

## 18.2 — Input with istream

BY ALEX ON MARCH 4TH, 2008 | LAST MODIFIED BY ALEX ON NOVEMBER 26TH, 2016

The `istream` library is fairly complex -- so we will not be able to cover it in its entirety in these tutorials. However, we will show you the most commonly used functionality. In this section, we will look at various aspects of the input class (`istream`).

Note: All of the I/O functionality in this lesson lives in the `std` namespace. That means all I/O objects and functions either have to be prefixed with “`std::`”, or the “`using namespace std;`” statement has to be used.

### The extraction operator

As seen in many lessons now, we can use the extraction operator (`>>`) to read information from an input stream. C++ has predefined extraction operations for all of the built-in data types, and you’ve already seen how you can **overload the extraction operator** for your own classes.

When reading strings, one common problem with the extraction operator is how to keep the input from overflowing your buffer. Given the following example:

```
1 char buf[10];
2 cin >> buf;
```

what happens if the user enters 18 characters? The buffer overflows, and bad stuff happens. Generally speaking, it’s a bad idea to make any assumption about how many characters your user will enter.

One way to handle this problem is through use of manipulators. A **manipulator** is an object that is used to modify a stream when applied with the extraction (`>>`) or insertion (`<<`) operators. One manipulator you have already worked with extensively is “`endl`”, which both prints a newline character and flushes any buffered output. C++ provides a manipulator known as **`setw`** (in the `iomanip.h` header) that can be used to limit the number of characters read in from a stream. To use `setw()`, simply provide the maximum number of characters to read as a parameter, and insert it into your input statement like such:

```
1 #include <iomanip.h>
2 char buf[10];
3 cin >> setw(10) >> buf;
```

This program will now only read the first 9 characters out of the stream (leaving room for a terminator). Any remaining characters will be left in the stream until the next extraction.

### Extraction and whitespace

The one thing that we have omitted to mention so far is that the extraction operator works with “formatted” data -- that is, it skips whitespace (blanks, tabs, and newlines).

Take a look at the following program:

```
1 int main()
2 {
3     char ch;
4     while (cin >> ch)
5         cout << ch;
6
7     return 0;
8 }
```

When the user inputs the following:

Hello my name is Alex

The extraction operator skips the spaces and the newline. Consequently, the output is:

HellomynameisAlex

Oftentimes, you'll want to get user input but not discard whitespace. To do this, the `istream` class provides many functions that can be used for this purpose.

One of the most useful is the **`get()`** function, which simply gets a character from the input stream. Here's the same program as above using `get()`:

```
1  int main()
2  {
3      char ch;
4      while (cin.get(ch))
5          cout << ch;
6
7      return 0;
8  }
```

Now when we use the input:

Hello my name is Alex

The output is:

Hello my name is Alex

`get()` also has a string version that takes a maximum number of characters to read:

```
1  int main()
2  {
3      char strBuf[11];
4      cin.get(strBuf, 11);
5      cout << strBuf << endl;
6
7      return 0;
8  }
```

If we input:

Hello my name is Alex

The output is:

Hello my n

Note that we only read the first 10 characters (we had to leave one character for a terminator). The remaining characters were left in the input stream.

One important thing to note about `get()` is that it does not read in a newline character! This can cause some unexpected results:

```
1  int main()
2  {
3      char strBuf[11];
4      // Read up to 10 characters
5      cin.get(strBuf, 11);
6      cout << strBuf << endl;
7
8      // Read up to 10 more characters
9      cin.get(strBuf, 11);
10     cout << strBuf << endl;
11     return 0;
12 }
```

If the user enters:

Hello!

The program will print:

Hello!

and then terminate! Why didn't it ask for 10 more characters? The answer is because the first `get()` read up to the newline and then stopped. The second `get()` saw there was still input in the `cin` stream and tried to read it. But the first character was the newline, so it stopped immediately.

Consequently, there is another function called **`getline()`** that works exactly like `get()` but reads the newline as well.

```
1  int main()
2  {
3      char strBuf[11];
4      // Read up to 10 characters
5      cin.getline(strBuf, 11);
6      cout << strBuf << endl;
7
8      // Read up to 10 more characters
9      cin.getline(strBuf, 11);
10     cout << strBuf << endl;
11     return 0;
12 }
```

This code will perform as you expect, even if the user enters a string with a newline in it.

If you need to know how many character were extracted by the last call of `getline()`, use **`gcount()`**:

```
1  int main()
2  {
3      char strBuf[100];
4      cin.getline(strBuf, 100);
5      cout << strBuf << endl;
6      cout << cin.gcount() << " characters were read" << endl;
7
8      return 0;
9  }
```

### A special version of `getline()` for `std::string`

There is a special version of `getline()` that lives outside the `istream` class that is used for reading in variables of type `std::string`. This special version is not a member of either `ostream` nor `istream`, and is included in the `string` header. Here is an example of its use:

```
1  #include <string>
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
7      string strBuf;
8      getline(cin, strBuf);
9      cout << strBuf << endl;
10
11     return 0;
12 }
```

### A few more useful `istream` functions

There are a few more useful input functions that you might want to make use of:

**`ignore()`** discards the first character in the stream.

**`ignore(int nCount)`** discards the first `nCount` characters.

**`peek()`** allows you to read a character from the stream without removing it from the stream.

**`unget()`** returns the last character read back into the stream so it can be read again by the next call.

**`putback(char ch)`** allows you to put a character of your choice back into the stream to be read by the next call.

istream contains many other functions and variants of the above mentioned functions that may be useful, depending on what you need to do. However, those topics are really more suited for a tutorial or book focusing on the standard library (such as the excellent [“The C++ Standard Library”](#) by Nicolai M. Josuttis).



[18.3 -- Output with ostream and ios](#)



[Index](#)



[18.1 -- Input and output \(I/O\) streams](#)

Share this:



[C++ TUTORIAL](#) | [PRINT THIS POST](#)

## 52 comments to 18.2 — Input with istream



prince

[June 18, 2018 at 10:38 am](#) · [Reply](#)

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4  int main()
5  {char ch_1[5], ch_2[6];
6     cin>>setw(5)>>ch_1;
7     cin>>setw(6)>>ch_2;
8
9     cout<<ch_1<<endl;
10    cout<<ch_2;
11
12    return 0;
13
14 }
```

why it is not asking for input twice

whats wrong with this?

if i have used only 'cin' then it will prompt twice what is the difference in both them .as told above setw()

is just a manipulator so it should not change 'cin' working except assigning char arrays (ch\_1,ch\_2) the required characters.



nascardriver

[June 20, 2018 at 3:24 am](#) · [Reply](#)

Hi prince!

When you enter a string that's longer than 5 characters the remaining characters will remain in the input buffer and will be extracted at the second call to @std::cin.operator>>.

You need to clear the input buffer after extraction to prevent this.

Also,

\* Don't use "using namespace"