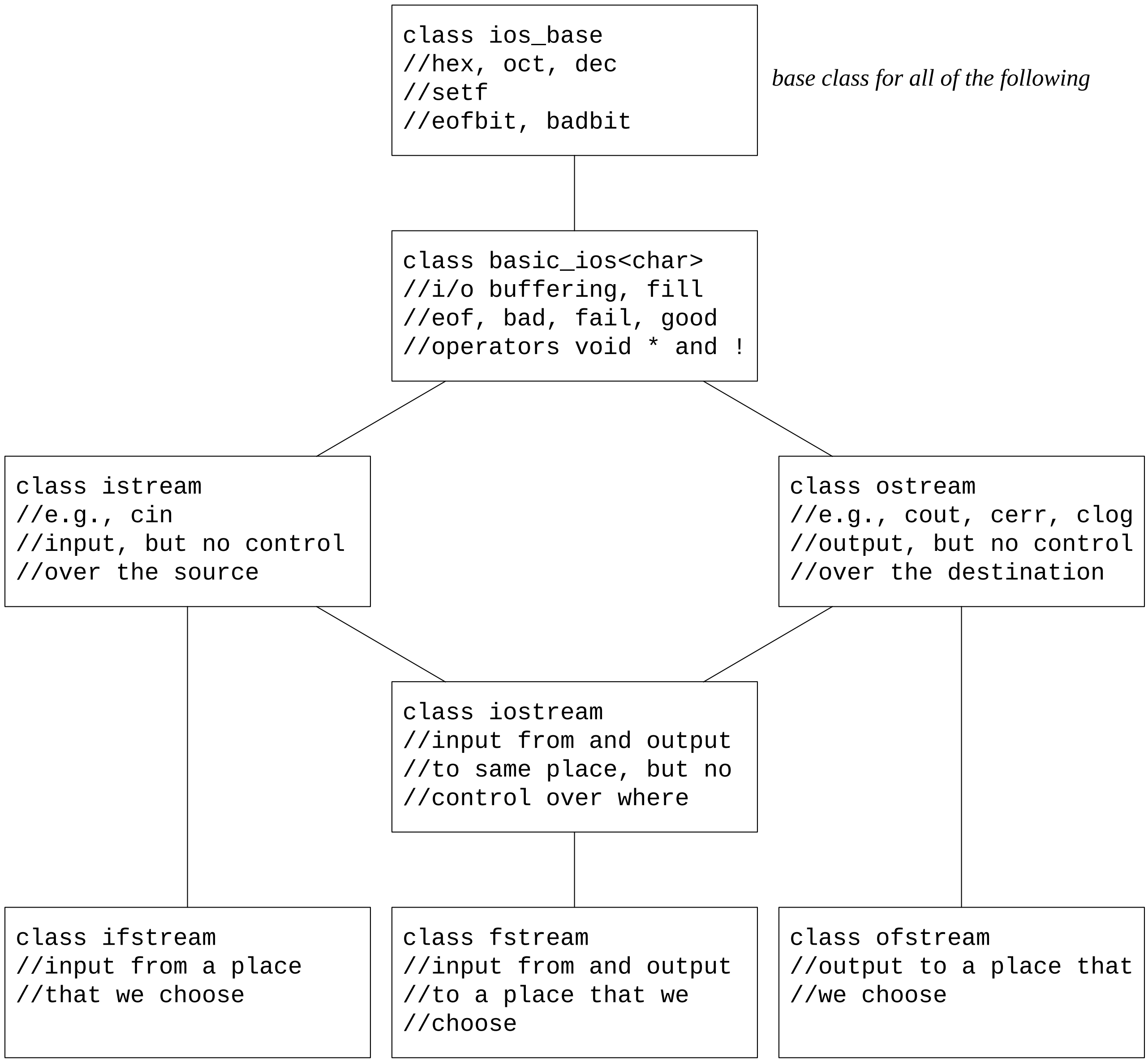
```
15
16     fstr << "hello\n" << flush;
17
18     cout << "input position == " << fstr.tellg() << "\n";
19     if (!fstr) {
20         cerr << "can’t tellg: " << strerror(errno) << ".\n";
21         return EXIT_FAILURE;
22     }
23
24     fstr.seekg(0);      //rewind file back to beginning
25     if (!fstr) {
26         cerr << "can’t seekg: " << strerror(errno) << ".\n";
27         return EXIT_FAILURE;
28     }
29
30     cout << "input position == " << fstr.tellg() << "\n";
31     if (!fstr) {
32         cerr << "can’t tellg: " << strerror(errno) << ".\n";
33         return EXIT_FAILURE;
34     }
35
36     char buffer[256];   //uninitialized variable
37     if (!(fstr >> buffer)) {
38         cerr << "can’t read from file.\n";
39         return EXIT_FAILURE;
40     }
41
42     cout << buffer << "\n";
43     return EXIT_SUCCESS;
44 }
```

`input position == 6`	*line 18*
`input position == 0`	*line 30*
`hello`	*line 42*

The file `file` will contain

```
hello
```

4.1.3 File I/O as a Preview of Inheritance



Class `ostream` is the right shoulder of the following diagram. The easiest way to remember what this class does is to think of its most famous objects: `cout` and `cerr`. An `ostream` object lets us perform output, but it gives us no control over the destination of the output.

Below class `ostream` is class `ofstream`. It provides all the functionality of class `ostream`, plus more. An `ofstream` lets us specify the name of the output file, and the name of the directory that holds the file. It also lets us specify the mode in which the file is opened: overwrite vs. append.

Class `ofstream` could have been written by copying and pasting most of the source code of class `ostream` into class `ofstream`. But it is never a good idea to have two copies of the same code. The day will come when someone fixes a bug in one copy and forgets to make the same fix in the other.

C++ gives us a better way to endow class `ofstream` with all the functionality of class `ostream`. There is a simple declaration, which we will see later, that lets us build a class with a head start. This declaration states that class `ofstream` should begin by having all the members of class `ostream`, plus additional members. This method of building a bigger class from a smaller one is called *inheritance.* The smaller class (`ostream`) is called the *base class;* the bigger and better one (`ofstream`) is called the *derived class.* In a diagram, the base class is always drawn above the derived class.

Classes `istream` and `ifstream` are another example of inheritance. An `istream` object such as `cin` lets us perform input, but it gives us no control over the source of the input. An `ifstream` provides all the functionality of class `istream`, plus more. It lets us specify the name of the input file, and the name of the directory that holds the file. In fact, an `ifstream` object is an improved `istream` object. This is the celebrated ‘‘is-a’’ relationship between a derived class and a base class.

C++ allows us to derive a class from more than one base; this is called *multiple inheritance.* Its absence in Java is one of the big differences between the two languages. For example, class `iostream` lets us perform input and output, although it gives us no control over the source of the input or the destination of the output. To offer this control, class `fstream` has been derived from `iostream`.

It would seem that the two shoulders, `istream` and `ostream`, are total opposites. But in fact, they have a lot in common. Both perform buffering; both let apply the `!` operator to check for error; they share manipulators such as `dec`, `oct`, and `hex`. The code that would be common to these two classes has been factored out and written once and for all in a base class `basic_ios<char>` and *its* base class `ios_base`. We've even seen some members of these ancestral classes: the enumerations `ios_base::failbit` in line 10 of `fail.C` on p. 332, and `ios_base::floatfield` in line 18 of `double.C` on p. 355.

The i/o classes are built in layers. The base class `ios_base` does not know what type of characters we are dealing with, `char` or `whar_t`. This knowledge is added in the next layer, `basic_ios<char>`. The <angle brackets> whow that this is a ‘‘template class’’.

Until now, our classes have been unrepresentative because they were created individually. In real life we often create a whole family of related classes. This family is our first example. Although we do not yet know how to create our own classes by means of inheritance, we can start using these stream classes that were created for us.

Why couldn't we build the above family using aggregation?

We can apply the same operators to an `ofstream` that we apply to an `ostream`; see lines 4−5.

```
 1      ostream cout(argument(s), if any, for constructor);     //in <iostream>
 2      ofstream out("outfile");
 3
 4      cout << "hello";                  //exactly the same operators
 5      out << "hello";
```

But if we had built class `ofstream` using aggregation,

```
 6 class ofstream {
 7 public:
 8      ostream os;
 9      //etc.
```

then we'd always need to mention the data member `os`:

```
10      cout << "hello";                  //all you need is <<
11      out.os << "hello";                //need .os in addition to <<
```

4.2 Dynamic Memory Allocation with `new` and `delete`

4.2.1 When is Dynamic Allocation Necessary?

The most common way to create a variable in C and C++ is with a declaration that is also a definition.

```
    int i = 10;
```

But in three situations the variable cannot be created this way.

(1) A variable constructed with a declaration has one of only two possible lifespans. If statically allocated, it is destructed when the program ends; if automatically allocated, it is destructed when we leave the block of statements in which it was defined. For these two storage classes and the definition of a ‘‘block’’, see pp. 180−185.

The following program has examples of these lifespans. The static variables are constructed once and for all in lines 5 and 18 and are destructed when the program ends in line 12. The automatic variable is constructed each time we arrive in line 17 and destructed each time we reach the closing curly brace in line 19.

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

```
 1 #include <cstdlib>
 2 #include "obj.h"
 3 using namespace std;
 4
 5 obj static_global = 10;           //static: destructed in line 12
 6 void f();
 7
 8 int main()
 9 {
10     f();
11     f();
12     return EXIT_SUCCESS;
13 }
14
15 void f()
16 {
17     obj automatic_local = 20;     //automatic: destructed in line 19
18     static obj static_local = 30; //static: destructed in line 12
19 }
```

`construct 10`	*line 5 constructs the global*
`construct 20`	*line 10 calls* `f`*, constructing the automatic at line 17*
`construct 30`	*line 18 constructs the static local*
`destruct 20`	`}` *in line 19 destructs the automatic*
`construct 20`	*line 11 calls* `f`*, constructing the automatic again at line 17*
`destruct 20`	`}` *in line 19 destructs the automatic again*
`destruct 30`	*return from* `main` *in line 12 destructs the statics*
`destruct 10`	

But we might need to give a different lifespan to a variable, perhaps constructing it in one function and destructing it in another. Such a variable could not be created with a declaration.

(2) A series of variables constructed with declarations, either all global or all defined in the same block, are always destructed in the reverse order. This discipline is called ‘‘last hired, first fired’’.

```
    obj o1 = 10;      //constructed first,  destructed third
    obj o2 = 20;      //constructed second, destructed second
    obj o3 = 30;      //constructed third,  destructed first
```

But we might need to destruct the variables in an order that cannot be predicted in advance. ‘‘In advance’’ means at *compile time:* when the program is written and compiled. For example, the current version of the rabbit game halts as soon as any rabbit is killed; the next version will continue until all of them are killed. We cannot predict in advance which rabbit the user will kill first, so they cannot be created with declarations. Not until *runtime*—when the program runs—will we know what order to destruct them in.

(3) An array can be constructed with a declaration only if we know at compile time how many elements it will have. But the following fragment does not know this number until runtime, so the array declaration will not compile.

```
#include <iostream>
#include <cstddef>   //for size_t
using namespace std;

    size_t n;        //uninitialized variable

    cout << "How many char’s do you want to allocate? ";
    cin >> n;

    char a[n];       //won’t compile: number of elements can’t be variable
```

Is there a way to create a variable without a declaration? Well, we can create it as an anonymous temporary. Here is one that holds the sum of `i` and `j`:

```
    cout << i + j << "\n";
```

But a temporary cannot outlive the expression in which it is created (unless it is referred to by a reference). As before, a variable needing a different lifespan must be created in a different way.

Two examples that do not need dynamic allocation

But let’s not go overboard. There are still plenty of situations in which variables can be created with declarations. For example, it is widely though erroneously believed that dynamic allocation is necessary when creating an unpredictable number of variables (unpredictable at compile time, that is). But the following program does this without dynamic allocation. Each time around the loop, it creates an object at line 13 and destructs the object at the closing curly brace in line 14.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     cout << "How many variables do you want to create?\n";
 9     int n;
10     cin >> n;
11
12     for (int i = 0; i < n; ++i) {
13         obj ob(i);
14     }
15
16     return EXIT_SUCCESS;
17 }
```

```
How many variables do you want to create?
4
construct 0        first time we arrive at line 13
destruct 0         first time we arive at line 14
construct 1        second time we arrive at line 13
destruct 1         second time we arrive at line 14
construct 2        third time we arrive at line 13
destruct 2         third time we arrive at line 14
construct 3        fourth time we arrive at line 13
destruct 3         fourth time we arive at line 14
```

The variables in the above program exist one at a time. It may be objected that dynamic allocation would still be necessary to create an unpredictable number of variables that exist simultaneously. But the following program can do this with recursion, not dynamic allocation. On the way down, the program constructs an unpredictable number of objects which all exist during the last call to the function. On the way back up, the objects are destructed.

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

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

int f(int n);

int main()
{
    cout << "How many variables do you want to create?\n";
    int n;
    cin >> n;

    f(n);
    return EXIT_SUCCESS;
}

int f(int n)
{
    obj ob(n);

    if (n > 1) {
        f(n - 1);
    }
}
```

```
How many variables do you want to create?
4
construct 4        first time we arrive at line 20
construct 3        second time we arrive at line 20
construct 2        third time we arrive at line 20
construct 1        fourth time we arrive at line 20
destruct 1         first time we arive at line 25
destruct 2         second time we arrive at line 25
destruct 3         third time we arrive at line 25
destruct 4         fourth time we arive at line 25
```

The above two programs did construct an unpredictable number of variables, but each variable was destroyed when we left the block of statements in which it was declared. To destroy a variable at another point, we must resort to the other way of creating it: by *dynamic memory allocation.* Dynamic means "as the program is running". We ask the operating system at runtime for a block of memory to hold the variable, and give the block back to the operating system when we are done with it.

A block of memory is allocated dynamically in C by calling the functions `malloc` and `free`. We tell them how many bytes we need, but not the data type of the variables that will occupy the block. Since these functions do not know what the block will be used for, they cannot initialize it for us. And when we relinquish the block, `free` does nothing except give it back to the operating system.

A block of memory is allocated dynamically in C++ by executing the `new` and `delete` operators. This time, we tell them the data type of the variables that will occupy the block. Since the `new` operator knows what the block will be used for, it can call the constructors for the variables in the block and give us a block full of initialized variables. And when we relinquish the block, `delete` calls the corresponding destructors before giving it back to the operating system.

4.2.2 Allocate a Scalar

Allocate a scalar in C

The following program reviews dynamic memory allocation in C, pointing out its shortcomings. The `struct node` that we allocate and deallocate is like the C++ class `node` in pp. 212−217, but stripped of its member functions and friends. We will allocate a linked list of these nodes in the next program. A `node`, by the way, is an example of a *scalar*—a variable that is not an array.

We call the function `malloc` in line 12 to get a block of memory which can be treated as a variable, in this case as a `struct node`. (Remember that C needs the `struct` keyword in line 12; C++ will not.) The argument of `malloc` tells it the number of bytes we want. If successful, the return value of `malloc` will be the address of the allocated block.

`malloc` was never told the data type of variable that will occupy the block. This means that `malloc` cannot initialize the block with any useful value. Even if it performs flawlessly, the most we can hope for from `malloc` is the address of a block of garbage. Let's hope the program never tries to read this garbage.

Always store the return value of `malloc` into a pointer and *keep it there* until the block is deallocated. To ensure that we do, line 12 declared `p` to be a `*const`: a pointer that always points to the same place. If the pointer were to point elsewhere, we would no longer be able to access the block or deallocate it. This painful situation called a *memory leak.*

Incidentally, the `=` in line 12 performs an implicit conversion. The return value of `malloc` is a pointer to `void`, while `p` is a pointer to a `struct node`. This one special case of pointer conversion, between a `void *` and another type of pointer, is the only one that C will do implicitly. I don't expect anything will go wrong with the conversion. But as we shall see, the corresponding `=` in C++ will avoid the conversion entirely.

If unsuccessful, `malloc` returns `NULL`. The conscientious C programmer will therefore have to write the follow-up `if` in lines 13−17 after every call to `malloc`. We will see that his or her equally conscientious C++ colleague will write the error checking only once. The message in line 14, by the way, is not portable because of the `%u`. `sizeof` yields a `size_t`, which is a typedef for `unsigned` on my machine. Your `size_t` might be a typedef for `long unsigned` (`%lu`).

Since a successful `malloc` delivers a block full of garbage, we need lines 19−20 to assign values to the fields in the block. Permitting these assignments is one reason why the `p` in line 12 must be a read/write pointer. In C++, the assignments will be unnecessary and `p` will be read-only. To verify that the assignments worked, line 22 uses the `%p` format to output each pointer field in the structure.

When we are done with the block, we give it back to the operating system by passing its address to the `free` function in line 29. But even if it performs flawlessly, `free` never calls the destructors for the variables in the block. Let's hope the program remembered to call them.

The only argument we should ever give to `free` is the address of a block obtained from a previous `malloc`, `realloc`, or `calloc`. If we mess up, there is no return value from `free` that we could check for error. The argument of `free` is a `void *`, not a `const void *`, which is another reason why `p` can't be read-only.

The `free` function will do nothing if its argument is `NULL`. This means that if the `malloc` in line 12 does return `NULL`, and if we forget the `if` in lines 13−17, *and* if we somehow get through lines 19−27, the `free` in line 29 will be harmless and not blow up.

If we forget to call `free`, the block will be freed anyway when the program ends in line 30. But be a good citizen of the global community and `free` the block as soon as you're done with it. Other people might be waiting for memory.

To ensure that `p` will never reference the block after it is deallocated, we could try to insert the statement

```
1       p = 0;
```

at line 29½. But `p` is a `*const` and this assignment will not compile. Instead of zeroing it, the C++ approach is to prevent a dangling pointer from outliving the block to which it points. This `p`, for example, is destructed in the very next line, before it can do any mischief. And on pp. 466−467 we will see a pointer that is elegantly destructed by the destructor for the object that occupies the block. The complementary goal, to prevent a block from outliving the pointer that points to it, will be achieved on p. 612 with an `auto_ptr`.

Let's poke around in memory to see how `malloc` records the number of bytes that `free` must free. The hidden machinery is completely unofficial and will be different on each platform. But looking at a typical implementation will show us how the *heap* (the pool of dynamically allocatable memory) could become corrupted in C++ if we deallocate incorrectly. It will also show us why writing our own allocation and deallocation functions may be advantageous in C++.

On my platform, a dynamically allocated block of memory is actually eight bytes longer than the size we ask for. `malloc` takes the number of bytes in the block, rounds it up to a multiple of 8 and adds 1, and stores the result in the first four bytes of the block (a slot of type `size_t`). It stores another number, usually zero, in the next four bytes, and returns the address of the ninth byte. The user is unaware of the eight-byte prefix before the official start of the block.

The following diagram shows what happens when we allocate the `struct node` in line 4. On my machine each field of the structure is four bytes, for a total of 12. To display the two numbers in the prefix, line 27 casts `p` to a pointer to `size_t` and then slaps on a negative subscript. We need parentheses to apply the cast to `p` before the subscript.

The `free` function in line 29 takes the address of the block and backpedals eight bytes to get to the hidden number. This number tells `free` how many bytes to free.

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

```
1 #include <stdio.h>    /* C example */
```

```
 2 #include <stdlib.h>   /* for malloc and free */
 3
 4 struct node {         /* a node on a doubly linked list */
 5     int value;
 6     struct node *prev;
 7     struct node *next;
 8 };
 9
10 int main(int argc, char **argv)
11 {
12     struct node *const p = malloc(sizeof (struct node));
13     if (p == NULL) {
14         fprintf(stderr, "%s: can’t allocate %u bytes\n", /* not portable */
15             argv[0], sizeof (struct node));
16         return EXIT_FAILURE;
17     }
18
19     p->value = 10;
20     p->prev = p->next = NULL;
21
22     printf("value == %d, prev == %p, next == %p.\n",
23         p->value, p->prev, p->next);
24
25     printf("A struct node occupies %u bytes.\n", sizeof (struct node));
26     printf("The hidden numbers are %u and %u.\n", /* unofficial; not portable */
27         ((size_t *)p)[-2], ((size_t *)p)[-1]);
28
29     free(p);
30     return EXIT_SUCCESS;
31 }
```

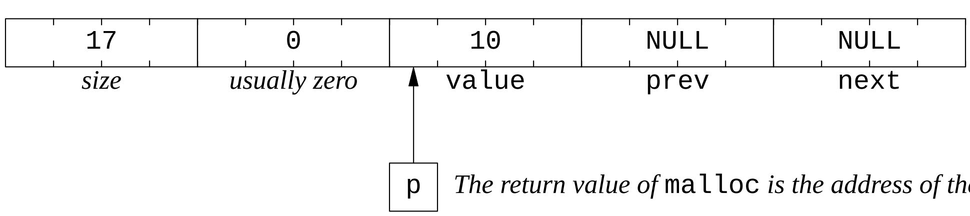
The above lines 12−13 can be rewritten

```
32     struct node *p;
33     if ((p = malloc(sizeof (struct node))) == NULL) {
```

But why would you want to? The pointer `p` could then no longer be `*const`.

```
value == 10, prev == 0, next == 0.
A struct node occupies 12 bytes.
The hidden numbers are 17 and 0.
```

17 = 2 × 8 + 1



The return value of `malloc` *is the address of the ninth byte.*

▼ Homework 4.2.2a: examine the hidden number on your platform

Is the hidden number on your platform at location `((size_t *)p)[-1]` or `((size_t *)p)[-2]` or elsewhere?

▲

Scalars that must be allocated dynamically

We have just created one structure; now we will create unpredictably many. But that by itself is not the reason why we must now allocate them dynamically. The above structure was created in a block of statements (the body of the `main` function) and destroyed in the same block. It could therefore have been created by a declaration that was also a definition. But the following structures are created in one block (the `while` loop in lines 19−58) and destructed in another (the `for` loop in lines 62−68). They must be allocated dynamically.

The program builds a doubly linked list of nodes, sorted in increasing numerical order by their `value`'s. The return value of the `scanf` in line 19 is the number of variables that were successfully read from input. It breaks us out of the `while` loop when we encounter end-of-file or garbage, and we return `EXIT_SUCCESS` or `EXIT_FAILURE` respectively.

I'm sorry that so much of this program, lines 28−48, is just a bunch of special cases. At least in the C++ version, some of the cases will be hidden in the member functions and friends of a class.

I'm also sorry that the `p = p->next` in line 66 can't be in its expected place, after the second semicolon in line 63. But the `p = p->next` must come *before* the `free(doomed)` in line 67, since the `free` might then wipe out the value of the pointer field `p->next`. This is an early example of the "increment of death" on pp. 444−445.

`doomed` must be a read/write pointer because the argument of `free` is a `void *`, not a `const void *`. In C++, `doomed` can be read-only.

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

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 struct node {                    /* a node on a doubly linked list */
 5     int value;
 6     struct node *prev;
 7     struct node *next;
 8 };
 9
10 int main(int argc, char **argv)
11 {
12     struct node *first = NULL;   /* List initially empty. */
13     struct node *last = NULL;
14     int i;
15     struct node *p;
16
17     printf("Type a series of integers, EOF (control-d) to quit.\n");
18
19     while (scanf("%d", &i) == 1) {
20         struct node *const n = malloc(sizeof (struct node));
21         if (n == NULL) {
22             fprintf(stderr, "%s: can't allocate %u bytes\n", /* not portable */
23                 argv[0], sizeof (struct node));
24             return EXIT_FAILURE;
25         }
26         n->value = i;
27
28         if (first == NULL) {
29             /* Insert n into an empty list. */
30             n->next = n->prev = NULL;
31             first = last = n;
```

```
32              continue;
33          }
34
35          if (i <= first->value) {
36              /* Insert n before the first node on the list. */
37              n->prev = NULL;
38              n->next = first;
39              first = first->prev = n;
40              continue;
41          }
42
43          if (i > last->value) {
44              /* Insert n after the last node on the list. */
45              n->prev = last;
46              n->next = NULL;
47              last = last->next = n;
48              continue;
49          }
50
51          for (p = first; i > p->value; p = p->next) {
52          }
53
54          /* Insert n between p->prev and p. */
55          n->prev = p->prev;
56          n->next = p;
57          p->prev = p->prev->next = n;
58      }
59
60      printf("\nHere are the integers in increasing order:\n");
61
62      for (p = first; p;) {
63          struct node *const doomed = p;
64          printf("%d\n", p->value);
65
66          p = p->next;
67          free(doomed);
68      }
69
70      return feof(stdin) && !ferror(stdin) ? EXIT_SUCCESS : EXIT_FAILURE;
71  }
```

The above lines 30−31 may be combined to

```
72      (first = last = n)->next = n->prev = NULL;
```

But the original is clearer.

Line 39 must not be rewritten as follows

```
73      first->prev = first = n;
```

We would be unable to predict whether the left `first` was evaluated before or after the right `first` was assigned to. Ditto for lines 47 and 57. See pp. 14−16.

```
Type a series of integers, EOF (control-d) to quit.
30          Insert first node into empty list (lines 28−33).
10          Insert before first node (lines 35−41).
40          Insert after last node (lines 43−49).
20          Insert between two existing nodes (lines 51−58).
control-d

Here are the integers in increasing order:
10
20
30
40
```

Allocate a scalar in C++

C allocates memory by calling two functions, `malloc` and `free`. C++ allocates memory by executing two operators, `new` and `delete`, in lines 8 and 19 of the following program.

Operators may be unary or binary, prefix or postfix. `new` and `delete` are unary prefix operators. Here are a few familiar examples. Most of them are written as punctuation marks, but at least one of them (`sizeof`) is written as a keyword. Most of them require an expression as their operand, but at least one of them (again, `sizeof`) can take the name of a data type.

```
1       -a
2       &a
3      ++a
4 sizeof a
5 sizeof (int)
```

Despite the parentheses, `sizeof` is not a function. Its `(int)` is an operand, not an argument list. Recall that a function is something that has a {body} somewhere. `sizeof` does not have a body.

`new` is a unary prefix operator like `sizeof`. It is not a function. Do not confuse it with the function `operator new` that we will see on p. 410.

An operand of `new` is always the name of a data type. It is similar to a data type operand of `sizeof`. The latter always has surrounding parentheses, but an operand of `new` almost never needs them. See pp. 407−410.

In the following program, the name of the data type is the single word `node` in line 8. (The keyword `struct` is not needed here in C++, as it was in line 12 of the above `scalar.c`.) In preparation for the more complex examples that follow, we show how the name is composed. Start with a declaration for a fictitious variable of the desired type. Then erase the name of the fictitious variable and the semicolon. What remains will be the name of the data type, which we can give to the `new` operator. For example, to allocate an `node`,

```
6                           node n;   //declaration for fictitious variable
7                           node      //name of data type
8 const node *const p = new node;     //dynamically allocate memory in line 8
```

Like `malloc`, `new` gives us the address of a block of memory. But unlike `malloc`, `new` knows what the block will be used for. The operand `node` in line 8 tells `new` the data type of the variable that will occupy the block, so `new` can initialize the variable by calling its constructor. We pass the argument `10` to the constructor; there could be more than one argument in the parentheses for constructors that accept them. The constructor initializes the data members `value`, `prev`, and `next`, so there is no need for the assignment statements in lines 19−20 of the above `scalar.c`. This means that the pointer `p` in our line 8 can now be read-only.

The next example will check if the `new` operator is successful in allocating the block of memory for us. For the time being, we are merely hoping it will be. If it isn't, no attempt will be made to construct the object in the block, because there is no block. `new` will abort the program by calling the `abort` function from the C Standard Library. A more detailed description of how this happens is in p. 590 and pp. 625−628; it involves ‘‘throwing an uncaught exception’’ of data type `bad_alloc`.

In C the return type of `malloc` is always `void *`, so the `=` in line 12 of the above C program `scalar.c` performed an implicit conversion. In C++, the value of `new` is a pointer to whatever data type has been allocated. The `new` in line 8, for example, returns a pointer to a `node`, so our `=` performs no conversion.

When we’re done with the block we give its address to another unary prefix operator, the `delete` in line 19, with a more traditional kind of operand. The operand of `new` is the name of a data type; the operand of `delete` is the address of a block to be deleted. It must be the address of a block that we got from a previous `new`, just as the argument of the C function `free` had to be the address of a block that we got from a previous `malloc` `realloc`, or `calloc`.

Like `free`, `delete` will do nothing if its operand is a zero pointer. But if the operand is non-zero, `delete` will do more than `free` does. The data type of the pointer operand tells `delete` what type of variable occupied the block; `delete` will call the destructor for that type of variable and then return the block to the operating system.

If we forget to write the `delete`, the block will be freed anyway when the program ends in line 20. But be a good citizen of the global community and `delete` the block as soon as you’re done with it: other people may be waiting for memory.

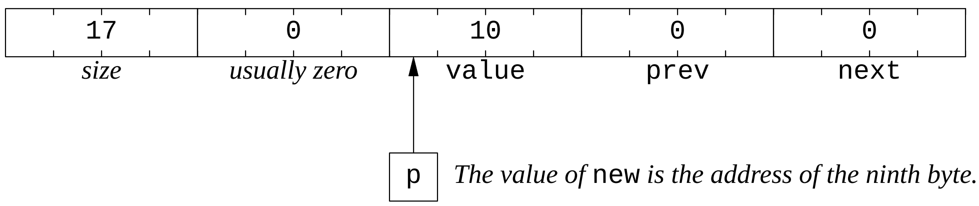
As in C, the number of bytes in a dynamically allocated block is (unofficially) stored in a hidden number at the start of the block, telling `delete` how many bytes to delete. Once again, we cast `p` to a pointer to a `size_t` in lines 16−17 before slapping on the subscripts. In C++, the casts must be `reinterpret_cast`’s to make the conversion between different pointer types more conspicuous. A `static_cast` would not compile here.

Class `node` was in pp. 212−217.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "node.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     const node *const p = new node(10);
 9
10     cout << "value == " << *p << ", prev == " << p->prev
11         << ", next == " << p->next << ".\n"
12
13         << "A node occupies " << sizeof (node) << " bytes.\n"
14
15         << "The hidden numbers are "   //unofficial
16         << reinterpret_cast<const size_t *>(p)[-2] << " and "
17         << reinterpret_cast<const size_t *>(p)[-1] << ".\n";
18
19     delete p;
20     return EXIT_SUCCESS;
21 }
```

```
value == 10, prev == 0, next == 0.
A node occupies 12 bytes.
The hidden numbers are 17 and 0.                         17 = 2 × 8 + 1
```



▼ Homework 4.2.2b: allocate an obj

Allocate and deallocate an `obj` (pp. 179−180) instead of a `node`. The output will prove that `new` calls a constructor and `delete` calls the destructor. If you remove the `delete`, will the `obj` still be destructed?

▲

▼ Homework 4.2.2c: allocate a variable of a built-in data type

Verify that you can allocate and initialize an `int` just like an object, even though an `int` has no constructor:

```
const obj *const p1 = new obj(10);
cout << "The obj is " << *p1 << ".\n";
delete p1;

const int *const p2 = new int(10);
cout << "The int is " << *p2 << ".\n";
delete p2;
```

▲

▼ Homework 4.2.2d: call the default constructor

Verify that the default constructor is called when you omit the arguments. You don't even need the empty parentheses:

```
 1     const obj *p1 = new obj(10);                   //call one-arg constructor
 2     cout << "The object is " << *p1 << ".\n";
 3     delete p1;
 4
 5     p1 = new obj();                                //call default constructor
 6     cout << "The object is " << *p1 << ".\n";
 7     delete p1;
 8
 9     p1 = new obj;                                  //call default constructor
10     cout << "The object is " << *p1 << ".\n";
11     delete p1;
```

When allocating a variable of a built-in type, the variable behaves as if it has a default constructor that puts zero into the newborn variable. But to call this constructor, you must write the empty parentheses in line 16. Without them, no attempt is made to initialize the variable (line 20).

```
12     const int *p2 = new int(10);                   //Put 10 into the int.
13     cout << "The int is " << *p2 << ".\n";
```

```
    delete p2;

    p2 = new int();                              //Put zero into the int.
    cout << "The int is " << *p2 << ".\n";
    delete p2;

    p2 = new int;                                //Put garbage into the int.
    cout << "The int is " << *p2 << ".\n";
    delete p2;
```

When allocating and default-constructing a variable whose type is unknown, we must therefore write the empty parentheses.

```
    //Suppose this typedef was off in another file where we can’t see it.
    typedef int T;

    const T *p3 = new T();
```

The type will certainly be unknown when we have ‘‘templates’’. See p. 660.

▲

Check for allocation failure in C++

Let’s check for allocation failure instead of allowing the program to abort itself. To check for error in C, we had to follow every `malloc` with an `if`. In C++, we can make the `new` operator check itself.

First, in lines 30−34, write a separate function containing the code to be executed upon allocation failure. The function can have any name, but it must have no arguments and no return value. For the time being, it must end with an `exit`.

In C and C++, the name of a function all by itself, with no parenthesized argument list after it, stands for the address of the function. For example, the name `my_new_handler` in line 13 is the address of that function. To tell the computer that this is the function to be called upon allocation failure, we pass its address to another function, the C++ Standard Library function `set_new_handler`. The header file `<new>` in line 3 is where `set_new_handler` is declared.

Our `my_new_handler` function must be declared (line 7) before its name can be otherwise mentioned (line 13). And we must pass the address of `my_new_handler` to `set_new_handler` (line 13) before our first attempt at allocation (line 15).

If the `new` operator cannot allocate a block of memory, it will now call `my_new_handler`. No attempt will be made to construct the object in the block, because there is no block.

There are other ways of checking for allocation failure; we will talk about them in pp. 625−628 after we cover ‘‘exceptions’’. Until then, we will always call `set_new_handler` before any attempt at dynamic allocation.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>           //for set_new_handler
 4 #include "node.h"
 5 using namespace std;
 6
 7 void my_new_handler();   //function declaration
 8 const char *progname;    //uninitialized variable
 9
10 int main(int argc, char **argv)
11 {
```

```
    progname = argv[0];
    set_new_handler(my_new_handler);

    const node *const p = new node(10);

    cout << "value == " << *p << ", prev == " << p->prev
        << ", next == " << p->next << ".\n"

        << "A node occupies " << sizeof (node) << " bytes.\n"

        << "The hidden numbers are "   //unofficial
        << reinterpret_cast<const size_t *>(p)[-2] << " and "
        << reinterpret_cast<const size_t *>(p)[-1] << ".\n";

    delete p;
    return EXIT_SUCCESS;
}

void my_new_handler()   //function definition
{
    cerr << progname << ": out of store\n";
    exit(EXIT_FAILURE);
}
```

```
value == 10, prev == 0, next == 0.
A node occupies 12 bytes.
The hidden numbers are 17 and 0.
```

17 = 2 × 8 + 1

In archaic versions of Microsoft Visual C++, your `my_new_handler` function must have one argument of data type `size_t` and a return type of `int __cdecl`. The return type is two separate words, the second one starting with two underscores and ending with lowercase L. The function `_set_new_handler` starts with an underscore, and the header file is `<new.h>`.

Scalars that must be allocated dynamically

Here is the linked list example in C++. Once again, the allocation must now be dynamic because the objects are constructed in one block (the `while` loop in lines 21−47) and destructed in another (the `for` loop in lines 51−57).

Note that the operand of `delete` can be a read-only pointer, unlike the argument of the C function `free`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>
 4 #include "node.h"
 5 using namespace std;
 6
 7 void my_new_handler();
 8 const char *progname;    //uninitialized variable
 9
10 int main(int argc, char **argv)
11 {
12     progname = argv[0];
```

```
    set_new_handler(my_new_handler);

    node *first = 0;   //List initially empty.
    node *last = 0;

    cout << "Type a series of integers, EOF (control-d) to quit.\n";

    value_type i;
    while (cin >> i) {   //while (cin.operator>>(i).operator void *()) {
        node *const n = new node(i);

        if (first == 0) {
            //Insert n into an empty list.
            first = last = n;
            continue;
        }

        if (i <= *first) {  //if (i <= (*first).operator value_type()) {
            n->insert_this_before(first);
            first = n;
            continue;
        }

        if (i > *last) {
            n->insert_this_after(last);
            last = n;
            continue;
        }

        node *p = first;
        for (; i > *p; p = p->next) {  //i > (*p).operator value_type();
        }

        n->insert_this_before(p);
    }

    cout << "\nHere are the integers in increasing order:\n";

    for (const node *p = first; p;) {
        cout << *p << "\n";   //cout << (*p).operator value_type() << "\n";

        const node *const doomed = p;
        p = p->next;
        delete doomed;
    }

    return cin.rdstate() == (ios_base::eofbit | ios_base::failbit)
        ? EXIT_SUCCESS : EXIT_FAILURE;

}

void my_new_handler()
{
    cerr << progname << ": out of store\n";
```

```
67     exit(EXIT_FAILURE);
68 }
```

```
Type a series of integers, EOF (control-d) to quit.
30            Insert first node into empty list (lines 17−21).
10            Insert before first node (lines 23−27).
40            Insert after last node (lines 29−33).
20            Insert between two existing nodes (lines 35−39).
control-d

Here are the integers in increasing order:
10
20
30
40
```

4.2.3 Allocate an Array

Allocate an array in C

The allocated variable in the following programs is an array, not a scalar. We made it an array of a built-in data type, `char`, to avoid the complication of calling constructors and destructors. The number of elements of the array is not known until runtime, so it must be allocated dynamically.

In C and C++, a variable that holds the number of elements in an array, or the number of bytes in a block of memory, should always be of data type `size_t` (line 6). On my machine, `size_t` is another name for `unsigned`, so the `scanf` in line 10 has the `%u` format. On other machines `size_t` might be `long unsigned` so the format would have to be `%lu`. This portability problem will go away in C++.

The `malloc` in line 12 returns the address of a block of memory which can then be treated as an array, in this case of `char`'s. The multiplication by `sizeof (char)` is unnecessary since `sizeof (char)` is always 1. But I wanted to remind you that the argument of `malloc` is the number of bytes we need, not the number of array elements. `malloc` is never told that the block will hold an array, let alone the number of elements.

To avoid memory leaks, we always want to store the address of an allocated block into a `*const` pointer: one that always points to the same place. I wish that line 7 could have defined `p` this way.

```
    char *const p;
```

But we can't do this: a definition for a constant will not compile without an initial value. So my second wish is to move the definition down to line 12, where we have an initial value to put into it.

```
    char *const p = malloc(n * sizeof(char));
```

But we can't do this either: C demands that local variables be declared immediately after the opening curly brace of the enclosing block of statements, in line 5. Forced to dangle up at line 7, `p` will have to remain a non-`*const` in C. In C++, we will do better.

Again, the `=` in line 12 performs an implicit conversion. The return value of `malloc` is a pointer to `void`, while `p` is a pointer to `char`. The corresponding `=` in line 11 of the C++ program `no_destructor.C` will avoid the conversion entirely.

As with all arrays, the subscripts start at zero. Since there are `n` elements, the highest subscript is `n - 1` (line 22). Don't go beyond it.

The following diagram shows what happens when we allocate an array of 12 `char`'s. The first `size_t` in the eight-byte prefix is one more than a multiple of eight; the second `size_t` is zero.

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

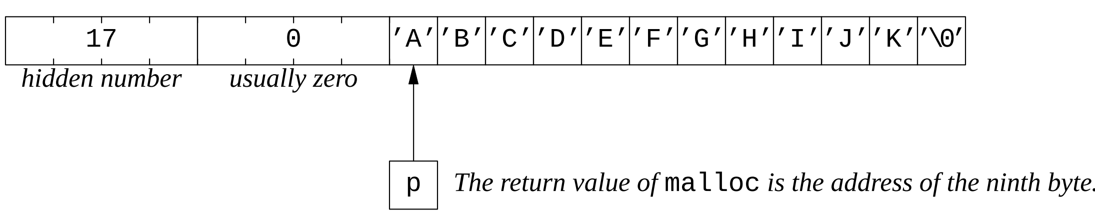
```
 1 #include <stdio.h>   /* C example */
 2 #include <stdlib.h>
 3
 4 int main(int argc, char **argv)
 5 {
 6     size_t n;
 7     char *p;
 8
 9     printf("How many char's do you want to allocate? ");
10     scanf("%u", &n);   /* not portable */
11
12     p = malloc(n * sizeof (char));
13     if (p == NULL) {
14         fprintf(stderr, "%s: can't allocate %u char's\n", argv[0], n);
15         return EXIT_FAILURE;
16     }
17
18     p[0] = 'A';        /* or *p = 'A'; */
19     p[1] = 'B';
20     p[2] = 'C';
21     /* etc. */
22     p[n - 1] = '\0';   /* Warning: the subscripts only go up to n - 1. */
23
24     printf("The hidden numbers are %u and %u.\n", /* unofficial; not portable */
25         ((size_t *)p)[-2], ((size_t *)p)[-1]);
26
27     free(p);
28     return EXIT_SUCCESS;
29 }
```

The above lines 12−13 can be combined to

```
30     if ((p = malloc(n * sizeof(int))) == NULL) {
```

But don't do it. We would just have to uncombine them in C++.

```
How many char's do you want to allocate? 12
The hidden numbers are 17 and 0.                 17 = 2 × 8 + 1
```



If we ask for too much, `malloc` returns `NULL`. Let's ask for the biggest number that would fit into the argument of `malloc`, which is a `size_t`. On my platform, this number is $2^{32} - 1 = 4,294,967,295$.

```
How many char's do you want to allocate? 4294967295
no_destructor: can't allocate 4294967295 char's
```

▼ Homework 4.2.3a: examine the hidden number on your platform

Run the above C program several times on your platform, asking for different amounts of memory. Is the hidden number at location `((size_t *)p)[-1]` or `((size_t *)p)[-2]` or elsewhere? Do you see any pattern in the values of the hidden number? Is it the number of bytes we asked for, rounded up to a multiple of 8 and incremented?

▲

Allocate an array of variables with no destructors in C++

To compose the name of the data type "array of `n` `char`'s" given to the `new` in line 18, we once again begin by writing a declaration for a fictitious variable of this type. In this declaration, the first (or only) dimension of an array can be a variable, although all subsequent dimensions (if any) must be constants. Then erase the name of the fictitious variable and the semicolon. What remains will be the name of the data type, which we can give to the `new` operator.

```
                    char a[n];   //declaration for fictitious variable
                    char  [n]    //name of data type
char *const p = new char  [n];   //dynamically allocate memory in line 18
```

When we're done with memory that held a scalar, we give it back to the unary prefix operator `delete`. When we're done with memory that held an array, we give it back to a different unary prefix operator, the `delete[]` in line 30. It is up to the programmer to write the correct form of `delete`; the next example will show why the heap will be corrupted if the programmer does it wrong.

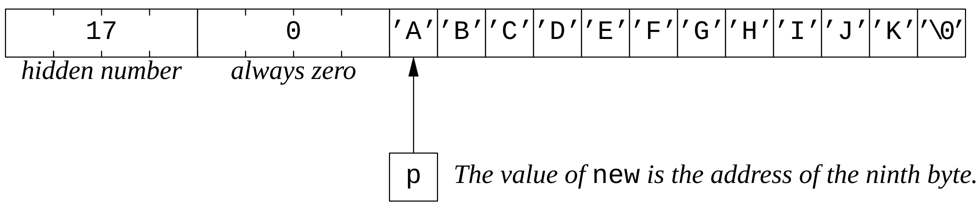
—On the Web at
`http://i5.nyu.edu/~mm64/book/src/new/no_destructor.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>
 4 using namespace std;
 5
 6 void my_new_handler();
 7 const char *progname;    //uninitialized variable
 8
 9 int main(int argc, char **argv)
10 {
11     progname = argv[0];
12     set_new_handler(my_new_handler);
13
14     cout << "How many char's do you want to allocate? ";
15     size_t n;
16     cin >> n;   //portable
17
18     char *const p = new char [n];
19
20     p[0] = 'A';
21     p[1] = 'B';
22     p[2] = 'C';
23     //etc.
24     p[n - 1] = '\0';   //Warning: subscripts only go up to n - 1.
25
26         cout << "The hidden numbers are "                         //unofficial
27                 << reinterpret_cast<size_t *>(p)[-2] << " and "
28                 << reinterpret_cast<size_t *>(p)[-1] << ".\n";
29
30     delete[] p;
```

```
31     return EXIT_SUCCESS;
32 }
33
34 void my_new_handler()
35 {
36     cerr << progname << ": out of store\n";
37     exit(EXIT_FAILURE);
38 }
```

```
How many char's do you want to allocate? 12
The hidden numbers are 17 and 0.
```

17 = 2 × 8 + 1



If we ask for too much, `new` calls `my_new_handler`.

```
How many char's do you want to allocate? 4294967295
no_destructor: out of store
```

▼ **Homework 4.2.3b: how many bytes can you allocate?**

How many bytes is the biggest block of memory you can allocate in C++? Is it always the same number?
▲

▼ **Homework 4.2.3c: a new_handler that doesn't end with exit**

If our `my_new_handler` didn't end with `exit`, it would return to the `new` operator that failed. The `new` would then try to allocate memory again. What would happen if we remove the `exit` from our `my_new_handler` and ask for more memory than is available?
▲

Allocate an array of objects with destructors

The `new` in line 19 allocates and constructs an array of objects. The operand of the `new` tells it the data type of each element and the number of elements we want. The `new` attempts to allocate memory, and, if successful, calls the constructor for each object in the array in order of ascending subscript.

Similarly, the `delete[]` in line 30 destructs and deallocates the array. It will call the destructor for each object in order of descending subscript, and then deallocates the memory occupied by the array. The data type of the operand of the `delete[]` tells it the data type of each element; the value of the operand tells it the address of the first element. But how does `delete[]` know how many elements there are?

On my platform, when `new` allocates an array of objects with destructors, it stores the number of elements in the array at subscript `[-1]` in the hidden prefix. `delete[]` calls this number of destructors; it is printed at line 28.

As usual, `new` also stores the total number of bytes. But when allocating an array of objects with destructors, `new` stores the total at subscript `[-3]`, not `[-2]`. It is printed at line 26.

Now we can see why we must choose the correct `delete` operator. The `delete` without the square brackets always expects to find the number of bytes to deallocate at subscript `[-2]`. The `delete[]` with square brackets expects to find the number of bytes at subscript `[-3]` if the array elements have destructors, or at `[-2]` if they do not. Of course, the layout of the hidden machinery will be different on each platform. But on any platform, choosing the wrong `delete`, or passing it a pointer to the wrong data type, will result in calling the wrong number of destructors and deallocating the wrong number of bytes.

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

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

void my_new_handler();
const char *progname;   //uninitialized variable

int main(int argc, char **argv)
{
    progname = argv[0];
    set_new_handler(my_new_handler);

    cout << "How many obj's do you want to allocate? ";
    size_t n;
    cin >> n;

    obj *const p = new obj [n];   //Call the default constructor n times.

    for (size_t i = 0; i < n; ++i) {
        cout << p[i] << "\n";
    }

    cout << "The hidden numbers are "                        //unofficial
        << reinterpret_cast<size_t *>(p)[-3] << ", "
        << reinterpret_cast<size_t *>(p)[-2] << ", and "
        << reinterpret_cast<size_t *>(p)[-1] << ".\n";

    delete[] p;
    return EXIT_SUCCESS;
}

void my_new_handler()
{
    cerr << progname << ": out of store\n";
    exit(EXIT_FAILURE);
}
```

```
How many obj's do you want to allocate? 3
default construct 0
default construct 0
default construct 0
0
0
0
The hidden numbers are 17, 0, and 3.            17 = 2 × 8 + 1
destruct 0
destruct 0
destruct 0
```

```
How many obj's do you want to allocate? 1
default construct 0
0
The hidden numbers are 9, 0, and 1.             9 = 1 × 8 + 1
destruct 0
```

```
How many obj's do you want to allocate? 0
The hidden numbers are 9, 0, and 0.             9 = 1 × 8 + 1
```

▼ Homework 4.2.3d: use the wrong delete

What happens, and what fails to happen, if we allocate and construct one `obj` and attempt to destruct and deallocate it with the `delete` with `[`square brackets`]`? Does the object get destructed?

What happens, and what fails to happen, if we allocate and construct an array of `obj`'s and attempt to destruct and deallocate them with the `delete` without `[`square brackets`]`? How many of the objects get destructed?

Is there an error message? In both cases, confine yourself to observing the destructors that are called or not called. There is no easy way to observe the damage to the heap.

▲

Allocate and initialize an array of objects with a constructor with arguments

We have passed arguments to the constructor for a dynamically allocated scalar; see line 15 of the above `set_new_handler.C`. But C++ does not allow us to pass arguments to the constructors for the elements in a dynamically allocated array. It forces us to call the default constructor for each element, as in line 19 of the above `destructor.C`.

We can use a surprising workaround to prevent the `new` from calling the default constructor for each element. For symmetry, we will use the same workaround to prevent the `delete[]` from calling the destructor for each element. In between, we will manually call a constructor with arguments for each element, and manually call the destructor for each element.

Under normal circumstances, `new` allocates memory and calls a constructor. But the `new` in the following line 19 will allocate memory without calling a constructor; the `new` in line 26 will call a constructor without allocating memory.

Line 19 deliberately misinforms `new` that what we are allocating is an array of `char`'s, not an array of objects. Since a `char` has no constructor, line 19 calls no constructor. But the array of `char`'s occupies exactly the same number of bytes as an array of `n` `date`'s. (The number of bytes must be written as `n * sizeof (date)`, not `sizeof (date[n])`, because a data type operand of `sizeof` cannot contain a variable-sized array.)

The value of the `new` in line 19 is a pointer to a `char`, but `p` is declared to be a pointer to a `date`. A `reinterpret_cast` must be used when converting between pointers to different non-`void` types. Note that `p` is initialized to the address of a chunk of memory that is not yet occupied by a `date` object.

The chunk will be converted to a `date` in line 26.

The cast in line 43 deliberately misinforms `delete[]` that what it is deallocating is an array of `char`'s. Since a `char` has no constructor, line 43 calls no constructor.

Since the constructors and destructors for the `date`'s were not called by lines 19 and 43, it is up to us to call them. Usually the `new` operator performs memory allocation followed by construction. But the `new` in line 19 performs allocation without construction; the one in line 26 performs construction without allocation.

The `new` in line 19 is the one that we have been using all along. It allocates a block of memory, and, if successful, it calls the constructor for each object in the block. At least it would call them, if the variables in this block had constructors. But our variables are merely `char`'s.

The `new` in line 26 is different. It allocates no memory. It merely constructs a `date` object at address `q`. The constructor receives the three explicit arguments, as well as the implicit argument `q`. This use of `new` is called the *placement syntax;* it makes an object out of the the raw memory to which `q` points. The implicit pointer `q` passed to the constructor must be read/write. As usual, the value of the `new` operator is the address of the newly constructed object. This value is ignored in line 26.

The placement `new` in line 26 allocated no memory, so there is no need for a corresponding `delete`. But it did call a constructor, so we have to call the corresponding destructor (if there is one). We never wrote a destructor for class `date`, but we can demonstrate the call to the destructor anyway. This is because the computer behaves as if class `date` had a destructor that does nothing; see p. 310.

For an object constructed with the placement `new`, the destructor must be called explicitly. In no other case in C++ is a destructor called explicitly. See the syntax in line 40, and pp. 662−663 for another example. Of course, line 40 is needed only for a class whose destructor actually does something.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>
 4 #include "date.h"
 5 using namespace std;
 6
 7 void my_new_handler();
 8 const char *progname;
 9
10 int main(int argc, char **argv)
11 {
12     progname = argv[0];
13     set_new_handler(my_new_handler);
14
15     cout << "How many date’s do you want to allocate? ";
16     size_t n;
17     cin >> n;
18
19     date *const p = reinterpret_cast<date *>(new char[n * sizeof (date)]);
20
21     //Call the constructor for each date in the array.
22     for (date *q = p; q < p + n; ++q) {
23         cout << "Month, day, year for date " << q - p << ": ";
24         int month, day, year;            //uninitialized variables
25         cin >> month >> day >> year;
26         new(q) date(month, day, year);   //the placement syntax
27     }
28
```

```
    for (const date *q = p; q < p + n; ++q) {
        cout << *q << "\n";
    }

    cout << "The hidden numbers are "                          //unofficial
        << reinterpret_cast<size_t *>(p)[-2] << " and "
        << reinterpret_cast<size_t *>(p)[-1] << ".\n";

    //Call the destructor for each date in the array.
    //(Required if class date has a destructor; does nothing otherwise.)
    for (const date *q = p + n - 1; q >= p; --q) {
        q->~date();
    }

    delete[] reinterpret_cast<char *>(p);
    return EXIT_SUCCESS;
}

void my_new_handler()
{
    cerr << progname << ": out of store\n";
    exit(EXIT_SUCCESS);
}
```

```
How many date’s do you want to allocate? 5
Month, day, year for date 0: 7 4 1776
Month, day, year for date 1: 10 29 1929
Month, day, year for date 2: 12 7 1941
Month, day, year for date 3: 7 20 1969
Month, day, year for date 4: 9 11 2001
7/4/1776
10/29/1929
12/7/1941
7/20/1969
9/11/2001
The hidden numbers are 65 and 0.                    65 = 8 × 8 + 1
```

Parentheses around the operand of new

Parentheses are always needed around a data type argument of `sizeof`.

```
sizeof (int)
```

On two rare occasions, parentheses are also needed around the operand of `new`.

(1) If the name of the data type contains `(`parentheses`)` not enclosed in a pair of `[`square brackets`]`, we must surround the entire name with another pair of parentheses before we give it to `new`. Don’t worry—this is not likely to happen. In fact, it took some effort to come up with the following example.

The simplest data type whose name contains parentheses is the data type of a function:

```
void f();                   //declaration for a function
void  ()                    //the name of the data type of this function
```

But `new` is used only to allocate memory for variables, not for functions.

The next simplest data types containing parentheses are “pointer to function” and “pointer to array”:

```
void (*p)();                //declaration for a pointer to a function
```

```
 4 void (* )()             //the name of the data type of this pointer
 5
 6 int (*p)[10];           //declaration for a pointer to an array
 7 int (* )[10]            //the name of the data type of this pointer
```

But `malloc` and `new` are never used to allocate memory for one pointer. It would gain us nothing, since another pointer, of equal size, would be needed to hold the address of the allocated pointer.

We therefore allocate an array of pointers to functions. As usual, we start with a declaration for a fictitious variable, this time an array of pointers to functions:

```
 8                           void (*a[n])();  //declaration for fictitious variable
 9                           void (* [n])()   //name of data type
10 void (**const p)() = new (void (* [n])()); //dynamically allocate memory
```

The above line 10 (and the following line 19) has a double asterisk because a `new` that allocates an array yields a pointer to the first element of the array. The elements of this array are pointers, so the value of this `new` is a pointer to a pointer. See the double asterisk in line 19 of `language.C` in p. 53.

The `const` in line 19 will keep the pointer `p` pointing to the same place. But a `const` immediately after the leftmost asterisk would make `p` a read-only pointer. It would prevent the assignment in line 20 from compiling.

Lines 22 and 23 are two ways to call the function that `p[0]` points to. If we do write the dereferencing operator `*` (line 22), we need the surrounding parentheses to execute it before the function call operator `()`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>
 4 using namespace std;
 5
 6 void my_new_handler();
 7 const char *progname;    //uninitialized variable
 8 inline void f() {cout << "f()\n";}
 9
10 int main(int argc, char **argv)
11 {
12     progname = argv[0];
13     set_new_handler(my_new_handler);
14
15     cout << "How many pointers to functions do you want to allocate? ";
16     size_t n;
17     cin >> n;
18
19     void (**const p)() = new (void (*[n])());
20     p[0] = f;
21
22     (*p[0])();   //call f
23     p[0]();      //call f
24
25     delete[] p;
26     return EXIT_SUCCESS;
27 }
28
29 void my_new_handler()
```

```
{
    cerr << progname << ": out of store\n";
    exit(EXIT_FAILURE);
}
```

(2) Even more unlikely, the name of the data type must also be surrounded by parentheses if it is followed immediately by tokens that could be part of a longer data type: `*`, `&`, or `[]`. `new` is greedy. It tries to appropriate to itself the longest series of tokens that could possibly be the name of a data type.

In line 14, the name of the data type is the `int` to the right of the word `new`. The value of the expression `new (int)` is a pointer; we cast it to `int` so that it can be bitwise and'ed with another `int`. I wrote the C-style cast `(int)` in front of it because a C++ cast would have required parentheses around the expression `new (int)`, which then would no longer be followed immediately by the `&`:

```
    if (reinterpret_cast<int>(new (int)) & 1) {
```

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>
 4 using namespace std;
 5 
 6 void my_new_handler();
 7 const char *progname;    //uninitialized variable
 8 
 9 int main(int argc, char **argv)
10 {
11     progname = argv[0];
12     set_new_handler(my_new_handler);
13 
14     if ((int)new (int) & 1) {
15         cout << "The allocated block is at an odd address.\n";
16     } else {
17         cout << "The allocated block is at an even address.\n";
18     }
19 
20     //memory leak: never deallocated
21     return EXIT_SUCCESS;
22 }
23 
24 void my_new_handler()   //function definition
25 {
26     cerr << progname << ": out of store\n";
27     exit(EXIT_FAILURE);
28 }
```

```
The allocated block is at an even address.
```

Without the parentheses in the expression `new (int)`, the above line 14 would try to allocate memory for an ‘‘`int &`’’, a reference to an `int`. But we can't do this. A reference has no memory address, so it would not compile. On my platform, the error message would be

```
odd.C: In function ’int main(int, char**)’:
odd.C:14:19: error: new cannot be applied to a reference type
odd.C:14:21: error: expected ’)’ before numeric constant
```

4.2.4 Allocation Functions: `operator new` and `operator delete`

Allocation and deallocation functions for scalars

The `new` operator allocates and constructs a scalar; the `delete` operator destructs and deallocates the scalar. This pair of operators calls a pair of functions to perform the allocation and deallocation. There is never any need to call these functions directly, but here are their declarations anyway. The declarations are not actually written in the source code anywhere, not even in a header file. Like the declaration for the `main` function, they are built into the language.

```
void *operator new(size_t n);
void operator delete(void *p);
```

Do not confuse the operators with the functions. An operator is something that takes operands; a function is something that takes arguments and has a `{`body`}`. We will refer to the operators as ‘‘the `new` operator’’ and ‘‘the `delete` operator’’, with the English word ‘‘operator’’ second and in Roman type. We will refer to the functions as the function ‘‘`operator new`’’ and the function ‘‘`operator delete`’’, with the keyword ‘‘operator’’ first and in `computer type`.

The `new` and `delete` operators do much more than just call the functions `operator new` and `operator delete`. When we say

```
    obj *const p = new obj(10);  //apply the new operator to an operand
```

and

```
    delete p;                    //apply the delete operator to an operand
```

the computer behaves as if we had said

```
    //call the allocation function
    obj *const p = static_cast<obj *>(operator new(sizeof (obj)));
    if (p == 0) {
        call the function whose address was passed to set_new_handler;
    } else {
        new(p) obj(10);          //call the constructor
    }
```

and

```
    if (p != 0) {
        p->~obj();               //call the destructor
        operator delete(p);      //call the deallocation function
    }
```

When we have ‘‘exceptions’’, we will see that the `new` and `delete` operators do even more. See p. 626.

Let’s consider the `new` operator that allocates and initializes a scalar. The operator determines the number of bytes the scalar will occupy, and passes this number as an argument of type `size_t` to the function `operator new`. This function assumes that the block will be occupied by a scalar, but is otherwise similar in its ignorance to the C function `malloc`. It tries to allocate a block of memory of the requested size, and if successful, returns the address of the block as a pointer to `void`. The `new` operator converts this to a pointer to the data type of the scalar, and calls the constructor, if there is one, for the scalar-to-be. The value of the `new` operator is the converted pointer, which is the address of the newborn scalar.

If the function `operator new` cannot allocate the memory we requested, it calls the function whose address was passed to `set_new_handler`. If there was no such function, the function `operator new` “throws a exception” of type `bad_alloc`, triggering a series of events which may end in a call to the `abort` function (p. 590).

The `delete` operator calls the scalar’s destructor, if there is one. It then calls the function `operator delete`, passing it the address of the block. The function `operator delete` backpedals, at least on my platform, to discover the number of bytes to deallocate. It is similar in its ignorance to the C function `free`; It knows it is deallocating a block that held a scalar, but it does not know the data type of the scalar.

Let’s summarize the responsibilities of the operators and functions.

(1) The `new` and `delete` operators know the data type of the variable in the block. The functions `operator new` and `operator delete` know that they are allocating and deallocating a scalar, but they do not know its data type.

(2) The constructor and destructor for the variable in the block are called by the `new` and `delete` operators.

(3) The function designated by `set_new_handler` is called by the function `operator new` if the memory cannot be allocated. The function `operator new` might also “throw an exception”, pp. 625−628.

The functions `operator new` and `operator delete` have already been written for us in the C++ Standard Library. We could write our own version of them, if we thought we could do better ourselves. All we have to do is write two functions with the same name, arguments, and return type as the original functions `operator new` and `operator delete`. The `new` operator that allocates and initializes a scalar, and its corresponding `delete` operator, would then call the functions `operator new` and `operator delete` that we wrote.

Let’s write a simple function `operator new` and `operator delete` that produce tracing output. Anticlimacticly, they rely on `malloc` and `free` to perform the actual allocation and deallocation. Stubbornly, our function `operator new` keeps looping as long as the call to `malloc` keeps failing (line 25).

If `malloc` has failed, line 27 checks to see if a handler has been established by a previous call to `set_new_handler`. Each call to `set_new_handler` returns the address of the previous handler function, or zero if there was no previous one. (The variable `f` is a pointer to a function. An `if` whose parentheses contain a variable declaration is true if the variable is non-zero; see pp. 38−39.) An unfortunate side effect of line 27 is to disestablish the handler function, so we need line 28 to re-establish it. If there was no handler function, line 31 constructs and throws an “exception”.

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

```
 1 #include <iostream>
 2 #include <cstdlib>   //for malloc and free, exit, EXIT_SUCCESS, EXIT_FAILURE
 3 #include <new>       //for set_new_handler, bad_alloc
 4 #include "obj.h"
 5 using namespace std;
 6
 7 void my_new_handler();
 8 void *operator new(size_t n);    //declaration for function in line 21
 9 void operator delete(void *p);   //declaration for function in line 39
10
11 int main()
12 {
13     set_new_handler(my_new_handler);
14
```

```
    const obj *const p = new obj(10);   //calls operator new in line 21
    delete p;                           //calls operator delete in line 39

    return EXIT_SUCCESS;
}

void *operator new(size_t n)
{
    void *p;                            //uninitialized variable

    while ((p = malloc(n)) == 0) {
        cerr << "operator new(" << n << ") out of store.\n";
        if (void (*const f)() = set_new_handler(0)) {
            set_new_handler(f);
            (*f)();             //call the handler function
        } else {
            throw bad_alloc();
        }
    }

    cout << "operator new(" << n << ") returns " << p << "\n";
    return p;
}

void operator delete(void *p)
{
    cout << "operator delete(" << p << ")\n";
    free(p);
}

void my_new_handler()
{
    cerr << "out of store\n";
    exit(EXIT_FAILURE);
}
```

```
operator new(4) returns 0x21c00
construct 10
destruct 10
operator delete(0x21c00)
```

An operator new function that does nothing

Any arguments written after a `new` operator will be passed along, after the `size_t` argument, to the function `operator new` that the `new` operator calls. For example, we have already seen the expression

```
    new(q) date(day, month, year)
```

in line 26 of `placement.C` in p. 406. This calls a standard library function `operator new` that returns its second argument. The first argument is unused, so it has no name (pp. 289−290).

```
void *operator new(size_t, void *p)
{
    return p;
}
```

See pp. 625−628 for another extra argument for `operator new`; pp. 625−628 and pp. 501−503 for an extra argument for `operator delete`.

Allocation and deallocation functions for arrays

A `new` operator that allocates and initializes an array, and the corresponding `delete[]` operator, call a different pair of functions to allocate and deallocate the memory for the array. The assumption is that bigger blocks are required for arrays, which might have to be allocated using a different strategy than for scalars.

```
void *operator new[](size_t n);
void operator delete[](void *p);
```

The argument of the function `operator new[]` tells it how many bytes we want. It assumes that the block will be occupied by an array, but is otherwise similar in its ignorance to the C function `malloc`. It does not know the data type of the elements of the array. The argument of the function `operator delete[]` is the address of the block to be deallocated.

These two functions, `operator new[]` and `operator delete[]` have already been written for us in the C++ Standard Library. We could write our own version of them, if we wanted to perform the allocation ourselves. All we have to do is write two functions with the same name, arguments, and return type as the original functions `operator new[]` and `operator delete[]`. A `new` operator that allocates and initializes an array, and the corresponding `delete[]` operator, would then call the functions `operator new[]` and `operator delete[]` that we wrote.

Here is a simple version of the functions `operator new[]` and `operator delete[]` that produce tracing output. Instead of writing our own memory allocator, we rely on `malloc` and `free` to allocate and deallocate.

The `new` operator in line 15 allocates an array whose elements have no destructor. It asks the function `operator new[]` for a block that is the same size as the array. The value of the `new` operator is the address it received from the function `operator new[]`, converted to the proper pointer type. The elements have no constructor either, so we get a block of garbage.

The `new` operator in line 32, on the other hand, allocates an array whose elements have a destructor. It asks the function `operator new[]` for a block that is `sizeof(size_t)` bytes bigger than the array. The `new` operator stores the number of array elements in this slot. The value of the `new` operator is the address of the byte after this slot, converted to a pointer to the data type of an array element.

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

```
 1 #include <iostream>
 2 #include <cstdlib>   //for malloc and free
 3 #include "obj.h"
 4 using namespace std;
 5
 6 void *operator new[](size_t n);
 7 void operator delete[](void *p);
 8
 9 int main()
10 {
11     cout << "How many elements do you want to allocate? ";
12     size_t n;
13     cin >> n;
14
15     int *const p1 = new int [n];   //calls operator new[] in line 48
16
17     for (size_t i = 0; i < n; ++i) {
18         cout << "The int at address " << p1 + i
```

```
19             << " holds " << p1[i] << ".\n";
20     }
21
22     cout << "The hidden numbers are "                          //unofficial
23         << reinterpret_cast<size_t *>(p1)[-2] << " and "
24         << reinterpret_cast<size_t *>(p1)[-1] << ".\n";
25
26     delete[] p1;                   //calls operator delete[] in line 62
27
28     cout << "\nAn obj is " << sizeof (obj) << " bytes, an array of " << n
29         << " of them is " << n * sizeof (obj)
30         << " bytes, and a size_t is " << sizeof (size_t) << " bytes.\n";
31
32     obj *const p2 = new obj [n];   //calls operator new[] in line 48
33
34     for (size_t i = 0; i < n; ++i) {
35         cout << "The obj at address " << p2 + i
36             << " holds " << p2[i] << ".\n";
37     }
38
39     cout << "The hidden numbers are "
40         << reinterpret_cast<size_t *>(p2)[-3] << ", "
41         << reinterpret_cast<size_t *>(p2)[-2] << ", and "
42         << reinterpret_cast<size_t *>(p2)[-1] << ".\n";
43
44     delete[] p2;                   //calls operator delete[] in line 62
45     return EXIT_SUCCESS;
46 }
47
48 void *operator new[](size_t n)
49 {
50     if (void *const p = malloc(n)) {
51         cout << "operator new[](" << n << ") returns " << p
52             << " with hidden numbers "
53             << reinterpret_cast<size_t *>(p)[-2] << " and "
54             << reinterpret_cast<size_t *>(p)[-1] << ".\n";
55         return p;
56     }
57
58     cerr << "operator new[](" << n << ") out of store.\n";
59     exit(EXIT_FAILURE);
60 }
61
62 void operator delete[](void *p)
63 {
64     cout << "operator delete[](" << p << ") with hidden numbers "
65         << reinterpret_cast<size_t *>(p)[-2] << " and "
66         << reinterpret_cast<size_t *>(p)[-1] << ".\n";
67
68     free(p);
69 }
```

```
How many elements do you want to allocate? 3
operator new[](12) returns 0x220f0 with hidden numbers 17 and 0.
The int at address 0x220f0 holds 139520.   three int's of garbage
The int at address 0x220f4 holds 0.
The int at address 0x220f8 holds 0.
The hidden numbers are 17 and 0.
operator delete[](0x220f0) with hidden numbers 17 and 0.

An obj is 4 bytes, an array of 3 of them is 12 bytes, and a size_t is 4 bytes.
operator new[](16) returns 0x220f0 with hidden numbers 17 and 0.
default construct 0
default construct 0
default construct 0
The obj at address 0x220f4 holds 0.       4 bytes after return value of operator new[]
The obj at address 0x220f8 holds 0.
The obj at address 0x220fc holds 0.
The hidden numbers are 17, 0, and 3.
destruct 0
destruct 0
destruct 0
operator delete[](0x220f0) with hidden numbers 17 and 0.
```

Reduce the overhead with class-specific allocation functions

> SIR THOMAS MORE.
> A dispensation was granted so that the King [Henry VIII] might marry Queen Catherine [daughter of Ferdinand and Isabella], for state reasons. Now we are to ask the Pope to—dispense with his dispensation, also for state reasons?
>
> —Robert Bolt, *A Man for All Seasons,* Act One

The functions `operator new` and `operator delete` in the C++ Standard Library will be called to allocate variables of any data type. So will the ones we substituted for them above, and the ones in lines 38 and 49 of the following `main.C`. They must be flexible enough to allocate blocks of any requested size.

But the member functions `operator new` and `operator delete` of class `cookie`, in lines 9 and 26 of `cookie.C`, will be called to allocate and deallocate objects of only that one class. They can assume that each block will be exactly the same size (namely, `sizeof (cookie)`), letting us reduce the overhead on each block. The `cookie`'s, incidentally, are so called because they are all the same size, stamped out with a cookie cutter.

A function `operator new` and `operator delete` that are member functions are always static, even without the keyword `static`. They have to be—no object exists when the allocation function is called or when the deallocation function returns. Not even the memory for the object exists at these times. Were the functions non-static, there would be nothing for their implicit pointers to point to.

The member functions `operator new` and `operator delete` of class `cookie` allocate blocks of memory from the `buffer` of characters in line 8, which is big enough to hold `n cookies`. We are not allowed to mention `sizeof (cookie)` until after the `}` that ends the class in line 25, so we cannot declare the size of the buffer here. But we can get away with the empty `[`square brackets`]` in line 8 because the `buffer` data member is static. The number of characters can wait until the buffer is defined in line 6 of `cookie.C`.

The array of `bool`'s in line 9 keeps track of which blocks in the `buffer` are currently allocated. Each block has one `bool`, which is a smaller overhead than the eight-byte prefix. But each `bool` still occupies at least one byte. What we really want is an array of bits, such as the `bitset` in the C++ Standard Library. We will retrofit it here on pp. 461−463.

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

```
 1 #ifndef COOKIEH
 2 #define COOKIEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class cookie {
 7     static const size_t n = 1000; //can allocate this many cookies
 8     static char buffer[];
 9     static bool b[n];             //true if this slot is currently allocated
10
11     int i;
12 public:
13     cookie(int initial_i): i(initial_i) {
14         cout << "construct cookie " << i << "\n";
15     }
16
17     cookie(): i(0) {
18         cout << "default construct cookie " << 0 << "\n";
19     }
20
21     ~cookie() {cout << "destruct cookie " << i << "\n";}
22
23     void *operator new(size_t);
24     void operator delete(void *p);
25 };
26 #endif
```

The function `operator new` assumes that every `cookie` is the same size (`sizeof (cookie)`), so it never uses the `size_t` argument in line 9. To avoid the ‘‘unused argument’’ warning, we give it no name.

But the `cookie`’s will not always be the same size. When we have inheritance, some of the objects of a class will be bigger variants called ‘‘derived objects’’. Think of them as heavier isotopes of a chemical element. The member function `operator new` of class `cookie` will then need to use its `size_t` argument, and the member function `operator delete` will get an extra argument, also of type `size_t`, giving the size of the object to be deallocated. See pp. 501−503.

The casts of the `void *p` in lines 29 and 36 can be `static_`. A cast to or from any other pointer type, in lines 15 and 37, must be a `reinterpret_cast`. See p. 389.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "cookie.h"
 4 using namespace std;
 5
 6 char cookie::buffer[n * sizeof (cookie)];
 7 bool cookie::b[n];
 8
 9 void *cookie::operator new(size_t)
10 {
11     for (size_t i = 0; i < n; ++i) {
12         if (!b[i]) {
```

```
13             b[i] = true;
14             cookie *const p =
15                 reinterpret_cast<cookie *>(buffer) + i;
16             cout << "cookie::operator new() returns " << p
17                 << " [block " << i << "].\n";
18             return p;
19         }
20     }
21
22     cerr << "cookie::operator new out of store\n";
23     exit(EXIT_FAILURE);
24 }
25
26 void cookie::operator delete(void *p)
27 {
28     if (p < buffer || p >= buffer + sizeof buffer ||
29         (static_cast<char *>(p) - buffer) % sizeof (cookie) != 0) {
30
31         cerr << "cookie::operator delete: " << p
32             << " not from previous cookie::operator new.\n";
33         exit(EXIT_FAILURE);
34     }
35
36     const size_t i = static_cast<cookie *>(p) -
37         reinterpret_cast<cookie *>(buffer);
38
39     if (!b[i]) {
40         cerr << "cookie::operator delete: " << p << " [block " << i
41             << "] not currently allocated.\n";
42         exit(EXIT_FAILURE);
43     }
44
45     cout << "cookie::operator delete(" << p << ") [block " << i << "].\n";
46     b[i] = false;
47 }
```

The operators in lines 12 and 13 call the general-purpose allocation functions in lines 38 and 49 because we wrote no functions specifically for class `obj`. Lines 17 and 18 call the allocation functions that are members of class `cookie`. Lines 22 and 23 call the functions `operator new[]` and `operator delete[]` in the C++ Standard Library because we did not write them ourselves, either as members of `cookie` or as non-members. Lines 25 and 26 revert to the general-purpose functions in lines 38 and 49 because of the unary scope resolution operator `::` we saw in pp. 122−124. Lines 32 and 33 call mismatching functions.

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

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "obj.h"
4 #include "cookie.h"
5 using namespace std;
6
7 void *operator new(size_t n);
8 void operator delete(void *p);
9
```

```
10 int main(int argc, char **argv)
11 {
12     const obj *const pd = new obj(10);    //call operator new in line 38
13     delete pd;                            //call operator delete in line 49
14
15     cout << "\n";
16
17     cookie *const pc1 = new cookie(10);   //call cookie::operator new
18     delete pc1;                           //call cookie::operator delete
19
20     cout << "\n";
21
22     cookie *const pc2 = new cookie[3]; //call standard lib operator new[]
23     delete[] pc2;                      //call standard lib operator delete[]
24
25     cout << "\n";
26
27     cookie *const pc3 = ::new cookie(30); //call operator new in line 38
28     ::delete pc3;                         //call operator delete in line 49
29
30     cout << "\n";
31
32     cookie *const pc4 = ::new cookie(40); //call operator new in line 38
33     delete pc4;          //deliberate mismatch: call cookie::operator delete
34
35     return EXIT_SUCCESS;
36 }
37
38 void *operator new(size_t n)
39 {
40     if (void *const p = malloc(n)) {
41         cout << "operator new(" << n << ") returns " << p << ".\n";
42                 return p;
43         }
44
45     cerr << "operator new(" << n << ") out of store.\n";
46     exit (EXIT_FAILURE);
47 }
48
49 void operator delete(void *p)
50 {
51     cout << "operator delete(" << p << ")\n";
52     free(p);
53 }
```

```
operator new(4) returns 0x23948.                          lines 12−13
construct 10
destruct 10
operator delete(0x23948)

cookie::operator new() returns 0x22490 [block 0].         lines 17−18
construct cookie 10
destruct cookie 10
cookie::operator delete(0x22490) [block 0].

operator new(16) returns 0x25d60.                         lines 22−23
default construct cookie 0
default construct cookie 0
default construct cookie 0
destruct cookie 0
destruct cookie 0
destruct cookie 0
operator delete(0x25d60)

operator new(4) returns 0x23948.                          lines 27−28
construct cookie 30
destruct cookie 30
operator delete(0x23948)

operator new(4) returns 0x23948.                          lines 32−33
construct cookie 40
destruct cookie 40
cookie::operator delete: 0x23948 not from previous cookie::operator new.
```

▼ **Homework 4.2.4a:**

Write member functions `operator new[]` and `operator delete[]` for class `cookie`. It will be easier to search the array of `bool`'s when we have the ‘‘algorithms’’ `find` (p. 861) and `search_n` (p. 949).

A harried programmer may choose to define `operator new[]` and `operator delete[]` first. The scalar functions `operator new` and `operator delete` can then be implemented by allocating an array of one element.

▲

4.3 Vectors and Lists

4.3.1 Endow a Data Type with a Last Name

A *container* is a big object that contains little objects. The little objects don't even have to be objects. They can be pointers, structures, or merely values of the built-in data types. And the big object doesn't have to be an object, either. It could be an array, which is the most rudimentary type of container.

This chapter will introduce better types of containers, including `vector`, `list`, and `string`. First, however, we will need two preliminary techniques: how to give a last name to a data type, and how to loop through a container with an ‘‘iterator’’.

So far, we've seen three kinds of class members: data members (line 7), member functions (lines 9 and 19), and enumeration members (lines 12−14). But a member can also be a data type. For example, the `month_type` in line 11, the `hillary_t` in line 17, and the `bill` in line 21 are all public members of the class `clinton` in line 6. First we will say what this does not mean, and then what it does mean. Finally, we will show why you would want to do this.

It does not mean that a `clinton` object contains a `month_type`, a `hillary_t`, or a `bill`. In fact, we have already seen that the only data member in a `clinton` object is the `i` in line 7. By making `month_type`, `hillary_t`, and `bill` members of `clinton`, we have merely endowed the names of these three data types with the last name `clinton`. For example, the full name of the data type `hillary_t` is `clinton::hillary_t`. `bill`, by the way, is called a *nested class* because its declaration is inside the declaration for another class. Recall that we have already seen a variable with a last name: the `std::cout` in p. 20.

We already know that we are on a first-name basis with all the members of a class within the {curly braces} of the class declaration (lines 6 and 27 of `clinton.h`), and within the {curly braces} of the body of a member function of the class. That's why inside the body of the member function `f` in line 19 of `clinton.h`, the `i`, `january`, and `hillary_t` needed nothing in front of them. But outside these {curly braces}, we have to identify which class the members of class `clinton` belong to. That's why in `main.C`, `january` in line 9, the `hillary_t` in line 10, and the `bill` in line 12 all need the `clinton::`. Of course, `january`, `hillary_t`, and `bill` all have to be public members of class `clinton` merely to appear in `main`.

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

```
 1 #ifndef CLINTONH
 2 #define CLINTONH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class clinton {
 7     int i;
 8 public:
 9     clinton(int initial_i): i(initial_i) {}
10
11     enum month_type {
12         january = 1,
13         february,
14         march
15     };
16
17     typedef unsigned hillary_t;
18
19     void f() const {cout << i << " " << january << " " << sizeof (hillary_t) << "\n";}
20
21     class bill {
22         int j;
23     public:
24         bill(int initial_j): j(initial_j) {}
25         void g() const {cout << j << "\n";}
26     };
27 };
28 #endif
```

We give a last name to a data type so that we can have a different data type with the same name in the same program. Class `vector` will provide our first real example of two data types with the same name. In the meantime, here is another class `bill`.

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

```
#ifndef GATESH
#define GATESH
#include <iostream>
using namespace std;

class gates {
public:
    class bill {
        double d;
    public:
        bill(double initial_d): d(initial_d) {}
        void g() const {cout << d << "\n";}
    };
};
#endif
```

The only purpose of class `gates` was to give the last name `gates` to its class `bill`. Class `gates` has no other members. If you feel that the `bill` inside it makes `gates` appear distended, here is another way to do the same thing. Now the curly braces of `gates` are close to each other (lines 6 and 9).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/clinton/gates2.h`

```
 1 #ifndef GATESH
 2 #define GATESH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class gates {
 7 public:
 8     class bill;       //declaration for class gates::bill
 9 };
10
11 class gates::bill {   //definition for class gates::bill
12     double d;
13 public:
14     bill(double initial_d): d(initial_d) {}
15     void g() const {cout << d << "\n";}
16 };
17 #endif
```

To give `bill` a last name by means of a ‘‘namespace’’, see pp. 1024−1025.

The variable `bc` in line 12 is not a `clinton` object or a data member of a `clinton` object. In fact, we haven’t constructed any `clinton` objects at all. `bc` is merely of a data type whose last name is `clinton`.

Similarly, the variable `bg` in line 15 is not a `gates` object or a data member of a `gates` object. In fact, we haven’t constructed any `gates` objects at all, and even if we did, a `gates` object would have no data members. `bg` is merely of a data type whose last name is `gates`.

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

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "clinton.h"
4 #include "gates.h"
5 using namespace std;
```

```
 6
 7 int main()
 8 {
 9     cout << clinton::january << "\n";  //The last name of january is clinton.
10     clinton::hillary_t n = 10;         //The last name of hillary_t is clinton.
11
12     clinton::bill bc = 20;             //The last name of bill is clinton.
13     bc.g();
14
15     gates::bill bg = 3.14159265358979323846;
16     bg.g();
17
18     return EXIT_SUCCESS;
19 }
```

```
1           line 9
20          line 13
3.14159     line 16
```

Where can we call hillary_t by her first name?

The following example underlines all the territory where we are on a first-name basis with `hillary_t`. Both ways of defining a member function, inline and non-inline, are shown.

As mentioned above, we are on a first-name basis with all the members of class anywhere within the {curly braces} of the body of a member function of that class. Therefore we don't need anything in front of the third `hillary_t` in line 10 of `clinton2.h` and the `hillary_t` in line 5 of `clinton2.C`.

We're also on a first-name basis with all the members of a class anywhere within the parentheses of the argument list of a member function of that class. Therefore we don't need anything in front of the second `hillary_t` in lines 10 and 11 of `clinton2.h` or in front of the second `hillary_t` in line 3 of `clinton2.C`.

We're also on a first-name basis with all the members of a class anywhere within the {curly braces} of the class declaration for that class (lines 4−12 of `clinton2.h`). Therefore we don't need anything in front of the first `hillary_t`'s in lines 10 and 11 of `clinton2.h`.

But outside of these three places, we are not on a first-name basis with `hillary_t`. That's why we need a `clinton2::` in front of the first `hillary_t` in line 3 of `clinton2.C`, and why we needed a `clinton::` in front of the `hillary_t` in line 10 of the above `main.C`.

At the start of line 3 of `clinton2.C`, the `clinton2::hillary_t` is the return type of the member function. Then the `clinton2::g` is the name of the member function.

```
 1 #ifndef CLINTON2H       //This file is clinton2.h.
 2 #define CLINTON2H
 3
 4 class clinton2 {
 5     int i;
 6 public:
 7     clinton2(int initial_i): i(initial_i) {}
 8
 9     typedef int hillary_t;
10     hillary_t f(hillary_t h) const {return sizeof(hillary_t);}
11     hillary_t g(hillary_t h) const;
12 };
13 #endif
```

```
#include "clinton2.h"   //This file is clinton2.C.

clinton2::hillary_t clinton2::g(hillary_t h) const
{
    return sizeof(hillary_t);
}
```

▼ **Homework 4.3.1a: make a typedef member**

We had typedefs floating near classes `stack`, `life`, and `employee`, but we didn't know where to put them. Now we have a place for them to go.

Let the typedef `value_type` on pp. 153−154 be a public member of class `stack`. The `typedef` must be in the `public` section of the class declaration, but it also has to come before it is used in line 6 of the `private` section. The `public` and `private` sections must therefore alternate

```
 1 class stack {
 2 public:
 3     typedef int value_type;               //must come before line 6
 4 private:
 5     static const size_t max_size = 100;   //must come before line 6
 6     value_type a[max_size];
 7     size_t n;
 8 public:
 9     stack(): n(0) {}
10     ~stack();
11
12     void push(value_type i);
13     value_type pop();
14     size_t size() const {return n;}
15 };
```

Do the same for the `value_type` of class `node` in `node.h` on p. 214, and the `ss_t` of class `employee` on p. 259.

▲

▼ **Homework 4.3.1b: make a typedef member**

You can do this homework only with a version of C++ that permits the initialization of a static data member in line 30 on p. 238.

In ¶ (2) of the homework on p. 239, we thought about letting let `life_xmax` and `life_ymax` be private static data members of the class `life` on pp. 144−147. Do it now, and rename them `xmax` and `ymax`. Initialize the new static data members as in line 30 on p. 238.

Then let the typedefs `_life_matrix_t` and `life_matrix_t` be members of class `life` (as `hillary_t` is a public member of class `clinton`), and shorten their names to `_matrix_t` and `matrix_t`.

```
1 class life {
2     static const size_t xmax = 10;          //must come before lines 4 and 7
3     static const size_t ymax = 10;
4     typedef bool _matrix_t[ymax + 2][xmax + 2];//must come before line 5
5     _matrix_t matrix;
6 public:
7     typedef bool matrix_t[ymax][xmax];      //must come before line 8
8     life(const matrix_t initial_matrix);
```

`_life_matrix_t` can become `_matrix_t` within the {curly braces} of the class declaration for class `life`, and within the bodies and argument lists of the member functions of class `life`. Similarly, `life_matrix_t` can become `matrix_t` within the {curly braces} of the class declaration for class `life`, and within the bodies and argument lists of the member functions of class `life`. For example, the first argument of the constructor can become a `matrix_t` in the above line 8; and we saw `hillary_t` in line 10 of `clinton2.h`.

But outside of these places, `life_matrix_t` will have to become `life::matrix_t`, just like `clinton::hillary_t` in line 10 of the above `main.C`. For example, in the `main` function that plays the game of life, you will have to change

```
 9     life_matrix_t glider_matrix = {
```

to

```
10     life::matrix_t glider_matrix = {
```

▲

4.3.2 Iterators

Looping through a container

A *container* is a big object that contains smaller objects. The smaller objects don't even have to be objects: they can be values of a built-in data type such as `int`. And the big object doesn't have to be an object, either: it can even be a plain, old array.

The values held in a container are called its *elements.* The elements of a container, like the elements of an array, can be pointers but not references. A reference has no memory address, so it cannot be contained in anything.

An array is only the most rudimentary type of container. As we are about to see, it lacks some of the standard features of a C++ container. Vectors and the standard library `stack` in pp. 155−157 are better containers because they are safer and easier to use. These and other container classes belong to a part of the C++ Standard Library called the Standard Template Library, or *STL.*

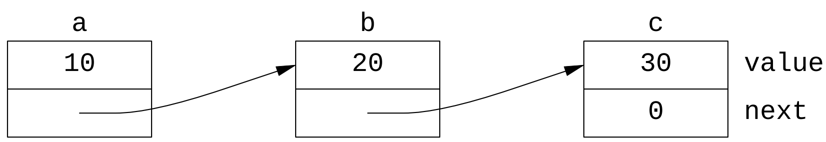
When looping through a container, we always need a loop variable to keep track of how far we have progressed. If the container is an array or vector, the variable could be the pointer to `int` in lines 4−6:

```
 1     int a[] = {10, 20, 30};
 2     const size_t n = sizeof a / sizeof a[0];
 3
 4     for (const int *p = a; p < a + n; ++p) {
 5         cout << *p << "\n";
 6     }
```

But a different type of container would need a different type of loop variable. If the container is a linked list, the variable would have to be the pointer to each element in the list in lines 18−20:

```
 7     struct node {
 8         int value;
 9         node *next;
10     };
11
12     node c = {30, 0};
13     node b = {20, &c};
14     node a = {10, &b};
15
16     node *begin = &a;   //point to 1st node in list, or 0 if list empty
17
```

```
18     for (const node *p = begin; p != 0; p = p->next) {
19         cout << p->value << "\n";
20     }
```



The loops in the above examples were quite different. Now let’s contemplate something that is rarely attempted in C. To make it easy to switch from one type of container to another, we would like to be able to loop through any container by writing the same looping code. To switch containers, we will have to hide the different loop variables, with their names and data types: the `int *p` in the above line 4 vs. the `node *p` in line 18. We will also have to hide three pieces of code:

(1) the different pieces of code that use the variable to access each item in the container: the `*p` in line 5 vs. the `p->value` in line 19;

(2) the different pieces of code that advance the variable: the `++p` in line 4 vs. the `p = p->next` in line 18;

(3) the different pieces of code that test the variable: the `p < a + n` in line 4 vs. the `p != 0` in line 18.

Iterators

In C++, a variable’s name and data type are hidden by making it a private data member of some object. An object that hides a loop variable is called an *iterator.* We say that the iterator *refers to* one of the elements in the container through which we are looping.

Code is hidden by putting it into the body of a function. The three functions of a C++ iterator are conventionally named `operator*`, `operator++`, and `operator!=`. Most iterators also have an `operator--`.

Each container class requires a different class of iterator. For example, an iterator for looping through the above array would contain the pointer to an `int` in line 5. We could also have made a postfix `operator++`, and a corresponding pair of `operator--`’s.

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

```
 1 #ifndef ARRAY_ITERATORH
 2 #define ARRAY_ITERATORH
 3
 4 class array_iterator {
 5     int *p;
 6 public:
 7     array_iterator(int *initial_p): p(initial_p) {}
 8     int& operator*() const {return *p;}
 9     array_iterator& operator++() {++p; return *this;}
10
11     friend bool operator!=(const array_iterator& it1,
12                            const array_iterator& it2) {
13         return it1.p != it2.p;
14     }
15 };
16 #endif
```

If we also create the two functions `begin` and `end` in lines 9−10, returning iterators that refer to the beginning and end of the array, we can rewrite the loop as follows.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/vector/array1.C`

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

int a[] = {10, 20, 30};
const size_t n = sizeof a / sizeof a[0];

inline array_iterator begin() {static const array_iterator it(a);   return it;}
inline array_iterator   end() {static const array_iterator it(a+n); return it;}

int main()
{
    for (array_iterator it = begin(); it != end(); ++it) {
        cout << *it << "\n";   //cout << it.operator*() << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
10
20
30
```

On the other hand, an iterator for looping through the linked list would contain the pointer to a `node` in line 10.

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

```
 1 #ifndef LIST_ITERATORH
 2 #define LIST_ITERATORH
 3
 4 struct node {
 5     int value;
 6     node *next;
 7 };
 8
 9 class list_iterator {
10     node *p;
11 public:
12     list_iterator(node *initial_p): p(initial_p) {}
13     int& operator*() const {return p->value;}
14     list_iterator& operator++() {p = p->next; return *this;}
15
16     friend bool operator!=(const list_iterator& it1,
17                            const list_iterator& it2) {
18         return it1.p != it2.p;
19     }
20 };
21 #endif
```

If we also create the two functions `begin` and `end` in lines 10 and 16, returning iterators that refer to the beginning and end of the linked list, we can rewrite the loop as follows.

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

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

node c = {30, 0};
node b = {20, &c};
node a = {10, &b};

inline const list_iterator& begin()
{
    static const list_iterator it(&a);
    return it;
}

inline const list_iterator& end() {
    static const list_iterator it(0);
    return it;
}

int main()
{
    for (list_iterator it = begin(); it != end(); ++it) {
        cout << *it << "\n";   //cout << it.operator*() << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
10
20
30
```

Our loops are now identical, except for the name of the data type of the iterator. (We will eventually use a “template” to switch this name.) All three iterators have the outward appearance of a pointer to `int`. In fact, a much simpler implementation is possible for one of the iterators. The `array_iterator` can be the typedef in line 8 for a plain old pointer to an `int`:

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/vector/array2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int a[] = {10, 20, 30};
 6 const size_t n = sizeof a / sizeof a[0];
 7
 8 typedef int *array_iterator;
 9 inline array_iterator begin() {return a;}
10 inline array_iterator   end() {return a + n;}
11
```

```
int main()
{
    for (array_iterator it = begin(); it != end(); ++it) {
        cout << *it << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
10
20
30
```

The iterators we will use in real life have two improvements over our `array_iterator` and `list_iterator`. First, each class of iterator will have the same name, simply `iterator`. To make this possible, each one will have a different last name. The name will always be that of the container class through which the iterator loops. Unfortunately, we could not illustrate this with our `array_iterator` and `list_iterator`. The array and linked list were not objects, so they belonged to no class. But we will soon see a container class with the strange name `vector<int>`, and will use a `vector<int>::iterator` to loop through an object of this class.

Second, each container class already has two public member functions, `begin` and `end`, which return the initial and final values for an iterator that will loop through the container. The return value of `begin` is an iterator that refers to the first element of the container, provided, of course, that the container has a first element. Some containers are empty.

On the other hand, the return value of `end` does *not* refer to the last element of the container. It refers to the empty location just beyond the last element. If the container is empty, the return value of `begin` will also refer to this location: `begin` and `end` will be the same.

Unfortunately, we had to illustrate our `array_iterator` and `list_iterator` with `begin` and `end` functions that were not member functions. The array and linked list were not classes, so they couldn't have member functions.

The operators `*`, `++`, and `!=` were chosen to make every iterator look like a pointer looping through an array. In fact, an iterator is sometimes thought of as any variable to which we can repeatedly apply these three operators to get data from somewhere or to put data to somewhere. The "somewhere" is called a container. By these definitions, there are many kinds of containers besides arrays and vectors.

Thanks to these operators, we can now use exactly the same notation to loop through any kind of container: array, vector, list, stack, queue, deque (double-ended queue), etc. The consistency of the notation will eventually make our templates applicable to more types of containers.

4.3.3 Class `vector`

Three drawbacks of an array

A C++ vector is an improved array. To motivate its introduction, we list the drawbacks of a C or C++ array.

(1) There is no way to make an array grow or shrink at *runtime,* as the program runs. Even if the size of the array is fixed, there is no way to determine the size at runtime.

```
#include <iostream>
#include <cstddef>      //for size_t
using namespace std;

    cout << "How many array elements do you need?\n";
    size_t n;           //use this data type for the number of elements in an array
```

```
 7      cin >> n;
 8      int a[n];           //won't compile: can't use a variable as the dimension
```

The number of elements must be fixed at *compile time*, when the program is written. To allocate a block of memory whose size may be set and changed at runtime, C programmers must resort to the functions `malloc`, `realloc`, and `free`. C++ programmers have a better alternative which we are about to see.

(2) An array performs no subscript checking. If a subscript is out of bounds, the program will blow up. If we're lucky.

```
 9      int a[] = {10, 20, 30};
10      cout << a[3] << "\n";           //subscript out of range
```

(3) To copy and compare arrays, we have to write loops:

```
11      int a[] = {10, 20, 30};
12      int b[3];
13
14      //Copy a into b.
15      for (size_t i = 0; i < 3; ++i) {
16          b[i] = a[i];
17      }
18
19      //Compare a and b.
20      for (size_t i = 0; i < 3; ++i) {
21          if (a[i] != b[i]) {
22              cout << "The arrays are unequal.\n";
23              goto done;
24          }
25      }
26      cout << "The arrays are equal.\n";
27      done:;
```

I wish that arrays could be copied and compared like *scalars,* i.e., variables that are not arrays:

```
28      int s = 10;         //s and t are scalars
29      int t = s;          //s can be copied with an =
30
31      if (s == t) {       //s and t can be compared with an ==
```

Line 35 will compile, but it does the wrong thing.

```
32      int a[] = {10, 20, 30};
33      int b[3] = a;       //won't compile: a cannot be copied with an =
34
35      if (a == b) {       //compares the addresses, not the contents
```

Class vector

A *vector* is an improved array. Class `vector` is a *template* class: one whose name contains the name of another data type, inserted into <angle brackets>. The vector of class `vector<int>` in line 8 will store and retrieve `int`'s, as will the stack of class `stack<int>` in pp. 155−157.

A vector acts as a one-dimensional array. If more than one dimension is needed, use a slice of a `valarray`. This kind of slice has nothing to do with the bad kind of slicing.

The template class `vector` is declared in the header file `<vector>`. The following program shows five constructors for class `vector<int>`, in lines 8−10, 14, and 16. The one-argument constructor in line 9 initializes each `int` in the vector to zero because it calls the no-argument constructor for the data type `int`. In line 19 of `main.C` on p. 142, we saw that this no-argument constructor creates an `int` whose value is zero.

Lines 9 and 16 call two different one-argument constructors for class `vector<int>`. The argument in line 9 has parentheses to emphasize that a function is being called; the one in line 16 has an equal sign to emphasize that the object `v4` is being copied. This is the conventional notation for calling the copy constructor.

A vector has three advantages over an array.

(1) A vector can grow as the program runs, which we will demonstrate shortly. Instead of growing a block of memory by calling the functions `malloc`, `realloc`, and `free`, we will call the member functions of a vector object. The number of elements currently in use is called the vector's *size;* the number of elements for which there is room is called the vector's *capacity.* To get the size and capacity, call the `size` and `capacity` member functions in lines 19−20. The output of these lines show that the vector `v5` is born filled to capacity.

Do not attempt to get the current number of elements in a vector `v` by saying

```
sizeof v / sizeof v[0]
```

The `sizeof` a variable never changes as the program runs. It is determined once and for all when the program is compiled.

The member function `empty` in line 18 returns a `bool`, true if the size of the vector is zero. By default, a `bool` prints as a `1` or `0`. To change this, see line 30 of `int.C` on p. 354.

(2) A vector will give us a civilized error message in response to a bad subscript. We won't be able to do this until we cover exceptions. But let's begin to look at what happens when we apply a subscript to a vector.

When we write line 22, the computer behaves as if we had written the code in the comment beside it. We are really calling the member function `operator[]`, and the number we wrote in the square brackets is passed as an argument to this function. The subscripts start at zero, so the `v5[1]` in line 22 is the second element of the vector.

The member function `operator[]` performs no subscript checking: it lives fast and dangerously. But another member function, `at`, will perform subscript checking. When we do exceptions, we will change the expression `v5[1]` in line 22 to `v5.at(1)`.

(3) A vector can be compared to another vector with the `<` in line 26, and copied into another vector with the `=` in line 27. To compare and copy arrays, we would have to write `for` loops.

The comparison in line 26 works the same way as string comparison. It loops through the two vectors in tandem, searching for the first mismatching pair of elements. In the case of `v3` and `v4`, the first mismatch is at subscript 1 (the second element). Since `v3[1]` is less than `v4[1]`, the comparison in line 26 yields the value `true`. If no mismatch is encounted, the vectors count as equal if they are the same size; otherwise, the shorter one counts as being smaller.

Warning: the two arguments in line 10 would be in the opposite order if `v3` were a `valarray`. See line 10 of `sieve.C` on p. 902.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     vector<int> v1;          //born empty, but we can insert int's later
 9     vector<int> v2(3);       //born containing 0, 0, 0
10     vector<int> v3(3, 10);   //born containing 10, 10, 10
```

```

    const int a[] = {10, 20, 30};
    const size_t n = sizeof a / sizeof a[0];
    vector<int> v4(a, a + n); //born containing 10, 20, 30

    vector<int> v5 = v4;      //born containing 10, 20, 30: copy constructor

    cout << "v5.empty() == " << v5.empty() << "\n"
        << "v5.size() == " << v5.size() << "\n"
        << "v5.capacity() == " << v5.capacity() << "\n\n";

    cout << v5[1] << "\n";    //cout << v5.operator[](1) << "\n";
    v5[1] = 21;               //Change the 20 to 21: v5.operator[](1) = 21;
    cout << v5[1] << "\n";    //cout << v5.operator[](1) << "\n";

    if (v3 < v4) {            //Compare two vectors: if (operator<(v3, v4)) {
        v1 = v5;              //assignment: v1.operator=(v5);
    }

    return EXIT_SUCCESS;
}
```

By default, a `bool` is output as `1` or `0`. To change this, see p. 354.

```
v5.empty() == 0
v5.size() == 3
v5.capacity() == 3

20
21
```

Make a vector larger by calling push_back

We can add an extra element to the end of a vector by calling its `push_back` member function. For a `vector<int>`, the argument of `push_back` will be an `int`.

Each call to `push_back` adds 1 to the size of the vector. If the new size exceeds the capacity, the latter is automatically increased. On my platform, the call to `push_back` in line 15 doubles the capacity from 3 to 6. Line 22 doubles it again, from 6 to 12.

My `vector` behaves this way because the more the size increases, the more probable it is that a further increase is coming. The C++ Standard doesn't actually say that the capacity has to be doubled each time it is increased. But let's see what would go wrong if the capacity was merely increased by 1.

An increase in capacity has to do more than just allocate a bigger block of memory. It must also copy the existing elements into the new block. For example, imagine that we started with an empty vector and called `push_back` n times. The second call to `push_back` would copy the one existing element into a new block. The third call to `push_back` would copy two existing elements. The n calls would copy a total of

$$1 + 2 + 3 + \cdots + n - 1 = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

elements. The time it would take is therefore proportional to the *square* of the number of elements. But this "quadratic" behavior is too slow for the C++ Standard, which demands "amortized constant time".

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const int a[] = {10, 20, 30};
 9     const size_t n = sizeof a / sizeof a[0];
10     vector<int> v(a, a + n);   //born containing 10, 20, 30
11
12     cout << "v.size() == " << v.size()
13         << ", v.capacity() == " << v.capacity() << "\n";
14
15     v.push_back(40);
16
17     cout << "v.size() == " << v.size()
18         << ", v.capacity() == " << v.capacity() << "\n";
19
20     v.push_back(50);
21     v.push_back(60);
22     v.push_back(70);
23
24     cout << "v.size() == " << v.size()
25         << ", v.capacity() == " << v.capacity() << "\n";
26
27     return EXIT_SUCCESS;
28 }
```

`v.size() == 3, v.capacity() == 3`	*lines 12−13*
`v.size() == 4, v.capacity() == 6`	*lines 17−18*
`v.size() == 7, v.capacity() == 12`	*lines 24−25*

Make a vector larger by calling reserve

The capacity of a vector can be changed manually by calling the `reserve` member function. Do this before calling `push_back` to avoid the automatic doubling.

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

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

int main()
{
    const int a[] = {10, 20, 30};
    const size_t n = sizeof a / sizeof a[0];
    vector<int> v(a, a + n);   //born containing 10, 20, 30

    cout << "v.size() == " << v.size()
        << ", v.capacity() == " << v.capacity() << "\n";

```

```
15     v.reserve(7);   //Prevent the push_back's from increasing the capacity.
16
17     cout << "v.size() == " << v.size()
18         << ", v.capacity() == " << v.capacity() << "\n";
19
20     v.push_back(40);
21     v.push_back(50);
22     v.push_back(60);
23     v.push_back(70);
24
25     cout << "v.size() == " << v.size()
26         << ", v.capacity() == " << v.capacity() << "\n";
27
28     return EXIT_SUCCESS;
29 }
```

`v.size() == 3, v.capacity() == 3`	*lines 12−13*
`v.size() == 3, v.capacity() == 7`	*lines 17−18*
`v.size() == 7, v.capacity() == 7`	*lines 25−26*

Two data types with the same first name and different last names

The above programs printed the return value of the `size` and `capacity` member functions of class `vector`. Now we would like to store these values into a variable. What data type should it be?

The C++ Standard Library contains a typedef `size_type` for the data type of a variable that holds the `size` or `capacity` member function of any vector. But there is a complication.

Suppose our machine has 1,000,000 bytes of memory. If `sizeof (int) == 4`, the biggest possible `vector<int>` would have 250,000 elements. A variable that holds the return value of `vector<int>::size` would have to be big enough to hold the number 250,000. Anything bigger would be wasteful.

```
    vector<int> vi(3, 10);          //born containing 10, 10, 10
    size_type s = vi.size();
```

If `sizeof (char) == 1`, the biggest possible `vector<char>` would have 1,000,000 elements. A variable that holds the return value of `vector<char>::size` would have to be big enough to hold the number 1,000,000. Again, anything bigger would be wasteful.

```
    vector<char> vc(3, 'A');        //born containing 'A', 'A', 'A'
    size_type s = vc.size();
```

To let us use the same name, `size_type`, for these two different data types, they have been given two different last names:

```
    vector<int> vi(3, 10);
    vector<int>::size_type s = vi.size(); //variable big enough to hold 250,000

    vector<char> vc(3, 'A');
    vector<char>::size_type s = vc.size(); //variable big enough to hold 1,000,000
```

Often the name of a container is used as the last name of a data type that helps us loop through the container. The other examples we have seen are `value_type` and `difference_type`:

```
    vector<int> v(3, 10);
    vector<int>::size_type s = v.size();
    vector<int>::value_type i = v[0];
```

```
12     vector<int>::difference_type d = v.end() - v.begin();
```

(In the above line 11, why not say a simple `int` instead of `vector<int>::value_type`? We will return to this when we know more about templates.)

Loop through a vector with an iterator

We now discard the `size_type` `i` in line 15 in favor of the iterator `it` in line 26. If the data type `vector<int>::iterator` is a typedef for `int *`, the operators `!=`, `*`, and `++` in lines 26−28 will be the built-in ones that operate on pointers. If the data type `vector<int>::iterator` is a class, lines 26−28 will make the computer behave as if we had written lines 20−23. Think of lines 20−23 as an "exploded view" of 26−28.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const int a[] = {10, 20, 30};
 9     const size_t n = sizeof a / sizeof a[0];
10     vector<int> v(a, a + n);    //born containing 10, 20, 30
11
12     const vector<int>::size_type s = v.size();
13     cout << "size == " << s << "\n\n";
14
15     for (vector<int>::size_type i = 0; i < s; ++i) {
16         cout << v[i] << "\n";   //cout << v.operator[](i) << "\n";
17     }
18     cout << "\n";
19
20     for (vector<int>::iterator it = v.begin(); operator!=(it, v.end());
21         it.operator++()) {
22         cout << it.operator*() << "\n";
23     }
24     cout << "\n";
25
26     for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
27         cout << *it << "\n";
28     }
29     cout << "\n";
30
31     return EXIT_SUCCESS;
32 }
```

```
size == 3                lines 12−13

10                       lines 15−17
20
30

10                       lines 20−23
20
30

10                       lines 26−28
20
30
```

Warning. When a `vector`’s capacity is increased, the elements are copied into a bigger block of memory. This means that an iterator referring to an element in the original block will behave unpredictably when dereferenced or incremented.

```
    vector<int> v(argument(s) for constructor);
    vector<int>::iterator it = v.begin();
    cout << *it << "\n";            //can dereference it here

    v.push_back(10);                //might increase the capacity
    v.reserve(v.size() + 10);       //definitely increases the capacity
    //cout << *it << "\n";          //can no longer dereference it here

    it = v.begin();
    cout << *it << "\n";            //can dereference new value of it
```

Two ways to make a pointer const

`p1` always points to the same variable. `p2` gives us read-only access to `a`. We saw this in pp. 50−52.

```
#include <cstdlib>

int main()
{
    int a[] = {10, 20, 30};

    int *const p1 = a;
    ++p1;      //won’t compile: p1 must always point to a[0]

    const int *p2 = a;
    ++*p2;     //won’t compile: can’t use p2 to change a[0] from 10 to 11

    const int *const p3 = a;        //both of the above
    ++p3;      //won’t compile: p3 must always point to a[0]
    ++*p3;     //won’t compile: can’t use p3 to change a[0] from 10 to 11

    a[0] = 11; //a is not a const array.
    return EXIT_SUCCESS;
}
```

Two ways to make an iterator const

An iterator can be made constant in the same two ways, but the syntax is different. `it1` always refers to the same element. `it2` gives us read-only access to `v`.

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

```
 1 #include <cstdlib>
 2 #include <vector>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int a[] = {10, 20, 30};
 8     size_t n = sizeof a / sizeof a[0];
 9     vector<int> v(a, a + n);
10
11     const vector<int>::iterator it1 = v.begin();
12     //++it1;     //won't compile: it1 must always refer to v[0]
13
14     vector<int>::const_iterator it2 = v.begin();
15     //++*it2;    //won't compile: can't use it2 to change v[0] from 10 to 11
16
17     const vector<int>::const_iterator it3 = v.begin();   //both of the above
18     //++it3;     //won't compile: it3 must always refer to v[0]
19     //++*it3;    //won't compile: can't use it3 to change v[0] from 10 to 11
20
21     v[0] = 11;   //v.operator[](0) = 11
22     return EXIT_SUCCESS;
23 }
```

A vector of objects

Class `obj` is in pp. 179−180. It will let us ‘‘x-ray’’ a vector to see exactly how many `obj`’s the vector constructs and destructs, and in what order. These statistics may be different on each platorm.

I thought line 11 would construct three `obj`’s by calling the default constructor for class `obj` three times. But the output shows that it constructed *four* `obj`’s: one by the default constructor and three by the copy constructor. The author of class `vector` must have assumed that for most classes, the copy constructor is less expensive than the default constructor. This is certainly the case for class `date`: its default constructor calls system functions to get and parse the current date, while its copy constructor merely copies the integer data member(s).

The choice of constructors is not only a performance issue. The calls to the copy constructor ensure that the three objects in the array will be as identical as the copy constructor can make them. If these objects had been constructed by three calls to the default constructor, they might not have been identical. Different constructors can do different things.

Line 14 can be used only for objects whose constructor takes exactly one argument. If the constructor needs more than one argument we must use line 17, which would allow more than one argument in the innermost parentheses.

On some platforms line 14 constructs fewer objects than line 17, and is therefore to be preferred. But a superficial work-around would let us use line 14 even for objects whose constructor takes more than one argument. Simply define a one-argument constructor whose argument is a structure containing more than one field.

Similarly, lines 20−26 can be used only for objects whose constructor takes one argument. If there is more than one argument (or with archaic versions of Microsoft Visual C++) we must use lines 29−36, which would allow more than one argument in the parentheses in lines 30−32. On some platforms lines 20−26 construct fewer objects than lines 29−36, and are therefore to be preferred. But we can apply the same workaround.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include "obj.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     vector<obj> v1;      //born empty
10
11     vector<obj> v2(3);
12     cout << "\n";
13
14     vector<obj> v3(3, 10);
15     cout << "\n";
16
17     vector<obj> v4(3, obj(20));
18     cout << "\n";
19
20     const int a[] = {
21         30,
22         40,
23         50
24     };
25     const size_t na = sizeof a / sizeof a[0];
26     vector<obj> v5(a, a + na);
27     cout << "\n";
28
29     const obj b[] = {
30         obj(60),
31         obj(70),
32         obj(80)
33     };
34     cout << "\n";
35     const size_t nb = sizeof b / sizeof b[0];
36     vector<obj> v6(b, b + nb);
37     cout << "\n";
38
39     for (vector<obj>::const_iterator it = v6.begin(); it != v6.end(); ++it) {
40         cout << *it << "\n"; //can also say (*it).print() or it->print()
41     }
42     cout << "\n";
43
44     return EXIT_SUCCESS;
45 }
```

The objects defined in `main` are destructed when we return from `main` in line 44. The vectors are destructed from youngest to oldest, and so are the `obj`'s in `b`.

The destructor for a vector calls the destructor for each object in the vector. But we get another shocker: the elements in a vector are not necessarily destructed from youngest to oldest. We can see this in `v6` and `v5`, because their elements have distinct values. In fact, the elements are not even destructed in order of descending subscripts. They are always destructed from front to back because internally, the destructor for a vector may call ‘‘algorithms’’ whose arguments are merely ‘‘forward’’ iterators (pp. 839−840).

When we write the above line 40, the computer behaves as if we had written line 46:

```
46      operator<<(operator<<(cout, it.operator*()), "\n");
```

(Line 46 assumes that `it` is an object. If `it` is merely a pointer, then line 46 would merely have `*it` in place of the `it.operator*()`.) The `it` in line 39 is an iterator for looping through a vector that holds `obj`’s, so the expression `*it` in lines 40 and 46 is of data type `obj`. This causes the right `operator<<` in line 46 to be the one whose second argument is an `obj`. This function is a friend of class `obj`; we saw its definition in line 18 of `obj.h` in p. 180.

I also want to demonstrate how to call a member function of an object retrieved from a vector with an iterator. Unfortunately, our class `obj` has only the member function `print`, rendered obsolete by the friend `operator<<`. But we’ll call it anyway, just to demonstrate the syntax. Change line 40 to lines 47−48.

```
47          (*it).print();               //Don't write this: line 49 is simpler.
48          cout << "\n";
```

Line 47 calls the `print` member function of the anonymous `obj *it`. It must first retrieve the `obj` from the vector before it can call the `print` member function of the `obj`. That’s why the `*` must be applied to the `it` before the `.print()` is applied to the `*it`. To make this happen even though the `*` has lower precedence than the dot, line 47 needs the parentheses around the expression `*it`. Without them, the computer would attempt to apply the `.print()` to the iterator first. That would be totally wrong: we want to call the `print` member function of an `obj`, not of the iterator.

But line 47 was for pedagogical purposes only. Change it to 49. In C and C++, the single operator `->` can do the work of a `*` followed by a dot. And now that there is only one operator, we no longer need the parentheses around the `*it` in line 47.

```
49          it->print();
50          cout << "\n";
```

`default construct 0`	*Line 11 constructs* `v2`.
`copy construct 0`	*Line 11 constructs* `v2`.
`copy construct 0`	*Line 11 constructs* `v2`.
`copy construct 0`	*Line 11 constructs* `v2`.
`destruct 0`	*Line 11 constructs* `v2`.
`construct 10`	*Line 14 constructs* `v3`.
`copy construct 10`	*Line 14 constructs* `v3`.
`copy construct 10`	*Line 14 constructs* `v3`.
`copy construct 10`	*Line 14 constructs* `v3`.
`destruct 10`	*Line 14 constructs* `v3`.
`construct 20`	*Line 17 constructs* `v4`.
`copy construct 20`	*Line 17 constructs* `v4`.
`copy construct 20`	*Line 17 constructs* `v4`.
`copy construct 20`	*Line 17 constructs* `v4`.
`destruct 20`	*line 17 constructs* `v4`*: destruct the first* `obj`.
`construct 30`	*Lines 20−26 construct* `v5`.
`construct 40`	*Lines 20−26 construct* `v5`.
`construct 50`	*Lines 20−26 construct* `v5`.
`construct 60`	*Lines 29−33 construct the array* `b`.
`construct 70`	*Lines 29−33 construct the array* `b`.
`construct 80`	*Lines 29−33 construct the array* `b`.
`copy construct 60`	*Line 36 constructs* `v6`.
`copy construct 70`	*Line 36 constructs* `v6`.
`copy construct 80`	*Line 36 constructs* `v6`.
`60`	*Lines 39−41*
`70`	*Lines 39−41*
`80`	*Lines 39−41*
`destruct 60`	*Line 44 destructs the three* `obj`*'s in* `v6` *in an unexpected order.*
`destruct 70`	*Line 44 destructs the three* `obj`*'s in* `v6` *in an unexpected order.*
`destruct 80`	*Line 44 destructs the three* `obj`*'s in* `v6` *in an unexpected order.*
`destruct 80`	*Line 44 destructs the three* `obj`*'s in the array* `b` *in the expected order.*
`destruct 70`	*Line 44 destructs the three* `obj`*'s in the array* `b` *in the expected order.*
`destruct 60`	*Line 44 destructs the three* `obj`*'s in the array* `b` *in the expected order.*
`destruct 30`	*Line 44 destructs the three* `obj`*'s in* `v5` *in an unexpected order.*
`destruct 40`	*Line 44 destructs the three* `obj`*'s in* `v5` *in an unexpected order.*
`destruct 50`	*Line 44 destructs the three* `obj`*'s in* `v5` *in an unexpected order.*
`destruct 20`	*Line 44 destructs* `v4`*; we can't tell in what order.*
`destruct 20`	*Line 44 destructs* `v4`.
`destruct 20`	*Line 44 destructs* `v4`.
`destruct 10`	*Line 44 destructs* `v3`.
`destruct 10`	*Line 44 destructs* `v3`.
`destruct 10`	*Line 44 destructs* `v3`.
`destruct 0`	*Line 44 destructs* `v2`.
`destruct 0`	*Line 44 destructs* `v2`.
`destruct 0`	*Line 44 destructs* `v2`*; then* `v1` *is destructed silently.*

Append an object to a vector of objects

The following program appears to construct and destruct only one `obj`, in lines 10 and 17. The output shows, however, that it actually constructs and destructs two. The underlined lines of output betray the presence of the second `obj`, constructed when the argument `ob` in line 11 is passed by value.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include "obj.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     vector<obj> v;
10     obj ob = 10;
11     v.push_back(ob);
12
13     for (vector<obj>::const_iterator it = v.begin(); it != v.end(); ++it) {
14         cout << *it << "\n";
15     }
16
17     return EXIT_SUCCESS; //Destruct ob and v, in that order.
18 }
```

`construct 10`	*Line 10 constructs* `ob`.
`copy construct 10`	*line 11*
`10`	*line 14*
`destruct 10`	*Line 17 destructs* `ob`.
`destruct 10`	*Line 17 destructs the* `obj` *within* `v`, *and then destructs* `v`.

I'm not telling you not to `push_back` onto a vector of objects. But you must understand the price to be paid: every object that you `push_back` into the vector will be duplicated. Is there a way to avoid this?

Avoid the unwanted copying

To avoid making an unwanted copy of each object inserted into a vector, let the vector be a vector of pointers to objects in line 10. The `push_back` function of this vector takes a pointer to an `obj` (line 11).

As usual, the destructor for a vector calls the destructor for each item in the vector. But the items in this vector are merely pointers, and a pointer has no destructor. (Or we can pretend that a pointer has a destructor that does nothing.) The destructor for the vector `v` will therefore not call the destructor for the object `ob`.

We construct `ob` before `v` to ensure that line 18 will destruct `ob` after `v`. Were `ob` destructed first, `v` would momentarily be left holding a pointer to the place where `ob` used to be. This is harmless, since `v` is destructed in the next moment. But it is potentially dangerous for a pointer to outlive the variable to which it points.

A vector can hold pointers, but not references. See p. 80.

The `it` in line 13 is an iterator for looping through a vector that holds pointers to `obj`'s. The expression `*it` in lines 14 and 15 is therefore of data type "pointer to `obj`", and the `**it` is of type `obj`.

I also want to demonstrate how to call a member function of one of these objects. Unfortunately, our class `obj` has only the member function `print`, rendered obsolete by the friend `operator<<`. But line

15 shows how to call it anyway, just to demonstrate the syntax.

The `(**it).print()` calls the `print` member function of the anonymous object `**it`. The first (i.e., rightmost) `*` retrieves a pointer to the `obj` from the vector. The second (i.e., leftmost) `*` dereferences the pointer to get the `obj` itself. Finally, the dot calls the `print` member function of the `obj`. To apply the two `*`'s to the `it` before the dot is applied to the `**it`, line 15 needs the parentheses around the expression `**it`. Without them, the computer would attempt to apply the dot to the iterator. That would be totally wrong: we want to call the `print` member function of the `obj`, not of the iterator. The iterator has no `print`.

The `(*it)->print()` in line 15 would do the same thing. In C and C++, the single operator `->` can do the work of a `*` followed by a dot. Does this make the code easier to read?

When we have inheritance, we will see another reason why vectors and other containers usually contain pointers to objects, rather than the objects themselves. See p. 487.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include "obj.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     obj ob = 10;
10     vector<obj *> v;
11     v.push_back(&ob);
12
13     for (vector<obj *>::const_iterator it = v.begin(); it != v.end(); ++it){
14         cout << "The obj at address " << *it << " is " << **it << ".\n";
15         //Can also say (**it).print() or (*it)->print()
16     }
17
18     //Destruct v and ob, in that order.
19     //Do not allow v to even momentarily hold a pointer to a destructed obj.
20     return EXIT_SUCCESS;
21 }
```

`construct 10`	*Line 9 constructs* `ob`.
`The obj at address 0xffbff0b0 is 10.`	*Lines 13−16*
`destruct 10`	*Line 18 destructs* `v` *and then* `ob`.

▼ Homework 4.3.3a: define an `operator-` to measure the distance between two life objects

Define an `operator<=` that would return `true` if the first `life` object would evolve into the second one, and an `operator-` that would tell us how many generations it would take. Since our playing board is of finite size, we don't have to worry about these functions going into an infinite loop. They should ignore the `g` data member of class `life`.

```
    if (g1 <= g2) {                  //if (operator<=(g1, g2)) {
        cout << "g1 will evolve into g2 after "
            << g2 - g1               //<< operator-(g2, g1)
            << " generations.\n";
    }
```

The `operator-` in the above line 3 will create a copy of `g1` and move the copy forward one generation at a time until one of the following happens, whichever comes first.

(1) The copy contains the same picture as `g2`.

(2) The copy contains the same picture as in an earlier generation;

(3) an `int` can't count any higher.

In the latter two cases, `operator-` should return `INT_MAX` to show that `g1` will never evolve into `g2`, at least not in any number of generations that can be counted with an `int`. `INT_MAX` is a macro for the largest `int` value, defined in the header file `<climits>`.

`operator-` will `push_back` each generation of `g1` into a local `vector<life>`. For the present, we will assume (i.e., pray) that each `push_back` will be successful; on p. 628 we will check if it ‘‘throws an exception’’. For the present, we will search the `vector` with a `for` loop; on p. 861 we will search it with the `find` algorithm.

`operator<=` can do almost all of its work by calling `operator-`. Also define an `operator<`, returning `true` if the objects are unequal and the left one can evolve into the right one. Note that for a single `life` object we can easily have `a < a`. For two `life` objects, we can have `a < b` and `b < a`. For three, we can have `a < b` and `b < c` without also having `a < c`; for example, the total distance from `a` to `c` may add up to more than `INT_MAX`. These nonstandard behaviors will make our `operator<` ineligible for most of the expected applications of an `<` in the Standard Template Library (pp. 776−777). Maybe we should have named it `can_evolve_into` instead of `operator<`.

▲

4.3.4 Class `list`

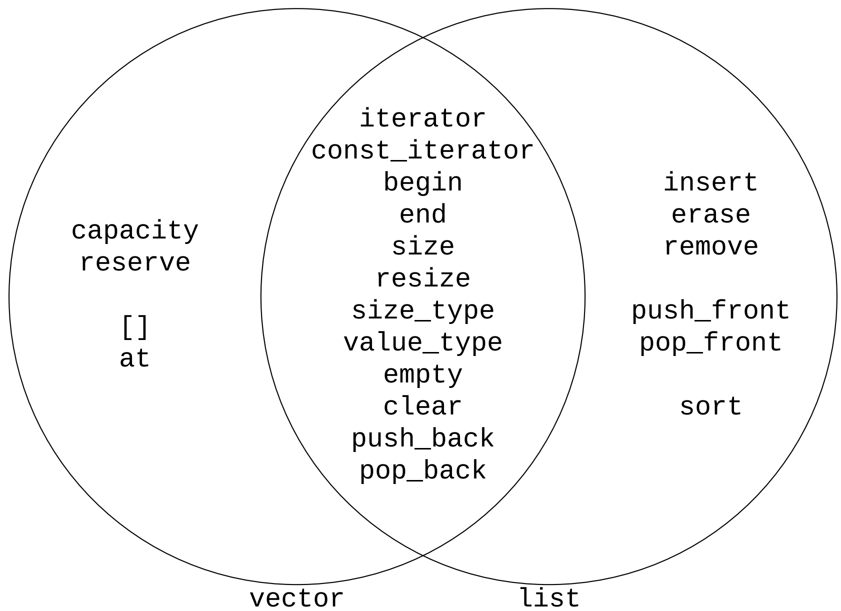
vector vs. list

A `vector` is like a CD and a `list` is like a tape. We can jump around in a `vector` but we must wind and rewind a `list`.

Use a `vector` to access the elements in a non-consecutive order, i.e., for random access. Use a `list` to perform many insertions and deletions quickly. Although class `vector` does have the member functions `insert` and `erase`, they're slower than the ones of class `list`.

There's another problem with the `insert` and `erase` member functions of class `vector`. All the elements after the insertion or deletion point get moved to new locations. This *invalidates* any iterator that refers to one of these elements. It's even worse when the capacity of a `vector` is changed: all the elements may be moved, and all the iterators are invalidated.

The words in the circles are names of public members of classes `vector` and `list`. Most of them are members that are member functions; the `[]` is a shorthand for `operator[]`. Four of them are members that are data types (like the `hillary_t` member of class `clinton`): `iterator`, `const_iterator`, `size_type`, and `value_type`.



Construct a list of int's

The constructors for class `list` take the same arguments as those for class `vector`.

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

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

int main()
{
    list<int> li1;           //born empty, but we can insert int's later
    list<int> li2(3);        //born containing 0, 0, 0
    list<int> li3(3, 10);    //born containing 10, 10, 10

    const int a[] = {10, 20, 30};
    const size_t n = sizeof a / sizeof a[0];
    list<int> li4(a, a + n); //born containing 10, 20, 30

    vector<int> v(a, a + n); //born containing 10, 20, 30
    list<int> li5(v.begin(), v.end());   //born containing 10, 20, 30

    list<int> li6 = li5;     //born containing 10, 20, 30: copy constructor

    for (list<int>::const_iterator it = li6.begin(); it != li6.end(); ++it) {
        cout << *it << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
10
20
30
```

Three ways to insert an element into a list

We must construct an iterator before we can call the `insert` in line 15. It must be a plain `iterator`, not a `const_iterator`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     list<int> li;        //born empty
 9
10     li.push_back(30);    //Class vector has the same push_back function,
11     li.push_front(10);   //but not a push_front function.
12
13     list<int>::iterator it = li.begin();   //it refers to the 10.
14     ++it;                                  //Now it refers to the 30.
15     li.insert(it, 20);                     //Insert a 20 before the 30.
16
17     for (list<int>::const_iterator it = li.begin(); it != li.end(); ++it) {
18         cout << *it << "\n";
19     }
20
21     return EXIT_SUCCESS;
22 }
```

A more complicated way to do line 11 would be

```
23     li.insert(li.begin(), 10);
```

We can combine lines 14−15 to

```
24     li.insert(++it, 20);
```

```
10
20
30
```

▼ Homework 4.3.4a: the increment of death

The `erase` in line 14 removes the element to which the iterator refers. There are no bugs up to and including line 14.

But after of the `erase`, the `++` in line 15 will behave unpredictably. We cannot increment a list iterator that refers to an element that has been erased. This is because each list element contains a pointer to the next element, which the `operator++` function uses to find the next element. If an element has been erased, the pointer inside it will also be erased, cutting the ground out from under any iterator that referred to the element. Its `operator++` will not be able to navigate to the next element.

The `++` in line 15 will therefore leave the iterator referring to an unpredictable location. Line 16 will then blow up—if you are lucky. Otherwise, it will output the wrong answer. How lucky are you?

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const int a[] = {10, 20, 30};
 9     const size_t n = sizeof a / sizeof a[0];
10     list<int> li(a, a + n);
11     list<int>::iterator it = li.begin();
12
13     cout << "The first element of the list is " << *it << ".\n";
14     li.erase(it);
15     ++it;
16     cout << "The second element of the list is " << *it << ".\n";
17
18     return EXIT_SUCCESS;
19 }
```

```
The first element of the list is 10.
The second element of the list is 0.    Should have been 20.
```

▲

Continue looping after an erasure

The `erase` function in line 19 removes one element each time it is called. The `remove` function in line 25 removes every element that is equal to `30`. The `clear` function in line 33 removes every element, period. If the elements have destructors (which these don't), all three functions will call the destructor for each element removed from the list.

The argument of `erase` is an *iterator* referring to the element to be removed; the argument of `remove` is the *value* of each element to be removed. `remove` contains a searching loop which applies the operator `==` to each element in the list. Before calling `remove` for a list of objects, we must therefore write an `operator==` function for that class of object.

I'm sorry that the `++i` is not at the traditional place in the `for` loop, at the end of line 17. But as we just saw, we cannot increment a list iterator referring to an element that has been erased. Fortunately, the `erase` function returns an iterator referring to the element after the one that was erased. (If there is no element after the one that was erased, `erase` will return the same value as the `end` function.) Unfortunately, the `++i` had to be buried in an `else`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 using namespace std;
 5
 6 int main()
 7 {
```

```
 8     const int a[] = {30, 20, 30, 10, 20, 10};
 9     const size_t n = sizeof a / sizeof a[0];
10     list<int> li(a, a + n);
11
12     for (list<int>::const_iterator it = li.begin(); it != li.end(); ++it) {
13         cout << *it << "\n";
14     }
15     cout << "\n";
16
17     for (list<int>::iterator it = li.begin(); it != li.end();) {
18         if (*it == 20) {
19             it = li.erase(it);           //Get rid of one 20.
20         } else {
21             ++it;
22         }
23     }
24
25     li.remove(30);                         //Get rid of every 30.
26
27     for (list<int>::const_iterator it = li.begin(); it != li.end(); ++it) {
28         cout << *it << "\n";
29     }
30     cout << "\n";
31
32     cout << "size == " << li.size() << "\n";
33     li.clear();
34     cout << "size == " << li.size() << "\n";
35     return EXIT_SUCCESS;
36 }
```

```
30              lines 12−14
20
30
10
20
10

10              lines 27−29
10

size == 2       line 32
size == 0       line 34
```

A list of objects

On my platform, line 11 constructs and destructs almost twice as many objects as the analogous line 26 of `vector_obj.C` on p. 437. What does your platform do? Is there documentation?

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

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <list>
4 #include "obj.h"
```

```
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const int a[] = {20, 30, 40};
10     const size_t n = sizeof a / sizeof a[0];
11     list<obj> li(a, a + n);
12
13     obj o1 = 10;
14     li.push_front(o1);
15
16     obj o2 = 50;
17     li.push_back(o2);
18
19     for (list<obj>::const_iterator it = li.begin(); it != li.end(); ++it) {
20         cout << *it << "\n";   //or (*it).print() or it->print()
21     }
22
23     for (list<obj>::iterator it = li.begin(); it != li.end();) {
24         if (*it == 20) {         //if ((*it).operator int() == 20) {
25             it = li.erase(it);   //Calls the object’s destructor.
26         } else {
27             ++it;
28         }
29     }
30
31     for (list<obj>::const_iterator it = li.begin(); it != li.end(); ++it) {
32         cout << *it << "\n";
33     }
34
35     return EXIT_SUCCESS;
36 }
```

Lines 14 and 17 construct copies of the pushed object; the evidence is underlined.

`construct 20`	*line 11*
`copy construct 20`	*line 11*
`destruct 20`	*line 11*
`construct 30`	*line 11*
`copy construct 30`	*line 11*
`destruct 30`	*line 11*
`construct 40`	*line 11*
`copy construct 40`	*line 11*
`destruct 40`	*line 11*
`construct 10`	*line 13*
`copy construct 10`	*line 14*
`construct 50`	*line 16*
`copy construct 50`	*line 17*
`10`	*lines 19−21*
`20`	*lines 19−21*
`30`	*lines 19−21*
`40`	*lines 19−21*
`50`	*lines 19−21*
`destruct 20`	*line 25*
`10`	*lines 31−33*
`30`	*lines 31−33*
`40`	*lines 31−33*
`50`	*lines 31−33*
`destruct 50`	*Line 35 destructs* `o2`.
`destruct 10`	*Line 35 destructs* `o1`.
`destruct 10`	*Line 35 destructs* `li`.
`destruct 30`	*Line 35 destructs* `li`.
`destruct 40`	*Line 35 destructs* `li`.
`destruct 50`	*Line 35 destructs* `li`.

A list of pointers to objects

The `erase` member function of a `list` will call the destructor for the element erased from the `list`. For example, the above line 25 called the destructor for the second object in the `list`. But each item in the following list is merely a pointer, and a pointer has no destructor. (Or we can pretend that a pointer has a destructor which does nothing.) Therefore the `erase` in the following line 23 calls no destructor, so the two `obj`'s survive to line 33.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 #include "obj.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     obj o1 = 10;
10     obj o2 = 20;
11
12     list<obj *> li;
13     li.push_front(&o1);
14     li.push_back(&o2);
15
```

```
16     for (list<obj *>::const_iterator it = li.begin(); it != li.end(); ++it) {
17         cout << "The obj at address " << *it << " is " << **it << ".\n";
18         //Can also say (**it).print() or (*it)->print()
19     }
20
21     for (list<obj *>::iterator it = li.begin(); it != li.end();) {
22         if (**it == 20) {            //if ((**it).operator int() == 20) {
23             it = li.erase(it);
24         } else {
25             ++it;
26         }
27     }
28
29     for (list<obj *>::const_iterator it = li.begin(); it != li.end(); ++it) {
30         cout << **it << "\n";
31     }
32
33     return EXIT_SUCCESS;
34 }
```

`construct 10`	*line 9*
`construct 20`	*line 10*
`The obj at address 0xffbff148 is 10.`	*lines 16−19*
`The obj at address 0xffbff144 is 20.`	*lines 16−19*
`10`	*lines 29−31*
`destruct 20`	*line 33 destructs* `v` *and then* `o2`.
`destruct 10`	*line 33 destructs* `o1`.

Sorting a vector vs. sorting a list

The functions in the STL are called *algorithms.* Most of their arguments are pairs of iterators. To sort a vector, for example, pass its beginning and end to the `sort` algorithm.

```
 1 #include <vector>       //for vector
 2 #include <algorithm>    //for sort
 3
 4     vector<int> v(argument(s) for constructor);
 5     sort(v.begin(), v.end());
```

But not every pair of iterators can be given to the `sort` algorithm. The arguments of `sort` must be *random access* iterators: ones to which we can add a large number (greater than 1) in a single operation. For example, line 7 demonstrates that a vector iterator is random access:

```
 6     vector<int>::iterator it = v.begin();
 7     it += 3;                        //okay: means it = it + 3
```

The `sort` algorithm adds large numbers to the pair of iterators that it receives as arguments, so they must be random access iterators.

On the other hand, lines 9 and 10 demonstrate that a `list` iterator is not random access. The best we can do is to increment it in lines 12−14:

```
 8     list<int>::iterator it = li.begin();
 9     it += 3;                  //Won't compile.
10     it += 1;                  //Even this won't compile.
11
12     ++it;                     //This is how we have to move it forward.
```

```
13     ++it;
14     ++it;
```

Therefore we cannot give a pair of `list` iterators to the `sort` algorithm. Instead, we'll have to call the `sort` member function in line 18, which eventually gets the job done by repeated increments instead of by adding large numbers. This `sort` is slower than the `sort` algorithm, but it's the best we can do.

```
15 #include <list>
16
17     list<int> li(argument(s) for constructor);
18     li.sort();
```

It might be worthwhile to copy a long list into a vector for sorting, and then copy it back again:

```
1 #include <vector>
2 #include <list>
3 #include <algorithm>
4
5     list<int> li(argument(s) for constructor);
6
7     vector<int> v(li.begin(), li.end());
8     sort(v.begin(), v.end());
9     copy(v.begin(), v.end(), li.begin());
```

4.3.5 Data types for pointer and iterator arithmetic

	array	*STL container*
unsigned	`size_t`	`size_type`
signed	`ptrdiff_t`	`difference_type`

The data type `size_t` is used for the number of elements in an array, or the number of bytes in a variable or dynamically allocated block of memory. It is the data type of the value of the `sizeof` operator, the argument of the C function `malloc`, and the return value of the C function `strlen`. See the following line 9. We also use `size_t` for an array subscript.

Similarly, a data type `size_type` is used for the number of elements in an STL container. For example, a `size_type` is the return type of the `size` member function of every container in the STL. In class `vector`, `size_type` is also as the return type of the member function `capacity`, and the argument of the member functions `resize`, `reserve`, `operator[]`, and `at`. See line 17.

We can subtract any two pointers that point to elements in the same array, yielding a result of data type `ptrdiff_t` (line 13). A `ptrdiff_t` is also what we add to a pointer to make the pointer point to a neighboring array element (line 14). `ptrdiff_t` is signed (it is another name for `int` or `long`), `size_t` is unsigned (it is another name for `unsigned` or `long unsigned`), but they are the same size.

Similarly, we can often subtract any two iterators that refer to elements in the same STL container, yielding a result of data type `difference_type` (line 21). A `difference_type` is also what we add to an iterator to make the iterator refer to a neighboring item (line 22). Iterators that permit these operations are called "random access" (p. 841). Pointers and `vector` iterators are random access, but a `list` iterator is not. An attempt to add a `list<nt>::difference_type` to a `list<int>::iterator` would not compile.

The only difference between a `difference_type` and a `size_type` is that `difference_type` is signed, while `size_type` is unsigned.

`size_t` and `ptrdiff_t` are typedefs in the C Standard Library, so they have no last name. Nothing in C has a last name, so there can be only one data type named `size_t` and only one named `ptrdiff_t`.

But size_type and difference_type are the names of many typedefs in the C++ Standard Library, one for each type of container. What makes this possible is that each one has a different last name. For example, a vector<int>::size_type holds the number of elements in a vector<int>, and a vector<char>::size_type holds the number of elements in a vector<char>. They might have to be different data types because a vector<char>::size_type might have to hold a much larger number than a vector<int>::size_type.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     int a[] = {10, 20, 30, 40, 50};
 9     size_t n = sizeof a / sizeof a[0];          //n is 5
10
11     int *p1 = a;                                //point to the 10
12     int *p2 = a + 4;                            //point to the 50
13     ptrdiff_t d1 = p2 - p1;                     //d1 is 4
14     p1 += d1;                                   //Now p1 points to the 50.
15
16     vector<int> v(a, a + n);
17     vector<int>::size_type s = v.size();        //s is 5
18
19     vector<int>::iterator it1 = v.begin();      //refer to the 10
20     vector<int>::iterator it2 = v.begin() + 4;  //refer to the 50
21     vector<int>::difference_type d2 = it2 - it1; //d2 is 4
22     it1 += d2;                                  //Now it1 refers to the 50.
23
24     return EXIT_SUCCESS;
25 }
```

4.3.6 Class string

Class string

A C program holds a string in an array of characters; a C++ program holds a string in an object of class string.

Lines 8 and 16 show two constructors for class string. Line 17 inputs a one-word string, expanding the string object to hold it. To do the job of line 17 in C, without a string object, we would need all the code in the following C program.

The C++ Standard Library has three header files with similar names:

<string>	*declaration for the C++ class* string
<string.h>	*declarations belonging to no namespace for the C string functions* strlen, strcat, *etc.*
<cstring>	*declarations belonging to namespace* std *for the C string functions* strlen, strcat, *etc.*

A group of functions and variables sharing the same last name is called a namespace. The version of the string functions declared in string.h belong to no namespace; that in cstring belong to the standard namespace std. The objects cin and cout also belong to namespace std; see p. 20. Ditto for the C++ Standard Library classes vector, list, and stack.

Instead of the `str-` functions `strlen`, `strcat`, etc., in the C Standard Library, we now call the member functions and friends of a `string` object. See lines 10, 19, 23, 24, and 32. There are member functions for searching for substrings and individual characters, forwards from the start or backwards from the end. As in a `vector` or `list`, there are also member functions to `insert` and `erase`.

Occasionally we need to load the characters of a `string` into consecutive memory addresses, add a `'\0'` after the last one, and get a pointer to the first one. For example, we may need to pass the characters to an older function whose argument is a `const char *`. (Two such functions are the constructors for classes `ofstream` and `locale`.) Lines 46−47 show how not to get this pointer. The pointer must be read-only as in line 48: it cannot be used to change the characters in the `string`.

Unlike a C array of characters, a C++ `string` object has no terminating `'\0'`. This means that a `string` object can hold the character `'\0'` (line 50), making it possible for a `string` object to hold arbitrary binary data. Of course, we would never want to call the `c_str` member function of a `string` that contained a `'\0'`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>   //for class string
 4 using namespace std;
 5
 6 int main()
 7 {
 8     string s = "Hello there";   //one-argument constructor
 9
10     cout << "s.size() == " << s.size() << "\n"           //instead of strlen
11         << "The 1st character is '" << s[0] << "'.\n"    //s.operator[](0)
12         << "The next 3 characters are \"" << s.substr(1, 3) << "\".\n"
13         << "The last character is '" << s[s.size() - 1] << "'.\n\n";
14
15     cout << "Please type your name and press RETURN: ";
16     string word;   //No-argument constructor puts null string into object.
17     cin >> word;   //Input 1 word like scanf(%s; string expands to hold it.
18
19     string line = s + ", " + word + "!  ";
20     line += "How are you?"; //instead of strcat: line.operator+=("How RU?");
21     cout << line << "\n\n";
22
23     if (s < line) {     //instead of strcmp: if (operator<(s, line)) {
24         line = s;       //instead of strcpy: line.operator=(s);
25     }
26
27     for (string::const_iterator it = s.begin(); it != s.end(); ++it) {
28         cout << *it;
29     }
30     cout << "\n\n";
31
32     string::size_type i = s.find('l');         //instead of strchr
33     if (i == string::npos) {                   //"no position"
34         cout << "The string \"" << s << "\" does not contain 'l'.\n";
35     } else {
36         cout << "Found the first 'l' at position " << i << ".\n";
37     }
38
```

```
39      i = s.find("lo");                            //instead of strstr
40      if (i == string::npos) {                     //"no position"
41          cout << "The string \"" << s << "\" does not contain \"lo\".\n";
42      } else {
43          cout << "Found the first \"lo\" at position " << i << ".\n";
44      }
45
46      //char *p = s;                //won't compile
47      //char *p = s.c_str();        //won't compile
48      const char *p = s.c_str();    //will compile: pointer must be read-only
49
50      s[0] = '\0';
51      return EXIT_SUCCESS;
52  }
```

The above line 19 behaves as if we had written the nested function calls

```
53      string line = operator+(operator+(operator+(s, ", "), word), "!  ");
```

```
s.size() == 11                     no terminating '\0' at end of Hello
The 1st character is 'H'.
The next 3 characters are "ell".
The last character is 'e'.

Please type your name and press RETURN: Mark
Hello there, Mark!  How are you?   line 21

Hello there                        lines 27−30

Found the first 'l' at position 2.  lines 32−37
Found the first "lo" at position 3. lines 39−44
```

To do the job of the above lines 16−17, a C program would need a loop with `malloc` and `realloc`.

The first time we arrive at line 10, `malloc` allocates a block of one byte. Even if the user never inputs any non-whitespace characters, we will still need one byte to hold the terminating `'\0'`. Line 26 places each incoming character at the end of the block. The `realloc` in line 10 then makes the block one byte bigger, because even if the user never inputs any more non-whitespace characters, we will still need one more byte to hold the terminating `'\0'`.

Line 22 unreads the whitespace character so that line 35 can read it again. For the cast in line 21, see pp. 63−64.

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

```
 1  #include <stdio.h>    /* for getchar, ungetc, stdin, EOF */
 2  #include <stdlib.h>   /* for malloc, realloc, free, size_t */
 3  #include <ctype.h>    /* for isspace */
 4
 5  int main(int argc, char **argv)
 6  {
 7      size_t n;
 8      char *p;
 9
10      for (p = malloc(n = 1);; p = realloc(p, ++n)) {
```

```
        int c;
        if (p == NULL) {
            fprintf(stderr, "%s: out of store\n", argv[0]);
            return EXIT_FAILURE;
        }

        if ((c = getchar()) == EOF) {
            break;
        }

        if (isspace((unsigned char)c)) {
            ungetc(c, stdin);
            break;
        }

        p[n - 1] = c;
    }
    p[n - 1] = '\0';

    printf("The word \"%s\" was terminated by the ", p);
    if (feof(stdin)) {
        printf("end of file.\n");
    } else {
        printf("whitespace character '\\x%02x'.\n",
            (unsigned char)getchar());
    }

    free(p);
    return feof(stdin) && !ferror(stdin) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

```
Mark
The word "Mark" was terminated by the whitespace character '\x0a'.
```

▼ Homework 4.3.6a: let a terminal display a string object

Add a public member function to class `terminal` declared as

```
void put(unsigned x, unsigned y, const string& s) const;
```

This function will simply pass the return value of `c_str` to the `terminal::put` whose third argument is a `const char *`. The function will therefore be short enough to be inline.

`terminal.h` will now have to include the header file `string` and use namespace `std`.

▲

String output

To demonstrate the versatility of our new `operator<<` and `operator>>`, we will write a `date` to three different destinations of output, and read one from three different sources of input. One of these destinations and sources will be a `string` of characters in memory. See Lippman pp. 1108−1112, Stroustrup pp. 640−641.

Here's how to write output to a string in C. The ''string'' is merely the array of characters `a` in line 12; it's up to us to create this array and remember how long it is. The `snprintf` in line 13 writes at most `N` characters (including the terminating `'\0'`) to the array. This demonstrates what string output is good for: pasting together strings, numbers, characters, etc., into one big string for later use.

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

```
#include <stdio.h>     /* C example */
#include <stdlib.h>

#define N 100          /* number of characters in string */

int main(int argc, char **argv)
{
    const char word[] = "size";
    const int i = 38;
    const char c = 'L';

    char a[N];         /* uninitialized variable */
    if (snprintf(a, N, "%s %d%c", word, i, c) < 0) {
        fprintf(stderr, "%s: snprintf failed\n", argv[0]);
        return EXIT_FAILURE;
    }

    printf("The string contains \"%s\".\n", a);
    return EXIT_SUCCESS;
}
```

```
The string contains "size 38L".
```

To write output to a string in C++, we construct the `ostringstream` object in line 12. It's a destination for output (line 13), just like `cout`, but the characters do not go to the standard output. They are written into a string in memory, making it longer and longer. We don't have to create or lengthen the string ourselves: it's all done automatically by the `ostringsteam` object.

To "harvest" the characters stored in the growing string, line 20 calls the `str` member function of the `ostringstream`. It returns a C++ `string` object containing the string of characters.

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

```
 1 #include <iostream>
 2 #include <sstream>   //for ostringstream; includes <string>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     const string word = "size";
 9     const int i = 38;
10     const char c = 'L';
11
12     ostringstream ost;
13     ost << word << " " << i << c;
14
15     if (!ost) {   //if (ost.operator!()) {
16         cerr << argv[0] << ": write to ostringstream failed\n";
17         return EXIT_FAILURE;
18     }
19
20     cout << "The string contains \"" << ost.str() << "\".\n";
```

```
21     return EXIT_SUCCESS;
22 }
```

The above lines 13−15 may be combined to

```
23     if (!(ost << word << " " << i << c)) {
```

```
The string contains "size 38L".
```

String input

Here's how to read input from a string in C. The "string" is merely the `'\0'`-terminated array of characters `a` in line 6. This demonstrates what string input is good for: breaking a big string into substrings, numbers, characters, etc.

The return value of `sscanf`, like the return value of plain old `scanf`, is the number of variables that were assigned new values. In this case, it should be three.

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

```
 1 #include <stdio.h>     /* C example */
 2 #include <stdlib.h>
 3
 4 int main(int argc, char **argv)
 5 {
 6     const char a[] = "size 38L";
 7
 8     char word[100];
 9     int i;
10     char c;
11
12     if (sscanf(a, "%s%d%c", word, &i, &c) != 3) {
13         fprintf(stderr, "%s: sscanf failed\n", argv[0]);
14         return EXIT_FAILURE;
15     }
16
17     printf("word == \"%s\"\n", word);
18     printf("i == %d\n", i);
19     printf("c == '%c'\n", c);
20
21     return EXIT_SUCCESS;
22 }
```

```
word == "size"
i == 38
c == 'L'
```

To read input from a string in C++, we construct the `istringstream` object in line 8. It's a source of input (line 14), just like `cin`, but the characters do not come from the standard input.

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

```
1 #include <iostream>
2 #include <sstream>   //for istringstream
3 #include <cstdlib>
4 using namespace std;
```

```
 5
 6 int main(int argc, char **argv)
 7 {
 8     istringstream ist("size 38L");
 9
10     string word;
11     int i;
12     char c;
13
14     ist >> word >> i >> c;
15     if (!ist) {
16         cerr << argv[0] << ": the istringstream failed\n";
17         return EXIT_FAILURE;
18     }
19
20     cout << "word == \"" << word << "\"\n"
21         << "i == " << i << "\n"
22         << "c == '" << c << "'\n";
23
24     return EXIT_SUCCESS;
25 }
```

The above lines 14−15 may be combined to

```
26     if (!(ist >> word >> i >> c)) {
```

```
word == "size"
i == 38
c == 'L'
```

operator<< and operator>> can take any destination or source

The original `print` member function of class `date` was hardwired to send output to only one destination: the standard output `cout`. See lines 99−107 on pp. 116−117. Our new `operator<<` friend of class `date` can send output to any destination. We demonstrate three of them: the standard output (line 11), an output file (line 19), and a string of characters (line 26).

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

```
 1 #include <iostream>
 2 #include <fstream>   //for ofstream
 3 #include <sstream>
 4 #include <cstdlib>
 5 #include "date.h"
 6 using namespace std;
 7
 8 int main(int argc, char **argv)
 9 {
10     const date d;
11     cout << d << "\n";    //operator<<(cout, d).operator<<("\n");
12
13     ofstream ofstr("outfile");
14     if (!ofstr) {
15         cerr << argv[0] << ": couldn't open outfile\n";
16         return EXIT_FAILURE;
```

```
    }

    ofstr << d << "\n";   //operator<<(ofstr, d).operator<<("\n");
    if (!ofstr) {
        cerr << argv[0] << ": couldn’t write to outfile\n";
        return EXIT_FAILURE;
    }

    ostringstream os;
    os << d;              //operator<<(os, d);
    if (!os) {
        cerr << argv[0] << ": couldn’t write to string\n";
        return EXIT_FAILURE;
    }

    cout << "The string contains \"" << os.str() << "\".\n";

    return EXIT_SUCCESS;
}
```

The standard output is

```
4/8/2014
The string contains "4/8/2014".
```

The file `outfile` contains

```
4/8/2014
```

Similarly, our `operator>>` friend of class `date` can read a `date` from any source. We demonstrate three of them: the standard input (line 13), an input file (line 26), and a string of characters (line 35).

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

```
 1 #include <iostream>
 2 #include <fstream>   //for ifstream
 3 #include <cstdlib>
 4 #include <sstream>
 5 #include "date.h"
 6 using namespace std;
 7
 8 int main(int argc, char **argv)
 9 {
10     date d;
11
12     cout << "Please type a date.\n";
13     cin >> d;     //operator>>(cin, d);
14     if (!cin) {
15         cerr << argv[0] << ": couldn’t read date from standard input\n";
16         return EXIT_FAILURE;
17     }
18     cout << "Read " << d << " from standard input.\n";
19
20     ifstream ifstr("infile");
21     if (!ifstr) {
```

```
        cerr << argv[0] << ": couldn’t open infile\n";
        return EXIT_FAILURE;
    }

    ifstr >> d;   //operator>>(ifstr, d);
    if (!ifstr) {
        cerr << argv[0] << ": couldn’t read date from infile\n";
        return EXIT_FAILURE;
    }
    cout << "Read " << d << " from infile.\n";

    istringstream is("12/31/2014");

    is >> d;      //operator>>(is, d);
    if (!is) {
        cerr << argv[0] << ": couldn’t read date from string\n";
        return EXIT_FAILURE;
    }
    cout << "Read " << d << " from string.\n";

    return EXIT_SUCCESS;
}
```

Given an `infile` containing

```
4/8/2014
```

the program’s output will be

```
Please type a date.
1/1/2014        The user types this line.
Read 1/1/2014 from standard input.
Read 4/8/2014 from infile.
Read 12/31/2014 from string.
```

Convert an object to a string

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

```
#ifndef DATEH
#define DATEH
#include <sstream>   //for ostringstream
#include <ctime>
using namespace std;

class date {
    int year;
    int month;
    int day;
public:
    date(int initial_month, int initial_day, int initial_year)
        : year(initial_year), month(initial_month), day(initial_day) {}

    date() {
        const time_t t = time(0);
```

```
17         const tm *const p = localtime(&t);
18
19         year = p->tm_year + 1900;
20         month = p->tm_mon + 1;
21         day = p->tm_mday;
22     }
23
24     friend ostream& operator<<(ostream& ostr, const date& d) {
25         return ostr << d.month << "/" << d.day << "/" << d.year;
26     }
27
28     operator string() const {
29         ostringstream ost;
30         ost << *this;   //calls line 24: operator<<(ost, *this);
31         return ost.str();
32     }
33 };
34 #endif
```

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

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

int main()
{
    const date d;
    const string s = d;   //string s = d.operator string();

    cout << "\"" << s << "\"\n"
        << "\"" << static_cast<string>(d) << "\"\n";

    return EXIT_SUCCESS;
}
```

```
"4/8/2014"
"4/8/2014"
```

▼ Homework 4.3.6b: fix the operator<< friend of class date

There's a bug in the `operator<<` we wrote for class `date` on p. 338. The following line 11 tries to print a `date` in a field of width 12. Unfortunately, it prints only the month number in the field.

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

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cstdlib>
4 #include "date.h"
5 using namespace std;
6
```

```
int main()
{
    date d(date::december, 31, 2014);
    cout << "123456789012\n"
        << setw(12) << d << "\n";

    return EXIT_SUCCESS;
}
```

```
123456789012
          12/31/2014
```

Fix the `operator<<` by writing the date to an `ostringstream`. Then get the string from the `ostringstream` and write it to the `ostream` that is the first argument of the `operator<<`.

```
ostream& operator<<(ostream& ost, const date& d)
{
    ostringstream stream;
    stream << d.month << "/" << d.day << "/" << d.year;
    return ost << stream.str();
}
```

```
123456789012
  12/31/2014
```

We will use the same technique at a lower level on p. 1048.

▲

4.3.7 Class `bitset`

A `bitset` is an ‘‘array’’ of bits. It is a template class whose argument is the number of bits in the set.

A `bitset` can be converted to and from a `string` (lines 12−18, 21), and to or from a `long unsigned` (lines 25−26). Line 24 discovers how many bits are in a `long unsigned`. If the `bitset` has more bits than a `long unsigned`, line 25 will put zeroes into the high-order bits of the `bitset`, and line 26 will ‘‘throw an exception’’ if the value of the `bitset` does not fit into a `long unsigned` (p. 622).

Warning: subscripts applied to a `bitset` access the bitset from right to left (line 19). Subscripts applied to the `string` representation access the bitset from left to right (line 22). The `[`square brackets`]` do not perform subscript checking. To get this checking, call the member functions `flip`, `set`, `reset`, and `test`. These functions throw an exception if the subscript is illegal (p. 622).

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <bitset>
 4 #include <string>
 5 #include <limits>  //for numeric_limits
 6 using namespace std;
 7
 8 int main()
 9 {
10     bitset<32> a;    //32 bits of zeroes
```

```
    bitset<32> b(string("00000000111111110000000011111111"));
    bitset<32> c(string( //more legible way to do the same thing
        "00000000"
        "11111111"
        "00000000"
        "11111111"
    ));
    cout << "c == " << c << ", rightmost bit is " << c[0] << ".\n";

    string s = c.to_string<char, char_traits<char>, allocator<char> >();
    cout << "s == " << s << ", leftmost bit is " << s[0] << ".\n";

    if (numeric_limits<unsigned long>::digits <= 32) {   //number of bits
        bitset<32> d = 0xFFFF0000;
        unsigned long ul = d.to_ulong();
        cout << "d == " << d << ", rightmost bit is " << d[0] << ".\n"
            << "ul == " << ul << " == " << hex << ul << dec << "\n";
    }

    a = b & c;  //can do & &= | |= ^ ^= ~ << <<= >> >>= == !=
    cout << "a == " << a << ", rightmost bit is " << a[0] << ".\n";

    a[0].flip();   //flip the rightmost bit
    a.flip(0);     //flip the rightmost bit
    a.flip();      //flip all the bits
    cout << "a == " << a << ", rightmost bit is " << a[0] << ".\n";

    if (a.none()) {
        cout << "None of the bits are on.\n";
    } else if ((~a).none()) {
        cout << "All of the bits are on.\n";
    } else {
        cout << a.count() << " of the bits are on.\n";
    }

    a[0] = true;   //Turn on the rightmost bit.
    a.set(0);      //Turn on the rightmost bit.
    a.set();       //Turn on all the bits.

    a[0] = false;  //Turn off the rightmost bit.
    a.reset(0);    //Turn off the rightmost bit.
    a.reset();     //Turn off all the bits.

    if (a.any()) {
        cerr << "None of the bits should be on after a reset.\n";
    }

    return EXIT_SUCCESS;
}
```

```
c == 00000000111111110000000011111111, rightmost bit is 1.
s == 00000000111111110000000011111111, leftmost bit is 0.
d == 11111111111111110000000000000000, rightmost bit is 0.
ul == 4294901760 == ffff0000
a == 00000000111111110000000011111111, rightmost bit is 1.
a == 11111111000000001111111100000000, rightmost bit is 0.
16 of the bits are on.
```

▼ **Homework 4.3.7a:**

Change the array of `bool` into a `bitset` in the program in pp. 415−419.

▲

4.4 Put it all Together: Aggregation, Dynamic Memory, and Lists

Keep the game going until all the rabbits are dead

The current version of the game stops as soon as *any* `rabbit` is killed. We will make it continue until *all* the `rabbit`'s are killed. We will use three features of C++ that we just covered: aggregation, dynamic memory allocation, and lists.

(1) To make it possible to delete the `rabbit`'s one by one, and someday to let them reproduce, we will change the array of `rabbit`'s to a `list` of `rabbit`'s. More precisely, it will be a `list` of pointers to `rabbit`'s, so we can insert them without duplicating them.

(2) To make it possible to delete the `rabbit`'s in an unpredictable order, we will allocate them dynamically with `new` and `delete`.

(3) The list and the terminal will be data members of a new object called a `game`. In other words, the `game` will be built using aggregation.

Class game

The list will be shared by all the animals. Where should it go? The animals already share a common terminal, which is a local object in the `main` function. Each animal has a pointer to the shared object:

```
int main()
{
    const terminal term('.');       //the object shared by all the animals

class rabbit {
    const terminal *const t;

class wolf {
    const terminal *const t;
```

These pointers are fine as long as there is only one shared object. But the animals will now share two objects, a terminal and a master list of pointers to `rabbit`'s. If they were both local objects in `main`, each animal would need *two* pointers:

```
int main()
{
    const terminal term('.');       //two objects shared by all the animals
    list<rabbit *> master;

class rabbit {
    const terminal *const t;
    list<rabbit *> *const m;
```

```
class wolf {
    const terminal *const t;
    list<rabbit *> *const m;
```

This solution does not scale up: it is unnatural for each animal to have two umbilicals cords leading to two placentas.

Another solution is to let the two shared variables be static data members of class `rabbit`.

```
class rabbit {
    static const terminal term;
    static list<rabbit *> master;
```

Or they could be global variables:

```
const terminal term('.');
list<rabbit *> master;

int main()
{
```

These last two solutions would even let us dispense with the pointer data members of the animals. But they would lock us into having only one terminal and one master list—*ein Volk, ein Reich, ein Führer.* See p. 106. In the future we might want to run more than one game simultaneously, each with its own terminal and master list. Don’t try this yet, though. Each game would need its own terminal, but right now we have only one.

To let each animal get by with only one pointer, we combine (“aggregate”) the terminal and the master list into a single object:

```
class game {                        //showing only the data members for now
    const terminal term;
    list<rabbit *> master;
};
```

Now that there is only one shared object in `main`, each animal will have only one pointer.

```
int main()
{
    game g('.');                    //the object shared by all the animals
```

```
class rabbit {
    game *const g;
```

```
class wolf {
    game *const g;
```

Class `game` is a holder for the objects that are shared by all the animals. Here is its header file. The terminal is constructed before any animal or list of animals (lines 10−11) because the canvas is logically prior to the painting, the plaster to the fresco, the cardboard to the acrylic.

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

```
 1 #ifndef GAMEH
 2 #define GAMEH
 3 #include <list>
 4 #include "terminal.h"
 5 using namespace std;
 6
 7 class rabbit;   //forward declaration
```

```
 8
 9 class game {
10     const terminal term;
11     list<rabbit *> master;
12
13     rabbit *get(unsigned x, unsigned y) const;
14 public:
15     game(char initial_c = '.'): term(initial_c) {}
16     ~game();
17
18     void play();
19
20     friend class rabbit;
21     friend class wolf;
22 };
23 #endif
```

Class `game` will mention the data type `list<rabbit *>` in many places. For your own convenience, insert the following typedef at the above line 10½

```
24     typedef list<rabbit *> master_t;
```

and change every subsequent `list<rabbit *>` to `master_t`.

Forward declaration

The forward declaration in the above line 7 is needed in front of two classes that mention each other. Here is a simpler example. If every `ying` contains a `yang`, and every `yang` contains a `ying`, they would both blow up to infinite size. That's one reason why the following code will not compile:

```
 1 class ying {
 2     yang y;
 3 };
 4
 5 class yang {
 6     ying y;
 7 };
```

But it is quite possible for a `ying` and a `yang` to contain pointers or references to each other:

```
 8 class ying {
 9     yang *y;
10 };
11
12 class yang {
13     ying *y;
14 };
```

The word `yang` makes its first appearance in the above line 9. Before this initial appearance, the computer needs some notification that `yang` is the name of a class. It doesn't need to see the complete definition of the class; it needs only the *forward declaration* in the following line 15. Note that the corresponding line 22 needs no forward declaration for a `ying`, since lines 17−19 have already declared what a `ying` is.

Our classes `game` and `rabbit` correspond to `ying` and `yang`, and need the same forward declaration. The right side of lines 15−19 are in the header file for class `game`; the right side of 21−23 are in the header file for class `rabbit`.

```
15 class yang;                        class rabbit;
```

```
16
17 class ying {
18     yang *y;
19 };
20
21 class yang {
22     ying *y;
23 };
```

```
class game {
    list<rabbit *> master;
};

class rabbit {
    game *const g;
};
```

For other examples of forward declarations, see pp. 295, 684.

Instead of the forward declaration for class `rabbit` in line 7 of the above `game.h`, why not simply include `rabbit.h` at line 4½? After all, including the header file for a class is the normal way of telling the computer that the class exists.

Unfortunately, we can't do that here. We're about to see that `rabbit.h` has to include `game.h` before the definition of class `rabbit`. If the two header files included each other, the program would not compile because of the following vicious circle. `rabbit.h` begins by defining the macro `RABBITH` and including `game.h`. The definition and the include are written at the top of the `rabbit.h` file, before the definition of class `rabbit` has been seen. If `game.h` now tried to include `rabbit.h` at line 4½ of `game.h`, nothing would be included because `RABBITH` has already been defined. The word `rabbit` in lines 11 and 13 of `game.h` would then cause error messages.

▼ Homework 4.4a:
Version 2.0 of the Rabbit Game: list of pointers to dynamically allocated rabbits and a `game` object to hold it

Keep the game going until all the `rabbit`'s have been killed. Create a master list of pointers to dynamically allocated `rabbit`'s and a `game` object to hold it.

(1) Change the `t` data member of class `rabbit` to

```
1     game *const g;
```

The `*const` keeps the `rabbit` tethered to the same game throughout its life. But one of the data members of the `game` (the master list) will be changed by a `rabbit` when the `rabbit` is constructed or destructed. That's why `g` is not a read-only pointer. On the other hand, nothing prevents the `t` data member of class `wolf` from being changed into a read-only pointer.

```
2     const game *const g;
```

(2) The first argument of the constructor for class `wolf` will now be

```
3 wolf::wolf(const game *initial_g, /* etc. */)
4     : g(initial_g), //etc.
5 {
```

The first argument of the constructor for class `rabbit` will be the same, but without the `const`.

Since `rabbit.h` and `wolf.h` now mention class `game` instead of class `terminal`, they will have to include `game.h` instead of `terminal.h`.

(3) Within the bodies of the member functions of classes `rabbit` and `wolf`, every `t->` will become a `g->term.` (with a dot after the `term`). See the following line 6 and its comment for an example.

(4) The last statement of the constructor for class `rabbit` will push the address of the newborn `rabbit` onto the master list. The first statement after the beep in the destructor for class `rabbit` will remove the address of the dying `rabbit` from the master list. Every constructor for class `rabbit` will therefore end with

```
6     g->term.put(x, y, c);          //used to be t->put(x, y, c);
7     g->master.push_back(this);
```

and the destructor for class `rabbit` will contain

```
    g->master.remove(this);
    g->term.put(x, y);              //used to be t->put(x, y);
```

after the beep.

Until now, every class has been barricaded from every other class. The private members of each class have been accessible only to the member functions and friends of that class. But now, our classes `game`, `wolf`, and `rabbit` will interpenetrate. As we have just seen, every member function of classes `rabbit` and `wolf` will mention the private members of class `game`; and at least one member function of class `game` is about to mention the private members of class `rabbit`.

Given their intimacy, it is neither possible nor desirable to keep class `game` barricaded from the other two. We will treat all three classes as one unit, protected from the outside world but not from each other. The wholesale `friend` declarations in lines 20−21 of the above `game.h` make every member function of classes `rabbit` and `wolf` a friend of class `game`.

(5) Until now, we have been using the characters of the screen to detect collisions between two animals. For example, when a `wolf` encounters a lowercase `'r'` in lines 45−46 of `wolf.C` on p. 199, it knows that it has stomped on a `rabbit`:

```
                const bool I_ate_him =
                    g->term.get(newx, newy) != g->term.background();
```

But it doesn't know *which* `rabbit` it has stomped on. You will have to define the following private member function of class `game`. It will loop along the master list, searching for a `rabbit` with the specified coördinates.

```
//Return the address of the rabbit at coordinates (x, y) in this game,
//or zero if no rabbit is there.

rabbit *game::get(unsigned x, unsigned y) const
{
```

The body of `game::get` will have to mention the `x` and `y` private members of class `rabbit`, so `game::get` will have to be a friend of class `rabbit`. Add the following declaration to the definition of class `rabbit` in `rabbit.h`:

```
    friend rabbit *game::get(unsigned x, unsigned y) const;
```

Then change lines 45−46 of `wolf::move` on p. 199 to call `game::get` instead of `terminal::get`:

```
                const bool I_ate_him = g->get(newx, newy) != 0;
```

(6) The constructor for class `game` in line 15 of the above `game.h` will pass its argument to the constructor for the terminal. It will pass no arguments to the constructor for the master list.

(7) The message and pause in lines 30−31 of `main.C` on p. 194 should be moved from the `main` function to the destructor for class `game` in line 16 of the above `game.h`.

(8) The main loop in the `main` function cannot loop through the master list, since the master list is a private member of class `game`. We therefore move the main loop to a member function of class `game`: *code follows the data members.* Move the following code from the `main` function to `game::play`.

```
    //Get the dimensions of the terminal.
    const unsigned xmax =
    const unsigned ymax =

    wolf w(this, xmax / 3, ymax / 2);

    //The array of rabbit's from Homework 2.13a.
```

```
26      rabbit a[] = {
27          rabbit(this, 2 * xmax / 3, 1 * ymax / 4),
28          rabbit(this, 2 * xmax / 3, 2 * ymax / 4),
29          rabbit(this, 2 * xmax / 3, 3 * ymax / 4)
30      };
31
32      for (;; term.wait(250)) {
33          if (!w.move()) {
34              return;                 //Return from game::play; no more goto.
35          }
36
37          for (some type of::iterator it = master.begin();
38              it != master.end(); ++it) {
39
40              if (!(*it)->move()) {   //if (!(**it).move()) {
41                  return;             //Return from game::play.
42              }
43          }
44      }
```

In the above line 37, use your `typedef` for `list<rabbit *>`.

(9) Since `game.C` mentions classes `rabbit` and `wolf`, it will have to include `rabbit.h` and `wolf.h`.

(10) The `main` function will now contain only the following.

```
45      call srand (and also set_new_handler, when we get to ¶ (11));
46
47      game g;
48      g.play();
49
50      return EXIT_SUCCESS;
```

Since the variable `g` is used only in the above line 48, lines 47−48 could even be combined to

```
51      game().play();
```

Now that `main` constructs and destructs only one object, remove the comment at the end of `main` about destructing the rabbit, wolf, and terminal. `main.C` will include `game.h`. Ideally the random number generator should be a data member of class `game`, rather than a global function shared by all the `game`'s. (Don't do this, though.) Does `main.C` still need to include `terminal.h`, `rabbit.h`, and `wolf.h`?

(11) Objects in an array are always destructed in the opposite order from that in which they were constructed. To let the `rabbit`'s be destructed in an unpredictable order, depending on the whims of the player, remove the array of `rabbit`'s in the above lines 25−30. Construct them with `new` in the constructor for class `game`, initializing each `rabbit` to a different position. The constructor for class `game` will now be too big to be inline. `main` should call `set_new_handler` before constructing the `game` object.

Class `rabbit` originally had an implicitly defined copy constructor. We made the copy constructor private and undefined on p. 200. When we introduced the array of `rabbit`'s on pp. 234−236, we were forced to reinstitute the copy constructor. Now that the array is gone, the copy constructor can, and therefore should, be private and undefined again.

The value of `new` must always be stored in a pointer. Lines 63−65 seem to have forgotten this, but they really have done it. A successful `new` will call the constructor for class `rabbit`, which stores the address of the newborn `rabbit` into the master list for us. (See ¶ (4) of this homework.)

For the time being, the `wolf` will still be constructed with a declaration in `game::play`. After all, we know in advance when the `wolf` will be destructed. It will always be the last animal to go.

```
52 //Excerpt from game.C.
```

```
53
54 game::game(char initial_c)
55     : term(initial_c)
56 {
57     //Get the dimensions of the terminal.
58     const unsigned xmax =
59     const unsigned ymax =
60
61     //Construct as many rabbits as you want, in different places.
62
63     new rabbit(three arguments for constructor);
64     new rabbit(three arguments for constructor);
65     new rabbit(three arguments for constructor);
66 }
```

(12) The `wolf` will now destruct any `rabbit` it steps on. Change the line

```
67                 const bool I_ate_him = g->get(newx, newy) != 0;
```

in `wolf::move` to

```
68                 if (const rabbit *const other = g->get(newx, newy)) {
69                     delete other;
70                 }
```

This assumes that the `other` rabbit is allocated dynamically. It would be a disaster to `delete` an object that wasn't (e.g., one created by a declaration).

`wolf.C` must include `rabbit.h` to tell it if class `rabbit` has a destructor that must be called in the above line 69.

Now that `wolf::move` contains a `delete` statement, we must allocate all of the `rabbit`'s dynamically: we can't `delete` a variable that was constructed with a declaration.

A C++ object is not allowed to commit suicide. For example, a `rabbit` that blunders into a `wolf` can not call its own destructor. Instead, the `move` function of a blundering `rabbit` will return `false` to `game:play`, and `game::play` will call the destructor for the moribund `rabbit`. Incidentally, the asymmetrical behavior on p. 199 will now disappear.

(13) The original main loop of the game relied on the return value of `wolf::move` to tell us if a `rabbit` was killed. But now `wolf::move` will `delete` the `rabbit` for us, destructing it and removing it from the master list. The main loop therefore no longer needs the return value of `wolf::move` to see if all the `rabbit`'s are dead: it can simply call the `empty` member function of the master list. See line 18 of `vector.C` on p. 431.

We therefore change the return type of `wolf::move` from `bool` to `void`. The variable `I_ate_him` will disappear entirely, and all the `return`'s with a value in `wolf::move` will become plain old `return`'s. The `return` in the last line of `wolf::move` can disappear entirely.

(14) Keep the main loop in `game::play`, but change it to the following.

The `rabbit` destructor called in line 81 will remove the dying `rabbit`'s address from the master list. This means that the increment must be at line 78 rather than the expected place, at the end of line 75. But the increment must be executed before the `delete` in line 81. We cannot increment a list iterator that refers to an element that has already been reoved from the list; see the ''increment of death'' on pp. 444−445.

`game.C` must include `rabbit.h` to tell if class `rabbit` has a destructor that must be called in line 81.

It's too bad that we need the two separate `move`'s in lines 72 and 80. We'll fix this when we have inheritance.

```
71      for (; !master.empty(); term.wait(250)) {
72          w.move();
73
74          for (some type of::const_iterator it = master.begin();
75              it != master.end();) {
76
77              rabbit *const p = *it;
78              ++it;
79
80              if (!p->move()) {
81                  delete p;   //Call the destructor and deallocate.
82              }
83          }
84      }
```

(15) The original destructors for classes `rabbit` and `wolf` were complicated by the fact that there might be another animal in the same place at the same time: when a `wolf` stomps on a `rabbit` or when a `rabbit` blunders into a `wolf`. We therefore needed the `if` around the `put` in line 86:

```
85      if (g->term.get(x, y) == c) {
86          g->term.put(x, y);
87      }
```

See p. 200. But now that `wolf::move` `delete`'s the `rabbit`, we will no longer have two animals in the same place at the same time. In the destructors for `rabbit` and `wolf`, remove the `if` in the above lines 85 and 87, but keep line 86.

(16) The destructor for class `game` should display the message "You killed all the rabbits!" and then pause for three seconds. Remove the message and pause from `main`.

(17) If you get the following Microsoft Visual C++ warning,

```
warning C4291:
'void *__cdecl operator new(unsigned int,const struct std::nothrow_t &)' :
no matching operator delete found;
memory will not be freed if initialization throws an exception
```

you can say

```
#pragma warning (disable : 4291)
```

▲

▼ **Homework 4.4b:**
Version 2.1 of the Rabbit Game: read the `rabbit` constructor arguments from an array

Let's get rid of the unsightly repetition in the above lines 63−65. In the constructor for class `game`, create the new data type

```
struct location {
    unsigned x, y;
};
```

Then construct a `const` array named `a` of as many `location`'s as you want, each initialized to the coördinates where you want to construct a `rabbit`. Use the `sizeof / sizeof` idiom to count the number of structures in the array. The constructor for class `game` will loop through the array:

```
    for (loop through the array with a read-only pointer p) {
        if (p->x and p->y are on the screen) {
            new rabbit(this, p->x, //etc.
        }
```

```
}
```

▲

▼ Homework 4.4c:
Version 2.2 of the Rabbit Game: pass the array to the constructor for class `game`

(1) Let struct `location` be a public member of class `game`, just like class `bill` was a public member of class `clinton` in lines 21−26 of `clinton.h` on p. 420.

(2) Add two new arguments to the constructor for class `game`, named `first` and `last`, that are read-only pointers to `location`'s. The existing argument `initial_c` will now be the third argument of the constructor. Let its default value remain `'.'`.

(3) The loop in the constructor for class `game` will now iterate through the array whose first and just-past-the-last elements (or at least their addresses) were passed to it.

(4) Before constructing the `game`, the `main` function should construct a `const` array of `location`'s. Make as many elements as you want, each initialized to the coördinates where you want to construct a `rabbit`. `main` does not know the dimensions of the terminal—there *is* no `terminal` object yet—so it will have to make an educated guess as to where the `rabbit`'s should be located.* Then pass the addresses of the first and just-past-the-last elements to the constructor for class `game`:

```
const game::location a[] = {
    { 0,  0},
    {20,  8},
    {40, 16},
    //etc.: as many rabbits as you want
};
const size_t n = the number of elements in the array a;

game g(a, a + n);        //Does this pair of arguments look familiar?
```

▲

* It would be nice if the `xmax` and `ymax` member functions of class `terminal` were `static`. If so, `main` could call them before constructing the game. But don't do this.