Chapter Eight: Streams, Part I

Slides by Evan Gallagher & Nikolay Kirov

Chapter Goals

- To be able to read and write files

Streams

A very famous bridge
over a “*stream*”

Streams

A ship

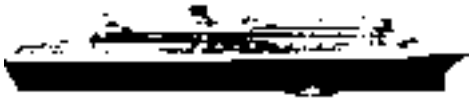
Streams

in the stream

Streams

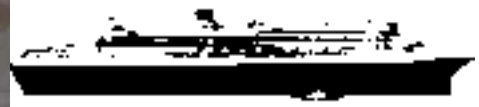
one at a time

Streams



A ***stream*** of ships

Streams



an *input stream* to
that famous city

Streams

Streams

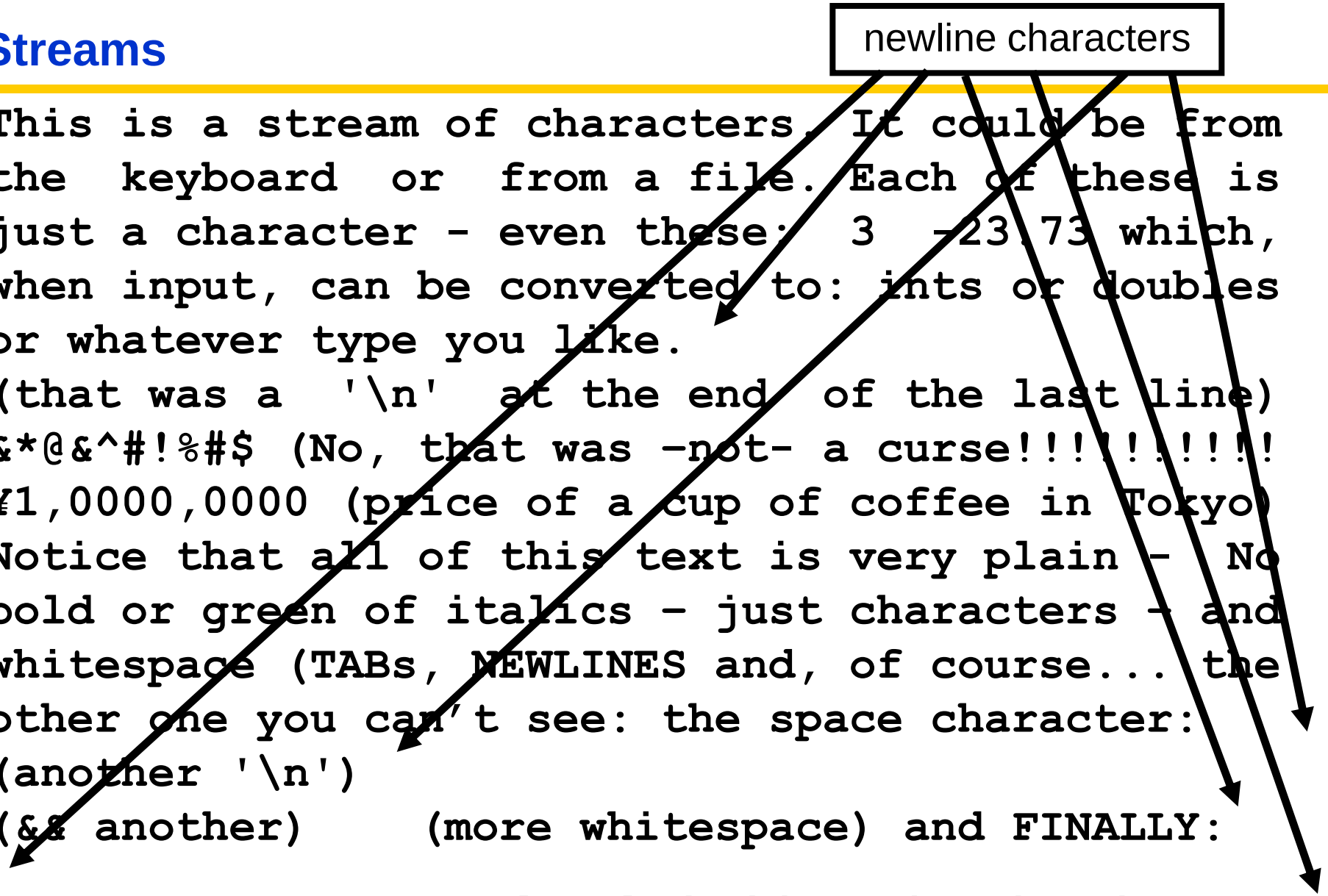
No more ships in the stream at this time

Let’s process what we just input…

Reading and Writing Files

- The C++ input/output library is based on the concept of *streams*.
- An *input stream* is a source of data.
- An *output stream* is a destination for data.
- The most common sources and destinations for data are the files on your hard disk.

Streams



Reading and Writing Files

The stream you just saw in action is a plain text file.
No formatting, no colors, no video or music
(or sound effects).

The program can read these sorts of plain text streams of characters from the keyboard, as has been done so far.

Reading and Writing Files

You can also read from files stored on your hard disk:

from plain text files
(as if the typing you saw had been stored in a file)
(or you wanted to write those characters to a disk file).

from a file that has binary information
(a binary file).

The picture of the ship in the stream of ships is stored as binary information.

Reading and Writing Files

You will learn to read and write both kinds of files.

Reading and Writing Files

To read or write disk files, you use *variables (objects)*.

You use *variables* of type:

- **`ifstream`** for input from plain text files.
- **`ofstream`** for output to plain text files.
- **`fstream`** for input and output from binary files.

Opening a Stream

To read anything from a file stream,

you need to *open* the stream.

(The same for writing.)

Opening a Stream

Opening a stream means associating your stream variable with the disk file.

Opening a Stream

The first step in opening a file is having the stream variable ready.

Here's the definition of an input stream variable named **`in_file`**:

```
ifstream in_file;
```

Looks suspiciously like every other variable definition you've done – it is!

Only the type name is new to you.

Opening a Stream

Suppose you want to read data from a file named **`input.dat`** located in the same directory as the program.

All stream variables are objects so we will use a method. The **`open`** method does the trick:

```
in_file.open("input.dat");
```

Opening a Stream

You use the name of the disk file *only* when you open the stream.

And the `open` method only accepts C strings.

(More about this later …)

Opening a Stream

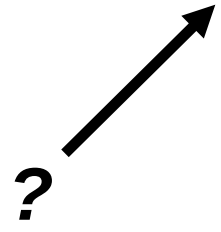
File names can contain directory path information, such as:

UNIX

```
in_file.open("~/nicework/input.dat");
```

Windows

```
in_file.open("c:\\nicework\input.dat");
```



Opening a Stream

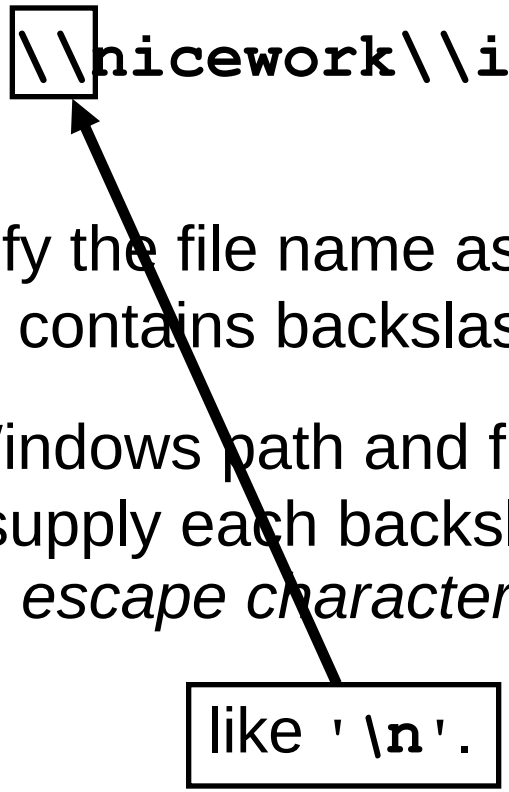
File names can contain directory path information, such as:

UNIX

```
in_file.open("~/nicework/input.dat");
```

Windows

```
in_file.open("c:\\nicework\\input.dat");
```



When you specify the file name as a string literal,
and the name contains backslash characters

(as in a Windows path and filename),
you must supply each backslash *twice*
to avoid having *escape characters* in the string,

like `'\n'`.

Opening a Stream

If you ask a user for the filename, you would normally use a **`string`** variable to store their input.

But the **`open`** method requires a C string.

What to do?

Opening a Stream

Luckily most classes provide the methods we need:
the **`string`** class has the **`c_str`** method
to convert the C++ string to a C string:

```
cout << "Please enter the file name:";
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename.c_str());
```

Opening a Stream

The **`open`** method is passed C string version of the filename the user typed:

```
cout << "Please enter the file name:";
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename.c_str());
```

Closing a Stream

When the program ends,
all streams that you have opened
will be automatically closed.

You *can* manually close a stream with the **`close`** member function:

```
in_file.close();
```

Closing a Stream

Manually closing a stream is *only* necessary if you want to open the file again in the same program run.

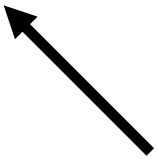
Reading from a Stream

You already know how to read and write using files.

Yes you do:

See, I told you so!

```
string name;
double value;
in_file >> name >> value;
```



`cout`? `in_file`?
No difference when it comes to reading using `>>`.

Reading from a Stream

The **>>** operator returns a “not failed” condition, allowing you to combine an input statement and a test. A “failed” read yields a **`false`** and a “not failed” read yields a **`true`**.

```
if (in_file >> name >> value)
{
   // Process input
}
```

Nice!

Reading from a Stream

You can even read ALL the data from a file because running out of things to read causes that same “failed state” test to be returned:

```
while (in_file >> name >> value)
{
   // Process input
}
```

x-STREAM-ly STREAM-lined
--- Cool!

Failing to Open

The `open` method also sets a “not failed” condition.
It is a good idea to test for failure immediately:

```
in_file.open(filename.c_str());
// Check for failure after opening
if (in_file.fail()) { return 0; }
```

Silly user,
bad filename!

Writing to a Stream

Let's review:

Do you already know everything about writing to files?

But you haven't started showing writing to files!
How can this be a review already?

But, of course, you already know!

Writing to a Stream

Here's everything:

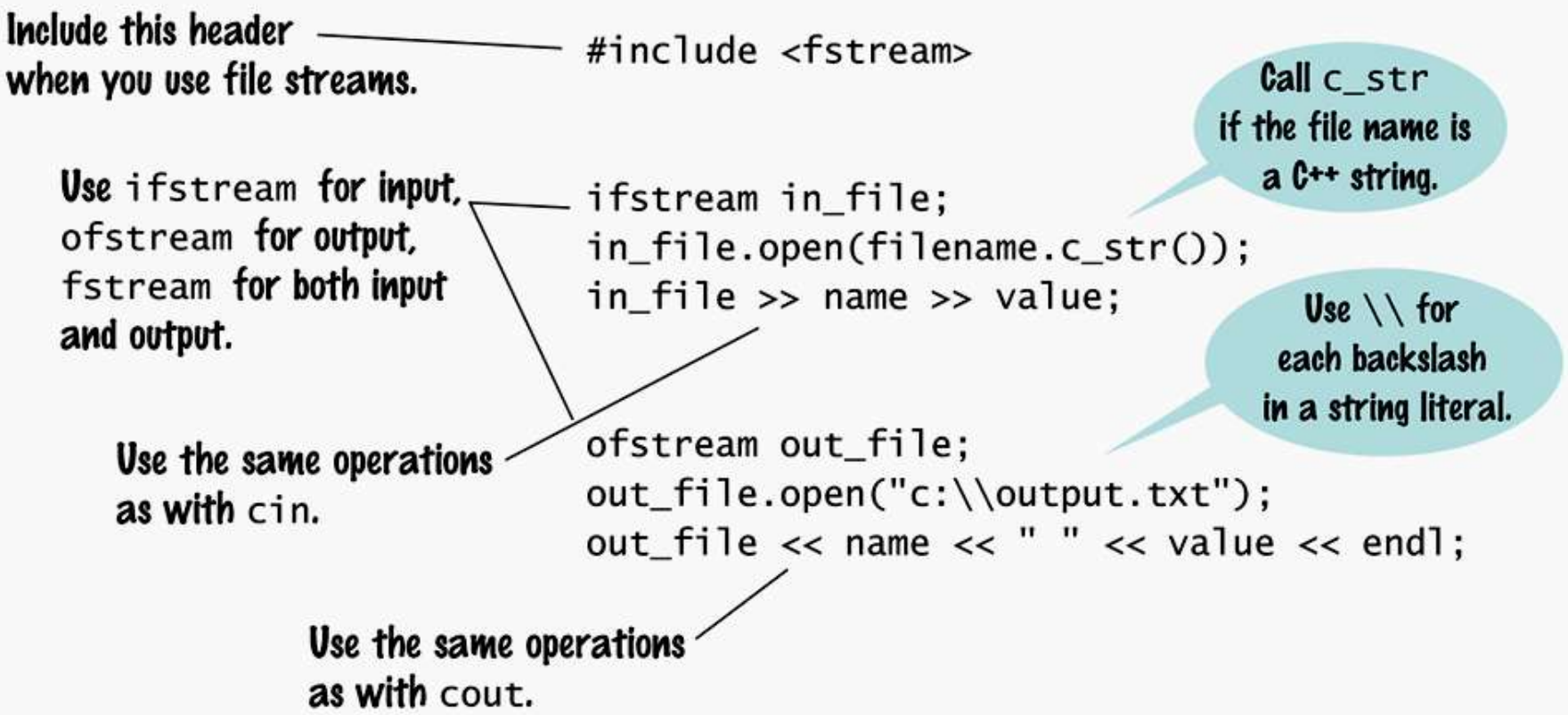
1. *create output stream variable*
2. *open the file*
3. *check for failure to open*
4. *write to file*
5. *congratulate self!*

```
ofstream out_file;
out_file.open("output.txt");
if (in_file.fail()) { return 0; }
out_file << name << " " << value << endl;

out_file << "CONGRATULATIONS!!!" << endl;
```

Working with File Streams

Syntax 8.1 Working with File Streams



Passing Streams to Functions

Functions need to be able to process files too, and there's an easy rule to follow:

As parameters, streams are

ALWAYS passed by reference.

(Did you notice that "ALWAYS"?)


Ezmereldza?
Why can I never find Ezmereldza?

A Programming Problem

Why can I never find Ezmereldza?

A Programming Problem

A Programming Problem

http://www.ssa.gov/OACT/babynames/

A Programming Problem: BABYNAMES

Popular baby names of the 1990s - Mozilla Firefox 3 Beta 5

File Edit View History Bookmarks Tools Help

http://www.ssa.gov/OACT/babynames/decades/n

Social Security Online **Popular Names**

www.socialsecurity.gov Home | Questions? | Contact Us | Search GO

Popular Baby Names

Popular Baby Names By Decade

Select another decade?

Decade Go

Number of births

Most Popular 1000 Names of the 1990s

All names are from Social Security card applications for births that occurred in the United States. The data below were extracted from our records at the end of February 2000. See limitations of such data. The most popular 1000 names of the 1990s were taken from a universe that includes 20,531,547 male births and 19,627,269 female births.

Most Popular Names of the 1990s

	Male			Female		
Rank	Name	Number	Percent[a]	Name	Number	Percent[a]
1	Michael	462,085	2.2506	Jessica	302,962	1.5436
2	Christopher	361,250	1.7595	Ashley	301,702	1.5372
3	Matthew	351,477	1.7119	Emily	237,133	1.2082
4	Joshua	328,955	1.6022	Sarah	224,000	1.1413
5	Jacob	298,016	1.4515	Samantha	223,913	1.1408
6	Nicholas	275,222	1.3405	Amanda	190,901	0.9726
7	Andrew	272,600	1.3277	Brittany	190,779	0.9720
8	Daniel	271,734	1.3235	Elizabeth	172,383	0.8783
9	Tyler	262,218	1.2771	Taylor	168,977	0.8609
10	Joseph	260,365	1.2681	Megan	160,312	0.8168
11	Brandon	259,299	1.2629	Hannah	158,647	0.8083
12	David	253,193	1.2332	Kayla	155,844	0.7940

Done Adblock

A Programming Problem: Baby Names

Let's write a program that might help Ezmereldza help her sister.

A Programming Problem: Baby Names

After copying the data from the Social Security Administration's table to a text file, we analyze the format of the file.

A Programming Problem: Baby Names

Each line in the file contains seven entries:

- The rank (from 1 to 1,000)
- The name, frequency, and percentage of the male name of that rank
- The name, frequency, and percentage of the female name of that rank

An example line from the file:

```
10 Joseph 260365 1.2681 Megan 160312 0.8
```

A Programming Problem: Baby Names

We will display the names of the top 50%
of the names stored in our file of names.

A Programming Problem: Baby Names

To process each line, we first read the rank:

```
int rank;
in_file >> rank;
```

We then read a set of three values for that boy's name:

```
string name;
int count;
double percent;
in_file >> name >> count >> percent;
```

Then we repeat that step for a girl's name.

A Programming Problem: Baby Names

Repeating a process reminds us of a good design principle:

Write a function to do this:

```
string name;
int count;
double percent;
in_file >> name >> count >> percent;
```

A Programming Problem: Baby Names

To stop processing after reaching 50%, we can add up the frequencies we read and stop when they reach 50%.

However, it turns out to be a bit simpler to have “total” variables for boys and girls and initialize these with 50.

Then we’ll subtract the frequencies as we read them.

A Programming Problem: Baby Names

When the total for boys falls below 0,
we stop printing boys' names.

Same for girls' names
– which means this can be part of the function.

When both totals fall below 0, we stop reading.

A Programming Problem: Baby Names

You'll get this when you read the code:

ch08/babynames.cpp

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

A Programming Problem: Baby Names

ch08/babynames.cpp

```
/**
Reads name information, prints the name if total >= 0, and
adjusts the total.
@param in_file the input stream
@param total the total percentage that should still be
processed
*/
void process_name(ifstream& in_file, double& total)
{
   string name;
   int count;
   double percent;
   in_file >> name >> count >> percent;
   // Check for failure after each input
   if (in_file.fail()) { return; }
   if (total > 0) { cout << name << " "; }
   total = total - percent;
}
```

A Programming Problem: Baby Names

ch08/babynames.cpp

```
int main()
{
   ifstream in_file;
   in_file.open("babynames.txt");
   // Check for failure after opening
   if (in_file.fail()) { return 0; }
   double boy_total = 50;
   double girl_total = 50;
   while (boy_total > 0 || girl_total > 0)
   {
      int rank;
      in_file >> rank;
      if (in_file.fail()) { return 0; }
      cout << rank << " ";
      process_name(in_file, boy_total);
      process_name(in_file, girl_total);
      cout << endl;
   }
   return 0;
}
```

Reading Text Input

There are more ways to read from stream variables than you have seen before, and there are some details you need to understand.

These follow…

Reading `strings`

What really happens when reading a **`string`**?

```
string word;
in_file >> word;
```

1. If any reading can be done at all, all whitespace is skipped (whitespace is this set: `'\t' '\n' ' '`).
2. The first character that is not white space is added to the string **`word`**. More characters are added until either another white space character occurs, or the end of the file has been reached.

Reading Characters

It is possible to read a single character
– including whitespace characters.

```
char ch;
in_file.get(ch);
```

The `get` method also returns the “not failed” condition so:

```
while (in_file.get(ch))
{
   // Process the character ch
}
```

Reading the Whole File Character by Character

The **`get`** method makes it possible to process a whole file one character at a time:

```
char ch;
while (in_file.get(ch))
{
   // Process the character ch
}
```

One-Character Lookahead

You can look at a character after reading it and then put it back if you so decide.

However you can only put back the *very last* character that was read.

This is called *one-character lookahead*.

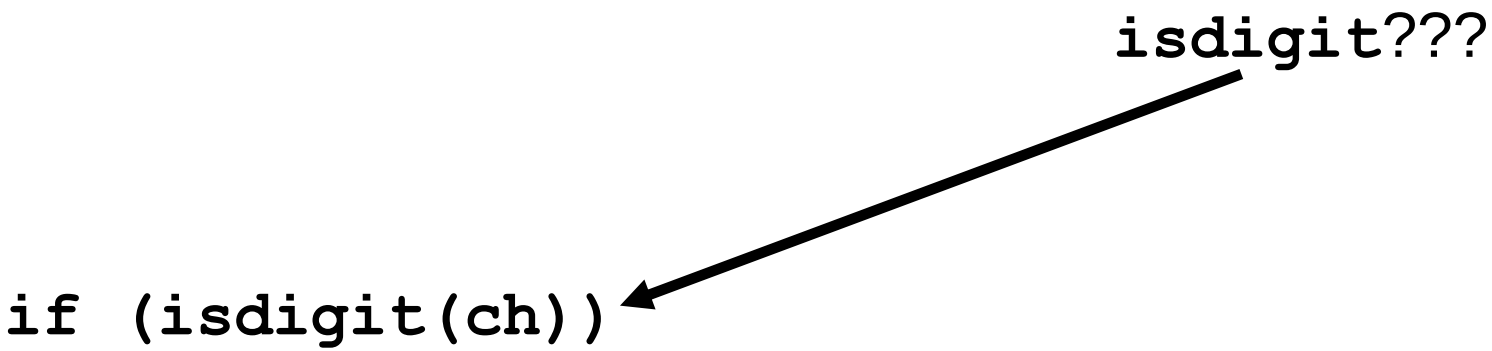
Reading a Number Only If It Is a Number

A typical situation for lookahead is to look for numbers before reading them so that a failed read won't happen:

```
char ch;
in_file.get(ch);
if (isdigit(ch)) // Is this a number?
{
   // Put the digit back so that it will
   // be part of the number we read
   in_file.unget();

   // Read integer starting with ch
   int n;
   data >> n;
}
```

Reading a Number Only If It Is a Number



Character Functions in `<cctype>`

The **`isdigit`** function is one of several functions that deal with characters.

`#include <cctype>` is required.

Character Functions in <cctype>

Table 1 Character Predicate Functions in <cctype>

Function	Accepted Characters
isdigit	0 ... 9
isalpha	a ... z, A ... Z
islower	a ... z
isupper	A ... Z
isalnum	a ... z, A ... Z, 0 ... 0
isspace	White space (space, tab, newline, and the rarely used carriage return, form feed, and vertical tab)

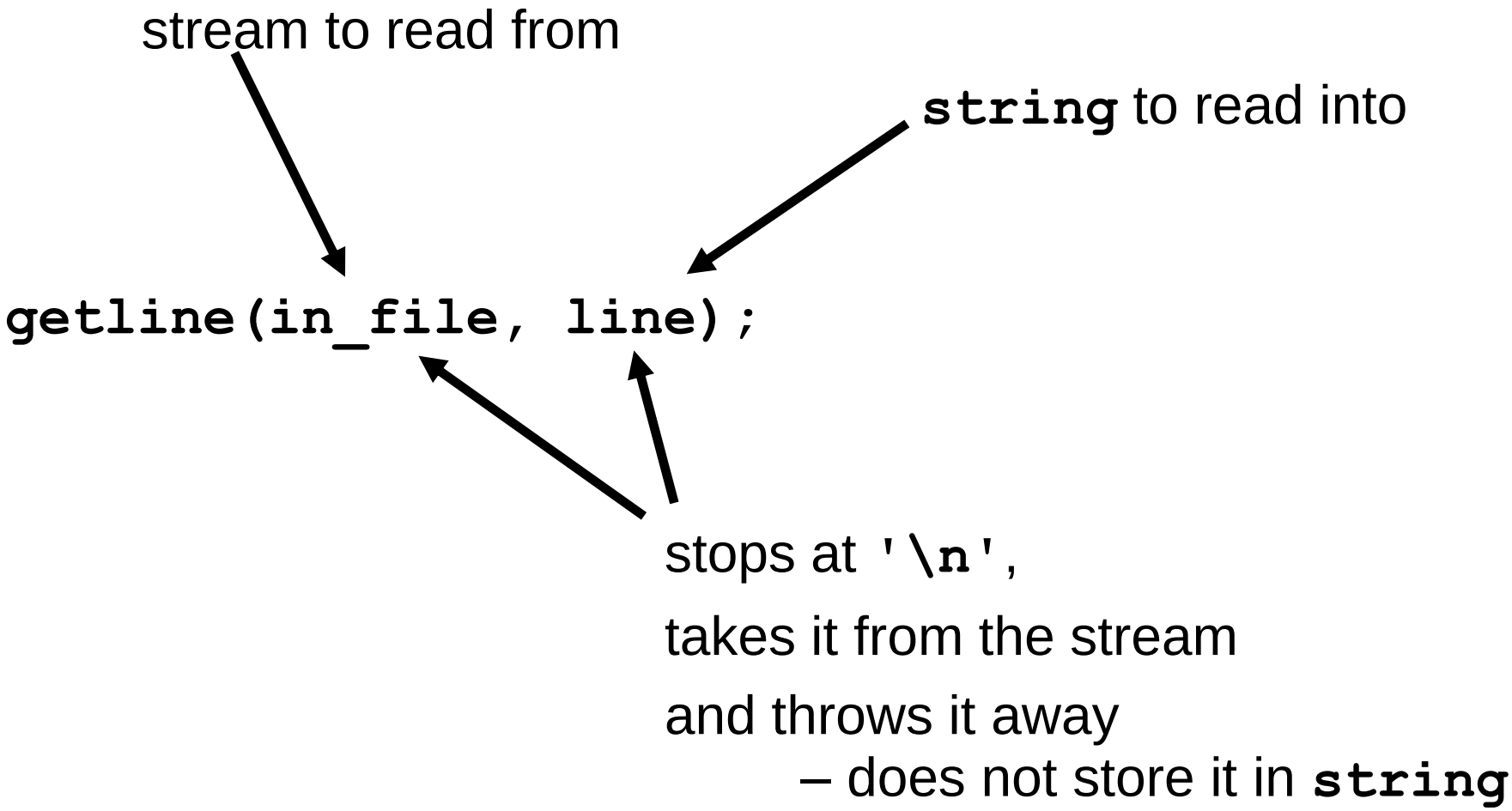
Reading A Whole Line: `getline`

The function (it's not a method):

`getline()`

is used to read a whole line
up to the next newline character.

Reading A Whole Line: `getline`



Reading A Whole Line: `getline`

Until the next newline character,
all whitespace and all other characters
are taken and stored into the string
– except the newline character – which is just “thrown away”

The *only* type that can be read into is the **`string`** type.

```
string line;
getline(in_file, line); // stops at '\n'
```

Reading A Whole Line: `getline`

The **`getline`** function, like the others we've seen, returns the "not failed" condition.

To process a whole file line by line:

```
string line;
while( getline(in_file, line))
{
   // Process line
}
```

Processing a File Line by Line

Reading one line and then processing that line,
taking it apart to get the individual pieces of data from it,
is a typical way of working with files.

Processing a File Line by Line

Here is a top secret online CIA file:

http://www.cia.gov/library/publications/the-world-factbook/

Don't tell anyone, but it looks like this:

```
China 1330044605
India 1147995898
United States 303824646
...
```

Each line has: country name, its population, a newline character.

(And there is some whitespace in there too.)

Processing a File Line by Line

After having copied the secret contents from the website into a text file we will read this file line by line:

```
// we've opened the file
// but we can't tell you it's name or...

string line;
while( getline(in_file, line))
{
   // Process line
}
```

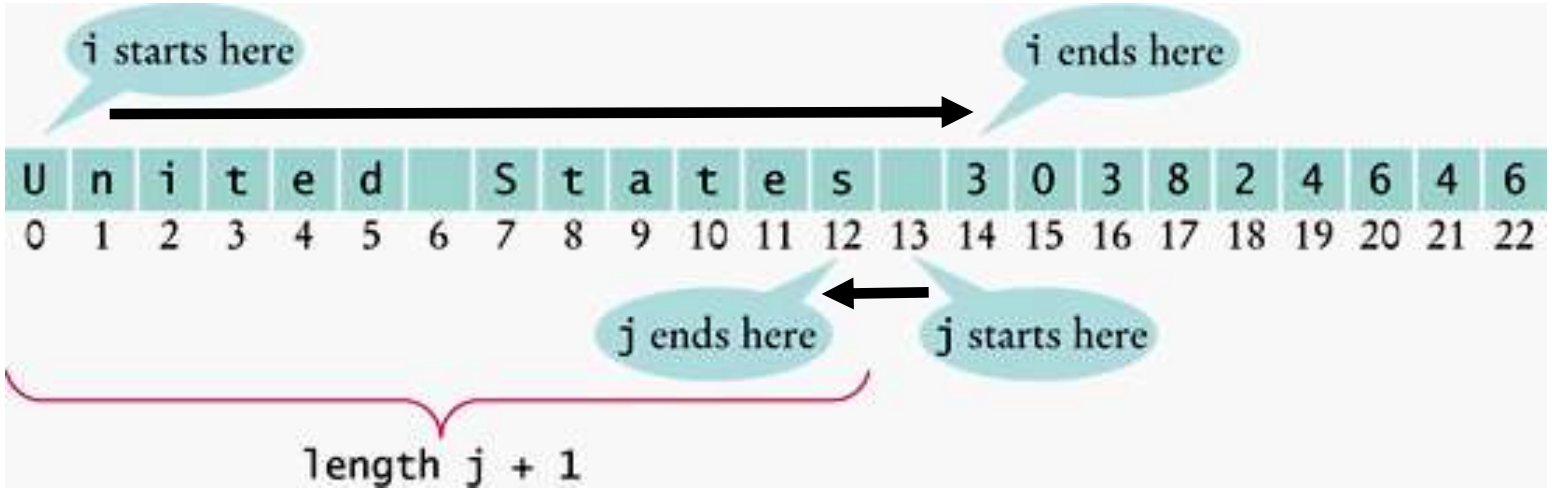
Processing a File Line by Line

To extract the data from the `line` variable, you need to find out where the name ends and the number starts.

Processing a File Line by Line

```
// Locate the first digit of population
int i = 0;
while (!isdigit(line[i])) { i++; }
```

```
// Find end of country name
int j = i - 1;
while (isspace(line[j])) { j--; }
```



Processing a File Line by Line

All clear – make the extraction at 09:01 hours.

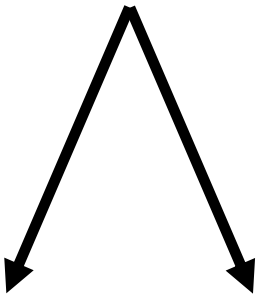
(that's CIA-speak for: use **`string`** methods to "extract" the country name and population values)

(we guess "at 09:01" hours means now)

```
string country_name = line.substr(0, j + 1);
string population = line.substr(i);
```

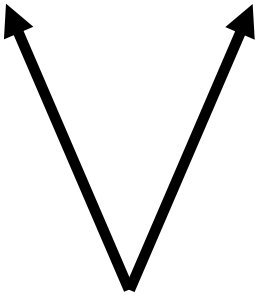
Processing a File Line by Line

Has the CIA made a blunder?
Aren't these being stored as **`string`**s?



(Is that a security risk?)

```
string country_name = line.substr(0, j + 1);
string population = line.substr(i);
```



Writing Text Output

Recall that you know how to output to a file.

(OK)

Writing Text Output

You use the operator **>>** to send **strings** and numbers to an output stream:

```
ofstream out_file;
out_file.open("output.txt");
if (in_file.fail()) { return 0; }
out_file << name << " " << value << endl;
```

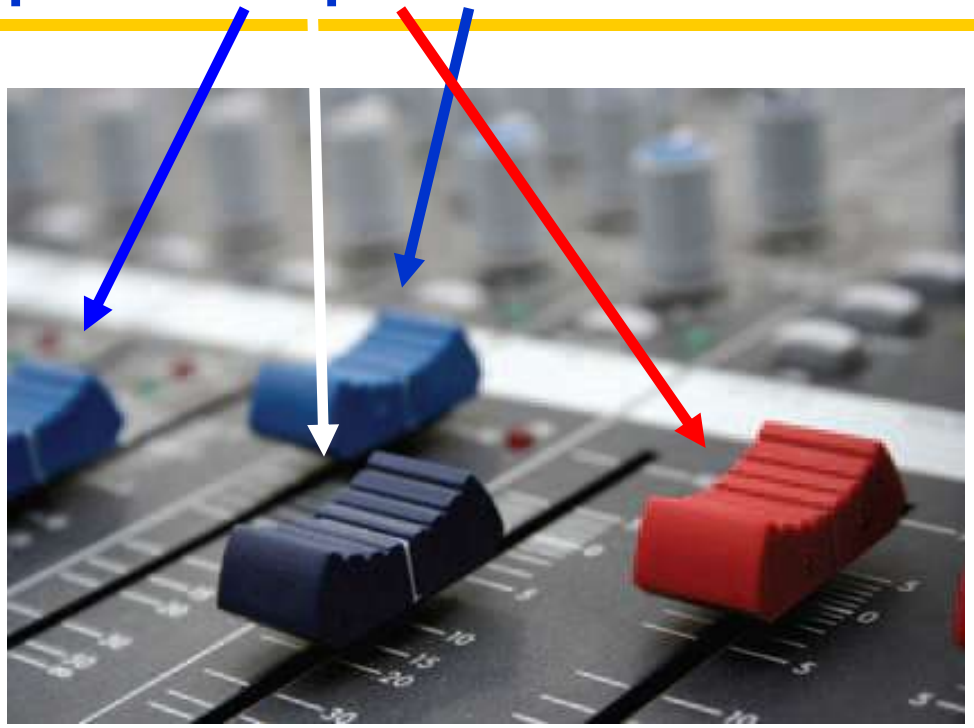
Writing Text Output

But what about a single character?

The **`put`** method:

```
out_file.put(ch);
```

Formatting Output – Manipulators



Formatting Output – Manipulators

I don't hear anything.

Not now, but when I ***manipulate*** that red slider,
you'd better have your earplugs ready!

OK, there' in. Go ahead…

Formatting Output – Manipulators

I don't hear anything.

Not now, but when I ***manipulate*** that red slider, you'd better have your earplugs ready!

OK, there' in. Go ahead

Nice. What's the name of this tune? *"Sliders On A Slide"*

We're gonna use it in a pretty aggressive, *manipulative* ad.

Formatting Output – Manipulators

To control how the output is *formatted*,
you use *stream manipulators*.

A *manipulator* is an object that:

- is sent to a stream using the << operator.
- affects the behavior of the stream.

Formatting Output – Manipulators

You already know one:

`endl`

Yes, **`endl`** is a manipulator.

Formatting Output – Manipulators

When you send the **`endl`** manipulator to the screen:

```
cout << endl;
```

`endl` causes the cursor
to go to the beginning of the next line.

Moving the cursor is definitely an *affect*ation!

Wait…

Formatting Output – Manipulators

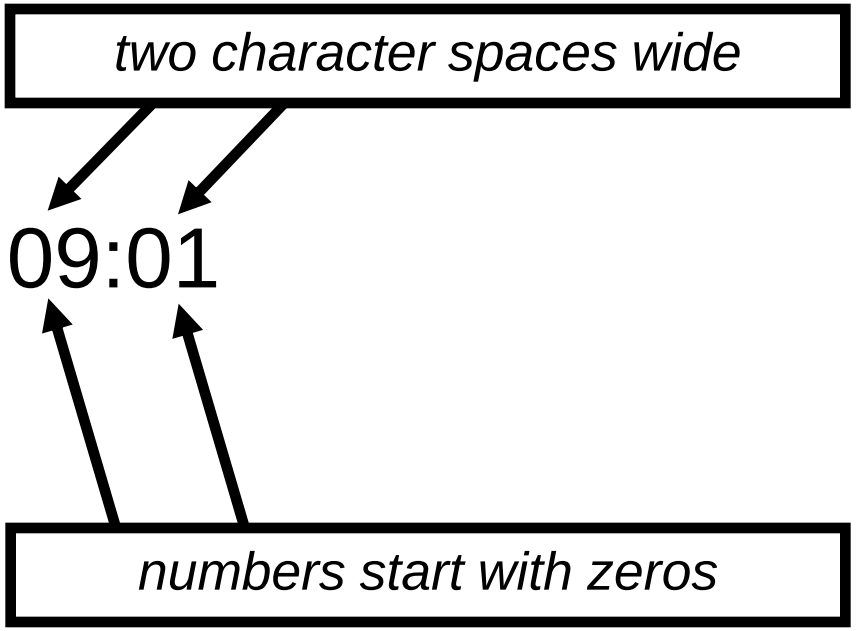
It's about *sending the manipulator*:

```
output_stream << manipulator
```

Some other output manipulators:

Formatting Output – Manipulators

Recall that CIA time display?



Formatting Output – Manipulators

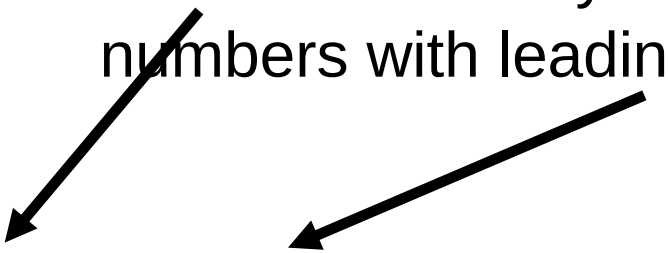
Recall that CIA time display?

09:01

very
military!

Formatting Output – Manipulators

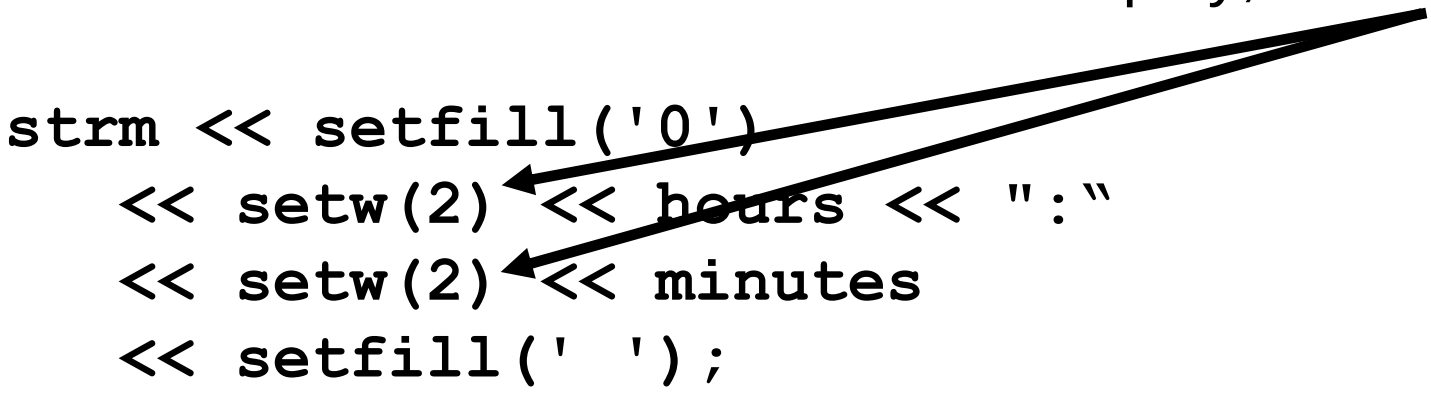
Use **`setfill`** when you need to pad numbers with leading zeroes.



```
strm << setfill('0')
   << setw(2) << hours << ":"
   << setw(2) << minutes
   << setfill(' ');
```

Formatting Output – Manipulators

Use **`setfill`** when you need to pad numbers with leading zeroes.
To set the width in which to display, use **`setw`**.



```
strm << setfill('0')
   << setw(2) << hours << ":"
   << setw(2) << minutes
   << setfill(' ');
```

Formatting Output – Manipulators

Use **`setfill`** when you need to pad numbers with leading zeroes.
To set the width in which to display, use **`setw`**.

```
strm << setfill('0')
   << setw(2) << hours << ":"
   << setw(2) << minutes
   << setfill(' ');
```

That last **`setfill`** re-sets the fill back to the default space character.

Formatting Output – Manipulators

Table 2 Stream Manipulators

Manipulator	Purpose	Example	Output

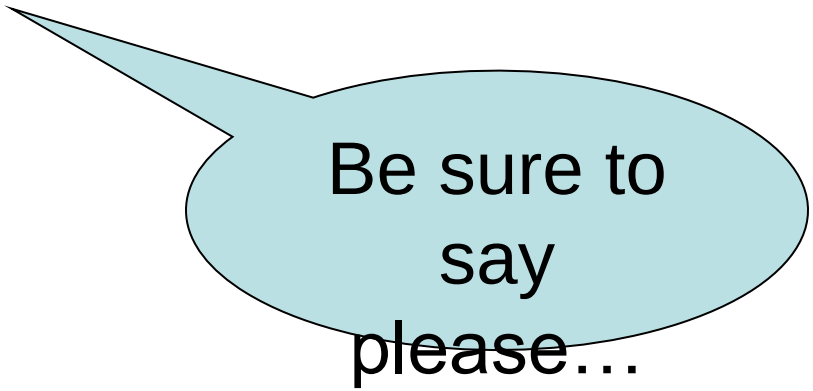
`setw`	Sets the field width of the next item only.	`strm << setw(6) << 123 << endl << 123 << endl << setw(6) << 12345678;`	`  123`
			`123`
			`12345678`

`setfill`	Sets the fill character for padding a field. (The default character is a space.)	`strm << setfill('0') << setw(6) << 123;`	`000123`
`left`	Selects left alignment.	`strm << left << setw(6) << 123;`	`123`
`right`	Selects right alignment (default).	`strm << right << setw(6) << 123;`	`  123`
`fixed`	Selects fixed format for floating-point numbers.	`double x = 123.4567;` `strm << x << endl << fixed << x;`	`123.457` `123.456700`
`setprecision`	Sets the number of significant digits for general format, the number of digits after the decimal point for fixed format.	`double x = 123.4567;` `strm << fixed << x << endl << setprecision(2) << x;`	`123.456700` `123.46`

Formatting Output – A Test

See if you can write the output statements to accomplish the following outputs to the stream variable, **`strm`**, which is already opened.

You can ask the person with the clicker or pointer, or whoever is in charge of this presentation, to go back to the slide with the table of manipulators.



Formatting Output – A Test

Produce this output:

```
12.3457 1.23457e+08
```

The code:

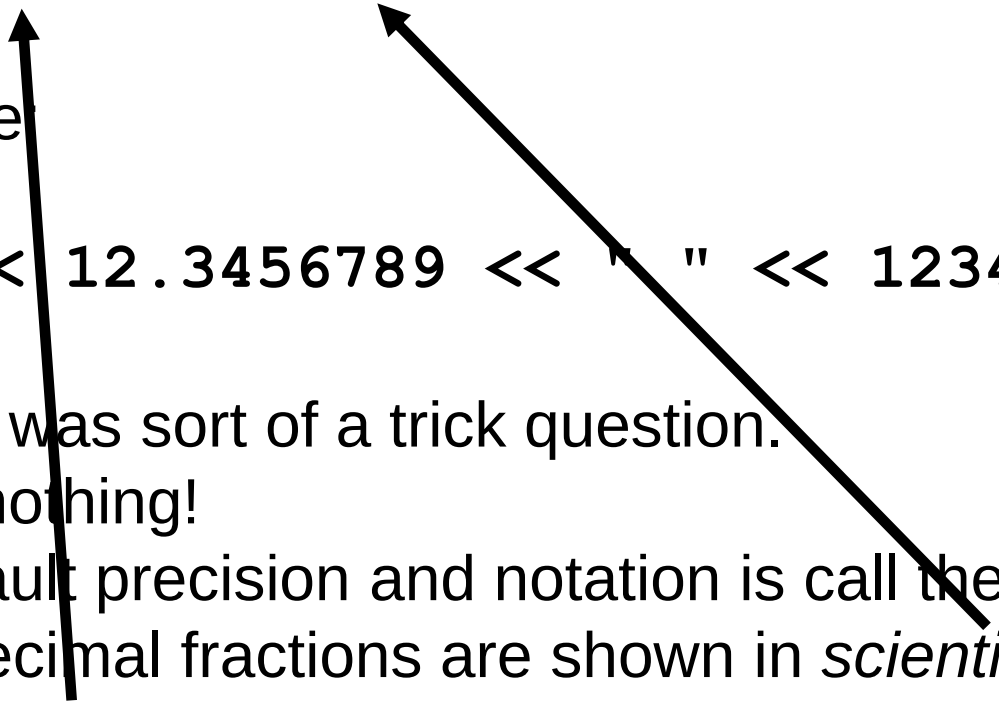
```
strm << 12.3456789 << " " << 123456789.0;
```

(will this be on the test?)

Formatting Output – A Test

Produce this output:

```
12.3457 1.23457e+08
```



The code:

```
strm << 12.3456789 << " " << 123456789.0;
```

OK, that was sort of a trick question.
We did nothing!
The default precision and notation is call the *general* format.
Some decimal fractions are shown in *scientific notation*, and the *default* precision to round off to is used.

Formatting Output – A Test

The **`fixed`** manipulator was in the table, but **`scientific`** wasn't shown (it also is a manipulator).

The question was about the *default* settings.

The lesson to be learned is:

Always take charge of formatting your output

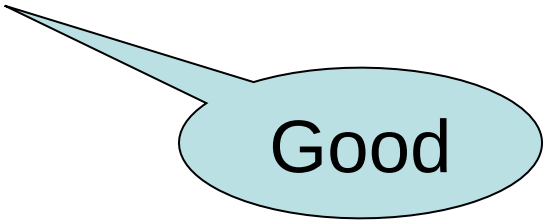
Formatting Output – A Test

Produce this output:

```
12.345679 123456789.000000
```

The code:

```
strm << setprecision(6) << fixed
   << 12.3456789 << " " << 123456789.0;
```



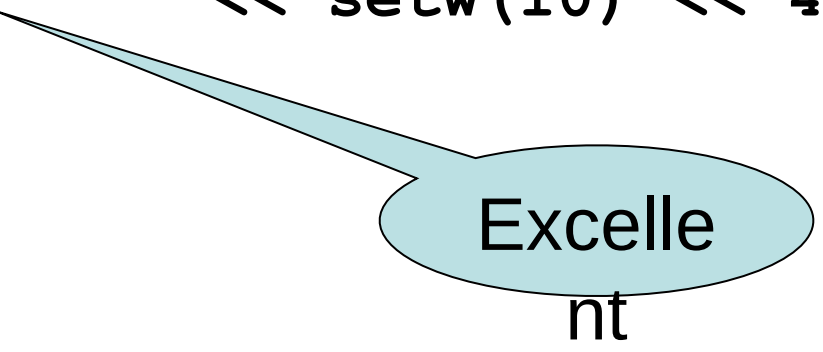
Formatting Output – A Test

Produce this output (note that the column width is 10):

```
       123
      4567
```

The code:

```
strm << setw(10) << 123 << endl
   << setw(10) << 4567;
```



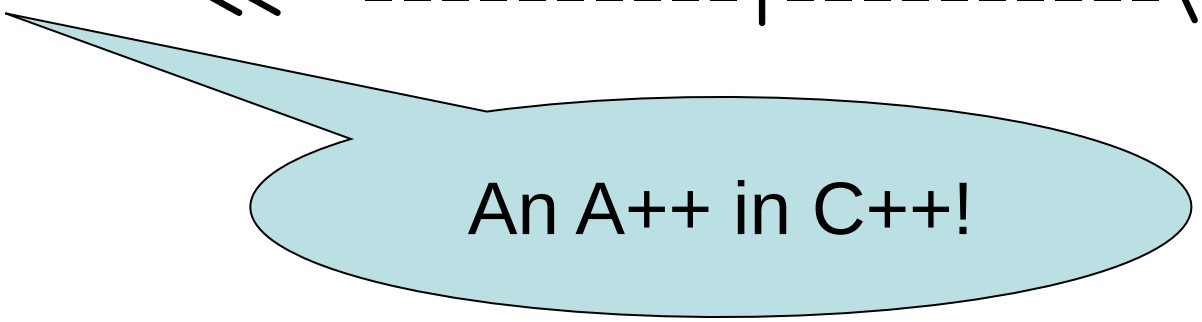
Formatting Output – A Test

Produce this output (note there are 10 dashes on each side):

```
----------|----------
Count:              177
----------|----------
```

The code:

```
strm << "----------|----------\n"
   << left << setw(10) << "Count:"
   << right << setw(11) << 177
   << "----------|----------\n";
```



Welcome to CIA headquarters

The time is now:

09:01

It's stringstream time

End Chapter Eight: Streams, Part I