Processing Lines

Since text files are usually arranged by lines, it is often useful to read an entire line of data at one time. The easiest way to do that is to use the function named `getline()` in the `<string>` library. `getline()` is not a member function, and it takes two arguments:

- the **input stream from which** the line is read. (Open the stream as shown in the previous sections.)
- a `string` variable **into which** the result is written

By default, `getline()` stops when it encounters a newline, which is **removed** from the stream and **discarded**. It **is not** stored as part of the string. Like `get()`, the `getline()` function **returns** the input stream, which allows you to test for end-of-file.

```
1  string line;
2  while (getline(in, line))
3      cout << line << endl;
```

This `while` loop reads each line of data from the stream into the `string` variable named `line`, until the stream reaches the end of the file. For each line, the body of the loop uses `<<` to send the line to `cout`, followed by a newline character to replace the one which was discarded by `getline()`.

Processing Tokens

We can also process input token by token or word by word. The word token means "a chunk of meaningful data". A token may be an integer, a number, a string, or a custom type, like stars or points.

As you've already seen, you read a token using the **extraction operator** `>> var`. If `var` is a `string`, this reads a single word. When `var` is an `int`, it reads an integer, and so on.

The input operation **returns the stream**, just as with raw input and line-oriented input. We could process input **word-by-word**, like this:

```
string word;
while (in >> word) // process the word
```

Of course, the phrase "word-by-word" is not exactly correct, since a "word" may include punctuation, or be a number, and so on. Technically, it is **token-by-token**.

Applied File I/O

Let's apply what you've learned about file I/O. Click on the "Running Man" to open CodeCheck in a separate window. Leave the window open as we work through the process of writing a function named `searchFile()` which takes two `string` arguments.

- The first argument is the **name of the file** to open
- The second argument is the **word or phrase to search for**

Neither `string` may be modified, and there is no return value. Here are the **specifications** for the function:

- Open the file and **read it line by line**.
- If the phrase you are looking for is found in that line, then print the line number (in a field 5 wide), a space, a colon, another space and then the line from the file, followed by a newline.
- Assume the first line number in the file is line #1.
- Your output should be printed with `cout`.
- If the file cannot be found, then print an error message (using `cerr`, of course): "File *fname* cannot be opened".

Once you have the program opened, go to the next page, and we'll complete the **mechanics** of the function.

Stub the Function

▶ *You can check your work here, or peek if you get stuck.*

You should be able to stub out the function on your own. You shouldn't have to think about this part; you should memorize and practice each of these steps until they become second nature.

1. **Open the file using the supplied filename**.
2. If it can't be opened, then print the error message using `cerr`. To check if the file was opened, **explicitly** use the `fail()` function or, **implicitly** check using the stream variable itself. If you use the second method, don't forget the **not operator**.
3. Instead of adding an `else` to the `if` statement, add a `return` statement to end the function if nothing else can be done.
4. Add the **line-oriented I/O pattern**. You will have read and printed every line, and so you'll be ready to go on to the next step, where you actually solve the problem.

Searching & Numbering

▶ *You can check your work here, or peek if you get stuck.*

The searching part of this problem is easy. Place the output inside an *if* statement. Use `find()` as part of your condition. If the word is found, then print the line. (Remember, if a word is not found, that the `find()` member function will return the value `string::npos`.)

Numbering the line is also very easy.

- Create a **line counter** right before the loop starts.
- In the loop, **increment the counter** each time a line is read.
- Instead of printing the line when the phrase is found, print the line number, using **formatted output** before printing the contents.

To print the line number in a field **five character wide** you'll need to use the `setw()` manipulator (which you met in `H01`). Follow that with a space, a colon and another space, and finally the line itself.

Validating Data

▶ *You can check your work here, or peek if you get stuck.*

With raw, line-by-line or string-based token-oriented input, a data loop only fails when it reaches end-of file. However, consider the `sumNumbers` filter, which you can open by clicking the little "running man" on the right.

This function reads and processes `doubles`, returning the sum. This function works fine when provided with a stream that contains nothing but `double`s. It **fails** when in cannot read a `double`; it also fails, of course, when it reaches end-of-file. Let's fix the function so that it processes **all** of the valid data in the file.

Stream Flags

All stream objects contain a set of *Boolean* variables, known as the **state flags**. You can check the value of these flags by calling one of the stream's member functions:

- `fail()` mean the stream is in the failed state. It will not accept any more input.
- `good()` means the stream is ready to read more input
- `eof()` means there is no more input for the stream to read.

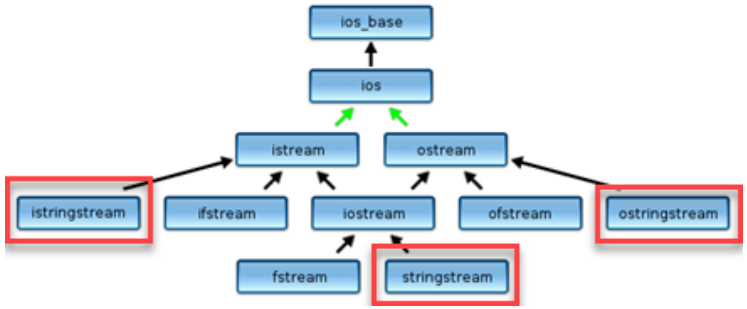
When the stream object is placed in a **failed state**, **no error message is printed**; the rest of the input is simply not processed and the input stream stops working. To fix:

1. Read the stream while it is good. The easiest way is simply `while (in)`.
2. Only sum the number `if (in >> number)`
3. Otherwise, if you **haven't** reached `in.eof()`
    1. **Reset** the **error state**, by calling the member function: `cin.clear()`
    2. **Remove** the offending token from the stream with `in >> bad_data` where `bad_data` is a string.
    3. You may **print an error message** to `cerr`.

See if you can get it to work. You can check your work with the solution above, (or peek if you get stuck).

String Streams

The `<sstream>` header contains classes which allow you to associate a stream with a `string` in memory, in the same way that the classes in `<fstream>` allow you to associate a stream with a file. Looking at the class hierarchy below, you can see that `istringstream` is a kind of `istream`, (just as `ifstream` is), while `ostringstream` is a kind of `ostream`, just like `ofstream` is.



To use a string stream for output, follow these three steps:

1. Create an `ostringstream` object.
2. Write to the stream object.
3. Collect the results using the stream's `str()` member function.

```
ostringstream out;               // 1. Create the stream
out << "The answer is " << 42;   // 2. Write to the stream
string result = out.str();       // 3. Collect the results
```

As you can see, this is most useful when you want a **formatted number** as part of some other output.

The format Function

Let's look at an example, where string strings can be very useful. The standard library has a **`to_string()`** function (starting in C++11) which 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`** for the value, and the number of decimal digits to display The function returns the value as a formatted string. (Note the default argument, which makes the function easier to use.)

```
1  string format(double value, int digits=2)
2  {
3      ostringstream out;
4      out << fixed << setprecision(digits) << value;
5      return out.str();
6  }
```

You can use the **`format()`** function 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()`, which convert `string` values like `42` to `int` or `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.

An Input String Stream Exercise

Using an input string stream is the easiest way to parse the individual parts of a line of text. Let's solve a problem which 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,*

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

Take a look at the starter code for the `inputStats` function. Reading the documentation comment, you can see that the two arguments it expects are an **input stream** and an **output stream**.

```
 9  /**
10   *  Write the procedure inputStats
11   *  @param in the input stream to read from
12   *  @param out the output stream to write to
13   *  @pre input stream is filled with at least one line of text
14   *  @post output stream populated as shown above
15   */
16  void inputStats(..., ...)
17  {
18    . . .
19  }
20
21  ////// Place your code above this line ///////////
```

The first question is "What kind of streams should you pass?" You should be able to **call** the function in any of these ways:

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

To make sure it will work with **all** of these stream types, you must use the **most general** types for your parameters: **`istream`** and **`ostream`**.

In addition, **all** stream types must be passed **by reference**, not by value. If you forget this, your code will not compile, and the error message will not be helpful at all.

options | compilation | execution

```
main.cpp:14:16: error: call to deleted constructor of 'istream'
    inputStats(cin, cout);
               ^~~
```

The Longest Line

▶ *Check Your Work*

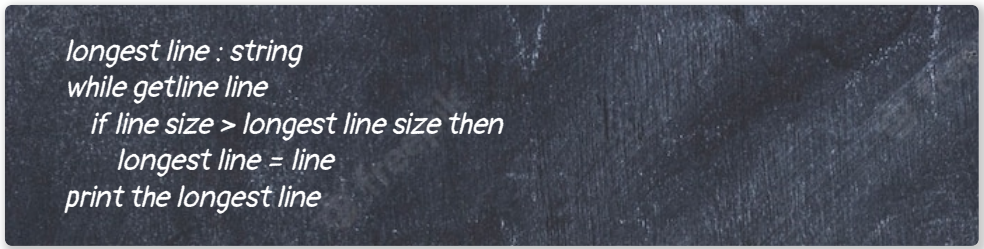
To read line-by-line, use `getline()` and the data-loop pattern:

```
string line;
while (getline(in, 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's the pseudocode for this part of the problem. Once you've turned it into C++, check your work with the code at the top of the page.



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.

Counting the Lines

▶ *Check Your Work Here*

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 pseudocode to do that with the new lines highlighted. Try it out and then check your work with the code I've supplied.

```
longest line : string
line number <- 0
while getline line
   line number = line number + 1
   if line size > longest line size then
      longest line = line
   Print "Line <line number> has . . ."
Print "Longest line: " longest line
```

Processing the Tokens

▶ *Check Your Work Here*

To process and count the tokens, you must re-read each line, token-by-token. To do that you must:

1. Create an **input string stream variable** using the line.
2. Read (and count) each **token** from the string stream.

Here's the pseudocode that does this:

```
str_in : istringstream(line)
number of tokens <- 0
token : string
while str_in >> token
   number of tokens = number of tokens + 1
Print "Line <line number> has <number of tokens> tokens . . ."
```

The Longest Token

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

```
longest token <- 0
while str_int >> token
   if token size > longest token then
      longest token = token size
```

Implement this code and your function should work correctly. If you have difficulty, then look at the code supplied above.