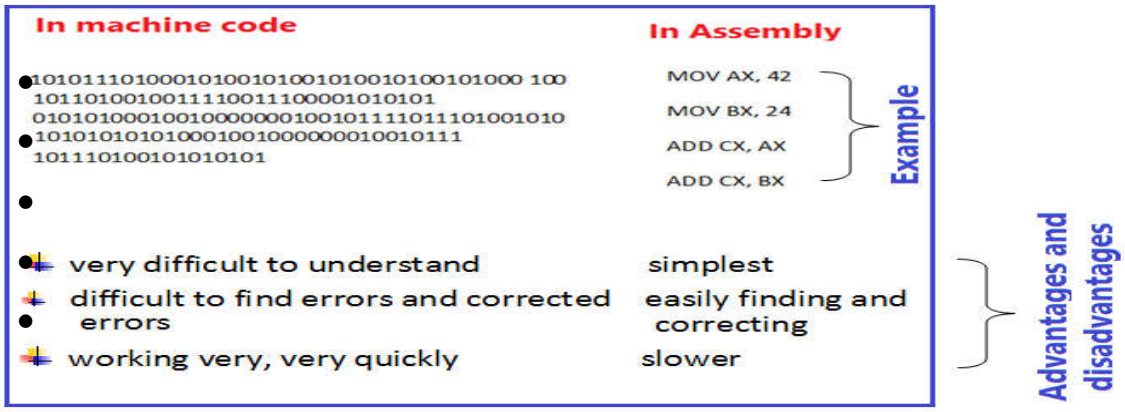
What is programming?

- Programming: is the process by which to determine how to deal with the data entered into the computer for the desired results.
- Computer Programming: is the process of providing(تزويد) the commands to computer to perform a specific task in a certain way .

Programming

- **Programming Language :** it's a sequence of instructions that convert An Algorithms (**in human language)** to the A program **(computer Language**) .
- Programming language can be classified into:
  - **Low Level Language** : in this type of languages the programmer can write programs must knowing the details of how the computer work, storage locations and details of the device like machine code and Assembly language, the fallowing example of adding the number (24 , 42) by these language .



- **High Level Language:** in this type of languages the programmer can write programs without the knowing details of how the computer work, storage locations and details of the device , like C++ , Java , …… , the fallowing example show part of program in C++ language .

```
void main(){
int x, y, z;
x = 1;
y = 12;
z = x + y ;}
```

Any Program in High Level language passing this stages

Program → interpreter/compiler → machine language

Difference between Compiler and Interpreter

No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

Basic elements of C++

C++ is a general-purpose programming language. ***C++ was derived from C, and is largely based on it.***

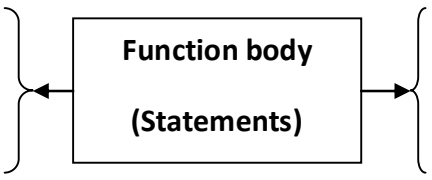
General Form of a C++ Program

Programming language is a set of rules, symbols, special words

- rules(syntax) – specifies legal instructions
- Symbols - special symbols (**+ - * !** …)
- Special Word (reserved words) (**int, float, double, char** …)
- A C++ program is a collection of one or more subprograms (functions)
- Function
  - Collection of statements
  - Statements accomplish a task
- Every C++ program must contain a function called **main**

Program structure in C++

```
int main ()
 {
   .
   .
   return 0;
 }
```




```
void  main()
   {
      .
      .
      .
   }
```

Where

- The **int** specifies that it returns an integer value
- The **void** specifies there will be no arguments

Example: Program in c++ to display "Welcome In Computer Programming Course"

```
#include <iostream.h>
 int main( )
  {
     cout   <<"   Welcome   In   Computer
                    Programming Course ";
    return 0;
  }
```

```
#include <iostream.h>
 void main( )
  {
     cout <<" Welcome In Computer
              Programming Course ";
  }
```

Program Result :

Welcome In Computer Programming Course

#include <iostream.h>	Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. **Preprocessor directives must be specified in their own line and do not have to end with a semicolon (;).** In this case the directive **#include <iostream.h>** tells the preprocessor to include input-output library in C++ .
int main() - void main()	Program execution begins with the main function. The entry point of every C++ program is **main()**.
Curly brackets { }	indicate the beginning and end of a function, which can also be called the function's body. The information inside the brackets indicates what the function does when executed.
;	In C++, the semicolon is used to terminate a statement. Each statement must end with a semicolon. It indicates the end of one logical expression.
Cout	is used in combination with the insertion operator, <<, to insert the data that comes after it into the stream that comes before.

Statements	A **block** is a set of logically connected statements, surrounded by opening and closing curly braces. For example
	{
	cout << "Welcome In Computer programming course ";
	
	return 0;
	}
	
	You can have multiple statements on a single line, as long as you remember to end each statement with a semicolon. failing to do so will result in an error.

return 0 ;	The line **return 0;** terminates the **main()** function and causes it to return the value 0 to the calling process. A non-zero value (usually of 1) signals **abnormal** termination

Note: The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines.

Exercise: Write a program in C++ to display :

Welcome In IS Dep.
I'm a C++ course
First Stage
Group A

Comments

Comments are explanatory statements(تعليمات توضيحية) that you can include in the C++ code to explain what the code is doing. The compiler ignores everything that appears in the comment, so none of that information shows in the result. There are two types of comment:

Comment type	Description	Example

Single-line comment	**// line comment**	#include <iostream.h>
		
		int main( )
		{
		// print "Welcome In IS Dep. ".
		cout << "Welcome In IS Dep. ";
		return 0;
		}

Multi-Line Comments	**/* block comment */**	include <iostream.h>
		
		int main( )
		{
		/* Welcome In IS Dep */
		
		/* Example for display
		Welcome In IS Dep
		*/
		
		cout << "Welcome In IS Dep. ";
		return 0;
		}

Note: Comments can be written anywhere, and can be repeated any number of times throughout the code. Within a comment marked with /* and */, // characters have no special meaning, and vice versa. This allows you to "nest" one comment type within the other.

Reserved Words (keywords)

Reserved words have a predefined meaning in C++ and that you cannot use as names for variables or anything else.

asm	dynamic_cast	new	template
auto	else	operator	this
bool	enum	private	throw
break	extern	protected	true
case	false	public	try
catch	float	register	typedef
char	for	reinterpret_cast	typeid
class	friend	return	union
const	goto	short	unsigned
const_cast	if	signed	using
continue	inline	sizeof	virtual
default	int	static	void
delete	long	static_cast	volatile
do	mutable	struct	wchar_t
double	namespace	switch	while

Note: *Keep in mind that the case of the keywords is significant. C++ is a case-sensitive language, and it requires that all keywords be in lowercase. For example,* ***RETURN*** *will not be recognized as the keyword* ***return****.*

Identifiers

Any item might define in a program is called an ***identifier.***

Rules for identifiers

- must begin with letter or the underscore _
- followed by any combination of numerals, letters or underscore
- recommend (نوصي) meaningful identifiers
- Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are ***reserved keywords (keyword)***.

Here are some correct and incorrect identifier names:

Correct	Incorrect	explain way incorrect
Count	1count	?
test23	hi!there	?
high_balance	high...balance	?
_name	_n ame	?

Note: The C++ language is a "**case sensitive**" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the **RESULT** variable is not the same as the result variable or the **Result** variable.

Use Meaningful Names

Variable names and other names in a program should at least hint at the meaning or use of the thing they are naming. It is much easier to understand a program if the variables have meaningful **names. Contrast the following**: **x = y * z;**
With the more suggestive: **distance = speed * time;**

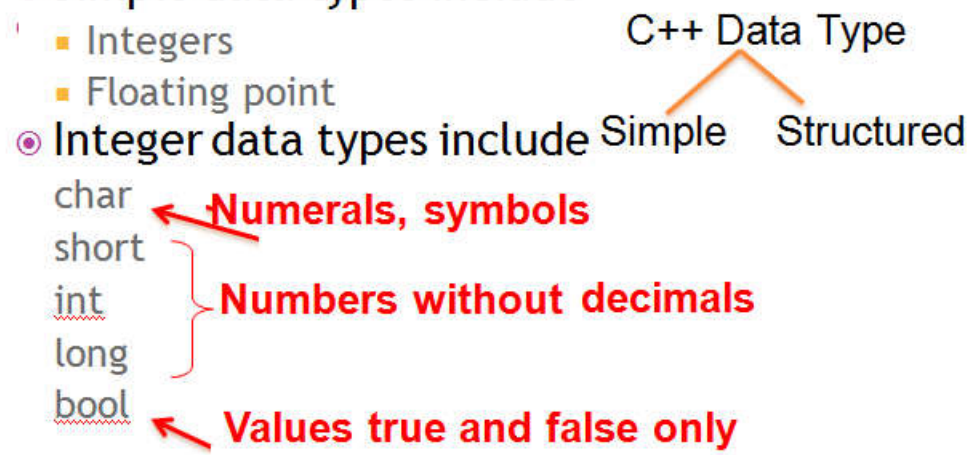
The two statements accomplish the same thing, but the second is easier to understand.

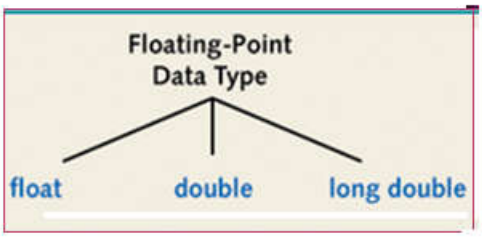
Data Types

- When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.
- The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++.
- In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers. In the following figure and table shown summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

DATA TYPES

- Simple data types include
  - Integers
  - Floating point
- Integer data types include
  - char
  - short
  - int
  - long
  - bool
- Different floating-point types
- Note that various types will
  - have different ranges of values
  - require different amounts of memory





Data type	Size(bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65535
int	4	-2147483648 to +2147483647
unsigned int	4	0 to 4294967295
long	4	-2147483648 to +2147483647
Unsinged long	4	0 to 4294967295
float	4	-3.4e-38 to +3.4e-38
double	8	1.7 e-308 to 1.7 e+308
long double	8	1.7 e-308 to 1.7 e+308
bool	1 bit	
void	-	-
wchar_t	2 or 4	1 wide character

Variables

- Programs manipulate data such as numbers and letters. C++ and most other programming languages use programming constructs known as ***variables*** to name and store data.
- Creating a **variable** reserves a memory location, or a space in memory for storing values. The compiler requires that you provide a **data type** for each variable you declare.
- Variables like small ***blackboards*** , can be written and then can be changed.
- The number or other type of data held in a variable is called its **value .**
- All integer, floating-point, and other values used in a program are stored in and retrieved from the computer's memory. Conceptually, locations in memory are arranged like the

rooms in a large hotel, and each memory location has a ***unique address***, like room numbers in a hotel .

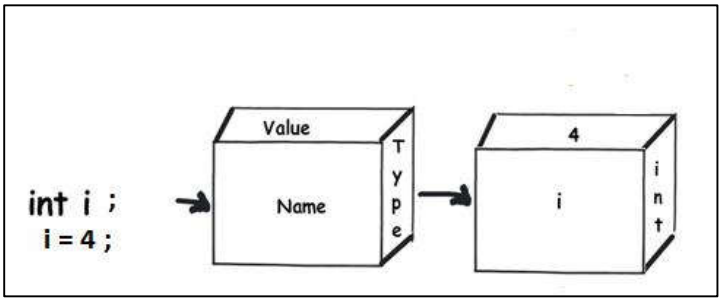
Variable Declarations

All variables must be declared before they are used. The syntax for variable declarations is as follows:

Syntax **Type_name variable_name1 , variable_name2, …… ;**

Example int count , number_of_students ;
double distance ;

Where:
type_name : must be a valid data type
Variable_Name_1, Variable_Name_2, . . . ; or (variable_list) : may consist of one or more Identifier names separated by commas (,). **Each Variable name** must follow the rules of identifier name.



Example for variable declaration using some of data type

Integer declaration : int x ; long int y ; short int z;	**Floating Point Numbers declaration:** float x1 ; double y1; long double z1;
Character declaration : char ch ;	**Boolean declaration :** bool b1 ;// true or flase

Exercise:

valid variable declaration	Not valid	Explain way not valid ?
int a, b, c;	int a; b,c ;	?
int a; int b; int c;	int a; int b , int c;	?

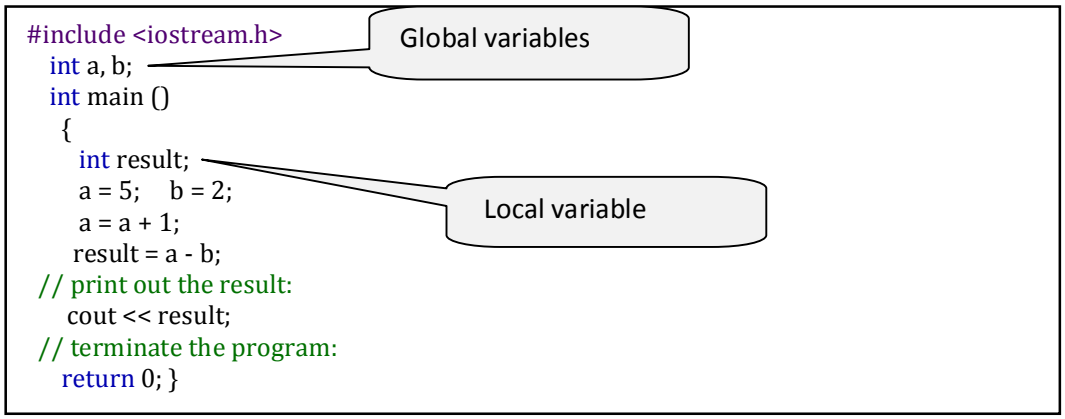
Example : To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory :

// operating with variables
#include <iostream.h>
int main ()
{
// declaring variables:
int a, b;
int result;
// process:
a = 5; b = 2;
a = a + 1;
result = a - b;
// print out the result:
cout << result;
// terminate the program:
return 0;
}
	Explain the execution steps of the program :

Scope of variables

A variable can be either of

- **Global** : A global variable is a variable declared in the main body of the source code, outside all functions.
- **Local**: while a local variable is one declared within the body of a function or a block.



Note: *The* ***scope*** *of* ***local variables*** *is* ***limited to the block*** *enclosed in braces (****{}****) where they are declared. T*his means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

Initialization of variables

When declaring local variable, its value is by default undetermined. But you may want a variable to store a value at the same moment that it is declared. In order to do that, you can initialize the variable.

The syntax for initialization variables is as follows:

type identifier = intial_value;

vaild examples	**Not valid examples**	**Way ???**

int a= 5 ;
int a=b=c= 0 ;
int a=5 , d, f=8;
	int a=5 ; b= 5;
	int a= b=0, int c= 0;
	int a=5 , d; f=8;	

// initialization of variables
#include <iostream.h>
int main ()
{
int a=5; // initial value = 5
int b=2; // initial value = 2
int result; // initial value
a = a + 3; result = a - b;
cout << result;
return 0;
}
	Result:???

Constants

Constants refer to fixed values that the program cannot alter. Constants can be of any of the basic data types. The way each constant is represented depends upon its ***type***. Constants are also called ***literals.***

Type	examples
Integer Numerals	1776 707 -273
Floating-Point Numerals	3.14159 6.02e23 // 6.02 x 10^23 1.6e-19 // 1.6 x 10^-19 3.0
Characters	'z' 'p'
Strings	"Hello world" "How do you do?"
Bool	There are only two valid Boolean values: **true** and **false**.

Declared constants (const)

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable *except that their values cannot be modified after their definition* :

```
const int pathwidth = 100;
const float pi=3.14 ;
```

Defined constants (#define)

You can define your own names for constants that you use by using the #define Preprocessor directive. Its Format is:

 # define identifer value

Example :

```
// defined constants: calculate circumference

#include <iostream.h>

#define PI 3.14159
#define NEWLINE  '\n'

int main ()
{
   double r=5.0;            // radius
   double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

 return 0;
}
```

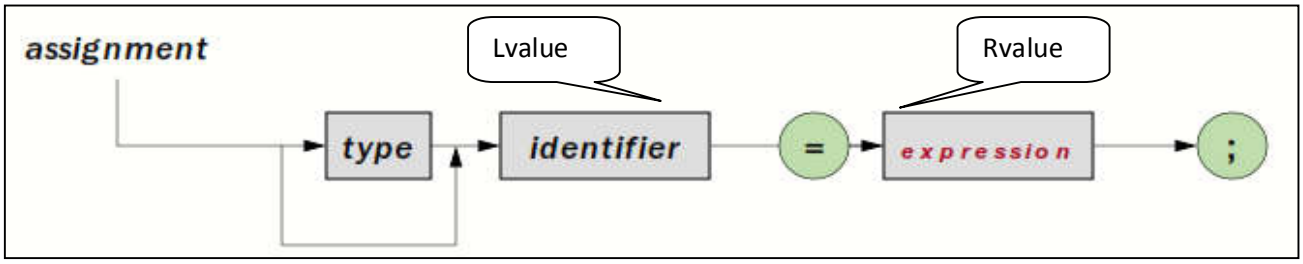
result :

```
31.4159
```

Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators.

Assignment (=) **:** The assignment operator (assignment statement) assigns a value to a variable. Its general syntax as follow :



- The **lvalue** has to be **a variable** whereas the **rvalue** can be either **a constant**, **a variable**, the result of an operation or any combination of these (expression) .

- The most important rule when assigning is the ***right-to-left* rule:** The assignment operation always takes place from **right to left,** and never the other way.

- Arithmetic expression is **any combination** of ***simple value***, **function call**, **binary expression**, **and unary expression.**

Where :

- ***Simple value :*** *constant number , string constant , character constant , identifier.*
- **Unary operator (العوامل الاحادية) are (+, - ,-- , ++).**

Example1 **(arithmetic expression)**

1. **double a= 10 + y / 5 – m1;**
2. **double a= (x + 2) / y * 5 * 9;**
3. **int a= y*10+(2-1);**
4. **int z ; z= sin(45) * 34 ;**
5. **float m*= m + x ++ ;**

Example2

// assignment operator	Result: ? ?
#include <iostream.h> **int main ( )** **{** **int a, b;** **a = 10;** **b = 4;** **a = b;** **b = 7;** **cout << "a:";** **cout << a;** **cout << " b:";** **cout << b;** **return 0;** **}**	

- the assignment operation can be used as the **rvalue** (or part of an rvalue) for another assignment operation. For example: **a = 2 + (b = 5);** is equivalent to:

```
b = 5;
a = 2 + b;
```

- The following expression is also valid in C++: **a = b = c = 5;** It assigns 5 to the all the three variables: a, b and c.

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

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

+	**Addition**
-	**Subtraction**
*****	**Multiplication**
/	**Division**
%	**Modulo** **(remainder of an integer division)**

- Arithmetic Operators require two variables to be evaluated.
- Modulo is the operation that gives the remainder of a division of two values.
- **For example**, if we write: **a = 11 % 3; // a=2**

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
x += y ;	x = x + y;
a-= 5;	a = a - 5 ;
price *=units + 1 ;	price =price * (units + 1) ;
a /= b;	a = a / b;
c %= 2;	C = c % 2;

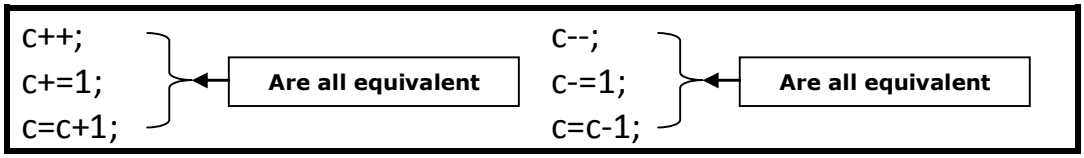
Example :

```
#include <iostream.h>
int main ()
{
    int a, b=3;
    a = b;
    a+=2;
   cout << a;
   return 0;
 }
```

Result: ??

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 (its unary operation) . They are equivalent to +=1 and to -=1, respectively. Thus:



Note:

- A characteristic of this operator is that it can be used both as a **prefix** and as a **suffix**. That means that it can be written either before the variable identifier **(++a**) or after it (**a++).**

- Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning:
  - **In the case** that the increase operator is used as **a prefix (++a)** the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression.
  - **in case** that it is used as a **suffix (a++)** the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1	Example 2
B=3; A=++B; // A contains 4, B contains 4	B=3; A=B++; // A contains 3, B contains 4

Note::

In Example 1, B is increased before its value is copied to A.

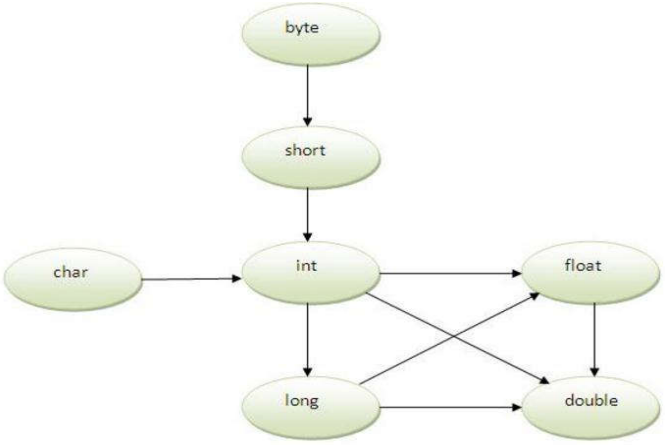
In Example 2, the value of B is copied to A and then B is increased.

Other Examples

```
a = 1;          // a = 1
b = ++a;        // a = 2, b = 2
c = a++;        // a = 3, c = 2

a = 5; b = 3;
n = ++a + b--; // a = 6, b = 2, n = 9

a = 5; b = 3;
n = ++a * ++b; // a = 6, b = 4, n = 24
n = a++ * b++; // a = 6, b = 4, n = 15

a = 5; b = 3;
n = a++ * --b; // a = 6, b = 2, n = 10
```

Automatic type conversion

If an expression contains operands of different types, an (to the type which is highest in the following hierarchy) is performed. ***Automatic type conversion***



Some of binary operation yield implicit type conversions

Integer / integer	= integer	► 39/7=5
Integer / float	= float	► 39/7.0 =5.57
float / integer	= float	► 39.0/7 =5.57
float / float	= float	► 39.0/7.0=5.57

while 39%5=7, since 39=7*5+4

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

In the term *relational operator*, relational refers to the relationships that values can have with one another. In the term ***logical operator***, logical refers to the ways these relationships can be connected. 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.

Note: C++ fully supports the zero/non-zero concept of **true** and **false**.

However, it also defines the **bool** data type and the Boolean constants **true** and **false**.

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

Here there are some examples:	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,**

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

NOT (!) Table:

A	!A
T	F
F	T

NOT (!) Table:

A	!A
1	0
0	1

Example

```
!(5 == 5)   // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)   // evaluates to true because (6 <= 4) 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.

AND (&&) Table:

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

AND (&&) Table:

A	B	A && B
1	1	1
1	0	0
0	1	0
0	0	0

The operator ||: corresponds with Boolean logical operation **OR**. This operation results true if either one of its two operands is **true**, thus being false only when both operands are false themselves.

OR (| |) Table:		
A	B	A | | B
T	T	T
T	F	T
F	T	T
F	F	F

OR (| |) Table:		
A	B	A | | B
1	1	1
1	0	1
0	1	1
0	0	0

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.
Conditional operator which can be used to replace **if...else** statement.
Its format is:

condition? result1 : result2

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

Example:

```
(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>

int main ()
 {
   int a,b;
   a=10;
    b= (a == 1) ? 20: 30;

    cout << b;
   return 0;
 }
```

Result ?

Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

a = 5 + 7 % 2

We may doubt (شك) if it really means:

a = 5 + (7 % 2) // with a result of 6, or
a = (5 + 7) % 2 // with a result of 0

The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Operator	Description
* / %	Multiplication/division/modulus
+ -	Addition/subtraction

High ⇩ Low

Operator	Description
!	Logical not
&&	Logical AND
||	Logical OR

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and), as in this example:

a = 5 + 7 % 2; ***can be written either as*** **a = 5 + (7 % 2); *or* a = (5 + 7) % 2;**

Examples

1. **x = 3 + 4 + 5;**
2. **z *= ++y + 5;**
3. **a || b && c || d;**

Input and output

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).

Examples

```
1. cout << "Output sentence";
2. cout << 120;
3. cout << x;
4. cout << "Hello";
5. cout << Hello;
6. cout << "Hello, " << "I am " << "a C++ statement";
7. cout << "Hello, I am " << age << " years old and my department is " << dep;
8. cout << "This is a sentence.";
   cout << "This is another sentence.";
```

Note: In order to perform a line break on the output we must explicitly insert a **new-line** character into cout . In C++ a new-line character can be specified as **\n** (backslash, n) it's also called **escape codes.** Additionally, to add a new-line, you may also use the endl manipulator.

Example1:

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

First sentence.
Second sentence.
Third sentence.

Example2:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

Would print out:

First sentence.
Second sentence.

Here you have a list of some of such escape codes:

	Escape Sequence	Description
\n	Newline	Cursor moves to the beginning of the next line
\t	Tab	Cursor moves to the next tab stop
\ b	Backspace	Cursor moves one space to the left
\r	Return	Cursor moves to the beginning of the current line (not the next line)
\\	Backslash	Backslash is printed
\'	Single quotation	Single quotation mark is printed
\"	Double quotation	Double quotation mark is printed

Example 3:

```
cout << "The total is\t "<< sum << endl
```

Would print out , if assume sum value is 100

The total is 100

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:

Example:

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

- **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 can also use cin to request more than one data input from the user:

cin >> a >> b;

is equivalent to:

cin >> a;
cin >> b;

Note: In both cases the user must give two data, one for variable **a** and another one for variable **b** that may be separated by any valid blank separator: **a space**, a tab character or **a newline**.

```
// Example 1
#include <iostream.h>
 int i;
int main ()
{
 cout << "Please enter an integer value: ";
 cin >> i;
 cout << "\n The value you entered is " << i;
```

```
// Example 2
#include <iostream.h>
  // this program read two variable then swap their values
  int x,y,z ;
 void main()
  {
    cout<<"Enter x,y :";
    cin >> x>>y ;
```

```
 cout << " and its double is " << i*2 << ".\n";
 return 0;
}
```

```
    cout<<"before  swap:";
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
    z=x;  x=y;   y=z;
   cout<<"After  swap:";
   cout<<"x="<<x<<endl;
   cout<<"y="<<y<<endl;
  }
```

```
// Example  3
#include <iostream.h>
 /* this program read two number and find the result
of  application the binary operation (+,-,&,/,%)  */
 int  x,y ;
void main()
 {
   cout<<"Enter x,y :";
  cin >> x>>y ;
  cout<<"x+y="<<x+y;
  cout<<"x-y="<<x-y;
  cout<<"x*y="<<x*y;
  cout<<"x/y="<<x/y;
   cout<<"x%y="<<x%y;
}
```

```
// Example  4
#include  <iostream.h>
 // this program find the product table for any input number
 int x;
 void main ()
  {
    cout <<"Enter x=";   cin>>x ;
    cout << " x * 1 ="<<x*1 <<endl;
    cout << " x * 2 ="<<x*2 <<endl;
    cout << " x * 3 ="<<x*3 <<endl;
    cout << " x * 4 ="<<x*4 <<endl;
    cout << " x * 5 ="<<x*5 <<endl;
    cout << " x * 6 ="<<x*6 <<endl;
    cout << " x * 7 ="<<x*7 <<endl;
    cout << " x * 8 ="<<x*8 <<endl;
    cout << " x * 9 ="<<x*9 <<endl;
    cout << " x * 10 ="<<x*10 <<endl;
    cout << " x * 11="<<x*11 <<endl;
    cout << " x * 12="<<x*12 <<endl;
}
```

Questions in Sequence

1.Write a program that outputs the following text on screen:
what a happy day!
Oh what
a happy day!
Oh yes,

2. prints each of the following C++ statements is performed? Assume x = 2 and y = 3

a.
```
cout<< x;
```

b.
```
cout<< x + x;
```

c.
```
cout<< "x=";
```

d.
```
cout<< "x = "
```

e.
```
cout<< x + y << " = " << y;
```

f.
```
z = x + y;
```

g.
```
cin>> x >> y;
```

h.
```
// cout<< "x +
```

i.
```
cout<< "\n";
```

k.
```
cout<< "*\n**\n***\n****\n*****" <<endl;
```

3.The following program contains several errors:

```
#include <stream>
 int main
  {
   cout<< "If this text",
   cout>> " appears on your display, ";
   cout<< " endl;"
   cout<< 'you can pat yourself on '<< " the back!" <<endl.
   return 0;
 )
```

Resolve the errors and run the program to test your changes.

4.Write a program to calculate and print the product of three integers.

5. Write a program that asks the user to enter two numbers and prints the sum, product and difference of the two numbers.

6. Write a program that reads in the radius of a circle as an integer and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for pi.

7. Write a program that inputs a five-digit integer, separates the integer into its individual digits and prints the digits.
For example, if the user types in 42339, the program should print:4 2 3 3 9

8. Write a program that converts Celsius to Fahrenheit according to the following equation:

$$F = 9/5\ C+32$$

9.Write a program to print out the perimeter of a rectangle given its height and width.
Hint: perimeter = 2 (width+height)

10.Write a program that converts kilometers per hour to miles per hour.
Hint: miles = (kilometers=0.6213712)

11.Write a program that takes hours and minutes as input and outputs the total number of minutes (1 hour 30 minutes = 90 minutes).

12.Write a program that takes an integer as the number of minutes and outputs the total hours and minutes (90 minutes = 1 hour 30 minutes).

13. Write a program to find the value of X:

X=2y+5z where y= r^2+7