Chapter Eight: Streams

Slides by Evan Gallagher

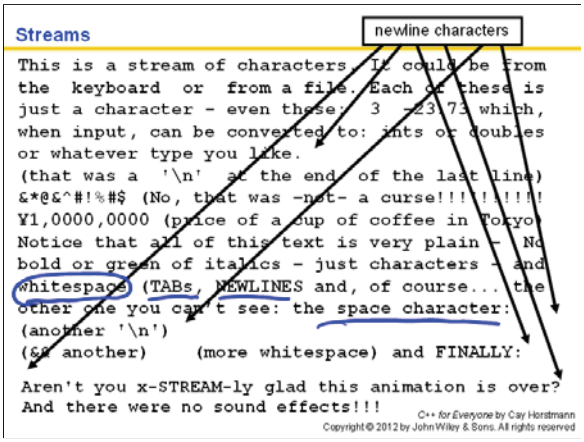
Chapter Goals

- To be able to read and write files
- To convert between strings and numbers using string streams
- To process command line arguments
- To understand the concepts of sequential and random access

Reading and Writing Files (8.1)

- 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).

OR

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

Pictures are 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*.

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.)

first we'll need to open the stream

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`:

type name ;

```
ifstream in_file;
```

declare the variable stream we wish to input data from

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.

indicates to humans we have a data file

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

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

variable we declared earlier

member function

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:

to get name of input file from user:

```
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();
```

if want to close manually

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:

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

See, I told you so!

cin?
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; }
```

good idea to make sure the file exists

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

Include this header when you use file streams.

```
#include <fstream>
```

Use `ifstream` for input, `ofstream` for output, `fstream` for both input and output.

```
ifstream in_file;
in_file.open(filename.c_str());
in_file >> name >> value;
```

Call `c_str` if the file name is a C++ string.

Use `\\` for each backslash in a string literal.

Use the same operations as with `cin`.

```
ofstream out_file;
out_file.open("c:\\output.txt");
out_file << name << " " << value << endl;
```

Use the same operations as with `cout`.

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.

must use & w/ stream name

(Did you notice that "ALWAYS"?)

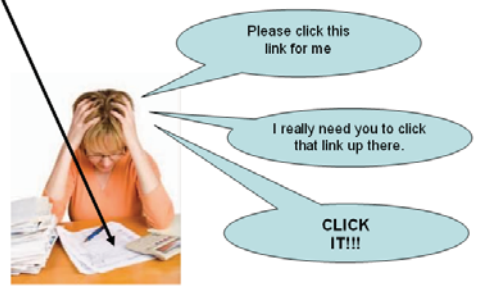
A Programming Problem

Why can I never find Ezmereldza?

A Programming Problem

A Programming Problem

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



A Programming Problem: BABYNAMES

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
```

rank name # %

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

And something else,

"ALL'S GOOD THAT ENDS WELL?"
(No.)

"ALL GOOD BOYS DO FINE?"
(No, that was from a music lesson.)

"ALL MEANS NECESSARY?"
(What?)

Aha! It was:

"ALWAYS pass streams by reference"

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.

(percents)

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;
}
```

must w/streams

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 (8.2)

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);
```

cin.get(ch); reads from keyboard

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
}
```

Not Wanting What You `.get()`

That which was got
can be ungot!

```
in_file.unget();
```

Thank you.

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;
}
```

ifstream variable declared prior

Reading a Number Only If It Is a Number

isdigit???

```
if (isdigit(ch))
```


Character Functions in `<cctype>`

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

`#include <cctype>` is required.

Character Functions in `<cctype>` STOPPED

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)