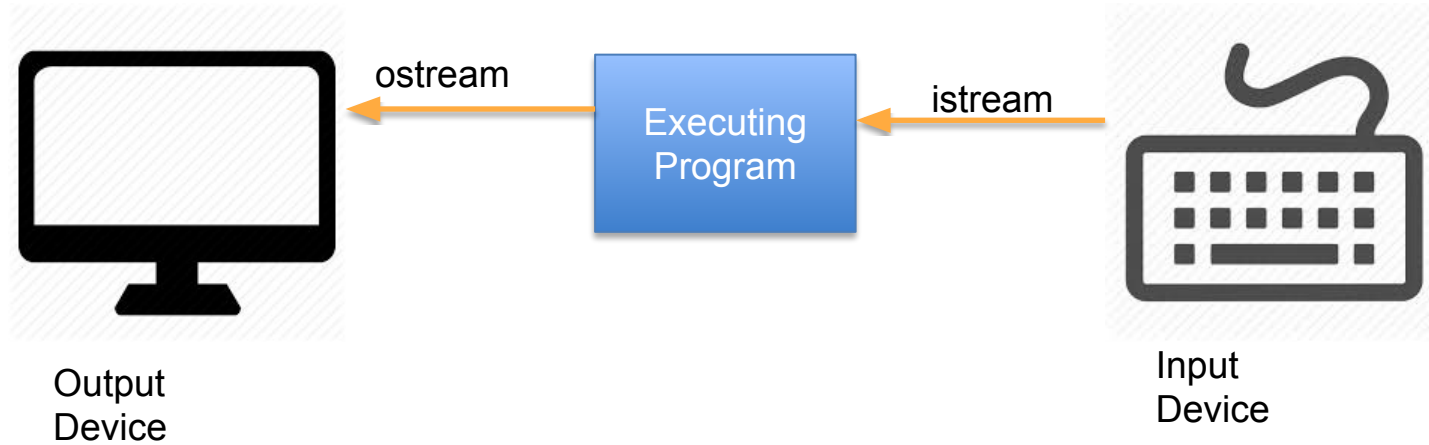


Streams

IO Streams

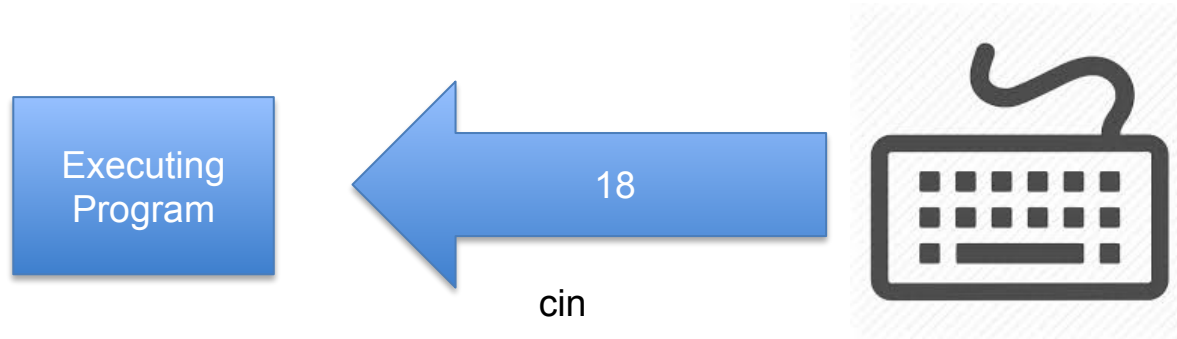


Streams are objects with names such as `cin`, `cout`, or `cerr`

Buffer

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

istream



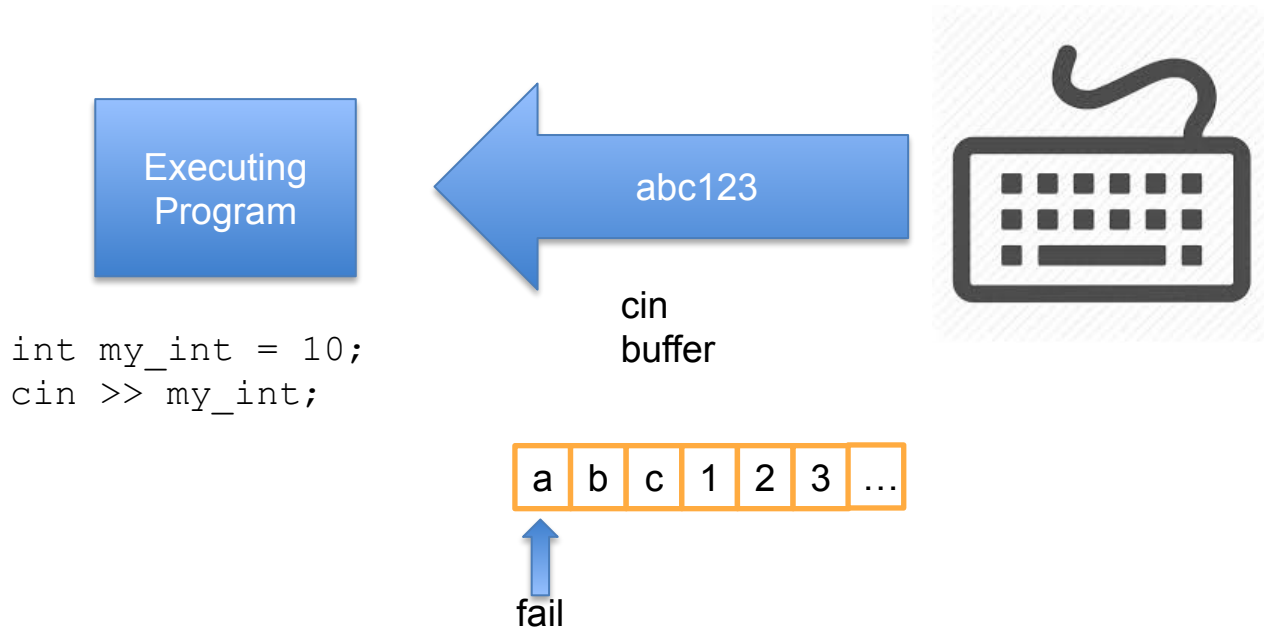
The >> (extraction) operator is an overloaded operator that takes values out of an input stream (cin) and stores them as the type indicated by the target variable.

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

The '1' is read and then the '8' is read.

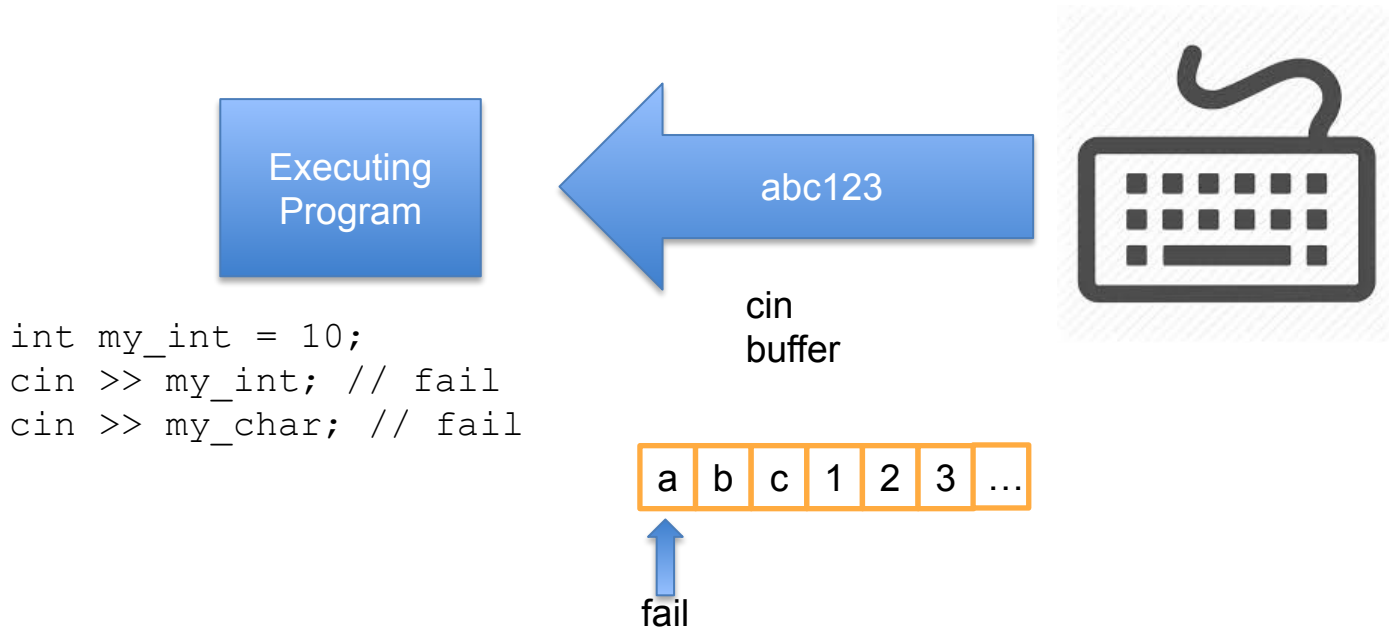
Then the two characters are converted into the integer 18 which is stored in variable my_var.

When cin goes bad



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

fail is fail, you must fix

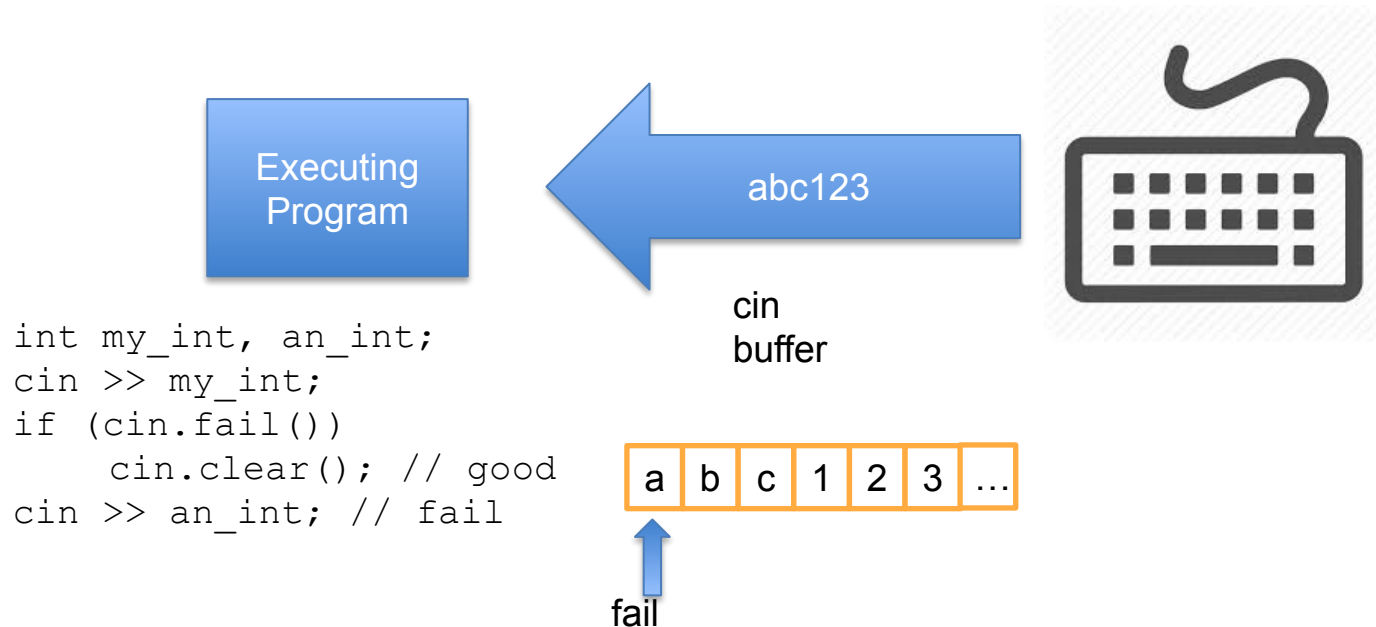


`cin` stays in a failed state until you clean it up. All subsequent reads fail until that happens

Status functions

- Useful boolean member functions
 - `cin.good()`: all is well in the istream
 - `cin.bad()`: something is wrong with the 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()`



`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)`
- Clears that number of characters up to and including `stop_char`
- E.g. `cin.ignore(20, '\n')` skips 20 characters or until `'\n'`, whichever comes first

clear, then ignore

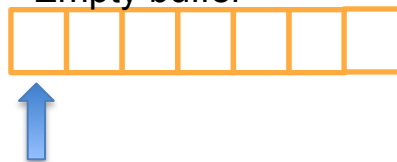
```
int my_int, an_int;  
cin >> my_int;  
if (cin.fail()) {  
    cin.clear(); // good  
    cin.ignore(1000, '\n');  
}  
// reprompt, try again
```

Executing
Program

abc123

cin
buffer

Empty buffer



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

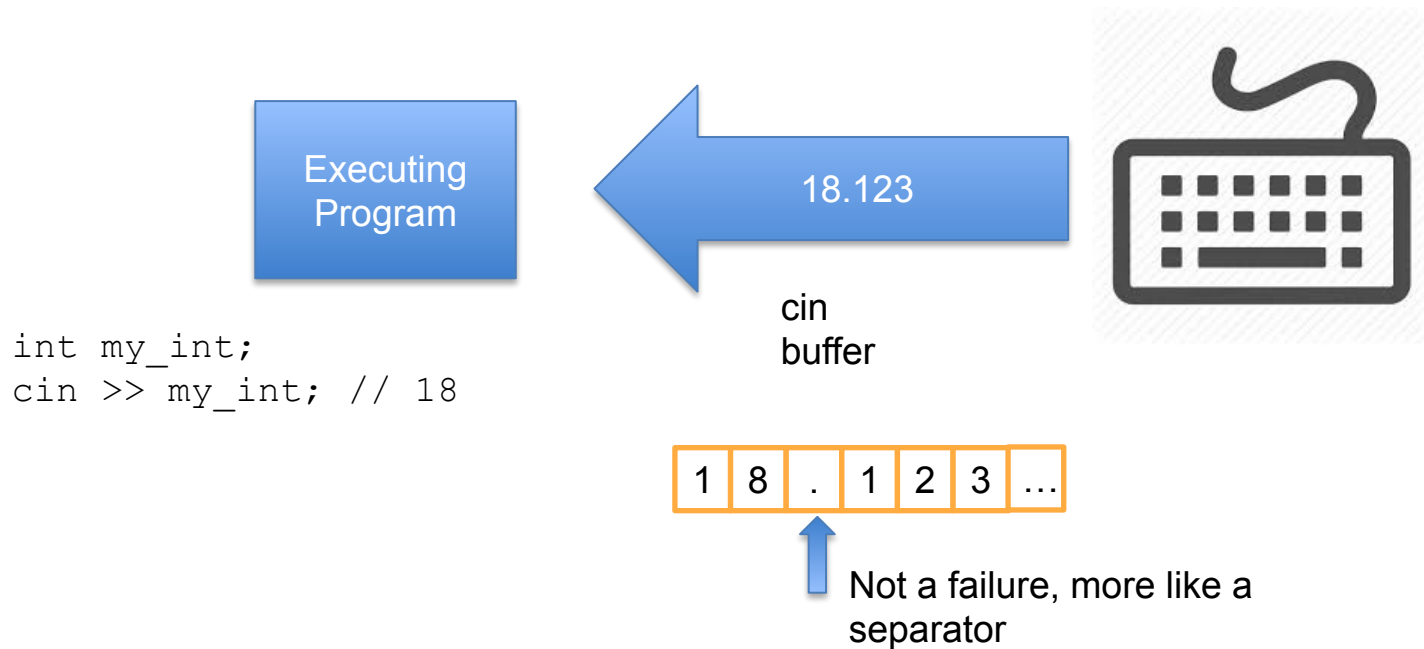
More on ignore

- Takes a default count of 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 at the `eof` char

More complicated for a float

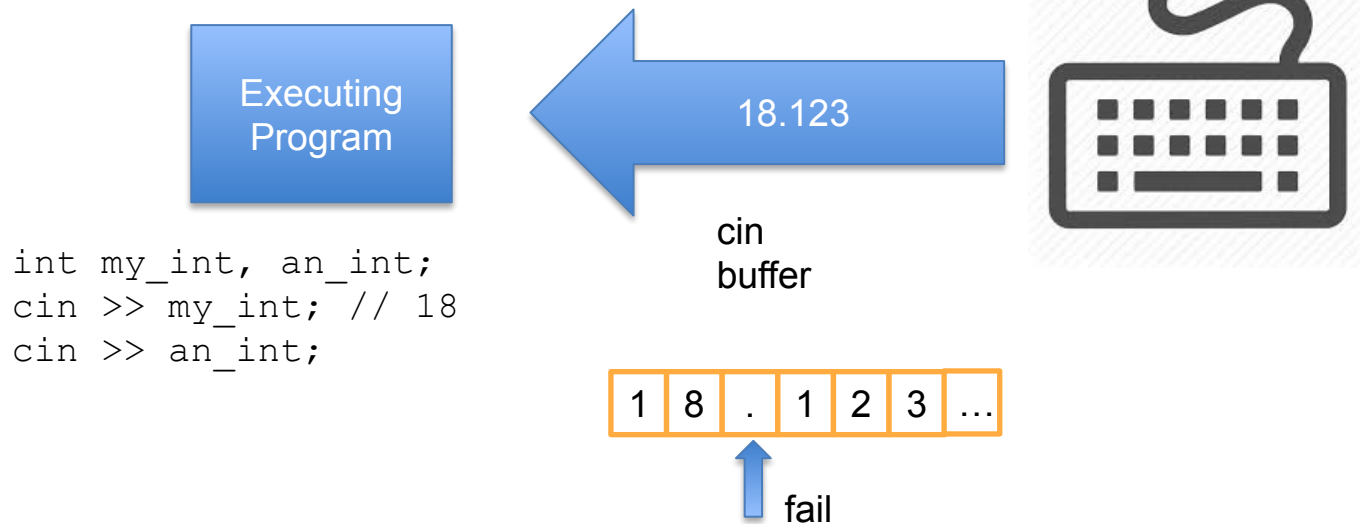
- The situation is more complicated for numbers. For example, try reading a float into an integer

When cin goes bad



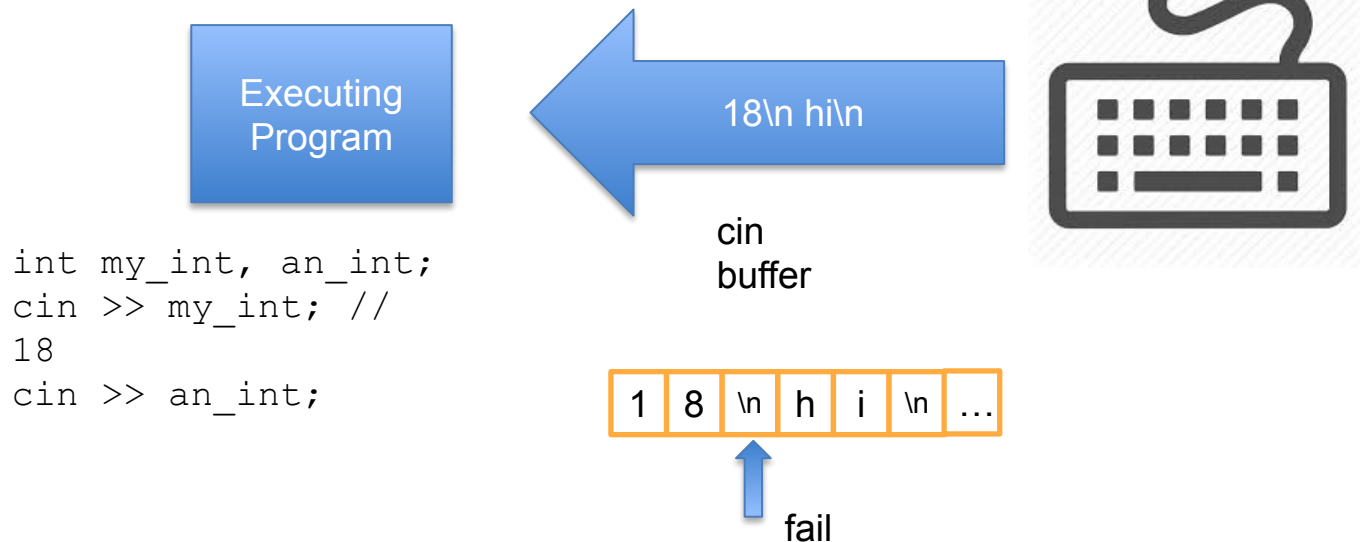
Typed operator, can only read `int` type stuff, stops (***not fails***) at the '.'

When cin goes bad



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

When cin goes bad



Next read is a failure (chokes on the '\n')

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* whitespace

```
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 whitespace
 - `cin.get(ch)` reads **exactly one** character, no matter the character

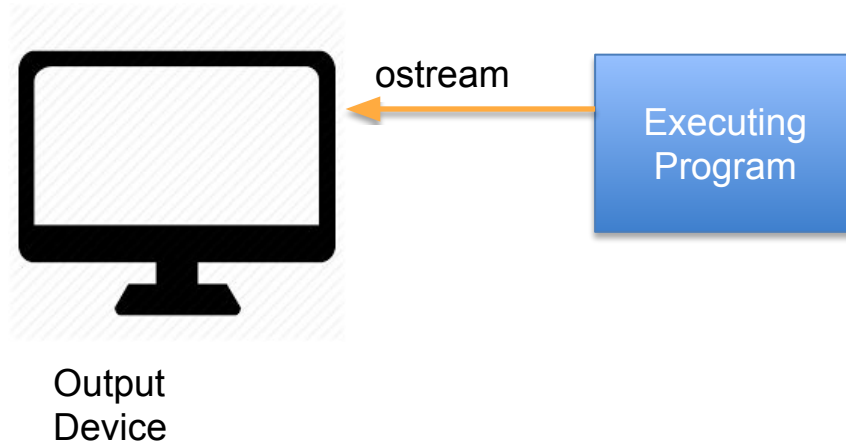
Single Character

- 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

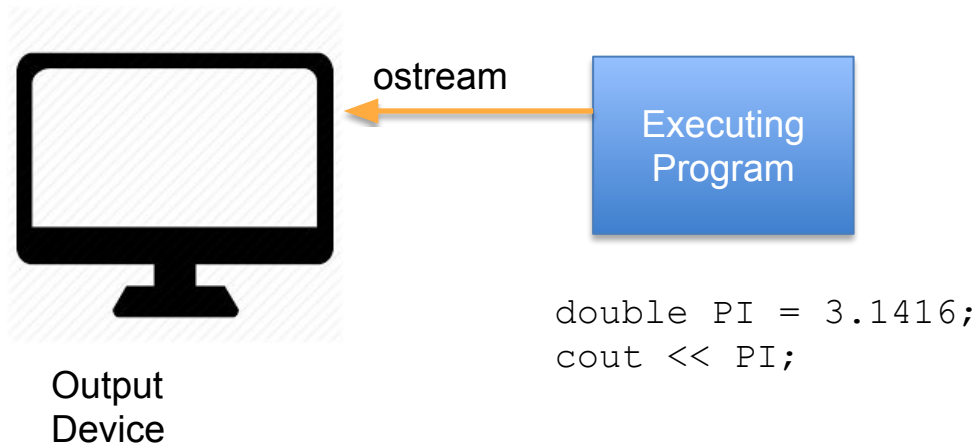
- Single character function:
 - `cout.put(ch)` puts a single character into the `ostream`

ostreams: cout and cerr



cout and cerr are particular instances of an ostream

<< operator



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

Output formatting

- We have seen many of the format codes (see pg 757 of book)
 - `skipws`, `left`, `right`, `dec`, `oct`, `hex`, `uppercase`, `scientific`, `fixed`
 - There are others
- `in_stream.setf(ios::skipws)` is an alternate way to the some of these.
- Book uses the former

Buffer

- Output characters are stored into a buffer before being output
 - i.e. Gather a bunch of characters before sending them to the screen.
- This can be a problem when debugging
 - Output may be in the buffer
 - You believe the error occurred before the output statement when it actually occurred afterward

Buffering and Debugging

```
double f(double x) {  
    cout << "entering f";  
    ...  
    cout << "exiting f";  
    return z;  
}
```

Flush buffer

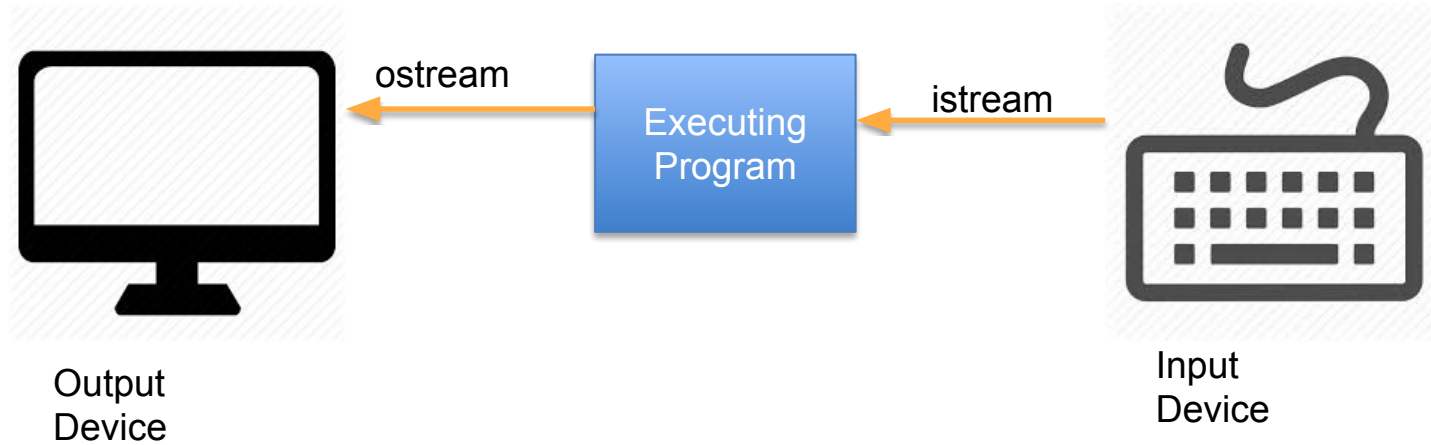
```
double f(double x) {  
    cout << "entering f" << endl;  
    ...  
    cout << "exiting f" << flush;  
    return z;  
}
```

File Streams

Files

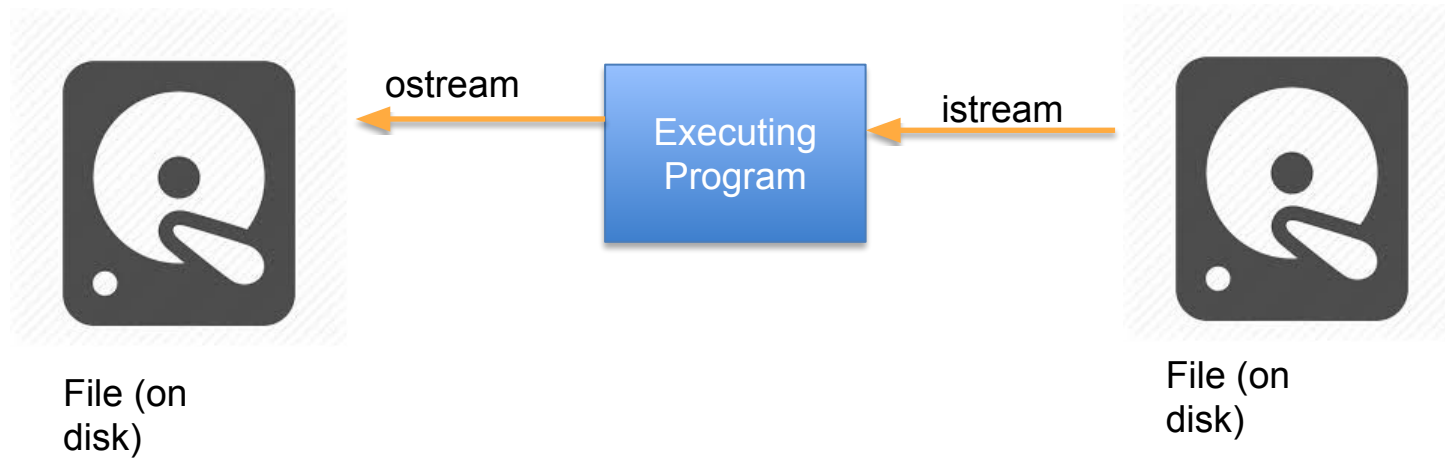
- Files are collections of data and are stored in nonvolatile memory
 - e.g. secondary storage such as disk
- Text files store characters such as ASCII
 - e.g. source code
- Binary files contain non-ASCII characters
 - e.g. compiled code
- Humans can read text files

Stream Review



Streams are objects with names such as `cin`, `cout`, or `cerr`

File Streams



Previous streams are objects with names such as `cin`, `cout`, or `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 directory / folder as the executing program.
- If not there, you may 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/joshnahum/file.txt

Standard Operations

- `>>`, `<<` input and output operations
- `getline(istream, 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

- `in`: open for input
 - `out`: open for output
 - `app`: seek to the end before every write
 - `ate`: seek to the end immediately after opening
 - `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.
 - By default, wipes out any info in the file being written to
 - See Table 8.4 on page 319

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

One can combine all aspects this
way

- This file can read from and write to and writing occurs at the end of the file