TCC CSCI-2843 C++ Programming Language

Lecture 8 Notes

8.1 Stream I/O

Streams are buffered, **character-oriented** objects that are capable of performing formatted I/O. The base classes

- std::istream
- std::ostream
- std::iostream

form the basis of I/O using the adopted **stream insertion** and **stream extraction** operators, **operator**<< and **operator**>>, respectively. They are an example of the use of polymorphism in the standard library as all streams inherit from one of these and implements device-specific behavior.

These classes themselves inherit from a common base class, call std::ios_base (I/O System Base), which holds stream states and perform common operations that do not exhibit polymorphic behavior. The overall hierarchy of streams is even more complicated than this lets on, so we will work with a simplified view until we need to know some of the details.

8.1.1 Stream Buffers

Being buffered, streams use buffer classes derived from the std::streambuf class. Most of the heavy lifting is done with these stream buffer classes, but these rarely appear in user code, instead being manipulated behind the scenes by the stream objects that own them. It is possible to perform buffered, unformatted, character-oriented I/O directly with these buffer classes if necessary, however this is rare compared to the number of times you will only need to use the public interface of the stream classes themselves.

8.1.2 Stream States

I/O streams are always in one of four possible states

- good – the stream is ready for input or output
- bad – the stream has encountered a hard error (usually in the underlying layers)
- fail – the last I/O operation failed
- eof – the stream has encountered the “end-of-file” condition

Given a stream, s, these states may be tested for directly using the functions

```
if (s.good()) { ... }
if (s.fail()) { ... }
if (s.bad()) { ... }
if (s.eof()) { ... }
```

When we use the type conversion operator on the stream,

```
if (!s) { ... error }
```

the stream can be used like a **bool** value, where if the stream is in the good state it will be converted to a **true** value, but if it is in the bad, fail, or eof state, it will convert to a **false** value.

Internally, the state is kept as a mask of bits that are specified by the std::ios_base::iostate **enum**. These are

```
std::ios_base::goodbit
std::ios_base::badbit
std::ios_base::failbit
std::ios_base::eofbit
```

We usually only use these bits when working with streams at a lower level. The mask that indicates the state of the stream can be read by the rdstate function:

```
std::ios_base::iostate state = s.rdstate();
```

and tested using the bite-wise "and"

```
if (state & std::ios_base::failbit) { … stream is in a "fail" state }
```

which is the same as

```
if (s.fail()) { … }
```

The state of the stream may be cleared using

```
s.clear();
```

or set to a particular mask

```
s.clear(std::ios_base::badbit | std::ios_base::failbit);
```

which in this example, will set the state to bad and fail.

8.1.3 File Streams

File I/O in C++ is usually accomplished through file streams, which are in the <fstream> header. The three classes we will discuss are

- std::ofstream
- std::ifstream
- std::fstream

All of these file streams have the ability to open and manage a file and contain a buffering mechanism that increases I/O efficiency. They provide constructors that are able to specify the file to open and also provide a destructor that will ensure that the buffer is flushed and the file is closed.

8.1.4 Output

The ofstream class provides streaming output to a file. An example of declaring and creating an std::ofstream is

```
#include <fstream>

std::ofstream a_stream{"test.dat"};
```

An std::ofstream object can be used like any other output stream, particularly using the **operator**<<:

```
int i = 0;
std::string s = "this is a test: ";
a_stream << s << i << '\n'
```

Since ofstream inherits from the std::ostream base class, an std::ofstream object can be passed in as a reference to an std::ostream:

```
#include <fstream>

void f(std::ostream& s) {
    s << "hello";
}

int main() {
    std::ofstream a_stream{"test.dat"};
    f(a_stream);
    return 0;
}
```

An ofstream may be default constructed and then opened at a later time:

```
std::ofstream a_stream;
...
a_stream.open("test.dat");
```

Likewise, an ostream may be closed manually if you wish to close the stream before the object goes out of scope:

```
a_stream.close();
```

Be warned that before a stream is successfully opened and after a file has been manually closed, all I/O operations to that stream will fail. It is common and perhaps best practice that (when feasible) to construct the file stream when it is needed, specifying the file to open on the constructor, and then letting the destructor close it when the stream goes out of scope.

8.1.5 Open Flags

There a number of flags that may be specified as a second argument to a file stream's constructor that will affect its operation.

For instance, if you wish to open a file for output, but leave the contents that are already there (i.e. append to the file), you may specify the ios_base::app flag as a second argument:

```
std::ofstream a_stream{"test.dat", std::ios_base::app};
```

All ensuing output will be tacked on to the existing content. There is a similar flag, std::ios_base::ate (at end) that will open the stream at the end of the file, but can have its file position freely moved around. The opposite of these flags is the default, which is the std::ios_base::trunc flags, which specified that any existing data in the file is deleted (truncated).

Another flag pertains to the old problem of treating end-of-line characters (EOL). Since some platform use carriage return/line feed (CRLF) pairs, but most use just line feed, but others in the past have used just carriage return, normally there is a "normalization" that occurs for text that changes these platform-specific EOL denotations into the single newline character (which is actually an LF in ASCII) that we have come to know. That means that when you write something like

```
std::cout << "Hello\n";
```

the single newline character can be expanded to whatever the EOL representation is on the platform performing the output. The inverse operation occurs on input.

To turn off this normalization on a stream and get exactly what characters are stored in a file, use the std::ios_base::binary flag on the constructor:

```
std::ifstream input{"data.file", std::ios_base::binary};
```

The reason they call this "binary" is that it also enables you to read a data file byte-by-byte in a binary-wise fashion without the normalization accidently transforming the values read from a file. Best advice: if you are performing **any** binary type of I/O or need to examine every byte in the file without interference, be sure to specify the ios_base::binary flag, otherwise, let the normalization take care of all the platform-specific requirements.

Note – these flags can be or'ed together so you can specify multiple flags upon construction:

```
std::ifstream input{"data.fil", std::ios_base::binary | std::ios_base::app};
```

8.1.6 Input

The std::ifstream class provides streaming input from a file. It inherits from std::istream, so it can be used everywhere a std::istream is taken by reference, in particular using the **operator**>>.

8.1.7 Input/Output

The std::fstream class inherits from the std::iostream base class which multiply inherits from **both** std::istream **and** std::ostream. This means that if we instantiate an fstream, we may pass it in wherever either an istream or ostream are taken by reference.

Because we can perform both input and output on an std::fstream, we need to be aware that when we switch from input to output or vice-versa, the underlying buffer of the stream is flushed. This is usually not a problem if we perform inputs and outputs separate and do not interleave them.

8.2 Text Input

String input is somewhat surprising and tedious. If we have a data file (test.dat) that looks like this

```
This is a line of data
This is another line
```

then, given an istream, in, the code

```
std::string s;
in >> s;
```

will only read in the string "This" from the first line. The **operator**>> for std::string will only read up to the first **whitespace** character (as defined by isspace). Even if we were to code

```
in >> std::setw(7) >> s;  // won't work as expected
```

while this compiles and runs just fine, you still get only "This" from the file instead of the expected seven characters, "This is". In other words, input ignores most formatting, which can be particularly irritating for std::string. This forces us to use other means to read in formatted data.

8.2.1 "Line"-oriented Input

To read an entire "line" at a time, use the std::getline function:

```
std::istream& getline(std::istream& s, std::string& str, char terminator = '\n');
```

which inputs from the stream, s, input characters into the string, str, until the terminator character is encountered (which defaults to a newline). Note that there is no practical limit to prevent ridiculous amounts of data from being read into a string (theoretically, a read operation could read **gigabytes**-worth of data into a dynamic string, causing a crash).

If the terminator is reached, the operation succeeds and the std::getline function removes the terminator character from the stream. If the terminator character is not reached before end of the stream, the stream's state is set to "fail". This ensures the integrity of the operation. The std::getline function returns a reference to the std::istream so that we can test the status of the operation.

Even though this is called get**line**, because you can specify the terminator character, you can use this to read any delimited data. For instance, given an std::istream, s, and the data

```
This is string 1, this is string 2, and string three.
```

we can write

```
std::string s1;
std::string s2;
std::string s3;
std::getline(s, s1, ',');
std::getline(s, s2, ',');
std::getline(s, s3);
```

which will input the value "This is string 1" in s1, "This is string 2" in s2, and "and string three" in s3. Note that we did not test the value returned from our std::getline calls, but probably should:

```
if (!std::getline(s, s1, ',')) { ... s1 failed }
if (!std::getline(s, s2, ',')) { ... s2 failed }
if (!std::getline(s, s3)) { ... s3 failed }
```

or you can combine the checks if you desire

```
if (!std::getline(s, s1, ',' || !std::getline(s, s2, ',') || !std::getline(s, s3) {
    ... one of them failed
}
```

8.2.2 Single Character Input

To read one character at a time, we can use two different methods. The first uses the "get" member function. Given an std::istream, in, we can write

```
char c;
in.get(c)
```

which will read the next character and **remove it** from the stream, so that the next call to get will read the next character and so on. The get function returns a reference to the std::istream, so that we may check

The second uses the "peek" member, which simply returns the next character in the stream **without removing it**. A peculiarity of peek is that is returns an **int** instead of a **char**. This is because it may return a value outside of the range of **char**, namely the value EOF.

```
int c = in.peek();
if (c == EOF) { ... end of file }
... use c as if it was an 'char'
```

8.2.3 Ignoring Characters

The peek function can be called multiple times and will always return the same character. To skip the character, use

```
in.ignore();
```

To ignore multiple characters, we can supply a count:

```
in.ignore(10);
```

which will skip ten characters.

To ignore multiple characters up to but **not including** a given character, we can supply a count and a terminating character:

```
in.ignore(10, ',');
```

which will ignore up to ten characters or when it encounters a comma, whichever occurs first. The operation is similar to std::getline.

Just like most stream functions, we can test the return value from ignore to ensure that the operation succeeded:

```
if (!in.ignore(10)) { ... cannot ignore ten characters }
```

8.3 Numeric Input

Numeric input can also be surprising, but just remember that most formatting is ignored during input operations. For instance, integer input ignores the width setting on the stream, just like string input does. Worse yet, the following statement

```
int i;
in >> i;
```

by definition will read characters from the stream, in, as long as those characters can be interpreted as belonging to an **int without regard** to width settings. As an example, given a file

```
1234987xyz
```

and the code,

```
int i, j;
std::string s;
in >> std::setw(4) >> i >> std::setw(3) >> j >> s;
```

one would expect that the integer, 1234, would be read into i and 987 into j. However, the input ignores the width setting and the input into i greedily eats **all** the digit characters, placing the value, 1234987 into i, leaving the input into j with the characters "xyz" (intended for our string, s). The input into j will fail on encountering the 'x', leaving the stream in the fail state and no further characters read in.

Floating point format has similar deficiencies:

```
float f;
in >> f;
```

will read characters from the stream as long as they form a valid **float**, irrespective of the width and precision settings.

We draw from this the conclusion that we cannot use the **operator**>> to perform fixed-width numeric input. We can however, use this for **delimited** data.

8.4 Delimited Data

In delimited data, where each item is separated by a known character such as a comma (',') or vertical bar ('|'), we can freely use the **operator**>> to input numeric data and std::getline to read string data. With a file, test.dat whose content is

```
1234,this is a string,12.5
```

we can write

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream input{"test.dat"};
    if (!input) {
        std::cerr << "cannot open file" << std::endl;
        return 1;
    }

    int field1;
    std::string field2;
    double field3;

    // field 1 input
    input >> field1;
    if (input.eof()) {
```

```
        // eof tested separate
        std::cerr << "no data" << std::endl;
        return 2;
    }
    if (input.fail() || input.peek() != ',' || !input.ignore()) {
        // either it was not an integer or it was not followed by ','
        std::cerr << "error on field1" << std::endl;
        return 3;
    }

    // field 2 input
    if (!getline(input, field2, ',')) {
        std::cerr << "error on field 2" << std::endl;
        return 4;
    }

    // field 3 input
    input >> field3;
    if (!input) {
        // we make no distinction between eof or other fail/bad conditions
        std::cerr << "error on field3" << std::endl;
        return 5;
    }

    // successful read of all three fields
    std::cout << field1 << ", " << field2 << ", " << field3 << std::endl;
    return 0;
}
```

8.4.1 Whitespace

The **operator**>> uses a flag on the stream known as the skipws flag. This flag can be set or unset manually using the std::ios_base::skipws flag. For convenience, it is usually set and unset by the std::skipws and std::noskipws manipulators, like

```
s >> std::noskipws >> i;
```

When set, the **operator**>> will skip all whitespace before it reads the object in question. When set off, the whitespace is treated as part of the data and must be accounted for.

Note that this is different than functions such as the get(**char**) function, which always gets the next character, regardless of whitespace. Only the **operator**>> uses the skipws flag.

8.5 String Streams

The standard provides stream types that work on internal memory instead of external devices. Because these buffers hold characters, these stream types are known as “string streams”. They provide input and output facilities to representations easily and efficiently converted from and to std::string. To access these classes, use the following include

```
#include <sstream>
```

8.5.1 The std::ostringstream Class

The std::ostringstream class provides a method to perform output to an internal, dynamic memory buffer that is immediately convertible to a std::string.

```
...
std::ostringstream s;
int result = 10;
std::string label = "The result is ";
s << label << result;
std::string text = s.str();
...
```

Once constructed, we can use an ostringstream as any other ostream (from which it inherits). When we are done, we can simply call the str member function which returns a string that has the formatted contents of whatever we output to the stream. All standard formatting available with any other ostream applies to the ostringstream, making this a nice way to convert numeric data into their textual equivalent.

8.5.2 The std::istringstream Class

To perform input using an already-constructed string, use the std::istringstream class:

```
std::string data = "123 456 text";
int i;
int j
std::string text;
std::istringstream input{data};
input >> i >> j >> text;
```

Once constructed, an istringstream provides all the input facilities of any other istream (from which it inherits). It also inherits all the limitations of any other istream (i.e. there is no high-level input formatting), but can be useful for turning textual representations of numeric data into their numeric representations or picking through a string as if it were coming from a file.

8.6 Example

The following example defines a class named complex that provides stream-based input and output facilities. For the purposes of this example, the acceptable format for a complex is

whitespace? (*whitespace*? *real-part whitespace*? , *whitespace*? *imaginary-part*)

where ? indicates "optional" and *real-part* and *imaginary-part* are floating point numbers formatted compatibly with the standard **operator**>> for **double**. Some examples would be

```
(0, 0)
( -10,   -10  )
   (      1.0e-3   ,   -7    )
```

The following is a sample program showing how this could be implemented. First, it defines the complex **class**, with the requisite operators. Then, the main constructs a complex, writes it to a file, then reads it back again into a different complex.

```
#include <iostream>
#include <fstream>

class complex {
private:
    double re_;
    double im_;

public:
    complex() : re_{0.0}, im_{0.0} { }
```

```
    complex(double init_re, double init_im) :
        re_(init_re),
        im_(init_im)
    { }

    double re() const { return re_; }

    double im() const { return im_; }

    friend std::ostream& operator<<(std::ostream& s, const complex& c) {
        return s << '(' << c.re_ << ", " << c.im_ << ')';
    }

    friend std::istream& operator>>(std::istream& s, complex& value) {
        double r;
        double i;
        char c;
        if (!(s >> c) || c != '(' || !(s >> r >> c) || c != ',' || !(s >> i >> c) || c != ')') {
            s.setstate(std::ios_base::failbit);
        } else {
            value.re_ = r;
            value.im_ = i;
        }
        return s;
    }
};

int main() {
    complex a = { 123.45, -67.89 };
    std::cout << "a: " << a << '\n';

    std::ofstream out{"complex.txt"};
    out << a << '\n';
    out.close();

    std::ifstream in{"complex.txt"};
    complex b;
    in >> b;
    if (!in) {
        std::cout << "ERROR!\n";
    } else {
        std::cout << "b: " << b << '\n';
    }
    return 0;
}
```

The line of code in the **operator>>**,

```
if (!(s >> c) || c != '(' || !(s >> r >> c) || c != ',' || !(s >> i >> c) || c != ')') {
```

works with the whitespace setting on the stream (i.e. the **operator**>> for **double** and **char** will **skip** whitespace as long as the streams skipws flag is **true**). If we wanted to force the ability to eat whitespace, we could have used the std::ws manipulator to manually eat whitespace between the components. As written, the caller has the ability to set the stream to either eat whitespace or not. In general, it is good not to enforce a policy on a caller, and instead allow the format to obey the "normal" I/O conventions.

Note also, the error checking in the **operator**>> might not be quite exact – we mix the error of a failure to read double or char with the error of the characters not being '(', ',', or ')', which is fundamentally different. To be more exact (if anyone actually cared), we would only set the fail bit if the wrong character is read, otherwise leaving the stream state alone and passing it back as is (fail, bad, or eof). This comes at the expense of more code and a less "elegant" solution.

```
friend std::istream& operator>>(std::istream& s, complex& value) {
    if (!(s >> c)) { return s; }
    if (c != '(') {
        s.setstate(std::ios_base::failbit);
        return s;
    }
    if (!(s >> r >> c)) { return s; }
    if (c != ',') {
        s.setstate(std::ios_base::failbit);
        return s;
    }
    if (!(s >> i >> c)) { return s; }
    if (c != ')') {
        s.setstate(std::ios_base::failbit);
        return s;
    }
    value.re_ = r;
    value.im_ = i;
    return s;
}
```