1.6 INPUTS/OUTPUTS IN C++

Introduction

Input/output operations are the operations by which a program receives data or displays results. These operations must be viewed from a program perspective:

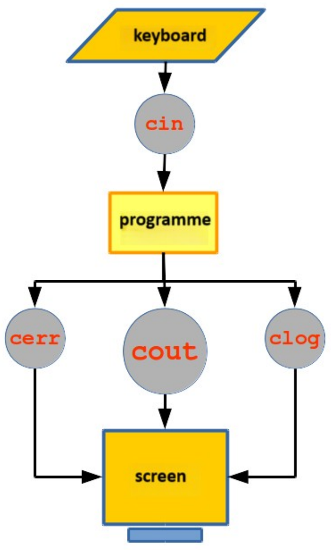
- **input operations**: data enters the program, the program reads data
- **output operations**: data exits the program, the program displays data

Basically, the data entering or leaving the program are strings of characters that the program receives and sends respectively.

The C++ language provides a uniform way to perform input/output operations, whether they are done at the console, in files, or with other devices that process characters. It is about stream or flow. A stream can be thought of as a sequence of characters that are sent in a well-defined order from a source to a destination. The program will insert characters into the stream (if it is an output stream, which displays data) or extract characters from the stream (if it is an input stream, from which data is read).

The C++ standard library allows working with **several categories** of streams. Among these we will next discuss console streams, the standard input-output device, in other words streams that allow reading from the keyboard and display on the screen. The objects that allow these operations are:

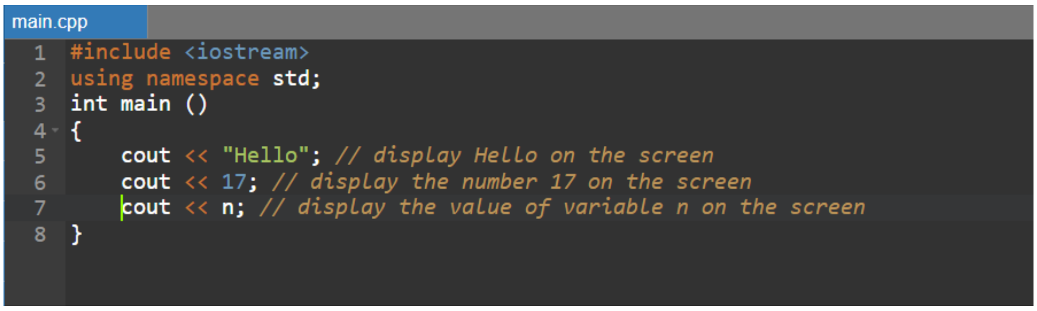
- *cin* – standard input stream
- *cout* – standard output stream
- *cerr* – standard output stream for errors
- *clog* – standard output stream for logging events



Next we'll talk about *cout* and *cin* – the standard output and input streams. *cerr* and *clog* are also output streams and work like *cout;* moreover, most of the time they do exactly the same thing as *cout* – display on the screen. However, there are situations when the output produced by *cerr* or *clog* is redirected to other devices.

The output stream *cout*

In most cases, the standard output device is the screen and can be accessed with the *cout* stream. For this, *cout* is used together with **the insertion operator** <<, followed by the date to be displayed:

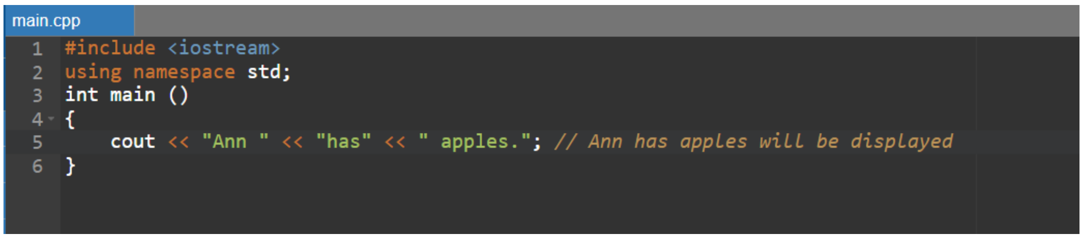


main.cpp

```cpp
#include <iostream>
using namespace std;
int main ()
{
    cout << "Hello"; // display Hello on the screen
    cout << 17; // display the number 17 on the screen
    cout << n; // display the value of variable n on the screen
}
```

The *cout* operator displays the value on the right in the left stream. Note that "Hello" is delimited by quotes because it is **a string literal constant**, and n is not delimited by quotes because it is **a variable**.

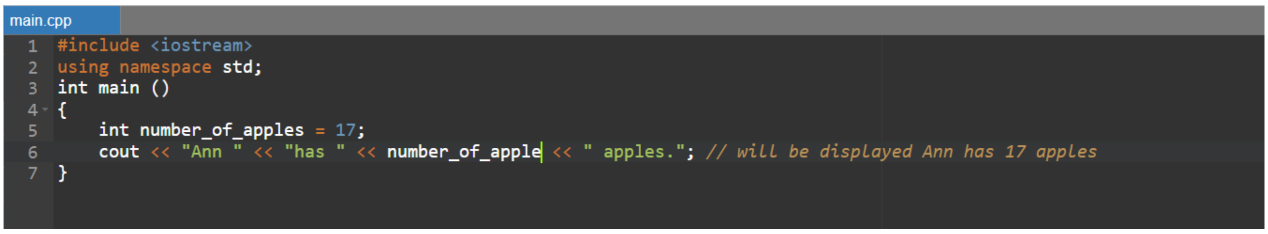
We can display multiple values in the same statement:



main.cpp

```cpp
#include <iostream>
using namespace std;
int main ()
{
    cout << "Ann " << "has" << " apples."; // Ann has apples will be displayed
}
```

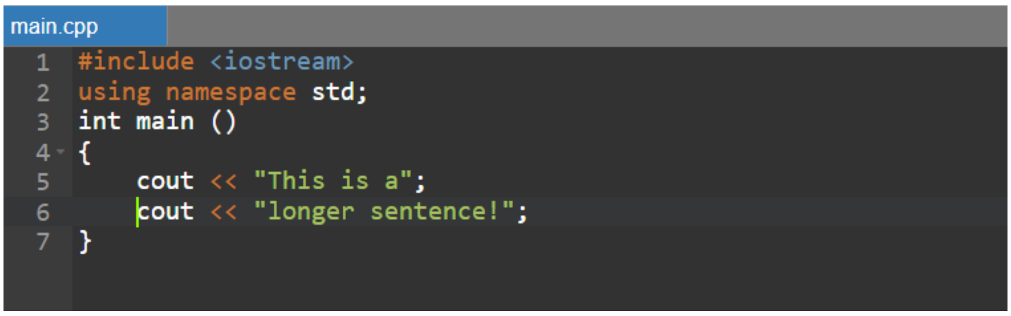
or



main.cpp

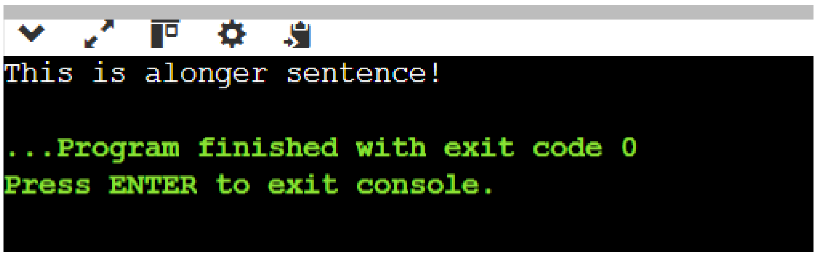
```cpp
#include <iostream>
using namespace std;
int main ()
{
    int number_of_apples = 17;
    cout << "Ann " << "has " << number_of_apple << " apples."; // will be displayed Ann has 17 apples
}
```

cout does not automatically add line breaks or spaces. For example:



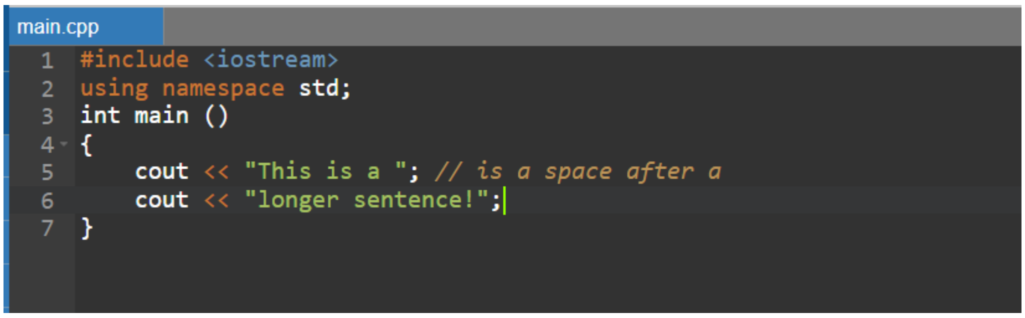
```cpp
#include <iostream>
using namespace std;
int main ()
{
    cout << "This is a";
    cout << "longer sentence!";
}
```

will display



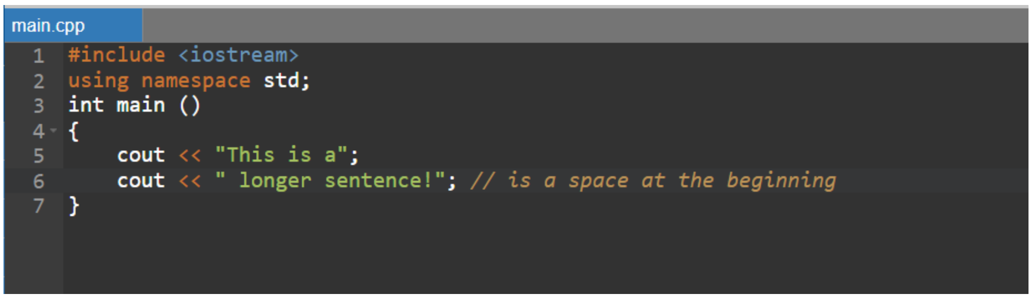
```
This is alonger sentence!

...Program finished with exit code 0
Press ENTER to exit console.
```

To insert a space between **a** and **sentence**, we specify it explicitly:



```cpp
#include <iostream>
using namespace std;
int main ()
{
    cout << "This is a "; // is a space after a
    cout << "longer sentence!";
}
```

or



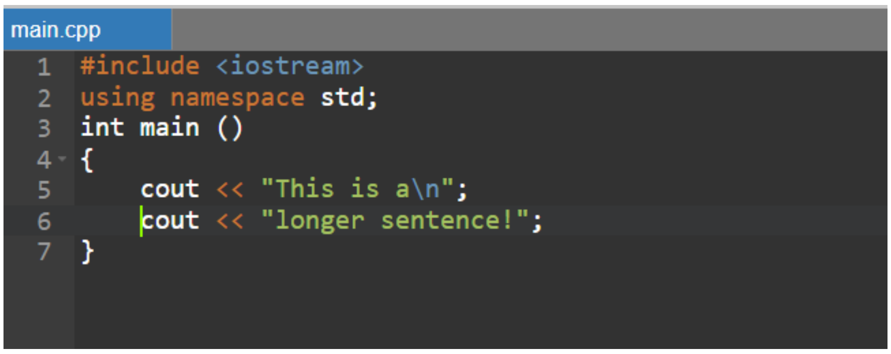
```cpp
#include <iostream>
using namespace std;
int main ()
{
    cout << "This is a";
    cout << " longer sentence!"; // is a space at the beginning
}
```

If we want to display on different lines, we proceed as follows:

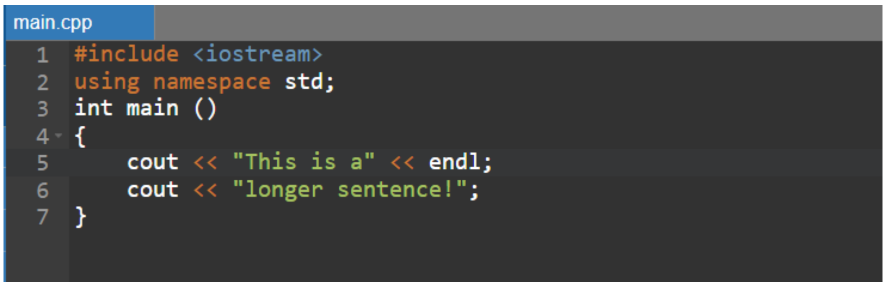
1. we put *n* at the end of the first part

   or

2. we use the *endl* handler to break the line

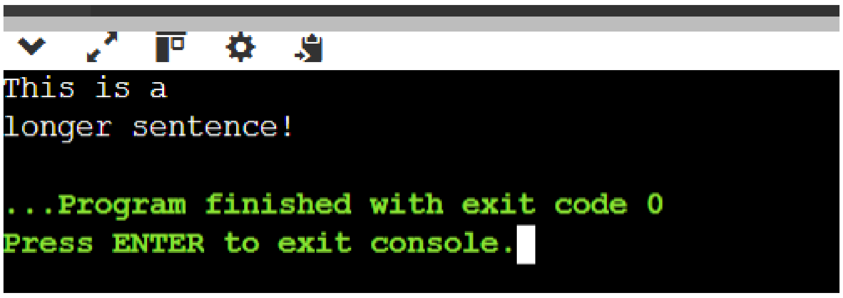
1. 

```cpp
#include <iostream>
using namespace std;
int main ()
{
      cout << "This is a\n";
      cout << "longer sentence!";
}
```

2. 

```cpp
#include <iostream>
using namespace std;
int main ()
{
      cout << "This is a" << endl;
      cout << "longer sentence!";
}
```

Both of the above options will display:

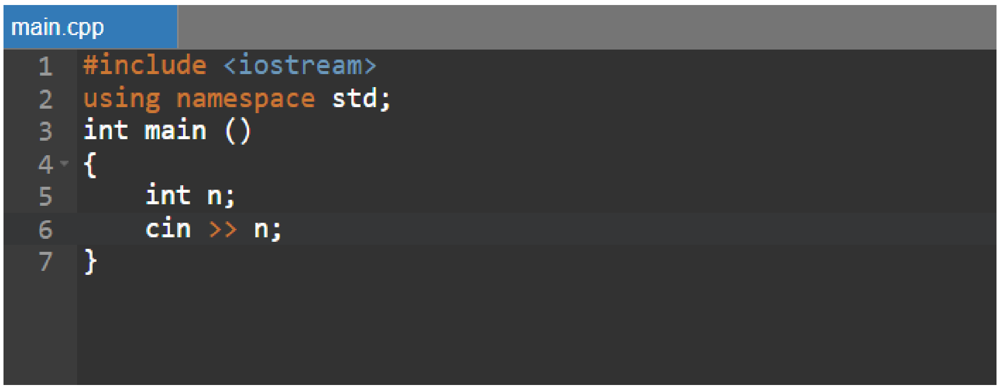


```
This is a
longer sentence!

...Program finished with exit code 0
Press ENTER to exit console.
```

endl produces a newline character, just like inserting \n, but it does something else: *endl* flushes the *cout* stream's buffer, that is, it forces all characters inserted into the stream up to this point to be displayed on the screen. *endl* can cause **delays** in program execution, so it should be used with caution.

The input stream cin

In most cases, the standard input device is the **keyboard** and can be accessed with the *cin* stream. For this, *cin* is used together with the extraction operator(>>), followed by the variable in which the extracted value will be stored (the variable to be read):

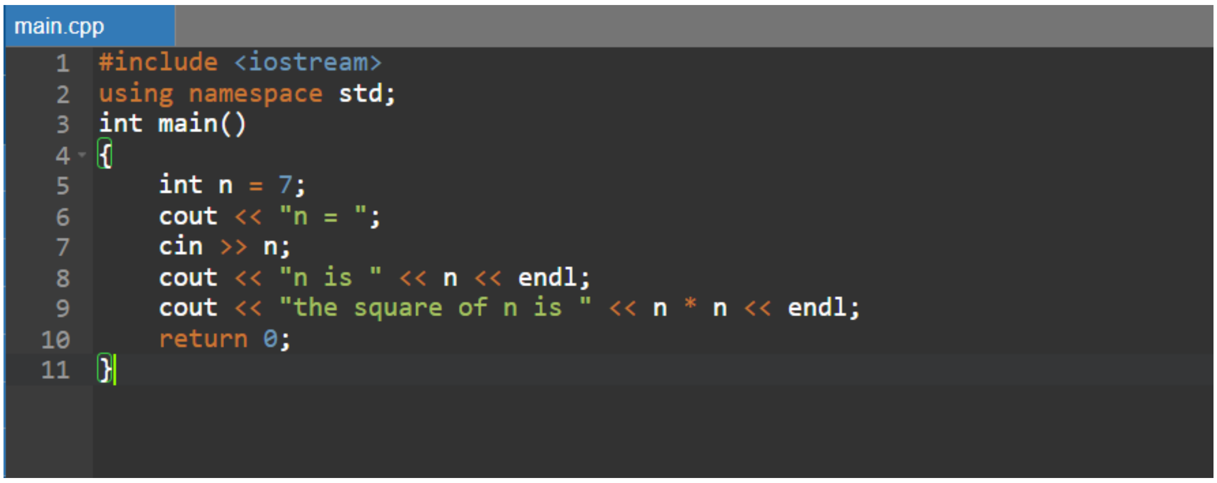


main.cpp

```cpp
#include <iostream>
using namespace std;
int main ()
{
    int n;
    cin >> n;
}
```

First the variable **n** is declared, then a value is read for it – a value is extracted from *cin* and stored in the variable n. On execution, the program waits for a value to **be entered** from the keyboard. In fact, the entered characters are passed to the program only when the **ENTER** key is pressed.

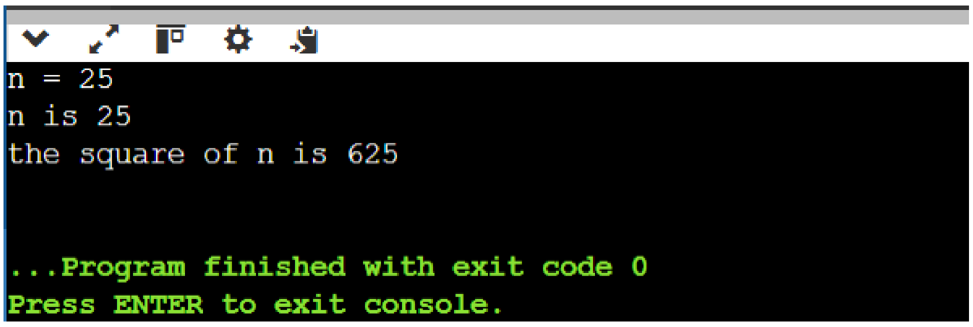
Consider the following program:



main.cpp

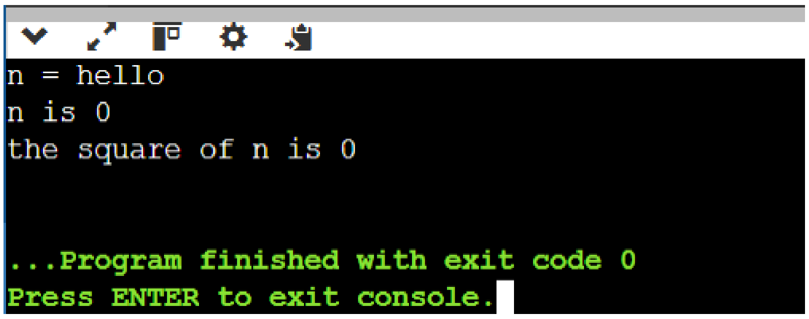
```cpp
#include <iostream>
using namespace std;
int main()
{
    int n = 7;
    cout << "n = ";
    cin >> n;
    cout << "n is " << n << endl;
    cout << "the square of n is " << n * n << endl;
    return 0;
}
```

Its result depends on the value entered for n. It could be:



```
n = 25
n is 25
the square of n is 625


...Program finished with exit code 0
Press ENTER to exit console.
```

But what if no number is entered?

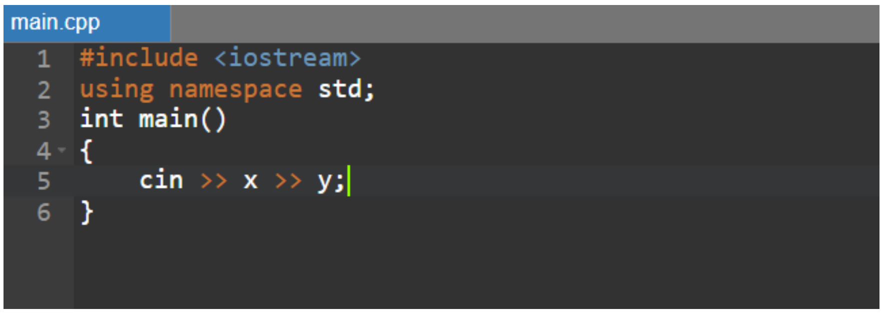


```
n = hello
n is 0
the square of n is 0


...Program finished with exit code 0
Press ENTER to exit console.
```

In the extraction operation from *cin*, the type of the variable after >> matters. The characters in the stream are interpreted according to the type of the variable. If these characters do not match the type of the variable, the extraction operation fails. If the extract from stream operation fails:

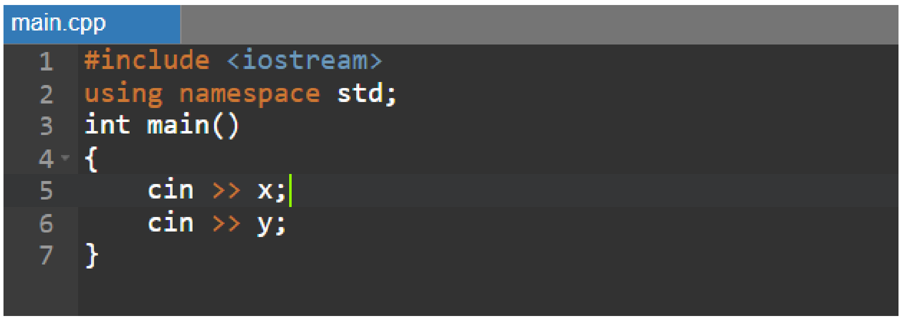
- the variable is not changed and an error flag (*failbit*) is set – up to **C++11**;
- the variable receives the value 0 and the *failbit* is set. If the extracted value exceeds the type bounds of the variable, it is given the maximum or minimum value of its type and the *failbit* is set – since **C++11**.

You can read the values of two or more variables in the same statement:



```cpp
#include <iostream>
using namespace std;
int main()
{
    cin >> x >> y;
}
```

Which is equivalent to

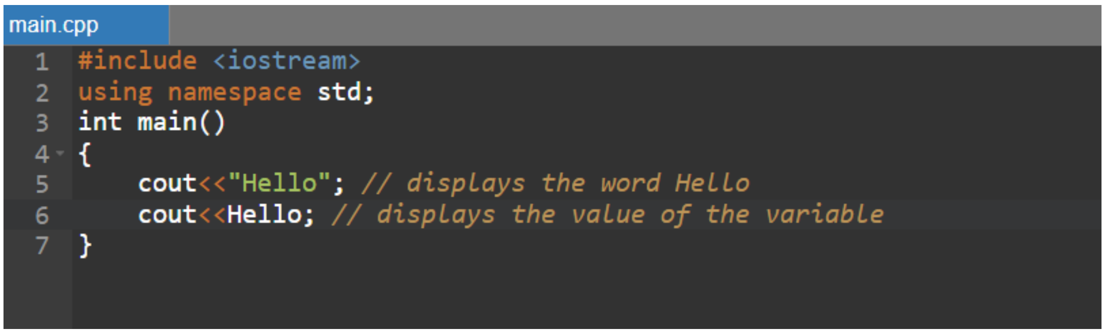


```cpp
#include <iostream>
using namespace std;
int main()
{
    cin >> x;
    cin >> y;
}
```

In both cases two values are expected; they can be **separated/preceded** by any kind of whitespace: **spaces, TABs, newline characters**.

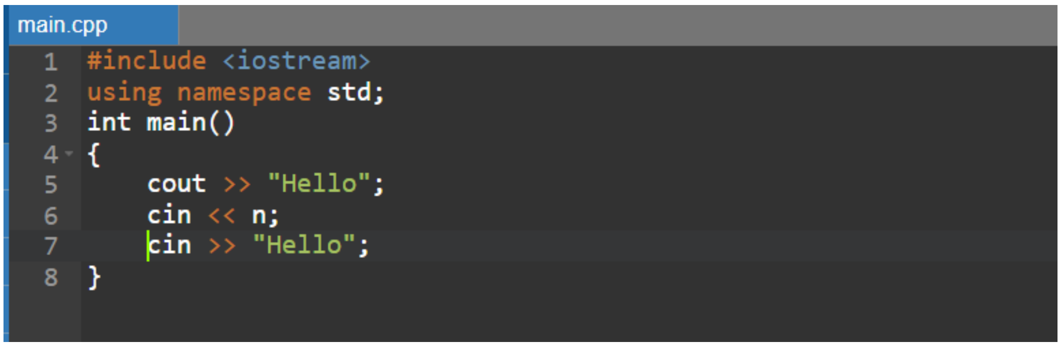
Pay attention to!

- **If the text following << is in quotes, it will be displayed as such. If it is not in quotes, it is assumed to be a variable, and its value is displayed.**


main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout<<"Hello"; // displays the word Hello
    cout<<Hello; // displays the value of the variable
}
```

- **One of the most common errors is reversing the operators for the cin and cout streams, or reading the value of a constant. The following instructions are wrong:**


main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout >> "Hello";
    cin << n;
    cin >> "Hello";
}
```