Lab 02

Objectives

After performing this lab, students will be able to

- Use variables
- Get input from users
- Understand and use data types
- Type conversion
- Use Operators

Variables

All computers have memory, called **RAM** (short for random access memory), that is available for your programs to use. You can think of RAM as a series of numbered mailboxes that can each be used to hold a piece of data while the program is running. A single piece of data, stored in memory somewhere, is called a **value**.

In C++, direct memory access is not allowed. Instead, we access memory indirectly through **variables**. **Variables** represent storage locations in the computer's memory that are given some name. You can consider them like small blackboards, where we can write some value and change/update it later if we want.

Literals

A **literal** (also known as a **literal constant**) is a fixed value that has been inserted directly into the source code. For example, in the following two code samples, the first one contains a fixed string (collection of characters) **Hello World**, and the second line shown an integer literal 45 is saved in a variable named **n**.

```
cout << "Hello World";

int n = 45;
```

When we give some value to a variable, we just use that variable (name) to access that value at any place in the program. For example:

```
int main()
{
  int a = 10;
  cout<<a;
  return 0;
}
```

Output: 10

Here we just print/display the variable "***a***" but we get its value (10) as output.

Identifiers

Variables (or function) names are called *identifiers*. An identifier is a programmer-defined name that represents some element of a program. Variable names are examples of identifiers. You may choose your own variable names in C++, as long as you do not use any of the C++ **key words**. The **key words** make up the "core" of the language and have specific purpose/meaning. So, you cannot use them as a name of your variables. A list of such key words is shown in the following table

alignas	const	for	private	throw
alignof	constexpr	friend	protected	true
and	const_cast	goto	public	try
and_eq	continue	if	register	typedef
asm	decltype	inline	reinterpret_cast	typeid
auto	default	int	return	typename
bitand	delete	long	short	union
bitor	do	mutable	signed	unsigned
bool	double	namespace	sizeof	using
break	dynamic_cast	new	static	virtual
case	else	noexcept	static_assert	void
catch	enum	not	static_cast	volatile
char	explicit	not_eq	struct	wchar_t
char16_t	export	nullptr	switch	while
char32_t	extern	operator	template	xor
class	false	or	this	xor_eq
compl	float	or_eq	thread_local	

Identifier naming rules

C++ gives you a lot of flexibility to name identifiers as you wish. However, there are a few rules that must be followed when naming identifiers:

- The identifier cannot be a keyword. Keywords are reserved.
- The identifier can only be composed of letters (lower or upper case), numbers, and the underscore character. That means the name cannot contain symbols (except the underscore) nor whitespace (spaces or tabs).

- The identifier must begin with a letter (lower or upper case) or an underscore. It cannot start with a number.
- C++ is case sensitive, and thus distinguishes between lower and upper case letters. So, **nvalue** is different than **nValue** and **NVALUE**.

Some examples of valid and invalid variable names are shown in the table below

Variable Name	Valid/Invalid
age	Valid
my_score	Valid
num1	Valid
2num	Invalid (starts with a number)
student id	Invalid (contains a space between two words)
_height	Valid
my_math_marks	Valid
productName	Valid

Defining variables

Before use, variables must be defined. The general syntax of defining a variable is shown below:

Data_type variable_name;

In other words, we just need to tell the compiler

1. type of data is going to be stored in this variable
2. name of the variable

For example,

```
int numberOfBars;
double one_weight;
```

The keyword ***int*** is an abbreviation for **integer**, which are used to store whole numbers (numbers without decimals/points). They can store values like 3, 102, 3211, -456, etc. The variable defined in the above-given figure, ***numberOfBars***, is of type integer.

The data type ***double*** and ***float*** represent numbers with a fractional component and can store values like 1.34, 4.0, -345.6, etc. In the code sample shown above, the variable ***one_weight*** is of type double.

Defining more than one variable in a single statement

You can declare more than one variable on the same line separating them by commas. The general syntax is shown below:

Data_type variable1_name, variable2_name, …;

For example, the following line defines three variables of type **int** in a single statement;

int math_marks, eng_marks, phy_marks;

Assignment Statements

An assignment statement is used to store a value in a variable (or to change the value of a variable). For example,

```
int a;

a = 10;
cout<<a<<endl;

a = 23;
cout<<a<<endl;
```

Output:

```
10
23
```

Assignment statements end with a semi-colon. The single variable to be changed is always on the left of the ***assignment operator*** '**=**'. On the right of the assignment operator can be

- Constants, like **age = 21;**
- Variables, like **myCost = yourCost;**
- Expressions, like **circumference = diameter * 3.14159;**

The value to be assigned cannot be on the left-side. For example, the following assignment is invalid

```
25 = price;
```

Initializing Variables

Defining a variable does not give it a value. Giving a variable its first value is called as initializing the variable. Variables are initialized in (or using) assignment statements

```
int age;   // define the variable
age = 23;  // initialize the variable
```

Definition and initialization can be combined in a single statement using the following methods:

Method 1: int age = 23;

Method 2: int age (23);

Method 3: int age {23};

Uninitialized variables

Unlike some programming languages, C/C++ does not initialize most variables to a given value (such as zero) automatically. Thus when a variable is assigned a memory location by the compiler, the default value of that variable is whatever (garbage) value happens to already be in that memory location! A variable that has not been given a known value (usually through initialization or assignment) is called an **uninitialized variable**. Using the values of uninitialized variables can lead to unexpected results. Consider the following short program:

```
// define an integer variable named x
int x; // this variable is uninitialized because we haven't given it a value
// print the value of x to the screen
std::cout << x; // who knows what we'll get, because x is uninitialized
```

In this case, the computer will assign some unused memory to ***x***. It will then send the value residing in that memory location to **std::cout**, which will print the value (interpreted as an integer). But what value will it print? The answer is “who knows!”, and the answer may (or may not) change every time you run the program.

Getting input from the user

cin is an input stream bringing data from the keyboard. And sends/stores in the variable given/written on its right side.

```
#include <iostream>

using namespace std;

int main()
{
    int age;
    cin>>age;

    cout<<"Age is "<<age<<endl;

    return 0;
}
```

Then, when the code is run, the program will wait for the user to give some input and show a cursor like shown below:

And when the user presses the Enter/Return key after giving some number, then that value is saved in the variable **age** and then displayed.

```
22
Age is 22
```

However, it is a good practice to show a prompt message to the user asking him/her to input some value before the ***cin*** statement.

```
#include <iostream>

using namespace std;

int main()
{
    int age;
    cout<<"Enter your age ";
    cin>>age;

    cout<<"Age is "<<age<<endl;

    return 0;
}
```

Then a prompt, like the one shown below, will be sown to the user

```
Enter your age
```

And the rest of the program will work like discussed on the previous slide

```
Enter your age 33
Age is 33
```

You can also get input for multiple variables using a single ***cin*** command, as shown below:

```
cin>>num1>>num2>>num3;
```

Then, at run-time, input three values **separated by spaces**.

```
34 25 12  (and press enter key)
```

Data Types

Because all data on a computer is just a sequence of bits, we use a data type (often called a "type" for short) to tell the compiler how to interpret the contents of memory in some meaningful way.

When you give an object a value, the compiler and CPU take care of encoding your value into the appropriate sequence of bits for that data type, which are then stored in memory. For example, if you assign an integer object the value 65, that value is converted to the sequence of bits 0100 0001 and stored in the memory assigned to the object.

Conversely, when the object is evaluated to produce a value, that sequence of bits is reconstituted back into the original value. Meaning that 0100 0001 is converted back into the value 65.

Fortunately, the compiler and CPU do all the hard work here, so you generally don't need to worry about how values gets converted into bit sequences and back.

All you need to do is pick a data type for your object that best matches your desired use

C++ Fundamental Data Types

The table below shows the fundamental data types, their meaning, and their sizes (in bytes):

Data Type	Meaning	Size (in Bytes)
int	Integer	2 or 4
float	Floating-point	4
double	Double Floating-point	8
char	Character	1
wchar_t	Wide Character	2
bool	Boolean	1
void	Empty	0

Now, let us discuss these fundamental data types in more detail.

The int data type

- The **int** keyword is used to indicate integers (whole numbers without having any fractional part/number after decimal).
- Its size may vary between 2 to 4 bytes on different compilers but is usually 4 bytes. Meaning, it can store values from -2147483648 to 2147483647.

Example:

```
int salary = 85000;
```

The float and double data types

- **float** and **double** data types are used to store floating-point numbers (decimals and exponentials).
- The size of **float** is 4 bytes and the size of **double** is 8 bytes. Hence, **double** has two times the precision of **float**.

Example

```
float area = 64.74f;

double volume = 134.64534;
```

Note that by default, floating point literals (values with decimal points) are considered of type **double**. An **f** suffix is used to denote a literal of type **float**.

These two data types are also used for exponentials.

For example:

```
double distance = 45E12 // 45E12 is equal to 45*10^12
```

This way of writing numbers is called **scientific notation**. The letter E denotes the exponent part and can be either lower case "e" or upper case "E".

The char data type

- Keyword **char** is used for characters.
- Character data is where we want to store only one character like any one of the alphabets or special characters.
- The size of **char** data type is 1 byte.
- Characters in C++ are enclosed inside single quotes ' '.

Example:

```
char test = 'h';
```

- There is another type of **char** that is called wide character **wchar_t**.
- Wide character **wchar_t** is similar to the char data type, except its size is 2 bytes instead of 1.
- It is used to represent characters that require more memory to represent them than a single char.

Example:

```
wchar_t test = L'ם' //storing Hebrew character;
```

We write the letter **L** with **wchar_t** values.

The bool data type

- The **bool** data type has one of two possible values: **true** or **false**.
- Booleans are mostly used in conditional statements and loops (which we will learn in later chapters).

Example

```
bool cond = false;
```

The void data type

- The **void** keyword indicates an absence of data. It means "nothing" or "no value".
- We will use **void** when we will learn about functions and pointers.

Note: We cannot declare/create variables of the **void** type.

C++ Type Modifiers

We can further modify some of the fundamental data types by using type modifiers. There are 4 type modifiers in C++. They are:

1. signed
2. unsigned
3. short
4. long

C++ Modified Data Types List

Data Type	Size (in Bytes)	Meaning
signed int	4	used for integers (equivalent to **int**)
unsigned int	4	can only store positive integers
short	2	used for small integers (range **-32768 to 32767**)
long	at least 4	used for large integers (equivalent to **long int**)
unsigned long	4	used for large positive integers or 0 (equivalent to **unsigned long int**)
long long	8	used for very large integers (equivalent to **long long int**).

unsigned long long	8	used for very large positive integers or 0 (equivalent to **unsigned long long int**)
long double	8	used for large floating-point numbers
signed char	1	used for characters (guaranteed range **-127 to 127**)
unsigned char	1	used for characters (range **0 to 255**)

Let's see a few examples.

```
long b = 4523232;
long int c = 2345342;
long double d = 233434.56343;
short d = 3434233; // Error! out of range
unsigned int a = -5; // Error! can only store positive numbers or 0
```

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

We will learn about these derived data types in later tutorials.

Type conversion

C++ allows us to convert data of one type to that of another. This is known as type conversion.

There are two types of type conversion in C++.

1. Implicit Conversion
2. Explicit Conversion (also known as Type Casting)

Implicit Type Conversion

The type conversion that is done automatically by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

Let us look at two examples of implicit type conversion.

Example 1: Conversion From int to double

```
// Working of implicit type-conversion
```

```
#include <iostream>
using namespace std;
int main() {
  // assigning an int value to num_int
int num_int = 9;
  // declaring a double type variable
  double num_double;
  // implicit conversion

  // assigning int value to a double variable
num_double = num_int;
  cout << "num_int = " << num_int << endl;
  cout << "num_double = " << num_double << endl;
return 0; }
```

Output:

num_int = 9

num_double = 9

In this program, we have assigned an **int** data to a **double** variable. Here, the **int** value is automatically converted to **double** by the compiler before it is assigned to the **num_double** variable. This is an example of implicit type conversion.

Example 2: Automatic Conversion from double to int

```
//Working of Implicit type-conversion
#include <iostream>
using namespace std;
int main() {
int num_int;    double
num_double = 9.99;
  // implicit conversion
  // assigning a double value to an int variable
num_int = num_double;
cout << "num_int = " << num_int << endl;
cout << "num_double = " << num_double << endl;
return 0; }
```

Output:

num_int = 9

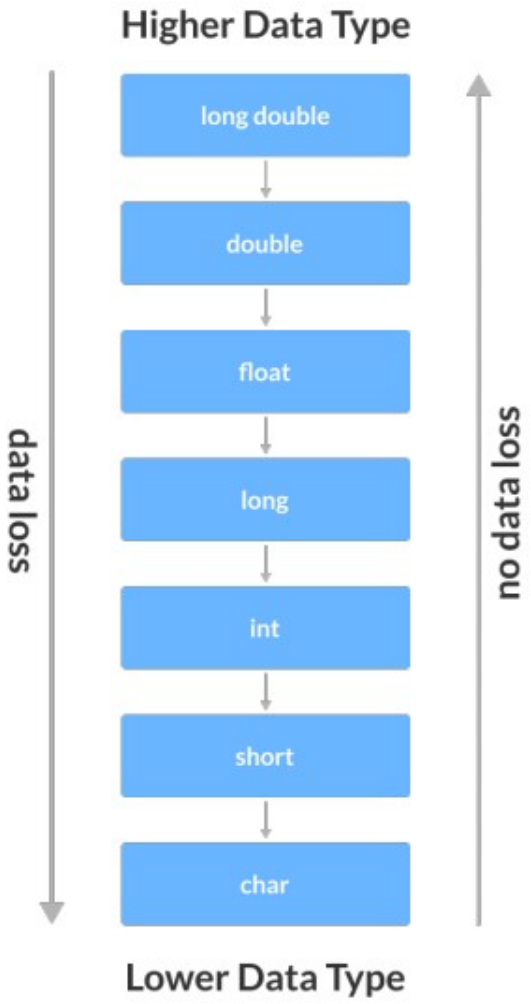
num_double = 9.99

In this program, we have assigned a **double** data to an **int** variable. Here, the **double** value is automatically converted to **int** by the compiler before it is assigned to the **num_int** variable. This is also an example of implicit type conversion.

Note: Since **int** cannot have a decimal part, the digits after the decimal point is truncated/removed in the above example.

Data Loss During Conversion (Narrowing Conversion)

As we have seen from the above example, conversion from one data type to another is prone to data loss. This happens when data of a larger type is converted to data of a smaller type.



C++ Explicit Conversion

When the user manually changes data from one type to another, this is known as **explicit conversion**. This type of conversion is also known as **type casting**.

Following are the major ways in which we can use explicit conversion in C++:

1. C-style type casting (also known as **cast notation**)
2. Function notation (also known as **old C++ style type casting**) **C-style Type Casting**

As the name suggests, this type of casting is favored by the **C programming language**. It is also known as **cast notation**. The syntax for this style is:

```
(data_type)expression;
```

For example:

```
// initializing int variable
int num_int = 26;
// declaring double variable
double num_double;
// converting from int to double
num_double = (double)num_int;
```

Function-style Casting

We can also use the function like notation to cast data from one type to another. The syntax for this style is:

```
data_type(expression);
```

For example

```
// initializing int variable
int num_int = 26;
// declaring double variable
double num_double;
// converting from int to double
num_double = double(num_int);
```

Example 3: Type Casting

```
#include <iostream>
using namespace std;
int main()
{
  // initializing a double variable
double num_double = 3.56;
```

```
  cout << "num_double = " << num_double << endl;

  // C-style conversion from double to int
int num_int1 = (int)num_double;
 cout << "num_int1 = " << num_int1 << endl;

  // function-style conversion from double to int
int num_int2 = int(num_double);
cout << "num_int2 = " << num_int2 << endl;
return 0; }
```

Output

```
num_double = 3.56
num_int1 = 3
num_int2 = 3
```

We used both the C style type conversion and the function-style casting for type conversion and displayed the results. Since they perform the same task, both give us the same output.

C++ Operators

An operator is a symbol used for performing operations on operands. Operands can be literal values or variables.

Consider the following operation:

```
a = x + y;
```

In the above statement, x and y are the operands while + is an addition operator. When the C++ compiler encounters the above statement, it will add x and y and store the result in variable a.

Operators in C++ can be classified into different types, like:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators

Arithmetic Operators

Arithmetic operators are used to perform arithmetic/mathematical operations on variables and data. For example,

```
a + b;
```

Here, the + operator is used to add two variables **a** and **b**. Similarly there are various other arithmetic operators in C++.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (gives remainder after division)

Example 1: Arithmetic Operators

```
#include <iostream>
using namespace std;

int main() {
int a, b;
a = 7;    b = 2;

  // printing the sum of a and b
cout << "a + b = " << (a + b) << endl;

  // printing the difference of a and b
cout << "a - b = " << (a - b) << endl;

  // printing the product of a and b
cout << "a * b = " << (a * b) << endl;

  // printing the division of a by b
cout << "a / b = " << (a / b) << endl;

  // printing the modulo of a by b
cout << "a % b = " << (a % b) << endl;
```

```
  return 0;
}
```

Output

```
a + b = 9
a - b = 5
a * b = 14
a / b = 3
a % b = 1
```

Here, the operators +, - and * compute addition, subtraction, and multiplication respectively as we might have expected.

A note about the division operator /

Note the operation (a / b) in our program. The / operator is the division operator. As we can see from the above example, if an integer is divided by another integer, we will get the quotient. However, if either divisor or dividend is a floating-point number, we will get the result in decimals. In C++

```
7/2 is 3
7.0 / 2 is 3.5 7
/ 2.0 is 3.5
7.0 / 2.0 is 3.5
```

% Modulo Operator

The modulo operator % computes the remainder. When a = 9 is divided by b = 4, the remainder is 1.

Note: The % operator can only be used with integers.

Increment and Decrement Operators

C++ also provides increment and decrement operators: ++ and --, respectively. ++ increases the value of the operand by 1, while -- decreases it by 1. For example,

```
int num = 5;
// increasing num by 1
++num;
```

Here, the value of **num** gets increased to 6 from its initial value of 5.

Increment/decrement operators work in two ways: **prefix** increment and **postfix** increment.

Operator	Symbol	Form	Operation
Prefix increment (preincrement)	++	++x	Increment x, then return x
Prefix decrement (pre decrement)	– –	– –x	Decrement x, then return x
Postfix increment (post increment)	++	x++	Copy x, then increment x, then return the copy
Postfix decrement (post decrement)	– –	x– –	Copy x, then decrement x, then return the copy

Note that there are two versions of each operator -- a prefix version (where the operator comes before the operand) and a postfix version (where the operator comes after the operand).

The prefix increment/decrement operators are very straightforward. First, the operand is incremented or decremented, and then expression evaluates to the value of the operand. For example:

```
#include <iostream>

int main()
{
    int x { 5 };
    int y = ++x; // x is incremented to 6, x is evaluated to the value 6, and 6 is assigned to y

    std::cout << x << ' ' << y;
    return 0;
}
```

This will print 6

6

The postfix increment/decrement operators are trickier. First, a copy of the operand is made. Then the operand (not the copy) is incremented or decremented. Finally, the copy (not the original) is evaluated. For example:

```
#include <iostream>

int main()
{
    int x { 5 };
    int y = x++; // x is incremented to 6, copy of original x is evaluated to the value 5, and 5 is assigned to y

    std::cout << x << ' ' << y;
    return 0;
}
```

This will print

6 5

Let's examine how this (y = x++) works in more detail.

First, a temporary copy of x is made that starts with the same value as x (5). Then the actual x is incremented from 5 to 6. Then the copy of x (which still has value 5) is returned and assigned to y. Then the temporary copy is discarded.

Consequently, y ends up with the value of 5 (the pre-incremented value), and x ends up with the value 6 (the post-incremented value).

Note that the postfix version takes a lot more steps, and thus may not be as performer as the prefix version.

Here is another example showing the difference between the prefix and postfix versions:

```
#include <iostream>

int main()
{
    int x{ 5 };
    int y{ 5 };
    std::cout << x << ' ' << y << '\n';
    std::cout << ++x << ' ' << --y << '\n'; // prefix
    std::cout << x << ' ' << y << '\n';
    std::cout << x++ << ' ' << y-- << '\n'; // postfix
    std::cout << x << ' ' << y << '\n';

    return 0;
}
```

This produces the output:

5 5

6 4

6 4

6 4

7 3

On the line where we have ++x and --y, we do a prefix increment and decrement. On this line, x and y are incremented/decremented before their values are sent to std::cout, so we see their updated values reflected by std::cout.

On the 10th line where we have x++ and y--, we do a postfix increment and decrement. On this line, the copy of x and y (with the pre-incremented and pre-decremented values) are what is sent to std::cout, so we don't see the increment and decrement reflected here. Those changes don't show up until the next line, when x and y are evaluated again.

Assignment Operators

In C++, assignment operators are used to assign values to variables. For example,

```
// assign 5 to a
a = 5;
```

Here, we have assigned a value of 5 to the variable **a**.

The arithmetic operators and assignment operator can also be used in the following ways:

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Example: Assignment Operators

```
#include <iostream>
using namespace std;
int main() {
 int a, b, temp;

  // 2 is assigned to a
a = 2;

  // 7 is assigned to b
  b = 7;
```

```
  // value of a is assigned to temp
temp = a; // temp will be 2
cout << "temp = " << temp << endl;

  // assigning the sum of a and b to a
a += b; // a = a +b
 cout << "a = " << a << endl;

  return 0; }
```

Output

```
temp = 2

a = 9
```

Relational Operators

A relational operator is used to check the relationship between two operands. For example, you may need to know which operand is greater than the other, or less than the other. Therefore, these are also called ***comparison operators***, sometimes. Their result is a Boolean value (**true** or **false**). For example,

```
// checks if a is greater than b
a > b;
```

Here, > is a relational operator. It checks if a is greater than b or not. If the relation is **true**, it returns true (or 1) whereas if the relation is **false**, it returns false (or 0).

Note: In C++, a Boolean false is also denoted by a 0 and a Boolean **true** is represented with a value greater than or equal to 1.

Operator	Meaning	Example
==	Is Equal To	3 == 5 gives us **false**
!=	Not Equal To	3 != 5 gives us **true**
>	Greater Than	3 > 5 gives us **false**
<	Less Than	3 < 5 gives us **true**
>=	Greater Than or Equal To	3 >= 5 give us **false**
<=	Less Than or Equal To	3 <= 5 gives us **true**

Example: Relational Operators

```
#include <iostream>
using namespace std;
int main() {
 int a, b;
a = 3;
b = 5;
bool result;
  result = (a == b); // false
  cout << "3 == 5 is " << result << endl;
result = (a != b); // true
  cout << "3 != 5 is " << result << endl;
result = a > b; // false
  cout << "3 > 5 is " << result << endl;
result = a < b; // true
  cout << "3 < 5 is " << result << endl;
result = a >= b; // false
  cout << "3 >= 5 is " << result << endl;
  result = a <= b; // true
  cout << "3 <= 5 is " << result << endl;
return 0; }
```

Output

```
3 == 5 is 0
3 != 5 is 1
3 > 5 is 0
3 < 5 is 1
3 >= 5 is 0 3
<= 5 is 1
```

Note: Relational operators are mostly used in decision making and loops.

Logical Operators

Logical operators are used to check whether an expression is true or false. OR

The logical operators check two or more conditions and return a Boolean value (true/false).

In C++, there are three logical operators

1. Logical AND operator (denoted by **&&**)
2. Logical OR operator (denoted by **||**)

3. Logical NOT operator (denoted by **!**)

Operator	**Description**
&& logical AND operator	The condition is true if both operands are non-zero (not ***false***).
|| logical OR operator	The condition is true if one of the operands is non-zero (not ***false***).
! logical NOT operator	It reverses operand's logical state. If the operand is ***true***, the ! operator makes it ***false*** and vice versa.

Logical Operator AND: &&

The logical AND operator is denoted by &&. It returns true if the condition on the left and the condition on the right are both true. Otherwise, it returns false. Here's its truth table:

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

For instance:

- (1 < 2 && 2 < 3) returns ***true***
- (1 < 2 && 2 > 3) returns ***false***

Logical Operator OR: II

The logical OR operator is denoted by ||. It returns true when the condition on the left is true or the condition on the right is true. Only one of them needs to be true. Here's its truth table:

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

For instance:

- (1 < 2 || 2 > 3) returns ***true***
- (1 > 2 || 2 > 3) returns ***false***

Logical Operator NOT: !

The logical NOT operator is denoted by "!". It reverses the bool outcome of the expression that immediately follows. Here's its truth table:

a	**!a**
false	true
true	false

For instance:

- (!true) returns false
- (!false) returns true
- (!(10 < 11)) returns false

Examples

a = 5
b = 8
Then,
(a > 3) && (b > 5) evaluates to true
(a > 3) && (b < 5) evaluates to false

(a > 3) || (b > 5) evaluates to true
(a > 3) || (b < 5) evaluates to true
(a < 3) || (b < 5) evaluates to false

!(a == 3) evaluates to true
!(a > 3) evaluates to false

Example: Logical Operators

```
#include <iostream>
using namespace std;
int main() {
bool result;
  result = (3 != 5) && (3 < 5); // true
 cout << "(3 != 5) && (3 < 5) is " << result << endl;
result = (3 == 5) && (3 < 5); // false
cout << "(3 == 5) && (3 < 5) is " << result << endl;
result = (3 == 5) && (3 > 5); // false
```

```
 cout << "(3 == 5) && (3 > 5) is " << result << endl;
result = (3 != 5) || (3 < 5); // true
cout << "(3 != 5) || (3 < 5) is " << result << endl;
result = (3 != 5) || (3 > 5); // true
cout << "(3 != 5) || (3 > 5) is " << result << endl;
result = (3 == 5) || (3 > 5); // false
 cout << "(3 == 5) || (3 > 5) is " << result << endl;
result = !(5 == 2); // true
cout << "!(5 == 2) is " << result << endl;
 result = !(5 == 5); // false
 cout << "!(5 == 5) is " << result << endl;
   return 0; }
```

Output

```
(3 != 5) && (3 < 5) is 1
(3 == 5) && (3 < 5) is 0
(3 == 5) && (3 > 5) is 0
(3 != 5) || (3 < 5) is 1
(3 != 5) || (3 > 5) is 1
(3 == 5) || (3 < 5) is 0
!(5 == 2) is 1 !(5
== 5) is 0
```

Explanation of logical operator program

- (3 != 5) && (3 < 5) evaluates to 1 because both operands (3 != 5) and (3 < 5) are 1 (true).
- (3 == 5) && (3 < 5) evaluates to 0 because the operand (3 == 5) is 0 (false).
- (3 == 5) && (3 > 5) evaluates to 0 because both operands (3 == 5) and (3 > 5) are 0 (false).
- (3 != 5) || (3 < 5) evaluates to 1 because both operands (3 != 5) and (3 < 5) are 1 (true).
- (3 != 5) || (3 > 5) evaluates to 1 because the operand (3 != 5) is 1 (true).
- (3 == 5) || (3 > 5) evaluates to 0 because both operands (3 == 5) and (3 > 5) are 0 (false).
- !(5 == 2) evaluates to 1 because the operand (5 == 2) is 0 (false).
- !(5 == 5) evaluates to 0 because the operand (5 == 5) is 1 (true).

Practice Task

Write a C++ program that asks user to input two integer values and then performs the addition, subtraction, multiplication, and division on those values. Something like shown below:

```
Enter first value: 25
Enter second value: 10
Addition is 35
Subtraction is 15
Multiplication is 250
Division is 2
```

Solution

```
#include <iostream>
using namespace std; int
main() {
        int num1;
int num2;

 cout << "Enter first number: ";
cin >> num1;

 cout << "Enter second number: ";
 cin >> num2;

 cout << "Addition is " << num1+num2 << endl;
cout << "Subtraction is " << num1-num2 << endl;
cout << "Multiplication is " << num1*num2 << endl;
cout << "Division is " << num1/num2 << endl;



        return 0;
}
```

Practice Task

Make some changes in the program written in the Activity above so that the output should be like the following

```
Enter first value: 25
Enter second value: 10
Addition of 25 and 10 is 35
Subtraction of 25 and 10 is 15
Multiplication of 25 and 10 is 250
Division of 25 and 10 is 2
```

```
#include <iostream>

using namespace std;

int main() {
 int num1;
int num2;

 cout << "Enter first number: ";
cin >> num1;

        cout << "Enter second number: ";
        cin >> num2;

        cout << "Addition of " << num1 << " and " << num2 << " is " << num1+num2 << endl;
cout << "Subtraction of " << num1 << " and " << num2 << " is " << num1-num2 << endl;
cout << "Multiplication of " << num1 << " and " << num2 << " is " << num1*num2 << endl;
        cout << "Division of " << num1 << " and " << num2 << " is " << num1/num2 << endl;


        return 0;
}
```

The logical NOT operator is denoted by "!". It reverses the bool outcome of the expression that immediately follows. Here's its truth table:

Exercises

1. Write a program that takes 3 values from user. Two values of integer and one value of float data type. Print each result on one line.
2. Write a program that inputs a five-digit integer, separates the integer into its Individual digits and prints the digits vertically. For example, if the user types 32156, the program should print:

   6
   5
   1
   2
   3
3. Make a program where it is asked form user to enter total amount, you have to answer how much ZAKAT to be paid on that amount. ZAKAT is the 2.5% of the total amount.

4. Write a program to take initialVelocity and acceleration from user, save them in respective data types and calculate FINAL VELOCITY as per following formula: FINAL VELOCITY = initialVelocity + acceleration
5. Take distance and time from user, save them in respective data types and calculate SPEED as per following formula:

$$Speed = \frac{distance}{time}$$

6. Write a program that finds the value of X by using given formula. Take value of 'a' and 'b' from user.

$$\mathbf{X = 2(a + b) - 2ab}$$

7. Write a program for mark sheet, where user should be able to enter the marks for five subjects. Then, your program should tell him/her his/her obtained marks and percentage. **Note**: You can assume each subject to be of 100 marks. So the total marks would become 500 for five subjects.