Tips and tricks for using C++ I/O

Table of Contents

There are three header files to include when using C++ I/O

`#include<iostream>`
: Include this file whenever using C++ I/O

`#include<iomanip>`
: This file must be included if any C++ manipulators will be used. If you don't know what a manipulator is, don't worry. Just include this file along with `iostream` and you can't go wrong

`#include<fstream>`
: Include this file whenever working with files.

By default, leading whitespace (carriage returns, tabs, spaces) is ignored by `cin`.

Given:

```
int i;
float fl;
cin >> fl;
cin >> i;
```

And you type: `3.14<return>42<return>`

1. 3.14 is read into `fl` . The carriage return (newline) following the 3.14 is still sitting on the input buffer.
2. Since cin ignores whitespace, the first return is "eaten" by `cin >> i` . Then the integer `42` is read into `i` and the second return is left on the input buffer.

`cin.getline()` can run into problems when used with `cin >> var`.

- `getline` can be provided a third argument--a "stop" character. This character ends getline's input. The character is eaten and the string is terminated. Example: `cin.getline(str, 100, '|')`
- If `cin.getline()` is not provided a "stop" character as a third argument, it will stop when it reaches a newline.

  Given:

```
float fl;
cin >> fl;
char str[101]
cin.getline(str, 101);
```

1. And you type: `3.14<return>`
2. 3.14 is read into `fl` . The newline following the 3.14 is still sitting on the input buffer.
3. `cin.getline(str, 101)` immediately processes the newline that is still on the input buffer. `str` becomes an empty string.
4. The illusion is that the application "skipped" the `cin.getline()` statement.

The solution is to add `cin.ignore();` immediately after the first `cin` statement. This will grab a character off of the input buffer (in this case, newline) and discard it.

`cin.ignore()` has 3 forms:

1. No arguments: A single character is taken from the input buffer and discarded:
   `cin.ignore(); //discard 1 character`
2. One argument: The number of characters specified are taken from the input buffer and discarded:
   `cin.ignore(33); //discard 33 characters`
3. Two arguments: discard the number of characters specified, or discard characters up to and including the specified delimiter (whichever comes first):
   `cin.ignore(26, '\n'); //ignore 26 characters or to a newline, whichever comes first`

Reading in numbers directly is problematic

- If cin is presented with input it cannot process, cin goes into a "safe" state
- The input it cannot process is left on the input stream.
- All input will be ignored by cin until the "safe" state is cleared: cin.clear()
- A routine that reads a number directly should:
  1. Read in the number
  2. Check to see that the input stream is still valid
  3. If the input stream is not good (!cin)

1. Call cin.clear() to take the stream out of the "safe" state.
2. Remove from the stream the input that caused the problem: cin.ignore(...)
3. Get the input again if appropriate or otherwise handle the error

Inputing numbers directly, version 1:

```
#include <climits> //for INT_MAX
float fl;
int bad_input;
do{
  bad_input=0;
  cin >> fl;
  if(!cin)
  {
    bad_input=1;
    cin.clear();
    cin.ignore(INT_MAX,'\n');
  }

}while(bad_input);
```

Inputing numbers directly, version 2:

```
#include <climits> //for INT_MAX
float fl;
while(!(cin >> fl))
{
  cin.clear();
  cin.ignore(INT_MAX,'\n');
}
```

A note on limits. In C++, rather than using INT_MAX, I should have used:

```
#include
...
cin.ignore(numeric_limits::max(), '\n');
```

As of this writing, g++ does not support the limits header file, so the c-style method of determining the maximum integer is used.

Using getline to input numbers is a more robust alternate to reading numbers directly

```
#include <cstdlib>
...
int i;
float fl;
char temp[100];

cin.getline(temp, 100);
fl=atof(temp);
cin.getline(temp, 100);
i=atoi(temp);
```

- getline will read both strings and numbers without going into a "safe" state.
- Include cstdlib to use the converter functions: ascii-to-integer (atoi), ascii-to-long (atol), and ascii-to-float (atof).

Once a file is opened, it may be used exactly as cin is used.

```
ifstream someVarName("data.txt");
float fl;
char temp[100];
someVarName.getline(temp, 100);
fl=atof(temp);
int i;
someVarName >> i;
```

When reading an entire file, embed the file input inside of the loop condition

```
ifstream inf("data.txt");
char temp[100];
while(!inf.getline(temp, 100).eof())
{
  //process the line
}
```

- the loop will exit once the end of the file is reached

Getline can be told to stop grabbing input at any designated character

```
char temp[100];
cin.getline(temp, 100, '|');
```

- If only two arguments are supplied to getline, getline will stop at the end of the line (at the newline character).
- If three arguments are supplied to getline, getline will stop at the character designated by the third argument.
- The stop character is not copied to the string.
- The stop character is "eaten" (removed from the input stream).

Delimited files can easily be read using a while loop and getline.

Given data file:

```
John|83|52.2
swimming|Jefferson
Jane|26|10.09
```

```
  sprinting|San Marin
Process using:

ifstream inf("data.txt");
char name[30];
while(!inf.getline(name, 30, '|').eof())
{
  Athlete* ap;
  char jersey_number[10];
  char best_time[10];
  char sport[40];
  char high_school[40];
  inf.getline(jersey_number, 10, '|'); #read thru pipe
  inf.getline(best_time, 10);          #read thru newline
  inf.getline(sport, 40, '|');         #read thru pipe
  inf.getline(high_school, 40);        #read thru newline
  ap = new Athlete(name, atoi(number), atof(best_time), sport,
high_school);
  //do something with ap

}
```

- In a delimited file, only the first field should be in the while loop
- For each field: If the field is the last field in the line or the only field in the line, be sure that getline stops at a newline and not some other delimiter

Using C++-style strings

All of the previous examples have assumed that C-style strings (null-terminated character arrays) were being used. C++ provides a string class that, when combined with a particular "getline" function, can dynamically resize to accomodate user input. In general, C++ strings are preferred over C strings.

Here is the same code shown above, this time using C++ strings:

```
#include <string>
ifstream inf("data.txt");
string name;
while(!getline(inf, name, '|').eof())
{
  Athlete* ap;
  string jersey_number;
  string best_time;
  string sport;
  string high_school;
  getline(inf, jersey_number, '|'); #read thru pipe
  getline(inf, best_time);          #read thru newline
  getline(inf, sport, '|');         #read thru pipe
  getline(inf, high_school);        #read thru newline
  ap = new Athlete(name, atoi(number.c_str()),
       atof(best_time.c_str()), sport, high_school);
```

```
  //do something with ap

}
```

How to prepare the output stream to print fixed precision numbers (3.40 instead of 3.4)

```
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
cout.precision(2);
```

How to set the width of a printing field

```
Given: int one=4, two=44;

  cout << one << endl;
  //output:  "4"

  cout << setw(2) << one << endl;
  //output:  " 4"

  cout.fill('X');
  cout << setw(2) << one << endl;
  //output:  "X4"

  cout.fill('X');
  cout << setw(2) << two << endl;
  //output:  "44"
```

- The default fill character is a space.
- A common fill character when printing numbers is zero "0".