

Reminders

Submissions:

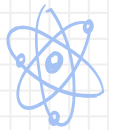
- Homework 5: Sat 10/12 @ 6pm
- Homework 6: Sat 10/19 at 6 pm



Readings:

- Ch. 8 Streams – for most of this week
- Starting Ch. 9 Classes and Objects on Friday, cont. next week

Office Hours Locations – changes – Check the Calendar!



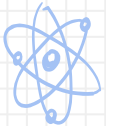


James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

Chapter Eight: Streams

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.
 - You need to know how to read/write disk files to work with large amounts of data that are common in business, administrative, graphics, audio, and science/math programs



$$E=mc^2$$



Reading and Writing Disk Files

You can also read and write files stored on your hard disk:

- plain text files
- binary information (a binary file)
 - Such as images or audio recording

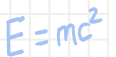
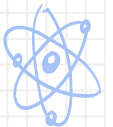
To read/write files, you use *variables* of the stream types:

ifstream for input from plain text files.

ofstream for output to plain text files.

fstream for input and output from binary files.

You must `#include <fstream>`



Opening a Stream

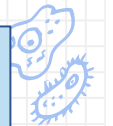
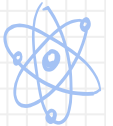
- To read anything from a file stream, you need to *open* the stream. (The same for writing.)
- *Opening a stream* means associating your stream variable with the disk file.
- The first step in opening a file is having the stream variable ready.

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.



Code for Opening Streams

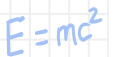
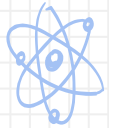
```
ifstream in_file;  
in_file.open("input.txt"); //filename is input.txt
```

An alternative shorthand syntax combines the 2 statements:

```
ifstream in_file("input.txt");
```

As your program runs and tries to find this file, it **WILL ONLY LOOK IN THE DIRECTORY (FOLDER) IT IS LOCATED IN!**

That is a common source of errors. If the desired file is not in the executing program's folder, the full file path must be specified.



File Path Names

File names can contain directory path information, such as:

UNIX

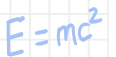
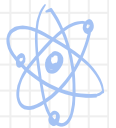
```
in_file.open("~/nicework/input.dat");
```

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 Windows), you must **supply each backslash twice** to avoid having unintended *escape characters* in the string.

\\ becomes a single \ when processed by the compiler.

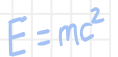
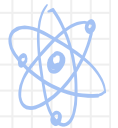


Closing a Stream

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();
```



Reading from a Stream

You already know how to read and write using files.

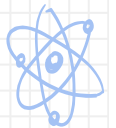
Yes you do:

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

See, I told you so!

cin? in_file?

No difference when it comes to reading using >>.



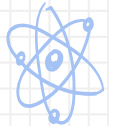
Reading from a Stream

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!

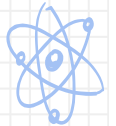


Reading from a Stream

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
}
```

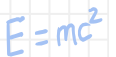
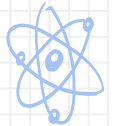
x-STREAM-ly STREAM-lined
--- Cool!



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);  
// Check for failure after opening  
if (in_file.fail())  
{  
    return 0;  
}  
  
or  
  
if (!in_file.is_open())  
{  
    return 0;  
}
```

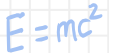
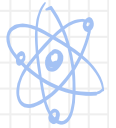


Reading A Whole Line: `getline`

The function **`getline()`** reads a whole line up to the next `'\n'`, into a C++ string.

The `'\n'` is then deleted, and NOT saved into the string.

```
string line;  
ifstream in_file("myfile.txt");  
  
getline(in_file, line);
```

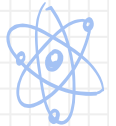


Reading A Whole Line in a Loop: `getline`

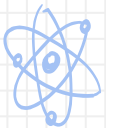
The **`getline`** function, like the others we've seen, returns the “not failed” condition.

To process a whole file line by line:

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



Reading Words and Characters



$$E=mc^2$$



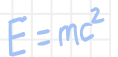
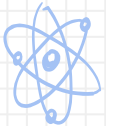
What really happens when reading a **string**?

```
string word;
```

```
in_file >> word;
```

1. Any whitespace is skipped (whitespace is: '**\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 Words and Characters



You can read a single character, including whitespace, using `get()` :

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

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

```
//reads entire file, char by char  
while (in_file.get(ch))  
{  
    // Process the character ch  
}
```

Reading a Number Only If It Is a Number

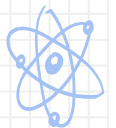
You can look at a character after reading it and then put it back.

This is called *one-character lookahead*. A typical usage: check for numbers before reading them so that a failed read won't happen:

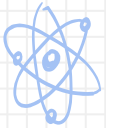
```
char ch;
int n=0; //for reading an entire int
in_file.get(ch);

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();

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



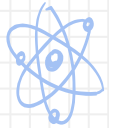
Functions in `<cctype>` (Handy for Lookahead)



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

Topic 3

1. Reading and writing text files
2. Reading text input
3. Writing text output
4. Parsing and formatting strings
5. Command line arguments
6. Random access and binary files



$$E=mc^2$$

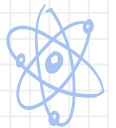


Writing to a Stream

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 the same operations
as with `cin`.

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

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

Use the same operations
as with `cout`.

