28.2 — Input with istream

ALEX[1] DECEMBER 23, 2024

The iostream 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).

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 (https://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators/)[2] 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 std::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 is "std::endl", which both prints a newline character and flushes any buffered output. C++ provides a manipulator known as **setw** (in the iomanip 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>
2 char buf[10]{};
3 std::cin >> std::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

As a reminder, the extraction operator skips whitespace (blanks, tabs, and newlines).

Take a look at the following program:

```
int main()
{
    char ch{};
    while (std::cin >> ch)
        std::cout << ch;

    return 0;
}
```

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():

```
int main()
{
    char ch{};
    while (std::cin.get(ch))
        std::cout << ch;

    return 0;
}
```

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:

```
int main()
{
    char strBuf[11]{};
    std::cin.get(strBuf, 11);
    std::cout << strBuf << '\n';

    return 0;
}
```

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:

```
int main()
{
    char strBuf[11]{};
    // Read up to 10 characters
    std::cin.get(strBuf, 11);
    std::cout << strBuf << '\n';

    // Read up to 10 more characters
    std::cin.get(strBuf, 11);
    std::cout << strBuf << '\n';
    return 0;
}
```

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 similarly to get(), but will extract (and discard) the delimiter.

```
int main()
{
    char strBuf[11]{};
    // Read up to 10 characters
    std::cin.getline(strBuf, 11);
    std::cout << strBuf << '\n';

    // Read up to 10 more characters
    std::cin.getline(strBuf, 11);
    std::cout << strBuf << '\n';
    return 0;
}
```

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()**:

```
int main()
{
    char strBuf[100]{};
    std::cin.getline(strBuf, 100);
    std::cout << strBuf << '\n';
    std::cout << std::cin.gcount() << " characters were read" << '\n';

    return 0;
}
```

gcount() includes any extracted and discarded delimiters.

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 or istream, and is included in the string header. Here is an example of its use:

```
#include <string>
#include <iostream>

int main()
{
    std::string strBuf{};
    std::getline(std::cin, strBuf);
    std::cout << strBuf << '\n';

    return 0;
}
```

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. You can find these on a reference site such as https://en.cppreference.com/w/cpp/io/basic_istream (https://en.cppreference.com/w/cpp/io/basic_istream)[3].

Next lesson
28.3 Output with ostream and ios

4

Back to table of contents

5

Previous lesson
28.1 Input and output (I/O) streams

6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

101 COMMENTS

Newest

kalypto
May 28, 2025 11:17 am PDT

The code example above
"and then terminate! Why didn't it ask for 10 more characters?"
is not correct. The input should consist of at least two lines for the example to make sense but it is only one.

0 Reply

axtimeo
May 21, 2025 2:45 pm PDT

ig i need to learn lot...LOL!!!

Last edited 1 month ago by axtimeo

0 Reply

axtimeo
May 21, 2025 2:42 pm PDT

Hi everyone

0 Reply

Nidhi Gupta
February 23, 2025 9:59 pm PST

The C++ extraction operator (>>) is used primarily for reading from input streams and gets along nicely with built-in types, but can be overloaded for user-defined classes as well. One of the problems when reading strings with >> is the potential to overflow fixed-size buffers—this can be circumvented by using manipulators like std::setw to limit the number of characters read. By default, the extraction operator ignores whitespace, i.e., spaces and newlines are ignored unless other functions are called. Functions such as get() allow you to read single characters (or a specified number of characters) without ignoring whitespace, while getline() reads the whole line and ignores the delimiter, ensuring that the input is handled as desired. Other assistant operations like ignore(), peek(), and unget() also contribute to input processing by managing stream content and behavior.

1 Reply

Nidhi Gupta
February 23, 2025 9:58 pm PST

this is nice to know!

0 Reply

Ashley Hawkins
December 14, 2024 6:41 am PST

Is referring to basic_istream's get() method as std::get() a typo? I think std::get is a separate function for getting an element of an aggregate or a variant.

Last edited 6 months ago by Ashley Hawkins

1 Reply

Alex Author

Reply to Ashley Hawkins [9] December 23, 2024 9:39 pm PST

Yup, just a typo. Fixed.

0 Reply

ta le

March 30, 2024 3:57 pm PDT

moreover we can use setw() from iomanip header for creating white space too.

1 Reply

Strain

January 25, 2024 10:43 pm PST

The third example's output is a bit... suspicious.

16 Reply

eklektos

December 9, 2023 6:21 am PST

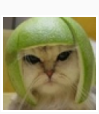
Consequently, there is another function called getline() that works exactly like get() but reads the newline as well.

```
int main()
{
char strBuf[11];
// Read up to 10 characters
std::cin.getline(strBuf, 11);
std::cout << strBuf << '\n';

// Read up to 10 more characters
std::cin.getline(strBuf, 11);
std::cout << strBuf << '\n';
return 0;
}
```

It doesn't read the newline character, the newline character is the default delimiter, when I press the newline, program stops reading the input.

0 Reply



Alex Author

Reply to _eklektos_ [10] December 11, 2023 10:31 am PST

Clarified in the lesson that the delimiter is extracted and discarded.

1 Reply

Jestin PJ

September 10, 2023 11:46 pm PDT

```cpp
#include <iostream>

int main() {
    char strBuf[20];
    int index = sizeof(strBuf) - 1; // Adjust the index to account for the null
terminator

    for (int i = 0; i < index; i++) {
        if (!std::cin.get(strBuf[i]) || strBuf[i] == '\n' || strBuf[i] == '\0') {
            // Exit the loop if EOF, newline, or null terminator is encountered
            strBuf[i] = '\0'; // Null-terminate the string
            break;
        }
    }

    std::cout << strBuf << '\n';

    return 0;
}
```

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators/
3. https://en.cppreference.com/w/cpp/io/basic_istream
4. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/input-and-output-io-streams/
7. https://www.learncpp.com/input-with-istream/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/input-with-istream/#comment-605234
10. https://www.learncpp.com/cpp-tutorial/input-with-istream/#comment-590706
11. https://g.ezoic.net/privacy/learncpp.com