8. C++ Inline Function

All the member functions defined inside the class definition are by default declared as Inline. Let us have some background knowledge about these functions.

You must remember Preprocessors from C language. Inline functions in C++ do the same thing what Macros do in C. Preprocessors were not used in C++ because they had some drawbacks.

❖ Drawbacks of Macro

In Macro, we define certain variable with its value at the beginning of the program, and everywhere inside the program where we use that variable, it's replaced by its value on Compilation.

1) Problem with spacing:

Let us see this problem using an example,

```
#define G(y) (y+1)
```

Here we have defined a Macro with name G(y), which is to be replaced by its value that is (y + 1) during compilation. But, what actually happens when we call G(y),

```
G(1) //Macro will replace it
```

The preprocessor will expand it like,

```
(y) (y+1) (1)
```

You must be thinking why this happened, this happened because of the spacing in Macro definition. Hence big functions with several expressions can never be used with macro, so Inline functions were introduced in C++.

2) Complex Argument Problem:

In some cases such Macro expressions work fine for certain arguments but when we use complex arguments problems start arising.

#define MAX(x,y) x>y?1:0

Now if we use the expression,

if(MAX(a&0x0f,0x0f)) // Complex Argument

Macro will Expand to,

if(a&0x0f > 0x0f ? 1:0)

Here precedence of operators will lead to problem, because precedence of & is lower than that of > , so the macro evaluation will surprise you. This problem can be solved though using parenthesis, but still for bigger expressions problem will arise.

3) No way to access Private Members of Class:

With Macros, in C++ you can never access private variables, so you will have to make those members public, which will expose the implementation.

```
class Y
{
 int x;  public :
 #define VAL(Y::x) // Its an Error
}
```

❖ **Inline Functions:**

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword **inline** with them.

For an inline function, declaration and definition must be done together. For example,

```
inline void fun(int a)
{
  return a++;
}
```

❖ **Some Important points about Inline Functions**

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to **code bloat**, and might affect the speed too.

3. Hence, it is advised to define large functions outside the class definition using scope resolution :: operator, because if we define such functions inside class definition, then they become inline automatically.
4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

❖ **Access Functions:**

We have already studied this in topic Accessing Private Data variables inside class. We use access functions, which are inline to do so.

```
class Auto
{
 int i;  public:
 int getdata()
  {
   return i;
  }
 void setdata(int x)
 {
  i=x;
 }
};
```

Here getdata() and setdata() are inline functions, and are made to access the private data members of the class Auto. getdata (), in this case is called **Accessor** function and setdata() is a **Mutator** function.

There can be overloaded Accessor and Mutator functions too. We will study overloading functions in next topic.

❖ **Limitations of Inline Functions:**

1. The compiler is unable to perform inlining if the function is too complicated. So we must avoid big looping conditions inside such functions. In case of inline functions, entire function body is inserted in place of each call, so if the function is large it will affect speed and memory badly.
2. Also, if we require address of the function in program, compiler cannot perform inlining on such functions. Because for providing address to a function, compiler will have to allocate storage to it. But inline functions doesn't get storage, they are kept in Symbol table.

C++ Inline function

* It is a process in which the compiler places a copy of the code of that function at each point where the function is called compile time.

```
#include<iostream>
using namespace std;
void sum(int x, int y)
{
    int z;
    z = x+y;
    cout<<z;
    return 0;
}
main()
{
  int a=10, b=20, c=30
  sum(a,b);
  sum(a,c);
}
```

(normal function)

In this code the sum() is called 2 times but if we writes inline then we can have a copy of full code at calling of function.

~~Note: It is~~ In normal function it is again and again the resources to move in another class that is the reason why we use inline function

Data is known as datamember and function is known as function member.

* Inline Member function: function defined within a class.
* Non-Inline Member function: function defined outside the class.

(To convert non-inline to inline function)
{using inline outside}

```
class A
{
  int a;
  public: void ip()
          {
            cin>>a;
          }
  void op()
  { obj;
```

```
inline void A::op() {using inline will make it
                     inline member function}
{
  cout<<a;
}
main()
{
  obj.ip()
  obj.op()
}
```

or other method is writting op() in class only instead writting inline outside

* Note :

Inlining is only a request to the compiler, not a command so it is up to compiler it can ignore the request for inlining.

→ If a function contain a loop for of more than 5 times.
→ If a function contain static variable.
→ If a function is recursive.
→ If a function return type is other than void and the return statement doesn't exist in function body.
→ If a function contain switch or goto statements.

* Function Overloading :

→ Overloading means assigning multiple meaning to a function name or operator symbol.
→ It allows multiple definition of a function with the same name, but different signature.
→ Requires each redefination of a function to use a different function signature that is :
  - different types of parameter
  - or sequence of parameters
  - or number of parameter.
→ function overloading allows function that conceptually perform the same task on objects of different types to be given the same name