

FORMS OF WRITE C++ PROGRAM

FIRST FORM : We will use this form in our lectures, this is the simplest one .

```
# include <iostream>
using namespace std;
int main ()
{
    cout << " This is My program \n \n";
    cout << " I am a c++ programer \n \n ";
    return 0 ;
}
```

SECOND FORM : We must use (std ::) before all instruction (included in standard (std) libraries) in our program instead of using the second line in first form (**using namespace std;**)

```
# include <iostream>
int main ()
{
    std :: cout << " This is My program \n \n";
    std :: cout << " I am a c++ programmer \n \n ";
    return 0 ;
}
```

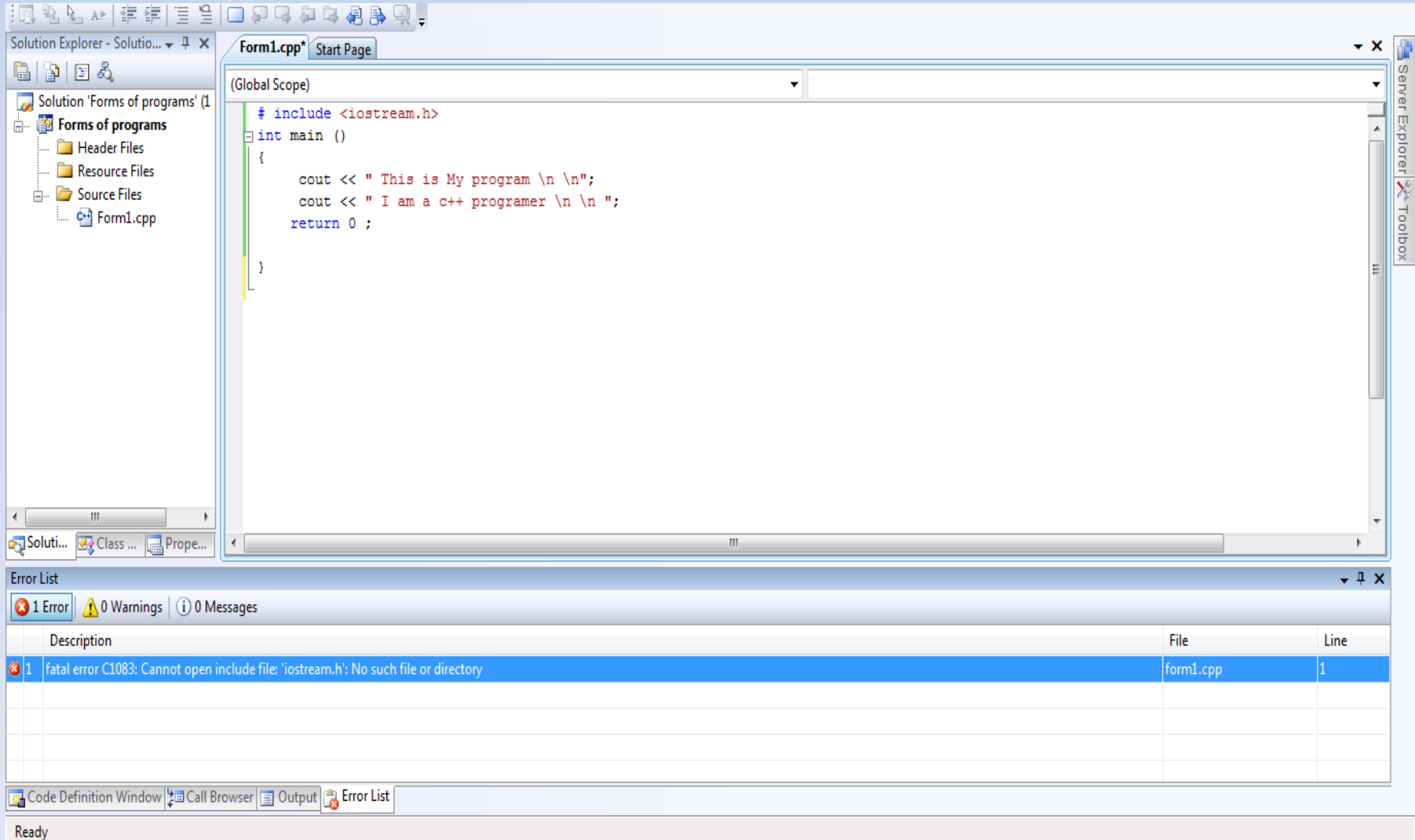
THIRD FORM : In this form, we must write the second line with each instruction included in these library instead of using the second line in first form (**using namespace std;**) for one time.

```
# include <iostream>
using std :: cout;
int main ()
{
    cout << " This is My program \n \n";
    cout << " I am a c++ programer \n \n ";
    return 0 ;
}
```

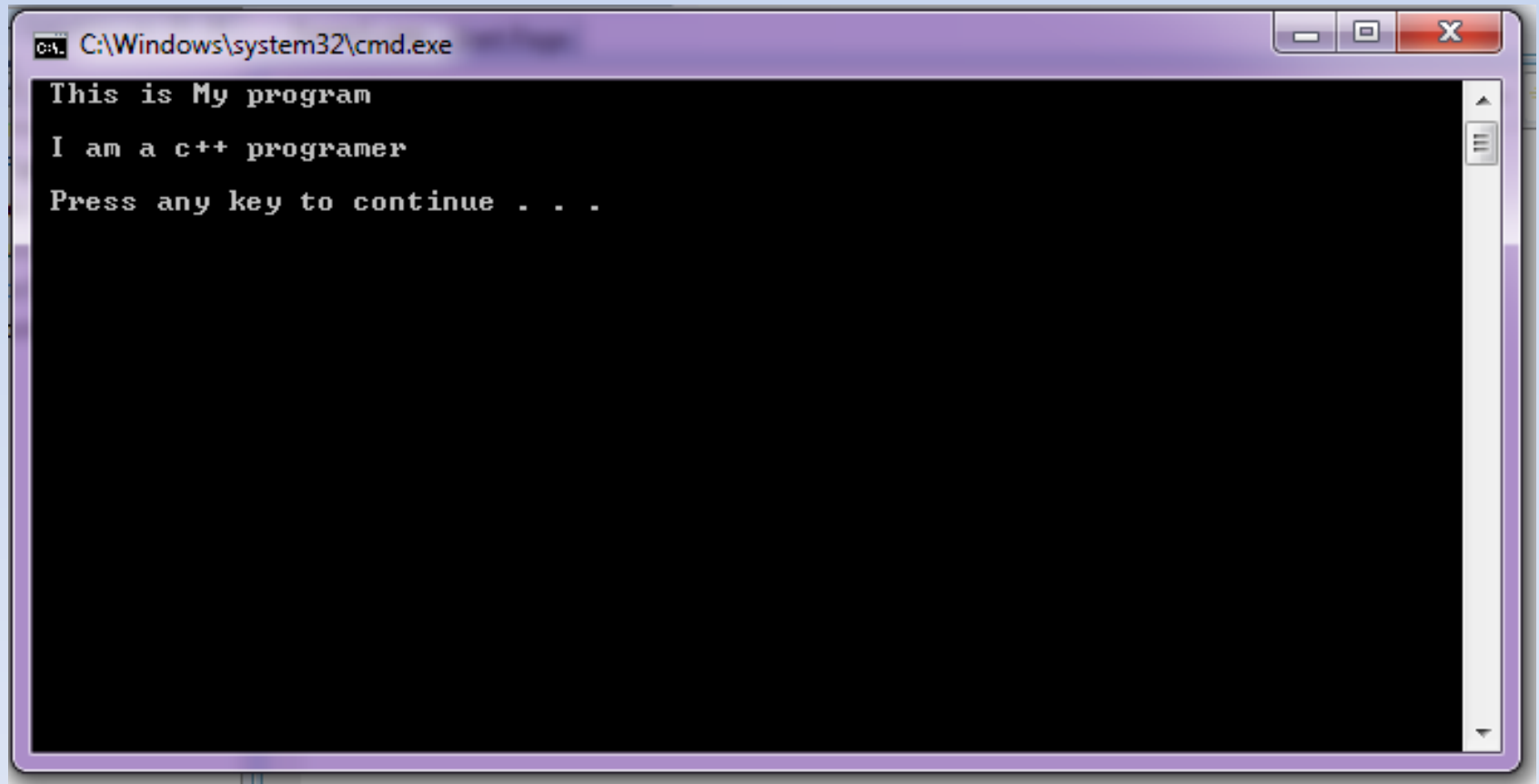
FOURTH FORM : This form was using in old version , by added (.h) with <iostream> instead of using the second line in first form (using namespace std;).

```
# include <iostream.h>
int main ()
{
    cout << " This is My program \n \n";
    cout << " I am a c++ programer \n \n ";
    return 0 ;
}
```

Note : the fourth form may give us errors with new version . Therefore, we prefer the first form all times .



When we press to (Ctrl + F5) to run our program, we will get the same output with all previous forms



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt area is black with white text. The text displayed is: 'This is My program', 'I am a c++ programer', and 'Press any key to continue . . .'. There is a vertical scrollbar on the right side of the command prompt area.

```
C:\Windows\system32\cmd.exe  
This is My program  
I am a c++ programer  
Press any key to continue . . .
```

Variables. Data Types.

Let us think that I ask you to retain the number 6 in your mental memory, and then I ask you to memorize also the number 3 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 2 to the first number I said, you should be retaining the numbers 6 (that is $6+2$) and 3 in your memory. Values that we could now for example subtract and obtain 5 as result.



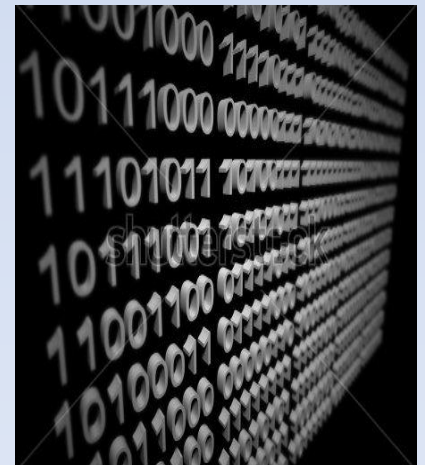
The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

`a = 6;`

`b = 3;`

`a = a + 2;`

`result = a - b;`



Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can **store** millions of numbers like these at the same time .

Therefore, we can define a **variable** as a portion of memory to **store** a determined value.

Each **variable** needs an identifier that **distinguishes** it from the others,

for example, in the previous code the variable identifiers were **a**, **b** and **result**, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Identifiers:

A valid identifier is a **sequence** of one or more **letters**, **digits** or **underscore characters** (**_**). Neither spaces nor punctuation marks or symbols can be part of an identifier.

Only letters, digits and single underscore characters are valid. In addition, variable identifiers **always** have to begin with a letter. They **can also begin with an underline character** (**_**), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. **In no case they can begin with a digit.**

Another rule that you have to consider when **creating** your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are **reserved keywords**. The standard reserved keywords are:

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	Else	Inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Additionally, alternative representations for some **operators** cannot be used as **identifiers** since they are reserved words under some circumstances:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Your compiler may also include some additional specific reserved keywords.

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. These are three different variable identifiers.

Fundamental 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++.

A byte can store a **relatively** small amount of data. In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as **long numbers or non integer numbers**.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. For example:

```
int a;  
float X;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier **a**. The second one declares a variable of type float with the identifier **X**. Once declared, the variables **a** and **X** can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (**a**, **b** and **c**), all of them of type int, and has exactly the same meaning as:

```
int a;  
int b;  
int c;
```

The integer data types **char**, **short**, **long** and **int** can be either signed or unsigned **depending** on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specified signed or the specified unsigned before the type name. For example:

unsigned short int NumberOfBrothers;
signed int MyAccountBalance;

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

int MyAccountBalance;

with exactly the same meaning (with or without the keyword signed)

An exception to this general rule is the **char** type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable.

short and **long** can be used alone as type specifies. In this case, they refer to their respective integer fundamental types: **short** is equivalent to **short int** and **long** is equivalent to **long int**. The following two variable declarations are equivalent:

short Year;

short int Year;

Finally, signed and unsigned may also be used as stand alone type specifies, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

unsigned Average;

unsigned int Average;

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 proposed at the beginning of this lecture:

```

≡ // operating with variables
└ #include <iostream>
  using namespace std;
≡ int main ()
  {
    // declaring variables:
    int a, b;
    int result;
    // process:
    a = 6;
    b = 3;
    a = a + 2;
    result = a - b;
    // print out the result:
    cout << "\n \n \t The result is : " << result << "\n \n \n" ;
    // terminate the program:
    return 0;
  }

```

to RUN the program, we will press (Ctrl +F5) to get the output

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\system32\cmd.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The main area is black with white text. It displays "The result is : 5" on one line and "Press any key to continue . . ." on the next line. A vertical scrollbar is visible on the right side of the command prompt area.

```
C:\Windows\system32\cmd.exe

The result is : 5
Press any key to continue . . .
```