Chapter 14 - Streams

While we have been using streams since the first session, writing data to standard out and standard err with our friends
`cout` and `cerr`, and reading user input with `cin`, we havent really touched on what streams are exactly,
and what you can do with them.

Well, ultimately, C++ streams are an abstraction to handle character input and output in a uniform
fashion, whether the target is a buffer, a file, a string, the networ, or any device that holds characters. Streams work
with built in data types, and provide a simple mechanism to allow you to extend them to work with custom data types,
through the stream operators (<< and >>).

Internally, streams are built on top of a character buffer - the streambuf - which is periodically
flushed to the target. Because the buffer does not hold the entire contents of the stream in memory, (hence stream),
the one thing you cannot do with a stream is access a random point in the stream. Streams are serial and they flow
from or to the user to the destination.

basic output

C++ provides standard stream objects for handling *Console I/O*. They include cin, cout, cerr, endl, and clog, adn they
are available via the **iostream** header. You remember these right? You should be using them all the time.

Review Time....

You write to standard output using the **cout** object, along with the stream operator <<.

```
std::cout << "I am an output" << "hear me roar" << std::endl;
```

Cout will handle type conversion for you, so you can write basic types other than strings as well.

```
int meaning=42;
std::cout << "Meaning of life is supposedly " << meaning << std::endl;
```

Likewise, you can write to standard error almost exactly the same way:

```
std::cerr << "Warning Will, an ERRROR has occured. Leave Mission Control immediately" << std
```

setting flags on cout and cerr

Of course, you can modify the formatting of the stream output; you are not limitted to how the stream operators handle
things by default. There are a wealth of options, and two wqys of affecting change. The first method involves invoking
a method on the stream output classes.

You can set output stream flags using the `setf` method on cout or cerr.

You can reset output stream using `unsetf`.

```
#include <iostream>
#include <ios>
using namespace std;

int main() {
    std::cout.setf(std::ios::hex, std::ios::basefield);
    std::cout.setf(std::ios::showbase);
    std::cout.setf(std::ios::uppercase) ;
    std::cout << "In hex: " << 77 << std::endl;

return 0;
}
```

formatted output

C++ provides a separate library to help you control the formatting of streams. It is called **iomanip**. It provides
commands which you apply on the stream, and which change all subsequent interaction. There are a couple of ways of
achieving this. First, you can set flags on the stream object itself. Flags are provided in the *ios* header, which
may or may not come in when you include *iostream* (it is supposed to as of c++11).

We use binary OR to select multiple flags.

```
#include <iostream>
#include <ios>
using namespace std;

int main() {
cout.setf(ios::uppercase | ios::showpos);
cout << "positive?" << 27 << " or negative " << -27 << endl;
```

```
return 0;
}
```

A word of warning. setf only turns flags off. It isn't smart enough to disable conflicting flags. That is your job.
For instance, the following wont work:

```
cout.setf(ios::hex);
cout << 27 << endl;
```

Because decimal input defaults to on (ios::dec). You have to do the following:

```
cout.unsetf(ios::dec);
cout.setf(ios::hex);
cout << 27 << endl;
```

You can drop configuration commands directly into the stream

The first mechanism presented to change formatting tends to be a bit of a pain. Fortunately, C++ provides a better
mechanism - an inline mechansim. You can drop formatting directly into the stream and it sticks around until you turn
it off.

```
cout << hex << 27 << endl;
cout << dec << 27 << endl;
```

Useful formatters

- boolalpha - booleans print true and false
- showpos - prefix positive numbers with +
- uppercase

These formatter has a related formatter formed by sticking the word "no" in front. This will turn the formatter off.
(eg noboolapha)

justifying text

- internal
- left
- right

Other formatters are part of exclusive groups.

floatfield group

- dec - decimal
- hex - base 16

- oct - base 8

formatting numbers

- fixed
- scientific
- normal
- showpoint - always show a decimal point

setw(n)

Sets the fill width of the input.

```
#include <iostream>
#include <iomanip>

int main() {

using namespace std;
cout << setw(7);
cout << 77 << 44 << "pizza" << endl;
return 0;
}
```

setfill(char)

Sets the fill character for manipulation. Can be used in concert with setw.

```
#include <iostream>
#include <iomanip>

int main() {

using namespace std;
cout << setfill('x') << setw(7);
cout << 77 << 44 << "pizza" << endl;
return 0;
}
```

setprecision(n)

setprecision is used to format floating point values on output operations.

```
#include <iostream>
#include <iomanip>
```

```
int main() {
float num 4.3452635684786634634574;

using namespace std;
cout << setprecision(3) << "the number is " << num << endl;
return 0;
}
```

setiosflags (ios_base::fmtflags mask);

```
#include <iostream>     // std::cout, std::hex, std::endl
#include <iomanip>      // std::setiosflags

int main () {
  std::cout << std::hex;
  std::cout << std::setiosflags (std::ios::showbase | std::ios::uppercase);
  std::cout << 100 << std::endl;
  return 0;
}
```

ios::flags

The std::ios module has a number of flags which can be set to change the stream behavior, either by using setiosflags

Input

Of course, printing to the console isn't the whole story here. You can also get input from the user in the console via
`cin`. Cin grabs a single input value from the user like so:

```
int age;
cout << "How old are you?";
cin >> age;
cout << "did you say that you are " << age " years old?" << endl;
```

If you want to grab multiple input values from the user, you can do that as well.

```
int age ;
int height;
cout << "state your age and height";
cin >> age >> height;
cout << "well you are " << age <<" with a height of " << height << endl;
```

What about reading a whole line of input from the user? Well, you can use std::getline, which takes a stream
and a string like so:

```
std::string line;
std::getline(std::cin, line);
// proceeed
```

A word of caution. If you are trying to call getline after calling cin previously, there is a good chance that
the getline call will pick up the last carriage return from the user. In order to handle this, you will have to
call ignore on the input before calling getline.

```
int age;
string quote;
cout << "please enter your age" ;
cin >> age;
cin.ignore();
cout << "And provide us with a memorable quote";
getline(cin,quote);
cout << "really? I am totally going to forget \"" << quote << "\""<< endl;
```

Error Handling

In the previous example, if you tried it out and entered some bogus answer - say "cow" for your age, you will see
something potentially disturbing. Cin doesn't barf or complain. It simply skips the offender. So how do you know if
the input values were good? You can check the state of the stream. If there was a problem, the state of the stream
will be bad and you will actually have to clear it before continuing.

You can check the stream state using a couple of methods. In true C++ fashion, there is some granularity around
the state itself:

Function	Explanation
bool good()	true if no error flag is set
bool eof()	true if eofbit is set
bool fail()	true if failbit or badbit is set
bool bad()	true if badbit is set
bool operator!()	same as fail
iostate rdstate()	state of stream

So, what is the different between *failbit* and *badbit* ?

According to cplusplus.com:

```
failbit is generally set by an input operation when the error was related to the internal lo
```

operation itself, so other operations on the stream may be possible. While badbit is general
when the error involves the loss of integrity of the stream, which is likely to persist ever
different operation is performed on the stream. badbit can be checked independently by calli
function bad.

In other words, if you get a number when expecting a letter, that results in a *failbit*. If a serious, non
recoverable error happens which disrupts the abiltiy to read from the stream at all, that is a *badbit*.

It turns out handling all the cases where the user can screw up is a bit of a pain. Read this article for more info:

https://gehrcke.de/2011/06/reading-files-in-c-using-ifstream-dealing-correctly-with-badbit-failbit-eofbit-and-pe

dealing with files

Up to this point, we have been dealing exclusively with the console. However, streams are good for more than this.
One common need is to read and/or write to a file. Streams have you covered.

Reading a file into a string

The header *ifstream* provides a stream class which is capable of reading from a file. If you want to read a
file into a string, you can do the following. Create an ifstream instance and use it along with a streambuf iterator
to populate a string:

```
std::ifstream in("file.txt");
std::string contents((std::istreambuf_iterator<char>(in)), std::istreambuf_iterator<char>())
```

reading a file line by line

You can read a file a line at a time using getline in a loop.

```
int cnt=0;
if(std::ifstream in("file.txt")) {
    for(std::string line; getline(input, line);) {
        std::cout << cnt << " " << line << endl;
        cnt++;
    }
} else {
    std::cerr << "unable to open file.txt" << std::endl;
}
```

ifstream implements RAII so the file is closed when the instance goes out of scope. You can also call
close() on it explicitly if you want.

Writing to a file

Writing to a file is quite simple. Given the fact that we already know how to write to a stream, we just need to
open a file stream, and close it when we are done.

We use the header *ofstream* to access a file stream class suitable for outputting.

```
ofstream fh("/tmp/foo.txt");

fh << "this is a test" << endl;
fh.close();
```

Modes

ifstream and ofstream support explicit modes of operation. These are provided via an optional second argument
which specifies a number of flags. Here are those flags:

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks to the end ofthe file before reading/writing
binary	Opens the file in binary mode instead of text mode
in	Opens the file in read mode (default for ifstream)
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it exists

strings as streams

In addition to the console and file, you can actually interact with strings using the stream api thanks to
stringstreams. You can create a stringstream by first including the *sstream* header and then using
std::stringstream.

Stringstreams can be used to compose strings using the very same stream api we have been learning.

```
stringstream foo;
string name{"Frank"};
```

```
int age = 22;
foo << "name:" << name << "age:" << age;
```

They are great when you need to convert between different types and strings, or use stream
formatting. And once you have composed a stringsream, you can get a string from it by calling the *str()* method.

```
string mystring = foo.str();
```

As a note, you can pass around references to streams in your functions, allowing you to operate on
a variety of types of streams. Want to write a function which can read from or write to a file or
a string? (for, say, testing purposes) Take an appropriate stream reference. (ostream or istream).

ofstream inherits from ostream. ifstream inherits from istream.

Extra Credit - string formatting

c-style - what's old is new again (thanks to C++11)

One thing missing here in C++ land is good string formatting a la python.
There are a couple of approaches to dealing with this.
The first is to use the c style sprintf in place of streams. This can work, although sprintf returns a char*, not a string.
However, with a bit of c++11 magic we can make this happen:

```
// from http://stackoverflow.com/questions/2342162/stdstring-formatting-like-sprintf
#include <memory>
#include <iostream>
#include <string>
#include <cstdio>

using namespace std; //Don't if you're in a header-file

template<typename ... Args>
string string_format( const std::string& format, Args ... args )
{
    size_t size = snprintf( nullptr, 0, format.c_str(), args ... ) + 1; // Extra space for '
    unique_ptr<char[]> buf( new char[ size ] );
    snprintf( buf.get(), size, format.c_str(), args ... );
    return string( buf.get(), buf.get() + size - 1 ); // We don't want the '\0' inside
}

// now use the template
```

This approach uses a variadic template to allow us to pass an arbitrary number of inputs to a function, along with
snprintf, which is c++11's version of c's sprintf. Lets try and understand what is going on here.

The first line, we are simply calculating the size of a buffer big enough to hold the resulting char*.

```
size_t size =snprintf(nullptr, 0, format.c_str(), args ...) +1
```

The next line we create a char* buffer of the appropriate size on the heap, storing it in a unique pointer so that
destruction is handled for us.

```
unique_ptr<char[]> buf( new char[ size ] );
```

The next line, we call snprintf again, this time passing it a pointer to our buffer (buf is a unique_ptr. calling
buf.get() returns a pointer to the internal char*). This is the line which is actually doing the formatting for us.
(ok well, we do it twice. but the first time we do it just to see how big the result will be)

```
snprintf( buf.get(), size, format.c_str(), args ... );
```

Finally, we populate a string with the buffer and return it. This form of string constructor takes pointers to
the start and end of the char*.

```
return string( buf.get(), buf.get() + size - 1 ); // We don't want the '\0' inside
```

We can use this template magic like so:

```
cout << format_string("I like %d goldfish in my %s. How about %s?", 3, "soup", "you") << end
```

New style python formatting

If you are a fan of new style python formatting, or if you just like following the cool kids, then *fmt* is the
project for you. It is hosted on github here:

https://github.com/fmtlib/fmt

and has documentation here:

http://fmtlib.net/latest/index.html

And is quite popular. Using it is very simple, as it is, like all cool projects, a header only library (optionally).
It is fast, safe, small, speedy, and doesn't really have any dependencies. To use, simply download the source,
include it in your project, and

```
#define FMT_HEADER_ONLY 1
```

Before #including any of the fmt headers.