Chapter 2 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Which of the following statements is most accurate regarding `operator>>` and the default behavior of `std::getline()` (without any custom delimitation character specified)?
   - **A)** The extraction operator >> ignores leading whitespace, but `std::getline()` does not
   - **B)** `std::getline()` ignores leading whitespace, but the extraction operator >> does not
   - **C)** Both the extraction operator >> and `std::getline()` ignore leading whitespace
   - **D)** Neither the extraction operator >> and `std::getline()` ignores leading whitespace
   - **E)** None of the above

2. Consider the following code, which reads from a file using input redirection:

```
1   int main() {
2     int32_t num = 0;
3     std::string line;
4     int32_t counter = 0;
5     while (std::cin >> num) {
6       std::getline(std::cin, line);
7       ++counter;
8     } // while
9     std::cout << "num: " << num << std::endl;
10    std::cout << "counter: " << counter << std::endl;
11  } // main()
```

   Suppose this code were run on the following input file, where the "•" character represents a space, and the "¶" character represents a new line. What is the output of this program?

   `input.txt`

   ```
   ••••• 17 • 22 •••• 14 ¶ 19 ¶ ¶ • ¶ •• ¶ 10 ¶ 13 ¶ 15 •• 27 ••• 12 •• 20 ¶
   ```

   - **A)**
     ```
     num: 15
     counter: 3
     ```
   - **B)**
     ```
     num: 15
     counter: 5
     ```
   - **C)**
     ```
     num: 15
     counter: 8
     ```
   - **D)**
     ```
     num: 20
     counter: 5
     ```
   - **E)**
     ```
     num: 20
     counter: 8
     ```

3. Consider the following function, `write_wordle_words()`, which reads input words from an input stream and writes them to an output stream only if they have a length of five.

```
1   void write_wordle_words(____________________ input, ____________________ output) {
2     std::string word;
3     while (input >> word) {
4       if (word.length() == 5) {
5         output << word << std::endl;
6       } // if
7     } // while
8   } // write_wordle_words
```

   You want this method to be able to take in *any* kind of input stream and *any* kind of output stream. For example, both of the following function calls should work:

```
9   // use file streams
10  std::ifstream fin("input.txt");
11  std::ofstream fout("output.txt");
12  write_wordle_words(fin, fout);
13
14  // use standard streams
15  write_wordle_words(std::cin, std::cout);
```

   To accomplish this, how should the parameter types of the `write_wordle_words()` function be defined? Choose the best answer.
   - **A)** `std::istream, std::ostream`
   - **B)** `std::istream&, std::ostream&`
   - **C)** `std::ifstream, std::ofstream`
   - **D)** `std::ifstream&, std::ofstream&`
   - **E)** None of the above

4. Which of the following statements regarding streams is/are **TRUE**?
   - **I.** Using `'\n'` to represent a newline may be more efficient than using `std::endl`, since the latter also flushes the stream buffer.
   - **II.** Input redirection can be used to direct the contents of an input file into the `std::cin` standard input stream.
   - **III.** A `std::stringstream` allows you to read and write data into a string as if it were a stream.

   A) I only
   B) II only
   C) III only
   D) II and III only
   E) I, II, and III

5. You are given the following file, `numbers.txt`:

```
13
95
22
51
18
82
```

   (a) Write a short program that takes in this file as standard input via input redirection and prints out the sum of these numbers to standard output.
   (b) Suppose you compiled this program into an executable named `exec`. Write out the command you would type in the terminal if you wanted to read in the numbers in `numbers.txt` as input and print out the sum in a text file named `sum.txt`.

6. Consider the following code, which reads input within a `while` loop:

```
1  int main() {
2    std::string word;
3    while (true) {
4      if (std::cin.good()) {
5        std::cin >> word;
6      } // if
7      else {
8        std::cout << "reached end of file" << std::endl;
9      } // else
10   } // while
11 } // main()
```

   What could go wrong with this implementation, and how could this issue be fixed?

7. Consider the following code:

```
1  int main() {
2    int32_t num_lines = 0;
3    std::string line;
4    std::cin >> num_lines;
5    for (int32_t i = 0; i < num_lines; ++i) {
6      std::getline(std::cin, line);
7      std::cout << line << '\n';
8    } // for i
9  } // main()
```

   Suppose this code were run on the following input file, where "•" represents a space, and "¶" represents a new line:

```
••••• 5 ••••• ¶
apple ¶
banana ¶
cherry ¶
dragonfruit ¶
eggfruit ¶
```

   Does the code work as intended? If not, make a fix to the original code so that it works properly. You may assume that all input files that are run with this program follow the same structure as the file detailed above.

Chapter 2 Exercise Solutions

1. **The correct answer is (A).** The extraction operator `>>` ignores leading whitespace, but the `std::getline()` does not (instead, `std::getline()` reads all characters up to the next delimitation character, which is `'\n'` by default).

2. **The correct answer is (B).** When you read something from the standard input stream `std::cin` using either `operator>>` or `std::getline()`, you also extract it from the stream. There are two places where `std::cin` is being read: the extraction operator in the `while` loop on line 5, and the call to `std::getline()` on line 6. `operator>>` ignores leading whitespace, while `std::getline()` consumes everything up to the next newline character. With this information, we will walk through the input file and observe what happens when it is redirected via standard input:
   - First, we use `operator>>` to read in the first value from the input stream. This reads the number `17` into `num`. After this first extraction, the stream now looks like this:

     ```
     num = 17
     ```

     ```
     line = ""
     ```

     `input.txt`

     ```
     • 22 •••• 14 ¶ 19 ¶ ¶ • ¶ •• ¶ 10 ¶ 13 ¶ 15 •• 27 ••• 12 •• 20 ¶
     ```

   - Next, `std::getline()` extracts everything up to the next newline, and then discards the newline character.

     ```
     num = 17
     ```

     ```
     line = "• 22 •••• 14"
     ```

     `input.txt`

     ```
     19 ¶ ¶ • ¶ •• ¶ 10 ¶ 13 ¶ 15 •• 27 ••• 12 •• 20 ¶
     ```

   - `counter` is incremented to `1`.
   - The next call to `operator>>` reads the next number into `num`, or `19`.

     ```
     num = 19
     ```

     ```
     line = "• 22 •••• 14"
     ```

     `input.txt`

     ```
     ¶ ¶ • ¶ •• ¶ 10 ¶ 13 ¶ 15 •• 27 ••• 12 •• 20 ¶
     ```

   - `std::getline()` then reads in all characters up to the next newline, and then discards this newline. However, there is nothing before the next newline, so `std::getline()` does not extract anything.

     ```
     num = 19
     ```

     ```
     line = ""
     ```

     `input.txt`

     ```
     ¶ • ¶ •• ¶ 10 ¶ 13 ¶ 15 •• 27 ••• 12 •• 20 ¶
     ```

   - `counter` is incremented to `2`.
   - The next call to `operator>>` reads the next number into `num`, or `10` (all whitespace is ignored up to this point).

     ```
     num = 10
     ```

     ```
     line = ""
     ```

     `input.txt`

     ```
     ¶ 13 ¶ 15 •• 27 ••• 12 •• 20 ¶
     ```

- `std::getline()` then reads in all characters up to the next newline, and then discards this newline. However, there is nothing before the next newline, so `std::getline()` does not extract anything.

```
num = 10
```

```
line = ""
```

`input.txt`

```
13 ¶ 15 •• 27 ••• 12 •• 20 ¶
```

- `counter` is incremented to `3`.
- The next call to `operator>>` reads the next number into `num`, or `13`.

```
num = 10
```

```
line = ""
```

`input.txt`

```
¶ 15 •• 27 ••• 12 •• 20 ¶
```

- `std::getline()` then reads in all characters up to the next newline, and then discards this newline. However, there is nothing before the next newline, so `std::getline()` does not extract anything.

```
num = 10
```

```
line = ""
```

`input.txt`

```
15 •• 27 ••• 12 •• 20 ¶
```

- `counter` is incremented to `4`.
- The next call to `operator>>` reads the next number into `num`, or `15`.

```
num = 15
```

```
line = ""
```

`input.txt`

```
•• 27 ••• 12 •• 20 ¶
```

- `std::getline()` then reads in all characters up to the next newline, and then discards this newline.

```
num = 15
```

```
line = "•• 27 ••• 12 •• 20 "
```

`input.txt`

```
```

- `counter` is incremented to 5.
- The stream is now empty, so `std::cin >> num` evaluates to `false` and the `while` loop terminates. `num` therefore ends with a value of `15`, and `counter` ends with a value of `5`.

3. **The correct answer is (B).** The `std::istream` and `std::ostream` serve as the base classes of input and output streams, so if you want a function to accept any kind of input or output stream, you can use these types. Additionally, we want to write directly to the stream we pass into the function instead of making a copy, so we will pass the stream by reference instead of by value.

4. **The correct answer is (E).** All of the statements are true. Statement I is true because using `std::endl` also flushes the stream buffer in addition to inserting a newline into a stream. Statement II is true because that is the definition of input redirection: we can treat the contents of an input file as if it were a stream. Statement III is true because `std::stringstream` provides stream-like functionality for strings.

5. (a) Since we are reading in this input file via input redirection, we can simply read the contents of this file using `std::cin`, and then write the output using `std::cout`. To identify when we are done reading from the input file, we can extract from `std::cin` within a `while` loop, which evaluates to `false` once there is no more input left to read. This is shown in the code below:

```
1  int main() {
2    int32_t num = 0, sum = 0;
3    while (std::cin >> num) {
4      sum += num;
5    } // while
6    std::cout << sum << '\n';
7  } // main()
```

   (b) To redirect input from a file, you can simply use the input redirection operator <. To redirect output into a file, you can use the output redirection operator >. In this case, the command would be:

```
./exec < numbers.txt > sum.txt
```

6. The issue with this implementation is that it is using `std::cin.good()` within a `while` loop. Although methods such as `.good()`, `.eof()`, `.bad()`, and `.fail()` can be used to determine whether an input stream is a readable state, it should not be put in the condition of the loop. This is because these methods return `false` only *after* an extraction from the stream has already failed, and the loop would still run one more iteration after the stream becomes invalid. This can be fixed by extracting from the stream directly in the `while` condition — i.e., `while(std::cin >> word)`.

7. This code does not work as intended because the first `std::getline()` ends up reading in the five spaces directly after the number `5` on the first line of the example input file (remember that `std::getline()` does not ignore whitespace and reads everything up to the next newline, and the extraction from `std::cin` on line 4 would only extract from the string up to the `5`). There are two ways to address this. One way is to invoke an additional `std::getline()` call after the `std::cin` extraction on line 4 to clear out any additional whitespace before the next newline in the stream. This is shown below:

```
1  int main() {
2    int32_t num_lines = 0;
3    std::string line;
4    std::cin >> num_lines;
5    std::getline(std::cin, line);  // clear out everything before next line
6    for (int32_t i = 0; i < num_lines; ++i) {
7      std::getline(std::cin, line);
8      std::cout << line << '\n';
9    } // for i
10 } // main()
```

   Another option is to use `std::cin.ignore()`, which can be used to ignore everything up to the next newline character after the first extraction from `std::cin` on line 4. This is shown below:

```
1  int main() {
2    int32_t num_lines = 0;
3    std::string line;
4    std::cin >> num_lines;
5    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
6    for (int32_t i = 0; i < num_lines; ++i) {
7      std::getline(std::cin, line);
8      std::cout << line << '\n';
9    } // for i
10 } // main()
```