Basic Input / Output

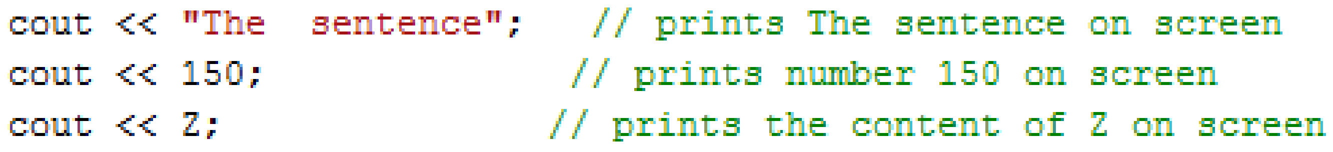
Until now, the example programs of previous lectures provided very little interaction with the user, if any at all.
Using the standard input and output library, we will be able to interact with the user by **printing** messages on the screen and **getting** the user's input from the keyboard.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A **stream** is an object where a program can either insert or extract characters to/from it

The standard C++ library includes the header file iostream, where the standard input and output stream objects are declared.

Standard Output (cout)

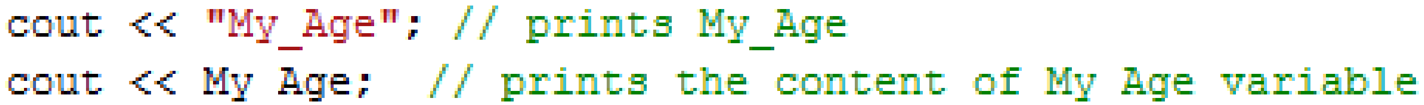
By default, the standard **output** of a program is the screen, and the C++ stream object defined to access it is **cout**. cout is used in conjunction with the *insertion operator*, which is written as << (two "less than" signs).



```
cout << "The  sentence";    // prints The sentence on screen
cout << 150;               // prints number 150 on screen
cout << Z;                // prints the content of Z on screen
```

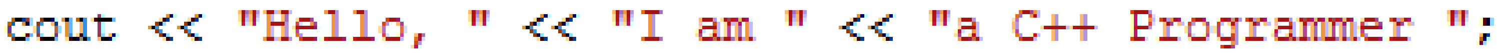
The << operator inserts the data that follows it into the stream preceding it. In the **examples** above it inserted the constant string **The sentence**, the numerical constant **150** and variable **Z** into the standard output stream cout.

Notice that the sentence in the **first** instruction is enclosed between **double quotes** (") because it is a **constant string of characters**. Whenever we want to use constant strings of characters we must enclose them between **double quotes** (") so that they can be clearly distinguished from **variable names**. For example, these two sentences have very different results:



```
cout << "My_Age"; // prints My_Age
cout << My_Age;  // prints the content of My_Age variable
```

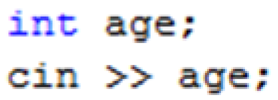
The insertion operator (<<) may be used more than once in a single statement:



```
cout << "Hello, " << "I am " << "a C++ Programmer ";
```

Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the **cin** stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:



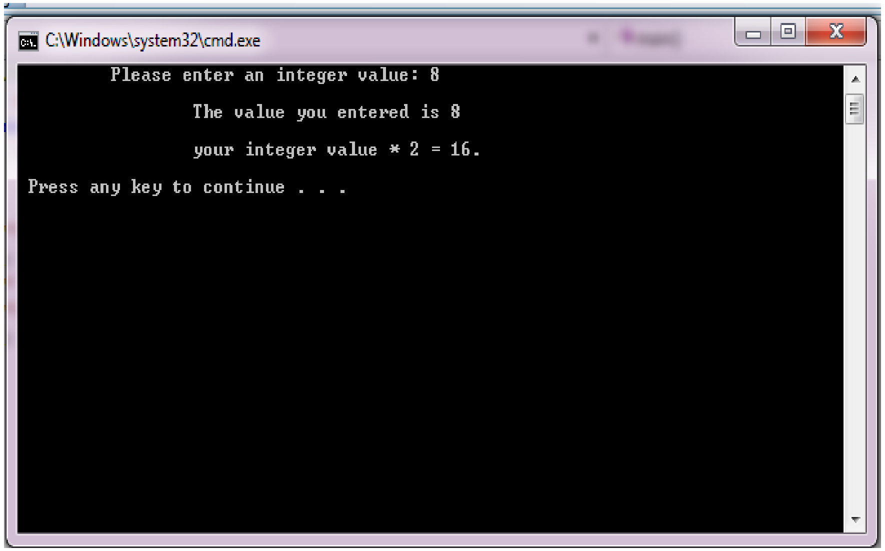
```
int age;
cin >> age;
```

The first statement declares a **variable** of type int called age, and the second one waits for an input from **cin** (the keyboard) in order to store it in this integer variable.

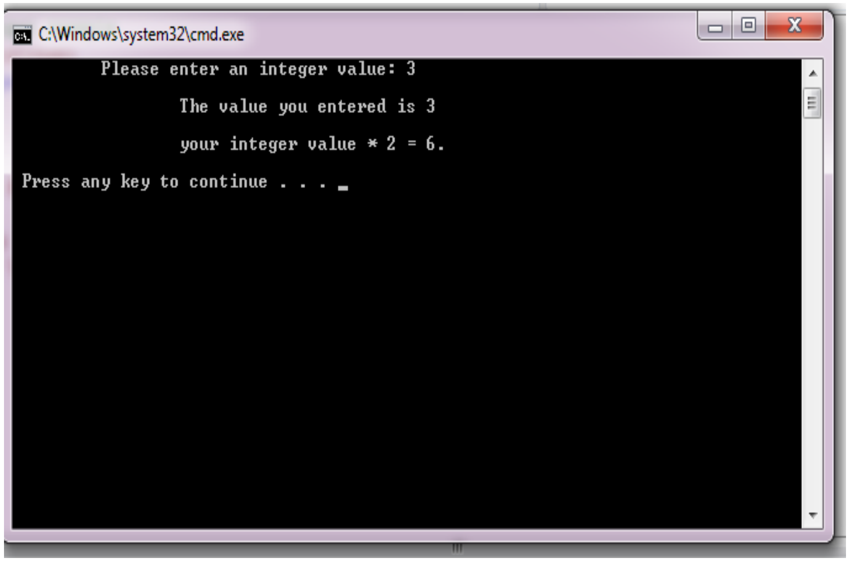
cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from **cin** will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// input/output example
#include <iostream>
using namespace std;
int main ()
{
int i;
cout << "\t Please enter an integer value: ";
cin >> i;
cout << " \n \t \t The value you entered is " << i;
cout << "\n \n \t \t your integer value * 2 = " << i*2 << ".\n \n ";
return 0;
}
```



```
        Please enter an integer value: 8
                The value you entered is 8
                your integer value * 2 = 16.
Press any key to continue . . .
```



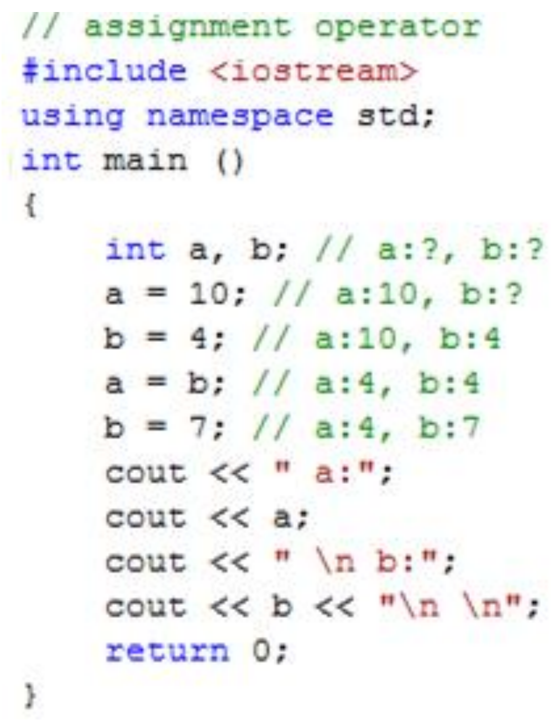
```
        Please enter an integer value: 3
                The value you entered is 3
                your integer value * 2 = 6.
Press any key to continue . . . _
```

Output 1 when i=8

Output 2 when i=3

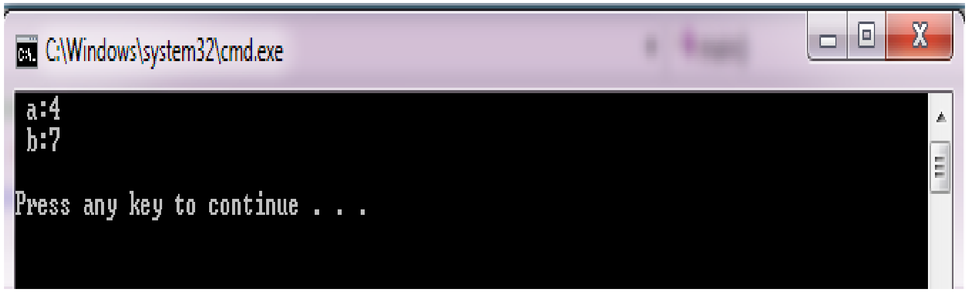
Operators

Assignment (=)



```
// assignment operator
#include <iostream>
using namespace std;
int main ()
{
    int a, b; // a:?, b:?
    a = 10; // a:10, b:?
    b = 4; // a:10, b:4
    a = b; // a:4, b:4
    b = 7; // a:4, b:7
    cout << " a:";
    cout << a;
    cout << " \n b:";
    cout << b << "\n \n";
    return 0;
}
```

The program



```
a:4
b:7

Press any key to continue . . .
```

The output

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the **percentage sign (%).** Modulo is the operation that gives the **remainder** of a **division** of two values. For example, if we write:

```
a = 11 % 3;
```

the variable **a** will contain the value **2**,

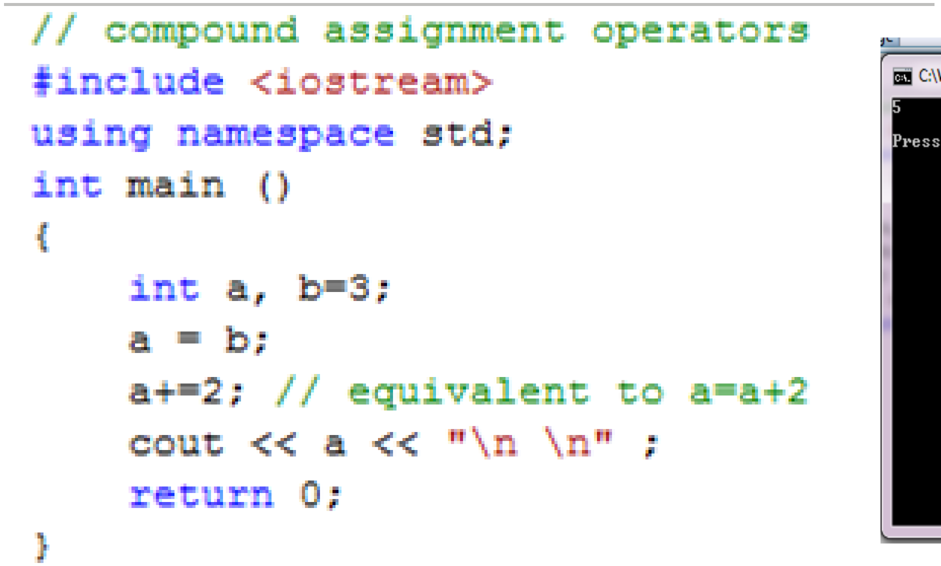
since **2** is the remainder from dividing **11** between **3**.

Compound assignment (+=, -=, *=, /=, %=

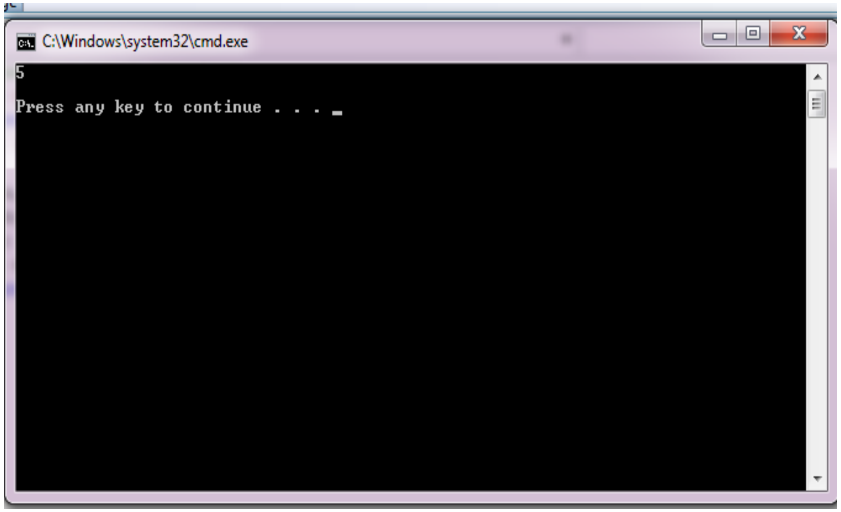
When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression	is equivalent to
`value += increase;`	`value = value + increase;`
`a -= 5;`	`a = a - 5;`
`a /= b;`	`a = a / b;`
`price *= units + 1;`	`price = price * (units + 1);`

and the same for all other operators. For example:



```
// compound assignment operators
#include <iostream>
using namespace std;
int main ()
{
     int a, b=3;
     a = b;
     a+=2; // equivalent to a=a+2
     cout << a << "\n \n" ;
     return 0;
}
```



```
5

Press any key to continue . . .
```

Increase and decrease (++, --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

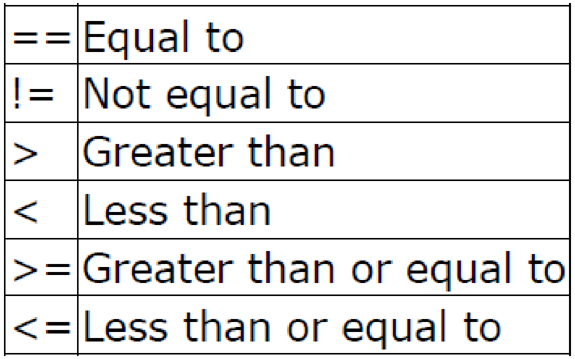
```
c++ ;
c+=1;
c=c+1;
```

Relational and equality operators (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The **result** of a relational operation is a **Boolean** value that can only be **true** or **false**, according to its Boolean result.

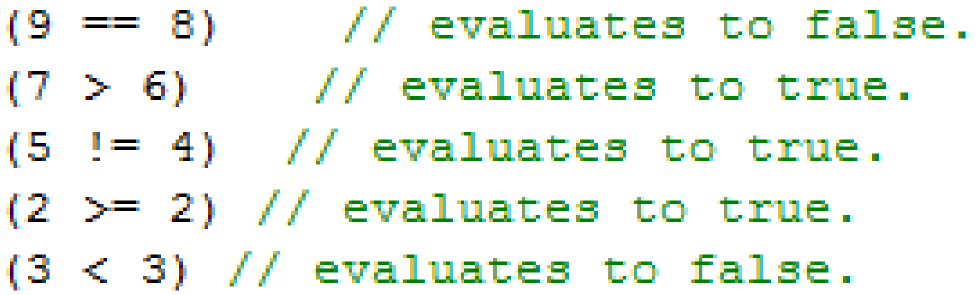
We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

Relational and equality operators (==, !=, >, <, >=, <=)



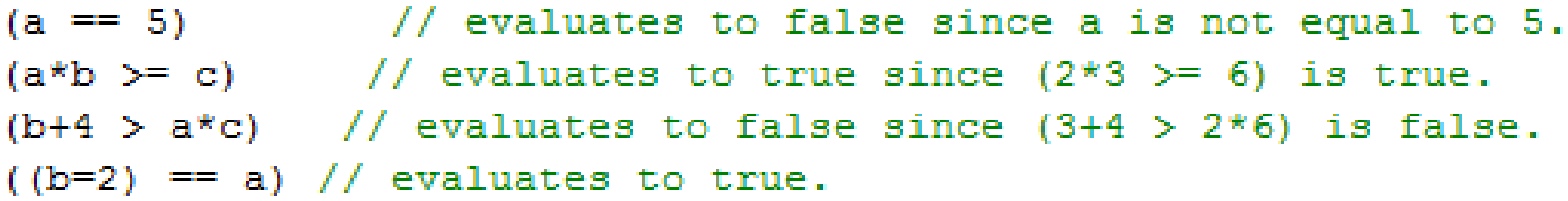
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:



```
(9 == 8)     // evaluates to false.
(7 > 6)     // evaluates to true.
(5 != 4)   // evaluates to true.
(2 >= 2)  // evaluates to true.
(3 < 3)  // evaluates to false.
```

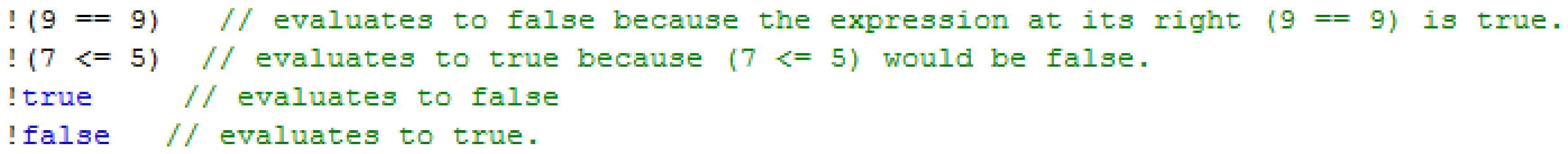
Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that **a=2**, **b=3** and **c=6**,



```
(a == 5)          // evaluates to false since a is not equal to 5.
(a*b >= c)       // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c)     // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a)  // evaluates to true.
```

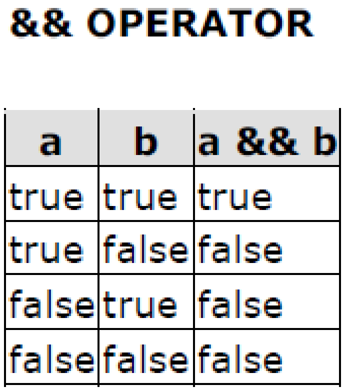
Logical operators (!, &&, ||)

The Operator **!** is the C++ operator to perform the Boolean operation **NOT**, it has only one operand, located at its right, and the only thing that it does is to **inverse the value** of it, producing **false if its operand is true** and **true if its operand is false**. Basically, it returns the opposite Boolean value of evaluating its operand. For example:



```
!(9 == 9)   // evaluates to false because the expression at its right (9 == 9) is true.
!(7 <= 5)  // evaluates to true because (7 <= 5) would be false.
!true      // evaluates to false
!false    // evaluates to true.
```

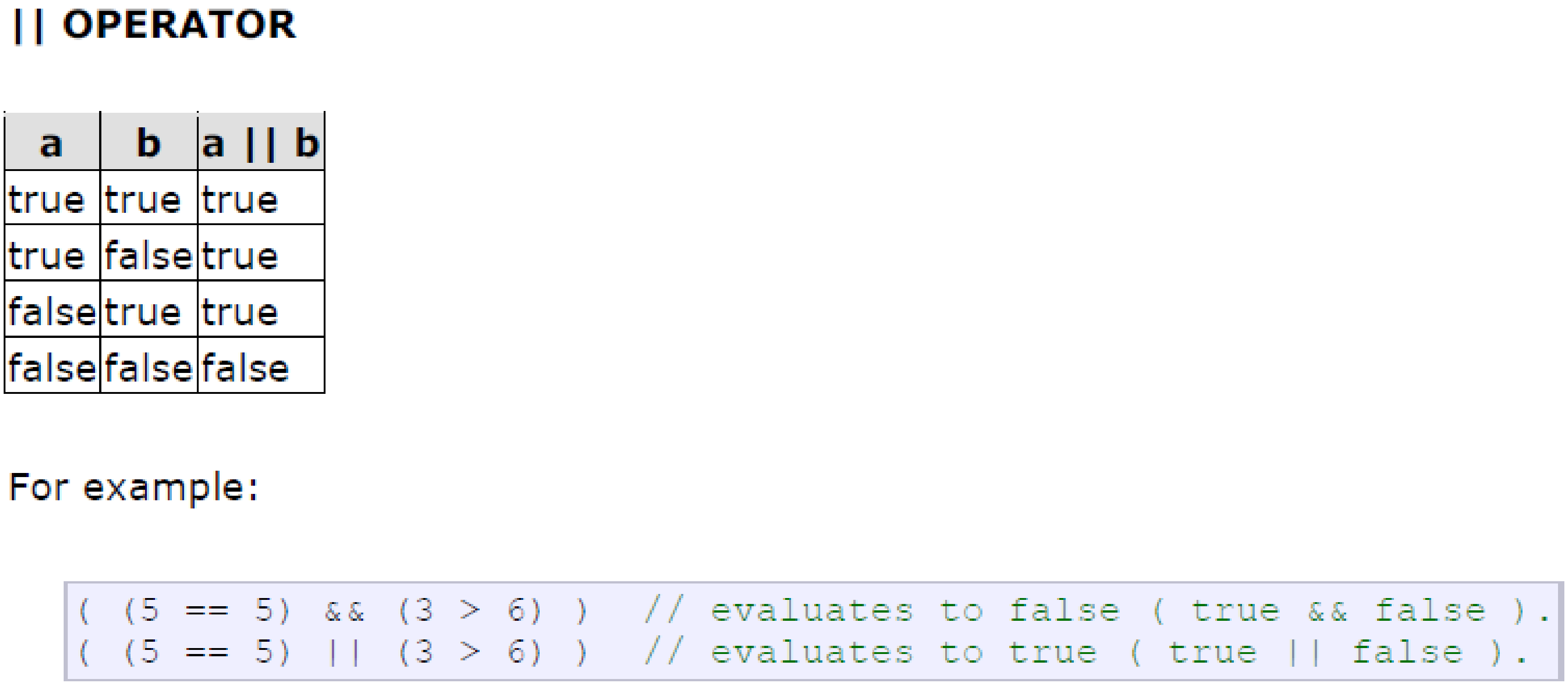
The logical operators **&&** and **||** are used when **evaluating** two expressions to obtain a single relational result. The operator **&&** corresponds **with Boolean logical operation AND**. This operation results **true** if **both its two operands are true**, and **false** otherwise. The following panel shows the result of operator **&&** evaluating the expression **a && b :**



&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator **||** corresponds with Boolean logical operation **<u>OR</u>**. This operation results **true** if **either** one of its two operands is **true**, thus being **false** <u>only</u> **<u>when both operands are false</u>** themselves. Here are the possible results of **a || b** :



|| OPERATOR

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

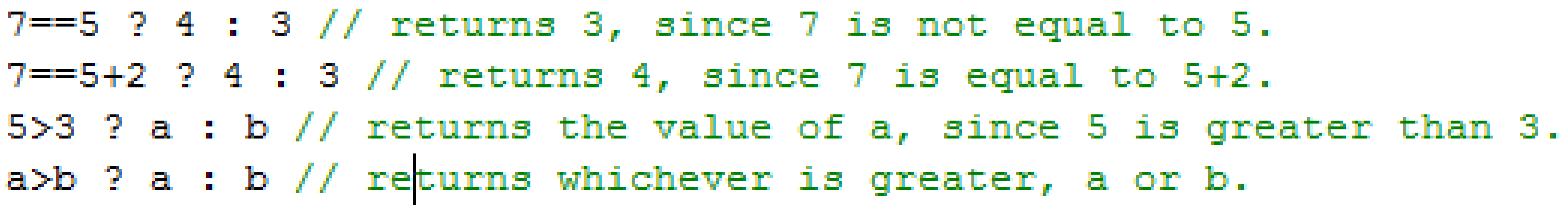
```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

Conditional operator (?)

The conditional operator **evaluates** an expression **returning** a value **if** that **expression is true** and a **different one if the expression is evaluated as false**. Its format is:

condition ? result1 : result2

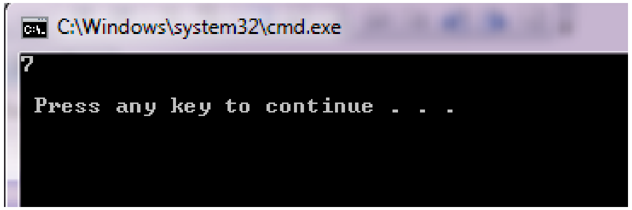
If condition is **true** the expression will return **result1**, if it is **not** it will return **result2**.



```
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.
```

```
// conditional operator
#include <iostream>
using namespace std;
int main ()
{
     int a,b,c;
     a=2;
     b=7;
     c = (a>b) ? a : b;
     cout << c << "\n\n ";
return 0;
}
```

The Output →



```
7

 Press any key to continue . . .
```