Introduction

Data input/output in C++ is done via stream classes defined in the Standard Template Library (STL). The `<iostream>` library provides all the functionalities for you to read from standard input and write to standard output. In this course, all I/O will be done via standard input and standard output, so it is wise to use the `<iostream>` library in every program that you will be writing. In addition, the `<sstream>` library deals with I/O using the STL’s `string` class. This library is useful in processing a string with a variety of data. To use these features, you should start any programs with the following header:

```
#include <iostream>
#include <sstream>
using namespace std;
```

Note that `<sstream>` automatically includes the `<string>` class. Also, identifying the namespace `std` is important, or you will need to prefix all standard library class, methods, and variables with a “std::”.

Standard Input

<iostream> has the global variable `cin`, which defines the standard input stream. The most common use of `cin` is its extraction operator `">>"`. This operator is overloaded for all standard types, including int, double, and string.

```
int n;
double f;
string s;
cin >> n >> f >> s;
```

Example 1. Reading an integer, then a double, then a string from standard input.

The extraction operator also does some input processing. Using this operator, the input stream will read and discard any whitespaces, then parse data until it reaches the next whitespace. For example, when the program in Example 1 is fed the following 3 input files, the results are the same:

```
10
4.5_abc
^  
```

```
10_______4.5__________
____abc____^ 
```

Example 2. Three different input files where Example 1’s program will read the same information for n, f, and s. Spaces are replaced with ‘_’ for clarity.

This is a very useful property of C++’s streams. You do not have to worry about spaces separating input data. On the other hand, this can be troublesome in some instances. For example, suppose you want to read in a person’s name:

```
string name;
cin >> name;
```

In this program, if the input has embedded whitespace, as in “John Doe”, then the input will only set `name=”John”`, ignoring the last name. To fix this problem, we need to read the input line by line.
To read input line by line, C++ provides the global `getline` function call.

```
istream& getline (istream& is, string &str, char delim = '\n');
```

This `getline` function reads from the input stream `is`, which in our case will be `cin`, until one of the followings occurs:
(a) The end-of-file is reached,
(b) The maximum number of characters to be fit into a string is read,
(c) The delimiter character `delim` is read.

```
string names[100];
int i = 0;
while ( getline( cin, names[i] ) ) {
    i++;
}
```

<table>
<thead>
<tr>
<th>John Doe</th>
<th>Mary Martin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 3. Reads from standard input line by line, storing each line in the `names` array.

**Note:** The `getline` function call will read in the delimiter and discard it. So in the example above, `names[0]="John Doe",` and not "John Doe\n".

**Standard Output**

The standard output stream in C++ is `cout`. In parallel to `cin`'s extraction operator, output streams in C++ define the **insertion operator** `<<`. Similarly, the insertion operator is overloaded for all standard data types, and can be used in sequences:

```
int age = 21;
string name = "John Doe";
cout << name << " is " << age << " years old." << endl;
```

This will print to standard output the following string:

```
John Doe is 21 years old.
```

The special variable `endl` is defined in `<iostream>`, standing for “end-line”. All it does is to print a `\n` character, then flush the output stream. This is helpful when printing to a console, where the output you are waiting for should be printed immediately, and not buffered.

**A special note in formatting**

Some simple output formatting is available with C++’s streams. For more complex formatting, C++ provides the `<iomanip>` class. However, these formatting are complicated and hard to use. Thus, we recommend using **printf** from C for all formatting. Please read the manual pages for more information about output formatting in C++. 
Input/Output using Strings

It is easy to read and write with strings in C++, once you have mastered standard I/O. This is because the STL provides stream classes for common objects like strings and files. Once a string or a file is converted to a stream, all input/output methods on that stream are the same as those of cin/cout.

```cpp
#include <iostream>
#include <sstream>  // defines the stringstream class
using namespace std;

int main()
{
    int age;
    string line, name;
    while ( getline( cin, line ) ) {
        stringstream strin( line );
        strin >> age;
        getline( strin, name );
    }
    return 0;
}
```

Example 4. Using `stringstream` to parse input.

The code above reads from standard input line by line, where each line starts with the person’s age, followed the person’s full name. Line 10 defines the `stringstream` object `strin`. The constructor takes one parameter: the string to be processed. Line 11 uses the extraction operator to read the age of the person. Line 12 will then process the rest of the line using `getline`.

**Question:** A small bug exists in the above code. Can you find it?

You can also use `stringstream` to write to strings, then use the `str()` member function to get the string of the current stream. The following is an easy way to convert any number to a string.

```cpp
string intToString( int n )
{
    stringstream strout;
    strout << n;
    return strout.str();
}
```

This creates an empty stream `strout`, writes an integer to it using the overloaded extraction operator, and returns the string written using `str()`. 