1.5 — Introduction to iostream: cout, cin, and endl

In this lesson, we'll talk more about std::cout, which we used in our *Hello world!* program to output the text *Hello world!* to the console. We'll also explore how to get input from the user, which we will use to make our programs more interactive.

The input/output library

The **input/output library** (io library) is part of the C++ standard library that deals with basic input and output. We'll use the functionality in this library to get input from the keyboard and output data to the console. The *io* part of *iostream* stands for *input/output*.

To use the functionality defined within the *iostream* library, we need to include the *iostream* header at the top of any code file that uses the content defined in *iostream*, like so:

```
#include <iostream>

// rest of code that uses iostream functionality here
```

std::cout

The *iostream* library contains a few predefined variables for us to use. One of the most useful is **std::cout**, which allows us to send data to the console to be printed as text. *cout* stands for "character output".

As a reminder, here's our *Hello world* program:

```
#include <iostream> // for std::cout

int main()
{
    std::cout << "Hello world!"; // print Hello world! to console

    return 0;
}
```

In this program, we have included *iostream* so that we have access to *std::cout*. Inside our *main* function, we use *std::cout*, along with the **insertion operator (<<)**, to send the text *Hello world!* to the console to be printed.

std::cout can not only print text, it can also print numbers:

```
#include <iostream> // for std::cout

int main()
{
    std::cout << 4; // print 4 to console

    return 0;
}
```

This produces the result:

```
4
```

It can also be used to print the value of variables:

```
#include <iostream> // for std::cout

int main()
{
    int x{ 5 }; // define integer variable x, initialized with value 5
    std::cout << x; // print value of x (5) to console
    return 0;
}
```

This produces the result:

```
5
```

To print more than one thing on the same line, the insertion operator (<<) can be used multiple times in a single statement to concatenate (link together) multiple pieces of output. For example:

```
#include <iostream> // for std::cout

int main()
{
    std::cout << "Hello" << " world!";
    return 0;
}
```

This program prints:

```
Hello world!
```

Here's another example where we print both text and the value of a variable in the same statement:

```
#include <iostream> // for std::cout

int main()
{
    int x{ 5 };
    std::cout << "x is equal to: " << x;
    return 0;
}
```

This program prints:

```
x is equal to: 5
```

Related content

We discuss what the *std::* prefix actually does in lesson 2.9 -- Naming collisions and an introduction to namespaces.

Using `std::endl` to output a newline

What would you expect this program to print?

```
#include <iostream> // for std::cout

int main()
{
    std::cout << "Hi!";
    std::cout << "My name is Alex.";
    return 0;
}
```

You might be surprised at the result:

```
Hi!My name is Alex.
```

Separate output statements don't result in separate lines of output on the console.

If we want to print separate lines of output to the console, we need to tell the console to move the cursor to the next line. We can do that by outputting a newline. A **newline** is an OS-specific character or sequence of characters that moves the cursor to the start of the next line.

One way to output a newline is to output `std::endl` (which stands for "end line"):

```
#include <iostream> // for std::cout and std::endl

int main()
{
    std::cout << "Hi!" << std::endl; // std::endl will cause the cursor to move to
the next line
    std::cout << "My name is Alex." << std::endl;

    return 0;
}
```

This prints:

```
Hi!
My name is Alex.
```

Tip

In the above program, the second `std::endl` isn't technically necessary, since the program ends immediately afterward. However, it serves a few useful purposes.

First, it helps indicate that the line of output is a "complete thought" (as opposed to partial output that is completed somewhere later in the code). In this sense, it functions similarly to using a period in standard English.

Second, it positions the cursor on the next line, so that if we later add additional lines of output (e.g. have the program say "bye!"), those lines will appear where we expect (rather than appended to the prior line of output).

Third, after running an executable from the command line, some operating systems do not output a new line before showing the command prompt again. If our program does not end with the cursor on a new line, the command prompt may appear appended to the prior line of output, rather than at the start of a new line as the user would expect.

Best practice

Output a newline whenever a line of output is complete.

std::cout is buffered

Consider a rollercoaster ride at your favorite amusement park. Passengers show up (at some variable rate) and get in line. Periodically, a train arrives and boards passengers (up to the maximum capacity of the train). When the train is full, or when enough time has passed, the train departs with a batch of passengers, and the ride commences. Any passengers unable to board the current train wait for the next one.

This analogy is similar to how output sent to `std::cout` is typically processed in C++. Statements in our program request that output be sent to the console. However, that output is typically not sent to the console immediately. Instead, the requested output "gets in line", and is stored in a region of memory set aside to collect such requests (called a **buffer**). Periodically, the buffer is **flushed**, meaning all of the data collected in the buffer is transferred to its destination (in this case, the console).

Author's note

To use another analogy, flushing a buffer is kind of like flushing a toilet. All of your collected "output" is transferred to … wherever it goes next. Eew.

This also means that if your program crashes, aborts, or is paused (e.g. for debugging purposes) before the buffer is flushed, any output still waiting in the buffer will not be displayed.

Key insight

The opposite of buffered output is unbuffered output. With unbuffered output, each individual output request is sent directly to the output device.

Writing data to a buffer is typically fast, whereas transferring a batch of data to an output device is comparatively slow. Buffering can significantly increase performance by batching multiple output requests together to minimize the number of times output has to be sent to the output device.

`std::endl` vs `\n`

Using `std::endl` is often inefficient, as it actually does two jobs: it outputs a newline (moving the cursor to the next line of the console), and it flushes the buffer (which is slow). If we output multiple lines of text ending with `std::endl`, we will get multiple flushes, which is slow and probably unnecessary.

When outputting text to the console, we typically don't need to explicitly flush the buffer ourselves. C++'s output system is designed to self-flush periodically, and it's both simpler and more efficient to let it flush itself.

To output a newline without flushing the output buffer, we use `\n` (inside either single or double quotes), which is a special symbol that the compiler translates into a newline. `\n` moves the cursor to the next line of the console without causing a flush, so it will typically perform better. `\n` is also more concise to type and can be embedded into existing double-quoted text.

Here's an example that uses `\n` in a few different ways:

```
#include <iostream> // for std::cout

int main()
{
    int x{ 5 };
    std::cout << "x is equal to: " << x << '\n'; // single quoted (by itself)
(conventional)
    std::cout << "Yep." << "\n";                 // double quoted (by itself)
(unconventional but okay)
    std::cout << "And that's all, folks!\n";     // between double quotes in existing
text (conventional)
    return 0;
}
```

This prints:

```
x is equal to: 5
Yep.
And that's all, folks!
```

When `\n` is not being embedded into an existing line of double-quoted text (e.g. `"hello\n"`), it is conventionally single quoted (`'\n'`).

For advanced readers

In C++, we use single quotes to represent single characters (such as `'a'` or `'$'`), and double-quotes to represent text (zero or more characters). Even though '\n' is represented in source code as two symbols, it is treated by the compiler as a single character, and thus is conventionally single quoted (unless embedded into existing double-quoted text).

We discuss this more in lesson 4.11 -- Chars.

Author's note

Although unconventional, we believe it's fine to use (or even prefer) double quoted `"\n"` in standard output statement.

This has two primary benefits:

1. It's simpler to double-quote all outputted text rather than having to determine what should be single-quoted and double-quoted.
2. More importantly, it helps avoid inadvertent multicharacter literals. We cover multicharacter literals and some of the unexpected output they can cause in lesson 4.11 -- Chars.

Single quotes should be preferred in non-output cases.

We'll cover what '\n' is in more detail when we get to the lesson on chars (4.11 -- Chars).

Best practice

Prefer `\n` over `std::endl` when outputting text to the console.

Warning

`'\n'` uses a backslash (as do all special characters in C++), not a forward slash.

Using a forward slash (e.g. `'/n'`) or including other characters inside the single quotes (e.g. `' \n'` or `'.\n'`) will result in unexpected behavior. For example, `std::cout << '/n';` will often print as `12142`, which probably isn't what you were expecting.

std::cin

`std::cin` is another predefined variable that is defined in the `iostream` library. Whereas `std::cout` prints data to the console using the insertion operator (`<<`), `std::cin` (which stands for "character input") reads input from keyboard using the **extraction operator (`>>`)**. The input must be stored in a variable to be used.

```
#include <iostream>  // for std::cout and std::cin

int main()
{
    std::cout << "Enter a number: "; // ask user for a number

    int x{};       // define variable x to hold user input (and value-initialize it)
    std::cin >> x; // get number from keyboard and store it in variable x

    std::cout << "You entered " << x << '\n';
    return 0;
}
```

Try compiling this program and running it for yourself. When you run the program, line 5 will print "Enter a number: ". When the code gets to line 8, your program will wait for you to enter input. Once you enter a number (and press enter), the number you enter will be assigned to variable *x*. Finally, on line 10, the program will print "You entered " followed by the number you just entered.

For example (I entered 4):

```
Enter a number: 4
You entered 4
```

This is an easy way to get keyboard input from the user, and we will use it in many of our examples going forward. Note that you don't need to use '\n' when accepting input, as the user will need to press the *enter* key to have their input accepted, and this will move the cursor to the next line of the console.

If your screen closes immediately after entering a number, please see lesson 0.8 -- A few common C++ problems for a solution.

Just like it is possible to output more than one bit of text in a single line, it is also possible to input more than one value on a single line:

```
#include <iostream>  // for std::cout and std::cin

int main()
{
    std::cout << "Enter two numbers separated by a space: ";

    int x{}; // define variable x to hold user input (and value-initialize it)
    int y{}; // define variable y to hold user input (and value-initialize it)
    std::cin >> x >> y; // get two numbers and store in variable x and y respectively

    std::cout << "You entered " << x << " and " << y << '\n';

    return 0;
}
```

This produces the output:

```
Enter two numbers separated by a space: 5 6
You entered 5 and 6
```

Best practice

There's some debate over whether it's necessary to initialize a variable immediately before you give it a user provided value via another source (e.g. std::cin), since the user-provided value will just overwrite the initialization value. In line with our previous recommendation that variables should always be initialized, best practice is to initialize the variable first.

We'll discuss how *std::cin* handles invalid input in a future lesson (9.5 -- std::cin and handling invalid input). For now, it's enough to know that *std::cin* will extract as much as it can, and any input characters that could not be extracted are left for a later extraction attempt.

For advanced readers

The C++ I/O library does not provide a way to accept keyboard input without the user having to press *enter*. If this is something you desire, you'll have to use a third party library. For console applications, we'd recommend pdcurses, FXTUI, cpp-terminal, or notcurses. Many graphical user interface libraries have their own functions to do this kind of thing.

Summary

New programmers often mix up `std::cin`, `std::cout`, the insertion operator (`<<`) and the extraction operator (`>>`). Here's an easy way to remember:

- `std::cin` and `std::cout` always go on the left-hand side of the statement.
- `std::cout` is used to output a value (cout = character output)
- `std::cin` is used to get an input value (cin = character input)
- `<<` is used with `std::cout`, and shows the direction that data is moving. `std::cout << 4` moves the value `4` to the console.
- `>>` is used with `std::cin`, and shows the direction that data is moving. `std::cin >> x` moves the value the user entered from the keyboard into variable `x`.

We'll talk more about operators in lesson 1.9 -- Introduction to literals and operators.

Quiz time

Question #1

Consider the following program that we used above:

```
#include <iostream>  // for std::cout and std::cin

int main()
{
    std::cout << "Enter a number: "; // ask user for a number
    int x{}; // define variable x to hold user input
    std::cin >> x; // get number from keyboard and store it in variable x
    std::cout << "You entered " << x << '\n';
    return 0;
}
```

The program expects you to enter an integer value, as the variable x that the user input will be put into is an integer variable.

Run this program multiple times and describe what happens when you enter the following types of input instead:

a) A letter, such as *h*

Show Solution

b) A number with a fractional part (e.g. `3.2`). Try numbers with fractional parts less than 0.5 and greater than 0.5 (e.g. `3.2` and `3.7`).

Show Solution

c) A small negative integer, such as *-3*

Show Solution

d) A word, such as *Hello*

Show Solution

e) A really big number (at least 3 billion)

Show Solution

f) A small number followed by some letters, such as *123abc*

Show Solution

Related content

We discuss how `std::cin` and `operator>>` handle invalid input in future lesson 9.5 -- std::cin and handling invalid input.