CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 4
September 10, 2018

C++ I/O

Streams

C++ I/O is done through **streams**.

Four standard streams are predefined:

- `cin` is the standard input stream.
- `cout` is the standard output stream.
- `cerr` is the standard output stream for errors.
- `clog` is the standard output stream for logging.

Data is read from or written to a stream using the input and output operators:

`>>` (for input).	Example: `cin >> x >> y;`
`<<` (for output).	Example: `cout << "x=" << x;`

Opening and closing streams

You can use streams to read and write files.

Some ways of opening a stream.

- `ifstream fin( "myfile.in" );` opens stream `fin` for reading. This implicitly invokes the constructor `ifstream( "myfile.in" )`.
- `ifstream fin;` creates an input stream not associated with a file. `fin.open( "myfile.in" );` attaches it to a file.

Can also specify open modes.

To test if `fin` failed to open correctly, write `if (!fin) {...}`.

To close, use `fin.close();`.

Reading data

Simple forms. Assume `fin` is an open input stream.

- `fin >> x >> y >> z;` reads three fields from `fin` into `x`, `y`, and `z`.
- The kind of input conversion depends on the types of the variables.
- No need for format or `&`.
- Standard input is called `cin`.
- Can read a line into buffer with `fin.get( buf, buflen );`. This function stops before the newline is read. To continue, one must move past the newline with a simple `fin.get(ch);` or `fin.ignore();`.

Writing data

Simple forms. Assume `fout` is an open output stream.

- `fout << x << y << z;` writes `x`, `y`, and `z` into `fout`.
- The kind of output conversion depends on the types of the variables or expressions..
- Standard output is called `cout`. Other predefined output streams are `cerr` and `clog`. They are usually initialized to standard output but can be redirected.
- Warning: The eclipse debug window does not obey the proper synchronization rules when displaying `cout` and `cerr`. Rather, the output lines are interleaved arbitrarily. In particular, a line written to `cerr` **after** a line written to `cout` can appear **before** in the output listing. This won't happen with a Linux terminal window.

Manipulators

Manipulators are objects that can be arguments of `>>` or `<<` but do not necessarily produce data.

Example: `cout << hex << x << y << dec << z << endl;`

- Prints `x` and `y` in hex and `z` in decimal.
- After printing `z`, a newline is printed and the output stream is flushed.

Manipulators are used in place of C formats to control input and output formatting and conversions.

Implementation of Manipulators

Manipulators are recognized by having a special function type, e.g, `std::ios_base& hex( std::ios_base& str );`.

The operators `>>` and `<<` have been predefined to recognize manipulators by their type and to take appropriate action when they are encountered.

Print methods in new classes

Each new class should have a `print()` function that writes out the object in human-readable form.

`print()` takes a stream reference as an argument that specifies which stream to write to.

The prototype for such a function should be:

```
    ostream& print( ostream& out ) const;
```

If `sq` is an object of the new class, we can print `sq` by writing

```
    sq.print(out);
```

Note that `const` prevents `print()` from modifying the object that it is printing.

Extending the I/O operators

While `sq.print()` allows us to print `sq`, we'd rather do it in the familiar way

`out << sq;`.

Fortunately, C++ allows one to extend the meaning of `<<` in this way. Here's how.

```
inline
ostream& operator<<( ostream& out, const Square& sq ) {
    return sq.print(out);
}
```

Since this function is inline, it should go in the header file for class `Square`.

Remarks on operator extensions

- Every definable operator has an associated function.
  The function for `<<` is `operator<<()`.
- Extending `<<` is simply a matter of defining the corresponding method for a new combination of parameters.
- In this case, we want to allow `out << sq`, where `out` has type `ostream&` and `sq` has type `const Square&`.
- The use of reference parameters prevents copying.
- The `const` is a promise that `operator<<` will not change `sq`.

Why << returns a stream reference

Both `print()` and `operator<<()` return a stream reference.

This allows compound constructs such as

```
    out << "The square is:  " << sq << endl;
```

By left associativity of `<<`, this is the same as

```
    ((out << "The square is:  ") << sq) << endl;
```

Must it be inline?

If one wants `operator<<()` to be an ordinary function, the following changes are needed:

1. Declare the operator in header file `Square.hpp`:

   ```
   ostream& operator<<(ostream& out, const Square& sq);
   ```

2. Define the operator in code file `Square.cpp`:

   ```
   ostream& operator<<(ostream& out, const Square& sq) {
       return sq.print(out);
   }
   ```

End of File and I/O Errors

Status bits

I/O functions set status flags after each I/O operation.

`badbit`	means there was a read or write error on the file I/O.
`failbit`	means the data was not appropriate to the field, e.g., trying to read a non-numeric character into a numeric variable.
`eofbit`	means that the end of file has been reached.
`goodbit`	means that the above three bits are all off.

The whole state can be read with one call to `rdstate()`.

Status functions

Functions are also provided for testing useful combinations of status bits.

- `good()` returns true if the `good` bit is set.
- `bad()` returns true if the `bad` bit is set.

  This is *not* the same as `!good()`.
- `fail()` returns true if the `bad` bit or the `fail` bit is set.
- `eof()` returns true if the `eof` bit is set.

As in C, correct end of file and error checking require paying close attention to detail of exactly when these state bits are turned on. To continue after a bit has been set, must call `clear()` to clear it.

What `eof` means

Detecting and properly handling end of file is one of the most confusing things in C++.

The `eof` flag may or may not be on after the last byte of the file has been read and returned to the user.

The `eof` flag is turned on *when the stream attempts to read beyond the end of the file*.

To understand `eof` requires a thorough understanding of how stream input works.

When eof is turned on

A stream is a sequence of bytes. `>>` reads bytes until it has a complete representation of the object that it is trying to read.

Whether `eof` is turned on depends on whether or not the current input operation can complete based on the bytes read so far, *without looking ahead at the following byte*.

- If it needs the lookahead to detect completion and the bytes representing the data object go all the way to the end of the file, then it will try to read beyond the end of the file and will turn on the `eof` bit.
- If it doesn't need the lookahead, then it will stop reading, and the `eof` flag will remain off.

Reading an `int`

Consider what `cin >> x` does when reading the `int x`.

1. It first skips whitespace looking for the start of the number in the stream. It reads bytes one at a time until either there are no more left to read or a non-whitespace byte is read. If the first happens, no data is read into `x`, and both the `fail` and the `eof` flags are turned on (and the `good` flag is turned off).
2. If step 1 ended by finding a non-whitespace byte, then the stream checks if the character just read can begin an integer. The ones that can are `+, -, 0, 1, ..., 9`. If it is not one of these, the `fail` flag is set, the `eof` flag remains off, and nothing is stored into `x`.

Reading an `int` (cont.)

3. If an allowable number-starting character is found, then reading continues character by character until a character is read that can *not* be a part of the number currently being read, or the end of file is encountered so no more characters can be read.

   Reading then stops. If a stopping character was read, it is put back into the input buffer and the stream pretends that it was not read. If reading stopped because of an attempt to read past the end of the file, the `eof` flag is turned on.

   In either case, the characters read so far are converted to an `int`, stored into `x`, and the `fail` flag remains off. The `eof` flag is on iff reading was stopped by attempting to read past the end of the file.

Examples

The following examples show the remaining bytes in the file, where ␣ represents any whitespace character such as space or newline.

1. File contents: ␣123
   An attempt to read past the end of the file is made since otherwise one can't know that the number is 123 is complete. `good` and `fail` are off and `eof` is on.
2. File contents: ␣␣123␣
   `eof` will be off and the next byte to be read is the one following the 3 that stopped the reading. `good` is on and `fail` and `eof` are off.
3. File contents: ␣
   No number is present. Step 1 reads and discards the whitespace and attempts to read beyond the end of file. `good` is off and `fail` and `eof` are on.

Common file-reading mistakes

We now talk about the practical issue of how to write your code to correctly handle errors and end of file.

Two programming errors are common when reading data from a file:

- Failing to read the last number.
- Reading the last number twice.

Failing to read the last number

`good` is not always true after a successful read.

If the last number is *not* followed by whitespace, then after it is successfully read, `eof` is true and `good` is false. If one incorrectly assumes this means no data was read, the last number will not be processed.

Here's a naive program that illustrates this problem:

```
do {
   in >> x;
   if (!in.good()) break;
   cout << " " << x;
}
while (!in.eof());
cout << endl;
```

On input file containing 1␣2␣3, it will print ␣1␣2.

Reading the last number twice

`eof` is not always true after the last number is read.

If the last number *is* followed by whitespace, then after it is read, `eof` will still be false. If one incorrectly assumes it is okay to keep reading as long as `eof` is false, the last read attempt will fail and the input variable won't change.

Here's a naive program that illustrates this problem:

```
while (!in.eof()) {
   in >> x;
   cout << " " << x;
}
cout << endl;
```

On input file containing 1␣2␣3␣, it will print ␣1␣2␣3␣3.

How to read all numbers in a file

Here's a correct way to correctly read and process all of the numbers. Instead of printing them out, it adds them up in the register s.

```
int s=0;
int x;
do {
   in >> x;
   if (!in.fail()) s+=x;  // got good data
} while (in.good());
if (!in.eof()) throw Fatal("I/O error or bad data");
```