6
Exceptions

6.1 Technical Preliminaries

Exceptions are a new way of handling errors in C++. Our first example will require three preliminary details: unused arguments, ellipsis, and integer overflow.

A function argument whose value is unused

The second argument of this `print` function will eventually specify the base in which to print the first argument. But right now the second argument is unused and the function prints only in base 10. We can indicate that the lack of use is deliberate, and avoid the ‘‘unused argument’’ warning, by giving no name to the second argument in the function definition in p. 13.

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void print(int i, int);
 6
 7 int main()
 8 {
 9     print(100, 10);
10     return EXIT_SUCCESS;
11 }
12
13 void print(int i, int)
14 {
15     cout << i << "\n";
16 }
```

Even if the *value* of an argument is unused, its *data type* may still be relevant. If two or more functions share the same name, the data type of an argument can tell the computer which function to call. In fact, the mere presence or absence of an argument can identify the function.

Here are four examples of arguments whose values are not used, or, more radically, arguments that have no value at all. In each case, the only purpose of the argument is to allow us to overload the function name.

(1) A class can have two `operator++` member functions, one for prefix and one for postfix. What allows them to have the same name is that their arguments are different. See pp. 289−290.

When we write lines 20 and 24, the computer behaves as if we had written the comments alongside them. The prefix operator `++` in line 20 calls the `operator++` function with no argument, while the

postfix operator ++ in line 24 calls the `operator++` function with an `int` argument whose value (always zero) is ignored. (Ditto for the two `operator--` functions.)

```
17 #include "obj.h"   //pp. 179−180
18
19     obj ob = 10;
20     cout << ++ob << "\n";          //cout << ob.operator++() << "\n";
21
22     //At this point, the object contains 11.
23
24     cout << ob++ << "\n";          //cout << ob.operator++(0) << "\n";
25
26     //At this point, the object contains 12.
```

```
11          line 20
11          line 24
```

(2) The Standard Library contains a global `operator new` function that dynamically allocates memory for all data types.

```
27 void *operator new(size_t n);
```

To call a different function to allocate memory for objects of a specific class, we write an `operator new` member function for that class:

```
28 class myclass {
29     //etc.
30 public:
31     void *operator new(size_t n);
32 };
```

Since this class-specific `operator new` can get the size of the object by saying `sizeof (myclass)`, it has no need to use the argument `n`. The argument will be used only when other classes are derived from `myclass`; see pp. 416 and 501−503.

(3) The `operator new` memory allocation functions in the C++ Standard Library come in pairs. With the extra argument `nothrow` they indicate failure by returning zero; without the `nothrow`, they indicate failure by ‘‘throwing an exception’’.

When we write lines 35 and 36, the computer behaves as if we had written the comments in 38 and 39 respectively. The computer always passes a first argument of data type `size_t` to the `operator new` function. Any arguments in the parentheses after the operator `new` in lines 35−36 are passed along after the `size_t`. The only purpose of the `nothrow` argument is to let us have two functions with the same name. Its value is ignored. (Ditto for the pairs of `operator delete` functions.) See p. 625.

```
33 #include <new>          //for nothrow
34
35     int *const p = new int;
36     int *const p = new(nothrow) int;
37
38     //int *const p = operator new(sizeof (int));          //Line 35 does this.
39     //int *const p = operator new(sizeof (int), nothrow); //Line 36 does this.
```

In fact, the `nothrow` object has no value at all: it contains no members. Its declaration is

```
40 //Excerpt from the header file <new>
41
42 class nothrow_t {                  //a class with no members
```

```
43 };
44
45 extern const nothrow_t nothrow;     //the only object of this class
```

You'll have to wait until we do templates, iterator categories, and dispatching before the next two examples will make sense.

(4) The C++ Standard Library functions whose arguments are iterators are called *algorithms;* examples are `find`, `copy`, `sort`, and `distance`. An algorithm sometimes passes the iterators to a *helper function* to do its work. An algorithm might have several versions of its helper functions, one for each "category" of iterator.

The helper functions for an algorithm all have the same name and the same iterator arguments, but each takes an additional argument called an *iterator tag.* Like `nothrow`, an iterator tag object has no members and no value. But there are several classes of tags, one for each category of iterator, allowing the algorithm to call the correct helper via function name overloading. See pp. 916−917.

(5) Obsolete implementations of the C++ Standard Library, including Microsoft's, had many other arguments whose value was ignored but whose data type was relevant. Often these arguments were merely pointers to objects of different classes, containing the value `NULL` or `0`, to avoid the expense of constructing any actual objects. The clearest example in the literature is the third argument of the function `iter_swap_impl` in p. 43 of *Generic Programming and the STL.* Newer versions of C++ avoid these arguments by using the template class `iterator_traits`.

Ellipsis

Here is a C declaration for a function whose arguments, except for the first, can be of any number or data type. The ellipsis dots constitute a single token, so there can be no whitespace among them.

```
1 int printf(const char *format, ...);
```

In C++, the comma before the ellipsis is optional for some reason.

```
2 int printf(const char *format ...);
```

The only common functions with indeterminate arguments are `printf` and `scanf` and their cousins `fprintf`, `sprintf`, etc. The functions are vital in C, deprecated in C++. Unix system programmers know the ellipsis from the functions `execlp` and `execl` (rhymes with "Doctor Jekyll"), used to fork and spawn a process. In C++, ellipses will be used primarily for the "exception handlers" below.

Integer overflow

`INT_MIN` and `INT_MAX` are the minimum and maximum values of data type `int`. They may be different on each platform. But on all two's complement platforms, the quotient `INT_MIN / -1` will not fit in an `int`:

	16-bit integer `sizeof(int) == 2`	*32-bit integer* `sizeof(int) == 4`	*64-bit integer* `sizeof(int) == 8`
`INT_MIN`	−32,768	−2,147,483,648	−9,223,372,036,854,775,808
`INT_MAX`	32,767	2,147,483,647	9,223,372,036,854,775,807

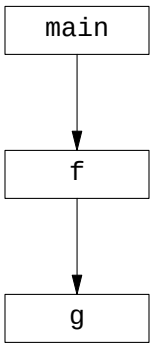
`INT_MIN` and `INT_MAX` are macros in the `<climits>` header file of the C++ Standard Library. But macros are deprecated in C++. We will get this information about integers from the template class `numeric_limits<int>` on pp. 745−747 instead.

6.2 Throw and Catch an Exception

C++ exceptions are a new way of responding to errors. They have nothing to do with operating system exceptions, interrupts, or signals.

There are four reasons to use exceptions.

(1) An exception transmits information from the point where an error was discovered to the point where remedial action is taken. These two locations may be far apart in your program. For example, let `main` call `f`, `f` call `g`, and `g` discover an error:



Our knee-jerk reaction has always been to output an error message and drop dead on the spot (lines 23−26):

```
 1 #include <iostream>
 2 #include <cstdlib>      //for exit and EXIT_FAILURE
 3 #include <cstring>      //for strcmp
 4 using namespace std;
 5
 6 void f();
 7 void g();
 8
 9 int main(int argc, char **argv)
10 {
11     const bool verbose = argc >= 2 && strcmp(argv[1], "-v") == 0;
12     f();
13     return EXIT_SUCCESS;
14 }
15
16 void f()
17 {
18     g();
19 }
20
21 void g()
22 {
23     if (something is wrong) {
24         cerr << "error message\n";
25         exit(EXIT_FAILURE);
26     }
27 }
```

But we might want to call the function `g` in many different programs. In this case we couldn't write the action in `g`, because each program might need a different error message or exit code. We would have to write the action up in one of the functions that called `g`: `f` or `main`.

Even if `g` were used only in this one program, we still might not be able to write the action in `g`. The program might run in two modes, terse and verbose, or English and Spanish, controlled by the `bool` in line 11. Since the `bool` is local to `main`, only `main` knows which message to print. Once again, we would have to write the action above `g`, up in `main`.

Of course, we could give return values to `g` and `f`, and bucket-brigade the result of the above line 23 back up to `main`:

```
28 int main(int argc, char **argv)
29 {
30     const bool verbose = argc >= 2 && strcmp(argv[1], "-v") == 0;
31
32     if (!f()) {
33         if (verbose) {
34             cerr << "verbose error message\n";
35         } else {
36             cerr << "terse error message\n";
37         }
38         return EXIT_FAILURE;
39     }
40
41     return EXIT_SUCCESS;
42 }
43
44 bool f()
45 {
46     return g();
47 }
48
49 bool g()
50 {
51     if (something is wrong) {
52         return false;
53     }
54
55     return true;
56 }
```

But this copies the return value of `g` over and over on its way up to `main`. Our return value is merely a `bool`, but other return values might be objects that are expensive to copy. In any case, it would be simpler for `g` and `f` to return `void`. Can we avoid burdening `g` and `f` with return values?

A C++ exception is a faster way to transmit information from `g` back up to `main`. It transmits the information directly from a lower-level function to one of the higher-level ones that called it. Along the way, the information is not repeatedly copied, like the `bool`'s in lines 49 and 52.

(2) Most of the time, an exception is *thrown,* or sent on its way, because something has gone wrong. But an exception can also be thrown whenever the information to be transmitted upwards won't fit into the normal channels, i.e., into the return type of the function. That is why it is called an ''exception'', not an ''error''. An exception gives a function an extra, high-bandwidth return type for unusual occasions, in addition to its normal return type.

(3) Every constructed object must be destructed, or disaster could result. Consider the humble `terminal::put` function in lines 36–47 of `terminal.C` on p. 161. It calls the `exit` function if its argument is a non-printable character, so the test program on pp. 157–159 would call `exit` if we type a

RETURN or any other non-printable character. Before terminating the program, `exit` will call the destructors for the statically allocated objects. But our `terminal` is allocated automatically, because it is a data member of an object that is allocated automatically, so the work done by the constructor for class `terminal` will never be undone. Our screen could be left in graphics mode, or with a derelict window.

(4) Even if we don't terminate the program, there's another reason why we might want to call destructors when an exception is thrown. Error handling in any language often requires us to *backtrack*, or dismantle some of our work to get back to a clean state from which we can continue onwards. In C++, the backtracking is performed by calling destructors. An exception will call the destructors for all the local objects in the functions that it flies over. For example, an exception thrown from `g` back up to `f` will cause a mass extinction of all the automatic objects in `g`.

The more severe the error is, the farther we have to backtrack. A very severe error in `g` will throw an exception all the way back up to `main`, destructing all the objects in `g` and `f`. A less severe exception may be caught halfway up to `main`, destructing fewer objects and backtracking less far.

Throw and catch an exception

An exception usually transmits information from one function to another. But our first example will transmit information within a function.

The keyword `try` and the following code in curly braces in line 20−45 is called a *try block*. The try block has to test for overflow before truncation because `INT_MIN %−1` is not equal to zero with my `g++` compiler.

The keyword `catch` and the following argument in parentheses and code in curly braces is called a *handler* (lines 47−49, 51−54, 56−59, 61−63). If any exception is `throw`'n inside a try block, we go directly to the appropriate handler in the list of handlers after the try block.

An automatically allocated variable is constructed with a declaration or as an anonymous temporary; a dynamically allocated variable is constructed with `new`. Any automatically allocated variable constructed in the `try` block (e.g., lines 22 and 26) will be destructed as we are catapulted out of the block. A dynamically allocated variable, however, will not destructed.

The handlers are tried one by one in the order in which they are written. The optional `...` handler in lines 61−63 must be last because it will catch any exception. Any handler that came after it would have no chance to catch anything.

If the code inside our `try { }` were somehow to throw an exception of any data type other than `int`, `truncation`, or `overflow`, you would go to the `...` handler. If there were no `...` handler, the `terminate` function would be called, which will call the `abort` function, which assassinates the program. (On my platform, Unix, it sends the ''abort signal'' `SIGABRT` to the program.) These two functions have already been written for you in the C++ Standard Library. To make the `terminate` function do something other than calling `abort`, you can write your own `terminate` function and make it operational by passing its address to the `set_terminate` function on pp. 614−615, analogous to the `set_new_handler` function on pp. 397−398. Include the header file `<exception>` for `set_terminate`.

Other than the `...` handler, every handler must have exactly one argument. And each handler must have an argument of a different type.

If the handler uses the value or members of the caught object, it must declare a name for the caught object (lines 47 and 51). But if the handler doesn't use them, or if the caught object contains no value or members, the handler doesn't have to declare a name for it (line 56).

After executing the body of one of the handlers, we go to the statement after the last handler (line 65). And if no exception has been thrown at all, we skip all the handlers and go straight from line 45 to line 65.

Line 14 is a declaration for a class that has no members.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/exception1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <climits>
 4 using namespace std;
 5
 6 struct truncation {
 7     int dividend;   //data members public for simplicity
 8     int divisor;
 9
10     truncation(int initial_dividend, int initial_divisor)
11         : dividend(initial_dividend), divisor(initial_divisor) {}
12 };
13
14 class overflow {};
15
16 int main()
17 {
18     int status = EXIT_FAILURE;   //guilty until proven innocent
19
20     try {
21         cout << "Please input the dividend and press RETURN: ";
22         int dividend;   //uninitialized variable
23         cin >> dividend;
24
25         cout << "Please input the divisor and press RETURN: ";
26         int divisor; //uuninitialized variable
27         cin >> divisor;
28
29         if (divisor == 0) {
30             throw dividend;
31         }
32
33         if (dividend == INT_MIN && divisor == -1) {
34             const overflow ov = overflow();
35             throw ov;
36         }
37
38         if (dividend % divisor != 0) {
39             const truncation t(dividend, divisor);
40             throw t;
41         }
42
43         cout << "The quotient is " << dividend / divisor << ".\n";
44         status = EXIT_SUCCESS;
45     }
46
47     catch (int i) {
48         cerr << "Attempt to divide " << i << " by zero.\n";
49     }
50
51     catch (truncation t) {
52         cerr << "Truncation would result when dividing " << t.dividend
53             << " by " << t.divisor << ".\n";
54     }
```

```
55
56      catch (overflow) {
57          cerr << "Integer overflow would result when dividing " << INT_MIN
58              << " by -1.\n";
59      }
60
61      catch (...) {
62          cerr << "Caught unexpected exception.\n";
63      }
64
65      return status;
66  }
```

Since the object `ov` in line 34 is `const` and has no user-defined default constructor (its class has no user-defined members at all), we must call its implicitly-defined constructor with explicit parentheses. See the C++ Standard, §8.5. We can't do this by writing

```
67      const overflow ov();
```

since that would look like a function call (pp. 134−135.) But don't worry: the declaration will shortly be removed.

To turn on exception handling in Microsoft Visual C++, select the ''Enable exception handling'' option in the C++ Language category of the C/C++ tab in the Project Settings dialog box, or use the `/GX` compiler switch.

```
Please input the dividend and press RETURN: 10
Please input the divisor and press RETURN: 5
The quotient is 2.
```

```
Please input the dividend and press RETURN: 10
Please input the divisor and press RETURN: 0
Attempt to divide 10 by zero.
```

```
Please input the dividend and press RETURN: −2147483648
Please input the divisor and press RETURN: −1
Integer overflow would result when dividing -2147483648 by -1.
```

```
Please input the dividend and press RETURN: 10
Please input the divisor and press RETURN: 3
Truncation would result when dividing 10 by 3.
```

Throw an object and catch it by reference

We destruct two objects as we pass line 16. To see which one is destructed first, line 15 tags one of them with a distinct value.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/throwobj1.C`

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

```
 7 {
 8     try {
 9         obj ob = 10;
10         throw ob;
11     }
12
13     catch (obj ob) {
14         cout << ob << "\n";
15         ++ob;
16     }
17
18     return EXIT_SUCCESS;
19 }
```

construct 10	*line 9*
copy construct 10	*line 10; let's call this one "the thrown object"*
destruct 10	*As we pass line 11, we destruct the object in line 9.*
copy construct 10	*line 13*
10	*line 14*
destruct 11	*As we pass line 16, we destruct the object in line 13.*
destruct 10	*Then line 16 destructs the thrown object.*

We can eliminate the underlined object by catching the thrown object by reference:

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/throwobj2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     try {
 9         obj ob = 10;
10         throw ob;
11     }
12
13     catch (const obj& ob) {
14         cout << ob << "\n";
15     }
16
17     return EXIT_SUCCESS;
18 }
```

construct 10	*line 9*
copy construct 10	*Line 10 constructs the thrown object.*
destruct 10	*Line 11 destructs the object in line 9.*
10	*line 14*
destruct 10	*Line 15 destructs the thrown object.*

6.3 Throw an Anonymous Object

Line 34 constructs an anonymous (i.e., nameless) object of class `overflow` by giving no arguments to its constructor. (Class `overflow` actually has no constructor, but you have to write line 34 as if it had a constructor with no arguments.) It then throws the anonymous object. Similarly, line 38 constructs an anonymous object of class `truncation` by giving two arguments to its constructor. It then throws the anonymous object.

Line 49 catches by reference to avoid constructing and destructing an unnecessary copy of the caught object. I didn't bother to catch by reference in lines 45 and 54, because those objects take almost no time to copy. We'll see another reason to catch by reference when we talk about exceptions and inheritance.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/exception2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <climits>
 4 using namespace std;
 5
 6 struct truncation {
 7     int dividend;
 8     int divisor;
 9
10     truncation(int initial_dividend, int initial_divisor)
11         : dividend(initial_dividend), divisor(initial_divisor) {}
12 };
13
14 class overflow {};
15
16 int main()
17 {
18     int status = EXIT_FAILURE;
19
20     try {
21         cout << "Please input the dividend and press RETURN: ";
22         int dividend;    //uninitialized variable
23         cin >> dividend;
24
25         cout << "Please input the divisor and press RETURN: ";
26         int divisor; //uninitialized variable
27         cin >> divisor;
28
29         if (divisor == 0) {
30             throw dividend;
31         }
32
33         if (dividend == INT_MIN && divisor == -1) {
34             throw overflow();
35         }
36
37         if (dividend % divisor != 0) {
38             throw truncation(dividend, divisor);
39         }
40
41         cout << "The quotient is " << dividend / divisor << ".\n";
42         status = EXIT_SUCCESS;
```

```
43      }
44
45      catch (int i) {
46          cerr << "Attempt to divide " << i << " by zero.\n";
47      }
48
49      catch (const truncation& t) {
50          cerr << "Truncation would result when dividing " << t.dividend
51              << " by " << t.divisor << ".\n";
52      }
53
54      catch (overflow) {
55          cerr << "Integer overflow would result when dividing " << INT_MIN
56              << " by -1.\n";
57      }
58
59      catch (...) {
60          cerr << "Caught unexpected exception.\n";
61      }
62
63      return status;
64  }
```

6.4 An Exception that Escapes from a Function

A *block* is a group of zero or more statements surrounded by {curly braces}. Examples are the body of a function, `for` loop, `if` statement, or `try` block.

When we throw an exception inside a block and catch it outside, or when we never catch it at all, we say that the exception has *escaped* from the block. Because of the *exception specification* in lines 17 and 58, the only exceptions that are allowed to escape from `f` are those of data types `int`, `overflow`, or `truncation`. If an exception of any other data type somehow escapes from `f`, the `unexpected` function would be called, which will call the `terminate` function, which will call the `abort` function.

Like `terminate` and `abort`, the `unexpected` function has already been written for us in the C++ Standard Library. But if we want the `unexpected` function to do something other than calling `terminate`, we can write our own `unexpected` function and make it operational by passing its address to the `set_unexpected` function. (Include `<exception>` for `set_unexpected`.)

A function with no exception specification is allowed to have any exception escape from it. Exception specifications are therefore optional in C++, but mandatory in Java.

If an exception travels all the way up through `main` but is never caught anywhere, the program would automatically call the `terminate` function, which will call the `abort` function. In this case, the `unexpected` function is not called.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/exception3.C`

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cstring>   //for strcmp
4  #include <climits>
5  using namespace std;
6
7  struct truncation {
8      int dividend;
9      int divisor;
```

```
10
11      truncation(int initial_dividend, int initial_divisor)
12          : dividend(initial_dividend), divisor(initial_divisor) {}
13  };
14
15  class overflow {};
16
17  void f() throw (int, overflow, truncation);   //exception specification
18
19  int main(int argc, char **argv)
20  {
21      int status = EXIT_FAILURE;
22      const bool verbose = argc >= 2 && strcmp(argv[1], "-v") == 0;
23
24      try {
25          f();
26          status = EXIT_SUCCESS;
27      }
28
29      catch (int i) {
30          if (verbose) {
31              cerr << "Attempt to divide " << i << " by zero.\n";
32          }
33      }
34
35      catch (const truncation& t) {
36          if (verbose) {
37              cerr << "Truncation would result when dividing " << t.dividend
38                  << " by " << t.divisor << ".\n";
39          }
40      }
41
42      catch (overflow) {
43          if (verbose) {
44              cerr << "Overflow would result when dividing " << INT_MIN
45                  << " by -1.\n";
46          }
47      }
48
49      catch (...) {
50          if (verbose) {
51              cerr << "Caught unexpected exception.\n";
52          }
53      }
54
55      return status;
56  }
57
58  void f() throw (int, overflow, truncation)
59  {
60      cout << "Please input the dividend and press RETURN: ";
61      int dividend;    //uninitialized variable
62      cin >> dividend;
63
```

```
64      cout << "Please input the divisor and press RETURN: ";
65      int divisor; //uninitialized variable
66      cin >> divisor;
67
68      if (divisor == 0) {
69          throw dividend;
70      }
71
72      if (dividend == INT_MIN && divisor == -1) {
73          throw overflow();
74      }
75
76      if (dividend % divisor != 0) {
77          throw truncation(dividend, divisor);
78      }
79
80      cout << "The quotient is " << dividend / divisor << ".\n";
81  }
```

6.5 A Class whose Member Functions Throw Exceptions

Four things can go wrong in the member functions of the following class `date`:

(1) The `initial_month` argument of the constructor could be invalid.

(2) The `initial_day` argument of the constructor could be invalid.

(3) We could call an `operator++` member function (prefix or postfix) of a `date` that already contains the last possible date, December 31 of the year `INT_MAX`.

(4) We could call an `operator--` member function (prefix or postfix) of a `date` that already contains the earliest possible date, January 1 of the year `INT_MIN`.

We therefore create four exception classes,

```
date::bad_month
date::month_and_day
date::overflow
date::underflow
```

To remind us what these four classes are for, we give them the last name `date` by declaring them inside the {curly braces} in lines 6 and 77 of `date.h`.

But they can't be declared just anywhere within the curly braces. The declaration for a class must always come before any other mention of the class. In fact, the same is true of almost all declarations in C and C++. Before we declare the constructor of class `date` in lines 56–57, we must write the entire declaration, not merely a forward declaration, for class `bad_month` in lines 31–39, and for class `bad_month_and_day` in lines 41–51. The declaration for class `overflow` in line 53 must come before the declaration of the prefix `operator++` in line 59 and the postfix `operator++` in lines 62–66. And the declaration for class `underflow` in line 54 must come before the declaration of the prefix `operator--` in line 60 and the postfix `operator--` in lines 68–72.

We also provide each exception class with an `operator<<` function, so that the handlers can output them without bothering with their internal details. The `operator<<`'s that print classes `bad_month` and `bad_month_and_day` must be friends of these classes to access their private data members. But the `operator<<`'s that print classes `overflow` and `underflow` have no need of friendship, since those classes contain no data members at all. We declare them after line 77 to show that they are neither friends nor members of any class.

All four of the `operator<<` functions for the exception classes are inline, but only the two in lines 79−85 need the keyword `inline`. The other two are inline because they are defined, not merely declared, within the curly braces in lines 6−77. The keyword `inline` also makes the functions in lines 79−85 static, in the sense of being visible only within the `.C` file that includes `date.h`. Without the keyword, these functions would be ''multiply defined'' if `date.h` were included in more than one `.C` file.

Within the all-important curly braces in lines 6 and 77, we're on a first-name basis with the members of class `date`. That's why line 36 can get away with saying `bad_month`. But outside the curly braces, we have to address each member of class `date` by its full name. That's why line 79 needs the stiffly formal `date::overflow`.

As usual, the postfix `operator++` in lines 62−66 calls the prefix `operator++` in line 59. The prefix `operator++` throws an `overflow` exception which the postfix `operator++` does not catch. Therefore the postfix `operator++` throws the same exception.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/except1/date.h`

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 using namespace std;
 5
 6 class date {
 7     static const int length[];
 8
 9     int year;
10     int month;   //date::january to date::december inclusive
11     int day;     //1 to length[month] inclusive
12
13 public:
14     enum month_t {   //indices into the length array
15         january = 1,
16         february,
17         march,
18         april,
19         may,
20         june,
21         july,
22         august,
23         september,
24         october,
25         november,
26         december
27     };
28
29     //Exceptions thrown by the member functions of class date:
30
31     class bad_month {
32         const int month;
33     public:
34         bad_month(int initial_month): month(initial_month) {}
35
36         friend ostream& operator<<(ostream& ost, const bad_month& bm) {
37             return ost << "bad month " << bm.month;
38         }
39     };
```

```
40
41      class bad_month_and_day {
42          const int month;
43          const int day;
44      public:
45          bad_month_and_day(int initial_month, int initial_day)
46              : month(initial_month), day(initial_day) {}
47
48          friend ostream& operator<<(ostream& ost, const bad_month_and_day& bd) {
49              return ost << "bad month " << bd.month << ", day " << bd.day;
50          }
51      };
52
53      class overflow {};
54      class underflow {};
55
56      date(int initial_month, int initial_day, int initial_year)
57          throw (bad_month, bad_month_and_day);
58
59      date& operator++() throw (overflow);
60      date& operator--() throw (underflow);
61
62      const date operator++(int) throw (overflow) {
63          const date old = *this;
64          ++*this;   //(*this).operator++();
65          return old;
66      }
67
68      const date operator--(int) throw (underflow) {
69          const date old = *this;
70          --*this;   //(*this).operator--();
71          return old;
72      }
73
74      friend ostream& operator<<(ostream& ost, const date& d) {
75          return ost << d.month << "/" << d.day << "/" << d.year;
76      }
77  };
78
79  inline ostream& operator<<(ostream& ost, const date::overflow&) {
80      return ost << "can't go beyond December 31, " << INT_MAX;
81  }
82
83  inline ostream& operator<<(ostream& ost, const date::underflow&) {
84      return ost << "Can't go before January 1, " << INT_MIN;
85  }
86  #endif
```

With my compiler, we're on a first-name basis with the members of class `date` from the double colon in line 20 to the closing curly brace in line 34. That's why line 21 doesn't need to mention the last name of `bad_month` and `bad_month_and_day`. With other compilers, line 21 must say `date::bad_month` and `date::bad_month_and_day`.

If the prefix `operator++` throws the `overflow` exception in line 46, it first restores the `date` to its original value in lines 43–45. We don't want to leave the `date` in an inconsistent, half-incremented

state. Lines 44−45 may be combined to

```
87              day = date_length[month = december];
```

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/except1/date.C

```
 1 #include <climits>
 2 #include "date.h"
 3
 4 const int date::length[] = {
 5      0,   //dummy
 6     31,   //january
 7     29,   //february
 8     31,   //march
 9     30,   //april
10     31,   //may
11     30,   //june
12     31,   //july
13     31,   //august
14     30,   //september
15     31,   //october
16     30,   //november
17     31    //december
18 };
19
20 date::date(int initial_month, int initial_day, int initial_year)
21     throw (bad_month, bad_month_and_day)
22 {
23     year = initial_year;
24
25     if (initial_month < january || initial_month > december) {
26         throw bad_month(initial_month);
27     }
28     month = initial_month;
29
30     if (initial_day < 1 || initial_day > length[month]) {
31         throw bad_month_and_day(month, initial_day);
32     }
33     day = initial_day;
34 }
35
36 date& date::operator++() throw (overflow)   //prefix
37 {
38     if (++day > length[month]) {
39         day = 1;
40         if (++month > december) {
41             month = january;
42             if (year >= INT_MAX) {
43                 //Undo the ++'s in lines 38 and 40.
44                 month = december;
45                 day = length[december];
46                 throw overflow();
47             }
48             ++year;
```

```
49          }
50      }
51
52      return *this;
53  }
54
55  date& date::operator--() throw (underflow)   //prefix
56  {
57      if (--day < 1) {
58          if (--month < january) {
59              month = december;
60              if (year <= INT_MIN) {
61                  //Undo the --'s in line 57 and 58.
62                  month = january;
63                  day = 1;
64                  throw underflow();
65              }
66              --year;
67          }
68          day = length[month];
69      }
70
71      return *this;
72  }
```

Line 12 of `main.C` constructs the last possible `date`, causing the `operator++` in line 13 to throw a `date::overflow` exception.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/except1/main.C`

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  #include <climits>
 4  #include "date.h"
 5  using namespace std;
 6
 7  int main()
 8  {
 9      int status = EXIT_FAILURE;
10
11      try {
12          date d(date::december, 31, INT_MAX);
13          cout << ++d << "\n";   //cout << d.operator++() << "\n";
14          status = EXIT_SUCCESS;
15      }
16
17      catch (const date::bad_month& bm) {
18          cerr << bm << "\n";    //operator<<(cerr, bm) <<  "\n";
19      }
20
21      catch (const date::bad_month_and_day& bd) {
22          cerr << bd << "\n";
23      }
24
25      catch (const date::overflow& ov) {
```

```
26          cerr << ov << "\n";
27      }
28
29      catch (const date::underflow& un) {
30          cerr << un << "\n";
31      }
32
33      catch (...) {
34          cerr << "Caught unexpected exception.\n";
35      }
36
37      return status;
38  }
```

```
can't go beyond December 31, 2147483647
```

6.6 Hierarchies of Exceptions

We needed five separate handlers in lines 17–35 of the above `main.C`. But we will now catch `overflow` and `underflow` with a single handler by publicly deriving them from a common base class, `flow`. Similarly, we will publicly derive class `bad_month_and_day` from class `bad_month`. The inheritance must be public so we can take advantage of it when we catch the exceptions in `main`.

All five of the above classes will still have the last name `date` because they are declared within the curly braces in lines 6 and 96 of `date.h`.

Each class derived from `flow` will have to be printed differently, so I'd like `flow`'s `operator<<` in lines 58–61 to be virtual. But only a member function, not a friend, can be virtual. As on pp. 496–497, our workaround will be to have the `operator<<` call the virtual member function `print` in line 54 to do all the work. The classes derived from `flow` will have to override this `print` because it is a pure virtual function, but they will not override `operator<<`. Finally, a class with a virtual function must also have a virtual destructor (line 56).

The `date::bad_month::print` in line 36 has to be public, since it is called by a function that is neither a member nor a friend of class `date::bad_month` (line 43). But the corresponding `date::flow::print` in line 54 can be private, since it is never called by a function that is neither a member nor a friend of class `date::flow`. In fact, it is never called at all.

The constructor in line 76 can throw a `bad_month_and_day` as well as a `bad_month`. But `bad_month_and_day` is derived from `bad_month`, so it doesn't need to be mentioned in the exception specification.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/except2/date.h`

```
1 #ifndef DATEH
2 #define DATEH
3 #include <iostream>
4 using namespace std;
5
6 class date {
```

```
 7     static const int length[];
 8
 9     int year;    //must construct data members in this order
10     int month;   //date::january to date::december inclusive
11     int day;     //1 to length[month] inclusive
12
13 public:
14     enum month_t {   //indices into the length array
15         january = 1,
16         february,
17         march,
18         april,
19         may,
20         june,
21         july,
22         august,
23         september,
24         october,
25         november,
26         december
27     };
28
29     //Exceptions thrown by the constructor of class date:
30
31     class bad_month {
32         const int month;
33     public:
34         bad_month(int initial_month): month(initial_month) {}
35         virtual ~bad_month() {}
36         virtual void print(ostream& ost) const {ost << "bad month " << month;}
37     };
38
39     class bad_month_and_day: public bad_month {
40         const int day;
41
42         void print(ostream& ost) const {
43             bad_month::print(ost);
44             ost << ", day " << day;
45         }
46     public:
47         bad_month_and_day(int initial_month, int initial_day)
48             : bad_month(initial_month), day(initial_day) {}
49     };
50
51     //Exceptions thrown by the 'crement operators functions of class date:
52
53     class flow {
54         virtual void print(ostream&) const = 0;
55     public:
56         virtual ~flow() {}
57
58         friend ostream& operator<<(ostream& ost, const flow& f) {
59             f.print(ost);
60             return ost;
```

```
61          }
62      };
63
64      class overflow: public flow {
65          void print(ostream& ost) const {
66              ost << "can't go beyond December 31, " << INT_MAX;
67          }
68      };
69
70      class underflow: public flow {
71          void print(ostream& ost) const {
72              ost << "can't go before January 1, " << INT_MIN;
73          }
74      };
75
76      date(int initial_month, int initial_day, int initial_year) throw (bad_month);
77
78      date& operator++() throw (overflow);
79      date& operator--() throw (underflow);
80
81      const date operator++(int) throw (overflow) {
82          const date old = *this;
83          ++*this;
84          return old;
85      }
86
87      const date operator--(int) throw (underflow) {
88          const date old = *this;
89          --*this;
90          return old;
91      }
92
93      friend ostream& operator<<(ostream& ost, const date& d) {
94          return ost << d.month << "/" << d.day << "/" << d.year;
95      }
96  };
97
98  inline ostream& operator<<(ostream& ost, const date::bad_month& bm) {
99      bm.print(ost);
100     return ost;
101 }
102 #endif
```

On some platforms, the `bad_month` in line 20 must be written as `date::bad_month`. Ditto for lines 35 and 54.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/except2/date.C`

```
 1 #include <climits>
 2 #include "date.h"
 3
 4 const int date::length[] = {
 5       0,   //dummy
 6      31,   //january
 7      29,   //february
```

```
 8     31,   //march
 9     30,   //april
10     31,   //may
11     30,   //june
12     31,   //july
13     31,   //august
14     30,   //september
15     31,   //october
16     30,   //november
17     31    //december
18 };
19
20 date::date(int initial_month, int initial_day, int initial_year) throw (bad_month)
21 {
22     year = initial_year;
23
24     if (initial_month < january || initial_month > december) {
25         throw bad_month(initial_month);
26     }
27     month = initial_month;
28
29     if (initial_day < 1 || initial_day > length[month]) {
30         throw bad_month_and_day(month, initial_day);
31     }
32     day = initial_day;
33 }
34
35 date& date::operator++() throw (overflow)   //prefix
36 {
37     if (++day > length[month]) {
38         day = 1;
39         if (++month > december) {
40             month = january;
41             if (year >= INT_MAX) {
42                 //Undo the ++'s in lines 37 and 39.
43                 month = december;
44                 day = length[month];
45                 throw overflow();
46             }
47             ++year;
48         }
49     }
50
51     return *this;
52 }
53
54 date& date::operator--() throw (underflow)   //prefix
55 {
56     if (--day < 1) {
57         if (--month < january) {
58             month = december;
59             if (year <= INT_MIN) {
60                 //Undo the --'s in lines 56 and 57.
61                 month = january;
```

```
62                  day = 1;
63                  throw underflow();
64              }
65              --year;
66          }
67          day = length[month];
68      }
69
70      return *this;
71  }
```

Line 16 of `main.C` will catch `date::bad_month`, or any other exception of a class that is publicly derived from class `date::bad_month`. Similarly, line 20 will catch any `date::flow`.

Lines 16 and 20 must catch by reference to avoid slicing off the additional members introduced in the derived classes. For slicing, see pp. 490−491. For example, if the argument in line 16 was declared as a plain old `date::bad_month`, not as a reference thereto, the `operator<<` in line 17 would always call `date::bad_month::print`. It would never call never call `date::bad_month_and_day::print`, since the `virtual` mechanism is used only when the object is specified by a pointer or reference.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/except2/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <climits>
 4 #include "date.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     int status = EXIT_FAILURE;
10
11     try {
12         date d(date::april, 31, 2014);
13         status = EXIT_SUCCESS;
14     }
15
16     catch (const date::bad_month& b) {
17         cerr << b << "\n";
18     }
19
20     catch (const date::flow& f) {
21         cerr << f << "\n";
22     }
23
24     catch (...) {
25         cerr << "Caught unexpected exception.\n";
26     }
27
28     return status;
29 }
```

```
bad month 4, day 31
```

As we saw earlier, the handlers are tried in the order in which they are listed. The handler for truncation comes before the one for overflow since truncation happens more often. If we inserted the following at the above line 15 of `main.C`, line 16 would no longer have a chance to catch `date::bad_month_and_day`.

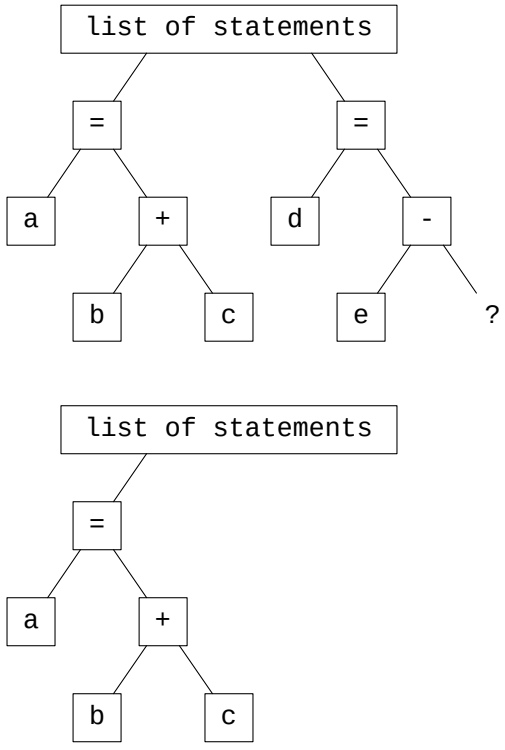
```
30      catch (const date::bad_month_and_day& b) {
31          cerr << b << "\n";
32      }
```

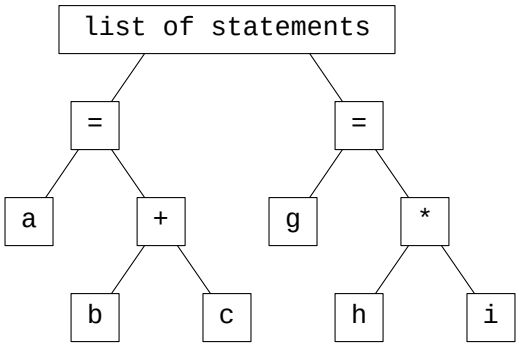
6.7 Backtracking

Error recovery sometimes requires backtracking or dismantling.

A compiler builds a tree whenever it sees an expression. Let's imagine a simple language in which each expression is on a separate line. When the compiler encounters the syntax error in line 2, it removes the partially constructed second tree before it begins to build the third one.

```
1 a = b + c
2 d = e -
3 g = h * i
```

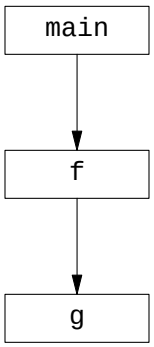




Throwing an exception destructs the automatic objects.

When `exit` is called, only the statically allocated objects of a program are destructed (pp. 184−185). When exceptions are thrown, the automatically allocated objects can be destructed as well. The dynamic objects will not be destructed, however, unless we provide for them as described below.

To illustrate, `main` will call `f`, `f` will call `g`, and on the way down we will construct objects of all three storage classes. The static and automatic objects are named `static`- and `auto`-. The pointer `p` in line 39 is automatic although the object it points to is dynamic. This object has no name.



When an exception escapes from a block, we destruct all the automatic objects constructed in the block. When the block is a function, we also destruct all the automatics in the function that called it, and the automatics in the function that called that one, all the way up to, but not including, the function that contains the `try` statement. This mass extinction of automatic objects is called *unwinding the stack.*

If the exception is never caught, the unwinding extends to every automatic object. The uncaught exception will then terminate the program, destructing the static objects as well.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "obj.h"
 4 using namespace std;
 5
 6 class bad {};
 7
 8 void f() throw (bad);
 9 void g() throw (bad);
10
11 obj static1 = 10;
```

```
12 obj static2 = 20;
13
14 int main()
15 {
16     int status = EXIT_FAILURE;
17     obj auto3 = 30;
18     obj auto4 = 40;
19
20     try {
21         obj auto5 = 50;
22         obj auto6 = 60;
23         f();
24         status = EXIT_SUCCESS;
25     }
26
27     catch (const bad&) {
28         cout << "caught bad in main\n";
29     }
30
31     return status;
32 }
33
34 void f() throw (bad)
35 {
36     obj auto7 = 70;
37     obj auto8 = 80;
38     static obj static9 = 90;
39     obj *const p = new obj(100);
40
41     g();
42     delete p;   //Line 50 prevents this from being executed.
43 }
44
45 void g() throw (bad)
46 {
47     obj auto11 = 110;
48     obj auto12 = 120;
49
50     throw bad();
51 }
```

The stack is unwound as the exception flies from the above line 50 up to 27. The output produced by the unwinding is displayed between the pair of horizontal lines. The static and dynamic objects in the above lines 38−39 are not destructed as part of the unwinding. The static object is eventually destructed when line 31 returns from `main`. The dynamic object is never destructed.

The dynamic object could be destructed after the unwinding by making `p` global and inserting another `delete p;` at line 27½. Alternatively, it could be destructed during the unwinding, by means of the `auto_ptr` on pp. 611−612 or the `throw` without an operand on p. 621.

The error message in the above line 28 would normally be written to `cerr`, not `cout`. But this would confuse the sequence of events. `cerr` is unbuffered, which would allow the message to elbow its way ahead of the other output.

```
construct 10          Lines 11−12 construct two global static objects.
construct 20
construct 30          Lines 17−18 construct two automatic objects local to main.
construct 40
construct 50          Lines 21−22 construct two automatic objects local to try block in main.
construct 60
construct 70          Lines 36−37 construct two automatic objects local to f.
construct 80
construct 90          Line 38 constructs one static object local to f.
construct 100         Line 39 constructs one dynamic object; never destructed.
construct 110         Lines 47−48 construct two automatic objects local to g.
construct 120
destruct 120          Line 50 unwinds the stack.
destruct 110
destruct 80
destruct 70
destruct 60
destruct 50
caught bad in main
destruct 40           Line 31 destructs the remaining automatic objects local to main.
destruct 30
destruct 90           Line 31 destructs the three static objects.
destruct 20
destruct 10
```

If we change the `throw bad()` in the above line 50 to a call to `exit`, the stack will no longer be unwound. The output below the line is produced after the call to `exit`.

```
construct 10
construct 20
construct 30
construct 40
construct 50
construct 60
construct 70
construct 80
construct 90
construct 100
construct 110
construct 120
destruct 90           Only the static objects are still destructed.
destruct 20
destruct 10
```

If we change the `throw bad()` in the above line 50 to `terminate()` or `abort()`, no object will be destructed, not even the static ones.

```
construct 10
construct 20
construct 30
construct 40
construct 50
construct 60
construct 70
construct 80
construct 90
construct 100
construct 110
construct 120
```

auto_ptr

The above program left its dynamically allocated object undestructed and undeleted. An elegant way to take care of this is with an `auto_ptr` object.

The dynamic object, having no name, is referenced by means of a pointer. Here is line 39 of the above `unwind.C` on p. 609.

```
1     //The "const" keeps p pointing to the same object.
2     obj *const p = new obj(100);
```

An `auto_ptr` object contains a pointer to a dynamic object; we say that the `auto_ptr` points to the latter. Like the `vector` in the standard library, class `auto_ptr` is a ‘‘template’’ (Chapter 7). The name of the data type to which the `auto_ptr` points is plugged into the <angle brackets>; the address of the dynamic object to which the `auto_ptr` points is passed as an argument to the constructor.

```
3     //The "const" keeps p pointing to the same object.
4     const auto_ptr<obj> p(new obj(100));
```

This constructor is `explicit`, so its argument must always be in parentheses. The above line 4 could not be written

```
5     const auto_ptr<obj> p = new obj(100);       //won't compile
```

When an `auto_ptr` is destructed, it applies the `delete` operator to the pointer it contains. This operator calls the destructor for the dynamic object. For example, we can change the function `f` of `unwind.C` to the following, removing the hapless `delete p;` in the above line 42. Include the header file `<memory>` for `auto_ptr`.

```
 6 void f() throw (bad)
 7 {
 8     obj auto7 = 70;
 9     obj auto8 = 80;
10     static obj static9 = 90;
11     const auto_ptr<obj> p(new obj(100));
12
13     g();
14 }
```

The objects `auto7`, `auto8`, and `p` are automatically allocated. Their destructors will always be called when we leave the {curly braces} of `f`, whether or not `g` throws an exception. The destructor for `p` will `delete` the object to which `p` points, fixing the memory leak.

Thanks to the magic of operator overloading, an `auto_ptr` can be dereferenced with the same syntax as a plain old pointer. For example, the `*p` in line 19 is the dynamic object to which `p` points, and the `p->` in line 21 lets us access a member of the dynamic object. But there is one difference. The unadorned

p in line 24 is not the address of the dynamic object. To get the address, we have to call the get in line 25. (The operator-> in line 21 actually returns the same value as get.)

```
15 void f()
16 {
17     const auto_ptr<obj> p(new obj(100));
18
19     cout << *p;                    //operator<<(cout, p.operator*());
20     cout << "\n";
21     p->print();                    //p.operator->()->print();
22     cout << "\n";
23
24     //cout << "Address of dynamic object: " << p << "\n";  //won't compile
25     cout << "Address of dynamic object: " << p.get() << "\n";
26 }   //obj is deleted here
```

Even in the absence of exceptions, an auto_ptr is a general mechanism for ensuring that a dynamically allocated variable does not outlive the pointer that points to it. For example, the following block deleted the dynamically allocated nodes in a linked list; see lines 52−57 of linked.C on p. 399. The const at the start of line 29 gives doomed read-only access to the dynamic object.

```
27     for (const node *p = first; p;) {
28         cout << *p << "\n";
29         const node *const doomed = p;
30         p = p->next;
31         delete doomed;
32     }
```

We can let an auto_ptr do the delete for us. The const in the angle brackets in line 35 gives doomed read-only access to the dynamically allocated object. (Resist the temptation to insert line 36 into the end of line 33.)

```
33     for (const node *p = first; p;) {
34         cout << *p << "\n";
35         const auto_ptr<const node> doomed(p);
36         p = p->next;
37     }
```

A similar auto_ptr can be used in the block in lines 23−27 of the destructor for game in p. 542.

Two warnings about auto_ptr.

(1) The pointer in the auto_ptr must always point to a scalar, not to an array. The destructor for auto_ptr always applies the delete operator, not the delete[] operator, to this pointer. And, of course, the pointed-to variable must be dynamically allocated.

(2) A dynamically allocated variable can be delete'd only once. This means we must never have two auto_ptr's pointing to the same object. To prevent this, the copy constructor and operator= for class auto_ptr set the internal pointer of the right-hand auto_ptr to zero. We say that these functions *transfer ownership* of the dynamic variable from the right auto_ptr to left one.

```
38     auto_ptr<obj> p1(new obj(10));
39
40     //if p1 were destructed at this point,
41     //p1's destructor would delete the obj.
42
43     auto_ptr<obj> p2 = p1;         //copy constructor
44
45     //If p1 and p2 were destructed at this point,
46     //p2's destructor would delete the obj
```

```
47      //and p1's destructor would do nothing.
48
49      p1 = p2;                        //p1.operator=p2();
50
51      //If p1 and p2 were destructed at this point,
52      //the p2's destructor would do nothing
53      //and p1's destructor would delete the obj.
```

The "algorithms" in Chapter 8 assume that an element of a container can be copied without damage to the original. But an `auto_ptr` has a copy constructor and `operator=` that drain their argument. Do not attempt to store an `auto_ptr` into a `vector` or `list`.

Finally, an `auto_ptr` can also yield its responsibility to a plain old pointer.

```
54      auto_ptr<obj> p(new obj(10));
55      obj *plain = p.release();
56
57      //If p was destructed at this point, p's destructor would do nothing.
58      delete plain;    //must remember to delete the dynamic object by hand.
```

An `auto_ptr` can also be reset:

```
59      auto_ptr<obj> p(new obj(10));
60
61      p.reset(new obj(10));           //p deletes 1st obj, takes ownership of 2nd obj
62      p.reset();                      //p deletes 2nd obj
63
64      //If p was destructed at this point, its destructor would do nothing.
```

Destruction is a privilege.

Does every automatic object get destructed when an exception leaves a block? Does every static object get destructed when an exceoption is thrown but not caught? It depends what we mean by "object".

A *completely constructed* object is one from whose constructor we have returned, either by a `return` statement or by reaching the closing curly brace `}` at the end of the constructor's body. Only a completely constructed object is eligible for the privilege of destruction. An example is the data member `e1` in the program below.

If an exception escapes from an object's constructor, the object will never be completely constructed. It is therefore ineligible for the privilege of destruction. Such is the case of the stillborn `e2`, even though part of its constructor has been executed. In fact, such is also the case of the surrounding object `b`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class even {
 6     int i;
 7 public:
 8     even(int initial_i) throw (int): i(initial_i) {
 9         cout << "constructor for even " << i << " started ";
10         if (i % 2 != 0) {
11             throw i;
12         }
13         cout << "and finished.\n";
```

```
14      }
15
16      ~even() {cout << "\ndestruct even " << i << "\n";}
17  };
18
19  class big {
20      even e1;
21      even e2;
22      even e3;
23  public:
24      big(int initial_e1, int initial_e2, int initial_e3) throw (int)
25          : e1(initial_e1), e2(initial_e2), e3(initial_e3) {
26          cout << "construct big\n";
27      }
28
29      ~big() {cout << "destruct big\n";}
30  };
31
32  int main()
33  {
34      try {
35          big b(10, 21, 30);
36      }
37
38      catch (int i) {
39          cerr << "main caught the integer " << i << ".\n";
40          return EXIT_FAILURE;
41      }
42
43      return EXIT_SUCCESS;
44  }
```

```
constructor for even 10 started and finished.
constructor for even 21 started
destruct even 10
main caught the integer 21.
```

We can't have two exceptions in the air simultaneously

An exception cannot escape from a function while another exception remains uncaught. For example, an exception cannot escape from a destructor that was triggered by an exception escaping from a block. If this happens, the function `terminate` will be called. To prove it, I wrote a `terminate` function that prints a message.

Line 22 throws an exception which destructs the `pit` in line 21 as it escapes from the block in lines 20−23. But before this exception is caught in line 25, the destructor of `pit` tries to throws another exception. This calls the `terminate` function.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/air1.C`

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <exception>   //for set_terminate
4  using namespace std;
5
```

```
 6 class pitcher {
 7 public:
 8     ~pitcher() throw (int) {
 9         cout << "~pitcher about to throw 20.\n";
10         throw 20;
11     }
12 };
13
14 void my_terminate();
15
16 int main()
17 {
18     set_terminate(my_terminate);
19
20     try {
21         pitcher pit;
22         throw 10;
23     }
24
25     catch (int i) {
26         cout << "main caught int " << i << ".\n";
27     }
28
29     return EXIT_SUCCESS;
30 }
31
32 void my_terminate()
33 {
34     cerr << "my_terminate has been called.\n";
35     exit(EXIT_FAILURE);
36 }
```

```
~pitcher about to throw 20.
my_terminate has been called.
```

A destructor *can* throw an exception, but we have to be careful. If another exception has been thrown but not yet caught, the destructor must catch any exception that it throws. The exception thrown by the destructor would terminate the program if it escaped from the destructor.

Line 9 shows how a destructor can tell if there is an exception that has been thrown but not yet caught. As the exception thrown in line 30 escapes from the block in lines 28−31, we call the destructor for the object `pit` in line 29. In this call, the `if` in line 9 is true, so the destructor catches the exception it throws in line 11. To verify that the program has

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/air2.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <exception>   //for uncaught_exception and set_terminate
4 using namespace std;
5
6 class pitcher {
7 public:
8     ~pitcher() throw (int) {
9         if (uncaught_exception()) {
```

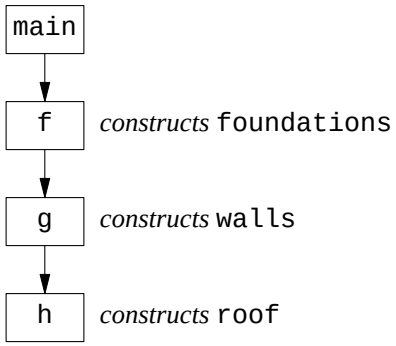
```
10              try {
11                  throw 20; //This exception will not escape from the destructor.
12              }
13              catch (int i) {
14                  cout << "~pitcher caught int " << i << ".\n";
15              }
16          } else {
17              throw 30;     //This exception will escape from the destructor.
18          }
19      }
20  };
21
22  void my_terminate();
23
24  int main()
25  {
26      set_terminate(my_terminate);
27
28      try {
29          pitcher pit;
30          throw 10;
31      }
32
33      catch (int i) {
34          cout << "main caught int " << i << ".\n";
35      }
36
37      return EXIT_SUCCESS;
38  }
39
40  void my_terminate()
41  {
42      cerr << "my_terminate has been called.\n";
43      exit(EXIT_FAILURE);
44  }
```

```
~pitcher caught int 20.
main caught int 10.
```

Recall that there is one other thing a destructor must not do: call the `exit` function. See p. 184.

6.8 Catch Exceptions at Two or More Levels

A trivial error requires the dismantling of only a few objects; a more severe one may require the dismantling of many. We can arrange this by declaring the objects in layers. The foundations will be laid in the function `f`, the walls are raised in `g`, and the roof is put on in `h`.



—On the Web at
http://i5.nyu.edu/~mm64/book/src/exception/level1.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <ctime>
 4 using namespace std;
 5
 6 class foundation {
 7 public:
 8     ~foundation() {cerr << "destruct the foundation\n";}
 9 };
10
11 class walls {
12 public:
13     ~walls() {cerr << "destruct the walls\n";}
14 };
15
16 class roof {
17 public:
18     ~roof() {cerr << "destruct the roof\n";}
19 };
20
21 //Data types of the exceptions:
22 class trivial {};  //handling this requires the destruction of only the roof
23 class medium {};   //requires destruction of roof and walls
24 class severe {};   //requires destruction of roof, walls, and foundations
25
26 void f() throw (severe);
27 void g() throw (severe, medium);
28 void h() throw (severe, medium, trivial);
29
30 int main()
31 {
32     srand(static_cast<unsigned>(time(0)));
33
34     try {
35         f();
36     }
37
38     catch (severe) {
39         cerr << "main caught a severe exception.\n";
```

```
40          return EXIT_FAILURE;
41      }
42
43      return EXIT_SUCCESS;
44  }
45
46  void f() throw (severe)
47  {
48      foundation found;
49
50      try {
51          g();
52      }
53
54      catch (medium) {
55          cerr << "f caught a medium exception.\n";
56      }
57  }
58
59  void g() throw (severe, medium)
60  {
61      walls w;
62
63      try {
64          h();
65      }
66
67      catch (trivial) {
68          cerr << "g caught a trivial exception.\n";
69      }
70  }
71
72  void h() throw (severe, medium, trivial)
73  {
74      roof r;
75
76      switch (rand() % 3) {
77      case 0:
78          throw trivial();
79          break; //this statement currently unnecessary
80
81      case 1:
82          throw medium();
83          break;
84
85      case 2:
86          throw severe();
87          break;
88
89      default:
90          break;
91      }
92  }
```

To handle a `trivial` exception, all we have to dismantle is the `roof`. The `walls` and `foundation` are destructed later, when we return from `g` and `f`.

```
destruct the roof
g caught a trivial exception.
destruct the walls
destruct the foundation
```

But to handle a `medium` exception, we have to dismantle the `roof` and the `walls`:

```
destruct the roof
destruct the walls
f caught a medium exception.
destruct the foundation
```

To handle a `severe` exception, we must dismantle the `roof`, `walls`, and `foundation`:

```
destruct the roof
destruct the walls
destruct the foundation
main caught a severe exception.
```

6.8.1 Catch and Re-throw

The operator `throw` can be written with no operand inside of an exception handler. It will re-throw the exception that the handler caught. See lines 71−76.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/exception/level2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <ctime>
 4 using namespace std;
 5
 6 class foundation {
 7 public:
 8     ~foundation() {cerr << "destruct the foundation\n";}
 9 };
10
11 class walls {
12 public:
13     ~walls() {cerr << "destruct the walls\n";}
14 };
15
16 class roof {
17 public:
18     ~roof() {cerr << "destruct the roof\n";}
19 };
20
21 //Data types of the exceptions:
22 class trivial {};  //handling this requires the destruction of only the roof
23 class medium {};   //requires destruction of roof and walls
24 class severe {};   //requires destruction of roof, walls, and foundations
25
```

```
26 void f() throw (severe);
27 void g() throw (severe, medium);
28 void h() throw (severe, medium, trivial);
29
30 int main()
31 {
32     srand(static_cast<unsigned>(time(0)));
33
34     try {
35         f();
36     }
37
38     catch (severe) {
39         cerr << "main caught a severe exception.\n";
40         return EXIT_FAILURE;
41     }
42
43     return EXIT_SUCCESS;
44 }
45
46 void f() throw (severe)
47 {
48     foundation found;
49
50     try {
51         g();
52     }
53
54     catch (medium) {
55         cerr << "f caught a medium exception.\n";
56     }
57 }
58
59 void g() throw (severe, medium)
60 {
61     walls w;
62
63     try {
64         h();
65     }
66
67     catch (trivial) {
68         cerr << "g caught a trivial exception.\n";
69     }
70
71     catch (...) {
72         cerr << "g caught a non-trivial exception.\n"
73             "Don't worry about having scratched the walls when destructing\n"
74             "the roof--we're about to destruct the walls too.\n";
75         throw;
76     }
77 }
78
79 void h() throw (severe, medium, trivial)
```

```
80 {
81     roof r;
82
83     switch (rand() % 3) {
84     case 0:
85         throw trivial();
86         break;
87
88     case 1:
89         throw medium();
90         break;
91
92     case 2:
93         throw severe();
94         break;
95
96     default:
97         break;
98     }
99 }
```

If the above line 89 throws a `medium` exception,

```
destruct the roof
g caught a non-trivial exception.
Don't worry about having scratched the walls when destructing
the roof--we're about to destruct the walls too.
destruct the walls
f caught a medium exception.
destruct the foundation
```

▼ Homework 6.8.1a: catch and rethrow

The program `unwind.C` on p. 609 fails to destruct the dynamically allocated object in line 39. Fix it by changing lines 39−42 to

```
 1     obj *const p = new obj(100);
 2
 3     try {
 4         g();
 5     }
 6
 7     catch (...) {
 8         delete p;
 9         throw;
10     }
11
12     delete p;
```

It was a lot simpler with the `auto_ptr` on pp. 611−612, wasn't it? We didn't have to write the `delete p;` twice.

▲

6.9 Exceptions in the C++ Standard Library

Many functions in the C++ Standard Library will throw an exception if something goes wrong. All of these exceptions are derived from the base class `exception` on p. 628, declared in the header file `<exception>`. There are often two ways of doing a given job, one that throws an exception and one that returns an error code. In addition to the examples in §6.9, there will be another on pp. 1014−1015.

6.9.1 Containers: `vector`, `string`, and `bitset`

> [C. A. R. Hoare] points out quite correctly that the current practice of compiling subscript range checks into the machine code while a program is being tested, then suppressing the checks during production runs, is like a sailor who wears a life preserver while training on land but leaves it behind when he sails! On the other hand, the sailor isn't so foolish if life vests are extremely expensive, and if he is such an excellent swimmer that the chance of needing one is quite small compared with the other risks he is taking.
>
> −Donald Knuth, *Structured Programming with go to Statements*

Any value read from input is suspicious. The variable `i` in line 14 will be a subscript for a vector. Since line 15 reads its value from input, line 18 calls the member function `at` to see if it is in range. If not, `at` throws the `out_of_range` exception declared in the header file `<stdexcept>`. (The `in_range` member function of class `terminal` was named after this class; see line 31 of `terminal.h` on p. 160.)

Every class derived from class `exception` has the `what` function in line 22, returning a string. We print it to see how helpful it is.

If the `at` in line 18 throws a different exception derived from class `exception`, it will be caught at line 26. If the `at` throws an exception not derived from this class, it will be caught at line 31.

We are confident (i.e., willing to gamble) that the loop in line 36 will keep the subscript within the legal range for the vector. Line 37 therefore calls the member function `operator[]` instead of `at`. It does the same job as `at` but without the error checking. `operator[]` is faster than `at`, but could blow up if the subscript is out of range. So could the loop in lines 40−42.

Class `string` has a member function `at` that does subscript checking (`out_of_range`), and an `operator[]` that does not. Class `bitset` has member functions `flip`, `set`, `reset`, and `test` that do subscript checking (`out_of_range`), and an `operator[]` that does not. The member function `to_ulong` of class `bitset` will throw an `overflow_error` if the `bitset`'s value cannot fit into a `long unsigned`.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <stdexcept>   //for out_of_range
 5 using namespace std;
 6
 7 int main()
 8 {
 9     int a[] = {10, 20, 30};
10     const size_t n = sizeof a / sizeof a[0];
11     vector<int> v(a, a + n);
12
13     cout << "Please input a subscript: ";
14     vector<int>::size_type i;   //data type for a vector subscript
15     cin >> i;
16
```

```
17      try {
18          cout << "Element number " << i << " is " << v.at(i) << ".\n";
19      }
20
21      catch (const out_of_range& out) {
22          cerr << "caught out_of_range exception " << out.what() << "\n";
23          return EXIT_FAILURE;
24      }
25
26      catch (const exception& e) {
27          cerr << "caught another exception " << e.what() << "\n";
28          return EXIT_FAILURE;
29      }
30
31      catch (...) {
32          cerr << "caught unexpected exception\n";
33          return EXIT_FAILURE;
34      }
35
36      for (vector<int>::size_type i = 0; i < v.size(); ++i) {
37          cout << v[i] << "\n";   //cout << v.operator[](i) << "\n";
38      }
39
40      for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
41          cout << *it << "\n";    //cout << it.operator*() << "\n";
42      }
43
44      return EXIT_SUCCESS;
45  }
```

```
Please input a subscript: 4
Element number 4 is caught out_of_range exception vector::_M_range_check
```

6.9.2 Class `istream`

Before they invented exceptions, the conscientious programmer had to write a complicated `if` statement after every attempt at input. See lines 11−32 of `why.C` on pp. 330−331.

But now, the `>>` in the following line 13 can throw an exception if anything goes wrong. To ask it to, simply call the input stream's member function `exceptions` in line 7.

On my platform, the argument in line 7 has the value 7. But think of it as ''yes, yes, yes'' as in *When Harry Met Sally.* Each bit position of the argument represents a condition for which we want `cin` to throw an exception. The enumerations of class `ios_base` provide a convenient name for each position, and we combine them with ''bitwise or''.

```
  ios_base::badbit    00000100
  ios_base::failbit   00000010
| ios_base::eofbit    00000001
                      00000111
```

The `int i` in line 9 must be declared outside the block in lines 12−14 if it is to be mentioned in line 35 outside the block.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     cin.exceptions(ios_base::badbit | ios_base::failbit | ios_base::eofbit);
 8
 9     int i;   //uninitialized variable
10     cout << "Please input an integer: ";
11
12     try {
13         cin >> i;
14     }
15
16     catch (const ios_base::failure& fail) {
17         cerr << "caught ios_base::failure exception " << fail.what()
18             << "\n";
19
20         if (cin.eof()) {
21             cerr << "encountered end of input\n";
22         } else if (cin.bad()) {
23             cerr << "couldn't input characters from outside world\n";
24         } else if (cin.fail()) {
25             cerr << "the first non-whitespace characters read\n"
26                 "from input were not one or more consecutive\n"
27                 "digits optionally preceded by a minus sign\n";
28         } else {
29             cerr << "unknown error\n";
30         }
31
32         return EXIT_FAILURE;
33     }
34
35     cout << "The number was " << i << ".\n";
36     return EXIT_SUCCESS;
37 }
```

```
Please input an integer: 10
The number was 10.
```

```
Please input an integer: hello
caught ios_base::failure exception basic_ios::clear
the first non-whitespace characters read
from input were not one or more consecutive
digits optionally preceded by a minus sign
```

```
Please input an integer: control-d                    (the end-of-file keystroke)
caught ios_base::failure exception basic_ios::clear   same retval from what
encountered end of input
```

The above line 7 will also cause exceptions to be thrown by an `operator>>` or `operator<<` that we wrote ourselves. In the `date.C` on p. 338, for example, a failure of the `>>` operator in lines 10 or 16

will throw an exception. The call to `setstate` in line 21 will also throw an exception.

6.9.3 new and delete

The `operator new` functions in the standard libary will throw an exception of type `bad_alloc` if they cannot allocate the requested memory, and if no `new_handler` has been established. Here are their declarations, together with the matching `delete`'s.

```
1 void *operator new(size_t n) throw (bad_alloc);
2 void operator delete(void *p) throw ();
3
4 void *operator new[](size_t n) throw (bad_alloc);
5 void operator delete[](void *p) throw ();
```

The standard library has an empty class `nothrow_t`, and one object named `nothrow` of that class.

```
6 //Excerpt from <new>
7
8 struct nothrow_t {};
9 extern const nothrow_t nothrow;
```

The only purpose of `nothrow` is to let us have an alternative series of functions that do not throw exceptions. The `operator new`'s in this series simply return zero if they cannot allocate the memory. The `operator delete`'s will be discussed below.

```
10 void *operator new(size_t n, const nothrow_t&) throw ();
11 void operator delete(void *p, const nothrow_t&) throw ();
12
13 void *operator new[](size_t n, const nothrow_t&) throw ();
14 void operator delete[](void *p, const nothrow_t&) throw ();
```

We now have three ways to respond to an allocation error. Line 33 and 34 call a `new_handler` function; line 25 throws an exception; and line 17 returns zero. You probably want to throw and catch an exception.

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

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <new>   //for nothrow, bad_alloc, set_new_handler
 4 using namespace std;
 5
 6 const char *progname;
 7 void my_new_handler();
 8
 9 int main(int argc, char **argv)
10 {
11     progname = argv[0];
12
13     cout << "How many bytes do you want to allocate? ";
14     size_t n;
15     cin >> n;
16
17     char *const p1 = new(nothrow) char [n];   //return 0 on error
18     if (p1 == 0) {
19         cerr << progname << ": out of store\n";
20         return EXIT_FAILURE;
```

```
21      }
22      delete[] p1;
23
24      try {
25          char *const p2 = new char [n];        //throw exception on error
26          delete[] p2;
27      }
28      catch (const bad_alloc& bad) {
29          cerr << progname << ": out of store: " << bad.what() << "\n";
30          return EXIT_FAILURE;
31      }
32
33      set_new_handler(my_new_handler);
34      char *const p3 = new char [n];            //call my_new_handler on error
35      delete[] p3;
36
37      return EXIT_SUCCESS;
38  }
39
40  void my_new_handler()
41  {
42      cerr << progname << ": out of store\n";
43      exit(EXIT_FAILURE);
44  }
45
```

```
How many bytes do you want to allocate? 4294967296
new: out of store
```

The above line 17 showed how to pass `nothrow` to an `operator new`. How do we pass it to an `operator delete`? We don't—the computer does. Assume that a `new` operator calls an `operator new` or `operator new[]` function, with or without a `nothrow`, followed by a constructor. If the constructor throws an exception, the `new` operator will call the corresponding `operator delete` or `operator delete[]` function, with or without a `nothrow`. This is the only way that the `nothrow` version of `operator delete` or `operator delete[]` can be called.

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

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  #include <new>
 4  using namespace std;
 5
 6  inline void *operator new(size_t n) throw (bad_alloc) {
 7      cout << "operator new without nothrow\n";
 8      if (void *const p = malloc(n)) {
 9          return p;
10      }
11      throw bad_alloc();
12  };
13
14  inline void operator delete(void *p) throw () {
15      cout << "operator delete without nothrow\n";
16      free(p);
```

```
17 }
18
19 inline void *operator new(size_t n, const nothrow_t&) throw () {
20     cout << "operator new with nothrow\n";
21     return malloc(n);
22 }
23
24
25 inline void operator delete(void *p, const nothrow_t&) throw () {
26     cout << "operator delete with nothrow\n";
27     free(p);
28 }
29
30 class obj {
31 public:
32     obj() throw (int) {throw 10;}
33 };
34
35 int main()
36 {
37     try {
38         obj *const p1 = new obj;
39     }
40     catch (int i) {
41         cout << "caught " << i << "\n";
42     }
43     catch (...) {
44         cout << "caught exception other than int\n";
45     }
46
47     try {
48         obj *const p2 = new(nothrow) obj;
49     }
50     catch (int i) {
51         cout << "caught " << i << "\n";
52     }
53     catch (...) {
54         cout << "caught exception other than int\n";
55     }
56
57     return EXIT_SUCCESS;
58 }
```

```
operator new without nothrow
operator delete without nothrow
caught 10
operator new with nothrow
operator delete with nothrow
caught 10
```

▼ **Homework 6.9.3a: let life::operator− catch bad_alloc**

Our `life::operator-` on pp. 441−442 kept calling `vector<life>::push_back` with wild abandon. If `push_back` throws a `bad_alloc` exception, let `operator-` catch it and return `INT_MAX`.

▲

▼ **Homework 6.9.3b:**
Version 3.8 of the Rabbit Game: throw exceptions

At the first sign of trouble, the game writes an error message to `cerr` and then `exit`'s. But calling `exit` means that the terminal will never be properly destructed. Our screen could be left in graphics mode or with a derelict window—a different outcome on each platform.

To ensure that every object is destructed, we will terminate the game by throwing an exception if anything goes wrong. The exception will carry an error message up to `main`. `exit` will no longer be called anywhere, except in the C code in `term.c`. The files that included `<cstdlib>` only for `exit` and `EXIT_FAILURE` will no longer need to do so.

The C++ Standard Library has a class named `exception`, containing the following members (among others).

```
 1 //Excerpt from <exception>
 2
 3 class exception {
 4 public:
 5     exception() throw () {}
 6     exception(const exception& other) throw ();
 7     virtual ~exception() throw ();
 8     exception& operator=(const exception& another) throw ();
 9
10     virtual const char *what() const throw ();
11 };
```

Like the object `cout`, class `exception` has the last name `std` (p. 20). We must therefore either call it `std::exception` or say `using namespace std;`.

The exceptions that we throw will be of the following class `except`, publicly derived from the above so we can catch them together. The base class destructor can throw nothing, so the derived class destructor must be declared to throw no more than nothing. That's the only reason the derived destructor had to be declared.

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

```
 1 #ifndef EXCEPTH
 2 #define EXCEPTH
 3 #include <sstream>    //for ostringstream
 4 #include <exception>  //for exception
 5 #include <string>
 6 using namespace std;
 7
 8 class except: public exception {
 9         const string s;
10 public:
11         ~except() throw () {}
12         except(const string& initial_s) throw () : s(initial_s) {}
13         except(const ostringstream& ost) throw () : s(ost.str()) {}
14         const char *what() const throw () {return s.c_str();}
15 };
```

```
16 #endif
```

Don't bother writing an exception specification for each function in the game. Just construct and throw an `except` if anything goes wrong. Here is an example. We saw the double cast in line 12 in line 15 of `static_cast` on p. 65. `terminal.C` will no longer include `<cstdlib>` since it no longer calls `exit`.

```
 1 //Excerpt from terminal.C.
 2 #include <sstream>      //for ostringstream
 3 #include "terminal.h"
 4 #include "except.h"
 5 using namespace std;
 6
 7 void terminal::put(unsigned x, unsigned y, char c)
 8 {
 9     if (isprint(static_cast<unsigned char>(c)) == 0) {
10         ostringstream ost;
11         ost << "unprintable character "
12             << static_cast<unsigned>(static_cast<unsigned char>(c))
13             << " at location (" << x << ", " << y << ")";
14         throw except(ost);
15     }
16
17     check(x, y);
18     term_put(x, y, c);
19 }
```

▲

▼ Homework 6.9.3c:
Version 3.9 of the Rabbit Game: catch the exceptions

The `wabbit`'s are constructed in `game::game`; the surviving ones are destructed in `game::~game`. But suppose an exception escapes from a `wabbit`'s constructor and from the `game::game` which called it. The `game` object, never having been completely constructed, will be ineligible for the privilege of destruction. The `game`'s destructor will never be called; the surviving `wabbit`'s will never be destructed.

Our solution is simple. Before any exception escapes from the `game`'s constructor, the `game`'s constructor will destruct any `wabbit`'s that have been constructed. The `game`'s destructor will still not be called. But this is now harmless because there is nothing that the `destructor` needs to do.

(1) Put the code that destructs the `wabbit`'s into a separate function. It will be called by the destructor for class `game`, which will now be short enough to be inline.

```
 1 //A new non-static, private member function of class game.
 2
 3 void game::depopulate()
 4 {
 5     //Destruct and deallocate all the wabbit's that exist at this point.
 6     //Use the delete loop that was in game::~game
 7     //(lines 23-27 of game.C on p. 542).
 8 }
```

(2) The constructor for `game` will catch any exception thrown by a constructor for any `wabbit`. It will delete any wabbit's that have already been constructed up to this point, and then re-throw the exception.

```
 9 game::game(char initial_c)
```

```
10      : term(initial_x)
11  {
12      try {
13          //code to construct all the wabbit's, e.g.,
14          new rabbit(this, /* etc. */);
15      }
16
17      catch (...) {
18          depopulate();
19          throw;
20      }
21  }
```

(3) The re-thrown exceptions, and all others, will be caught in `main`. The file `main.C` will have to include `<new>` for class `bad_alloc` in line 12, and `<exception>` for class `exception` in line 16. We no longer call the function `set_new_handler`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/game6/main.C`

```
 1  int main(int argc, char **argv)
 2  {
 3      int status = EXIT_FAILURE;   //guilty until proven innocent
 4      srand(static_cast<unsigned>(time(0)));
 5
 6      try {
 7          game g;
 8          g.play();
 9          status = EXIT_SUCCESS;
10      }
11
12      catch (const bad_alloc& bad) {
13          cerr << argv[0] << ": new failed: " << bad.what() << "\n";
14      }
15
16      catch (const exception& e) {
17          cerr << argv[0] << ": " << e.what() << "\n";
18      }
19
20      catch (...) {
21          cerr << argv[0] << ": main caught unexpected exception.\n";
22      }
23
24      return status;
25  }
```

▲

▼ Homework 6.9.3d: why did this solution crash the program?

In an early version of the game, the master list contained `auto_ptr`'s.

```
1       list<auto_ptr<wabbit> > master;
```

The `game`'s constructor constructed the `terminal` and the master list and then started to construct the `wabbit`'s. If an exception escaped from the a `wabbit`'s constructor, it was also allowed to escape from the `game`'s constructor. In this case, the `game` was never completely constructed, and hence never destructed.

Although the `game` wqas never completely constructed, the `game`'s `terminal` and `master` data members were. The `wabbit`'s were destructed by the `master`'s destructor.

Or so I hoped. What actually happened? Hint: why must the `game` object outlive its `wabbit`'s? See p. 613 for another reason why a container of `auto_ptr`'s is dangerous.
▲