Fall 2004 Handout 7

A date class

Here is a program that performs a simple calendar computation. See pp. 224–242. To avoid complications, we make three false assumptions:

(1) There are no leap years.

(2) There was a Year Zero between 1 B.C. and 1 A.D.

(3) We've always been using the Gregorian calendar, not the Julian calendar. There was no switch-over in September 1752:

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

We will rewrite the program two times. The original version has individual variables, Version 2 has a structure, and Version 3 will have our first object. All three versions produce the same output.

Introducing an object into the following program is like using a sledgehammer to kill ants. The program is too simple to need objects, but I don't want to burden you with a complicated example. In fact, a job this simple would normally all be done in **main**. I split it into four separate functions only to make it easier to introduce objects in Version 3. So even in Version 1 we are using more machinery than we need.

Version 1: a program with individual variables

Lines 42 and 50 define two different functions with the same name. We can get away with this because they have different arguments.

The first three arguments in line 35 must be the pointers to permit the **date_next** function to change the values of the variables in lines 31–33. On the other hand, the three arguments in line 37 do not have to be pointers: pass-by-value would have worked just as well, because the **date_print** function does not attempt to change the values of the variables. Line 37 passes three pointers only because it will be easier to introduce obejcts if all the functions take the same kind of arguments.

To ensure that **date_print** does not use the pointers to change the values ofthe variables, the pointers are declared to be read-only in line 62. Note that we have to write the keyword **const** three times (six, if you include line 23).

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9264/src/version1.C**

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 const int date_length[] = {
 6        0, //dummy element so that January will have subscript 1
 7       31, //January
 8       28, //February, ignoring for now the possibility of leap year
 9       31, //March
10       30, //April
```

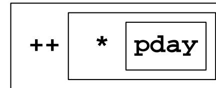
 All rights reserved ©2004 Mark Meretzky

```
11     31, //May
12     30, //June
13     31, //July
14     31, //August
15     30, //September
16     31, //October
17     30, //November
18     31  //December
19 };
20
21 void date_next(int *pday, int *pmonth, int *pyear, int count);
22 void date_next(int *pday, int *pmonth, int *pyear);
23 void date_print(const int *pday, const int *pmonth, const int *pyear);
24
25 int main()
26 {
27     cout << "How many days forward from January 1, 2003 do you want to go?\n";
28     int count;   //uninitialized variable
29     cin >> count;
30
31     int year = 2003;
32     int month = 1;   //1 to 12 inclusive
33     int day = 1;     //1 to date_length[month] inclusive
34
35     date_next(&day, &month, &year, count);
36     cout << "The new date is ";
37     date_print(&day, &month, &year);
38     cout << ".\n";
39     return EXIT_SUCCESS;
40 }
41
42 void date_next(int *pday, int *pmonth, int *pyear, int count)
43 {
44     //Call date_next count times.  Pass along the three pointers we received.
45     while (--count >= 0) {
46         date_next(pday, pmonth, pyear);
47     }
48 }
49
50 void date_next(int *pday, int *pmonth, int *pyear)
51 {
52     //Move to the next date.
53     if (++*pday > date_length[*pmonth]) {
54         *pday = 1;
55         if (++*pmonth > 12) {
56             *pmonth = 1;
57             ++*pyear;
58         }
59     }
60 }
61
62 void date_print(const int *pday, const int *pmonth, const int *pyear)
63 {
64     cout << *pmonth << "/" << *pday << "/" << *pyear;
```

 All rights reserved ©2004 Mark Meretzky

```
65 }
```

The **++** in the above line 53 adds 1 to ***pday**, not to **pday**. The box around the subexpression ***pday** causes the **++** outside it to treat the ***pday** inside it as a unit. The **++** cannot reach into the box to single out the sub-subexpression **pday**.



Line 53 therefore does the work of the following two lines:

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

Line 64 outputs only the last two digits of the year. It divides the year by 100, yielding a remainder in the range 0–99 inclusive.

```
How many days forward from January 1, 2004 do you want to go?
280
The new date is 10/8/2004.
```

Version 2: consolidate the variables into a structure

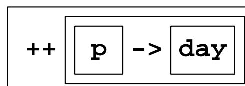
—Source code on the Web at **http://i5.nyu.edu/˜mm64/x52.9264/src/version2.C**

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 const int date_length[] = {
 6       0, //dummy element so that January will have subscript 1
 7      31, //January
 8      28, //February
 9      31, //March
10      30, //April
11      31, //May
12      30, //June
13      31, //July
14      31, //August
15      30, //September
16      31, //October
17      30, //November
18      31  //December
19 };
20
21 struct date {
22      int year;    //In what order are year, month, day created?
23      int month;   //1 to 12 inclusive
24      int day;     //1 to date_length[month] inclusive
25 };
26
27 void next(date *p, int count);
28 void next(date *p);
29 void print(const date *p);
30
```

 All rights reserved ©2004 Mark Meretzky

```
31 int main()
32 {
33     cout << "How many days forward from January 1, 2003 do you want to go?\n";
34     int count;              //uninitialized variable
35     cin >> count;
36
37     date d = {2003, 1, 1};   //C++ doesn't need the word "struct" in front.
38
39     next(&d, count);
40     cout << "The new date is ";
41     print(&d);
42     cout << ".\n";
43     return EXIT_SUCCESS;
44 }
45
46 void next(date *p, int count)
47 {
48     //Call next count times.  Pass along the pointer we received.
49     while (--count >= 0) {
50         next(p);
51     }
52 }
53
54 void next(date *p)
55 {
56     //Move to the next date.
57     if (++p->day > date_length[p->month]) {
58         p->day = 1;
59         if (++p->month > 12) {
60             p->month = 1;
61             ++p->year;
62         }
63     }
64 }
65
66 void print(const date *p)
67 {
68     //cout << (*p).month << "/" << (*p).day << "/" << (*p).year;
69     cout << p->month << "/" << p->day << "/" << p->year;
70 }
```

The **++** in the above line 57 adds 1 to **p->day**, not to **p**. The box around the subexpression **p->day** causes the **++** outside it to treat the **p->day** inside it as a unit. The **++** cannot reach into the box to single out the sub-subexpression **p**.



Line 57 therefore does the work of the following two lines:

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

The parentheses in line 68 force the star operator to be executed before the dot operator. But the arrow operator in line 69 is a simpler way to do the same thing. It does the work of the star and dot,

 All rights reserved ©2004 Mark Meretzky

eliminating the need for the parentheses.

Version 2 runs faster than version 1 because it passes fewer arguments to the functions. But the bodies of the functions have become more complicated: the arrows in Version 2 are even more confusing than the stars in Version 1.

As a footnote, Version 2 was able to simplify the function names. Version 1 had to add **date_** to the name of each function, because we might need other functions taking arguments of the same data types:

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

But Version 2 can overload the function names, because each function now takes an argument of a distinctive data type:

```
75 void print(const date *p);
76 void print(const time *p);
```

We'll change the name of the array **date_length** to **length** when we do **static** data members (pp. 641–648;
pp. 228–229, 249).

Version 3: an object instead of a structure

Our program does all its work in **main**, and has only three major variables: **year**, **month**, and **day**. But suppose it was big enough to divide into functions, and suppose it had enough variables to make it worthwhile to clump them together into structures. Then our problem would be to pass these structures down to the functions where the real work was done.

This is the first thing that we will use objects for. An object is a structure which can be passed to a function as quickly as the structures in Version 2, using a notation even simpler than Version 1. See
pp. 30–51, 613–619;
pp. 32–34, 224–242.

The *class* of an object is its data type. For example, the object **d** in line 40 is of class **date**. We say "object" rather than "structure" and "class" rather than "data type" because the terminology of C++ is borrowed from the language Simula67 rather than from C. See
pp. 10, 38; *Design and Evolution*, pp. 31, 40, 49, 57.

Members of an object

In C, we speak of the *fields* of a structure; in C++, they are called the *members* of a class. The first three members of class **date** are declared in lines 22–24. They are variables, just like the fields of a structure in C.

But the next four members are functions, declared in lines 26–29. The fields of a C structure must all be variables, but the members of a C++ object can be functions—and that is the biggest difference between C and C++. The members that are variables are called *data members*; the members that are functions are called *member functions*. See
pp. 31–32, 616–617;
pp. 32–34, 224–225, 238–240. We'll explain the **public:** in line 25 shortly.

A data member is located inside the object to which it belongs, just like a structure field in C. But a member function has a different relationship to its object. Instead of being inside the object to which it belongs, a member function receives an invisible argument giving the *address* of its object. This implicit (invisible) argument is passed in addition to whatever explicit (visible) arguments there may be. For example, line 45 passes one argument to the **print** function in line 88; this argument is the address of the object **d**. Line 42 passes two arguments to the **next** function in line 68: the address of **d** and the value of **count**.

The variable **this** holds the value of the invisible argument, and can therefore be used only in the body of a member function. It is a pointer to the object to which the member function belongs. When called from line 45 for example, the **this** in line 90 is the address of the **d** in line 45. See

 All rights reserved ©2004 Mark Meretzky

pp. 636–641,
pp. 231, 278, 754;

A star in front of any pointer in C or C++ will give you the pointed-to object. The expression **`*this`** in line 91 is therefore the object to which the member function belongs.

Line 93 could be used to print the data members of the object. But don't write it: line 94 is a simpler way to do the same thing. In the body of a member function, a member with no arrow or dot in front of it is assumed to be a member of the object to which the member function has received the invisible pointer. When line 45 calls the **`print`** in line 88, for example, the **`month`** in lines 93 and 94 will be the **`month`** data member of the object **`d`** in line 45. And when line 42 calls the one-argument **`next`** in line 68, the zero-argument **`next`** in line 72 will be the zero-argument **`next`** member function of the object **`d`** in line 42. In other words, the invisible pointer that line 42 passes to line 68 will be passed along when line 72 calls line 76.

Line 29 is the declaration for **`print`**; line 88 is the definition. The definition must mention the class to which the member function belongs, since there might be another member function with the same name in another class. The double colon operator in line 88 appeared in Handout 1, pp. 5–6; it pastes together a last name and a first name.

A **`const`** member function is one that cannot change the value of any data member of its object. For example, **`print`** is declared to be **`const`** in lines 29 and 88. On the other hand, the **`next`** functions do change the data members of their object; that is in fact their *raison d'être*. See
pp. 631–634;
p. 229–230.

Public and private members

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

The public members of a class can be used by any function. The **`main`** function uses two public members: **`next`** in line 42 and **`print`** in line 45. But the private members of a class can be used only by the member functions of that class. For example, the private members **`month`**, **`day`**, **`year`** can be accessed in line 94 by the **`print`** function, but not in line 44 by **`main`**. See
pp. 30–33, 617–619;
pp. 32, 225–226.

It therefore takes more effort to plan a C++ class than a C structure. Any function can access any field of any structure. But you have to decide in advance which C++ functions can access the private members of a class, and you have to decide which of these functions will have read/write access or read-only access.

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

A constructor

A *constructor* is a member function with the same name as the class to which it belongs; ours is declared in line 26 and defined in lines 51–66. A constructor is called automatically when you create a new object. That's why line 40 has parentheses, while line 37 of Version 2 had {curly braces}: you always write parentheses around the argument(s) of a function.

When line 40 constructs **`d`**, the data members **`year`**, **`month`**, and **`day`** inside of **`d`** are created in the order in which they are declared in lines 22–24. For the time being, however, this is only of academic interest. No one cares in what order the three integers are created, because nothing happens when an integer is born.

 All rights reserved ©2004 Mark Meretzky

But the order will become important later if we let **year**, **month**, and **day** be objects in their own right, each with their own constructor. When that happens, the error checking that is now in lines 53–56 will be in the constructor for **month**, and the checking in lines 58–61 will be in the constructor for **day**. Lines 53–56 must be executed before lines 58–61, so we will have to construct **month** before **day**. Lines 22–24 cause this to happen now, even though it is not currently necessary, so there will be one less thing to change when our three integer data members become objects.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9264/src/version3.C**

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 const int date_length[] = {
 6      0, //dummy element so that January will have subscript 1
 7     31, //January
 8     28, //February
 9     31, //March
10     30, //April
11     31, //May
12     30, //June
13     31, //July
14     31, //August
15     30, //September
16     31, //October
17     30, //November
18     31  //December
19 };
20
21 class date {
22     int year;
23     int month;               //1 to 12 inclusive
24     int day;                 //1 to date_length[month] inclusive
25 public:
26     date(int initial_month, int initial_day, int initial_year);
27     void next(int count);   //Go count days forward.
28     void next();             //Go one day forward.
29     void print() const;      //Output date to cout; this is function declaration.
30 };                           //don't forget ; at end of class declaration
31
32 int main()
33 {
34     cout << "How many days forward from January 1, 2003 do you want to go?\n";
35     int count;   //uninitialized variable
36     cin >> count;
37
38     //Create d.  Initialize d's data members by calling d's
39     //constructor.  Give the constructor the three arguments shown.
40     date d(1, 1, 2003);
41
42     d.next(count);
43     cout << "The new date is ";
44     //cout << d.month << "/" << d.day << "/" << d.year;   //won't compile
45     d.print();
46     cout << ".\n";
47
```

 All rights reserved ©2004 Mark Meretzky

```
48     return EXIT_SUCCESS;
49 }
50
51 date::date(int initial_month, int initial_day, int initial_year)
52 {
53     if (initial_month < 1 || initial_month > 12) { //Why check month before day?
54         cerr << "bad month " << initial_month << "\n";
55         exit(EXIT_FAILURE);
56     }
57
58     if (initial_day < 1 || initial_day > date_length[initial_month]) {
59         cerr << "bad day " << initial_day << "\n";
60         exit(EXIT_FAILURE);
61     }
62
63     year = initial_year;
64     month = initial_month;
65     day = initial_day;
66 }
67
68 void date::next(int count)
69 {
70     //Call next count times.
71     while (--count >= 0) {
72         next(); //Call another member function of the same object: this->next();
73     }
74 }
75
76 void date::next()
77 {
78     //Move to the next date.
79     if (++day > date_length[month]) {
80         day = 1;
81         if (++month > 12) {
82             month = 1;
83             ++year;
84         }
85     }
86 }
87
88 void date::print() const   //This is a function definition.
89 {
90     //cout << "The address of this object is " << this << ".\n";
91     //cout << "The size of this object is " << sizeof *this << ".\n";
92
93     //cout << this->month << "/" << this->day << "/" << this->year;
94     cout << month << "/" << day << "/" << year;
95 }
```

Put the declaration and definition of a class in separate files

To use the same class in many different C++ programs without having to copy and paste, put the declaration of the class in a separate *header* file named after the class (p. 225). Put the definitions of the member functions of the class in a **.C** file, also named after the class. This file is sometimes called the class's *implementation* file.

 All rights reserved ©2004 Mark Meretzky

The definition of the **date_length** array also goes in the implementation file. A header file must never contain the definition of anything that occupies memory, except a function that is inline.

You might get error messages if you **#include**'d the same header file twice:

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

because the header file might contain statements that are legal to write once but not twice.

The same problem would occur even if you **#include**'d two different header files

```
3 #include <stdio.h>
4 #include <another.h>
```

if **another.h** contained the line **#include <stdio.h>**.

Use the directives in line 2, 3, and 18 (pp. 13–14, p. 216) to compile the header file only the first time that it's **#include**'d. This will be the only time that you use **#define**'s in C++. The macro **DATEH** counts as being defined after line 3 even though it contains only the null string.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9264/src/date/date.h**

```
 1 //This file is date.h.  It is the header file for class date.
 2 #ifndef DATEH
 3 #define DATEH
 4
 5 #include <iostream>
 6 using namespace std;
 7
 8 class date {
 9     int year;
10     int month;                //1 to 12 inclusive
11     int day;                  //1 to date_length[month] inclusive
12 public:
13     date(int initial_month, int initial_day, int initial_year);
14     void next(int count);     //Go count days forward.
15     void next();              //Go one day forward.
16     void print() const {cout << month << "/" << day << "/" << year;}
17 };
18 #endif
```

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9264/src/date/date.C**

```
 1 //This file is date.C.  It is the implementation file for class date.
 2
 3 #include <iostream>
 4 #include <cstdlib>
 5 using namespace std;
 6 #include "date.h"
 7
 8 const int date_length[] = {
 9      0, //dummy element so that January will have subscript 1
10     31, //January
11     28, //February
12     31, //March
13     30, //April
14     31, //May
15     30, //June
16     31, //July
```

 All rights reserved ©2004 Mark Meretzky

```
17     31, //August
18     30, //September
19     31, //October
20     30, //November
21     31  //December
22 };
23
24 date::date(int initial_month, int initial_day, int initial_year)
25 {
26     if (initial_month < 1 || initial_month > 12) {
27         cerr << "bad month " << initial_month << "\n";
28         exit(EXIT_FAILURE);
29     }
30
31     if (initial_day < 1 || initial_day > date_length[initial_month]) {
32         cerr << "bad day " << initial_day << "\n";
33         exit(EXIT_FAILURE);
34     }
35
36     year = initial_year;
37     month = initial_month;
38     day = initial_day;
39 }
40
41 void date::next(int count)
42 {
43     //Call next count times.
44     while (--count >= 0) {
45         next();
46     }
47 }
48
49 void date::next()
50 {
51     //Move to the next date.
52     if (++day > date_length[month]) {
53         day = 1;
54         if (++month > 12) {
55             month = 1;
56             ++year;
57         }
58     }
59 }
```

```
 1 //This file is main.C.
 2
 3 #include <iostream>
 4 #include <cstdlib>
 5 #include "date.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     cout << "How many days forward from January 1, 2004 do you want to go?\n";
```

 All rights reserved ©2004 Mark Meretzky

```
11      int count;          //uninitialized variable
12      cin >> count;
13
14      //Create d.  Initialize d's data members by calling d's
15      //constructor.  Give the constructor the three arguments shown.
16      date d(1, 1, 2004);
17
18      d.next(count);
19      cout << "The new date is ";
20      d.print();
21      cout << ".\n";
22
23      return EXIT_SUCCESS;
24 }
```

```
g++ -o ~/bin/prog main.C date.C
ls -l ~/bin/prog
prog
```

Create **~/bin/prog**.

Why an object is better than a struct

(1) The body of a member function of a class is written with a simpler notation than a function that takes a pointer to a structure:

```
55          if (++*pmonth > 12) {                  //Version 1, p. 21
59          if (++p->month > 12) {                 //Version 2, p. 23
81          if (++month > 12) {                    //Version 3, p. 27
```

(2) A member function of an object is called with one less explicit argument than a function that takes a pointer to a structure. You can emphasize which argument is the most important by writing it in front of the name of the function:

```
35      date_next(&day, &month, &year, count);   //Version 1, p. 21
39      next(&d, count);                         //Version 2, p. 22
42      d.next(count);                           //Version 3, p. 26
```

(3) For the time being, write a constructor member function for every class. This will make it impossible to create an object of that class without initializing it. But a **struct** can easily be created without being initialized, leaving it full of garbage.

(4) If a data member of an object receives the wrong value, it's easy to make a list of every possible suspect: it must be one of the non-**const** member functions. (See p. 226, "localization".) But if a member of a **struct** receives the wrong value, any function in the program might be guilty.

(5) If you change the **private** members of an object, only the member functions of the object would have to be rewritten, as shown below. No other function in the program would need to be changed. See the thought-provoking rule of thumb in p. 707, §23.4.3.4.

For example, if I change

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

in Version 3 to

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

then only the member functions of **date** (and not even all of them) would have to change. But if you

All rights reserved ©2004 Mark Meretzky

change the members of a structure, however, there's no easy way to list all the functions that would have to change.

Let's walk through a call to the **print** member function of a two-data-member **date** object that contains October 8, 2004. In line 31, the data member **julian** and the local variable **d** will both contain 281.

(1) The first time we do the **>** comparison in line 34, **m** is 1 and **d** is 281, which represents the outrageous date January 281, 2004.

(2) The second time we do the comparison in line 34, **m** is 2 and **d** has been reduced to 250, which represents the slightly less outrageous date February 250, 2004.

(3) The third time we do the comparison in line 34, **m** is 3 and **d** has been reduced to 222, which represents the even less outrageous date March 222, 2004.

(4) The tenth (and last) time we do the comparison in line 34, **m** is 10 and **d** has been reduced to 8, which represents the legitimate date October 8, 2004.

```
 1 date::date(int initial_month, int initial_day, int initial_year)
 2 {
 3     if (initial_month < 1 || initial_month > 12) {
 4         cerr << "bad month " << initial_month << "\n";
 5         exit(EXIT_FAILURE);
 6     }
 7
 8     if (initial_day < 1 || initial_day > date_length[initial_month]) {
 9         cerr << "bad day " << initial_day << "\n";
10         exit(EXIT_FAILURE);
11     }
12
13     year = initial_year;
14
15     for (julian = initial_day; --initial_month > 0;
16         julian += date_length[initial_month]) {
17     }
18 }
19
20 void date::next()
21 {
22     //Change to the next date.
23     if (++julian > 365) {
24         julian = 1;
25         ++year;
26     }
27 }
28
29 void date::print() const
30 {
31     int d = julian;      //compute day of month
32     int m;               //Why can't we declare m after the ( in line 34?
33
34     for (m = 1; d > date_length[m]; ++m) {
35         d -= date_length[m];        //d = d - date_length[m];
36     }
37
38     cout << m << "/" << d << "/" << year;
39 }
```

 All rights reserved ©2004 Mark Meretzky

The value of the expression **sizeof(date)** would also change when you switch from three data members to two, but that's harmless unless you say things like

```
40      if (sizeof(date) == 3 * sizeof(int)) {
```

But you have no business writing this **if**.

▼ Homework 7.1: modify the member functions of class date

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

(1) Write a new member function named **julian** that will return the **date**'s Julian date. The Julian date is an integer in the range 1 to 365 inclusive, giving the day of the year. For example,
The Julian date of January 1 (of any year) is 1.
The Julian date of January 31 (of any year) is 31.
The Julian date of February 1 (of any year) is 32.
The Julian date of December 31 (of any year, including 1 BC) is 365, since we're ignoring leap years.

julian must be a non-inline public member function of class **date**, taking no arguments and returning an **int**. Like **print**, it must be a **const** member function.

(2) A member function can easily call another member function of the same object. For example, the one-argument **next** function called the no-argument **next** in Handout 2, p. 27, line 72. But this moved the one-argument **next** only one day toward the answer with each iteration of its loop.

The one-argument **next** could be faster if it did all its work in-house, without calling the other function. Recall how the two-data-member class **date** had a **print** function that strode *one full month* toward the answer with each iteration of its loop. Do the same for the one-argument **next** function of the three-data-member class **date**. It must remain a non-**const**, non-inline public member function of class **date**, taking one **int** argument and returning **void**.

(3) Remove the no-argument **next** function. Provide a default value of 1 (one) for the argument of the one-argument **next** function. Remember that a default value is specified only in the function declaration, not in the function definition. See Handout 1, pp. 31–33.

(4) Our one-argument **next** function works only if its argument is non-negative. Make it work for a negative argument as well:

```
 1      date d(10, 8, 2004);
 2      cout << "280 days before ";
 3      d.print();
 4      cout << " is ";
 5      d.next(-280);
 6      d.print();
 7      cout << ".\n";
 8
 9      cout << "280 days after ";
10      d.print();
11      cout << " is ";
12      d.next(280);
13      d.print();
14      cout << "\n";
```

```
280 days before 10/8/2004 is 1/1/2004.
280 days after 1/1/2004 is 10/8/2004.
```

(We'll be able to produce the same output with fewer statements when we do operator overloading.)

(5) If the target date is 10 years in the future or the past, our **next** would have to loop 120 times to get through the 120 intervening months. Write a smarter **next** that gets to within one year of the target in a single bound, simply by dividing its argument by 365. The quotient will tell us how many years to travel;

 All rights reserved ©2004 Mark Meretzky

the remainder will tell us how many additional days. We could then get the rest of the way to the target with only at most 11 loops.

If the argument of **next** is a non-negative number, every machine will give us the same quotient and remainder. For example,

```
  1 / 365 ==  0                    1 % 365 ==    1
  2 / 365 ==  0                    2 % 365 ==    2
  3 / 365 ==  0                    3 % 365 ==    3
363 / 365 ==  0                  363 % 365 ==  363
364 / 365 ==  0                  364 % 365 ==  364
365 / 365 ==  1                  365 % 365 ==    0
```

But if the argument of **next** is a negative number, different machines will give us different quotients if the division truncates the quotient. It is of no consolation that every machine guarantees the equality

$quotient \times divisor + remainder = dividend$

—it means that the remainders will be different too! Some machines truncate the quotient downward (i.e., towards negative infinity), yielding a remainder in the range 0 to 364 inclusive:

`-1 / 365 == -1`	*(truncation)*	`-1 % 365 ==`	`364`
`-2 / 365 == -1`	*(truncation)*	`-2 % 365 ==`	`363`
`-3 / 365 == -1`	*(truncation)*	`-3 % 365 ==`	`362`
`-363 / 365 == -1`	*(truncation)*	`-363 % 365 ==`	`2`
`-364 / 365 == -1`	*(truncation)*	`-364 % 365 ==`	`1`
`-365 / 365 == -1`	*(no truncation)*	`-365 % 365 ==`	`0`

Other machines truncate the quotient towards 0, yielding a remainder in the range –364 to 0 inclusive:

`-1 / 365 == 0`	*(truncation)*	`-1 % 365 ==`	`-1`
`-2 / 365 == 0`	*(truncation)*	`-2 % 365 ==`	`-2`
`-3 / 365 == 0`	*(truncation)*	`-3 % 365 ==`	`-3`
`-363 / 365 == 0`	*(truncation)*	`-363 % 365 ==`	`-363`
`-364 / 365 == 0`	*(truncation)*	`-364 % 365 ==`	`-364`
`-365 / 365 == -1`	*(no truncation)*	`-365 % 365 ==`	`0`

See K&R p. 205, §A7.6;
pp. 143–144; ARM p. 72, §5.6.

What our **next** function needs is a remainder that is always in the range 0 to 364 inclusive. We therefore need a quotient that's truncated downwards on all platforms. But what the C++ Standard Library gives us is a **div** function that truncates the quotient towards zero, yielding a remainder in the range –364 to 0 inclusive. See
p. 661.

```
15 #include <cstdlib>
16
17     const div_t d = div(-1, 365);
18
19     cout << "quotient " << d.quot << "\n";
20     cout << "remainder " << d.rem << "\n";
```

```
quotient 0
remainder -1
```

To get a quotient that's truncated downwards, yielding a remainder in the rage 0 to 364 inclusive, we apply the following simple fix to the result of **div**:

```
21 #include <cstdlib>
22
```

 All rights reserved ©2004 Mark Meretzky

```
23      div_t d = div(-1, 365);
24
25      if (d.rem < 0) {
26          d.rem += 365;               //d.rem = d.rem + 365;
27          --d.quot;
28      }
29
30      cout << "quotient " << d.quot << "\n"
31          << "remainder " << d.rem << "\n";
```

```
quotient -1
remainder 364
```

(6) Do not remove the **print** member function or the constructor. Do not add any data members to class **date**. Do not use the value of the dummy array element **date_length[0]**. Ignore leap years. You get no credit if you handle leap years, even if you do it correctly.

Hand in **date.h** and **date.C**, in that order. Demonstrate that your new class **date** is correct by handing in the output of **http://i5.nyu.edu/~mm64/x52.9264/src/date/test1/main.C**.

▼ Homework 7.2: modify the data members of class date

Handout 2, pp. 32–33 showed how to change the number of data members of class **date** from three to two. Now change the three data members

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

to the single data member

```
4       int day;      //journey into night: # of days before or after Jan 1, 0 A.D.
```

For simplicity, we're pretending that there was a year 0 A.D. Therefore
day will contain 0 when the object contains January 1, 0 AD.
day will contain 1 when the object contains January 2, 0 AD.
day will contain 30 when the object contains January 31, 0 AD.
day will contain 31 when the object contains February 1, 0 AD.
day will contain 59 when the object contains March 1, 0 AD.
day will contain 364 when the object contains December 31, 0 AD.
day will contain 365 when the object contains January 1, 1 AD.
day will contain $365 \times 2004 = 731460$ when the object contains January 1, 2004 AD.
day will contain –1 when the object contains December 31, –1 AD.
day will contain –365 when the object contains January 1, –1 AD.

The above values were chosen to make your life easy. Just divide the data member by 365 to get the year; the remainder gives you the day of the year (in the range 0 to 364 inclusive, which almost gives you the Julian date). Write a division that will truncate downwards on all machines.

Class **date** must have no data member other than the **int day**. Do not create any global variables. Do not declare any local variable inside a function to be **static**. Do not use the value of the dummy array element **date_length[0]**. Ignore leap years.

Make no change in the names, argument types, return values, or **const**'ness of the public member functions of class **date** from the previous Homework. For example, the constructor for **date** must still take three arguments (month, day, year) even though class **date** now has only one data member. And the **print** member function must still print the **date** in the format **m/d/y**. The argument of the **next** member function must still be able to be negative, positive, or zero. The default value of this argument must still

 All rights reserved ©2004 Mark Meretzky

be **1**. The **julian** member function will still take no arguments and return an **int** in the range 1 to 365 inclusive. (If your **julian** member function needs an **if** or a **?:**, then you probably have a bug in your constructor and will get no credit for any part of this homework.)

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

Hand in **date.h** and **date.C**, in that order. Demonstrate that your new class **date** is correct by handing in the output of **http://i5.nyu.edu/~mm64/x52.9264/src/date/test/main.C**.

Two groups of variables in scope in a free function

A function which is not a member function is called a *free* function. In C, for example, every function is free. In C and C++, **main** is always free.

In the body of a free function, the following two groups of variables are in scope. See
pp. 391–395,
pp. 82–83.

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

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

When identifying a variable in the body of a free function, the computer considers the local variables before the global ones. This makes a difference when a local and a global variable have the same name. In this case, the local will hide the global (line 7), and we would need the scope resolution operator **::** to access the global (line 8). See
pp. 425–426,
p. 82.

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

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

Three groups of variables in scope in a member function

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

(1) The local variables.

(2) The members of the class.

(3) The global variables, which we now define to be those that are neither local nor members of the class.

When identifying a name in the body of a member function of a the computer first considers the locals, then the members of the class, and finally the globals. If two variables have the same name, the local will therefore hide the member (line 9), and the member will hide the global. In this case we would need the scope resolution operator **::** to access the member (line 10) and the global (line 13). As always, a local will hide a global.

 All rights reserved ©2004 Mark Meretzky

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

The above rules apply not only to variables and data members, but also to anything that can have a name: functions and member functions, **typedef**'s, **enum**'s, etc. (Keywords such as **main**, **this**, and **sizeof** do not count as names.) A local will hide a member with the same name (line 23), and a member will hide a global with the same name (line 26).

```
15 void print();    //Declaration for a function that is not a member function.
16
17 //Class date has a data member named day and a member function named print.
18
19 void date::next() const
20 {
21     enum {day, night};
22
23     cout << day << "\n";            //the local day in line 21
24     cout << date::day << "\n";      //the day member of class date
25
26     print();                        //the print member of class date
27     ::print();                      //the global print declared in line 15
28 }
```

An object can be broken into:
p. 226, "cheating"

> Hiding [i.e., making the data members **private**] is for the prevention of accidents, not the prevention of fraud.
>
> —Bjarne Stroustrup[1]

A determined intruder can change the values of the data members of an object even if the intruder is not a non-**const** member function of the object. For example, here is a different **main** function for Version 3:

```
1 int main(int argc, char **argv)
2 {
3     date d(1, 9, 2004);
4     char *const p = (char *)&d;  //Let the value of p be the address of d.
5
6     for (int i = 0; i < sizeof d; ++i) {
7         p[i] = '\0';             //Fill d with zeroes.
```

[1] Booch, p. 54.

 All rights reserved ©2004 Mark Meretzky

```
 8      }
 9
10      cout << "The new date is ";
11      d.print();
12      cout << ".\n";
13      return EXIT_SUCCESS;
14  }
```

To make the "type punning" in line 4 more conspicuous, write it with the following C++-style cast. The **static_cast** in Handout 1, p. 28 means that you're 100 percent certain it will work on all platforms; **reinterpret_cast** means that you're 98 percent certain. See
p. 182;
pp. 130, 256.

```
15      char *const p = reinterpret_cast<char *>(&d);
```

```
The new date is 0/0/00.
```

Declare a constructor

A constructor has four distinguishing features. See
pp. 34–39, 630–631, 689–702;
pp. 32, 226–228.

(1) A constructor's name is the same as that of its class. For example, in Version 3 the class is named **date** (p. 25, line 21) and the class's constructor is also named **date** (declared in line 26, defined in lines 51–66).

(2) A constructor never returns a return value. Don't even declare the constructor's return value to be **void**: just write nothing at all as in lines 26 and 51 of Version 3.

(3) Don't declare the constructor to be **const** (as in lines 29 and 88 of Version 3), even if it doesn't change the value of any data member.

(4) For the time being, a constructor must always be **public**, not **private**. (The constructor for the class **type_info** in p. 1030 and for class **Assoc** in p. 286 are exceptions.) If the constructor for a class was **private**, it could be called only by the member functions of another object of that class. In that case, who could have constructed the other object? This is called a "chicken and egg" situation.

(5) I follow the convention that the name of each argument of the constructor is the same as the name of the data member initialized by that argument, plus a leading **initial_**. But if you want to use the same name for the argument and the data member, here are two ways to do it:

```
 1  date::date(int month, int day, int year)
 2  {
 3      //Error checking removed for brevity.
 4      this->year = year;
 5      this->month = month;
 6      this->day = day;
 7  }

 8  date::date(int month, int day, int year)
 9  {
10      //Error checking removed for brevity.
11      date::year = year;
12      date::month = month;
13      date::day = day;
14  }
```

 All rights reserved ©2004 Mark Meretzky

For the time being, a constructor should initialize every member of the object. (The constructor for class **stack** will be an exception.) The number of arguments of the constructor does not necessarily have to be the same as the number of data members. For example, see the constructor for **date** with a **julian** data member, above.

Syntax for calling a constructor

See pp. 32–33, 226–228, 229, 245–246, 271, 284–286.

When calling a constructor with two or more arguments, we must always surround them with parentheses. See line 26.

When calling a constructor with exactly one argument, we have a choice. We can surround it with parentheses (line 29), or precede it with an equal sign (line 30). Both notations do the same thing. Write the parentheses in line 29 to emphasize that a function is being called to initialize the object. Write the equal sign to make the user think of the object as primarily a container for the value to the right of the equal.

To ease the transition to templates, we're encouraged to program in the same style with variables of all data types. We can therefore use either line 36 or 37 to create and initialize an **int** or any other built-in type.

—Source code on the Web at **http://i5.nyu.edu/~mm64/x52.9264/src/duo.C**

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class duo {
 6     int member1;
 7     int member2;
 8 public:
 9     duo(int initial_member1, int initial_member2);
10     duo(const duo& another);   //copy constructor
11 };
12
13 class mono {
14     int member1;
15 public:
16     mono(int initial_member1);
17 };
18
19 class zero {
20 public:
21     zero();
22 };
23
24 int main()
25 {
26     duo d1(10, 20);   //as in Version 3, line 40 (p. 26)
27     duo d2;           //won't compile
28
29     mono m1(10);      //as in Version 3, line 40 (p. 26)
30     mono m2 = 10;     //another way to do the same thing when the
31                       //constructor has exactly one argument
32
33     zero z1();        //creates no object, calls no constructor
```

 All rights reserved ©2004 Mark Meretzky

```
34      zero z2;          //call a constructor that has no arguments
35
36      int i = 10;
37      int j(20);        //another way to do same thing: Design & Evolut, p. 380
38
39      duo d3 = d1;      //call copy constructor: d3.duo(d1)
40
41      return EXIT_SUCCESS;
42 }
43
44 duo::duo(int initial_member1, int initial_member2)
45 {
46      cout << "constructor for duo\n";
47      member1 = initial_member1;
48      member2 = initial_member2;
49 }
50
51 duo::duo(const duo& another) //copy constructor
52 {
53      cout << "copy constructor for duo\n";
54      member1 = another.member1;
55      member2 = another.member2;
56 }
57
58 mono::mono(int initial_member1)
59 {
60      cout << "constructor for mono\n";
61      member1 = initial_member1;
62 }
63
64 zero::zero()
65 {
66      cout << "constructor for zero\n";
67 }
```

zero is a class with no data members, like **popOnEmpty** in p. 548, **Overflow** in p. 29. The above line 33 declares a function, not an object. See pp. 694–695.

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

Line 34 has no parentheses because it imitates the syntax of the declaration of an uninitialized **int**.

Copy constructor

A constructor whose only argument is a **const** reference to another object of the same class as the one being constructed is called a *copy constructor*. See pp. 238–239, 700–701; pp. 229, 245–246, 271, 283–284; ARM p. 264. For example, see the one declared in the above line 10, defined in lines 51–56, and called in line 39. I always use the name **another** for the argument of a copy constructor.

Even though it's a member function of **d3**, the copy constructor called in line 39 can also access the private members of **d1**. When called from line 39, the **another.member1** in line 54 is a member of **d1** and the plain **member1** in line 54 is a member of **d3**.

 All rights reserved ©2004 Mark Meretzky

A copy constructor must never receive its argument via pass-by-value. A function argument that is passed-by-value is always copied before the function is called. If the argument is an object, the copying is performed by the object's copy constructor. We would then go into an infinite loop: every call to the copy constructor would be preceded by a previous call to the copy constructor.

To ensure that the copy constructor does not change the value of **d1** in line 39, we declare its reference argument to be **const** in lines 10 and 51.

If you hadn't defined a copy constructor for class **duo**, the compiler would write one for you like this:

```
70 duo::duo(const duo& another)
71 {
72     //"memberwise" copy: copy each data member of another
73     //object into the corresponding member of the new object.
74
75     member1 = another.member1;
76     member2 = another.member2;
77 }
```

(Later in the course, we'll see that the compiler actually defines it as

```
78 duo::duo (const duo& another)
79     : member1(another.member1), member2(another.member2)
80 {
81 }
```

i.e., it *initializes* rather than *assigns to* each data member of the new object.) I defined my own copy constructor because the one provided automatically would not have line 53.

Bad things can happen when you initialize an object with a copy constructor that the compiler writes for you rather than with a copy constructor that you wrote yourself. But don't worry yet: you're safe as long as none of the data members of the object are pointers or iterators.

While learning C++, I often put a **cout** statement into a constructor just to make sure the constructor is called:

`constructor for duo`	*line 26. Must remove or comment out lines 27, 33.*
`constructor for mono`	*line 29*
`constructor for mono`	*line 30*
`constructor for zero`	*line 34*
`copy constructor for duo`	*line 39*

▼ Homework 7.3: why does the argument of a copy constructor have to be a reference?

Suppose that the argument of a copy constructor was passed by value:

```
1     duo(duo another);   //function declaration

2 duo::duo(duo another)                 //function definition
3 {
4     cout << "copy constructor for duo\n";
5     member1 = another.member1;
6     member2 = another.member2;
7 }
```

What now happens in the above line 39? How long does it take to happen? Or won't it even compile? See
p. 702, Exercise 14.5;
p. 271.
▲

 All rights reserved ©2004 Mark Meretzky

Call a constructor without declaring an object

We have made it sound as if a constructor can be called only in the declaration for an object. But a constructor can also be called to create an *anonymous* object—one that has no name. An object needs no name if it is used only once. This is true not only for objects but for all variables in C and C++. We'll use a **double** and a **date** as our examples.

(1) To print the square root of 2, all we have to do is to print the **double** returned by the **sqrt** function in line 1. There is no need to declare a variable **do** to hold the square root in line 3 before we print it in line 4.

Similarly, to print a **date**, all we have to do is to print the **date** returned by the constructor function in line 1. A constructor always returns the object that it constructed, even though we write no return value or **return** statement when defining the constructor. There is no need to declare a variable **da** to hold the **date** in line 3 before we print it in line 4.

(This example assumes that a **date** can be output with the same **<<** operator used to output a **double**. We'll be able to do this when we do operator overloading. For the time being, however, we have to output the date by writing line 2 instead of line 1.)

```
 1      cout << sqrt(2.0) << "\n";      cout << date(1, 9, 2004) << "\n";
 2                                          //date(1, 9, 2004).print(); cout << "\n";
 3      double do = sqrt(2.0);          date da(1, 9, 2004);
 4      cout << do << "\n";             cout << da << "\n";
```

(2) To pass the square root of 2 to a function **f**, all we have to do is to take the **double** returned by the **sqrt** function and pass it to **f** in line 5. There is no need to declare a variable **do** to hold the square root in line 7 before we pass it to **f** in line 8.

Similarly, to pass a **date** to a function **f**, all we have to do is to take the **date** returned by the constructor function and pass it to **f** in line 5. There is no need to declare a variable **da** to hold the **date** in line 7 before we pass it to **f** in line 8. (Thanks to function name overloading, the **f** that receives the **double** is not the same function as the one that receives the **date**.)

```
 5      f(sqrt(2.0));                   f(date(1, 9, 2004));
 6
 7      double do = sqrt(2.0);          date da(1, 9, 2004);
 8      f(do);                          f(da);
```

(3) To change the value of a variable **xdo** to the square root of 2, all we have to do is to take the **double** returned by the **sqrt** function and assign it to **xdo** in line 10. There is no need to declare a variable **do** to hold the square root in line 12 before we assign it to **xdo** in line 13.

Similarly, to change the value of a variable **xda** to a new **date**, all we have to do is to take **date** returned by the constructor function and assign it to **xda** in line 10. There is no need to declare a variable **da** to hold the new **date** in line 12 before we assign it to **xda** in line 13.

```
 9      double xdo = 10.0;              date xda(1, 1, 2004);
10      xdo = sqrt(2.0);                xda = date(1, 9, 2004);
11
12      double do = sqrt(2.0);          date da(1, 9, 2004);
13      xdo = do;                       xda = da;
```

(4) To return the square root of 2 from a function, all we have to do is to return the **double** returned by the **sqrt** function in line 14. There is no need to declare a variable **do** to hold the square root in line 16 before we return it in line 17.

Similarly, to return a **date** from a function, all we have to do is to return the **date** returned by the constructor function in line 14. There is no need to declare a variable **da** to hold the **date** in line 16 before we return it in line 17. (The useless variables in line 16 can now be **const**'s because we are sure they will never be changed before they are destructed in the very next line.)

 All rights reserved ©2004 Mark Meretzky

```
14      return sqrt(2.0);                  return date(1, 9, 2004);
15
16      const double do = sqrt(2.0);       const date da(1, 9, 2004);
17      return do;                         return da;
```

In the above line 14, we can omit the name of the constructor and the parentheses around the argument if there is exactly one argument:

```
18      return duo(10, 20);                //Must write the "duo" and the parentheses.
19      return mono(10);                   //Could write the "mono" and the parentheses,
20      return 10;                         //but don't have to.
```

This will work only if the constructor was not declared to be **explicit**. See pp. 35–36, 697, 781–782; pp. 284–286, 341, 435, 447, 475, 477, 478, 484, 493, 569, 585, 624, 636, 637, 640, 642, 649, 662, 681, 873, 880, 889, 892, 893, 894, 900, 901, 902, 903, 908, 911, 921, 925, 927, 928.

(5) A final warning. Although there would be no error message, never write the following call to the **sqrt** function in a statement all by itself in C or C++. It would return an anonymous **double**, and then discard the **double** without making any use of it. Similarly, never write the following call to the constructor function for class **date** in a statement all by itself. It would construct and return an anonymous **date**, and then discard the **date** without making any use of it.

```
21      sqrt(2.0);                         date(1, 9, 2004);
```

Can one constructor call another one for the same object?

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

The DI constructor attempts to call the ID constructor in line 13. We hope this will work because the syntax of line 13 imitates that of line 10, which successfully calls another member function of the same object. We saw one member function calling another member function of the same object as early as Handout 2, p. 27, line 72.

But this syntax will work only if the other function is not a constructor. If the other function *is* a constructor, as in line 13, we will be committing the blunder in the above line 21. Line 13 does not call another member function of the same object. It constructs and discards a separate, anonymous object, which has no effect on the object that the DI constructor is trying to construct.

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

```
 1 class myobj {
 2     int i;
 3     double d;
 4 public:
 5     myobj(int initial_i, double initial_d) {i = initial_i; d = initial_d;}
 6
 7     myobj(double initial_d, int initial_i) {
 8
 9         //Call another member function of the same object:
10         f(initial_i, initial_d);
11
12         //2 unsuccessful attempts to call another mem func of same object:
13         myobj(initial_i, initial_d);
```

 All rights reserved ©2004 Mark Meretzky

```
14          myobj::myobj(initial_i, initial_d);
15
16          this->myobj(initial_i, initial_d);      //won't compile
17
18          *this = myobj(initial_i, initial_d);
19      }
20
21      void f(int i, double d) const {}
22 };
23
24 int main()
25 {
26      myobj m1(10, 3.14159);          //ID: int and double
27      myobj m2(3.14159, 10);          //DI: double and int
28
29      return EXIT_SUCCESS;
30 }
```

How to do it

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

```
31 class myobj {
32      int i;
33      double d;
34      void init(int initial_i, double initial_d) {i = initial_i; d = initial_d;}
35 public:
36      myobj(int initial_i, double initial_d) {init(initial_i, initial_d);}
37      myobj(double initial_d, int initial_i) {init(initial_i, initial_d);}
38 };
```

Use an array of structures to avoid the repetition

```
 1 struct {
 2      char type;
 3      char usrperm[4];
 4      char grpperm[4];
 5      char othperm[4];
 6 } permission;
 7
 8 char perm[8][4] = {
 9      "---",  /* 0 */
10      "--x",  /* 1 */
11      "-w-",  /* 2 */
12      "-wx",  /* 3 */
13      "r--",  /* 4 */
14      "r-x",  /* 5 */
15      "rw-",  /* 6 */
16      "rwx"   /* 7 */
17 };
18
19          /* inside of loop through all the files in the directory */
20
```

 All rights reserved ©2004 Mark Meretzky

```
21          memcpy(permission.usrperm, perm[status.st_mode >> 6 & 07], 4);
22          memcpy(permission.grpperm, perm[status.st_mode >> 3 & 07], 4);
23          memcpy(permission.othperm, perm[status.st_mode >> 0 & 07], 4);
24
25          printf("%s%s%s",
26              permission.usrperm,
27              permission.grpperm,
28              permissions.othperm);
```

```
 1 char type;
 2
 3 typedef struct {
 4     char string[4];
 5     int shift;
 6 } trio_t;
 7
 8 trio_t a[] {
 9     {"---", 6},
10     {"---", 3},
11     {"---", 0}
12 };
13
14 #define N (sizeof a / sizeof a[0])
15
16 char perm[][4] = {
17     "---",  /* 0 */
18     "--x",  /* 1 */
19     "-w-",  /* 2 */
20     "-wx",  /* 3 */
21     "r--",  /* 4 */
22     "r-x",  /* 5 */
23     "rw-",  /* 6 */
24     "rwx"   /* 7 */
25 };
26
27         /* inside of loop through all the files in the directory */
28
29         for (i = 0; i < N; ++i) {
30             strcpy(a[i].string, perm[status.st_mode >> a[i].shift & 07]);
31             printf("%s", a[i].string);
32         }
```

□

 All rights reserved ©2004 Mark Meretzky