INLINE FUNCTION:

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.
A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example which makes use of inline function to returns max of two numbers:

```
#include <iostream>

inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

int main( )
{

   cout << "Max (20,10): " << Max(20,10);
   cout << "Max (0,200): " << Max(0,200);
   cout << "Max (100,1010): " << Max(100,1010);
   return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

STORAGE CLASS:

A storage class defines the scope (visibility) and life time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes which can be used in a C++ Program

- auto
- register
- static
- extern
- mutable

The auto Storage Class:

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e. local variables.

The Register Storage Class:

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class:

The **static** storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <iostream>

// Function declaration
void func(void);

static int count = 10; /* Global variable */

main()
```

```
{
    while(count--)
    {
       func();
    }
    return 0;
}
// Function definition
void func( void )
{
    static int i = 5; // local static variable
    i++;
    std::cout << "i is " << i ;
    std::cout << " and count is " << count << std::endl;
}
```

When the above code is compiled and executed, it produces following result:

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

The extern Storage Class:

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.
When you have multiple files and you define a global variable or function which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another files.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include <iostream>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

Second File: write.cpp

```
#include <iostream>

extern int count;

void write_extern(void)
{
  std::cout << "Count is " << count << std::endl;
}
```

Here *extern* keyword is being used to declare count in another file. Now compile these two files as follows:

```
$g++ main.cpp write.cpp -o write
```

This will produce **write** executable program, try to execute **write** and check the result as follows:

```
$./write
5
```

The mutable Storage Class

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override constness. That is, a mutable member can be modified by a const member function.

FUNCTIONS IN C:

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.
Functions serve two purposes.

- They allow a programmer to say: `this piece of code does a specific job which stands by itself and should not be mixed up with anything else'.
- Secondly, they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.

Function Declaration and Definition:

A function declaration does not have any body and they just have their interfaces. A function declaration is usually declared at the top of a C source file, or in a separate header file.
On the other hand, when a function is defined at any place in the program then it is called function definition. At the time of definition of a function actual logic is implemented with-in the function.

A function declaration (also known as ***function Prototype***) consists of four parts:

- Function type/ Return type
- Function name
- Parameter list
- Semicolon

Syntax for function declaration**:** <Return type> <Function name> (Parameter list);

Example,   int add_data(int, int);

Actual Parameters and Formal Parameters:

Parameters (also known as arguments) are used in three places:

- In function declaration
- In function call
- In function definition or function body

The parameters used in function prototypes and function body are called ***formal Parameters***. Again those parameters which are used in function call are called ***Actual parameters.***

Category of functions:

Depending on whether arguments are present or not and whether a value is returned or not, a function may belong to one of the following categories

1. Functions with **no arguments** and **no return type**.
2. Functions with **arguments** and **no return type**.
3. Functions with **arguments** and **return type**.
4. Functions with **no arguments** and **return type**.
5. Functions with **multiple return types**.

What are variables?

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.

A variable name can be chosen by the programmer in a meaningful way with referenced to the program.

Local variables
A local variable is a variable that is declared inside a function. These variables only exist inside the specific function that creates them. They are unknown to other functions and also to the main program. A local variable can only be used in the function where it is declared. As such, they are normally implemented using a stack. A local variable do not exist once the function that created it is completed. They are recreated each time a function is executed or called.

Global variables (also known as External variables)
A global variable is a variable that is declared outside **all** functions. These variables can be accessed (ie known) by any function within the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

Friend Functions:

A **friend** function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members. They are normal external functions that are given special access permissions.
A friend function is declared with the '***friend***' keyword inside the class where it wants be a friend.

A friend function can be accessed without any object name and dot (.) operator like normal function. A **friend** function is declared by the class that is granting access. The **friend** declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

Post increment & Pre Increment:

Increment and **decrement operators** are unary operators that *add* or *subtract* 1 from their operand, respectively. In C-like languages, the increment operator is written as ++ and the decrement operator is written as --. Both pre-increment and post-increment increment the value. The difference is in what they return. Post-increment returns the original value, while pre-increment returns the incremented value.

The increment operator increases the value of its operand by 1. Similarly, the decrement operator decreases the value of its modifiable arithmetic operand by 1.
Examples

The following C code fragment illustrates the difference between the *pre* and *post* increment and decrement operators:

```
int  x;
int  y;

// Increment operators
x = 1;
y = ++x;    // x is now 2, y is also 2
y = x++;    // x is now 3, y is 2

// Decrement operators
x = 3;
y = x--;    // x is now 2, y is 3
y = --x;    // x is now 1, y is also 1
```

++x is pre-increment and x++ is post-increment,
i.e in the first case, x is incremented before being used and in the second case, x is incremented after being used.

<u>**Virtual Function in C++ Programming**</u>

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, we have to use the keyword ***virtual***. When a class containing virtual function is inherited, the derived class redefines the virtual function to perform different task.

When a Base class pointer point to derived class object, using base class pointer if we call some function which is in both classes, then base class function is invoked. But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.

```cpp
#include <iostream>
using namespace std;
class B
{
    public:
       void display()
         { cout<<"Content of base class.\n"; }
};

class D : public B
{
    public:
       void display()
         { cout<<"Content of derived class.\n"; }
};

int main()
{
    B *b;
    D d;
    b->display();

    b = &d;    /* Address of object d in pointer variable */
    b->display();
    return 0;
}
```

Note: An object(either normal or pointer) of derived class is type compatible with pointer to base class. So, b = &d; is allowed in above program.

Output

```
Content of base class.
Content of base class.
```

In above program, even if the object of derived class d is put in pointer to base class, display( ) of the base class is executed(member function of the class that matches the type of pointer).

Implementation of Virtual Functions

If you want to execute the member function of derived class then, you can declare display( ) in the base class virtual which makes that function existing in appearance only but, you can't call that function. In order to make a function virtual, you have to add keyword **virtual** in front of a function.

```
/* Example to demonstrate the working of virtual function in C++ programming. */

#include <iostream>
using namespace std;
class B
{
    public:
     virtual void display()      /* Virtual function */
         { cout<<"Content of base class.\n"; }
};

class D1 : public B
{
    public:
       void display()
         { cout<<"Content of first derived class.\n"; }
};

class D2 : public B
{
    public:
       void display()
         { cout<<"Content of second derived class.\n"; }
};

int main()
```

```
{
    B *b;
    D1 d1;
    D2 d2;

/* b->display();  // You cannot use this code here because the function of base class is virtual.
*/

    b = &d1;
    b->display();   /* calls display() of class derived D1 */
    b = &d2;
    b->display();   /* calls display() of class derived D2 */
    return 0;
}
```

Output

```
Content of first derived class.
Content of second derived class.
```

After the function of base class is made virtual, code b->display( ) will call the display( ) of the derived class depending upon the content of pointer.

In this program, display( ) function of two different classes are called with same code which is one of the example of polymorphism in C++ programming using virtual functions.
- See more at: http://www.programiz.com/cpp-programming/virtual-functions#sthash.R5EABiM1.dpuf

STORAGE CLASS in C language:

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program. These specifiers precede the type that they modify. There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

The auto Storage Class:

The **auto** storage class is the default storage class for all local variables.

```
{
  int roll_no;
  auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e. local variables.

The Register Storage Class:

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
            register int  pages;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class:

The **static** storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

In C, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

The extern Storage Class:

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.
In general, the *extern* is used to declare a global variable or function in another files.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions.