- We use **output streams** to send data out of our program to an external destination
    - E.g., cout, or "console out"---sends data from our program to standard output for printing in the terminal
- We use **input streams** to pipe data into our program
    - E.g., cin, or "console input"---pipes data from standard input (the terminal) into our program
- These same rules apply when we want our program to write data to files (output) or read data from files (input)
    - To write data to a file, you create and use an **ofstream** object (output file stream)
    - To read data from a file, you create and use an **ifstream** object (input file stream)
- For both ofstreams and ifstreams, we need to `#include <fstream>`. Also, they're part of the standard namespace (`std::ofstream` and `std::ifstream`).
- Once you've included the `<fstream>` library and dumped the standard namespace (`using namespace std;`), you can declare ofstream variables and ifstream variables just like any other variable. Example:

```
ofstream my_ofstream;
ifstream my_ifstream;
```

- After declaring your ofstream / ifstream variable, you then have to tell it which file you want it to write to (ofstream) or read from (ifstream). You can do this by using the dot operator on your ofstream / ifstream variable to call its open() function.
    - `my_ifstream.open()` accepts one argument that we care about: a string containing the name of the file to work with. For example,

      ```
      my_ifstream.open("data.txt");
      ```

      will tell my_ifstream that we want it to read data from the file called "data.txt" in the **working directory** (the folder from which we run the program in the terminal). If the file with the given name doesn't exist, my_ifstream.open() will fail. The filename can be either a relative path or an absolute path.
    - `my_ofstream.open()` accepts two arguments that we care about: 1) a string containing the name of the file to work with, and 2) a constant denoting the "write mode". By default, the write mode is "truncate", which means that it will erase the existing contents of the file (if it exists) and start writing it from scratch. Alternatively, you can specify the constant `ios::app`, which will tell the ofstream that we just want to append more data to the end of the file (if it exists). In either case, if the file does not exist, it will be created. For example,

      ```
      my_ofstream.open("data.txt");
      ```

will tell my_ofstream that we want to erase the contents of the file called "data.txt" in the working directory and start writing new data to it from scratch. In contrast,

```
my_ofstream.open("data.txt", ios::app);
```

will tell my_ofstream that we want to open the file called "data.txt" in the working directory and append data to the end of it.

Creating and opening an ofstream object

Syntax:

```
ofstream <variable name>;
<variable name>.open("path/to/file.txt");
// OR, FOR APPENDING
<variable name>.open("path/to/file.txt", ios::app);
```

Example 1 (truncate mode):

```
ofstream my_cool_ofstream;
my_cool_ofstream.open("data.txt");
```

Example 2 (append mode):

```
ofstream my_neat_ofstream;
my_neat_ofstream.open("data.txt", ios::app);
```

Creating an ifstream object (exactly the same syntax as ofstream, but no write mode is necessary)

Example:

```
ifstream my_awesome_ifstream;
my_awesome_ifstream.open("data.txt");
```

- Once you have created and opened your ifstream / ofstream object, you can then proceed to use it just like you would cin / cout, respectively.
    - You can use the stream extraction operator, >>, on an ifstream object
        - Recall that the >> operator reads exactly one "word" at a time—it reads characters until it reaches an invalid character (e.g., a decimal point when trying to read an integer) or any whitespace (e.g., a space, tab, or new line).
    - You can use getline() with an ifstream (more on this later)
    - You can use the stream insertion operator, <<, on an ofstream object.
    - These functions and operators work exactly the same on ifstream / ofstream objects as they do on cin / cout, respectively. The only difference is that the data is read from / written to the file rather than the terminal.

- Important: ifstreams and ofstreams cannot be passed or returned by value. More generally, they cannot be copied in any way. For example, the following fails to compile:

  ```
  ofstream my_ofstream = some_existing_ofstream;
  ```

  as will this:

  ```
  ifstream my_function_that_returns_an_ifstream() {
        … // Streams cannot be returned!
  }
  ```

  The general strategy is this: create the ifstream / ofstream object early (e.g., in main()), then pass it around from function to function by reference (usually non-constant reference is required since `>>` and `<<` modify the internal state of the stream object).

"Hello, world!" ofstream example:

```cpp
#include <iostream>
// Don't forget: we need to include <fstream>
#include <fstream>

using namespace std;

string get_filename() {
    cout << "Enter the name of the file: ";
    string name;
    cin >> name;
    return name;
}

int main() {
    // Ask the user for the name of the file
    // they want us to write to.
    string filename = get_filename();

    // Create the ofstream
    ofstream my_ofstream;

    // Tell the ofstream to open the file whose name is stored in `filename`
    my_ofstream.open(filename);

    // Write "Hello, world!" to the file. This is exactly like printing
    // "Hello, world!" to the terminal---we just use my_ofstream instead of cout.
    my_ofstream << "Hello, world!" << endl;
}
```

Here's what it looks like when we run the above program:

```
[alex@alex-laptop tmp]$ ls
main.cpp
[alex@alex-laptop tmp]$ g++ main.cpp
[alex@alex-laptop tmp]$ ./a.out
Enter the name of the file: hello.txt
[alex@alex-laptop tmp]$ ls
a.out  hello.txt  main.cpp
```

And here's what "hello.txt" looks like if we open it in vim (the "1" is just the line number—ignore that):

```
  1 Hello, world!
```

Simple ifstream example:

First, suppose we have a file called "file.txt" with the following contents (again, ignore the 1):

```
  1 the auspicious dog 3.141592
```

Suppose we want our program to read the contents of the file and then print 1) the second word in the file (auspicious), and 2) the decimal value at the end of the file (3.141592). Here's the code:

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

string get_filename() {
    cout << "Enter the filename: ";
    string name;
    cin >> name;
    return name;
}

int main() {
    // Ask the user for the name of the file
    // they want us to write to.
    string filename = get_filename();

    // Create the ifstream
    ifstream my_ifstream;

    // Tell the ifstream to open the file whose name is stored in `filename`
    my_ifstream.open(filename);

    // Read the first word from the file. Yes, even though we don't need
    // to print it to the terminal, we still have to read it in order to get
    // "past" it and proceed to read the second word.
    string word;
    my_ifstream >> word;

    // Read the second word from the file, then print it to the terminal
    my_ifstream >> word;
    cout << "Second word: " << word << endl;

    // Read the third word from the file.
    my_ifstream >> word;

    // Read the decimal number to the file, then print it to the terminal.
    double decimal_value;
    my_ifstream >> decimal_value;
    cout << "Decimal value: " << decimal_value << endl;
}
```

Here's what the program looks like when we run it and enter "file.txt" as the filename:

```
[alex@alex-laptop tmp]$ ls
file.txt  main.cpp
[alex@alex-laptop tmp]$ g++ main.cpp
.[alex@alex-laptop tmp]$ ./a.out
Enter the filename: file.txt
Second word: auspicious
Decimal value: 3.14159
```

Great!

- The stream extraction operator, >>, reads one "word" at a time. What if we want to read an entire line of text from the file and store it all in one big string? To do that, we can use the getline() function, provided by <iostream>. It accepts two arguments: 1) the input stream that we want to read a line of text from (e.g., cin or an ifstream), and 2) the string variable that we want to store the line of text in (it's passed by reference—getline() will overwrite the contents of the string variable with the line of text).

  Important: You can use getline() with cin as well! It reads an entire line of input from the terminal.

  Warning: Try to avoid using both the stream extraction operator (>>) and getline() on a single stream in your program. Try to choose one and stick with it. They interact with whitespace in different, but subtle ways, so switching between them can cause issues. If you really **must** use both of these on a single stream (e.g., cin or an ifstream object), research how they interact with whitespace / delimiters so that you know how to use them together effectively.

ifstream and getline example

Here's "file.txt":

```
1 Hello, world!
2 The quick brown fox
3 jumps over
4 the lazy dog.
```

Suppose we want to ask the user for an integer N, and then print the Nth line in the file. Here's a program that will do that:

```cpp
// Task: Print the Nth line in the file, where N is supplied by the user

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

string get_filename() {
    cout << "Enter the filename: ";
    string name;
    cin >> name;
    return name;
}

int get_n() {
    cout << "Which line do you want to be printed? ";
    int n;
    cin >> n;
    return n;
}

int main() {
    // Ask the user for the name of the file
    string filename = get_filename();

    // Create the ifstream
    ifstream my_ifstream;

    // Tell the ifstream to open the file whose name is stored in `filename`
    my_ifstream.open(filename);

    // Get N from the user
    int n = get_n();

    // Read the first N - 1 lines to "skip" them
    string line;
    for (int i = 0; i < n - 1; i++) {
        getline(my_ifstream, line);
    }

    // Read the nth line and print it
    getline(my_ifstream, line);
    cout << "The nth line is:" << endl;
    cout << line << endl;
}
```

Here's what it looks like when we run it:

```
[alex@alex-desktop tmp2]$ ./a.out
Enter the filename: file.txt
Which line do you want to be printed? 4
The nth line is:
the lazy dog.
```

- Those ifstream examples are great, but they require us to know the format of the file ahead of time (how many lines of text, or how many words are in the file, etc). Suppose you don't know the format of the file, and you just want to keep reading data until you get to the end of the file.

  As you read data from the file, you can think of it like there's a little cursor iterating through the file as you go. Every time you use >>, a word is extracted, and the cursor shifts over a word. Every time you use getline(), a line is extracted, and the cursor shifts down to the beginning of the next line. Once the cursor reaches the very end of the file, the ifstream's **internal EOF (end-of-file) boolean** will be set to **true**. At any point, you can get the value of an ifstream's internal EOF boolean by calling the .eof() function on it (which takes no arguments and returns the boolean value). So, in theory, we can accomplish our goal by using >> or getline() in a loop while my_ifstream.eof() is false (where my_ifstream is the name of our ifstream variable).

  However, take care: most files have a subtle newline character at the end of them (and most text editors and IDEs will automatically insert a newline character at the end of a file whenever you save it, if one isn't present already). You should always assume your input file has a newline at the end of it, unless it's explicitly stated otherwise. Both the >> operator and getline() will stop whenever they encounter a newline character. So if that newline character is present at the end of the file, you'll likely have to use the >> operator or getline() one extra time before the internal EOF boolean will be set to true. For example, if there are 11 words in the file, .eof() will return false until you've used the >> operator 12 times. On the 12th time, it will not extract anything useful—so you usually have to call and check the return value of .eof() immediately after using the >> operator each time, and stop immediately when it returns true. A similar rule applies to getline() (e.g., if there are 4 lines in the file, eof() will be false until you call getline() 5 times; on the 5th time, it will extract an empty line).

ifstream and .eof() example to count the number of words in a file

Suppose we want to ask the user for a file and print out the number of words in the file. Here's a simple program to do that:

```cpp
// Task: Count how many words are in the file

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

string get_filename() {
    cout << "Enter the filename: ";
    string name;
    cin >> name;
    return name;
}

int main() {
    // Ask the user for the name of the file
    string filename = get_filename();

    // Create the ifstream
    ifstream my_ifstream;

    // Tell the ifstream to open the file whose name is stored in `filename`
    my_ifstream.open(filename);

    // The stream extraction operator will read one "word" at a time. Let's
    // just keep using it until it fails due to triggering eof() (i.e., until
    // we've tried to read past the end of the file, including the newline
    // character at the end of it). Count how many times it runs
    // successfully---that will be the number of words in the file.
    int count = 0;
    do {
        string s;
        my_ifstream >> s;
        // If you wanted to count the number of lines in the file, you could
        // replace `my_ifstream >> s` with `getline(my_ifstream, s)`.

        // Assume the file has a newline character at the end. Then eof()
        // will be false until we read all of the words in the file, and THEN
        // read one extra time after that. If it's currently still true, then
        // that means we just successfully read another word---increment count.
        if (!my_ifstream.eof()) {
            count++;
        }

        // If my_ifstream.eof() is false, on the other hand, then we've read
        // all of the words AND read the newline character at the end.
    } while (!my_ifstream.eof());

    // Print the count
    cout << "There are " << count << " words in the file" << endl;
}
```

Let's use the same file as in the last example, file.txt. It had 11 words in it. Here's what the output looks like:

```
[alex@alex-desktop tmp3]$ ./a.out
Enter the filename: file.txt
There are 11 words in the file
```

- Sometimes, errors can occur when using an ifstream / ofstream object. Here are some examples:
    - An ifstream's open() function is called, but the specified file doesn't exist
    - An ifstream's or ofstream's open() function is called, and the specified file exists, but the user does not have the required permissions to read from (ifstream) / write to (ofstream) the file.
    - An ifstream or ofstream is created and opened, and then the file is deleted by some other program before the ifstream / ofstream is done with it.
- When such an error occurs, the ifstream / ofstream object's **internal fail boolean** is set to **true**. At any point, you can get the value of the ifstream / ofstream object's internal fail boolean by calling the .fail() function on it (which accepts no arguments and simply returns a boolean—true if some operation on the stream has failed, and false otherwise).
- You can reset an ifstream / ofstream object's internal fail boolean back to false by calling the .clear() function on it (which accepts no arguments and returns nothing).

ifstream error recovery example (ofstream errors are handled similarly—call .fail() on it and check if it's true)

In this example, we'll keep asking the user for file names until they supply one that we can open with an ifstream. Recall, to open a file with an ifstream, the file must exist, and we must have necessary permissions to read from it. This example also shows how to pass streams to functions (they cannot be copied / passed by value / returned, so we pass by reference). Here's the code:

```cpp
// Task: Count how many words are in the file

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

// Remember: We can't return ifstream objects for the same reason
// that we can't pass them by value (they can't be copied). So instead,
// we create it in main() and pass it by reference. Here, we just call
// .open() on it to tell it which file to use.
void get_file(ifstream& my_ifstream) {
    // Ask the user for a filename until they give you one that
    // we can open successfully
    do {
        cout << "Enter the filename: ";
        string name;
        cin >> name;

        // If there is a leftover fail boolean set to true, reset it
        // to false
        my_ifstream.clear();

        // Try to open the file
        my_ifstream.open(name);

        // If we failed to open the file, scold the user and repeat
        if (my_ifstream.fail()) {
            cout << "Couldn't open the file!" << endl;
        }
    } while(my_ifstream.fail());
}

int main() {
    // Create the ifstream object
    ifstream in_file;

    // Get a VALID file name from the user, and open it via the ifstream
    get_file(in_file);

    cout << "Woo! Successfully opened the file" << endl;
}
```

Here's the output:

```
[alex@alex-desktop tmp4]$ ./a.out
Enter the filename: file_that_doesnt_exist.txt
Couldn't open the file!
Enter the filename: file.txt
Woo! Successfully opened the file
```

Closing streams

Whenever you .open() a stream object, you should remember to .close() it when you're done with it, if necessary. Now, this is rarely necessary. When the stream object falls out of scope (or is otherwise "destroyed"), it will .close() automatically. Similarly, if .open() fails (e.g., because you lack permissions to open the file, or it's an ifstream and the file doesn't exist), then there's no need to call .close().

However, in rare cases, you may want to reuse a file stream object rather than creating a new one. In such a case, you could .close() it, and then .open() it again with a new file. To close a file stream, simply call its .close() function with no arguments:

```
my_ifstream.close();
```

Note: if a stream object is already open, and you proceed to call .open() on it again, it will fail (the stream's internal failure boolean will be set to true, and subsequent calls to .fail() will return true until you call .clear()). So you must call .close() on a file stream object if you want to reuse it to .open() a different file.