3

Operator Overloading

3.1 Introduction

Not just syntactic sugar

Data members should be private. The internals of an object should be accessible only through its public member functions and friends. Our paradigm examples are in column 1.

But column 2 offers a more familiar notation for these interactions. The operators that we routinely apply to built-in types such as `int` and `char` would also be applied to user-defined types such as classes and enumerations. Extending an operator to accept operands of a user-defined type is called *operator overloading.*

When we write the expressions in column 2, the computer will behave as if we had written the corresponding ones in column 3. These expressions call member functions and friends with the admittedly bizarre names `operator++`, `operator+=`, etc. The operators in column 2 are overloaded by defining the `operator` functions in column 3.

For the present, we adopt the fiction that operator overloading is intended to provide a *convenient* notation for all data types, including classes and enumerations. Its real purpose, however, is to provide the *same* notation for all data types. Convenience has nothing to do with it. And the purpose of a uniform notation is to permit us to plug these data types into a ‘‘template’’. Take a peek at p. 634 and bear in mind that not until then will the real purpose of operator overloading be revealed.

Coming back to earth, column 2 also gives us a more flexible notation for input and output. The `print` in line 15 of column 1 is hardwired to output to `cout`, while the `<<` in column 2 will let us specify any destination: `cout`, `cerr`, `clog`, an output file, etc.

```
 1 /* column 1 */        /* column 2 */         /* column 3 */
 2 date d;               date d;                date d;
 3
 4 d.next();             ++d;                   d.operator++();
 5
 6 d.next(280);          d += 280;              d.operator+=(280);
 7
 8 date e = d;           date e = d + 10;       date e = operator+(d, 10);
 9 e.next(10);
10
11 int n = dist(d, e);   int n = d - e;         int n = operator-(d, e);
12
13 if (equals(d, e)) {   if (d == e) {          if (operator==(d, e)) {
14
15 d.print();            cout << d;             operator<<(cout, d);
```

```
16
17 d.next();            cout << ++d;          operator<<(cout, d.operator++());
18 d.print();
19
20 d.print();           cout << d << "\n";    operator<<(
21 cout << "\n";                                  operator<<(cout, d), "\n");
22
23 d.next();            cout << ++d << "\n";  operator<<(
24 d.print();                                     operator<<(cout,
25 cout << "\n";                                  d.operator++()), "\n");
```

What we can't do with operator overloading

(1) The following six operators can never be overloaded. For example, `sizeof` always yields the number of bytes in its operand no matter what its data type is. It never calls an `operator` function that does something else.

```
sizeof x
typeid(x)
b ? x : y
c::m
o.m
o.*pm
```

(2) We can define an `operator` function for an operator that has at least one operand of a user-defined data type: a class or an enumeration. But we can't define an `operator` function for an operator whose operands are all of the built-in data types. The expression `cout << d` might perform output, but `10 << 2` will always perform left shift.

(3) We cannot change an operator's precedence, associativity, or arity (number of operands, pp. 2−3). We cannot change whether a unary operator is prefixed or postfixed to its operand. For example, we can make it possible to apply the negation operator (minus sign) to an object, but it must remain prefix.

```
1     date d;
2
3     //Change AD to BC.
4     d = -d;   //can make prefix compile: d = d.operator-();
5     d = d-;   //but can't make postfix compile
```

(4) We cannot create new operators. For example, we cannot define an `operator**` function to implement an exponentiation operator.

```
x = base ** power;              //won't compile
```

A monolithic example

	read the value of existing object(s)	*change the value of existing object(s)*	*construct and return a new object*

binary	`operator==`
	`operator-(const obj&, const obj&)`
	`operator<<`
		`operator+=`
		`operator>>`
			`operator+(obj, int)`
			`operator-(obj, int)`

unary	`operator!` `operator int`	`operator++` *(prefix)* `operator++` *(postfix)*	`operator-` `operator++` *(postfix)*

Let's overload the familiar operators to accept operands of class `date`. For this purpose, the simplest implementation of the class is the one with one non-static data member, the `int day` in line 7 of `date.h` on p. 273. The two constructors in lines 26 and 27 call the common subroutine in line to perform

error checking and install a value into the data member. The `next` member function has been renamed `operator++` in line 47. It will be replaced by lines 48 and 79. The superseded lines are commented out but remain in the source code for pedagogical reasons. We didn't bother with an `operator--` to replace `prev`.

Since many of the `operator` functions call each other, we present them in a single monolithic example. In order of increasing invasiveness, we will see functions that examine an existing object; ones that change the value of an existing object; and ones that construct a new object. The postfix `operator++` is a special case: it will construct a new object *and* change

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

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class date {
 7     int day;              //number of days before or after Jan 1, 0 AD
 8     static const int length[];
 9     static const int    pre[];
10 public:
11     enum month_type {     //indices into the length array
12         january = 1,
13         february,
14         march,
15         april,
16         may,
17         june,
18         july,
19         august,
20         september,
21         october,
22         november,
23         december
24     };
25
26     date(int month, int d, int year) {install(month, d, year);}
27     date();               //initialize to the current date
28     bool install(int month, int d, int year);
29
30     int julian() const;
31     void print() const;   //output the date to cout
32
33 //  bool operator==(const date& another) const {return day == another.day;}
34     friend bool operator==(const date& d1, const date& d2){return d1.day==d2.day;}
35     friend int  operator- (const date& d1, const date& d2){return d1.day-d2.day;}
36
37 //  friend bool operator!=(const date& d1, const date& d2){return d1.day!=d2.day;}
38 //  friend bool operator!=(const date& d1, const date& d2) {return !(d1 == d2);}
39
40     date& operator+=(int count) {day += count; return *this;}
41 //  friend date& operator+=(date& d, int count) {d.day += count; return d;}
42     date& operator-=(int count) {day -= count; return *this;}
43
```

```
44 //  const date operator+(int count) const {date d=*this; d.day+=count; return d;}
45 //  const date operator+(int count) const {date d = *this; return d += count;}
46
47 //  date& operator++() {++day; return *this;}                   //prefix
48 //  date& operator++() {return *this += 1;}                     //prefix
49
50 //  const date operator++(int) {                                //postfix
51 //      const date old = *this;
52 //      ++day;
53 //      return old;
54 //  }
55
56 //  const date operator++(int) {                                //postfix
57 //      const date old = *this;
58 //      ++*this;     //(*this).operator++();
59 //      return old;
60 //  }
61
62      const date operator-() {date d = *this; d.day=-d.day; return d;} //unary
63
64      operator int() const {return julian();}
65      operator long() const {return day;}
66      operator bool() const {return abs(day) < 4000 * 365;}
67
68      friend ostream& operator<<(ostream& ost, const date& d);
69      friend istream& operator>>(istream& ost,       date& d);
70 };
71
72 inline bool operator!=(const date& d1, const date& d2) {return !(d1 == d2);}
73 inline bool operator!(const date& d) {return !static_cast<bool>(d);}//call l. 64
74
75 inline const date operator+(date d, int count) {return d += count;}
76 inline const date operator+(int count, date d) {return d += count;}
77 inline const date operator-(date d, int count) {return d -= count;}
78
79 inline date& operator++(date& d) {return d += 1;}              //prefix
80
81 inline const date operator++(date& d, int)                     //postfix
82 {
83     const date old = d;
84     ++d;     //d.operator++();
85     return old;
86 }
87
88 inline date::month_type& operator++(date::month_type& m)       //prefix
89 {
90     if (m == date::december) {
91         m = date::january;
92     } else {
93         m = static_cast<date::month_type>(m + 1);
94     }
95     return m;
96 }
97
```

```
 98 inline const date::month_type operator++(date::month_type& m, int) //postfix
 99 {
100     const date::month_type old = m;
101     ++m;     //operator++(m);
102     return old;
103 }
104 #endif
```

The above lines 90–95 can be written as a single expression with only one assignment and one `return`. The `?:` executes before the `=` because of their equal precedence and right-to-left associativity. This is much harder to read.

```
105     return m = m == date::december ? date::january :
106         static_cast<date::month_type>(m + 1);
```

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

```
  1 #include <iostream>
  2 #include <cstdlib>
  3 #include <ctime>
  4 #include "date.h"
  5 using namespace std;
  6
  7 const int date::length[] = {
  8     0,                      //dummy element to give january subscript 1
  9     31,   //january
 10     28,   //february
 11     31,   //march
 12     30,   //april
 13     31,   //may
 14     30,   //june
 15     31,   //july
 16     31,   //august
 17     30,   //september
 18     31,   //october
 19     30,   //november
 20     31    //december
 21 };
 22
 23 const int date::pre[] = {
 24     0,                      //dummy element to give january subscript 1
 25     0,                      //january
 26     pre[ 1] + length[ 1],   //february
 27     pre[ 2] + length[ 2],   //march
 28     pre[ 3] + length[ 3],   //april
 29     pre[ 4] + length[ 4],   //may
 30     pre[ 5] + length[ 5],   //june
 31     pre[ 6] + length[ 6],   //july
 32     pre[ 7] + length[ 7],   //august
 33     pre[ 8] + length[ 8],   //september
 34     pre[ 9] + length[ 9],   //october
 35     pre[10] + length[10],   //november
 36     pre[11] + length[11]    //december
 37 };
 38
```

```
39 date::date()     //Initialize to the current date.
40 {
41     const time_t t = time(0);
42     if (t == static_cast<time_t>(-1)) {
43         cerr << "time failed\n";
44         exit(EXIT_FAILURE);
45     }
46     const tm *const p = localtime(&t);
47     install(p->tm_mon + 1, p->tm_mday, p->tm_year + 1900);
48 }
49
50 bool date::install(int month, int d, int year)
51 {
52     if (month < january || month > december) {
53         cerr << "bad month "
54             << month << "/" << d << "/" << year << "\n";
55         return false;
56     }
57
58     if (d < 1 || d > length[month]) {
59         cerr << "bad day "
60             << month << "/" << d << "/" << year << "\n";
61         return false;
62     }
63
64     day = 365 * year + pre[month] + d - 1;
65     return true;
66 }
67
68 int date::julian() const
69 {
70     int j = day % 365;
71
72     if (j < 0) {
73         j += 365;
74     }
75
76     return j + 1;
77 }
78
79 void date::print() const
80 {
81     div_t divide = div(day, 365);
82     if (divide.rem < 0) {
83         divide.rem += 365;
84         --divide.quot;
85     }
86
87     int d = divide.rem + 1;     //Julian date (1 to 365)
88     int month;                  //uninitialized variable
89
90     for (month = january; d > length[month]; ++month) {
91         d -= length[month];
92     }
```

```
 93
 94     cout << month << "/" << d << "/" << divide.quot;
 95 }
 96
 97 ostream& operator<<(ostream& ost, const date& d)
 98 {
 99     div_t divide = div(d.day, 365);
100     if (divide.rem < 0) {
101         divide.rem += 365;
102         --divide.quot;
103     }
104
105     int day = divide.rem + 1;   //Julian date (1 to 365)
106     int month;                  //uninitialized variable
107
108     for (month = date::january; day > date::length[month]; ++month) {
109         day -= date::length[month];
110     }
111
112     return ost << month << "/" << day << "/" << divide.quot;
113 }
114
115 istream& operator>>(istream& ist, date& d)
116 {
117     int month;
118     if (!(ist >> month)) {   //if (ist.operator>>(month).operator!()) {
119         return ist;
120     }
121
122     char c;
123     if (!(ist >> c)) {
124         return ist;
125     }
126     if (c != '/') {
127         ist.setstate(ios_base::failbit);
128         return ist;
129     }
130
131     int day;
132     if (!(ist >> day)) {
133         return ist;
134     }
135
136     if (!(ist >> c)) {
137         return ist;
138     }
139     if (c != '/') {
140         ist.setstate(ios_base::failbit);
141         return ist;
142     }
143
144     int year;
145     if (!(ist >> year)) {
146         return ist;
```

```
147     }
148
149     if (!d.install(month, day, year)) {
150         ist.setstate(ios_base::failbit);
151     }
152
153     return ist;
154 }
```

3.2 An `operator` that examines an object

Apply the == operator to two `date`'s

The first operator that we will apply to a `date` object will be `==`. We chose it because it does not create any new object, or even change the value of an existing object. It merely examines the value of existing objects.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date d1;   //Initialize to today's date.
 9     date d2;
10
11     if (d1 == d2) {
12         cout << "They're the same date.\n";
13     } else {
14         cout << "Midnight occurred between lines 8 and 9.\n";
15     }
16
17     return EXIT_SUCCESS;
18 }
```

```
They're the same date.
```

To use `date` objects as the operands of an `==`, we must define a function named `operator==`. One advantage of operator overloading is that it offers standard names for the most common functions. Instead of `equal`, `is_equal`, or the `equals` we had on p. 211, we can simply name our function `operator==`.

The return value of the `operator==` will be the value of the expression `d1 == d2`. `operator==` will therefore return `bool`.

Since our `operator==` will need access to the private members of class `date`, it will have to be a member function or a friend of that class. *Any* function that accesses the private members of any class would have to be a member function or a friend, and the `operator` functions obey the same rules.

Deciding between a member function or a friend (or neither) is the first big decision we have to make when defining an `operator` function. Fortunately, the rules are the same as for a normal function. For pedagogical purposes we will try it both ways. The alternatives are equally correct and efficient; our choice can be based purely on æsthetics.

operator== as a member function

In the expression `d1 == d2` in the above line 11, `d1` and `d2` are called the *left* and *right operands* of the `==`. If we define an `operator` function for a binary operator as a member function, the language says that the function must be a member function of the left operand and must receive the right operand as its argument. In this case, line 11 would behave as if we had written

```
19      if (d1.operator==(d2)) {
```

Compare p. 282.

The definition of this `operator==` is in line 33 of `date.h` on p. 273. To ensure that the function cannot change or damage `d1`, we made it a `const` member function. To avoid copying `d2`, we passed it by reference. To ensure that the function cannot change or damage `d2`, we passed it as a read-only reference.

But the function definition in line 33 is commented out because we can do it better. The problem is that the notation is lopsided. The function deals with two objects, but only one of them has a name.

The advantage of a member function is that it can refer to the members of an object by means of the simplest possible notation: no notation at all. But this simplicity becomes a liability when we have two objects. How can we provide names for both?

operator== as a friend

We saw the solution back on pp. 204−206. Instead of the asymmetrical member function in line 33 of `date.h`, we can define the perfectly balanced friend in line 34. Now that `operator==` is a free function (i.e., not a member function, p. 113), the above line 11 will now behave as if we had written

```
20      if (operator==(d1, d2)) {
```

This change makes the function body more symmetrical, but does not increase its speed. Deep down in the machine, both versions take the same two arguments. Only the surface notation is different: one explicit and one implicit argument for the member function; two explicit arguments for the friend.

But notation is important. The friend definition acts as documentation, announcing that we will be dealing evenhandedly with more than one object. The member function in line 33 is for demo purposes only, and is commented out to avoid a name collision with 34.

We motivated our decision carefully because it will almost always go the same way. Operator overloading is mostly uniform boilerplate. `operator==` will almost always be a friend, not a member function, for all classes. (For a pathological example of an `operator==` that has to be a member function, see line 302 of `terminal.h` on p. 976.)

Apart from the strange name, the only special feature of an `operator` function is the shorthand notation with which it can be called. Instead of the above lines 19 or 20, we can write line 11.

An operator− that occupies the same ecological niche

The `operator-` in line 35 of `date.h` is a friend for the same reason as the `operator==` in 34. It supersedes the `dist` function on p. 211. We can now see why the return value of `dist` was positive when its left argument was a date that was later than its right argument: we wanted to make it compatible with the `operator-` that would replace it.

There is another `operator-` in line 77 of `date.h`. As usual, we can have two functions with the same name as long as their arguments are different. This function is more complicated because it constructs a new object; we will talk about it on p. 286.

An operator== that doesn't necessarily compare all the data members

Why do we have to define `operator==` at all? Why isn't it just built into the language that two objects are equal if their corresponding data members are equal? Well, often we want to provide our own definition for equality. For example, two objects in an engineering application might be considered equal if

they are within 5% of each other. Two objects in the federal budget might be considered equal if they are within a billion dollars of each other: ‘‘close enough for government work’’.

Consider the `stack` objects on pp. 149−154. When deciding if two are equal, only the first `n` elements of their `a` data members should be examined. And if both stacks are empty they should be considered equal without comparing the `a`’s at all.

```
1 //Excerpt from stack.C showing a friend of class stack.
2
3 bool operator==(const stack& s1, const stack& s2)
4 {
5     if (s1.n != s2.n) {
6         return false;
7     }
8
9     for (size_t i = 0; i < s1.n; ++i) {
10        if (s1.a[i] != s2.a[i]) {
11            return false;
12        }
13    }
14
15    return true;
16 }
```

Apply the != operator to two date’s

The `operator!=` in line 37 of `date.h` on p. 273 is a friend, like the `operator==` in line 34. It has to be a member function or a friend, because it mentions the private member `day`.

But a class should have no unnecessary members or friends: we want to minimize the number of suspects when the data members are found to have the wrong values. Accordingly, line 38 rewrites `operator!=` so that it no longer mentions any private member. Note that the `!=` in the {body} of line 37 compares two integers; it is built into the language. The `==` in the body of 38 compares two `date`’s; we created it ourselves in line 34. The parentheses in 38 force the `==` to execute before the `!`.

Line 38 is merely a *call-through*, a function that calls another to do most of its work. But the extra call and return do not exist at the machine level. The expression

```
1     !(d1 == d2)
```

in 38 behaves as if we had written

```
2     !operator==(d1, d2);
```

This `operator==` is an inline function, so any call to it is replaced by a copy of its body in line 34. The `!(d1 == d2)` therefore behaves as if we had written a direct comparison of the two integers

```
3     !(d1.day == d2.day)
```

and the compiler will optimize this to

```
4     d1.day != d2.day
```

It is no sin for one `operator` function to call another if the first is inline.

The advantage of line 38 is that it mentions no private member of class `date`. It therefore no longer needs to be a member function or a friend of the class `date`, and has been moved down to line 72. The definitions in 37 and 38 were for pedagogical purposes only.

To keep the `operator!=` inline it must be defined in `date.h`, not `date.C`. But now that it is defined outside the curly braces of the class definition for `date` (lines 6 and 70), the function requires the keyword `inline`. This makes it static in the C sense: visible only in this header file and the `.C` files that include it. Were it not static, the function would be ‘‘multiply defined’’ if the header file were included in

more than one `.C` file of a program. See p. 99.

▼ Homework 3.2a: define the four other comparison operators

> We write $y \geq x$ in case $x \leq y$ . . .
>
> —Paul R. Halmos, *Naïve Set Theory*, §14

We have defined `operator==` and `operator!=` for class `date`. Define the four remaining comparison operators

```
operator<
operator>
operator<=
operator>=
```

Define `operator<` as a friend like `operator==`. Define the other three in `date.h` as inline functions that are neither members nor friends. `operator>=` should call `operator<` just like `operator!=` calls `operator==`. `operator>` should call `operator<`, and `operator<=` should call `operator>=`, by reversing the operands:

```
1 inline bool operator>(const date& d1, const date2 d2) {return d2 < d1;}
```

The function `equals` that you wrote on p. 211 should be renamed `operator==`. Simplify the body of the `min` function in that homework to

```
2     {return d2 < d1 ? d2 : d1;}
```

Do not simplify `min` to `d1<d2?d1:d2`; this would change the behavior of `min` when `d1 == d2`.

For advanced applications, an `operator<` must obey additional rules. See pp. 776–777.

▲

▼ Homework 3.2b: define an operator< for a class date with three data members

Define an `operator<` friend for a class `date` with the three private, non-static data members

```
1     int year;
2     int month;          //date::january to date::december inclusive
3     int day;            //1 to date::length[month] inclusive
```

One way to do this would be to compute each `date`'s distance from January 1, 0 A.D. (In other words, compute the value that would be in the one-data-member version of the object.) We could the compare the resulting numbers to find out if the first date is earlier than the second.

Another way would be to encode each date as an integer value in a format such as `20141231` or `2014365`. We could then compare the two resulting numbers. A third approach would be to pass the two `date`'s to the `operator-` that returns the distance between two `date`'s. If the resulting number is negative, we could return true.

But all of the above strategies do more work than is necessary. Figure out which `date` is earlier by comparing the data members of the two `date`'s. Your `operator<` must not call any other function, including `julian`. Demonstrate that your `operator<` is correct by handing it in together with the output of `http://i5.nyu.edu/~mm64/book/src/less/main.C`.

▲

▼ Homework 3.2c: define an operator! for a class date with three data members

Define an `operator!` that would test if the three integer data members of a `date` have a legal combination of values:

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "date.h"
```

```
 4 using namespace std;
 5
 6     date d;
 7
 8     //Recklessly sabotage the date; see p. 133
 9     int *const p = reinterpret_cast<int *>(&d);
10     p[1] = 100;
11
12     if (!d) {               //if (d.operator!()) {
13         cerr << "d is in an inconsistent state.\n"
14         return EXIT_FAILURE;
15     }
```

`operator!` will be a public `const` member function returning `bool`. Since the `!` operator is unary and the `operator!` function is a member function, the language says that the function will be a member of its operand. The above line 11 will behave as if we had written the comment alongside. Compare p. 279.

`operator!=` will return true if the `month` data member is between `january` and `december` inclusive and the `day` data member is between 1 and `length[month]` inclusive; false otherwise. Note that an `operator!` for the one-data-member class `date` would always return `true`, because any value for the data member is legal.

`cin` and `cout` have an `operator!` that serves the same purpose; see pp. 319−320.

▲

3.3 An `operator` that changes an object

More invasive than `operator==` is `operator+=`, which changes the value of an object.

Always define `operator+=` before `operator+`. C programmers find this surprising because they consider `+=` more exotic than `+`. But `+=` merely changes the value of an existing variable, while `+` constructs a whole new variable. For example, the `+=` in the following line 4 deposits a sum into the existing variable `a`, while the `+` in line 5 constructs an *anonymous temporary* variable to hold the sum:

```
1     int a = 10;
2     int b = 20;
3
4     a += b;                   //Change value of the existing variable a.
5     cout << a + b << "\n";    //Construct a new variable.
6     cout << (a += b) << "\n"; //The value of a += b is the new value of a.
```

Note also that the expression `a += b` does more than change the value of `a`. The expession also produces a value of its own, which is the new value assigned to `a`. Line 4 never uses the value of `a += b` for anything, but line 6 uses the value of `a += b` as an operand in a larger expression.

To agree with everyone's expectations, our `operator+=` will have to behave the same way. The following line 13 shows that the expression `today += 7` changes the value of `today`. Lines 16, 19, and 20 show that the value of the expression is the new value of `today`. To make this happen, `operator+=` will have to return the new value of the object.

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

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

```
 6 void f(const date& d);
 7
 8 int main()
 9 {
10     date today;
11     f(today);
12
13     today += 7;  //changes the value of today
14     f(today);
15
16     f(today += 7);
17
18     cout << "The julian date that is 7 days after ";
19     (today += 7).print();
20     cout << " is " << (today += 7).julian() << ".\n";
21
22     f((today += 7) += 365);
23     return EXIT_SUCCESS;
24 }
25
26 void f(const date& d)
27 {
28     d.print();
29     cout << " has julian date " << d.julian() << ".\n";
30 }
```

Member function or friend?

`operator+=` needs to mention the private member `day`, so it will have to be a member function or friend of class `date`. If it is a member function, the above line 13 will behave as if we had written

```
1     today.operator+=(7);
```

If it is a free function, line 13 will behave as

```
2     operator+=(today, 7);
```

Lines 40 and 41 of `date.h` on p. 273 define `operator+=` as a member function and friend. Line 41 requires us to define a reference `d`. Line 40 is simpler and better. For an unusual `operator+=` that does not need to be a member function or a friend, see line 49 of `printable.h` on p. 736. For an off the grid `operator+=` that can't be a member function or a friend, see p. 903.

Return by value or by reference?

The `+=` operator always returns the new value of its left operand. In the above line 16, the left operand is an object. Should it be returned by value or by reference? This is the second big decision we have to make when defining an `operator` function. Again, the rules are the same as for a normal function.

Recall that *return by value* constructs and returns a copy of an object; *return by reference* merely returns the address of the object. Return by value is slower and might have side effects, so we return by reference whenever possible. The latter is always possible except for one case: when the returned object is destructed by the very act of being returned. This happens when the returned object is automatically allocated in the function from which we are returning.

In line 40 of `date.h`, `operator+=` is a member function. The returned object is the object `*this` to which the member function belongs. `*this` is not an automatic variable in `operator+=`, so there is nothing to prevent a return by reference. Line 40 specifies the return by reference by declaring a return type of `date&`.

In line 41 of `date.h`, `operator+=` is a friend. The returned object is the one to which `d` refers. Although the reference `d` is automatically allocated, the object to which it refers is not, thus allowing a return by reference. But we have already rejected 41 as inferior to 40.

Assignment operators such as `operator+=`, `operator-=`, and `operator=` always return the new value of their object (`*this`) so that it can be used in a larger expression. Examples are in lines 16, 19, and 20 above. For a pathalogical case where they do not return `*this`, see p. 903.

Dereference and re-reference

Despite the star, the `return *this;` in line 40 of `date.h` returns the object's address. The star dereferences the pointer `this`; the return by reference ''re-references'' it. Since the star and the return by reference cancel each other out, couldn't we return the object's address more simply by getting rid of both of them?

```
1     date operator+=(int count) {day += count; return this;} //doesn't compile
```

The new function seems plausible, but it will make it harder to use the return value. First of all, we have to get it to compile. Since `operator+=` now returns `this`, we would have to change the return type to ''pointer to `date`'':

```
2     date *operator+=(int count) {day += count; return this;} //does compile
```

When we use the return value, we would always have to remember to write a star in front of it. The above line 16 would become

```
3     f(*(today += 7));
```

with an extra pair of parentheses. It would be simpler to banish the star in the above line 3 to line 40 of `date.h`, where it can be written once and for all. This, by the way, is exactly why Stroustrup introduced references into C++.

Return an lvalue or an rvalue?

Return by reference is faster than return by value. More importantly, return by reference is necessary to make our `operator+=` behave like the built-in `+=`. Consider the following expression, which adds 10 to an `int i` and then knocks the sum down to the range 0 to 19 inclusive. It demonstrates that the value of `i += 10` is an *lvalue,* an expression that can be assigned to (pp. 12−13). A realistic example is in line 20 of `main.C` on p. 998.

```
1     (i += 10) %= 20;
```

To permit our `operator+=` to return an lvalue, it must return by reference and the reference must be read/write. The above line 22 shows that the value of `today += 7` is an lvalue. (The inner parentheses override the right-to-left associativity of `+=`.) Of course the example could be written more simply as

```
2     f(today += 372);
```

but for the time being the extra `+=` is our only way to demonstrate that `today += 7` is an lvalue.

We have motivated our decisions carefully because they will almost always go the same way. For all normal clases, `operator+=` will be a public, non-static member function that returns `*this` as a read/write reference.

An operator−= that occupies the same ecological niche

Line 42 of `date.h` shows the correct way to implement `operator-=`. Resist the temptation to implement it in `date.h` as neither a member function nor a friend:

```
1                                    //return d.operator+=(-count);
2 date& operator-=(date& d, int count) {return d += -count;}
```

The problem in the above line is that the negation could overflow. For example, if our integers are 32-bit

two's complement, their values will be limited to the slightly lopsided range −2,147,483,648 to 2,147,483,647 inclusive. If `count` were −2,147,483,648, the integer expression `-count` could not possibly hold the correct value of 2,147,483,648.

```
4/8/2014 has julian date 98.                                  lines 10−11
4/15/2014 has julian date 105.                                lines 13−14
4/22/2014 has julian date 112.                                line 16
The julian date that is 7 days after 4/29/2014 is 126.        lines 18−20
5/13/2015 has julian date 133.                                line-22
```

3.4 An `operator` that constructs an object

`operator+` computes a sum and constructs a new object to hold it. The sum does not go into any previously existing object. The `today` in the following line 13, for example, remains unchanged.

The value of the expression `today + 7` is the new object. To make this happen, `operator+` must return the new object that it constructs.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 void f(const date& d);
 7
 8 int main()
 9 {
10     date today;
11     f(today);
12
13     f(today + 7);    //does not change the value of today
14     f(today);
15
16     (today + 7).print();
17     cout << " has julian date " << (today + 7).julian() << ".\n";
18
19     return EXIT_SUCCESS;
20 }
21
22 void f(const date& d)
23 {
24     d.print();
25     cout << " has julian date " << d.julian() << ".\n";
26 }
```

Return by value or by reference?

If `operator+` were a member function of class `date`, the above line 13 would behave as if we had written

```
27     f(today.operator+(10));
```

This member function is defined in line 44 of `date.h` on p. 274, where the new object is named `d`. The new object begins its life as a copy of the existing object, but is immediately increased. Since it is

automatically allocated, it cannot be returned by reference. It must be returned by value.

Return an lvalue or an rvalue?

Return by value is not sufficient: the new object must be returned as a `const` value. Surprisingly, it's up to us to ensure that the expression `today + 7` is not an lvalue:

```
28     //Must not be allowed to compile.
29     (today + 7) = tomorrow;         //today.operator+(7) = tomorrow;
```

The way to prevent `today + 7` from being an lvalue is to make the value of the expression a `const`. We accomplish this by having `operator+` return a `const date`.

Member function or friend?

The `operator+` in line 44 of `date.h` has to be a member function or friend because it mentions the private member `day`. But we can easily rewrite it in line 45 without any mention of `day`. The `+=` in 45 that allows us to do this is the `operator+=` we defined in 40. Now that `day` is unmentioned, we can redefine `operator+=` in line 75 as neither a member function nor a friend. This function creates the new object by receiving its first argument by value. As above, the new object begins its life as a copy of an existing object, and the most natural way to do this is via pass by value.

Of lines 44, 45, and 75 of `date.h`, the last is the simplest. For every class, `operator+` will be neither a member function nor a friend. It will be a call-through to `operator+=`. The first argument will by passed by value; the return value will be passed as a `const` value. When we call this `operator+`, the above line 13 will behave as if we had written

```
30     f(operator+(today, 7));
```

operator+ and operator−

We also need lines 76 and 77 of `date.h` to allow us to say `10 + today` and `today - 10`. We now have two different `operator-` functions:

```
1      date today;
2      date d = today - 10;  //date d = operator-(today, 10); line 77 of date.h
3      int dist = today - d; //int dist = operator-(today, d); line 35 of date.h
```

`4/8/2014 has julian date 98.`	*lines 10−11: the original value*
`4/15/2014 has julian date 105.`	*line 13: print the value of the expression* `today + 10`
`4/8/2014 has julian date 98.`	*line 14: demonstrate that line 13's* `+` *had no effect on* `today`
`4/15/2014 has julian date 105.`	*lines 16−17*

▼ Homework 3.4a:

Use the `operator+` that creates a new `date` to simplify `employee::retire` in line 17 of `employee.h` on p. 259.

▲

3.5 Member Functions vs. Friends

Not every `operator` function needs to mention the private members of the object(s) passed to it. But if it does need to mention them, an `operator` function (or indeed any function) must be either a member function or a `friend`.

When do we have no choice of member function vs. friend vs. neither?

In four cases, we have no choice:

(1) If the `operator` function must be private, it must be a member function. The terms ''public'' and ''private'' apply only to members, not to friends. `operator=` is the one that is most frequently private.

(2) The following `operator`'s must be member functions of their left operand.* They must therefore be member functions, and non-static ones to boot.

```
operator=
operator[]
operator()
operator->
```

Oddly, `operator+=` and the other reassignment operators do not have to be member functions.

(3) If an `operator` function is a member function, it must be a member function of its left or only operand. The `+` in line 2 could be a member function of its left operand `today`, or it could be a non-member function. But the `+` in line 4 could not be a member function of its left operand. Only objects have member functions, and 7 is not an object. The `+` in line 4 is the non-member function in line of `date.h`.

```
1     date today;
2     date e = today + 7;            //could be date e = today.operator+(7);
3                                    //or date e = operator+(today, 7);
4     date e = 7 + today;            //must be e = operator+(7, today);
```

Similarly, the `++` and `--` operators (prefix and postfix) whose operands are enumerations cannot be member functions. Only objects have member functions, and an enumeration is not an object. The following increments are the non-member functions defined in lines 88 of date.h and 98 of date.h of `date.h`.

```
5     date::month_type m = date::january;   //m is an enumeration
6     ++m;                           //must be operator++(m);
7     m++;                           //must be operator++(m, 0);
```

(4) An `operator` function can never be a member function of its right operand. If it needs to mention a private member of its right operand, it must be a `friend` of the operand. For example, we will see later that the `operator<<` function

```
8     date today;
9     cout << today;                 //operator<<(cout, today);
```

must be a `friend` of class `date`. Of course, an `operator` function could also be a member of its left operand. A function can be a `friend` of many classes as well as a member of one class.

What should we do when we do have a choice?

When not constrained by the above four cases, an `operator` function that needs to mention a private member can be either a member function or a `friend`, whichever is more natural.

(1) If the `operator` function does not need to mention any private member, make it neither a member function nor a `friend`. Examples we have seen include `operator!=` and the binary `operator+`.

The remaining cases assume that the `operator` function does need to mention a private member.

(2) An `operator` function for a unary operator should be a member function of the operator's operand. Any function that does something to one object should be a member function of that object:

* This requirement will cramp our style on p. 903.

```
if (!d) {        //if (d.operator!()) {
d = -d;          //d = d.operator-();
```

(3) If the operator is binary and the `operator` function treats the operands the same way, make it a friend.

```
a == b           //operator==(a, b)
a < b            //operator<(a, b)
a + b            //operator+(a, b)
a - b            //operator-(a, b)
```

(4) If the operator is binary and the left operand plays the starring rôle, make the `operator` function a member function of the left operand. For example, many `operator` functions change the value of their left operand but not their right:

```
a += b           //a is affected, b remains untouched: a.operator+=(b)
a -= b           //a is affected, b remains untouched: a.operator-=(b)
```

3.6 `operator++`, Prefix and Postfix

Prefix operator++ changes the value of an existing object

The prefix ++ operator occupies the same ecological niche as +=. In the following line 13, the ++ changes the value of its operand. The value of the expression `++today` is the new value of the object. And the value of the expression `++today` is an lvalue, allowing it to be subjected to another ++ in line 17 and to += in line 18.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 void f(const date& d);
 7
 8 int main()
 9 {
10     date today;
11     f(today);
12
13     ++today;             //operator++(today);
14     f(today);
15
16     f(++today);          //f(operator++(today));
17     f(++++today);        //f(operator++(operator++(today)));
18     f(++today += 365);   //f(operator++(today).operator+=(365));
19
20     cout << "The julian date of the day after ";
21     (++today).print();   //operator++(today).print()
22     cout << " is " << (++today).julian() << ".\n";
23
24     return EXIT_SUCCESS;
25 }
26
```

```
27 void f(const date& d)
28 {
29         d.print();
30         cout << " has julian date " << d.julian() << ".\n";
31 }
```

Line 47 of `date.h` on p. 274 defines `operator++` as a member function in the mold of the `operator+=` in line 40. (An `operator++` with no arguments is the prefix operator.) It returns `*this` as a read/write reference to allow the above lines 17 and 18 to give the object an even newer value.

The `operator++` in line 47 of `date.h` must be a member function because it mentions the private member `day`. But we can easily rewrite it in line 48 without any mention of `day`. The `+=` in line 48 that allows us to do this is the `operator+=` we defined in 40. Now that `day` is unmentioned, we can redefine `operator++` in line 79 as neither a member function nor a friend.

```
4/8/2014 has julian date 98.                          lines 10−11: the original value
4/9/2014 has julian date 99.                          lines 13−14
4/10/2014 has julian date 100.                        line 16
4/12/2014 has julian date 102.                        line 17
4/13/2015 has julian date 103.                        line 18
The julian date of the day after 4/14/2015 is 105.    lines 20−22
```

Postfix operator++ constructs a new object

The postfix `++` operator constructs a new object *and* changes the value of an existing object. In the following line 16, for example, three actions are performed.

(1) The `++` constructs a copy of `today`.

(2) Then it changes the value of `today`, but this has no effect on the copy of the original value.

(3) Finally, it returns the copy.

The value of the expression `today++` is therefore not the current value of `today`. It is a copy of the value that was in `today` before it was incremented.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 void f(const date& d);
 7
 8 int main()
 9 {
10     date today;
11     f(today);
12
13     today++;           //operator++(today, 0);
14     f(today);
15
16     f(today++);        //f(operator++(today, 0));
17
18     cout << "The julian date that is 1 day after ";
19     today++.print();   //operator++(today, 0).print();
20     cout << " is " << today++.julian() << ".\n";
21
```

```
22     //today++ += 365;  //won't compile: today++ is not an lvalue
23     return EXIT_SUCCESS;
24 }
25
26 void f(const date& d)
27 {
28         d.print();
29         cout << " has julian date " << d.julian() << ".\n";
30 }
```

Line 50 of `date.h` on p. 274 shows the three actions. As in line 44, the automatically allocated variable must be returned by value. An it must be returned as a `const` value to prevent the above line 22 from compiling.

The value of the `int` argument of postfix `operator++` is never used. It's just a kludge to let us have two different member functions with the same name. Don't even give it a name: if we did, we would get the ''unused argument'' warning. For another argument whose value is unused, see p. 756.

Line 50 of `date.h` has to be a member function or friend because it mentions the private member `day`. But we can easily rewrite it in line 56 without any mention of `day`. The `++` in line 58 that allows us to do this is the prefix `operator++` we defined in 79. Now that `day` is unmentioned, we can redefine `operator++` in line 81 as neither a member function nor a friend. Note that the argument in line 81 must be pased as a read/write reference,

```
4/8/2014 has julian date 98.                                  lines 10−11: the original value
4/9/2014 has julian date 99.                                  lines 13−14
4/9/2014 has julian date 99.                                  line 16
The julian date that is 1 day after 4/10/2014 is 101.         lines 18−20
```

At last we have a reason to prefer

```
31     for (int i = 0; i < 10; ++i) {              //fast
```

to

```
32     for (int i = 0; i < 10; i++) {              //slow
```

Postfix `operator++` for an object is much slower than prefix. The above `i` is not an object, but it may become one in the future. Get into the habit of using prefix whenever possible.

Increment an enumeration

Line 88 of `date.h` on p. 274 is a prefix that increments an enumeration. It cannot be a member function or friend of the enumeration. The reason is simple: an enumeration can have no members or friends. Only an object can have them.

Our `operator++` wraps around from `december` to `january`. Most of the time, however, it merely does the arithmetic in line 93 `date.h`. This line converts the enumeration to `int`, does some arithmetic, and converts the `int` result back to an enumeration. As we saw on p. 223, enumeration-to-`int` conversion can be implicit, but the conversion back requires a cast.

Since the `operator++` functions change the value of their enumeration arguments, the arguments must be passed as read/write references. As usual, the postfix `operator++` in line 98 of `date.h` does its work by calling the prefix `operator++` in line 88.

```
33 #include <iostream>
34 #include "date.h"
35 using namespace std;
36
37     date::month_type m = date::january;
38     cout << ++m << "\n";            //cout << operator++(m) << "\n";
```

```
39      cout << m++ << "\n";            //cout << operator++(m, 0) << "\n";
```

▼ **Homework 3.6a: define operators for class life**

(1) Change the name of `life::next` to `life::operator++`. Give it no arguments, so it will be the prefix `operator++`. Change its return type from `void` to `life&` and have it `return *this;`.

(2) Define a postfix `operator++` for class `life`, which will call the prefix `operator++` to do most of its work. Do not define `operator--`'s for class `life`: there is no way to work backwards to the previous generation.

(3) Create `life::operator+=`. If its right operand is negative, write an error message (including the negative value) to `cerr` and call `exit(EXIT_FAILURE)`. Otherwise go into a loop that repeatedly calls the prefix `operator++`.

(4) Write two `life::operator+`'s, just like the two `date::operator+`'s in lines 58−59 of `date.h`.

Test them with the following object `g`. Line 4 shows that prefix `++` returns a value; line 8 shows that postfix `++` returns a value. Line 13 shows that `+=` returns a value; line 17 shows that `+` returns a value.

```
 1      life g = glider_matrix;
 2      ++g;                //g.operator++();
 3      g.print();
 4      (++g).print();      //g.operator++().print();
 5
 6      g++;                //g.operator++(0);
 7      g.print();          //Lines 7 and 8 should print the same picture.
 8      g++.print();        //g.operator++(0).print();
 9      g.print();          //Lines 8 and 9 should print different pictures.
10
11      g += 4;             //g.operator+=(4);
12      g.print();
13      (g += 4).print();   //g.operator+=(4).print();
14
15      life g2 = g + 4;    //life g2 = operator+(g, 4);
16      g2.print();
17      (g + 4).print();    //operator+(g, 4).print();
18
19      for (life g = glider_matrix;; g += 4) {
20          g.print();
21          ask the user if they want to fast-forward 4 generations;
22      }
```

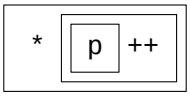
(5) Define an `operator==` friend of class `life` that will compare the `matrix` data members of the two `life` objects. It should ignore the `g` data members. We only want to know if the two objects contain the same picture; we don't care how many generations it took them to get there.

Define an `operator!=` that is neither a member nor a friend of class `life`. The `operator!=` will call `operator==`.

▲

Operator overloading and operator precedence

We considered the expression `*p++` on p. 7 when discussing operator precedence and associativity. The two operators have equal precedence and right-to-left associativity. Why did p. 7 lavish so much effort on establishing that the operator `++`, although postfix, will be executed *before* the `*`?



If the operands are objects, the operators will trigger function calls. The functions will be called in the same order as the operators were executed. If we do not know the rules of precedence and associativity, and the gaps in the rules, our programs will be impossible to debug.

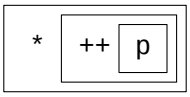
If `p` is an object, everyone expects that its postfix `operator++` will return another oject of the same class. Don't disappoint them. This second object is an anonymous temporary, whose `operator*` is then called. The computer behaves as if we had written the following expression.

```
1       p.operator++(0).operator*()
```

For another example of a call to a member function of an anonymous temporary object returned by a function, see line 2 on pp. 137−138.

Of course, `p` might be a pointer. In this case the expression `*p++` calls no functions.

Had the increment been prefix, it would still be executed before the `*`.

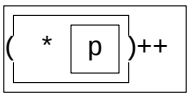


If `p` is an object, `operator++` would still be called first. But it would be the `operator++` that takes no arguments.

```
2       p.operator++().operator*();
```

If `p` is a pointer, the expression `*++p` would call no functions.

With parentheses, the `*` operator would be executed first.



This time there are four possibilities. If `p` and `*p` are objects, we would call `operator*` before the postfix `operator++`.

```
3       p.operator*().operator++(0);
```

If `p` is a pointer to an object, the `*` would be the built-in dereferencing operator.

```
4       (*p).operator++(0);
```

If `p` is an object whose `operator*` returns a non-`const` reference to a built-in, pointer, or enumeration, the `++` would be the built-in postincrement operator.

```
5       p.operator*()++;
```

If `p` is a pointer to a built-in, pointer, or enumeration, the expression `(*p)++` would call no functions.

▼ Homework 3.6b: vector arithmetic

A *vector* is a point in three-dimensional space. You can also think of it as an arrow from the origin to the point. Since the name `vector` is already used by the C++ Standard Library, we will name our class `vec`.* Define the class in a `vec.h` file. All of the followng functions will be inline, so there will be no

* We actually could name our class `vector` while also including the standard library headers. There are two ways to do this. If we say `using namespace std;`, we can refer to the standard library vector as `vector` and to our vector as `::vector`. Otherwise, we can refer to the standard library vector as `std::vector` and to ours as `vector`.

`vec.C`.

Give the class three private, non-static `double` data members named `x`, `y`, and `z`, and no other data members. Give it a public, three-argument constructor with a default value of zero for each argument.

Give it a public, non-static, `const` member function named `print` that will take no explicit arguments and return no value. `print` will output the object to `cout` with commas and parentheses: `(x, y, z)`. `vec.h` will have to include `<iostream>` and say `using namespace std` for `cout` and `<<`.

Give it a public, non-static, `const` member function named `length` that will return the length of the `vec`: $\sqrt{x^2 + y^2 + z^2}$. `vec.h` will have to include `<cmath>` and say `using namespace std` for the `sqrt` function.

Define `operator` functions for the following. Imitate what we did for class `date`.

(1) Make it possible to compare two `vec` objects: `v1 == v2`. Two `vec`'s are equal if `v1.x == v2.x`, `v1.y == v2.y`, etc. Also make it possible to say `v1 != v2`, but do not define the other four comparison operators.

(2) Make it possible to add one `vec` to another: `v1 += v2`. This will have the effect of adding `v2.x` onto `v1.x`; `v2.y` onto `v1.y`; etc. `operator+=` must return the new value of the left operand of the `+=` operator. Also make it possible to say `v1 + v2`, `v1 -= v2`, and `v1 - v2`.

`operator+` will take two `vec`'s as arguments, the first passed by value. The function must construct a new `vec`, and pass-by-value is the easiest way to do this.

(3) Make it possible to negate a `vec`: `-v`. This will construct and return a new `vec` whose `x` is the negative of `v.x`; whose `y` is the negative of `v.y`; etc.

(4) Make it possible to multiply a `vec` by a `double`: `v *= d`. This will have the effect of multiplying `v.x` by `d`; multiplying `v.y` by `d`; etc. `operator*=` must return the new value of the left operand of the `*=` operator. Also make it possible to say `v * d`, `d * v`, `v /= d` (check for division by zero), and `v / d`.

(5) The *dot product* of two `vec`'s `v1` and `v2` is a `double` whose value is

```
v1.x * v2.x + v1.y * v2.y + v1.z * v2.z
```

Make it possible to compute the dot product `v1 * v2`. (If the data members were one array of three `double`'s rather than three separate `double`'s, `operator*` could have called the algorithm `inner_product` in the standard library.) Do not define an `operator*=` for dot product.

(6) The *cross product* of two `vec`'s `v1` and `v2` is a `vec` whose `x` is `v1.y*v2.z - v1.z*v2.y`, and whose `y` is `v1.z*v2.x - v1.x*v2.z`, and whose `z` is `v1.x*v2.y - v1.y*v2.x`. The usual symbol for cross product is $\times$, but we will have to write a caret: `v1 ^ v2`. First define an `operator^` to construct and return the cross product. Then define an `operator^=` that will do its work by calling `operator^`. `operator^=` will be neither a member function nor a friend. Its first argument will be a read/write reference; its second will be a read-only reference. What would have gone wrong had we written `operator^=` first and had `operator^` call it?

`operator^` and the unary `operator-` will construct and return a new `vec` by saying `return vec(three arguments);`. The binary versions of `operator+`, `operator-`, `operator*`, and `operator/` will construct a new `vec` by copying an existing one. In each case, the new `vec` will automatically allocated, so it must be returned by value.

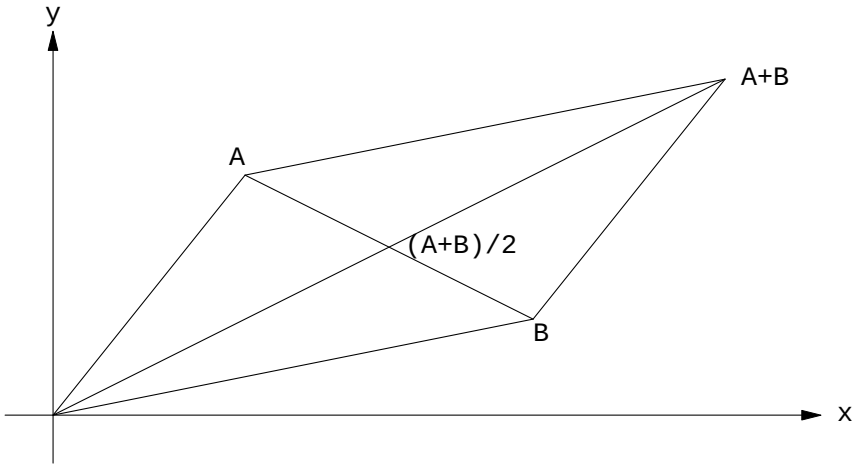
Demonstrate that your `operator` functions are correct by handing in your `vec.h` file together with the output of `http://i5.nyu.edu/~mm64/book/src/vec/main.C`.

Extra credit: invent a `matrix` class and multiply a `vec` and a `matrix`; the `operator*` will be a friend of both classes. For space cadets only: invent a `quaternion` class and multiply a `vec` and a `quaternion`.

▲

▼ **Homework 3.6c: midpoint between two point's**

A and B are vectors coming out of the origin. Their sum is A+B. Their average, (A+B)/2, extends only half as far from the origin as the sum, and is the midpoint of A and B.



The `midpoint` member function of class `point` was in lines 19−21 of `point.h` on pp. 201−202. Change it to the following function by writing an `operator+` and `operator/` that return `const point`. `operator/` will be a `const` member function of class `point`; `operator+` will be a friend. Now that `midpoint` no longer mentions any private members of class `point`, it does not need to be a member function or friend. Keep its definition in the `point.h` file, though, and keep it inline.

```
1 inline const point midpoint(const point& A, const point& B)
2 {
3     //Return the average.
4     return (A + B) / 2;            //return operator+(A, B).operator/(2);
5 }
```

▲

▼ **Homework 3.6d: fanciful `operator/` and `operator%` member function for class date**

Imagine two member functions of class `date` that would return the `date`'s year and day of the year in the range 0 to 264 inclusive. Why not name them `operator/` and `operator%`? With the argument 7, `operator%` could also return the `date`'s day of the week: 0 for Sunday, 1 for Monday, etc.

If the argument is any number other than 7 or 365, `operator%` will output an error message to `cerr` and `exit`.

```
 1     const char *const name[] = {
 2         "Sunday",
 3         "Monday",
 4         "Tuesday",
 5         "Wednesday",
 6         "Thursday",
 7         "Friday",
 8         "Saturday"
 9     };
10
11     date today;
12
13     cout << "Current year: " << today / 365 << "\n"   //today.operator/(365)
14         << "Julian date: " << today % 365 + 1 << "\n" //today.operator%(365)
```

```
15          << "Day of week: " << name[today % 7] << "\n";
```

▲

▼ **Homework 3.6e: midpoint between two `date`'s**

Change your definition of `midpoint` in Homework 2.11b ¶ (4) on pp. 211−212 to the following. Now that `midpoint` no longer mentions any private members of class `point`, it does not need to be a member function or friend of the class.

Both operands of the `-` in line 7 are `date`'s, so this `-` calls the `operator-` friend defined in line 32 of `date.h` rather than the neither-member-nor-friend in line 54.

```
 1 //Declared in date.h, defined in date.C
 2
 3 const date midpoint(const date& d1, const date& d2)
 4 {
 5     //return (d1 + d2) / 2;   //average of two dates
 6
 7     div_t d = div(d1 - d2, 2);
 8     if (d.rem < 0) {
 9         --d.quot;
10     }
11     return d2 + d.quot;
12 }
```

I wish the body of this function could simply be the above line 5. But this line will not compile, because we have written no `operator+` that will accept two `date`'s, To do so would expose the values of the private data member(s). For example, assume that `d1` and `d2` were `date`'s early in the year 2014. If each `date` had one private data member `day` giving the number of days since January 1, 0 A.D. then `d1 + d2` would be a `date` in the year 4028. But if each `date` had one private data member `day` giving the number of days since January 1, 1970 A.D. then `d1 + d2` would be a `date` in the year 2058.

▲

A fictitious but useful intermediate result

> When a newcomer enters the field and finds himeself confronted by the nuances of international questions, he becomes an easy target for the military-CIA-paramilitarry-type answers which can be added, subtracted, multiplied, or divided.
>
> —Chester Bowles, *Promises to Keep*, p. 330

Adding the two `date`'s in the above line 5 would be so useful that I am tempted to make it possible. But we must take care that the sum of two `date`'s is never output to the user. It will be strictly an intermediate result, like the unsightly infinities that are ''renormalized'' away in quantum theory.

We will use the class `date` with one non-static data member, the `int day` in line 27 of the following `date.h`.

The sum of two `date` objects will be a `timebomb` object. A `timebomb` must eventually be defused by dividing it by 2, i.e., by calling its `operator/` member function. If we do not, the `timebomb`'s destructor will give us an error message. We saw similar error checking in the destructor for class `stack` in lines 6−13 of `stack.C` on p. 150. Remember the warning about not incinerating aerosol cans that still have pressure?

Classes `date` and `timebomb` will always be used together, so they share the same header file. The forward declaration in line 6 allows 19 and 20 to mention the name `date`; see pp. 465−466.

To ensure that a `timebomb` can be constructed only by adding together two `date`'s, its constructor is private in line 11. Only the member functions and friends of class `timebomb` will be able to call this constructor. And the only friend of this class is the `operator+` that adds together two `date`'s (line 20).

This `operator+` must also be a friend of class `date`, since it mentions a private member of `date` (line 43). It is our first example of a friend of more than one class.

Human beings will construct a `date` by calling the three-argument constructor in line 35. The `operator/` will construct a `date` by calling the one-argument constructor in line 28. To ensure that this constructor will be called only by `operator/`, the constructor is private and the `operator/` is a friend of class `date`. It is our first example of a function that is a member of one class and a friend of another.

Class `timebomb` had to be defined before class `date`, since class `date` mentions a member of `timebomb` (line 39). If both classes had mentioned a member of the other, we would have been forced to throw in the towel and let *every* member function of both classes be a friend of the other class. This can be done without mentioning the names of the individual member functions; see p. 467.

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

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class date;
 7
 8 class timebomb {
 9     int sum;
10     bool ticking;   //true if this timebomb is still ticking
11     timebomb(int initial_sum): sum(initial_sum), ticking(true) {}
12 public:
13     ~timebomb() {
14         if (ticking) {
15             cerr << "forgot to divide the sum of two dates by 2\n";
16         }
17     }
18
19     const date operator/(int n);
20     friend timebomb operator+(const date& d1, const date& d2);
21 };
22
23 class date {
24     static const int length[];
25     static const int    pre[];
26
27     int day;
28     date(int initial_day): day(initial_day) {}
29 public:
30     enum month_type {
31         january = 1, february, march, april, may, june,
32         july, august, september, october, november, december
33     };
34
35     date(int month, int day, int year);
36     void print() const;
37
38     friend timebomb operator+(const date& d1, const date& d2);
39     friend const date timebomb::operator/(int n);
40 };
41
```

```
42 inline timebomb operator+(const date& d1, const date& d2) {
43     return d1.day + d2.day;
44 }
45
46 inline date midpoint(const date& d1, const date& d2) {
47     return (d1 + d2) / 2;   //return operator+(d1, d2).operator/(2);
48 }
49 #endif
```

The above line 43 calls the constructor for class `timebomb`, behaving as if we had said the following.

```
50          return timebomb(d1.day + d2.day);
```

Another example of an implicit constructor call is in line 20 of the following `timebomb.C`. See p. 138.

The `operator+` could have been defined within the {curly braces} of class `date`. We simply replace the declaration in line 38 with the definition in lines 42−44, changing the keyword `inline` to `friend`. But we left the definition where it is, since the function is a friend of more than one class.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <ctime>
 4 #include "date.h"
 5 using namespace std;
 6
 7 const int date::length[] = {
 8     0,                       //dummy element to give january subscript 1
 9     31, 28, 31, 30, 31, 30,
10     31, 31, 30, 31, 30, 31
11 };
12
13 const int date::pre[] = {
14     0,                       //dummy element to give january subscript 1
15     0,                       //january
16     pre[ 1] + length[ 1],    //february
17     pre[ 2] + length[ 2],    //march
18     pre[ 3] + length[ 3],    //april
19     pre[ 4] + length[ 4],    //may
20     pre[ 5] + length[ 5],    //june
21     pre[ 6] + length[ 6],    //july
22     pre[ 7] + length[ 7],    //august
23     pre[ 8] + length[ 8],    //september
24     pre[ 9] + length[ 9],    //october
25     pre[10] + length[10],    //november
26     pre[11] + length[11]     //december
27 };
28
29 date::date(int month, int d, int year)
30 {
31     //Error checking omitted for brevity.
32     day = 365 * year + pre[month] + d - 1;
33 }
34
35 void date::print() const
```

```
36 {
37     div_t divide = div(day, 365);
38     if (divide.rem < 0) {
39         divide.rem += 365;
40         --divide.quot;
41     }
42 
43     int d = divide.rem + 1;     //Julian date (1 to 365)
44     int month;                  //uninitialized variable
45 
46     for (month = january; d > length[month]; ++month) {
47         d -= length[month];
48     }
49 
50     cout << month << "/" << d << "/" << divide.quot;
51 }
```

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5 
 6 const date timebomb::operator/(int n)
 7 {
 8     if (n != 2) {
 9         cerr << "sum of 2 dates must be divided by 2, not by "
10             << n << "\n";
11         exit (EXIT_FAILURE);
12     }
13 
14     ticking = false;
15 
16     div_t d = div(sum, 2);
17     if (d.rem < 0) {
18         --d.quot;
19     }
20     return d.quot;
21 }
```

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5 
 6 int main()
 7 {
 8     const date d1(date::january, 1, 2014);
 9     const date d2(date::january, 3, 2014);
10     const date d3(date::january, 4, 2014);
11 
```

```
12     midpoint(d1, d2).print(); cout << "\n";   //even distance
13     midpoint(d2, d1).print(); cout << "\n";
14
15     midpoint(d1, d3).print(); cout << "\n";   //odd distance
16     midpoint(d3, d1).print(); cout << "\n";
17
18     return EXIT_SUCCESS;
19 }
```

```
1/2/2014      line 12: left operand is the earlier date, even distance
1/2/2014      line 13: right operand is the earlier date
1/2/2014      line 15: left operand is the earlier date, odd distance
1/2/2014      line 16: right operand is the earlier date
```

3.7 `operator()` for an Object that Does Only One Thing

An object with only one public member function, not counting the constructor and destructor, does only one thing for us. For example, the class `myrandom` had only one member function, the `rand` in lines 3–8 of `myrandom.C` on p. 176.

When an object does only one thing, give the member function the admittedly strange name `operator()`. The parentheses are part of the function name. Change line 10 of `myrandom.h` on p. 176 to

```
1     int operator()();
```

The first pair of parentheses are part of the function name; the second pair surround the empty argument list. Change line 3 of `myrandom.C` on p. 176 to

```
2 int myrandom::operator()()
```

We can then call this member function simply by applying the `()` operator to its object. When we write the expression `r1()` in line 7, the computer will behave as if we had written the expression `r1.operator()()` in the comment. The first pair of parentheses is part of the name of the member function `operator()`; the second pair is the empty argument list.

```
3     myrandom r1;
4     myrandom r2(2014);
5
6     for (int i = 0; i < 3; ++i) {
7         cout << r1() << "\n"        //cout << r1.operator()() << "\n"
8              << r2() << "\n";       //     << r2.operator()() << "\n";
9     }
```

`r1` and `r2` look like functions. But they are really objects, and we can construct as many of them as we need. An object endowed with an `operator()` to make it look like a function is called a *function object*. Such objects can be used to customize the ‘‘algorithms’’ in the C++ Standard Library; our first example will be on pp. 764–770.

3.8 Initialization vs. Assignment: a Constructor vs. operator=

Initialization puts the first value into a new variable. *Assignment* puts a new value into an existing variable. We saw on p. 262 that for a variable of a built-in data type, a pointer, or an enumeration, there is little difference between these two operations.

But for an object, initialization and assignment may do very different jobs even though they may still be written with the same symbol. We will begin with classes `date` and `stack`, where initialization and

assignment just happen to do the same job. But this will be the exception rather than the rule.

An object is initialized by calling its constructor; for an object, *initialization* and *construction* mean the same thing. In a declaration, a constructor with one argument can be called with an equal sign. A copy constructor, for example, always has one argument as in line 2.

The symbol = is also used for assignment to an object. But this = calls a different member function, one named `operator=`. When we write line 3, for example, the computer behaves as if we had written the corresponding comment. The right operand `d1` is passed to `operator=` by reference to avoid constructing an unnecessary copy of it; the reference is read-only to ensure that `d1` cannot be damaged by the `operator=`.

```
1     date d1(date::april, 8, 2014); //initialization: call 3-arg constructor
2     date d2 = d1;                  //initialization: call copy constructor
3     d2 = d1;                       //assignment: d2.operator=(d1);
```

Two member functions defined for us implicitly

An added complication (or simplification, depending on you point of view) is that the computer will define these two member functions for us if we have not defined them ourselves. The first is the constructor whose argument is another object of the same class, i.e., the copy constructor. The second is the `operator=` whose argument is another object of the same class. A class may have several `operator=`'s, each with an argument of a different type.

For example, we never defined a copy constructor for class `date` since we were satisfied with the computer's. But the above line 2 compiles anyway. When it calls the copy constructor, the computer behaves as if we had defined and called the following public copy constructor. Assume that the class has the three original data members `year`, `month`, and `day`.

```
1 //Copy the non-static data members from the other object to this one,
2 //in the order in which they were declared.
3
4 date::date(const date& another)
5     : year(another.year), month(another.month), day(another.day)
6 {
7 }
```

Similarly, we never defined a member function `operator=` for class `date` since we were satisfied with the one the computer provided for us. But line 3 of the previous fragment compiles anyway. When it calls `operator=`, the computer behaves as if we had defined and called the following public `operator=`. It returns `*this` by reference, just like the `operator+=` and prefix `operator++` for class `date` in lines 40 and 47. of `date.h` on pp. 273−274.

```
 8 date& date::operator=(const date& another)
 9 {
10     //Copy the non-static data members from the other object to this one,
11     //in the order in which they were declared.
12
13     year = another.year;
14     month = another.month;
15     day = another.day;
16
17     return *this;
18 }
```

The `operator=` in line 3 of the previous fragment is a member function of the object `d2`, so the value returned by the `return *this` in the above line 17 is the new value of `d2`. This is used as the value of the expression `d2 = d1`. Since this expression has a value, we could use it as the right operand of another assignment expression. And since that expression has a value, we could use it as the right

operand of yet another.

```
19              d2 = d1;  //                          d2.operator=(d1);
20          d3 = d2 = d1;  //             d3.operator=(d2.operator=(d1));
21     d4 = d3 = d2 = d1;  //d4.operator=(d3.operator=(d2.operator=(d1)));
```

Operator overloading gives us a nice, linear notation that hides the nested function calls. The = operator has right-to-left associativity, so the first function to be called is the one that implements the rightmost =.

Compare the hidden nesting of the `operator<<`'s on p. 340.

For class `date`, the implicitly defined copy constructor and `operator=` are good enough for us. But for more complicated classes we will have to write them ourselves.

Our copy constructor and `operator=` can be faster than the computer's

Here is a class where the implicitly defined copy constructor and `operator=` are not good enough for us. We will have to write them ourselves. In this simple class, the two functions will still do the same job.

We did not define a copy constructor and `operator=` for the class `stack` on pp. 149−154, so they were defined implicitly.

```
 1     stack s1;        //initialization: call the default constructor
 2     s1.push(10);
 3     cout << s1.pop() << "\n";
 4
 5     stack s2 = s1;   //initialization: call the copy constructor
 6     s2 = s1;         //assignment: s2.operator=(s1);
```

Assume that class `stack` has the two original non-static data members, `a` and `n`. The constant `stack_max_size` was renamed `max_size` when it became a static data member in Homework 2.14.1b, ¶ (3) on p. 239. `max_size` and `n` have been given their correct data type, `size_t`. And we have followed the C++ convention of creating a `value_type` typedef (pp. 153−154).

```
 7 typedef int value_type;
 8
 9 class stack {
10     static const size_t max_size = 100;
11     value_type a[max_size];
12     size_t n;
13     //etc.
```

The = in the above line 5 will call the following copy constructor, which was defined implicitly.

```
14 stack::stack(const stack& another)
15     : n(another.n)
16 {
17     for (size_t i = 0; i < max_size; ++i) {
18         a[i] = another.a[i];
19     }
20 }
```

And the = in the above line 6 will call the following `operator=` member function, also defined implicitly.

```
21 stack& stack::operator=(const stack& another)
22 {
23     n = another.n;
24
25     for (size_t i = 0; i < max_size; ++i) {
26         a[i] = another.a[i];
```

```
27      }
28
29      return *this;
30  }
```

But both of these functions have a performance bug: they do more copying than is necessary. We can define faster ones that loop only as far as they have to.

```
31  //Excerpt from stack.C.
32
33  stack::stack(const stack& another)
34      : n(0)
35  {
36      for (; n < another.n; ++n) {
37          a[n] = another.a[n];
38      }
39  }
40
41  stack& stack::operator=(const stack& another)
42  {
43      if (&another != this) {
44          for (n = 0; n < another.n; ++n) {
45              a[n] = another.a[n];
46          }
47      }
48
49      return *this;
50  }
```

The two objects in the following assignment are the same object.

```
51      stack s;
52      s.push(10);
53      s = s;           //self-assignment: s.operator=(s);
```

In this case, we must not execute the `n = 0` in the above line 44. Fortunately, our `operator=` will do nothing thanks to the `if` in the above line 43.

Admittedly, no one will write the self-assignment in the above line 53. But there are less obvious ways in which the same object might be used as both operands of an assignment. If, for example, `p` and `q` are pointers to objects, we might accidentally say

```
54      *p = *q;         //(*p).operator=(*q);
```

when `p` and `q` point to the same object.

A constructor and `operator=` that must do different jobs

For classes `date` and the original `stack`, the copy constructor and `operator=` did the same job: they merely copied the data from one object to another. (`operator=` also returned `*this` by reference). Now we will see a class whose constructor and `operator=` must do different jobs.

Imagine a class `outputfile` whose constructor opens an output file (line 3) and whose destructor closes it (line 8). In between, there are member functions such as `writeline` for writing to the file (line 4). The C++ Standard Library has a similar class named `ofstream` (“output file stream”).

When line 3 initializes an object, we are calling its constructor. The constructor called in line 3 does only one job: it opens `outfile1`.

When line 6 assigns to the object, we are calling its `operator=`. But the `operator=` called in line 6 does *two* jobs: it closes `outfile1` and opens `outfile2`. `outfile2` will eventually be closed

by the destructor in line 8.

```
1 void f()
2 {
3     outputfile out = "outfile1";   //Initialization: call a constructor.
4     out.writeline("hello");        //Write one line to outfile1.
5
6     out = "outfile2";              //Assignment: out.operator=("outfile2");
7     out.writeline("goodbye");      //Write one line to outfile2.
8 }                                  //Destruction.
```

Assignment is usually more expensive than initialization because it does more work. For class `outputfile`, the `operator=` did the work of the destructor, followed by the work of a constructor.

A class for which we must write our own copy constructor and operator=

To show how to write a class whose constructors and `operator=` must do different jobs, we will invent our own class of string objects. In real life, though, we would never write this class. We would just use the class `string` in the C++ Standard Library.

A C program would store a string of characters into an array of `char`. But there are two drawbacks to this approach: an array cannot grow and shrink as the program runs, and its subscripts are not range checked.

Our string objects will be free from these flaws. To permit a string to grow without bounds, we will not attempt to store the characters in the object itself. They will be stored offshore, in a dynamically allocated buffer. Each object will have its own buffer, and will contain a pointer to the start of its buffer. For the present, the buffer will be allocated with the C functions `malloc` and `free`. Later we will use the corresponding C++ operators `new` and `delete`.

We must initialize the data member `n` before `p`, since the value of `n` is used in the initial value of `p` in lines 8 and 19 of `mystring.C` below. See pp. 113−114 for the order of initialization for data members.

We will talk about the `operator[]` and `operator<<` functions in lines 19−26 later. First we will demonstrate why we had to write our own copy constructor and `operator=` for this class.

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

```
 1 #ifndef MYSTRINGH
 2 #define MYSTRINGH
 3 #include <iostream>   //for ostream and <<
 4 #include <cstddef>    //for size_t
 5 #include <cstdlib>    //for free
 6 using namespace std;
 7
 8 class mystring {
 9     size_t n;   //number of characters, not counting the terminating '\0'
10     char *p;    //pointer to the 1st character; must be constructed after n
11 public:
12     mystring(const mystring& another);   //the copy constructor
13     mystring(const char *s = "");
14     ~mystring() {free(p);}
15
16     mystring& operator=(const mystring& another);
17     mystring& operator=(const char *s);
18
19     char& operator[](size_t i);
20     const char& operator[](size_t i) const;
21
```

```
22      void print() const {cout << p;}
23
24      friend ostream& operator<<(ostream& ost, const mystring& m) {
25          return ost << m.p;
26      }
27  };
28  #endif
```

`malloc` is called whenever we put a value into a `mystring`, in lines 8, 19, 34, and 51. We always allocate one extra byte for the `'\0'` at the end of a string of characters. `malloc` returns a `void *`, which we store into the `char *` data member `p`. C would let us implicitly convert a `void *` to a pointer to a variable, but C++ requires an explicit `static_cast`. The conversions and casts will disappear on p. 395, when we switch from `malloc` to `new`.

A bug lurks in the `operator=` in line 47−60. Try to find it before we fix it on pp. 314−315.

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

```
 1  #include <iostream>
 2  #include <cstdlib>   //for malloc, size_t, exit, EXIT_FAILURE
 3  #include <cstring>   //for strcpy, strlen
 4  #include "mystring.h"
 5  using namespace std;
 6
 7  mystring::mystring(const mystring& another)   //the copy constructor
 8      : n(another.n), p(static_cast<char *>(malloc(n + 1)))
 9  {
10      if (p == 0) {
11          cerr << "couldn't allocate " << n + 1 << " bytes\n";
12          exit(EXIT_FAILURE);
13      }
14
15      strcpy(p, another.p);
16  }
17
18  mystring::mystring(const char *s)
19      : n(strlen(s)), p(static_cast<char *>(malloc(n + 1)))
20  {
21      if (p == 0) {
22          cerr << "couldn't allocate " << n + 1 << " bytes\n";
23          exit(EXIT_FAILURE);
24      }
25
26      strcpy(p, s);
27  }
28
29  mystring& mystring::operator=(const mystring& another)
30  {
31      if (&another != this) {
32          free(p);
33          n = another.n;
34          p = static_cast<char *>(malloc(n + 1));
35
36          if (p == 0) {
37              cerr << "couldn't allocate " << n + 1 << " bytes\n";
38              exit(EXIT_FAILURE);
```

```
39          }
40
41          strcpy(p, another.p);
42      }
43
44      return *this;
45  }
46
47  mystring& mystring::operator=(const char *s)
48  {
49      free(p);
50      n = strlen(s);
51      p = static_cast<char *>(malloc(n + 1));
52
53      if (p == 0) {
54          cerr << "couldn't allocate " << n + 1 << " bytes\n";
55          exit(EXIT_FAILURE);
56      }
57
58      strcpy(p, s);
59      return *this;
60  }
61
62  char& mystring::operator[](size_t i)
63  {
64      if (i > n) {
65          cerr << "Subscript " << i << " must be in range 0 to " << n
66              << " inclusive.\n";
67          exit(EXIT_FAILURE);
68      }
69
70      return p[i];
71  }
72
73  const char& mystring::operator[](size_t i) const
74  {
75      if (i > n) {
76          cerr << "Subscript " << i << " must be in range 0 to " << n
77              << " inclusive.\n";
78          exit(EXIT_FAILURE);
79      }
80
81      return p[i];
82  }
```

The above lines 33−36 may be combined to

```
83          if ((p = static_cast<char *>(malloc((n = another.n) + 1))) == 0) {
```

and lines 50−53 may be combined to

```
84          if ((p = static_cast<char *>(malloc((n = strlen(s)) + 1))) == 0) {
```

But don't do it. C++ does not share C's rage to cram as much code as possible into a single expression.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/mystring/main1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "mystring.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     mystring s = "hello"; //initialization: constructor, mystring.C l. 18
 9     mystring t = s;       //initialization: copy constructor, mystring.C l.7
10
11     cout << "s and t were initialized to the values \"";
12     s.print();
13     cout << "\" and \"";
14     t.print();
15     cout << "\".\n";
16
17     s = "goodbye"; //assignment: s.operator=("goodbye"); mystring.C l. 47
18     t = s;         //assignment: t.operator=(s); mystring.C l. 29
19
20     cout << "s and t were assigned the values \"";
21     s.print();
22     cout << "\" and \"";
23     t.print();
24     cout << "\".\n";
25
26     return EXIT_SUCCESS;
27 }
```

```
s and t were initialized to the values "hello" and "hello".
s and t were assigned the values "goodbye" and "goodbye".
```

We must write our own copy constructor to avoid Siamese twins

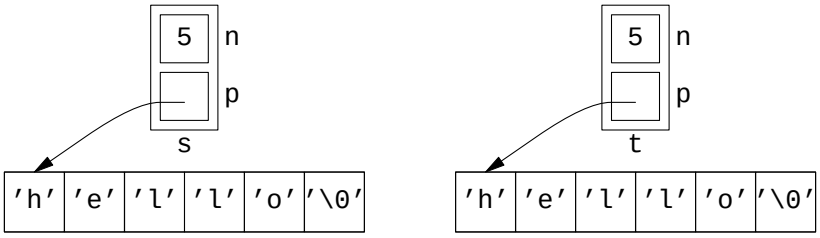
`s` and `t` start with the same value (`"hello"`), but line 11 changes one of them.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "mystring.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     mystring s = "hello";   //the constructor that takes a const char *
 9     const mystring t = s;   //the copy constructor of t
10
11     s[0] = 'H';             //s.operator[](0) = 'H'; in mystring.C line 62
12
13     cout << "s == \"";
14     s.print();
15     cout << "\"\n";
16
17     cout << "t == \"";
```

```
18      t.print();
19      cout << "\"\n";
20
21      return EXIT_SUCCESS;    //Call the destructors for t and s.
22  }
```

After the above line 9 finishes calling the copy constructor in line 7 of `mystring.C`, we have two cleanly separated objects.



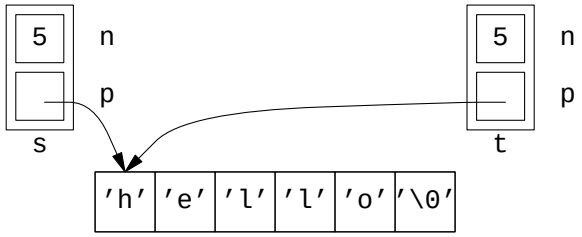
The above line 21 will call the destructors for `t` and `s`, in that order. When `t`'s destructor frees the block of memory pointed to by `t.p`, it will have no effect on the block of memory pointed to by `s.p`. They are two different blocks.

```
s == "Hello"
t == "hello"
```

Had we not written the copy constructor in line 7 of `mystring.C`, the above line 9 would behave as if we had written the following copy constructor.

```
23  mystring::mystring(const mystring& another)
24      : n(another.n), p(another.p)
25  {
26  }
```

It is okay for the above line 24 to copy the `n` data member from one object to another. But when the line does the same for `p`, it turns `s` and `t` into Siamese twins.



Any change to the characters of `s` would therefore have the same effect on `t`. For example, the above line 11 would also change `t` to `"Hello"`.

```
s == "Hello"
t == "Hello"
```

When the above line 21 destructs `t`, something much worse would happen: `t` would drag down `s` with it. The destructor for `t` frees the block of memory pointed to by `t.p`, but this would be the same block as the one that is pointed to by `s.p`. Then when line 21 tries to destruct `s`, it would free the same block of memory again, corrupting the heap of memory doled out by `malloc`. If we are lucky, there might be an error message about the twice-freed block.

We must write our own `operator=` to avoid Siamese twins

Lines 10 and 11 change the original values of `s` and `t`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/mystring/main3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "mystring.h"
 4
 5 int main()
 6 {
 7     mystring s = "hello";
 8     mystring t = "goodbye";
 9
10     s = t;         //s.operator=(t);       in line 29 of mystring.C
11     t = "Hello";   //t.operator=("Hello"); in line 47 of mystring.C
12
13     cout << "s == \"";
14     s.print();
15     cout << "\"\n";
16
17     cout << "t == \"";
18     t.print();
19     cout << "\"\n";
20
21     return EXIT_SUCCESS;
22 }
```

After the above line 10 finishes calling the `operator=` in line 29 of `mystring.C`, we have two cleanly separated objects. Line 11 changes one of them.

```
s == "goodbye"
t == "Hello"
```

Had we not written the `operator=` in line 29 of `mystring.C`, the above line 10 would behave as if we had written the following `operator=`:

```
23 mystring& mystring::operator=(const mystring& another)
24 {
25     n = another.n;
26     p = another.p;
27
28     return *this;
29 }
```

At line 11 the pointers `s.p` and `t.p` would then both point to `"goodbye"` and we would again have Siamese twins. No pointer would point to `"hello"`. `"hello"` would never be freed, leaving us with a memory leak. Meanwhile, line 11 would damage the value of `s`. Finally, when line 21 destructs `t`, `t` would once again drag `s` down with it.

```
s == "Hello"
t == "Hello"
```

The difference between a copy constructor and operator=

An `operator=` must do three things over and above what a copy constructor does:

(1) `operator=` must destruct the old value of the object that is receiving the new value; see the `free` in line 32 of the `mystring.C` on pp. 304–305. But there is no old value for a constructor to destruct, since a constructor is erecting a new value on virgin territory.

(2) No one will ever say

```
1     mystring a = a;
```

or if they do, they deserve to be punished. A copy constructor can safely assume that the old and new objects are not the same object. But we might say

```
2     *p = *q;                          //(*p).operator=(*q);
```

when `p` and `q` point to the same object. An `operator=` must therefore check that the object of which it is a member, and the object that it receives as an argument, are indeed two different objects. This is done by checking that the two objects occupy different addresses; see the `if (&another != this)` in line 31 of `mystring.C`. Were the `free` in line 32 of `mystring.C` *not* inside the `if` in line 31, the expression `*p = *q` would destroy the object that `p` and `q` point to.

On the other hand, an `operator=` does not need this `if` when the data type of the argument differs from that of the object to which it belongs. For example, the `operator=` in line 47 of `mystring.C` has an argument that is not an object. In this case, there is no possibility that the object of which the `operator=` is a member, and the argument of the `operator=`, are the same object. The argument of the `operator=` is not an object at all; another example is on p. 735. (There is still a bug in this `operator=`; see pp. 314–315.)

(3) `operator=` must return (by reference) the new value of the object that received the new value; see the declaration in line 47 and the `return *this;` in line 59 of `mystring.C`. The last statement of an `operator=` must always be `return *this;`, just like the last statement of an `operator+=` or a prefix `operator++`.

The big three

If we need to write any one of the following member functions, we probably need to write all three:

(1) destructor

(2) copy constructor

(3) `operator=`

Here are examples of classes which need two or more of the above member functions.

(1) Class `counted` on pp. 241–244 has a static data member that needs to be updated whenever an object is constructed or destructed. We had to write a copy constructor and a destructor for this class, but not an `operator=`.

(2) Class `mystring` on pp. 303–308 has a non-static data member pointing to data located outside the object but which is owned by the object. We had to write a copy constructor and `operator=` to avoid Siamese twins, and a destructor to avoid memory leaks.

(3) Classes `rabbit` and `wolf` on pp. 194–197 and 197–199 have constructors and destructors that must call member functions of other objects. We deliberately declared undefined private copy constructors for these classes. Ditto for class `node` on pp. 212–217, whose constructor and destructor change the data members of other objects.

Four public member functions implicitly defined

Here are the four public member functions that will be implicitly defined for us if we don't write them ourselves. If we're not satisfied with them, we can define different versions explicitly. We demonstrate with class `mystring` on pp. 303–308.

(1) The copy constructor. The computer will write a public copy constructor that simply calls the copy constructor of each non-static data member. The data members will be copied in the order in which they were declared. The data members will be constructed in the order in which they were declared. If a data member is a built-in type, pointer, or enumeration, we can program as if it has a public copy constructor. For example,

```
mystring::mystring(const mystring& another)
    : n(another.n), p(another.p)    //bug: Siamese twins
{
}
```

The computer will not write a copy constructor for a class that has a non-static data member with no public copy constructor of its own.

(2) The default constructor. If we have written no other constructor, the computer will write a public default constructor that simply calls the default constructor of each non-static data member. The data members will be constructed in the order in which they were declared. If a data member is a built-in type, pointer, or enumeration, we can program as if it has a public default constructor that leaves it full of garbage. For example,

```
//n is built-in, p is pointer.
//Bug: their default constructors leave them full of garbage.

mystring::mystring()
{
}
```

The computer will not write a default constructor for a class that has a non-static data member with no public default constructor of its own.

(3) The destructor. The computer will write a public destructor that simply calls the destructor of each non-static data member. The data members will be destructed in the opposite order from that in which they were declared. If a data member is a built-in type, pointer, or enumeration, we can program as if it has a public destructor that does nothing. For example,

```
//n is built-in, p is pointer.
//Bug: memory leak.

mystring::~mystring()
{
}
```

The computer will not write a destructor for a class that has a non-static data member with no public destructor of its own.

(4) The `operator=` member function whose argument is a constant reference to another object of the same class as this one. The computer will write a public `operator=` that simply calls the `operator=` of each non-static data member. The data members will be assigned to in the order in which they were declared. Finally, the `operator=` will return `*this`. If a data member is a built-in type, pointer, or enumeration, we can program as if it has a public `operator=` taking an argument of the same type. For example,

```
mystring& mystring::operator=(const mystring& another)
{
    n = another.n;
    p = another.p;                  //bug: Siamese twins

    return *this;
}
```

The computer will not write a `operator=` for a class that has a non-static data member with no public

`operator=` of its own.

The only `operator=` that the computer will write for us is one whose argument is a `const` reference to another object of the same class. If we want a different argument, we must always write the `operator=` ourselves. For example, the computer would gladly have written us a(n incorrect) `mystring::operator=` taking a `const mystring&`, but we had to write the `mystring::operator=` taking a `const char *`.

▼ Homework 3.8a: define an `operator=` for the pointer version of class `stack`

Write an `operator=` for the class `stack` with a pointer data member on pp. 152–153. Return `*this` by reference.

This `operator=` is long overdue. Let's hope no one has attempted to assign one `stack` to another.

▲

▼ Homework 3.8b: rewrite `point::assign` as an `operator=`

Class `point` has a member function named `assign` in line 21 of `point.h` on p. 207. Rename it `operator=`. The `operator=` does not need the `if` that ensures that the two `point`'s are two different objects. Return `*this` by reference.

▲

▼ Homework 3.8c: define an `operator=` for class `obj`

Write an `operator=` for class `obj` on pp. 179–180 that will output the string `"operator= "` and the value of the data member `i`. Output another message if the object of which it is a member, and the object that it receives as an argument, are the same object. Return `*this` by reference.

The `operator=` will be too long to be inline. But make it inline anyway so we won't have to create an `obj.C` file.

▲

▼ Homework 3.8d: make it impossible to assign one animal to another

We have already made it impossible to create an animal that is a copy of another (p. 200). Let's also ensure that no animal can be assigned to another:

```
1     rabbit r1(argument(s) for constructor);
2     rabbit r2(argument(s) for constructor);
3
4     r1 = r2;                          //Let's make this illegal.
```

A C++ object can be assigned to only by its `operator=` member functions. To make it impossible to assign one object to another of the same class, all we have to do is make sure that it has no `operator=` whose argument is another object of the same class. In fact, we wrote no such `operator=` for classes `rabbit` and `wolf`. But for that very reason, the computer wrote them for us. See p. 300.

To prevent the computer from doing this, declare a private `operator=` for class `rabbit` whose argument is a read-only reference to a `rabbit`, and one for class `wolf` whose argument is a read-only reference to a `wolf`, but do not define them. If a member function or friend of one of these classes tries to call the `operator=` for that class, the program will not link because the `operator=` was never defined. And if any other function tries to call the `operator=`, the program will not even compile because the `operator=` is private. In either case, it will be impossible to assign one animal to another of the same class.

```
1 class rabbit {
2     static const char c = 'r';
3     const terminal *t;
4     unsigned x, y;
5
```

```
6     rabbit& operator=(const rabbit& another);  //deliberately undefined
7 public:
```

While you're at it, go to class `node` on pp. 212−217 and declare a private, undefined `operator=` whose argument is a read-only reference to a `node`.
▲

An implicit call to a constructor

Line 2 calls the `operator=` in line 47 of the above `mystring.C`:

```
1     mystring s = "hello";
2     s = "goodbye";                  //s.operator=("goodbye");
```

But even if we had never written this function, line 2 would still work. The computer would behave as if we had written

```
3     mystring s = "hello";
4     s = mystring("goodbye");        //s.operator=(mystring("goodbye"));
```

In other words, it would call the constructor in line 18 of `mystring.C`, and then pass the newly-constructed object to the `operator=` in line 29 of `mystring.C`. We wrote the `operator=` in line 47 to avoid the construction of this extra object in the above line 2.

Had we not written the `operator=`, the above line 2 would have contained an *implicit* call to the constructor. Line 4 contains an *explicit* call to the constructor.

To prevent an implicit constructor call from compiling, add the keyword `explicit` to the start of the declaration for the constructor in line 13 of `mystring.h`. We would do this to make sure that the above line 2 never constructs the unwanted object.

3.9 `operator[]` Returns a Reference

When we apply a subscript in [square brackets] to an object, the computer behaves as if we had called the `operator[]` member function of that object. The subscript that we wrote in the square brackets is passed to the `operator[]` function as its argument.

The test in lines 64 and 75 of the above `mystring.C` will catch a subscript that is too large. It will also catch a negative subscript, because it will have been converted to a large positive number when copied into the `size_t` argument of `operator[]`.

If your object contains many items of data, and each item has an identifying number (subscript), the name you should use for the member function that accesses them is `operator[]`. This dresses the object up to look like an array.

The `operator[]` function needs to access the private members of class `mystring`. It must therefore be either a member function or a friend of that class. But we have no choice in this matter—by the rule on p. 287, ¶ (2), `operator[]` must be a member function.

Why the operator[] in line 62 of mystring.C can return a reference

Recall that the `operator+` and the postfix `operator++` in lines 44 and 50 of `date.h` on p. 274 constructed new objects, which had to be returned by value. But the `operator[]` in line 62 of `mystring.C` on pp. 304−305 does not construct a new `char`. It returns an existing `char`, so the `char` can be returned by reference.

Why the operator[] in line 62 of mystring.C must return a reference

An expression that can be the left operand of the assignment operator `=` is called an *lvalue;* one that can be the right operand is called an *rvalue.* For example, a variable could be an lvalue or an rvalue.

```
1     x = y;
```

A literal can be an rvalue but not an lvalue:

```
2     x = 10;             //10 is a literal.
```

In C, the return value of a function can not be an lvalue:

```
3     y = sqrt(x);        //sqrt(x) can be an rvalue.
4     sqrt(x) = y;        //won't compile: sqrt(x) can not be an lvalue.
5     cout << &sqrt(x);   //won't compile: sqrt(x) can not be an lvalue.
```

But that's exactly what line 11 `main4.C` is trying to do: use the return value of a function as an lvalue. The comment beside line 11 exposes the misdeed. Is there any way to make this legal?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/mystring/main4.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "mystring.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     mystring s = "hello";
 9     cout << s[0] << "\n";   //cout << s.operator[](0) << "\n";
10
11     s[0] = 'H';             //s.operator[](0) = 'H';
12     cout << s[0] << "\n";   //cout << s.operator[](0) << "\n";
13
14     const mystring t = "goodbye";
15     cout << t[0] << "\n";   //cout << t.operator[](0) << "\n";
16     //t[0] = 'G';           //won't compile: t.operator[](0) = 'G';
17
18     return EXIT_SUCCESS;
19 }
```

```
h
H
g
```

If a function returns a pointer, the return value with an asterisk in front of it can be an lvalue. See line 18 of `return_int.C` on p. 75. If a function returns a reference, the return value can be an lvalue, even without an asterisk in front of it. See line 19 of the same program.

To be an lvalue, the return value of `mystring::operator[]` must therefore be a reference. This, in fact, is why Stroustrup decided that C++ needed references as well as pointers: to permit line 11 of the above `main4.C` to use the return value of `operator[]` as an lvalue *without an asterisk.* See Stroustrup, *Design and Evolution,* pp. 85–87.

Three operators that must return a reference

People expect to be able to use the value of the following three operators as the left operand of the assignment operator `=`. Don't disappoint them. The corresponding `operator` functions must therefore return a reference.

(1) If the object contains many items of data, dress it up to look like an array. The member function that accesses the data should be named `operator[]`, and it should take one argument.

```
1     v[20] = 10;                    //v.operator[](20) = 10;
```

(2) If the object contains only one item of data, or if it makes only one item available at a time, dress it up to look like a pointer. The member function that accesses the item should be named `operator*`, and it should take no arguments.

```
2       *it = 10;                       //it.operator*() = 10;
```

(3) If the object contains only a few items of data, or if it makes only a few items available at a time, dress it up to look like a pointer to a structure. This one is harder. Create a member function named `operator->`, taking no arguments, that will load the items into a structure and return a pointer to the structure. After the arrow, write the field of the structure that we wish to access.

```
3       p->f = 10;                      //p.operator->()->f = 10;
```

For example, imagine a database whose records contain three fields, `f`, `g`, and `h`. Our object `p` contains the identification number of a record on the disk. The member function `operator->` reads the record, deposits the three fields into the fields of a structure in memory, and returns the address of the structure. We can then say `p->f` to get the value of the `f` field of the record identified by `p`.

What if we want to do more than get the value? What if we want to *change* the field, and write the new value back into the database as in the above line 3? For this we will need the elaborate machinery in pp. 967−968.

Two member functions with the same name and arguments

A non-`const mystring` object has the `operator[]` member function in line 62 of `mystring.C` on pp. 304−305, but not the one in line 73. When we apply the subscript operator to one of these objects, the computer behaves as if we had called the `operator[]` in line 62. Examples of this are in lines 9−12 of the above `main4.C`.

Conversely, a `const mystring` object has the `operator[]` member function in line 73 of `mystring.C`, but not the one in line 62. When we apply the subscript operator to one of *these* objects, the computer behaves as if we had called (or tried to call) the `operator[]` in line 73. Examples of this are in lines 15−16 of the above `main4.C`.

It looks like the two functions have the same name and arguments. But in reality they differ in the data types of their invisible arguments. The `operator[]` in line 62 receives the read/write pointer `mystring *const`, the `operator[]` in line 73 receives the read-only pointer `const mystring *const`. It is these invisible arguments which permit us to have two functions with the same name. For other examples, see pp. 641, 857, 896, and 900.

Why the operator[] in line 73 of mystring.C must construct and return a const

A `const` can never be an lvalue. The `operator[]` in line 73 of `mystring.C` returns a `const` to prevent line 16 of the above `main4.C` from compiling. Recall that the `operator+` and the postfix `operator++` for class `date` returned a `const` for the same reason; see lines 44 and 50 `date.h` on p. 274.

This is the first time we've seen a pair of member functions with the same name and the same arguments. We can do this only if one function is `const` and the other non-`const`. In other words, function name overloading takes into account the invisible argument too.

▼ Homework 3.9a: fix the bug in mystring::operator=

Now that class `mystring` has an `operator[]` that returns a reference to one of the characters in the string, there's a potential bug in the `operator=` that takes a pointer to a `char`.

Suppose someone says the following line 1. Then line 2 is useless but harmless. Line 3 should be equally useless but equally harmless.

```
1       mystring s = "hello";           //a non-const mystring
2       s = s; //s.operator=(s);
3       s = &s[0]; //s.operator=(&s.operator[](0));
```

Why will the `operator=` in the above line 3 probably destroy the contents of `s`? Have the `operator=` in line 47 of `mystring.C` on pp. 304–305 check if its pointer argument points to one of the characters in the `mystring` to which the `operator=` belongs.
▲

3.10 `operator int` Converts an Object to an Integer

Convert an object to another data type

We already know how to convert a `double` to an `int`. Now we will convert a `date` to an `int` with the same syntax. Using the same syntax for all data types, built-ins and objects, will make it easier to convert our code to "templates"; see p. 634.

One way to convert a `double` to an `int` is in line 9: we declare an `int` variable and copy the `double` into the `int`. The `double`-to-`int` compilation warning can be avoided by changing `pi` to `static_cast<int>(pi)`. We can also cast the `double` to `int` without storing the result into a named variable; see line 12.

Lines 16 and 23 show what we will do for class `date`. Let's decide that the resulting `int` should be the number of days from January 1, 0 A.D. to that `date`. One way to convert a `date` to an `int` is in line 16: we declare an `int` variable and copy the `date` into the `int`. We can also cast the `date` to `int` without storing the result into a named variable; see line 23.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     double pi = 3.14159265358979323846;
 9     int i = pi;                  //convert double to int
10
11     cout << pi << " converted to int is " << i << "\n"
12         << pi << " converted to int is " << static_cast<int>(pi)
13         << "\n\n";
14
15     date d;
16     i = d;                       //int i = d.operator int();
17
18     d.print();
19     cout << " converted to int is " << i << "\n";
20
21     d.print();
22     cout << " converted to int is "
23         << static_cast<int>(d)   //<< d.operator int()
24         << "\n";
25
26     return EXIT_SUCCESS;
27 }
```

```
3.14159 converted to int is 3
3.14159 converted to int is 3

4/8/2014 converted to int is 735208
4/8/2014 converted to int is 735208
```

To make the above conversions compile, the following line 10 defines a member function with the unusual name `operator int`. It will be called implicitly whenever we attempt to convert a `date` to an `int`. Please use this conventional name for a conversion function. Do not invent your own names—`date_to_int`, `date2int`, `elapsed_time`, etc.

A conversion function does not actually convert the object into anything. To ensure that the object remains unchanged, a conversion function should be a `const` member function. Our function merely returns an integer representing the object's value.

Give no arguments to a conversion function, and declare no data type for the return value. The data type is indicated by the name of the function itself. (In the same way, we declared no data type for the return value of a constructor.)

```
 1 //Excerpt from date.h.
 2 #ifndef DATEH
 3 #define DATEH
 4
 5 class date {
 6     static const int length[];
 7     static const int    pre[];
 8     int day;         //number of days before or after January 1, 0 A.D.
 9 public:
10     operator int() const {return day;}
11     //etc.
```

We can now use a `date` object in any context in which an `int` would be accepted.* For example, when used as the right operand of the assignment in the above line 16, the `operator int` function will be transparently called.

Ambiguous conversion

Here is a class of `date` objects that can be converted to either `int` or `long`. The comment in line 18 shows that the `static_cast<int>()` calls `operator int` to do its work.

```
 1 //Excerpt from date.h.
 2
 3 class date {
 4     static const int length[];
 5     static const int    pre[];
 6
 7     int year;
 8     int month;          //date::january to date::december inclusive
 9     int day;            //1 to date::length[month] inclusive
10
11 public:
12
13     //Return the Julian date of this date.
14     operator int() const {return pre[month] + day;}
```

* One exception: passing a `date` to a ''template'' that is unsure of whether it should accept an `int` or a `date`. See pp. 652−653.

```
15
16      //Return the number of days from January 1, 0 A.D. to this date.
17      operator long() const {
18          //return static_cast<long>(365) * year + operator int() - 1;
19          return static_cast<long>(365) * year
20              + static_cast<int>(*this) - 1;
21      }
```

There's a problem with having two or more conversion functions. Line 24 is torn between converting the `date` to an `int` or to a `long`. We have to make the decision for it, in lines 26 or 27.

```
22      date d;
23
24      if (d == 10) {                    //won't compile
25
26      if (static_cast<int>(d) == 10) {  //okay: if (d.operator int() == 10) {
27      if (static_cast<long>(d) == 10) { //okay: if (d.operator long() == 10) {
```

It is awkward for a class to have more than one conversion function to types that can be converted to each other. Most classes have only one. The following class `istream` will be an example.

Check for error with a conversion function

A conversion function gives us a convenient notation for checking the health of an object. We will demonstrate with a `date` and with the object `cin`.

We often speak of the ''logical'' expression of an `if`, `while`, `do-while`, or `for` statement:

```
1       if (a == b) {
```

But the expression is actually of type `bool` or any type that can be converted thereto. This includes `int`, `double`, enumerations, pointers, etc. And now that we have conversion functions, it also includes any object with an `operator` function that converts to `bool`, `int`, `double`, any enumeration type, or any pointer type.

In class `date`, let's replace the `operator int` and `operator long` with an `operator bool` that would be useful in an `if` statement. We also switch to the three-data-member implementation of class `date`.

```
2  //Excerpt from date.h.
3
4  class date {
5      static const int length[];
6      int year;
7      int month;          //date::january to date::december inclusive
8      int day;            //1 to date::length[month] inclusive
9
10 public:
11
12     //Return true if this object is internally consistent.
13
14     operator bool() const {
15         return january <= month && month <= december &&
16             1 <= day && day <= length[month];
17     }
```

We can now write the tests in lines 20 and 24.

```
18      date d;
19
```

```
20     if (d) {                         //if (d.operator bool()) {
21         cout << "d is healthy.\n";
22     }
23
24     if (!d) {                        //if (!d.operator bool()) {
25         cout << "d is unhealthy.\n";
26     }
```

Of course, the `if`'s in the above lines 20 and 24 can be combined into a single `if-else`.

```
27     date d;
28
29     if (d) {                         //if (d.operator bool()) {
30         cout << "d is healthy.\n";
31     } else {
32         cout << "d is unhealthy.\n";
33     }
```

The operator void * member function of cin and cout

The familiar `cin` is actually an object; its class is named `istream`. This class has a member function named `operator void *`, similar to the `operator bool` we wrote for class `date`. It returns a non-zero pointer if the `istream`'s most recent attempt at input was successful; a zero pointer if the `istream` encountered end-of-input or an i/o error.

It would have been simpler to indicate success or failure with an `operator bool`, in addition to whatever other conversion function(s) the class may have. But we have just seen why it is awkward for a class to have more than one function that converts to types that can be converted to each other. If a stream object is to have only one such function, they wanted it to be the one with the maximum bandwidth; it will be our only opportunity to get the stream's contents in the form of another data type. The non-zero pointer returned by a stream's `operator void *` is the address of the stream. With this pointer, the entire contents of the stream could be recovered. (The function actually belongs to a smaller object of class `basic_ios<char>` within the stream, and it returns the address of this object. A class whose name contains <angle brackets> is called a ''template class''; the smaller object was placed into the larger one by means of ''inheritance''. These are imposing topics; we'll do them later.)

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     cout << "Please type an integer: ";
 8     int i;
 9     cin >> i;
10
11     cout << "cin.operator void * returns " << cin.operator void *() << ".\n"
12         << "The address of cin is " << &cin << ".\n"
13         << "The address of the basic_ios<char> object within cin is "
14         << static_cast<const basic_ios<char> *>(&cin) << ".\n";
15
16     return EXIT_SUCCESS;
17 }
```

```
Please type an integer: 10
cin.operator void * returns 0x213d8.
The address of cin is 0x213d0.
The address of the basic_ios<char> object within cin is 0x213d8.
```

We can now write the test in line 21.

```
18      int i;              //uninitialized variable
19
20      cin >> i;           //cin.operator>>(i);
21      if (cin) {          //if (cin.operator void *()) {
22          cout << "The integer input succeeded.\n";
23      }
```

The expression `cin >> i` in the above line 20 has the value `cin`; we saw this on pp. 30−31. The above lines 20−21 may therefore be combined to the single line 26. It calls the `operator void *` member function of the return value of `operator>>`.

```
24      int i;              //uninitialized variable
25
26      if (cin >> i) {     //if (cin.operator>>(i).operator void *()) {
27          cout << "The integer input succeeded.\n";
28      }
```

Class `istream` has another member function `operator!`, which returns a `bool`. It is the opposite of `operator void *`, returning false if the `istream` is healthy, true if unhealthy. This lets us say line 32:

```
29      int i;
30
31      cin >> i;
32      if (!cin) {         //if (cin.operator!()) {
33          cerr << "The attempt at integer input failed.\n":
34      }
```

Incidentally, the above line 32 would have worked even if `cin` had no `operator!` function. In this case, the line would have called `operator void *` and applied the unary `!` operator to the return value.

```
35      if (!cin) {         //if (!cin.operator void *()) {
```

The above lines 31−32 can be combined to the single line 38. It calls the `operator!` member function of the return value of `operator>>`.

```
36      int i;
37
38      if (!(cin >> i)) {  //if (cin.operator>>(i).operator!()) {
39          cerr << "The attempt at integer input failed.\n":
40      }
```

Without the parentheses, the above line 38 would have begun by calling `operator!`, which returns a `bool`. The `>>` would then have right-shifted the `bool`.

```
41      if (!cin >> i) {    //won't compile: if (cin.operator!() >> i) {
```

Of course, the `if`'s in the above lines 26 and 38 can be combined into a single `if-else`.

```
42      int i;
43
44      if (cin >> i) {     //if (cin.operator>>(i).operator void *()) {
45          cout << "The integer input succeeded.\n";
46      } else {
```

```
47          cerr << "The attempt at integer input failed.\n":
48      }
```

If the follow-up `if` in the above lines 44−48 is too much trouble to write around every attempt at input, the conscientious programmer could also perform the error checking by throwing exceptions. See pp. 623−625.

▼ Homework 3.10a: which conversion function will be called?

Give class `date` the following two conversion functions. `operator void *` must be a non-`const` member function in order to return `this`. In a `const` member function, `this` would be a `const date *`, which could not be implicitly converted to a `void *`.

```
 1 class date {
 2     //etc.
 3 public:
 4     //etc.
 5     operator bool() const {   //implicit argument is read-only pointer
 6         cout << "operator bool\n";
 7         return true if the date is healthy, false otherwise;
 8     }
 9
10     operator void *() {       //implicit argument is read/write pointer
11         cout << "operator void *\n";
12         return this if the date is healthy, 0 otherwise;
13     }
```

Which function is called when you say

```
14     date d(date::january, 1, 2014);
15     if (d) {
```

Is this disconcerting?

Which function is called when you say

```
16     const date d(date::january, 1, 2014);
17     if (d) {
```

Is this more disconcerting?

The computer picks the conversion function whose (implicit) argument best matches the object `d`; it's just like function name overloading. We can level the playing field by giving `operator void *` the same implicit argument as `operator bool`. To get it to compile, we will have to convert `this` from a read-only pointer to a read/write pointer. The cast that does this conversion is `const_cast`.

```
18     operator void *() const {
19         cout << "operator void *\n";
20         return const_cast<date *>(this) if the date is healthy,
21             0 otherwise;
22     }
```

Which function is now called in the above lines 15 and 17? The moral is that we shouldn't have an `operator bool` and an `operator void *` in the same class.

▲

The computer will not apply more than one implicit conversion

> HENRY V [to PRINCESS KATHERINE].
> If thou would have such a one, take me;
> and take me, take a soldier;
> take a soldier, take a king.
>
> —*Henry V*, V i 163−164

> When you're a Jet, you're the swingin'est thing—
> Little boy, you're a man,
> Little man, you're a king!
>
> —*West Side Story*

The `operator gas` member function of class `liquid` will be our first example of a function that converts one type of object to another: a `liquid` to a `gas`. Its return value is the anonymous temporary `gas` object constructed in line 11 of `liquid.h`, when the return statement passes `liquid::n` to the constructor for class `gas`. (See p. 138, ¶ (4), for a call to a constructor in a return statement.) Since this return value is an automatic variable, our `operator gas` must return via pass-by-value, not pass-by-reference.

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

```
 1 #ifndef GASH
 2 #define GASH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class gas {      //Mason Williams
 7     int n;
 8 public:
 9     gas(int initial_n): n(initial_n) {}
10 };
11 #endif
```

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

```
 1 #ifndef LIQUIDH
 2 #define LIQUIDH
 3 #include <iostream>
 4 #include "gas.h"
 5 using namespace std;
 6
 7 class liquid {
 8     int n;
 9 public:
10     liquid(int initial_n): n(initial_n) {}
11     operator gas() const {cout << "liquid to gas\n"; return n;} //return gas(n);
12 };
13 #endif
```

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

```
 1 #ifndef SOLIDH
 2 #define SOLIDH
 3 #include <iostream>
```

```
 4 #include "liquid.h"
 5 using namespace std;
 6
 7 class solid {
 8     int n;
 9 public:
10     solid(int initial_n): n(initial_n) {}
11     operator liquid() const {cout << "solid to liquid\n"; return n;}
12 };
13 #endif
```

Lines 15−16 need no casts. Their comments show what's going on: the compiler is willing to apply an implicit conversion.

I wanted line 18 to convert the `solid` to a `liquid`, and then the `liquid` to a `gas`. But the computer will not apply more than one implicit user-defined conversion, so line 18 did not compile. I tried to help it along with the cast in line 19, but that wouldn't compile either.

The path from `solid` to `gas` is more than one step. We must therefore spell out the intermediate step `liquid` in line 20. The comment shows what's going on: the `operator liquid` member function of the `solid` returns an anonymous `liquid`, and then the `operator gas` member function of the anonymous `liquid` returns an anonymous `gas`. Our first example of calling a member function of an anonymous temporary object returned by a function was in line 2 on pp. 137−138.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3
 4 #include "gas.h"
 5 #include "liquid.h"
 6 #include "solid.h"
 7 using namespace std;
 8
 9 int main()
10 {
11     solid ice = 10;
12     liquid water = 20;
13     gas steam = 30;
14
15     water = ice;   //convert solid to liquid: water = ice.operator liquid();
16     steam = water; //convert liquid to gas:   steam = water.operator gas();
17
18     //steam = ice;                      //won't compile
19     //steam = static_cast<gas>(ice);    //won't compile
20     steam = static_cast<liquid>(ice);   //ice.operator liquid().operator gas();
21
22     return EXIT_SUCCESS;
23 }
```

```
solid to liquid         line 15
liquid to gas           line 16
solid to liquid         line 20
liquid to gas           line 20
```

Convert a built-in to an object

Now that we have converted `date` to `int`, let's convert `int` to `date`. We can't do this with an `operator date` member function of class `int`. The reason is simple: there is no class `int`.

But we already know how to perform this conversion. Simply define a constructor for class `date` taking one `int` in line 36. The resulting `date` will be the specified number of days before or after January 1, 0 A.D. The cast in line 21 calls this constructor, as shown in the comment. Of course, the constructor can also be called explicitly in line 25; see pp. 137−138, ¶ (1).

A constructor that can be called with one argument is called a *converting constructor*. It might have only one argument, or it may have a default value for every additional argument. One example of a converting constructor is the copy constructor, although it does not perform any conversion. To permit line 14 to compile, the converting constructor must not be `explicit` (p. 137). If it was `explicit`, line 14 would have to be changed to

```
1     date d(i);
```

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date today;
 9     int i = today;   //int i = today.operator int();
10     cout << "The original date ";
11     today.print();
12     cout << " converted to int is " << i << ".\n\n";
13
14     date d = i;
15
16     cout << i << " converted back to date is ";
17     d.print();
18     cout << "\n";
19
20     cout << i << " converted back to date is ";
21     static_cast<date>(i).print();   //date(i).print();
22     cout << "\n";
23
24     cout << i << " converted back to date is ";
25     date(i).print();
26     cout << "\n";
27
28     return EXIT_SUCCESS;
29 }
```

We'll switch back to the class `date` with one data member:

```
30 //Excerpt from date.h.
31
32 class date {
33     static const int length[];
34     int day;         //number of days before or after January 1, 0 A.D.
35 public:
```

```
36      date(int initial_day): day(initial_day) {}
37      operator int() const {return day;}
```

It is pleasant that the `int` in the above line 12 contains enough information for lines 14, 21, and 25 to reconstruct the value of the original `date` object. This is another reason why the conversion function for class `istream` returns a `void *`. The return value could be the address of the `istream`, from which the entire value of the `istream` could be recovered.

```
The original date 4/8/2014 converted to int is 735208.

735208 converted back to date is 4/8/2014
735208 converted back to date is 4/8/2014
735208 converted back to date is 4/8/2014
```

3.11 `operator<<` and `operator>>` Perform I/O

Stream objects are impossible to copy

The conventional way to input and output a C++ object is by overloading the operators >> and <<. Doing it this way will let us use the same syntax for i/o with all data types, built-ins and objects. The payoff will come when we do templates; see p. 634.

```
1       date d;
2
3       cin >> d;           //operator>>(cin, d);
4
5       cout << d;          //operator<<(cout, d);   No more d.print();
```

Before we define an `operator>>` and `operator<<` for class `date`, we will examine `cin` and `cout` more closely by doing i/o with integers.

`cin`, `cout`, `cerr`, and `clog` are actually objects. `cin` is of class `istream`, and `cout`, `cerr`, and `clog` are of class `ostream`. They are called *stream objects* because an i/o channel carries a stream of characters.

The `c` stands for ‘‘character’’, since they perform i/o one character at a time. There are also `wcin`, `wcout`, etc., which perform i/o one wide character at a time.

`clog` is just like `cerr`, except that `clog` is buffered and `cerr` is not. `clog` is intended for large-volume logging and tracing output; `cerr` for shorter error messages.

Lines 10 and 11 show two ways of trying to call the copy constructor for `cout`. But the copy constructors for classes `istream` and `ostream` are private, like those for classes `wolf` and `node`, so they cannot be called from outside the bodies of the member functions or friends of their classes. We therefore have no way to construct a copy of an `istream` or `ostream`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f(ostream ost);
 6
 7 int main()
 8 {
 9     cout << "hello";
10     ostream another = cout;   //won't compile
```

```
11     f(cout);                  //won't compile: attempt to pass cout by value
12     return EXIT_SUCCESS;
13 }
14
15 void f(ostream ost)
16 {
17     ost << "goodbye";
18 }
```

My compiler complained only about line 10, but when I comment out that line it complains about 11. The cryptic error message says that the copy constructor for class `ostream` calls the one for class `basic_ios<char>`, which in turn calls the one for class `ios_base`, which is the copy constructor that is private. We will see why one copy constructor calls another when we do inheritance on p. 476. For now, take a peek at the relationships between the classes in the diagrams that accompany pp. 383−385.

```
In file included from
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/i
os:39:0,
                 from
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/o
stream:40,
                 from
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/i
ostream:40,
                 from copy.C:1:
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/b
its/ios_base.h: In copy constructor 'std::basic_ios<char>::basic_ios(const
std::basic_ios<char>&)':
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/b
its/ios_base.h:785:5: error: 'std::ios_base::ios_base(const std::ios_base&)' is
private
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/i
osfwd:77:11: error: within this context
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/i
osfwd: In copy constructor 'std::basic_ostream<char>::basic_ostream(const
std::basic_ostream<char>&)':
/usr/gcc/4.5/lib/gcc/sparc-sun-solaris2.11/4.5.2/../../../../include/c++/4.5.2/i
osfwd:86:11: note: synthesized method 'std::basic_ios<char>::basic_ios(const
std::basic_ios<char>&)' first required here
copy.C: In function 'int main()':
copy.C:10:20: note: synthesized method
'std::basic_ostream<char>::basic_ostream(const std::basic_ostream<char>&)'
first required here
```

Why are we forbidden to construct a copy of `cout`? Well, if `cout` is copied in lines 10 or 11, it might still contain buffered data that has not yet been output to the outside world. The new objects `another` in line 10 and `ost` in line 15 would then be constructed pregnant with their own copies of the string `"hello"`. All three objects, `cout`, `another`, and `ost`, would eventually output `hello`, and the user would see the word three times.

To avoid line 11's attempt to copy `cout`, we must pass the argument of `f` by reference. See pp. 185−189. The reference must be read/write because the i/o operation in line 21 changes the stream object. Any `istream` or `ostream` passed to or from a function must be passed as a read/write reference. For the rarity and danger of a read/write reference argument, see pp. 73−74, 158.

```
19 void f(ostream& ost)
```

```
20 {
21     ost << "goodbye";
22 }
```

Perform input with operator>>

When we write the expression `cin >> i` in line 2, the computer behaves as if we had written the call to the member function `operator>>` in the comment beside it. This `operator>>` performs integer input.

```
1     int i;              //uninitialized variable
2     cin >> i;           //cin.operator>>(i);
```

The argument `i` is passed as a read/write reference, allowing `operator>>` to install a new value into it. For the same reason, the variables passed to the C function `scanf` are passed by reference.

There is a similar member function for every built-in data type except `char`. For example, this `operator>>` performs `double` input.

```
3     double d;           //uninitialized variable
4     cin >> d;           //cin.operator>>(d);
```

For reasons too trivial to go into now, the `operator>>`'s that input a `char` and an array of `char`'s happen not to be member functions. We'll see why on p. 330. This has no effect on the code you write; only the expansion in the comments is slightly different.

```
5     char c;             //uninitialized variable
6     cin >> c;           //operator>>(cin, c), not cin.operator>>(c)
7
8     char s[10];         //uninitialized variable
9     cin >> s;           //operator>>(cin, s), not cin.operator>>(s)
```

Call operator>> and operator void * in an if statement

The expression `cin >> i` in line 3 gives a new value to `i`. But we also know that the expression has a value of its own. We even know what this value is: `cin`, the left operand of the `>>`. This value is used whenever we input two or more values in the same expression (`cin >> i >> j`; see p. 31). And now that we've done operator overloading, we know where the value came from: it is the return value of the `operator>>` function.

The `operator>>`'s that are member functions (for every built-in type except `char`) return the `istream` to which they belong. For example, the call to `cin.operator>>(i)` in the above line 2 returns `cin`. Similarly, the `operator>>`'s that are not member functions (for `char` and `char *`) return the `istream` that was passed to them. For example, the call to `operator>>(cin, c)` in the above line 6 also returns `cin`.

Since the value of `cin >> i` is `cin`, we can combine lines 3–4 to the single line 8. The comment in that line shows what's really going on: we're calling a member function (`operator void *`) of an anonymous temporary object returned by another function (`operator>>`). Our first example of this was in line 2 on pp. 137–138.

```
1     int i;
2
3     cin >> i;           //cin.operator>>(i);
4     if (cin) {          //if (cin.operator void *()) {
5         cout << "The integer input succeeded.\n";
6     }
7
8     if (cin >> i) {     //if (cin.operator>>(i).operator void *()) {
9         cout << "The integer input succeeded.\n";
```

```
10      }
```

We can also combine lines 13–14 to line 18. The inner parentheses in line 18 are necessary to make the >> execute before the !.

```
11      int i;
12
13      cin >> i;           //cin.operator>>(i);
14      if (!cin) {         //if (cin.operator!()) {
15          cerr << "The attempt at integer input failed.\n";
16      }
17
18      if (!(cin >> i)) {   //if (cin.operator>>(i).operator!()) {
19          cerr << "The attempt at integer input failed.\n";
20      }
```

Of course, we can write a single if-else:

```
21      int i;
22
23      if (cin >> i) {     //if (cin.operator>>(i).operator void *()) {
24          cout << "The integer input succeeded.\n";
25      } else {
26          cerr << "The attempt at integer input failed.\n";
27      }
```

With data type char, the corresponding expressions have slightly different expansions in the comments in lines 30–32, since the operator>> for char happens not to be a member function.

```
28      char c;
29
30      cin >> c;           //operator>>(cin, c);
31      if (cin >> c) {     //if (operator>>(cin, c).operator void *()) {
32      if (!(cin >> c)) {  //if (operator>>(cin, c).operator!()) {
```

Call operator>> in a while loop

Here's a while loop that inputs integers until the input is exhausted or an error is encountered. (On my platform, the end-of-input keystroke is control-d.) Line 9 calls the operator void * member function of the return value of the operator>> member function of cin, which keeps returning non-zero as long as healthy integers are still coming in.

We break out of the loop when operator void * returns zero. This tells us that an attempt at input has failed, but it doesn't tell us *why* the attempt failed. We can get more detailed information from the member functions in lines 13–15. eof returns true if the most recent attempt at input encounted end-of-file. bad returns true if the most recent attempt was unable to read characters (or the end-of-file indication) from the outside world. fail returns true if the most recent attempt failed for any reason, including the two above. Another possible cause of fail is if the characters that were read do not spell out a legal value for the receiving variable, in this case the integer in line 9.

It is to be hoped that we broke out of the while loop because of end-of-input. In this case, the eof and fail bits will be on, and the bad bit will be off. All three bits are available in the bit pattern returned by the rdstate member function in line 17. For convenience, there is an enumeration corresponding to each bit. The enumerations can be ''bitwise or'ed'' together to form a desired bit pattern. Just remember to do the ''or'' within parentheses, since it has lower precedence than ==. See p. 9.

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

```
1 #include <iostream>
```

```
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int i;
 8
 9     while (cin >> i) {       //while (cin.operator>>(i).operator void *()) {
10         cout << i << "\n";   //operator<<(cout.operator<<(i), "\n");
11     }
12
13     cout << "cin.eof() == " << cin.eof() << "\n"
14         << "cin.bad() == " << cin.bad() << "\n"
15         << "cin.fail() == " << cin.fail() << "\n";
16
17     return cin.rdstate() == (ios_base::eofbit | ios_base::failbit)
18         ? EXIT_SUCCESS : EXIT_FAILURE;
19 }
```

The exit status is `EXIT_SUCCESS` if we broke out of the loop because of end-of-input.

```
10 20 30        You type this input and press RETURN.
10
20
30              After seeing these three lines of output, you press the end-of-input keystroke.
cin.eof() == 1
cin.bad() == 0
cin.fail() == 1
control-d       You type the end-of-file keystroke, and the exit status is EXIT_SUCCESS.
```

The exit status is `EXIT_FAILURE` if we broke out of the loop for any other reason, e.g., invalid input.

```
10 20 abc       You type this input and press RETURN,
10
20              and before you even type the end-of-file keystroke, the program terminates.
cin.eof() == 0
cin.bad() == 0
cin.fail() == 1
```

The loop in the above line 9 can read a series of values of any data type, or at least any data type that has an `operator>>`. The corresponding expression with a `char` has a slightly different expansion in the comment in line 9, since the `char operator>>` is not a member function.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     char c;
 8
 9     while (cin >> c) {      //while (operator>>(cin, c).operator void *()) {
```

```
10          cout << c;          //operator<<(cout, c);
11      }
12
13      return cin.rdstate() == (ios_base::eofbit | ios_base::failbit)
14          ? EXIT_SUCCESS : EXIT_FAILURE;
15  }
```

a b c	*You type this input and press* RETURN.
abc	*It echoes only the non-whitespace characters.*
control-d	*You type the end-of-file keystroke, and the exit status is* EXIT_SUCCESS.

The C++ equivalent of the while-getchar loop

As the above output shows, the `operator>>` for `char` discards the whitespace characters that it inputs. Sometimes, this is what we want. One way to avoid this would be to use the `noskipws` i/o manipulator on p. 359.

Another way would be to call the member function `get` instead of `operator>>`. `get` reads one character from its `istream` without skipping whitespace. Like the `char operator>>`, `get` accepts its `char` argument as a read/write reference and returns the `istream`.

The `put` member function in line 10 was invented only for symmetry with `get`; saying `cout << c` would have worked just as well. See p. 854 for another way to copy every character.

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

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      char c;
 8
 9      while (cin.get(c)) {   //while (cin.get(c).operator void *()) {
10          cout.put(c);
11      }
12
13      return cin.rdstate() == (ios_base::eofbit | ios_base::failbit)
14          ? EXIT_SUCCESS : EXIT_FAILURE;
15  }
```

a b c	*You type this input and press* RETURN.
a b c	*It now echoes every character of input, including the spaces.*
control-d	*You type the end-of-file keystroke, and the exit status is* EXIT_SUCCESS.

The above loop is the C++ equivalent of the classic `while-getchar` loop in C. The return value of `getchar` must not be stored in a `char`. Let's assume that a `char` is eight bits. Then there are 256 possible `char` values. But `getchar` will return one of 257 possible values, a range that will not fit in a `char`.

If `getchar` and `putchar` are macros, we cannot take their addresses. `putchar` may be a macro that evaluates its argument more than once.

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

```
 1 #include "stdio.h"
 2 #include "stdlib.h"
 3
 4 int main()
 5 {
 6     int c;   /* must be int, not char, for getchar */
 7
 8     while ((c = getchar()) != EOF) {
 9         putchar(c);
10     }
11
12     return ferror(stdin) ? EXIT_FAILURE : EXIT_SUCCESS;
13 }
```

a b c	*You type this input and press* RETURN.
a b c	*It echoes every character of input, including the spaces.*
control-d	*You type the end-of-file keystroke, and the exit status is* EXIT_SUCCESS.

Incidentally, we can now explain why the `char operator>>`, unlike the other ones, is not a member function of class `istream`. The `char operator>>` calls the member function `get` to do most of its work, so it needs no access to the private members of class `istream`. (Similarly, the `char operator<<` is not a member function of class `ostream`. It calls the member function `put` to do its work.)

Discover why an attempt at input failed

In addition to `eof`, class `istream` has two other member functions that return true or false to indicate why the most recent attempt at input failed. Lines 11−32 show the complete incantation that the conscientious programmer will write after an `if`.

We must test `eof` before `fail` (lines 12 and 21), because `fail` would be true if we encountered end-of-input *or* if the first non-whitespace character was wrong. If we tested `fail` first, we would be unable to distinguish between these two causes of failure.

We must test `bad` before `fail` (lines 16 and 21), because `fail` would be true if we could not input characters *or* if the first non-whitespace character was wrong. If we tested `fail` first, we would be unable to distinguish between these two causes of failure.

We test `eof` before `bad` because `eof` is the more common occurrence.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main(int argc, char **argv)
 6 {
 7     cout << "Please type an integer: ";
 8     int i;        //uninitialized variable
 9     cin >> i;     //cin.operator>>(i);
10
11     if (!cin) {   //if (cin.operator!()) {
12         if (cin.eof()) {
13             cerr << argv[0] << ": end of input\n";
14         }
15
```

```
16          else if (cin.bad()) {
17              cerr << argv[0] << ": can't input characters "
18                  "from the outside world\n";
19          }
20
21          else if (cin.fail()) {
22              cerr << argv[0] << ": first non-whitepace character "
23                  << "encountered was neither a digit,\n"
24                  << "nor a minus sign followed by a digit.\n";
25          }
26
27          else {
28              cerr << argv[0] << ": unknown input error\n";
29          }
30
31          return EXIT_FAILURE;
32      }
33
34      cout << "The integer was " << i << ".\n";
35      return EXIT_SUCCESS;
36  }
```

We can combine the above lines 9−11 to

```
37      if (!(cin >> i)) {              //if (cin.operator>>(i).operator!()) {
```

```
Please type an integer: 10
The integer was 10.
```

```
Please type an integer: control-d    (the end-of-file keystroke)
why: end of input
```

```
Please type an integer: abc
why: first non-whitepace character encountered was neither a digit,
nor a minus sign followed by a digit.
```

If we sabotage `cin` by inserting `cin.rdbuf(0);` at the above line 8½ (equivalent in its destructive effect to `stdin->_base = stdin->_ptr = garbage;` in C), `bad` would return true in line 17.

```
Please type an integer:
why: can't input characters from the outside worldbefore I have time to type anything
```

Taint cin by hand

The following program makes `istream::fail` return `true` even though nothing has failed. We'll need to do this when we write our own `operator>>` functions.

The functions in the previous section returned values that are determined by the settings of three bits inside the `istream`. These bits are turned on automatically when anything goes wrong. When we encounter end-of-file, the ''eof'' and ''fail'' bits are turned on. When some other reason prevents us from reading characters, the ''bad'' and ''fail'' bits are turned on. When we have read characters that do not spell out a legal value, the ''fail'' bit is turned on.

The bits can also be turned on manually by calling the `setstate` member function of the `istream` (line 10). It turns on the specified bit(s), leaving the others unchanged. The argument is any

combination of three enumeration values that are members of class `ios_base`, bitwise-or'ed together. These values correspond to the three bits in the `istream`: `eofbit`, `badbit`, or `failbit`,

Now that we know about the bits, let's see all six of the `const` member functions that return them. The `rdstate` in line 17 returns a bit pattern of type `iostate`; we examine the individual; bits in lines 21, 24, and 27. The next four functions return `bool`. The function `good` returns `true` if all three bits are off. The functions `eof` and `bad` return `true` if the corresponding bit is on. The function `fail` returns `true` if either one of the `bad` or `fail` bits is on. The function `operator!` returns the same value as `fail`. Finally, the `operator void *` function returns exactly the opposite: zero if `operator!` returns `true`, non-zero otherwise.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void health(const istream& is);
 6
 7 int main()
 8 {
 9     health(cin);
10     cin.setstate(ios_base::failbit);
11     health(cin);
12     return EXIT_SUCCESS;
13 }
14
15 void health(const istream& is)
16 {
17     const ios_base::iostate state = is.rdstate();
18
19     cout
20         << "eof returns " << is.eof() << ", eofbit is "
21         << static_cast<bool>(state & ios_base::eofbit) << "\n"
22
23         << "bad returns " << is.bad() << ", badbit is "
24         << static_cast<bool>(state & ios_base::badbit) << "\n"
25
26         << "fail returns " << is.fail() << ", failbit is "
27         << static_cast<bool>(state & ios_base::failbit) << "\n"
28
29         << "good returns " << is.good() << "\n"
30         << "operator void * returns " << is.operator void *() << "\n"
31         << "operator! returns " << is.operator!() << "\n\n";
32 }
```

```
eof returns 0, eofbit is 0
bad returns 0, badbit is 0
fail returns 0, failbit is 0
good returns 1
operator void * returns 0x21a38
operator! returns 0

eof returns 0, eofbit is 0
bad returns 0, badbit is 0
fail returns 1, failbit is 1
good returns 0
operator void * returns 0
operator! returns 1
```

When will an operator>> stop reading characters?

Let's examine the fine points of the `operator>>` functions that input the built-in data type `int` and the standard library data type `complex<double>` (a complex number whose two data members are `double`'s). We will then make our `operator>>` for class `date` follow the same conventions.

First, let's see when the `int operator>>` will stop inputting characters, especially when unsuccessful. We will feed the following program one line of input, ending with a newline. Most of these characters will be input by the `operator>>` in line 11. The remaining characters will be input and output by the loop in lines 34−36. (Line 34 is expanded in the comment in 32.) We have to use `get` to input these characters, since `operator>>` would discard whitespace.

Before line 34 calls `get`, however, wene 29 must call `clear`. It does the opposite of the `setstate` in line 10 of the previous program, restoring `cin` to health by turning off `eofbit`, `badbit`, and `failbit`. This permits `cin` to attempt further input after a failure. The C Standard Library has a similar function named `clearerr`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main(int argc, char **argv)
 6 {
 7     int status = EXIT_FAILURE;   //guilty until proven innocent
 8     cout << "Please input an integer: ";
 9
10     int i;               //uninitialized variable
11     if (cin >> i) {      //if (cin.operator>>(i).operator void *()) {
12         cout << "The integer is " << i << ".\n";
13         status = EXIT_SUCCESS;
14     } else {
15         cout << argv[0] << ": integer input failed, ";
16         if (cin.eof()) {
17             cout << "eof\n";
18         } else if (cin.bad()) {
19             cout << "bad\n";
20         } else if (cin.fail()) {
21             cout << "fail\n";
22         } else {
```

```
23              cout << "unknown\n";
24          }
25      }
26
27      cout << "operator>> did not eat the following characters: \"";
28
29      cin.clear();
30      char c;
31
32      //while (cin.get(c).operator void *() && c != '\n') {
33
34      while (cin.get(c) && c != '\n') {
35          cout.put(c);
36      }
37
38      cout << "\".\n";
39      return status;
40  }
```

(1) The following line of input has three blanks before the number. The `operator>>` for `int` inputs and ignores this whitespace. Our `operator>>` for `date` will do the same thing.

```
Please input an integer:    10
The integer is 10.
operator>> did not eat the following characters: "".
```

(2) This line of input has three blanks after the number. `operator>>` for `int` stops inputting characters as soon as it encounters one that could not legally be part of the `int`. Our `operator>>` for `date` will do the same thing.

```
Please input an integer: 10
The integer is 10.
operator>> did not eat the following characters: "   ".
```

(3) This line of input has `abc` after the number. Once again, the `operator>>` for `int` stops inputting characters as soon as it encounters one that could not legally be part of the `int`. Our `operator>>` for `date` will do the same thing.

```
Please input an integer: 10abc
The integer is 10.
operator>> did not eat the following characters: "abc".
```

(4) On my platform, an `int` is 32 bits. The largest number that will fit in it is 2,147,483,647; the smallest is −2,147,483,648. (Look up these numbers in the header file `<climits>` in the macros `INT_MIN` and `INT_MAX`; they will be used on p. 539). The `operator>>` for `int` keeps inputting characters as long as they are *syntactically* legal; it then rejects a value that is out of range. Our `operator>>` for `date` will do the same thing.

```
Please input an integer: 2147483647000abc
eat: integer input failed, fail
operator>> did not eat the following characters: "abc".
```

(5) Here is a case where the `operator>>` for `int` inputs at least one character that could be part of an integer, an then fails when it does not find the rest of the integer. It inputs the negative sign but no additional characters, and makes no attempt to regurgitate (`ungetc`, as we would say in C) the negative sign.

Our operator>> for date will do the same thing.

```
Please input an integer: –abc
eat: integer input failed, fail
operator>> did not eat the following characters: "abc".
```

(6) Finally, the operator>> for class complex<double> permits whitespace between the tokens of a value read from input. Our operator>> for date will do the same thing.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <complex>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     complex<double> c;
 9     cout << "Please input a complex number: ";
10     cin >> c;
11     cout << "The number was " << c << ".\n";
12     return EXIT_SUCCESS;
13 }
```

```
Please input a complex number: ( 10 , 20 )
The number was (10,20).
```

To be consistent with int and complex<double>, our operator>> for class date will do the following.

(1) Input and discard leading whitespace.

(2) Not input trailing whitespace or any other character after the date that is not part of the date.

(3) Keep inputting characters as long as they could be part of a syntactically legal date (12/31/2014), even if the numbers are out of range (12/310/2014).

(4) Set the istream's failbit if the date is out of range.

(5) Make no attempt to regurgitate the leading part of a date when it discovers that the rest of the date is not there (12/31/abc). It will set the istream's failbit.

(6) Permit whitespace before each slash: 12 / 31 / 2014.

Input and output a date

Until now we have output our objects by calling ad hoc member function such as .print(). We will now input and output an object with the conventional C++ operators << and >>.

```
1     date d;
2
3     cout << "Today is ";
4     d.print();                    //The old way is bad.
5     cout << ".\n";
6
7     cout << "Today is ";
8     cout << d;                    //The new way is good.
9     cout << ".\n";
```

One advantage of the above lines 7−9 is that they may be combined to one statement.

```
10      cout << "Today is " << d << ".\n";
```

Another is that we can specify any destination for the output:

```
11      cout << "Today is " << d << ".\n";
12      cerr << "Today is " << d << ".\n";
13      clog << "Today is " << d << ".\n";
```

The same advantages will accrue to our our operator `>>`. Using the same syntax to perform i/o with all data types, built-ins and objects, will make it easy to rewrite our code in the form of ‘‘templates’’. See p. 634.

There is no way we could have invented a new `printf` format for outputting a `date`. `printf` is not extensible. See pp. 29−30, ¶ (3).

```
14      date d;
15      scanf("%D", &d);                        //Can't invent a %D for scanf.
16      printf("Today's date is %D.\n", d);     //Can't invent a %D for printf.
```

As in every language, much more can go wrong during input (lines 10−23) than output (lines 25−26).

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     date d;
 9
10     cout << "Please input a date in the format m/d/y and press RETURN: ";
11     cin >> d;     //operator>>(cin, d);
12
13     if (!cin) {   //if (cin.operator!()) {
14         if (cin.eof()) {
15             cerr << argv[0] << ": end of file\n";
16         } else if (cin.bad()) {
17             cerr << argv[0] << ": can't hear from outside world\n";
18         } else if (cin.fail()) {
19             cerr << argv[0] << ": input not in the format m/d/y\n";
20         } else {
21             cerr << argv[0] << ": don't know why input failed\n";
22         }
23     }
24
25     //operator<<(operator<<(operator<<(cout, "The date was "), d), ".\n");
26     cout << "The date was " << d << ".\n";
27
28     return cin ? EXIT_SUCCESS : EXIT_FAILURE;
29 }
```

The above lines 11−13 may be combined to

```
30      if (!(cin >> d)) {          //if (operator>>(cin, d).operator!()) {
```

Member functions or friends?

The operator<< and operator>> functions for most of the built-in types needed to use the private members of classes ostream and istream. That's why they were member functions of these classes. But our operator<< and operator>> for class date can be written without any mention of the private members of the streams. They will not be member functions or friends of those classes.

On the other hand, our operator<< and operator>> will need to use the private members of class date. They must therefore be either members or friends of that class. But by p. 287, ¶ (4), they can't be member functions. Were they member functions, they would be member functions of their left operand, and the left operand of << and >> is always a stream (as in the above lines 25−26 and 11). They must therefore be friends of class date.

Pass by value or pass by reference?

The date argument of operator<< in line 14 does not have to be a reference. We made it one only to avoid constructing and destructing an unnecessary copy of the date. It is a read-only reference to ensure that the operator<< cannot damage the date. On the other hand, the date argument of operator>> in line 15 must be a reference, and a read/write one to boot, so that the operator>> can install a new value into it. For the same reason, the arguments of the C function scanf had to be pointers.

The stream argument and the return value of operator<< and operator>> must be passed by reference, since we're not allowed to copy an ostream or istream. The argument and return value are therefore the same object. The references must be read/write, because output and input change the ostream and istream objects. For functions that return a reference, see line 19 of return_int.C on p. 75.

```
 1 #ifndef DATEH           //Excerpt from date.h.
 2 #define DATEH
 3 #include <iostream>     //for ostream
 4 using namespace std;
 5
 6 class date {
 7     static const int length[];
 8     static const int    pre[];
 9     int year;
10     int month;               //1 to 12 inclusive
11     int day;                 //1 to length[month] inclusive
12     //etc.
13
14     friend ostream& operator<<(ostream& ost, const date& d);
15     friend istream& operator>>(istream& ist,       date& d);
16     //etc.
```

Error detection

To read a date in the format 12/31/2014, operator>> must perform five separate input operations:

(1) line 10, int

(2) line 16, char

(3) line 26, int

(4) line 31, char

(5) line 41, int

If any operation fails, the operator>> returns the failed stream (lines 12, 18, 28, 33, 43), and line 13 of the above main.C will detect the failure.

Even if one of the five input operations is successful, the value that was input may be invalid. Line 16 may have read a character successfully, leaving us with a healthy input stream. But if that character is anything other than a slash, line 21 taints the stream and once again line 13 of `main.C` detects the failure.

The `main` function is interested in knowing about these failures because it may want to make a second attempt at inputting the `date`. To do this, it will have to read and discard the remaining characters of the invalid `date` from the input stream. There is no reliable way to recognize the last of these characters; usually the best we can do is read and discard up to the next blank or newline.

`operator>>` must put no values into the `date` unless all three integers are valid. Once again, an invalid value makes us return prematurely (lines 51 and 57).

Note that our `operator>>` and the `operator>>`'s for the built-in types respond to the different kinds of invalid input in the same way. The ''integer'' `abc` or the ''date'' `12-31-2014` (without the requisite slashes) will cause the member functions `operator!` and `fail` of the input stream to return true. The syntactically legal but out-of-range integer `2147483648` or date `12/32/2014` will also cause these member functions to return true. Later, our `operator>>` will also be able to ''throw an exception''; see pp. 624−625.

Our `operator>>` also agrees with the others in consuming only as much input as is syntactically legal. For example, if we try to feed the characters `-abc` into the integer `operator>>`, it will input the minus sign but not the letters. The letters can be read by a subsequent input operation after the input stream has been `clear`'ed. Similarly, if we try to feed the characters `12/31-2014` into our `operator>>` for class `date`, it will input the `12/31` but not the `-2014`.

An object should be constructed only once, so `operator>>` can not install a new value into a `date` object by calling its constructor. For example, the `date` object in line 11 of the above `main.C` was already constructed in line 8. To avoid writing the same code in a constructor and in the `operator>>`, both functions can call a common subroutine, which should be a private member function.

The `operator<<` and `operator>>` functions are sometimes called *inserters* and *extractors*. We will improve the `operator<<` on pp. 460−461.

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

```
 1 ostream& operator<<(ostream& ost, const date &d)
 2 {
 3     ost << d.month << "/" << d.day << "/" << d.year;
 4     return ost;
 5 }
 6
 7 istream& operator>>(istream& ist, date& d)
 8 {
 9     int month;      //uninitialized variable
10     ist >> month;
11     if (!ist) {     //if (ist.operator!()) {
12         return ist;
13     }
14
15     char c;         //uninitialized variable
16     ist >> c;
17     if (!ist) {
18         return ist;
19     }
20     if (c != '/') {
21         ist.setstate(ios_base::failbit);
22         return ist;
23     }
24
```

```
25      int day;        //uninitialized variable
26      ist >> day;
27      if (!ist) {
28          return ist;
29      }
30
31      ist >> c;
32      if (!ist) {
33          return ist;
34      }
35      if (c != '/') {
36          ist.setstate(ios_base::failbit);
37          return ist;
38      }
39
40      int year;       //uninitialized variable
41      ist >> year;
42      if (!ist) {
43          return ist;
44      }
45
46      //Put no values into d until we've verified that all three are valid.
47
48      if (month < date::january || month > date::december) {
49          cerr << "bad month " << month << "\n";
50          ist.setstate(ios_base::failbit);
51          return ist;
52      }
53
54      if (day < 1 || day > date::length[month]) {
55          cerr << "bad day " << day << " of month " << month << "\n";
56          ist.setstate(ios_base::failbit);
57          return ist;
58      }
59
60      d.year = year;
61      d.month = month;
62      d.day = day;
63
64      return ist;     //as in the above line 4
65  }
```

The above lines 3−4 should be combined to to

```
66      return ost << d.month << "/" << d.day << "/" << d.year;
```

since the value of the whole expression is `ost`.

The above lines 10−11 may be combined to

```
67      if (!(ist >> month)) {       //if (ist.operator>>(month).operator!()) {
```

Ditto for lines 16−17, 26−27, 31−32, and 41−42.

```
Please input a date in the format m/d/y and press RETURN: 4/8/2014
The date was 4/8/2014.
```

```
Please input a date in the format m/d/y and press RETURN: abc/8/2014
progname: input not in the format m/d/y         line 19 of main.C
The date was 4/8/2014.
```

```
Please input a date in the format m/d/y and press RETURN: 4/8-2014
progname: input not in the format m/d/y         line 19 of main.C
The date was 4/8/2014.
```

```
Please input a date in the format m/d/y and press RETURN: 12/32/2014
bad day 32 of month 12                          line 55 of date.C
progname: input not in the format m/d/y         line 19 of main.C
The date was 4/8/2014.
```

The hidden nesting

Operator overloading gives us a nice, linear notation to hide a series of nested function calls. When we write lines 1−3, the computer behaves as if we had written lines 5−7; when we write lines 9−11, the computer behaves as if we had written 13−15. The `<<` and `>>` operators have left-to-right associativity, so the first function called is the one that corresponds to the leftmost operator.

Compare the hidden nesting of the `operator=`'s on pp. 300−301.

```
 1     cout << "Today is ";
 2     cout << "Today is " << d;
 3     cout << "Today is " << d << "\n";
 4
 5                          operator<<(cout, "Today is ");
 6                operator<<(operator<<(cout, "Today is "), d);
 7     operator<<(operator<<(operator<<(cout, "Today is "), d), "\n");
 8
 9     cin >> d1;
10     cin >> d1 >> d2;
11     cin >> d1 >> d2 >> d3;
12
13                          operator>>(cin, d1);
14                operator>>(operator>>(cin, d1), d2);
15     operator>>(operator>>(operator>>(cin, d1), d2), d3);
```

▼ Homework 3.11a: call the operator<< we just wrote

In the constructors and `print` member function of class `employee`, take advantage of the `operator<<` we just wrote for class `date`. Write the `employee` error messages to `cerr`, not to `cout`. Consolidate consecutive output statements into a single statement in the `employee` constructors.

```
1     cout << "birth date: " << birth
2         << "hired on: " << hired
3         << "ss #: " << ss;
```

▲

▼ Homework 3.11b: define an `operator<<` friend for classes life, point, and employee

Define an `operator<<` friend for classes `life` (pp. 145−146), `point` (pp. 201−204), and `employee` (pp. 257−262). Remove their `print` member functions.

For class `point`, our `operator<<` will eventually let us produce output in either Cartesian or polar coördinates (pp. 362−366). And for all classes, converting from `print` to `operator<<` will let us

direct output to any destination, not merely to the `cout` hardwired into the `print` functions.

But there is one place where we lose functionality. The `print` function of class `life` took arguments letting us specify the two characters with which to draw the picture (p. 146). We even provided default values for them. But an `operator<<` function has no room for extra arguments. It always takes the same pair: a read/write reference to an `ostream` and a read-only reference to the object being printed.

We hasten to assure the reader that this loss is only temporary. We will regain it with the same machinery that lets us format a `point` in Cartesian or polar. See pp. 367−371.

▲

▼ Homework 3.11c: what functions are called by these expressions?

If `a`, `b`, and `c` were objects, what functions would be called by the following expressions and in what order? Assume that the `operator*` and `operator+` are not member functions. Write an expansion for each expression as in the above lines 5−7, 13−15.

```
1     a + b + c
2     a * b + c * d
```

▲

3.12 Put it All Together: A Constrained Class

Look and feel

An SAT score (“Scholastic Aptitude Test”) is an integer that is a multiple of 10 in the range 200 to 800 inclusive. An `sat` object has the look and feel of an `int`, except that its value must be a legal SAT score. Whenever we *use* the value of an `sat`, we are actually using the return value of its member function `operator int` in line 23 of `sat.h`.

```
1     sat s = 700;
2
3     cout << s << "\n";        //cout << s.operator int() << "\n";
4     int i = s;                //int i = s.operator int();
5     if (s <= 700) {           //if (s.operator int() <= 700) {
```

But whenever we *change* the value of an `sat`, we are calling the `sat`'s `operator=` member function in line 5 of `sat.C`, or another member function that ultimately calls this one. The `operator=` does bounds checking, allowing an `sat` object to police itself.

```
6     s = 600;                  //s.operator=(600);
7     s += 20;                  //s.operator+=(20);
8     ++s;                      //s.operator++();
9     if (++s <= 700) {         //if (s.operator++().operator int() <= 700) {
10    cin >> s;                 //operator>>(cin, s);
```

Most of the functions that would normally need to be members or friends are neither. Only one member function is non-inline, the `operator=` in line 9 of the following `sat.h`.

The prefix operators in lines 19−20 return `*this`. `*this` is not going to evaporate when they return, so it can be returned by reference. But the postfix operators in lines 23−24 return a local variable. The local variable will evaporate when they return, so it must be returned by value.

It is no sin for one member function to call another member function of the same class. The member functions are all inline, so we waste no time. Ditto for the friends.

(1) In lines 15 and 16 of `sat.h`, the expression `s` that is an operand of the `+` and `-` implicitly calls the `operator int` in line 11. The `+` and `-` do not call the `operator+` and `operator-` in 26 and 28; these functions have not yet been seen.

(2) The assignment operator = in lines 10, 15, and 16 of `sat.h` calls the `operator=` in line 9: its left operand is an `sat`. But the assignment in line 12 of `sat.C` does not call this function: its left operand is an `int`.

(3) The operator += in lines 19, 26, and 27 of `sat.h` calls the `operator+=` in line 15.

(4) The operator -= in lines 16 and 28 of `sat.h` calls the `operator-=` in line 16.

(5) The prefix ++ in line 23 calls the prefix `operator++` in line 19.

(6) The prefix -- in line 24 calls the prefix `operator--` in line 20.

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

```
 1 #ifndef SATH
 2 #define SATH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class sat {
 7     int n;
 8 public:
 9     sat& operator=(int i);
10     sat(int new_n = 200) {*this = new_n;} //(*this).operator=(new_n);
11     operator int() const {return n;}
12 };
13
14 //Reference argument is read/write.  return s.operator=(s.operator int() + i);
15 inline sat& operator+=(sat& s, int i) {return s = s + i;}
16 inline sat& operator-=(sat& s, int i) {return s = s - i;}
17
18 //Prefix operators
19 inline sat& operator++(sat& s) {return s += 10;}
20 inline sat& operator--(sat& s) {return s -= 10;}
21
22 //Postfix operators
23 inline const sat operator++(sat& s, int) {const sat old = s; ++s; return old;}
24 inline const sat operator--(sat& s, int) {const sat old = s; --s; return old;}
25
26 inline const sat operator+(sat s, int i) {return s += i;}
27 inline const sat operator+(int i, sat s) {return s += i;}
28 inline const sat operator-(sat s, int i) {return s -= i;}
29
30 istream& operator>>(istream& i, sat& s);
31 #endif
```

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

```
 1 #include <cstdlib>
 2 #include "sat.h"
 3 using namespace std;
 4
 5 sat& sat::operator=(int i)
 6 {
 7     if (i < 200 || i > 800 || i % 10 != 0) {
 8         cerr << "sat can't contain " << i << ".\n";
 9         exit(EXIT_FAILURE);
```

```
10     }
11
12     n = i;
13     return *this;
14 }
15
16 istream& operator>>(istream& istr, sat& s)
17 {
18     int i;           //uninitialized variable
19
20     if (istr >> i) { //if (istr.operator>>(i).operator void *()) {
21         s = i;       //s.operator=(i);
22     }
23
24     return istr;
25 }
```

The following two public member functions are defined for us implicitly, so we do not have to write them. The argument of the `operator=` has the same data type as the object that the `operator=` belongs to. If we want an `operator=` with a different argument, we'll have to write it ourselves as in the above lines 5−14.

```
26 public:
27     sat(const sat& another) {n = another.n;}
28     sat& operator=(const sat& another) {n = another.n; return *this;}
```

▼ Homework 3.12a: create class printable

Create a class `printable` having the look and feel of a `char`, except that it can hold only printable values. Class `printable` will have exactly one data member, a private, non-static `char` named `c`. Imitate class `sat`.

Define the following public `operator=`. For the single cast in line 7, see pp. 63−64; for the double cast in 9, see p. 64.

```
 1 #include <iostream>
 2 #include <cctype>  //for isprint
 3 #include "printable.h"
 4 using namespace std;
 5
 6 printable& printable::operator=(char new_c)
 7 {
 8     if (isprint(static_cast<unsigned char>(new_c)) == 0) {
 9         cerr << "character code "
10             << static_cast<unsigned>(static_cast<unsigned char>(new_c))
11             << " is not printable\n";
```

The above `operator=` will detect the out-of-range value in the following line 17, but might miss the one in 18. The `rand` function returns an `int`, converted to `char` when passed to the `operator=`. At this point, there are two routes to failure on machines where `char` is narrower than `int`. If `char` is signed and too small to hold the random number, the result of this conversion will be ''implementation defined''. If `char` is unsigned, the conversion might just happen to yield a printable character.

```
12 #include <cstdlib>  //for rand
13 #include "printable.h"
14 using namespace std;
15
```

```
16      printable p(argument(s) for constructor);
17      p = '\a';        //p.operator('\a'); alarm character is not printable
18      p = rand();
```

To avoid the conversion from `int` to `char` in line 18, define the additional public `operator=` in line 22. To call `isprint` safely with an arbitrary `int`, we need the preliminary tests in line 24. The `int` passed to `isprint` must be the value `EOF` (“end-of-file”) or a number that can be held in an `unsigned char`; otherwise `isprint` could crash the program with a clear conscience (pp. 63–64).

```
19 #include <climits> //for UCHAR_MAX, the maximum value for unsigned char
20 #include "printable.h"
21
22 printable& printable::operator=(int new_c)
23 {
24     if (new_c < 0 || new_c > UCHAR_MAX || isprint(new_c) == 0) {
25         cerr << "character code " << new_c << " is not printable\n";
```

The two `operator=`’s will be the only member functions of class `printable` that call `isprint`. If desired, they can call a common subroutine, or one could call the other. Do not bother to define the `operator=` that takes a `const printable&`; it has already been defined for you implicitly.

Give class `printable` a public constructor that takes a `char` and passes it to the `operator=` that takes a `char`; and a public constructor that takes an `int` and passes it to the `operator=` that takes an `int`. Also give class `printable` a public `operator char`.

The following ten functions will be neither members nor friends. Declare all of them in the `printable.h` file. `operator>>` is the only one too big to be inline; define it in `printable.C`. Define the others in `printable.h`.

The `operator>>` that inputs a `printable` should call the `operator>>` that inputs a `char`. Since the latter skips whitespace, the former will too. (To turn the skipping off, see `noskipws` on p. 359.) The `operator>>` that inputs a `printable` will then pass the `char` to the `printable::operator=` that takes a `char`.

`operator+=` *and* `operator-=` *whose argument is an* `int`
`operator++` *(prefix and postfix)*
`operator--` *(prefix and postfix)*
`operator>>` *perform* `char` *input and then assign the* `char` *to the* `printable`
`operator+` *add a* `printable` *and an* `int`*, yielding a* `printable`
`operator+` *add an* `int` *and a* `printable`*, yielding a* `printable`
`operator-` *subtract an* `int` *from a* `printable`*, yielding a* `printable`

We will turn `printable` into a “template class” on pp. 735–738, and incorporate it into the Rabbit Game on pp. 740–745.

▲

3.13 A Model for Operator Overloading

Whenever we define `operator` functions for a class, we face the same four decisions.

(1) Must the `operator` be a member function or a friend? If so, which should it be?

(2) If the `operator` is a member function, should it be `const`?

(3) If the `operator` takes an object as an explicit argument, must it be passed by value or can it be passed by reference? If the later, should the reference be read-only or read/write?

(4) If the `operator` function returns an object, must it be returned by value or can it be returned by reference? If the later, should the reference be read-only or read/write?

Fortunately, these issues will be decided the same way for almost every class. In fact, the `operator` functions are so stereotyped that we can provide a copy-and-paste model for their declarations and even for some of their definitions. Later we will see a similar model called a ‘‘template’’. The dummy classname `T` in line 6 will reappear in our templates as the conventional name for a dummy data type.

Define `operator==` or `operator!=`, whichever is easier. It should be a friend function, since there are two objects. The other one can be implemented as a call-through that is neither a member nor a friend (lines 16 and 28). Similarly, `operator>` can be implemented as a call-through to `operator<` (lines 17 and 29), and `operator>=` as a call-through to `operator<=` (lines 18 and 30). The `operator-` that takes two objects in line 19 should be a friend just like the functions in lines 16−18.

Objects are usually passed to an `operator` function by reference to avoid the expense of copying them. Plentiful examples are in lines 16−19, 24−25, 28−30. The reference is always read-only, except for the object passed to `operator>>` in line 25. But the function in lines 32−34 must construct a new object, one that is most easily constructed by starting with a copy of an argument. In this case the new object may be constructed by passing the argument by value.

The member functions in lines 9−12 must return the object to which they belong. This is done by saying `return *this;`. The object will not evaporate as the member function returns, so the functions can get away with return-by-reference. We return a read/write reference to permit the object to be modified after it is returned. An example is in line 20 of `main.C` on p. 998.

```
1     ++ob %= 10;  //modify ob after it is returned by operator++.
```

If a function constructs and returns a new object, it must return the new object by value. See lines 32−34, 39−40. In the postfix functions (lines 39−40), the new object is constructed with a declaration (`const T old = t;`). In lines 32−34, the new object is constructed by passing an existing object by value. The new object must be `const` to prevent it from being modified after it is returned.

```
2   (ob1 + 10) = 20;                  //don't let this compile
```

The argument of the `operator=` in line 9 does not necessarily have to be another object of the same class. There can be several `operator=`’s, each with an argument of a different data type.

The postfix operators in lines 39−40 should *always* do their work by calling the prefix ones in lines 36−37. This ensures that the increments and decrements will be identical, apart from the time at which they are performed.

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

```
 1 #ifndef TH   //This is not C++ code, just a model to copy and paste.
 2 #define TH
 3 #include <iostream>   //for ostream and istream
 4 #include <cstddef>    //for size_t
 5 using namespace std;
 6
 7 class T {
 8 public:
 9     T& operator=(const T& another);
10
11     T& operator+=(int i);
12     T& operator-=(int i);
13
14     operator int() const;
15
16     friend bool operator==(const T& t1, const T& t2);
17     friend bool operator< (const T& t1, const T& t2);
18     friend bool operator<=(const T& t1, const T& t2);
19     friend int  operator- (const T& t1, const T& t2);
```

```
20
21      int& operator[](size_t i);
22      const int& operator[](size_t i) const;
23
24      friend ostream& operator<<(ostream& ost, const T& t);
25      friend istream& operator>>(istream& ist,       T& t);
26  };
27
28  inline bool operator!=(const T& t1, const T& t2) {return !(t1 == t2);}
29  inline bool operator> (const T& t1, const T& t2) {return t2 < t1;}
30  inline bool operator>=(const T& t1, const T& t2) {return t2 <= t1;}
31
32  inline const T operator-(T t, int i) {return t -= i;}
33  inline const T operator+(T t, int i) {return t += i;}
34  inline const T operator+(int i, T t) {return t += i;}
35
36  inline T& operator++(T& t) {return t += 1;}
37  inline T& operator--(T& t) {return t -= 1;}
38
39  inline const T operator++(T& t, int) {const T old = t; ++t; return old;}
40  inline const T operator--(T& t, int) {const T old = t; --t; return old;}
41  #endif
```

For some classes, the prefix `operator++` in the above line 36 could be implemented most simply by calling the `operator+=` in line 11. Such was the case with the class `date` with one data member (`day`) and class `sat` (line 19 of `sat.h` on p. 342.)

```
42  inline T& operator++(T& t) {return t += 1;}   //return t.operator+=(1);
```

For other classes, `operator+=` could be implemented by calling the prefix `operator++`. Such was the case with the original class `date` with three data members (`year`, `month`, and `day`) and class `life`.

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

```
 1  //Excerpt from T.C
 2  #include "T.h"
 3
 4  T& T::operator+=(int i)
 5  {
 6      for (; i > 0; --i) {
 7          ++*this;   //(*this).operator++();
 8      }
 9
10      for (; i < 0; ++i) {
11          --*this;   //(*this).operator--();
12      }
13
14      return *this;
15  }
```

Other decisions may vary as well. The `operator>>` we wrote for class `date` needed to mention the private members of that class, so we made it a friend (pp. 338−339). But the `operator>>` function for class `sat` mentioned no private members, so it was neither a member function nor a friend (p. 343). Ditto for the `operator>>` for class `printable` (p. 344).

The member function `operator->` can often be implemented by calling `operator*`; see p. 823.

Our example uses the data type `int` for two distinct purposes. It is used for numbers that are added to or subtracted from a `T` object, and for numbers that represent the distance between two `T` objects. See lines 11−12, 19, 32−34 of the above `T.h`. To show the intent of these numbers, we can make a typedef for their data type. The C++ convention is to use the name `difference_type` for the integral data type that is added to or subtracted from an object or that measures the distance between two objects, at least when the objects are ‘‘iterators’’.

```
 1 typedef int difference_type;
 2
 3 class T {
 4 public:
 5     T& operator+=(difference_type d);
 6     friend difference_type operator-(const T& t1, const T& t2);
 7     //etc.
 8 };
 9
10 inline const T operator+(T t, difference_type d) {return t += d;}
```

Classes `date` and `life` could also benefit from a `difference_type` typedef.

The data type `int` was also used for the little values that are contained in a `T` object; see lines 21−22 of the above `T.h` The C++ convention is to use the name `value_type` for the little values that are contained in a larger object.

```
11 typedef int value_type;
12
13 class T {
14 public:
15     value_type& operator[](size_t i);
16     const value_type& operator[](size_t i) const;
17     //etc.
18 };
```

See the `value_type` in class `stack` on pp. 153−154. Eventually these typedefs will become members of their classes; see the one in line 17 of `clinton.h` on p. 420.