BJARNE STROUSTRUP

THE CREATOR OF C++

Using
C++11
and
C++14

PROGRAMMING

Principles and Practice Using C++

SECOND EDITION

Programming

Second Edition

```
                for (char ch; cin>>ch; ) {      // throw away non-digits
                        if (isdigit(ch) || ch=='-') {
                                cin.unget();    // put the digit back,
                                                // so that we can read the number
                                return;
                        }
                }
        }
        error("no input");                      // eof or bad: give up
}
```

Given the **skip_to_int()** "utility function," we can write

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (true) {
        if (cin>>n) {           // we got an integer; now check it
                if (1<=n && n<=10) break;
                cout << "Sorry " << n
                        << " is not in the [1:10] range; please try again\n";
        }
        else {
                cout << "Sorry, that was not a number; please try again\n";
                skip_to_int();
        }
}
// if we get here n is in [1:10]
```

This code is better, but it is still too long and too messy to use many times in a program. We'd never get it consistently right, except after (too) much testing.

What operation would we really like to have? One plausible answer is "a function that reads an **int**, any **int**, and another that reads an **int** of a given range":

```
int get_int();                          // read an int from cin
int get_int(int low, int high);         // read an int in [low:high] from cin
```

If we had those, we would at least be able to use them simply and correctly. They are not that hard to write:

```
int get_int()
{
        int n = 0;
        while (true) {
                if (cin >> n) return n;
                cout << "Sorry, that was not a number; please try again\n";
                skip_to_int();
        }
}
```

Basically, get_int() stubbornly keeps reading until it finds some digits that it can interpret as an integer. If we want to get out of get_int(), we must supply an integer or end of file (and end of file will cause get_int() to throw an exception).

Using that general get_int(), we can write the range-checking get_int():

```
int get_int(int low, int high)
{
        cout << "Please enter an integer in the range "
                << low << " to " << high << " (inclusive):\n";

        while (true) {
                int n = get_int();
                if (low<=n && n<=high) return n;
                cout << "Sorry "
                        << n << " is not in the [" << low << ':' << high
                        << "] range; please try again\n";
        }
}
```

This get_int() is as stubborn as the other. It keeps getting ints from the non-range get_int() until the int it gets is in the expected range.

We can now reliably read integers like this:

```
int n = get_int(1,10);
cout << "n: " << n << '\n';

int m = get_int(2,300);
cout << "m: " << m << '\n';
```

Don't forget to catch exceptions somewhere, though, if you want decent error messages for the (probably rare) case when get_int() really couldn't read a number for us.

10.7.2 Separating dialog from function

The get_int() functions still mix up reading with writing messages to the user. That's probably good enough for a simple program, but in a large program we might want to vary the messages written to the user. We might want to call get_int() like this:

```
int strength = get_int(1,10, "enter strength", "Not in range, try again");
cout << "strength: " << strength << '\n';

int altitude = get_int(0,50000,
                    "Please enter altitude in feet",
                    "Not in range, please try again");
cout << "altitude: " << altitude << "f above sea level\n";
```

We could implement that like this:

```
int get_int(int low, int high, const string& greeting, const string& sorry)
{
        cout << greeting << ": [" << low << ':' << high << "]\n";

        while (true) {
                int n = get_int();
                if (low<=n && n<=high) return n;
                cout << sorry << ": [" << low << ':' << high << "]\n";
        }
}
```

It is hard to compose arbitrary messages, so we "stylized" the messages. That's often acceptable, and composing really flexible messages, such as are needed to support many natural languages (e.g., Arabic, Bengali, Chinese, Danish, English, and French), is not a task for a novice.

Note that our solution is still incomplete: the **get_int()** without a range still "blabbers." The deeper point here is that "utility functions" that we use in many parts of a program shouldn't have messages "hardwired" into them. Further, library functions that are meant for use in many programs shouldn't write to the user at all – after all, the library writer may not even know that the program in which the library runs is used on a machine with a human watching. That's one reason that our **error()** function doesn't just write an error message (§5.6.3); in general, we wouldn't know where to write.

10.8 User-defined output operators

Defining the output operator, **<<**, for a given type is typically trivial. The main design problem is that different people might prefer the output to look different, so it is hard to agree on a single format. However, even if no single output format is good enough for all uses, it is often a good idea to define **<<** for a user-defined type. That way, we can at least trivially write out objects of the type during debugging and early development. Later, we might provide a more sophisticated **<<** that allows a user to provide formatting information. Also, if we want output that looks different from what a **<<** provides, we can simply bypass the **<<** and write out the individual parts of the user-defined type the way we happen to like them in our application.

Here is a simple output operator for **Date** from §9.8 that simply prints the year, month, and day comma-separated in parentheses:

```
ostream& operator<<(ostream& os, const Date& d)
{
        return os << '(' << d.year()
                      << ',' << int(d.month())
                      << ',' << d.day() << ')';
}
```

This will print August 30, 2004, as **(2004,8,30)**. This simple list-of-elements representation is what we tend to use for types with a few members unless we have a better idea or more specific needs.

In §9.6, we mention that a user-defined operator is handled by calling its function. Here we can see an example of how that's done. Given the definition of **<<** for **Date**, the meaning of

```
cout << d1;
```

where **d1** is a **Date** is the call

```
operator<<(cout,d1);
```

Note how **operator<<()** takes an **ostream&** as its first argument and returns it again as its return value. That's the way the output stream is passed along so that you can "chain" output operations. For example, we could output two dates like this:

```
cout << d1 << d2;
```

This will be handled by first resolving the first **<<** and after that the second **<<**:

```
cout << d1 << d2;        // means operator<<(cout,d1) << d2;
                         // means operator<<(operator<<(cout,d1),d2);
```

That is, first output **d1** to **cout** and then output **d2** to the output stream that is the result of the first output operation. In fact, we can use any of those three variants to write out **d1** and **d2**. We know which one is easier to read, though.

10.9 User-defined input operators

Defining the input operator, **>>**, for a given type and input format is basically an exercise in error handling. It can therefore be quite tricky.

Here is a simple input operator for the **Date** from §9.8 that will read dates as written by the operator **<<** defined above:

```
istream& operator>>(istream& is, Date& dd)
{
        int y, m, d;
        char ch1, ch2, ch3, ch4;
        is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
        if (!is) return is;
        if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') {     // oops: format error
                is.clear(ios_base::failbit);
                return is;
        }
        dd = Date{y,Month(m),d};                    // update dd
        return is;
}
```

This **>>** will read items like **(2004,8,20)** and try to make a **Date** out of those three integers. As ever, input is harder to deal with than output. There is simply more that can – and often does – go wrong with input than with output.

If this **>>** doesn't find something in the **(** *integer* **,** *integer* **,** *integer* **)** format, it will leave the stream in a not-good state (**fail**, **eof**, or **bad**) and leave the target **Date** unchanged. The **clear()** member function is used to set the state of the **istream**. Obviously, **ios_base::failbit** puts the stream into the **fail()** state. Leaving the target **Date** unchanged in case of a failure to read is the ideal; it tends to lead to cleaner code. The ideal is for an **operator>>()** not to consume (throw away) any characters that it didn't use, but that's too difficult in this case: we might have read lots of characters before we caught a format error. As an example, consider **(2004, 8, 30}**. Only when we see the final **}** do we know that we have a format error on our hands and we cannot in general rely on putting back many characters. One character **unget()** is all that's universally guaranteed. If this **operator>>()** reads an invalid **Date**, such as **(2004,8,32)**, **Date**'s constructor will throw an exception, which will get us out of this **operator>>()**.

10.10 A standard input loop

In §10.5, we saw how we could read and write files. However, that was before we looked more carefully at errors (§10.6), so the input loop simply assumed that we could read a file from its beginning until end of file. That can be a reasonable assumption, because we often apply separate checks to ensure that a file is valid. However, we often want to check our reads as we go along. Here is a general strategy, assuming that **ist** is an **istream**:

```
for (My_type var; ist>>var; ) {       // read until end of file
        // maybe check that var is valid
        // do something with var
}
// we can rarely recover from bad; don't try unless you really have to:
if (ist.bad()) error("bad input stream");
if (ist.fail()) {
        // was it an acceptable terminator?
}
// carry on: we found end of file
```

That is, we read a sequence of values into variables and when we can't read any more values, we check the stream state to see why. As in §10.6, we can improve this a bit by letting the **istream** throw an exception of type **failure** if it goes bad. That saves us the bother of checking for it all the time:

```
// somewhere: make ist throw an exception if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

We could also decide to designate a character as a terminator:

```
for (My_type var; ist>>var; ) {       // read until end of file
        // maybe check that var is valid
        // do something with var
}
if (ist.fail()) {           // use '|' as terminator and/or separator
        ist.clear();
        char ch;
        if (!(ist>>ch && ch=='|')) error("bad termination of input");
}
// carry on: we found end of file or a terminator
```

If we don't want to accept a terminator – that is, to accept only end of file as the end – we simply delete the test before the call of **error()**. However, terminators are very useful when you read files with nested constructs, such as a file of monthly readings containing daily readings, containing hourly readings, etc., so we'll keep considering the possibility of a terminating character.