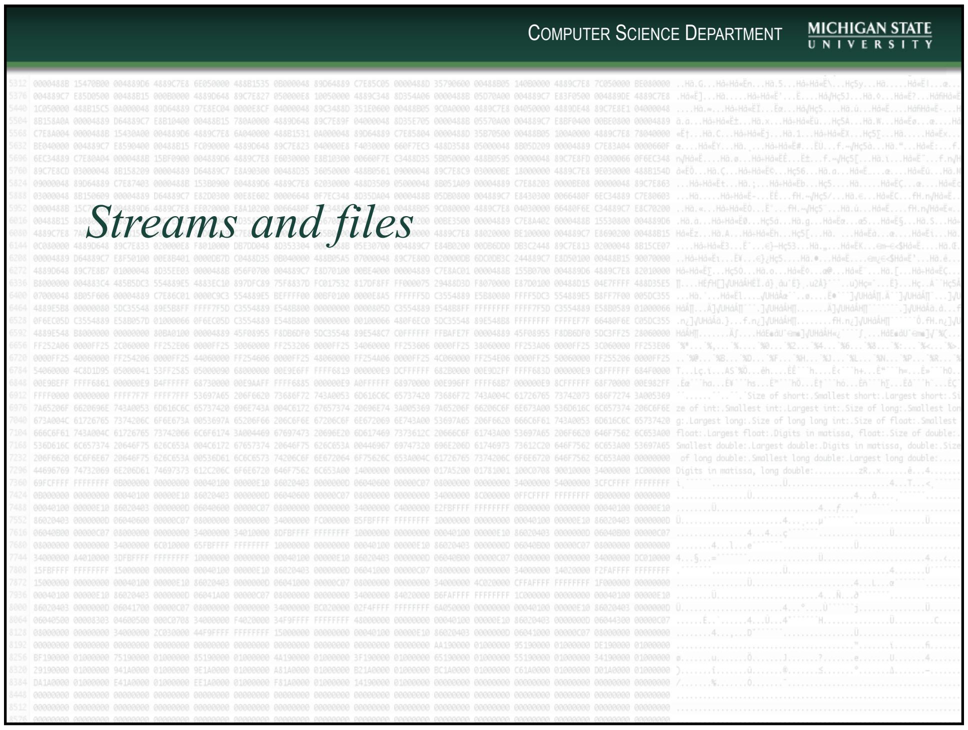
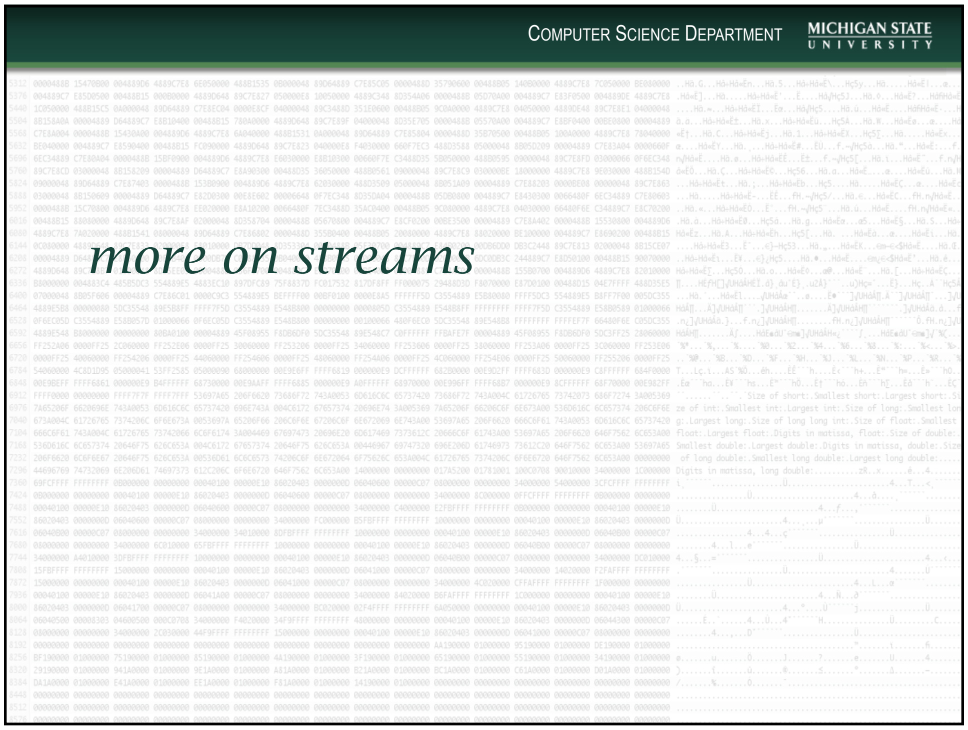


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

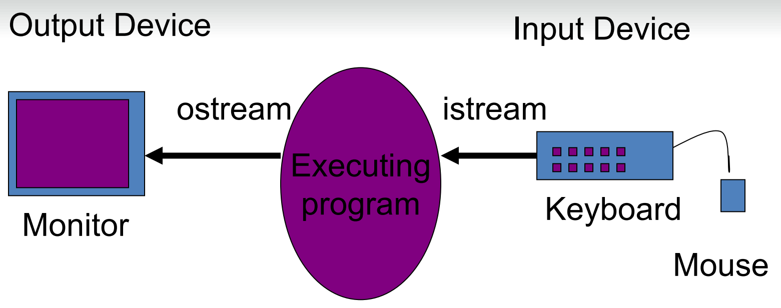
Streams and files



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

more on streams

IO Streams

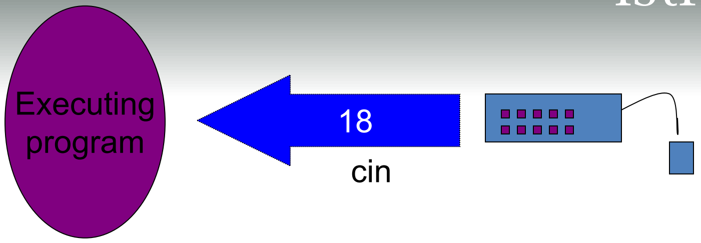


Streams are objects with names such as cin, cout, cerr.

Buffer

Each stream has an associated buffer, part of the stream object, that data is pulled/pushed from

istream

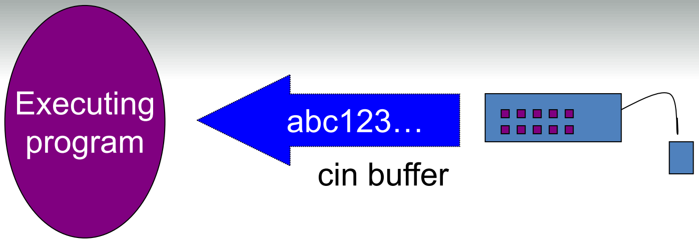


The >> (extraction) operator is an overloaded operator that takes values out of an input stream (`cin`), and stores them as the type indicated by the target variable.

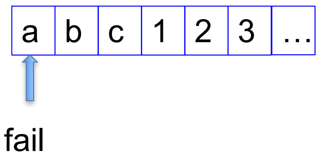
```
int my_var; cin >> my_var;
```

The '1' is read, and then the '8' is read.
Then the two characters are converted to the integer 18 which is stored in variable `my_var`.

When cin goes bad

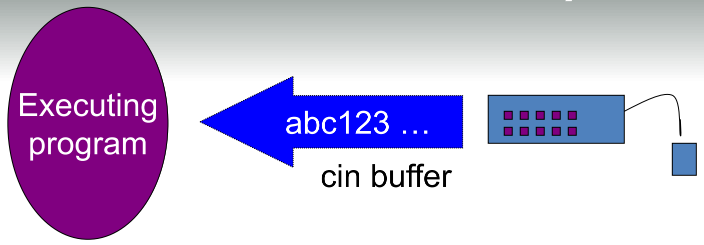


```
int my_int=10;
cin >> my_int; //
```

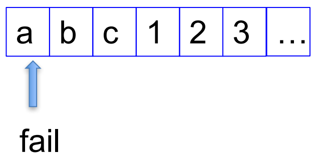


typed operator, can only read `int` type stuff, fails at the 'a'

fail is fail, you must fix



```
int my_int=10;
char my_char='a';
cin >> my_int; // fail
cin >> my_char;// fail
```



cin stays in failed state until you clean it up. All subsequent reads fail until that happens

Status Functions

- Useful boolean member functions:

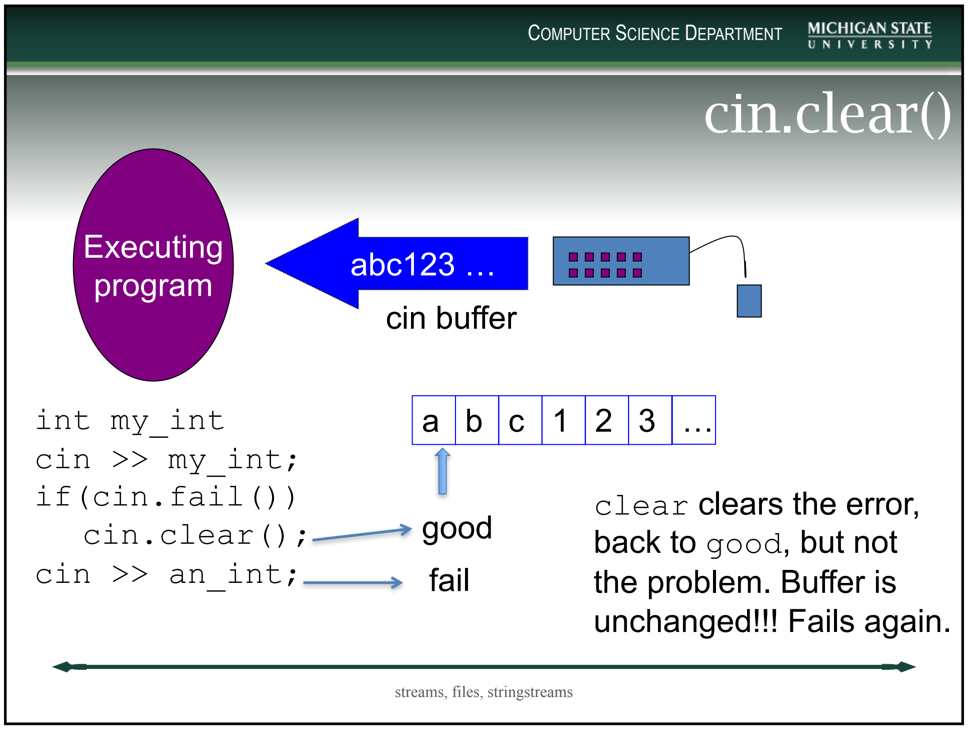
  `cin.good()`: all is well in the istream

  `cin.bad()`: something is wrong with istream

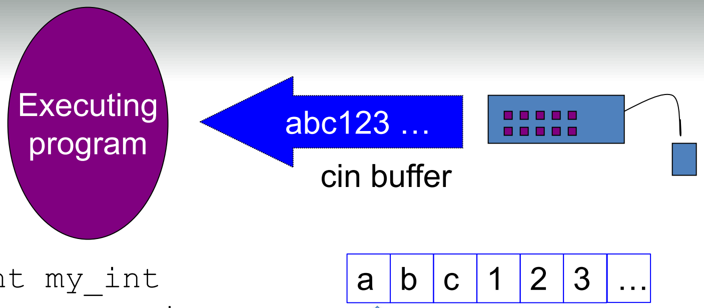
  `cin.fail()`: last op could not be completed

  `cin.eof()`: last op encountered end-of-file

- Useful with the `assert()` function: e.g. `assert(cin.good())`



cin.clear()

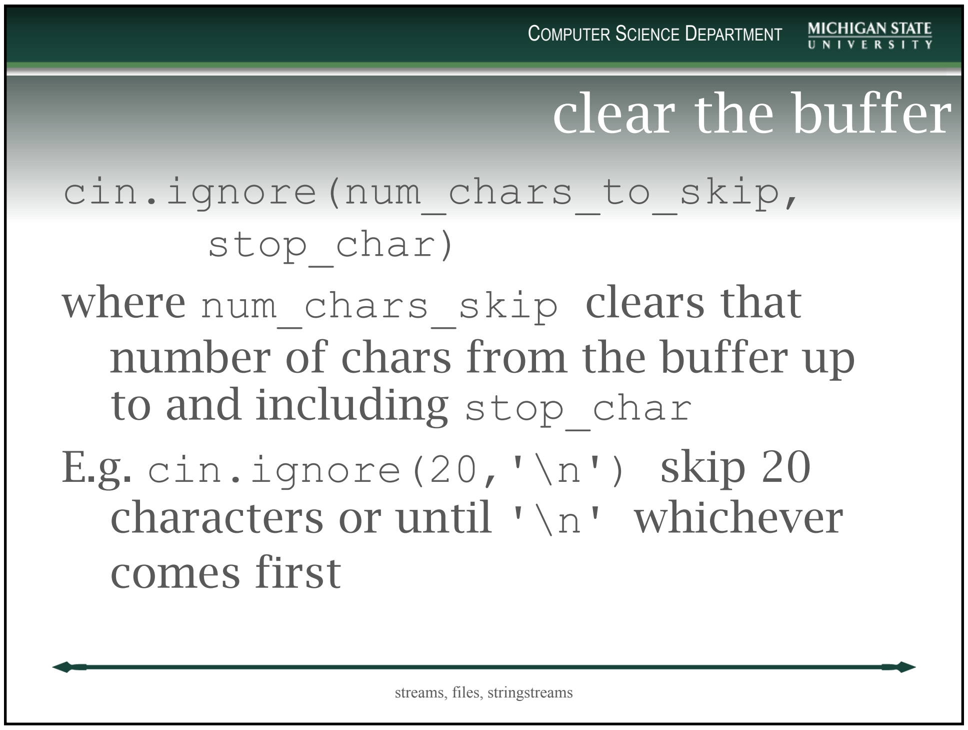


```
int my_int
cin >> my_int;
if(cin.fail())
   cin.clear();
cin >> an_int;
```

good

fail

`clear` clears the error, back to `good`, but not the problem. Buffer is unchanged!!! Fails again.



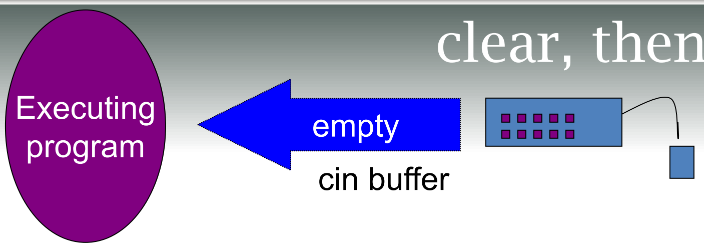
clear the buffer

```
cin.ignore(num_chars_to_skip,
           stop_char)
```

where `num_chars_skip` clears that number of chars from the buffer up to and including `stop_char`

E.g. `cin.ignore(20,'\n')` skip 20 characters or until `'\n'` whichever comes first

clear, then ignore



empty buffer

```
int my_int, an_int;
cin >> my_int;
if(cin.fail()){
   cin.clear();
  cin.ignore(1000, '\n');
}
// reprompt, try again
```

good

`ignore` empties the buffer. Now you can try again (reprompt for example).

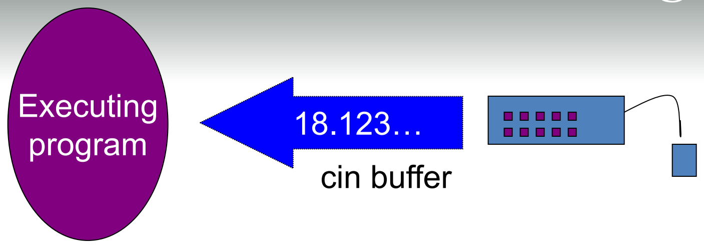
more on ignore

- takes a default count as 1
  - any number works
  - `numeric_limits<streamsize>::max()` (requires `#include<limits>`) means as many as necessary to hit the stop char.
- takes a default stop as the `eof` char

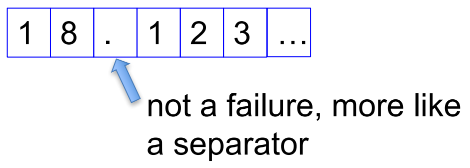
more complicated for a float

The situation is more complicated for numbers. For example, try reading a float into an integer.

When cin goes bad

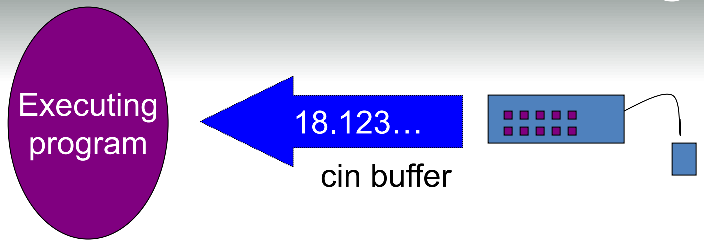


```
int my_int;
cin >> my_int; //18
```

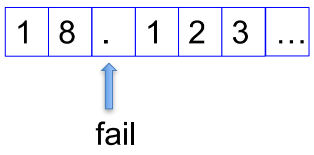


typed operator, can only read `int` type stuff, stops (***not fails***) at the '.'

When cin goes bad

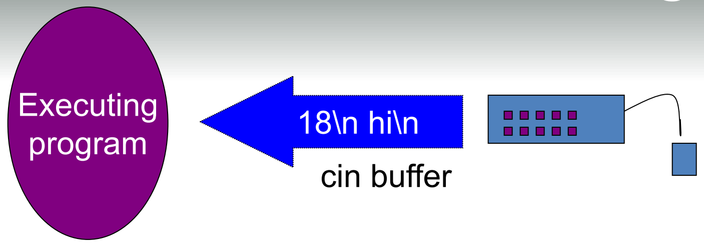


```
int my_int, an_int;
cin >> my_int; //18
cin >> an_int;
```

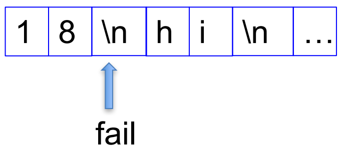


next read is a failure (chokes on the '.')

When cin goes bad



```
int my_int, an_int;
cin >> my_int; //18
cin >> an_int;
```



next read is a failure (chokes on the '.')

Better to treat as a string and cast

We'll see it is easier to treat this as a string and try to cast it.

cin returns ?

`cin >> some_var` returns:

- `cin` if things go well
- `false` if you hit `eof`
- `false` if the stream is in a `fail` or `bad` mode

Thus you can:

```
while(cin >> some_var)
…
```

White space

- White space: blanks, tabs, and returns
- By default, the >> operator skips *leading* white space
- `int X, Y, Z;`
- `cin >> X >> Y >> Z;`
- Input: `3      4                    5`

`X` is 3, `Y` is 4, `Z` is 5

Controlling White Space

- Turn off skipping white space:
  - `cin >>noskipws`
- Turn skipping white space back on:
  - `cin >> skipws`
- ALTERNATIVE: use an input function which does not skip white space: `cin.get(ch)` reads *exactly one character* no matter the character

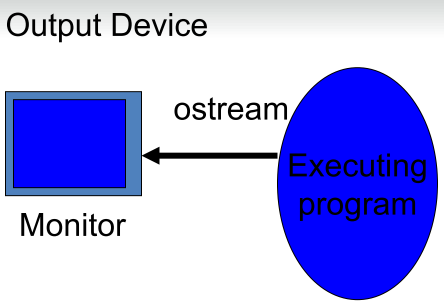
Single Character

- To read a single character, not skipping:
  - `cin.get(ch)`
- To put that character back into the buffer
  - `cin.putback(ch)`
- To peek without removing it:
  - `cin.peek()`

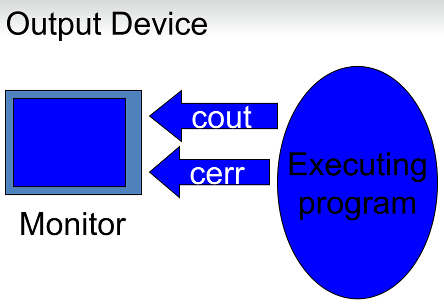
Output functions

- Single character function: `cout.put(ch)` puts a single character into the `ostream`.

ostream

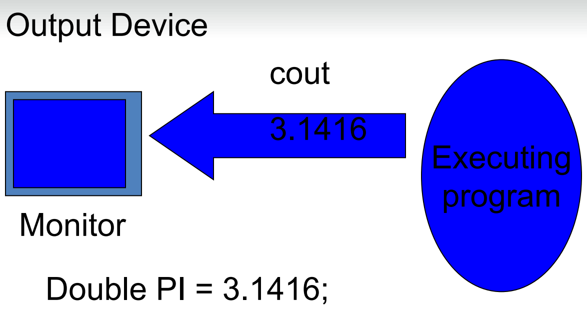


cout and cerr



Cout and cerr are particular instances of an ostream.

<< operator



Double PI = 3.1416;
cout << PI;

The double PI is converted to the six characters for output: '3' '.' '1' '4' '1' '6'

Output formatting

- We have seen many of the format codes (descriptions are on pg. 757 of the text): `skipws, left, right, dec, oct, hex, uppercase, scientific, fixed` but look at there are others
- `in_stream.setf(ios::skipws)` is an alternate way to set some of these. Book uses the former.

Buffer

- Output characters are stored into a buffer before being output, i.e. gather up a bunch of characters before sending them to the screen. This can be a problem for debugging: output may be in the buffer leading you to believe that an error occurred before the output statement when it actually occurred afterward.

Buffering & Debugging

```
double f(double X)
{
  cout << “entering f”;
  …
  cout << “exiting f”;
  return Z;
}
```

Flush buffer

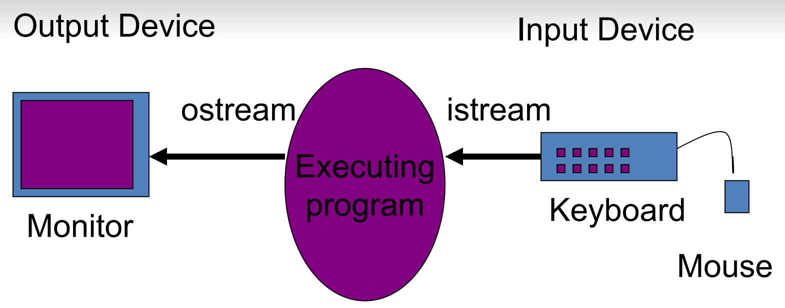
```
double f(double X)
{
  cout << “entering f” << endl;
  …
  cout << “exiting f” << flush;
  return Z;
}
```

Files and Streams

Files

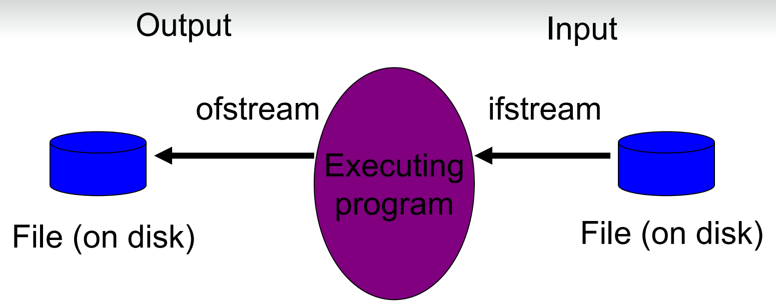
- Files are collections of data and are stored in nonvolatile memory, e.g. secondary storage such as disk.
- *Text Files* store characters such as ASCII, e.g. source code.
- *Binary Files* contain non-ASCII characters, e.g. compiled code.
- Humans can read text files.

Stream Review



Streams are objects with names such as cin, cout, cerr.

File Streams



Previous streams were objects with names such as cin, cout, cerr.
Now we add streams which are files. We can name them.

just another stream

Because we are working with the stream object, the pipe, we do not have to worry about particular devices (that is the software's problem).

Result is that many of the operations we used with `cin` and `cout` work with files.

to work with a file

- required `#include<fstream>`. This provides two kinds:
  - `ifstream` (input files)
  - `ofstream` (output files)
- Can establish a connection by:
  - declare with the name (as a string) to open automatically
  - `.open(string)` method to establish connection between a program and a file.

example

```
#include<fstream>
// automatically open in_file
ifstream in_file("my_file.txt");
ofstream out_file;
string file_name;
cin >> file_name;
// out_file created and now opened
out_file.open(file_name);
```

Where is that file?

When you open a file with a simple name, like "file.txt", the assumption is that the file is located in the same director/folder as the executing program.

If not there, you have to give a fully qualified path.

fully qualified path

Sadly, this can depend on the underlying operating system:

- C:\Documents\My Folder\file.txt
- /usr/local/bill/file.txt

Know that it is true and assume the file is in the correct place

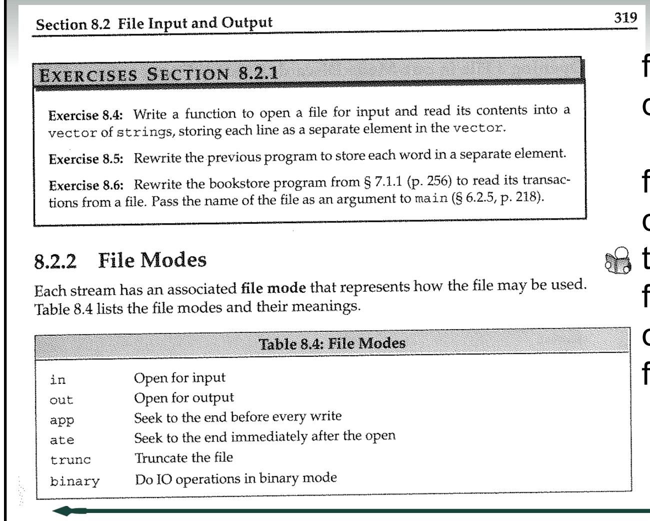
standard operations

- `>>, <<` input and output operations
- `getline(instream, str)` reads a line into a string
- `eof()` true if end-of-file mark was read
- `get()` or `put()`
- etc.

unique operations

- `.open()` method
- `is_open()` true if file was successfully opened
- `close()` method terminates the connection between a program and a file
  - flushes the buffer

other file modes



Section 8.2 File Input and Output 319

EXERCISES SECTION 8.2.1

Exercise 8.4: Write a function to open a file for input and read its contents into a `vector` of `strings`, storing each line as a separate element in the `vector`.

Exercise 8.5: Rewrite the previous program to store each word in a separate element.

Exercise 8.6: Rewrite the bookstore program from § 7.1.1 (p. 256) to read its transactions from a file. Pass the name of the file as an argument to `main` (§ 6.2.5, p. 218).

8.2.2 File Modes

Each stream has an associated **file mode** that represents how the file may be used. Table 8.4 lists the file modes and their meanings.

Table 8.4: File Modes

`in`	Open for input
`out`	Open for output
`app`	Seek to the end before every write
`ate`	Seek to the end immediately after the open
`trunc`	Truncate the file
`binary`	Do IO operations in binary mode

for input files, the default is input

for output files, the default is open and trunc. Thus out files, by default, wipe out any info in the file being written to

specify yourself

If you declare a file as an `fstream`, you get to decide what aspects you want.

```
fstream in_out_file ("file.txt",
     fstream::in | fstream::out | fstream::ate);
```

vertical bars are bitwise or operator, look it up. We combine all aspects this way

This a file one can read from and write to, and writing occurs at the end of the file.