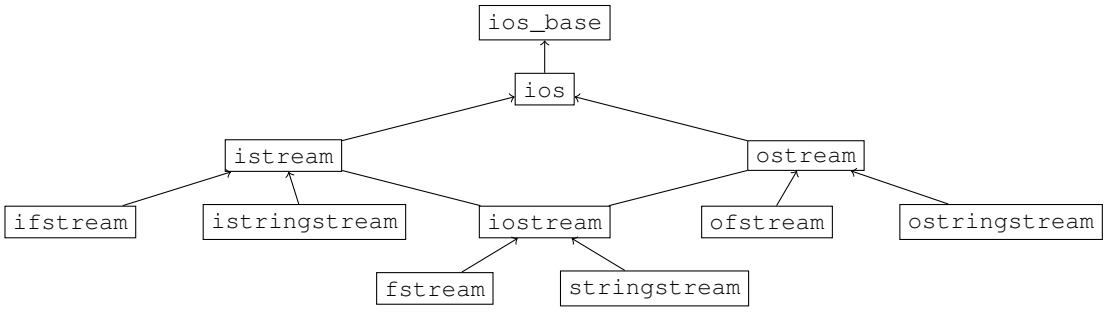
Chapter 2
File and Stream I/O

2.1 Standard I/O Channels

A **stream** is an abstraction that allows programs to send and receive data sequentially. Data can be extracted from a stream using the *extraction operator* `operator>>`, and data can be inserted into a stream using the *insertion operator* `operator<<`.

The C++ `<iostream>` library implements much of the functionality associated with streams. The `<iostream>` library is object-oriented, so streams are defined as objects with the following inheritance hierarchy (all of these objects are part of the `std` namespace, which is not explicitly included in the diagram below due to space constraints):



The `std::ios_base` class serves as the base class of this library. Furthermore, the `std::iostream` class inherits all the members of both the `std::istream` (input stream) and `std::ostream` (output stream) objects, which allows it to perform both input and output operations. Note that `<ios>`, `<istream>`, and `<ostream>` are not explicitly included in most C++ programs, since they are automatically included by the header files of derived class libraries, such as `<iostream>`.

The most common streams that you will use in this class are `std::cin`, `std::cout`, and `std::cerr`. The first of these streams, `std::cin`, is an object of type `std::istream` known as the *standard input stream* for character input (in fact, the name `cin` is short for ***c**haracter **in**put*). The `std::cin` stream accepts input from the user via a standard input device such as a keyboard. To extract characters from the `std::cin` stream, the extraction operator `operator>>` should be used.

In the following code, the user is prompted to enter a string into the console. When the user inputs the string, it is held in the `std::cin` stream and extracted using `operator>>`. After the value is extracted from the string, it is stored in the `name` variable.

```
std::string name;
std::cin >> name;
```

The extraction step would fail if the input from the user cannot be interpreted as the type of the variable that the data is being read into. For instance, the following would fail if the user input cannot be interpreted as an integer.

```
int32_t val;
std::cin >> val; // the next item in the stream must be an int
```

The `std::cout` stream is an object of type `std::ostream` known as the *standard output stream* for character output (where the name `cout` is short for ***c**haracter **out**put*). The `std::cout` object allows characters to be displayed on a standard output device, such as your console. To insert characters into the `cout` stream for printing, the insertion operator `operator<<` should be used.

```
std::string text = "This text is printed out as program output.";
std::cout << text;
```

Lastly, the `std::cerr` output stream is an object of type `std::ostream` that represents the *standard error stream*. The error output stream is distinct from the program output stream, which allows the programmer to work with these two outputs separately. Similar to `std::cout`, writing data to the `std::cerr` stream requires use of the insertion operator `operator<<`.

```
std::string text = "This text is printed out as error message output.";
std::cerr << text;
```

Another thing to note is that these operations work with Boolean values as integer values (0 and 1) instead of their text forms (true and false). In the example below, 1 is printed to the console.

```
bool b = true;
std::cout << b;
```

To work with Booleans in their text representations, use the `std::boolalpha` keyword. The following would print out the word `true` rather than the number 1.

```
bool b = true;
std::cout << std::boolalpha << b;
```

The same keyword applies to reading input as well; using `std::boolalpha` after extracting from `cin` would allow you to read the words "true" and "false" into a Boolean.

```
bool b;                             // user inputs the word "true"
std::cin >> std::boolalpha >> b;    // b is true (would not work without boolalpha)
```

Lastly, the insertion and extraction operators can be chained together in a single expression to insert or extract multiple objects. For example, the following code would print `"EECS281"`, as both the string and integer are inserted into the `std::cout` stream:

```
std::string str = "EECS";
int32_t num = 281;
std::cout << str << num;
```

2.2 Input and Output Redirection

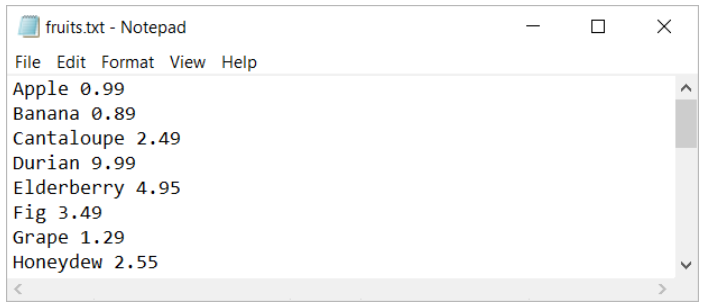
In this class, you will frequently be working with input files for your projects. Given a file, how can you read its contents into your program? One method that can be used is **input redirection**. The goal of input redirection is to redirect the contents of the input file into the `std::cin` standard input stream and then extract the contents out of the stream in your program. As a result, if you use input redirection, you can read from a file by directly extracting from `std::cin` (and treating the contents of the file as if they were inputted by the user on the console).

To illustrate this process, suppose we compiled the following program into an executable named `myProgram` that takes in input from the standard input stream, `std::cin`. If we were to run this program on the command line:

```
./myProgram
```

the user would be prompted to give it input (in this case, item and price).

In this example, we would manually enter each item and its price into the console, which the program reads in using `std::cin`. However, suppose we have all of our inputs in a separate file:



Instead of manually inputting this information into the console one item at a time, we can use the input redirection command to send the entire input file to the program via `std::cin` and treat its contents as user input:

```
./myProgram < students.txt
```

Here, the < symbol represents the *input redirection operator*. This operator redirects the contents of `students.txt` to the standard input stream `std::cin`, so that we can extract its contents using `operator>>` in our program.

Output redirection is a similar process that works the other way around. By using the output redirection operator, or >, we can send the output of a program to a file. This is done by redirecting the contents of the standard output stream `std::cout` to a specified file on the command line. In the following command, the console output of the `myProgram` executable gets written to a file named `report.txt`.

```
./myProgram > report.txt
```

You can also combine input and output redirection in a single command. In the following command, the contents of `students.txt` will be used as input for the program, and the output of the program will be sent to a file named `report.txt`.

```
./myProgram < students.txt > report.txt
```

By using redirection, everything you extract from `std::cin` will come from the file name that follows the < symbol, and everything you print to `std::cout` will be sent to an output file with the file name that follows the > symbol.

Remark: In addition to `std::cin` and `std::cout`, redirection can also be done on the standard error stream, `std::cerr`. To redirect the contents of the `std::cerr` stream to a file, you can use the error redirection operator (`2>`). For example, the following redirects the error output of the `myProgram` executable to a file named `error_report.txt`.

```
./myProgram 2> error_report.txt
```

2.3 Reading Input in a Loop

One interesting feature about `std::cin` and other objects of type `std::istream` is that they can be implicitly converted to a Boolean. If an input stream has no errors and is ready for extraction, the stream object can be implicitly converted to the value `true`. If the input stream is in an error state, the stream can be implicitly converted to a value of `false`.

Because it is possible to implicitly convert between a stream and a Boolean, you can read in an entire file using input redirection by extracting from `std::cin` in the condition of a `while` loop. Because `std::cin` implicitly converts to `false` once it is in an error state, the loop will automatically terminate once there is nothing left to read!

```
1  std::string word;
2  while (std::cin >> word) {
3    // word stores the next word in the input stream
4  } // while
```

Reading the contents of a stream inside a `while` loop is the best way to guarantee that everything is read in correctly. This is especially true when input from `std::cin` is redirected from an input file rather than the user's keyboard.

When working with streams, you may encounter several `std::cin` member functions that can be used to check the status of an input stream, such as `cin.good()`, `cin.eof()`, `cin.bad()` and `cin.fail()`. These should **not** be used as the condition of a `while` loop, as these functions do not return `false` until *after* an operation has failed. This means that the loop would still have to run once after the input becomes invalid.

To illustrate this using an example, suppose we are redirecting input from a text file with 100 values. After we are done reading the 100th value, we go back to the beginning of the 101st iteration and check if `std::cin` is still good. There is no 101st value, but the program doesn't know that! At this point, `std::cin` hasn't gone bad yet, so the loop continues. Only when the program attempts to read in the 101st value does it realize that this value does not exist, and the value of `cin.good()` becomes `false` (the other three functions behave similarly). However, at this point, it is too late: the program has already attempted to read in a non-existent value from the file, so it has already failed.

2.4 Stream Extraction

※ 2.4.1 *The Extraction Operator*

There are several ways to extract the contents out of an input stream. One way, as mentioned previously, is the extraction operator `operator>>`. The extraction operator *ignores* leading whitespace and consumes as many items as possible (for the type being read into) until it encounters whitespace or the *end-of-file* flag (a special flag that denotes that there is no more data to read). The extraction operator also stops reading from the stream if the stream is in an error state (which can happen if the next element extracted from the stream does not match the data type it is being read into). Every time the extraction operator is called, it returns a reference to the `std::istream` object that it reads from (which can be implicitly converted to a Boolean). Let's look at the following example file.

`input.txt`

```
•••here•••is•••a¶
•••file•••to•••be¶
¶
read
```

Here, the "•" character represents a space, and the "¶" character represents a newline character. Let's redirect this file to `std::cin` and extract its contents using `operator>>` to see what happens.

```
1   std::string word;
2   while (std::cin >> word) {
3     std::cout << word;
4   } // while
```

When the program first begins reading from the file, the extraction operator ignores the whitespace before the word "here". Then, it consumes each character of the word "here" until it reaches the whitespace after it (this is because we are trying to read into an object of type `std::string`, `operator>>` extracts everything until it reaches the next whitespace). Once the word "here" is stored in the `word` variable, the program prints it to the standard output stream, or `std::cout`. This continues for the rest of the file, where the extraction operator reads the next string in the stream into the `word` variable, and the newline character acts as a delimiter that separates the previous word from the next. Since whitespace is ignored, the output of the above program is:

`output.txt`

```
hereisafiletoberead
```

The extraction operator begins where you previously left off: if you extract something from the stream, you also remove it from the stream. As a result, you can read in the first five values of a stream, do some other stuff, and then come back to extract the sixth value immediately after. You do not have to start over from the beginning of the file whenever you extract information from a file using input redirection.

※ 2.4.2 *Getline*

Another method for extracting data from an input stream is the `std::getline()` function.

```
std::istream& std::getline(std::istream& is, std::string& str, char delim);
```

Extracts characters from the input stream `is` and stores them into the string `str` until the delimitation character `delim` is found (if `delim` is not specified, the newline character `'\n'` is used by default).

The first parameter is the input stream that should be read from (such as `std::cin`), the second parameter is destination of the input (the object that the input value is stored in), and the third parameter is the delimiting character (the character to read up to). The third parameter is optional, and it's perfectly okay to call `std::getline()` on just two parameters:

```
std::istream& std::getline(std::istream& is, std::string& str);
```

If the third parameter is omitted, the newline character (`'\n'`) acts as the delimiter.

The `std::getline()` function consumes all characters — even whitespace — until it reaches the delimiting character. This delimiting character is discarded, and everything that was read up to it is stored. Like with the extraction operator, `std::getline()` returns a reference to the input stream, which can be implicitly converted to `false` once the stream goes bad. This means that `std::getline()` can be placed as the condition of a `while` loop if the function is being called in a loop. Let's look at the same example file, but this time we will use `std::getline()` to extract characters, using the newline character as a delimiter.

`input.txt`

```
•••here•••is•••a¶
•••file•••to•••be¶
¶
read
```

The following code will be used.

```
1   std::string line;
2   int32_t counter = 1;
3   while (std::getline(std::cin, line)) {
4     std::cout << "line " << counter++ << ": " << line << std::endl;
5   } // while
```

Here, the first call to `std::getline()` consumes the entire first line up to the first newline character (¶) and stores the result in the variable `line`. Hence, during the first iteration of the loop, the string `line` has a value of "••• here ••• is ••• a", including the spaces (only the delimiting character is discarded). With each subsequent iteration, `std::getline()` retrieves the part of the stream up to the next newline character, storing its contents into `line`. The final output of the above code is as follows:

`output.txt`

```
line 1: ••• here ••• is ••• a
line 2: ••• file ••• to ••• be
line 3:
line 4: read
```

On the third iteration, `std::getline()` does not retrieve any characters at all, since there are no characters between the second and third newline characters. As such, the output for line 3 is empty.

As mentioned previously, we can specify a custom delimiter with the `std::getline()` function. For example, suppose we have the following file (with no newlines):

`input.txt`

```
0|12332|99:74:11:21:61|TCP|Connection attempt failed|149|36240|70:17:34:28:94|TCP|Connection lost|281|
25313|11:30:32:34:70|OS|System failure
```

To split this file using `'|'` as the delimiting character, simply pass it as the third argument of each `getline()` call:

```
std::string line;
int32_t counter = 1;
char delim = '|';
while (std::getline(std::cin, line, delim)) {
  std::cout << "line " << counter++ << ": " << line << std::endl;
} // while
```

This code would produce the following output:

```
line 1:  0
line 2:  12332
line 3:  99:74:11:21:61
line 4:  TCP
line 5:  Connection attempt failed
line 6:  149
line 7:  36240
line 8:  70:17:34:28:94
line 9:  TCP
line 10:  Connection lost
line 11:  281
line 12:  25313
line 13:  11:30:32:34:70
line 14:  OS
line 15:  System failure
```

One common mistake with `std::getline()` involves reading from an input stream using both `operator>>` and `std::getline()` on the same line of input. Because `>>` and `std::getline()` both start from where you previously left off, both operations may end up reading from the same line! For example, consider this file, where "•" represents a space and "¶" represents a newline character:

`input.txt`

```
2 ¶
apple¶
banana¶
```

Let's try to run the following code, which uses both the extraction operator `>>` and `getline()` to extract from standard input. In this code, we assume that the first line of the input file represents the number of fruits in the file, and we loop through these fruits and print them out:

```
int32_t num_fruits;
std::cin >> num_fruits;
std::string line;
for (int32_t i = 1; i <= num_fruits; ++i) {
  std::getline(std::cin, line);
  std::cout << "fruit " << i << ": " << line << std::endl;
} // for i
```

However, this is the output we get:

```
fruit 1:
fruit 2:  apple
```

We called `std::getline()` twice after reading in the number on the first row, so `"banana"` should have been extracted. Why wasn't it? Recall that `std::getline()` reads until it reaches a newline character (or delimiting character, if otherwise specified) and then discards that delimiting character. However, after extracting `2` out of the input stream, the next character in line is a newline character; as a result, `std::getline()` would not read in anything at all! To fix this, you would have to run `std::getline()` an additional time to remove the residual input that was extracted on line 1. One possible approach is shown below:

```
1  int32_t num_fruits;
2  std::cin >> num_fruits;
3  std::string line;
4  for (int32_t i = 1; i <= num_fruits; ++i) {
5    std::getline(std::cin, line);
6    // get rid of the line if it is all whitespace
7    if (line.find_first_not_of(" ") == std::string::npos) {
8      --i;
9      continue;
10   } // if
11   std::cout << "fruit " << i << ": " << line << std::endl;
12 } // for i
```

Remark: The `std::string::find_first_not_of()` function can be used to find the first character in a string that does not match any of the characters in a given input string. We will discuss `std::string` operations at greater length in chapter 16, so you do not have to worry about this now.

However, a cleaner way would be to use `std::istream::ignore()` whenever you switch from `std::cin` to `std::getline()`, as shown on line 3 below (include the `<limits>` library to use `std::numeric_limits<>`).

```
1  int32_t num_fruits;
2  std::cin >> num_fruits;
3  std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // remove whitespace and newline
4  std::string line;
5  for (int32_t i = 1; i <= num_fruits; ++i) {
6    std::getline(std::cin, line);
7    std::cout << "fruit " << i << ": " << line << std::endl;
8  } // for i
```

※ 2.4.3 *Ignore* (✻)

As shown, one useful feature of input streams is `std::istream::ignore()`, which can be used to discard undesired characters.

```
std::istream& std::istream::ignore(std::streamsize n, int delim);
```

Extracts and discards either

- `n` characters from the input stream
- until and including `delim`, if `delim` is encountered
- until the end of the file or stream

whichever comes first. If `n` is not specified, then its value defaults to 1. If `delim` is not specified, then its value is set to a special end-of-file identifier (which indicates the end of a data source). If you do not care about `n` and simply want to discard all characters up to a delimiter, you should set `n` equal to `std::numeric_limits<std::streamsize>::max()`.

Note: `std::streamsize` is essentially a signed version of `size_t` (i.e., it can be positive or negative).

This ignore feature can be handy if you know that certain characters of your input should be discarded. For instance, suppose we had the following input file, but with the label `"Count:"` directly before the number of fruits specified.

`input.txt`

```
Count:•2¶
apple¶
banana¶
```

If we wanted to ignore this word before reading in the number of fruits, we could have used an additional junk variable to store this undesired input, as shown in the code below:

```
1  std::string junk;
2  std::cin >> junk; // reads in "Count:"
3  int32_t num_fruits;
4  std::cin >> num_fruits; // reads in 2
5  std::string line;
6  for (int32_t i = 1; i <= num_fruits; ++i) {
7    std::getline(std::cin, line);
8    // get rid of the line if it is all whitespace
9    if (line.find_first_not_of(" ") == std::string::npos) {
10     --i;
11     continue;
12   } // if
13   std::cout << "fruit " << i << ": " << line << std::endl;
14 } // for i
```

However, as mentioned earlier, a cleaner method would be to use `ignore()` to discard all characters up to the first space.

```
1  // ignore everything up to and including the first space character
2  std::cin.ignore(std::numeric_limits<std::streamsize>::max(), ' ');
3  int32_t num_fruits;
4  std::cin >> num_fruits; // reads in 2
5  // ignore everything up to the next newline
6  std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
7  std::string line;
8  for (int32_t i = 1; i <= num_fruits; ++i) {
9    std::getline(std::cin, line);
10   std::cout << "fruit " << i << ": " << line << std::endl;
11 } // for i
```

Another example is shown below, this time ignoring a specified number of characters instead of using a delimiter:

input.txt

```
123456789
```

```
1  std::cin.ignore(5); // extract and discard first five characters
2  int32_t num;
3  std::cin >> num;
4  std::cout << num << std::endl; // prints out 6789
```

※ 2.4.4 *Summary of Stream Operations*

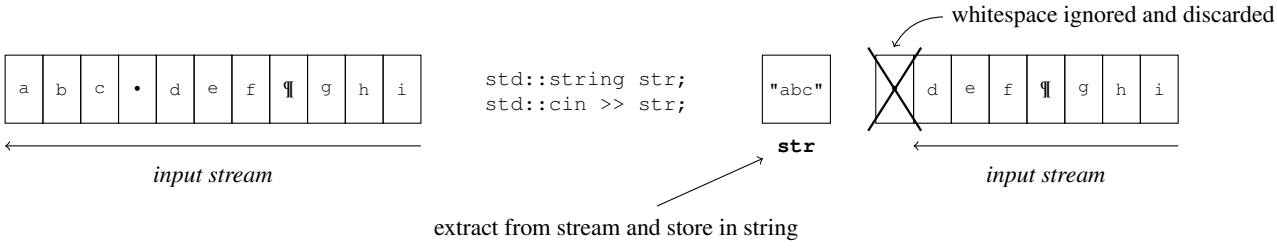
So far, we have discussed several methods that can be used to extract from a stream. The two most important of these are the extraction operator `>>` and `std::getline()`. Illustrations depicting the functionality of these two operations are shown below:

`operator>>`

Extracts as many items out of the stream as possible for the type being read into, until whitespace/newline or end-of-file is reached.

input.txt

```
abc•def¶
ghi¶
```

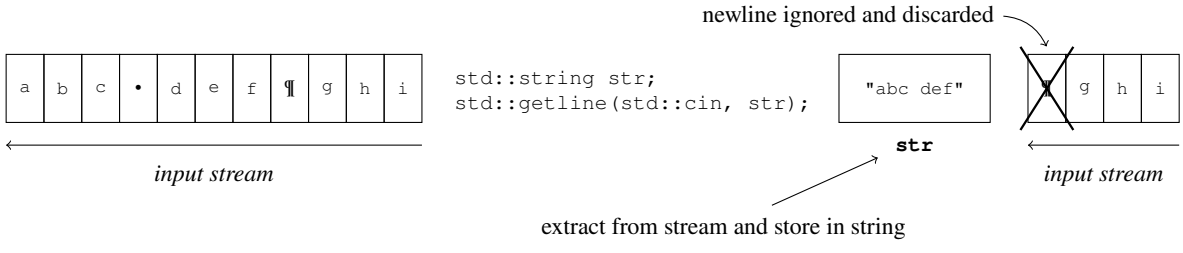


`std::getline()`

Extracts as many items out of the stream as possible and stores into a string, until a newline (or specified delimiter) or end-of-file is reached.

input.txt

```
abc•def¶
ghi¶
```



2.5 Stream Buffers and Flushing (*)

When you send data to an output stream, the characters are not always immediately written to the console (or a file if you are using output redirection). Instead, depending on the stream type, the output values may be buffered before they are fully printed out.

A **buffer** is a block of memory allocated for a stream object to hold temporary stream data. When you send data to `std::cout`, for instance, the data gets stored in this buffer instead of appearing immediately in an output console or file. This is done for efficiency reasons; it is much faster to write data in blocks rather than one character at a time.

A buffer usually gets emptied (or *flushed*) once it is filled to capacity. However, buffers can be manually flushed by applying the `std::flush` manipulator on its output stream.

```
1   std::cout << "Hello World";   // "Hello World" gets sent to cout output buffer
2   std::cout << std::flush;      // flushes buffer, "Hello World" written to console
```

The `std::cout` stream is *block buffered*, meaning that characters are stored in a buffer and written in chunks rather than one at a time. The same is true for the `std::cin` input stream, which buffers input values so that they are read and processed in chunks. The one exception to this rule is the standard error stream `std::cerr`, which is *unbuffered by default* (i.e., output sent to `std::cerr` will appear in the destination location as soon as it is written). We will look at a reason why `std::cerr` is unbuffered later in this section.

The fact that excessive flushing is inefficient has consequences regarding the printing of newlines. In previous classes, you likely used `std::endl` to print out a newline character. However, you can also print out the newline character using `'\n'` (the escape sequence representing a newline character). Let's consider the following two implementations, which do the exact same thing:

```
1   for (int32_t i = 0; i < 100000; ++i) {
2     std::cout << "Hello World" << std::endl;     // use std::endl to print a newline
3   } // for i
4
5   for (int32_t i = 0; i < 100000; ++i) {
6     std::cout << "Hello World\n";                // use '\n' to print a newline
7   } // for i
```

Upon first glance, both loops should take roughly the same amount of time to complete. However, the second approach using `'\n'` is actually much faster than the first! How is this even possible? The reasoning lies in the implementation details of `std::endl`. The `std::endl` operation doesn't just print out a newline — it also flushes the buffer. In other words, calling `std::endl` is equivalent to calling

```
std::cout << '\n' << std::flush;
```

In the first implementation, the output stream buffer is flushed at the end of *every single iteration* of the loop. Meanwhile, `'\n'` does not invoke an immediate flush after every iteration, so you save time by buffering the output and printing it out in chunks. Therefore, if you do not intend on writing output immediately to your destination, it is faster to use `'\n'` instead of `std::endl` to print a newline to the output stream.

There are cases, however, where using `std::endl` is preferred. When output is buffered, and the program crashes before the buffer is flushed, you may end up losing the information in the buffer. Consider the following:

```
280   ...
281   std::cout << "Reached line 281\n";
282   ...


310   ...
311   raise(SIGSEGV); // oops, there's a segfault on this line
312   std::cout << "Reached line 312\n";
313   ...
```

If the output on line 281 was not flushed from the buffer before the program crash on line 311, it is lost. Running this program may result in the following console output, which would lead one to believe that the program never made it to line 281:

```
Segmentation fault (core dumped)
```

However, if these debug lines were flushed immediately to the terminal, then the output "Reached line 281" would appear as soon as the code reached line 281! If we ran the following code instead:

```
280   ...
281   std::cout << "Reached line 281" << std::endl;
282   ...


310   ...
311   raise(SIGSEGV); // oops, there's a segfault on this line
312   std::cout << "Reached line 312" << std::endl;
313   ...
```

the following would be printed to console:

```
Reached line 281
Segmentation fault (core dumped)
```

Now, we know that the program made it to line 281, but crashed before making it to line 312. This is a reason why the `std::cerr` error stream is unbuffered and prints to its destination immediately (and why it is preferred to send error output to `std::cerr` instead of `std::cout`)!

2.6 C and C++ Stream Synchronization (⁎)

The `std::cin` and `std::cout` streams are C++ objects. In C, however, things are a bit different. Unlike C++, the three standard streams in C are `stdin`, `stdout`, and `stderr`. To print output to standard output in C, you would use the `printf()` function instead of the insertion operator and `std::cout`. To read input from standard input in C, you would use the `scanf()` function instead of the extraction operator and `std::cin`. You do not need to know how to work with C in this class, but it is still worthwhile to know that these different I/O methods exist.

Because C and C++ I/O can be used in conjunction, there needs to be a way to ensure that streams in C and C++ are synchronized if a programmer uses both C and C++ style I/O in their program. For example, if you print output using C style I/O, and then print out additional output using C++ style I/O, the output from the C I/O should be printed to the console *before* the output from the C++ I/O. In C++, this synchronization is ensured by making C++ streams share a buffer with their corresponding C stream (e.g., `std::cout` and `stdout` write to the same buffer). As an illustration, consider the following example:

```
1  int main() {
2    printf("EECS");
3    std::cout << "281";
4  } // main()
```

Here, `"EECS"` is sent to the C `stdout` stream and `"281"` is sent to the C++ `std::cout` stream. However, since `"EECS"` is sent to an output stream first, you should expect `"EECS"` to be printed before `"281"`. This process is known as *stream synchronization*.

However, synchronization can slow down the process of I/O, since extra work is needed to maintain that the correct ordering of characters. It is possible to turn off synchronization between C and C++ streams using the `sync_with_stdio()` member function in `std::ios_base`:

```
std::ios_base::sync_with_stdio(false);
```

By running this line of code, you turn off synchronization with `stdio` (the name of the C standard input and output library). This can be used to speed up the execution time of your program, especially if you process a lot of input or output. However, it is important to note the side effects of turning off synchronization — this line should not be added if your program uses both C and C++ style I/O! If we run the following code with synchronization turned off:

```
1  int main() {
2    std::ios_base::sync_with_stdio(false);
3    printf("EECS");
4    std::cout << "281";
5  } // main()
```

there is no longer a guarantee that `"EECS"` will be printed before `"281"`. Since there is no synchronization between C and C++ streams, the two are allowed to have their own independent buffers. If the C stream's buffer flushes first, the output would be `"EECS281"`, but if the C++ stream's buffer flushes first, the output would be `"281EECS"`. If both streams try to flush at the same time, you might even end up with something like `"E2E8C1S"`! Without synchronization, all bets are off.

That being said, you will not be doing C-style I/O in this class, so it is perfectly okay to turn synchronization off for the projects in this class. Just note that the improved performance is a side effect of this operation, and that you should not blindly include this line in every program you write, just because it might speed things up.

2.7 Stringstreams (⁎)

A **stringstream** allows you to treat a `std::string` object as if it were a stream. That is, it gives you the ability to read/write to a string just like how you would read/write a stream. To use stringstreams in your program, make sure to include the `<sstream>` library.

If you want to treat a `std::string` as an *input* stream, you can declare an `std::istringstream`. An object of this type inherits from `std::istream` (see the figure in section 2.1), so an `std::istringstream` can be used wherever an input stream of type `std::istream` is expected (due to polymorphism).

An example using an `std::istringstream` is shown below. In this example, a `std::string` containing numbers is converted to an `std::istringstream`, and the extraction operator `>>` is used to extract the numbers out of the `std::istringstream` as integers. The `str()` member function can be used to set the contents of the stream. The following code outputs `"4 66 30 46 "`.

```
1  std::istringstream iss;
2  std::string numbers = "2 33 15 23";
3  iss.str(numbers); // initializes iss with the contents of numbers string
4  int32_t val;
5  for (int32_t i = 0; i < 4; ++i) {
6    iss >> val;                      // extracts number from iss and stores in val
7    std::cout << val * 2 << ' ';     // multiplies val by two and prints it out
8  } // for i
```

If you want to write to a `std::string` like an output stream, you can declare an `std::ostringstream`. An object of this type inherits from `std::ostream`, so an `std::ostringstream` can be used wherever an output stream of type `std::ostream` is expected (again due to polymorphism).

An example using an `std::ostringstream` is shown below. In this example, we initialize an `std::ostringstream` and write data to it using the insertion operator `<<`. Then, we convert the contents of the stream into a string using the `str()` member function.

```
1  std::ostringstream oss;
2  oss << "I really enjoy EECS 281";   // sends text to the ostringstream
3  oss << "!!!";                       // sends text to the ostringstream
4  std::string s = oss.str();          // converts contents of oss to a string
5  std::cout << s << '\n';             // prints the final string
```

The output of the above code is `"I really enjoy EECS 281!!!"`.

2.8 File Streams (*)

Input redirection is not the only way you can read or write data from files. Another method is to use **file streams** in the <fstream> library. There are three main types of file streams that we will cover in this section. An std::ifstream object, or input file stream, is a type of stream that allows you to read from a file. An std::ofstream, or output file stream, is a type of stream that allows you to write to a file. Lastly, a generic std::fstream object is a general file stream type that is able to do both file reading and file writing.

To use an std::ifstream, here are a few member functions that are important to know:

Function	Behavior
void open(**const char*** filename);	Opens the file with the name filename and associates it with the input file stream object
bool is_open();	Checks to see if the input file stream is associated with a file that was successfully opened
void close();	Closes the file associated with the input file stream and disassociates it from the stream
operator>>	Extracts input from the input file stream (this is inherited from std::istream)

Once an std::ifstream is associated with a file, you can use it in the same way as any other input stream. The code below creates an std::ifstream object that reads from a file named input.txt. When working with file streams, it is good practice to check if the file was successfully opened (using is_open()) before extracting from it.

```
1  // create ifstream object and associate it with input.txt
2  std::ifstream fin;
3  fin.open("input.txt");
4
5  // check if file was successfully opened
6  if (!fin.is_open()) {
7    std::cerr << "Failed to open file." << std::endl;
8    exit(1);
9  } // if
10
11 // read from file using ifstream
12 std::string word;
13 while (fin >> word) {
14   // do stuff here
15 } // while
16
17 // close file after done
18 fin.close();
```

An std::ofstream object has similar functions for opening and closing files. Instead of the extraction operator, it supports the insertion operator for writing output to a file.

Function	Behavior
void open(**const char*** filename);	Opens the file with the name filename and associates it with the output file stream object
bool is_open();	Checks to see if the output file stream is associated with a file that was successfully opened
void close();	Closes the file associated with the output file stream and disassociates it from the stream
operator<<	Inserts output into the output file stream (this is inherited from ostream)

The following code creates an std::ofstream object, associates it with the "output.txt" file, and then writes to the file using the std::ofstream object. Much like before, the status of the output file stream should be checked with is_open() before writing to it.

```
1  // create ofstream object and associate it with output.txt
2  std::ofstream fout;
3  fout.open("output.txt");
4
5  // check if file was successfully opened
6  if (!fout.is_open()) {
7    std::cerr << "Failed to open file." << std::endl;
8    exit(1);
9  } // if
10
11 fout << "This data is written to the file." << std::endl;
12
13 // close file after done
14 fout.close();
```

The std::fstream type supports both insertion and extraction, as well the other member functions listed above. Thus, an std::fstream object provides both read and write access to a file. However, it is good practice to use std::ifstream and std::ofstream instead of std::fstream unless you absolutely need both read and write access to a file in a single object (which is not a common occurrence). This is because the std::ifstream and std::ostream objects make it clear whether you are reading or writing, which minimizes the chance for human error compared to a generic std::fstream object.

2.9 Reading Input Using Polymorphism (*)

To recap, most command line shells allow programs to send output to or read input from files. This is accomplished by using the input redirection operator (>) or the output redirection operator (<). Reading input from a file can be thought of as temporarily disconnecting the keyboard and getting all input from a specified input file. This technique is known as **input redirection**, and replaces keyboard input with input from a file.

On a similar vein, writing output to a file can be thought of as temporarily disconnecting the screen and sending everything that would have been printed to the console directly to a file. The file does not need to exist before redirection; it will be created as necessary. If the file already exists, its original contents will be replaced with the new output. This technique is known as **output redirection** and sends text that would have been displayed on the screen directly to the specified file.

Redirection can be done on input, output, both, or neither, as well as on the standard error stream `std::cerr`. As previously mentioned, the error redirection operator (`2>`) can be used to redirect output in `std::cerr` to a file.

Often, a program will try to read from a file specified at the command line, but if one is not specified, it will try to read from standard input. Programs that do this perform identically when invoked with the following two commands:

```
./program < input.txt
 ./program input.txt
```

The first variant redirects the contents of the `input.txt` file to the `program` executable using input redirection, while the second variant specifies the file name directly as an argument on the command line (and thus should be read in using a file stream). How can we write a program that supports both methods of specifying an input file? That is, the program must be able to retrieve input from the `input.txt` file, regardless of whether it is directly specified on the command line (no < before the file name) or sent in via input redirection (using <).

It turns out this can be done with a few conditionals, thanks to polymorphism! Recall that the `std::ifstream` class inherits from the `std::istream` class (see the diagram in section 2.1). Polymorphism thus allows an object of type `std::istream` to be assigned an object of type `std::ifstream`, since `std::ifstream` is a subclass of `std::istream`. As a result, we can create a single `std::istream` object in our program and assign it to either `std::cin` (if a file is passed in using input redirection) or an input file stream (if the file name is specified as an argument on the command line).

```
1  std::ifstream fin;
2  std::string filename;     // use argc and argv to retrieve file name from the
3  if (argc > 1) { ... }     // command line if it is specified (see chapter 3)
4
5  if (!filename.empty()) {
6    fin.open(filename);
7    if (!fin.is_open()) {
8      std::cerr << "Unable to open input file: " << filename << std::endl;
9      exit(1);
10   } // if
11 } // if
12
13 std::istream& in = fin.is_open() ? fin : std::cin;
14
15 // Read input by extracting from 'in' for the rest of the program (in >> ...)
16 // If a file was specified, 'in == fin' (extracts from file stream)
17 // If no file was specified, 'in == cin' (extracts from standard input)
18
19 fin.close();
```

In this code, we first create an input file stream object named `fin`. Then, we look on the command line to see if a file name was specified (command line parsing will be covered in the next chapter). On line 5, we check to see if we found a file name on the command line. If we found a file name, we use the file stream to open it on line 6.

Line 13 is where the polymorphism comes into play. We create a single `std::istream` object that is used to read input in the program. If the file stream `fin` was successfully associated with a file, then we would read from the filestream (i.e., we would assign `in` to `fin`). Otherwise, we would read from standard input (i.e., we would assign `in` to `std::cin`). After line 13 runs, the stream `in` will hold the input we want to read, regardless of whether that input is sent via a file stream (`fin`) or standard input (`std::cin`). Note that `in` is a *reference* to the stream it is assigned; thus, when we read from `in`, we are always reading from the correct input stream (`in` is a reference to `fin` if the file is passed in via the command line, and a reference to `std::cin` if input redirection is used).