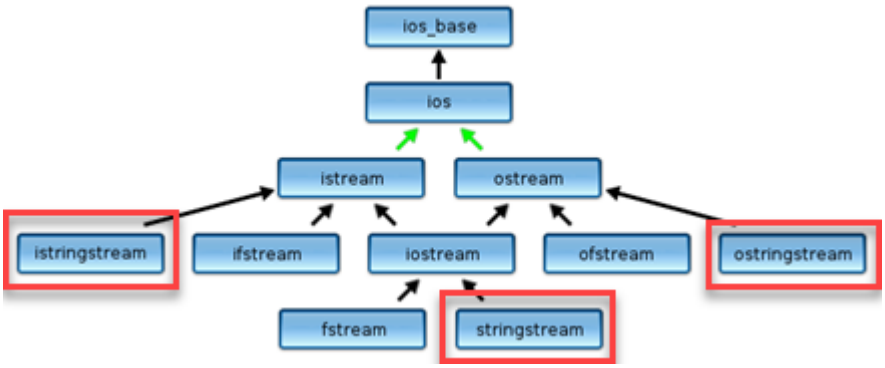
Chapter

13

String Streams

The **<sstream> header contains classes that allow you to associate a stream with a string** in memory in the same way that **<fstream>** allows you to associate a stream with a file. The ***istringstream*** class is similar to ***ifstream***; it allows you to use stream operators to read data from a **string** (instead of a file.)



The ***ostringstream*** class works like ***ofstream*** except that the output is directed to a **string** rather than a file. Here are the three steps you need to follow:

1. Create an ***ostringstream*** object: ***ostringstream out;***
2. Write to the object: **out << "stuff";**
3. Collect the results: **string result = out.str();**

This is most useful when you want a **formatted number** as part of some other output. The standard library has a **to_string()** function (starting with C++11) that works for all types of numbers. Unfortunately, for floating-point numbers, you have **no control over the output format**, which isn't very useful. Let's fix that.

Here's a short function with two arguments: a **double**, and the number of decimals to display, and returns the value as a **string**. Note the default argument as well.

```
string format(double amt, int prec=2)
{
    ostringstream out;
    out << fixed << setprecision(prec) << amt;
    return out.str();
}
```

Use this to format output like this:

```
cout << format(2.456) << endl;        // 2.46
cout << format(1.0 / 3.0, 5) << endl; // .33333
```

Input String Streams

The C++11 **string** class introduced several new functions, like **stoi()** and **stod()**, that convert **string** to **int** and **double**. Prior to C++11, you used the ***istringstream*** class.

```
istringstream in{"January 3, 2020"};
```

The ***istringstream*** (or **input string stream**) variable named **in** allows you to **parse** all of the pieces of the date that has been supplied, and, in a much easier manner than using the **string** functions like **find()** and **substr()**:

```
string month; int day, year;
in >> month >> day; // read month and day
in.get();           // consume comma
in >> year;         // read and convert year
```

For complex values, like dates, the ***istringstream*** technique is the most efficient.

A String Stream Exercise

Using an input string stream is also the easiest way to **parse** the individual parts of a line of text. Let's solve a problem that puts this to work. Click on the running man to open the starter code in **CodeCheck**.

> *Write a function* `inputStats` *which takes an input stream and an output stream as arguments. Report the number of lines in the file, the longest line, the number of tokens on each line, and the length of the longest token on each line. Assume at least one line of input and that each line has at least one token.*

For example, if input contains the following text:

```
"Beware the Jabberwock, my son,
the jaws that bite, the claws that catch,
Beware the JubJub bird and shun
the frumious bandersnatch."
```

Then the output should be:

```
Line 1 has 5 tokens (longest = 11)
Line 2 has 8 tokens (longest = 6)
Line 3 has 6 tokens (longest = 6)
Line 4 has 3 tokens (longest = 14)
Longest line: the jaws that bite, the claws that catch,
```

Step 1: The Longest Line

When you tackle a complex problem like this, you should always tackle it **one piece at a time**. Let's start with this:

1. Reading the entire input file, line-by-line
2. Finding the longest line
3. Printing the longest line to the output file

The Arguments

The two arguments to `inputStats` are an **input stream** and an **output stream**. What kind of streams should you pass? You should be able to **call** the function like this:

```
inputStats(cin, cout);
inputStats(inFile, outFile);  // ifstream, ofstream
inputStats(inStr, outStr);    // istringstream, ostringstream
```

To make sure it will work with all of these stream types, you must use the **most general** types: `istream` and `ostream`. In addition, **all** stream types must be passed **by reference**, not by value.

Reading the Input

To read line-by-line, use `getline()` and the **data-loop pattern**:

```
string line;
while (getline(in, line))
{
   . . .
}
```

Finding and Printing the Longest Line

To find the longest line, you must:

1. **Create a variable** to hold the longest line (before the loop)
2. Compare the **current line size to the longest line size** (in the loop)
3. **Print** the longest line (after the loop)

Here I've highlighted the added lines:

```
string line;
string longestLine;
while (getline(in, line))
{
    if (line.size() > longestLine.size())
    {
        longestLine = line;
    }
}
out << "Longest line: " << longestLine << endl;
```

Note: make sure that you read from your input stream, and write to your output stream. You **should not** use **`cin`** or **`cout`** in this problem; use your parameters.

Step 2: Counting the Lines

To process and **count the lines**, you must:

1. Create a variable to hold the **line number** **before** the **`while`** loop
2. **Increment** the variable every time **a line is read**
3. **Print** "Line <line number>:" at the end of the **`while`** loop

Here's the code to do that:

```
…
int lineNumber{0};
while (getline(in, line))
{
    ++lineNumber;
    if (line.size() > longestLine.size()) {
       longestLine = line;
    }
    out << "Line " << lineNumber << " has ..." << endl;
}
out << "Longest line: " << longestLine << endl;
```

Step 3: Counting the Tokens

To process and **count the tokens**, you must re-read each line, token-by-token. Do this by:

1. Create an **input string variable** using the line
2. Read (and count) each **token** using the extraction operator and a **`while`** loop

Here's the code that does this:

```
. . .
while (getline(in, line))
{
    if (line.size() > longestLine.size()) . . .
    istringstream strIn(line);
    int nTokens{0};
    string token;
    while (strIn >> token) { ++nTokens; }
    out << "Line " << lineNumber << " has "
        << nToken << " tokens ..." << endl;
}
out << "Longest line: " << longestLine << endl;
```

Step 4: The Longest Token

To find the longest token, follow the same pattern which you used to find the longest line.

```
    int longestToken{0};
    while (strIn >> token)
    {
        ++nTokens;
        if (token.size() > longestToken) {
            longestToken = token.size();
        }
    }
    out << "Line " << lineNumber << " has "
        << nToken << " tokens (longest = "
        << longestToken << ")" << endl;
```

Integrate this code and your function should work correctly.

Conditional Compilation

The standard library has several functions, in the `<string>` header, that will convert a C++ string to a number.

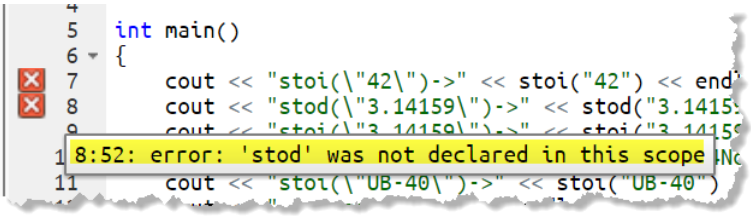
- the **`stoi(string)`** function returns an **`int`**
- the **`stod(string)`** function returns a **`double`**
- There are also **`stof()`**, **`stol()`**, **`stoul()`** and **`stold()`** for the types **`float`**, **`long`**, **`unsigned long`** and **`long double`** respectively.

These functions **don't appear** in C++ 98. If your employer is using Visual Studio 10 and doesn't want to change to a later version, for a wide (and reasonable) variety of reasons. What can you do? Implement them yourself, of course.

> *This is not a made-up scenario; we are using a recent version of C++ in CS 150, but in the real world, you'll need to be prepared for older versions.*

Using the Conversion Functions

Your goal is to be able to compile **exactly the same file** in C++98, C++11, C++14 and C++17, and to **get the same results each time**. Click the "running-man" icon on the left to find our test program. Click Run and you'll see that it looks fine. Now do **exactly** the same thing, but **change to C++98** in the Compiler Options.



OOPS! That doesn't look encouraging! What's wrong???

C++98 does not have those functions, so you get an **undeclared identifier**. You can fix that by **implementing these two functions** yourself, using the string stream classes.

Stub the Functions

Place these **definitions** for the functions right above **`main`**.

```
int stoi(const string& str) { return 0; }
double stod(const string& str) { return 0.0; }
```

Make sure you are compiling with C++98 and **click Run**. The code compiles and "runs" (although our stubs don't produce the correct value, of course).

What happens when you upgrade to Visual Studio 19? Will the code still compile? **Nope!** Your version of **`stoi()`** **conflicts with** the one already defined inside the new C++ standard library; you get a **clash of symbols**.

At link time, there can be **only one copy** of the **`stoi()`** function in the executable; if the library already has one, **your program won't link**. What you would like to say is: "if using C++11 or later use the library version, and, if using an older version of C++, use the version that I've written". You can do this with **conditional compilation**.

Conditional Compilation

The preprocessor supports **conditional expressions**, using **preprocessor directives** to conditionally include a section of code based on **`#defined`** values:

```
#if statement
...
#elif another-statement
...
#else
...
#endif
```

This occurs **before the rest of the code is sent to the compiler;** these conditions can **only** refer to **`#defined`** constants, integer values, and arithmetic and logical expressions of those values.

```
#define MY_CONSTANT 42
#if MY_CONSTANT > 137               // Legal
#if MY_CONSTANT * 42 == MY_CONSTANT // Legal
#if sqrt(MY_CONSTANT) < 4 // Illegal, cannot call function sqrt
#if MY_CONSTANT == 3.14 // Illegal, can only use integral values
```

You can use the predefined preprocessor function **defined()** to check if a constant has been previously been **#defined**.

- **#ifdef** is short for **if defined**; **#ifdef symbol** is like **#if defined(symbol)**.
- **#ifndef** is short for **if not defined**, the opposite of **#ifdef**.

Conditional compilation determines whether pieces of code are sent to the compiler.

```
#if defined(A)
    cout << "A is defined." << endl;
#elif defined(B)
    cout << "B is defined." << endl;
#else
    cout << "Neither A or B is defined." << endl;
#endif
```

When the preprocessor encounters this, whichever conditional expression is true will have its corresponding code block included in the final program. If **A** is **#defined**, then this entire portion of code will go to the compiler as:

```
cout << "A is defined." << endl;
```

Predefined Compiler Symbols

Your toolchain **predefines several constants** that you can test in your code[1] [2]. If you want to define a particular function only when working on a particular platform, you can surround the code some **#ifdef** directives like this from StackOverflow which tests for different operating systems:

```
#ifdef _WIN32
   //define something for Windows (32-bit)
#elif __APPLE__
  // define something for OSX
#elif __linux
    // linux
#endif
```

You, of course, for this problem, only care about a **particular version of** C++. In the list of predefined standard constants, you'll see that **__cplusplus** (double leading underscores) contains version numbers for each release of C++. You can use that to bracket our own versions of the **stoi()** and **stod()** functions.

[1] http://sourceforge.net/p/predef/wiki/OperatingSystems/

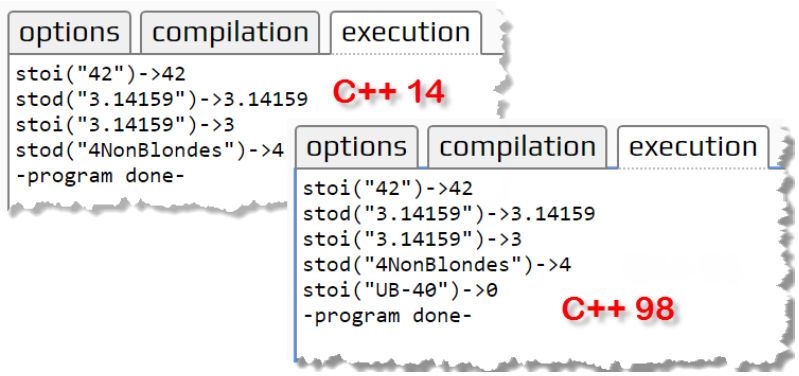
[2] http://sourceforge.net/p/predef/wiki/Compilers/

Go back to your test program and use this facility to define the functions **only** if the identifier `__cplusplus` is `<= 199711L`. Now you can compile and run with C++98 and with C++11/14 using the same source.

To actually implement the functions, just use code like this:

```
function stoi <- input str -> output int
    set result to 0
    construct an input string stream using str
    read from str into result
    return result
```

The `stod()` function is almost identical. Now, when you run our test program C++, **it produces exactly the same output** as it does under C++11, which uses the `stoi()` from the standard library.



Or does it?

What if you pass `stod()` or `stoi()` invalid input? Look at the lines highlighted in yellow:

```
cout << "stoi(\"42\")->" << stoi("42") << endl;
cout << "stod(\"3.14159\")->" << stod("3.14159") << endl;
cout << "stoi(\"3.14159\")->" << stoi("3.14159") << endl;
cout << "stod(\"4NonBlondes\")->" << stod("4NonBlondes") << endl;
cout << "stoi(\"UB-40\")->" << stoi("UB-40") << endl;
```

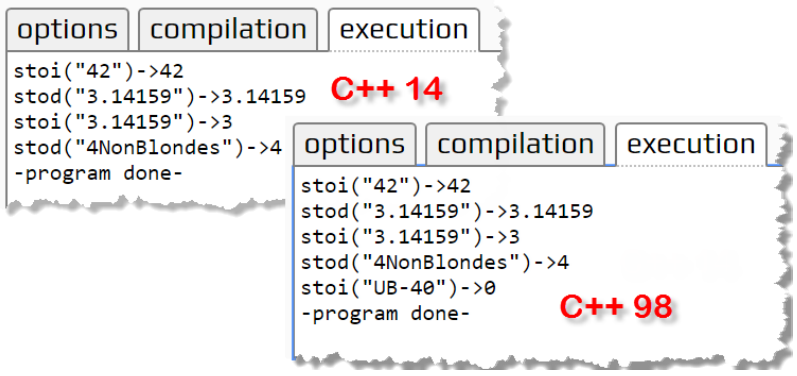
C++14 and C++98 produce the same output for the first four inputs, but the last one fails entirely. Neither the library nor your version fails on `stoi("3.14159")`. Both convert what they can (the `3`) and leaves the rest. But, the library version crashes with `stoi("UB-40")`; there is no possible conversion.

So, that means the version we wrote is better, right?
After all, who wants a function that crashes?

Well, not so fast. The question is, **what should** `stod()` and `stoi()` do with invalid input? In the next lesson, we'll use these techniques to look at more **input validation**.

Errors & Assertions

What should the `stoi()` and `stod()` functions do when given invalid input? The C++ 14 version, from the standard library, does one thing, while the version you wrote, does something else entirely. Open that version by clicking the "running man" icon.



To answer this question, first consider what **should** happen when you try to:

- Print the square root of `-2`?
- Open a file that doesn't exist? Read data from that file?
- Convert a `string` that doesn't contain a number to a number?

Each of these is **handled in a different** way. None of them are syntax or linker errors. Instead, they are **runtime errors**. The compiler and linker produced an executable, but when it runs, an error occurs.

Let's examine a few ways to **handle** such errors.

The Terminator

One option is to write a function like `die()`, which prints an error message and then terminates. This is not really a good solution for runtime errors unless they are very severe, since it **doesn't give the user a chance to recover**. Imagine if your Web browser **shut down** every time you typed a URL incorrectly or clicked on a dead link.

Using a function like `die()`prevents the program from continuing with garbage values, but it is simply too drastic and **too inflexible** to be a good universal approach.

However, there is one time when a "terminator" **is the correct way** to handle errors:

- When you are developing your code and ...
- When the error is a programming problem that can be fixed.

In fact, the C++ library has a **built-in macro** which does this, called **`assert()`**.

The assert Macro

An assertion is a statement about a condition **which must be true** when encountered. If the condition is **not true**, then **`assert`**, (declared in **`<cassert>`**), causes the program to immediately fail, printing an error message. Here's an example:

Programmers use assertions to **reason about logical correctness**. Assertions can be used to check **preconditions** (what must be true before the program runs correctly), and **postconditions** (what must be true after a calculation completes).

Here is an (admittedly silly) example using **`assert`**.

```
cout << "Making sure that 2 + 2 is 5?" << endl;
assert(2 + 2 == 5); // false
```

The programmer assumed (**wrongly**) that **`2 + 2`** was **`5`**. The assertion causes the program to stop and **print an error message**, so the programmer can fix the mistake. The message will depend on the toolchain. Here is **g++** on Unix.

```
Making sure that 2 + 2 is 5?
a.out: main.cpp:10: int main():
    Assertion `2 + 2 == 5' failed.
Aborted (core dumped)
```

The message includes the executable name (**`a.out`**), the source file (**`main.cpp`**), the line number (**`10`**), the function name and the assertion which failed.

Steve Maguire, one of the original developers of Excel, wrote a classic book named Writing Solid Code, which contains a chapter on assertions in C. He writes:

- Assertions are shorthand way to write **debugging checks**
- Use assertions to check for **illegal conditions**, not error conditions
- Use assertions to **validate function arguments** under your control
- Use assertions to validate any **assumptions** you have made

Static Assertions

C++ 11 also introduced the **static_assert()** declaration which may be used to double-check your assumptions about the platform you are developing on. For instance, if your code assumes that the **int** type is a 32-bit signed number, you can check that with:

```
static_assert(sizeof(int) == 4, "Int must be 32 bits.");
```

Unlike regular assertions, **static_assert** is checked when you compile; it does not check for runtime errors. You can only check on compile-time constants and the error message must be a string literal; you cannot include variables.

In C++17 you may omit the error message.

Tradition! Completion Codes

A second error-reporting option is the "tried-and-true" traditional **completion code technique** used for years in C, Pascal and FORTRAN. Have your function **return a special value** meaning that "the function failed to execute correctly."

In a way, this is what **sqrt()** does; it returns the "special" **nan** value when its answer cannot be converted to a valid **double**. You can test for this value using the **isnan()** function in the header **<cmath>**. You could use the "error code" like this:

```
if (isnan(answer = sqrt(-1))) { /* error */ }
```

The **isnan()** function was added to C++ 11. Before that, **sqrt()** set the global variable **errno**, defined in **<cerrno>**, which was used like this.

```
double answer = sqrt(-1.0); // invalid
if (errno == EDOM) { /* invalid DOMain */ }
```

A Variation: Error State

With the advent of object-oriented programming, a variation on completion codes was birthed—**error state** which is encapsulated in objects. Of course, you've already encountered this with the input stream classes.

Here's an example. What happens if the user enters "twelve"? What happens if the keyboard is not working?

```
cout << "Enter an integer: ";
int n;
cin >> n;    // Error state is set here
```

Each stream object has an **internal data member** that contains an individual error code, or **error flag**. These flags are given names like **badbit**, **goodbit** and **failbit**. If the user enters "twelve", then the **failbit** is "set". If the keyboard isn't working, the **badbit** is set.

In the C-style of programming, you use **bitwise logical operators** (something we won't cover in this class, but you'll probably encounter in Computer Architecture) to read or set each of these error codes. In C++, however, you have **member functions**:

```
cin >> n;    // Error state is set here
if (cin.fail()) // Check if failbit is set
{
    cin.clear(); // clear all of the error flags
    // empty the input stream and try again
}
```

The big problem with completion codes and with error states, is that **you can ignore the return value without encountering any warnings**. Research has shown that programmers almost **never** check them. To better handle these kinds of problems, C++ introduced **exception handling**. If an error occurs inside a function, rather than returning a value, you report the problem and **jump to the proper error-handling code**.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **make submit**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.