MODULE 5- PART C

Types of Member Function

Types of Member Function

- Simple Function
- Static Function
- Const Function
- Inline Function
- Friend Function

Purpose of Making Data Member Static

- It is used to maintain Data which is specific for class not for object.
- Only one copy of that member is created for the entire class and is shared by all the objects of the class.
- Virtually eliminates need of GLOBAL Variable.
- Static data member behave exactly like global variable, difference is what its scope is within the class, global variable scope is within the program.
- Its lifetime is entire program , and is visible only within the class.

Static members

- Variant 1: Static data members
- Variant 2: Static function

What does static keyword actually mean?

- The static keyword in C++ makes any variable, object, or function a constant one. That means, only one copy of that *particular variable or function* would be **created** and will be **accessed** by all **the instances of the class.**
- Thus, it would help the programmers assist and manage the **memory** in a better manner because only a **single copy** of the static variables/functions resides **every time** an object calls for it.

Variant 1: Static data members

- If any data member or variable in C++ is declared as **static**, it acts like a **class variable**. Moreover, **only one copy** of the variable is available for the class objects to access.
- The memory for the **static variables** get allocated only once, and further all the function that calls the variable will available by a **single copy** of the variable. So, no instance copy of the static variable is created.

Syntax: Static variable in a member function

```
static data_type variable = value;
```

Syntax: Static variable within a Class

```
#Declaration of a static variable within a class
static data_type variable;
#Initializing a static variable within a class
data_type class_name::variable=value;
#Accessing the value of a static variable within a class
Class_name.variable_name;
```

Declaration

Static members are declared once inside the class as follows

```
static data_type variable_name;

// Example
static int count;
```

Definition(initialization)

Static members must be defined(initialized) outside the class using scope resolution as follows

```
data_type class_name::variable_name = value

// example
int Student:: count = 1
```

Example 1: Static variable within a member function

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

void count_static()
{

    static int stat_var = 1;
    cout << stat_var <<'\t';
    stat_var=stat_var*2;
}
void count_local()
{

     int loc_var = 1;
    cout << loc_var <<'\t';
    loc_var=loc_var*2;
}

int main()
{   cout<<"Manipulation on static variable:\n";
    for (int x=0; x<2; x++)
        count_static();
        cout<<'\n';
    cout<<"Manipulation on local variable:\n";
    for (int y=0; y<2; y++)
        count_local();
    return 0;
}
```

In the above example, we have done manipulations on a static and local variable to understand the difference between their functioning.

When we call the count_static() function, the value is once initialized and creates a single instance of the variable. When the main function encounters the count_static() function, the variable's value travels through the function.

On the other hand, when we call the count_local() function two times in a loop, the variable will be initialized every time. Thus, the value of local variable remains 1.

Output:

```
Manipulation on static variable:
1   2
Manipulation on local variable:
1   1
```

Example 2: Static variable in a class

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


class stat_var
{
    public:
    static int var;
    int x;
    stat_var()
    {
       x=0;
    }
};

int stat_var::var=10;

int main()
{
    stat_var V;
    cout << "Static Variable:"<<V.var;
    cout<<'\n';
    cout<<"Local Variable:"<<V.x;
}
```

we have used a static variable within a class. We need to initialize the static variable outside the class, else an exception will be raised.

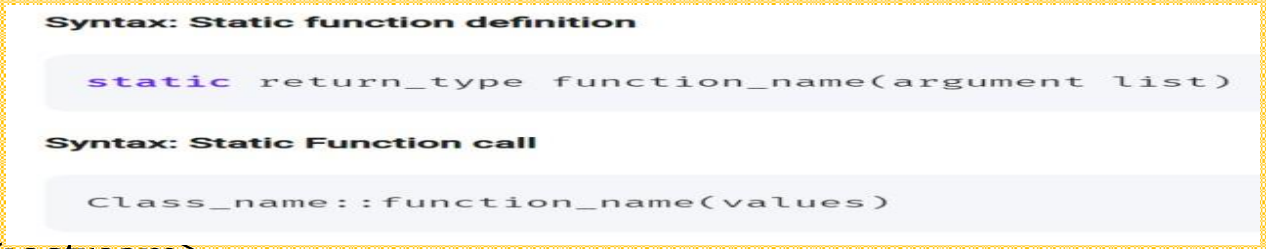
Further, we can access the value of the static variable using the class name. Thus, we no need to create an object for the same.

Output:

Static Variable:10
Local Variable:0

Variant 2: Static function in C++

- Static member **functions** work with **static data** values and thus cannot access any **non-static variables** of the class.




```
#include <iostream>
#include <string>
using namespace std;
class ABC
{
public:
static void stat_func(int a, int b)
{
   static int res = a+b;
   cout<<"Addition:"<<res;
   }
};
  int main()
  {
    ABC::stat_func(10,30);
```

output Addition:40

Note on Using Static

Basic Properties of Static data members

1. **Static data member is initialized with *zero automatically.***
2. **Static data members can be called directly using the class name**

Important points regarding static member function

- Static member functions cannot be overloaded
- Any static member function cannot be virtual
- Static functions do not operate with **this** pointer

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

class test
{
  public:
    // static int count = 1;
    // the above will give error
    // static member definition(initialization) must be outside the class

    // static member definiton only
    static int count;

    test(){
        // static members of a class can't be initialised
        // in a constructor as they are not associated
        // with each object of the class. It is shared by all objects.
    }
};
```

```cpp
// static data members definition must be outside the class
int test::count = 1;

int main()
{
    // count is 1 now after object creation
    test obj1, obj2;

    cout << "obj1.count :" << obj1.count << endl;
    cout << "obj2.count :" << obj2.count << endl;

    // obj1.count, obj2.count both become 2 as static member
    obj1.count = 2;

    cout << "\n\nobj1.count :" << obj1.count << endl;
    cout << "obj2.count :" << obj2.count << endl;

    return 0;
}/pre>
```

```
obj1.count :1
obj2.count :1


obj1.count :2
obj2.count :2

--------------------------------
Process exited after 0.02428 seconds with return value 0
Press any key to continue . . .
```

- **Change in obj1.count will also reflect on the change in obj2.count**
- **Since both share the same static count variable**
- **As ‘count’ is static, only a single copy is shared by all objects hence change in any object will have a change in all other objects.**

Static data member is initialized with zero automatically

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

class Test
{
    public:
    static int count; // static data member declaration
};

// not defining(initializing) but making accessible
int Test::count;

int main()
{
    Test t1;

    cout << 'Value of static member = "<< t1.count << endl;

    return 0;
}
```

Output

```
Value of static member = 0
```

Static data members can be called directly using the class name

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

class Test
{
    public:
    static int count; // static data member declaration
};

// not defining(initializing) but making accessible
int Test::count;
int main()
{
    //object not created like test t1, t2

    //static members can be accessed without created object
    //just use name_Of_Class::variable_name
    //if not initialised, static member's data value will be 0 by
    default
    cout << "Value of static member = "<< Test::count << endl;
}       return 0;
```

Output

```
Value of static member = 0
```

Static Member functions

Example Program

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

class sampleClass
{
  public:
    // static data members
    static int a,b;

    // non static data member
    int c;
```

```
    // static function definition
    // will only work with static data members
    static void add()
    {
        // static function can only access static data members
        cout << "Enter a, b values: ";
        cin >> a >> b;
        cout << "a + b = " << a + b;

        // following will give error -
        // error:cannot access non static data 'c' in static function
        // cin >> c;
    }
};

// both values will become 0 by default
int sampleClass::a;
int sampleClass::b;

int main()
{
  // calling static function with out object
  sampleClass::add();

  return 0;
}
```

Output

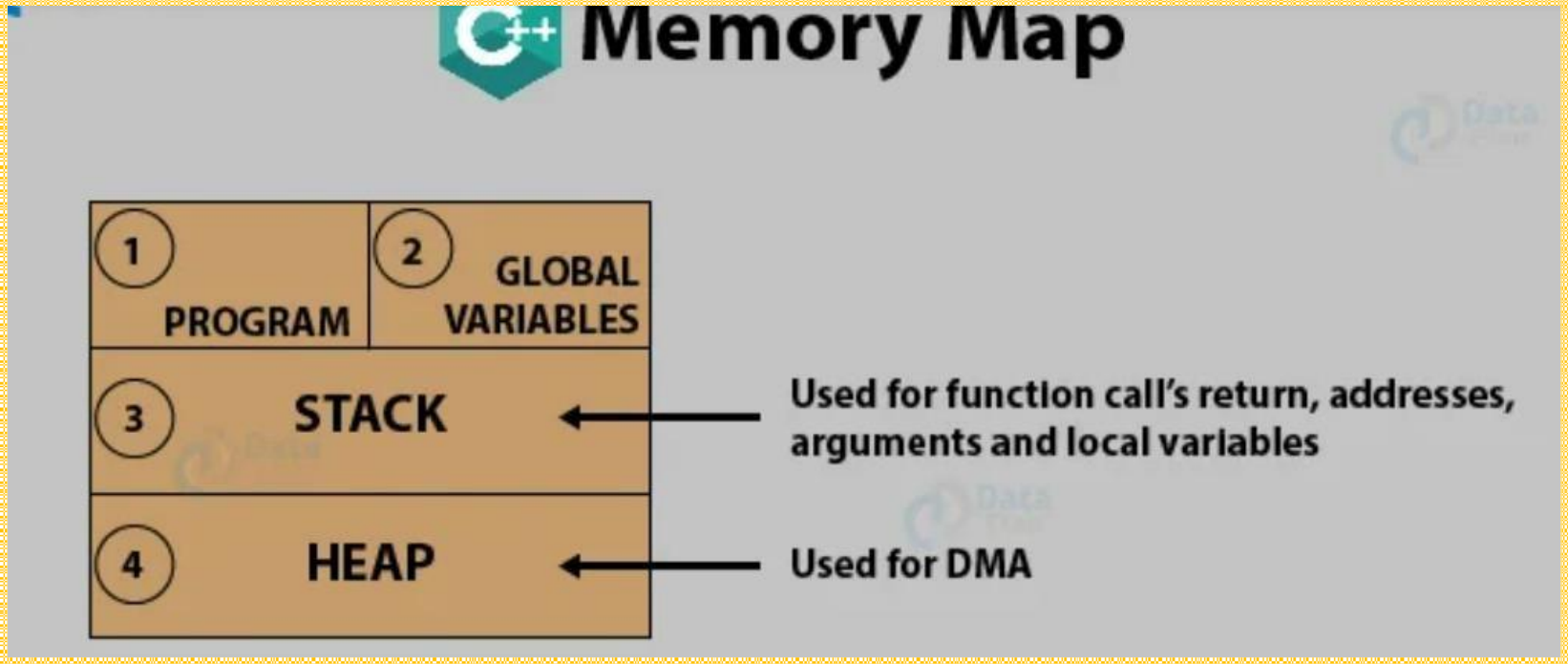
```
Enter a, b values: 10 20
a + b = 30
```

C++ Dynamic Allocation

Dynamic memory allocation

- In the Dynamic memory allocation, the memory is allocated to a variable or an array of programs at the run time
- C uses functions like malloc(), calloc(), realloc() and free() to handle operations based on DMA.
- C++ also uses all 4 functions in addition to **2 different** operators called **new and delete** to allocate memory dynamically.
- In C++, we need to deallocate the dynamically allocated memory **manually** after using a variable.
- We can allocate and then deallocate memory dynamically using the **new and delete** operators respectively.
- The only way to access this dynamically allocated memory is through a **pointer.**
- It is important to note that the memory is dynamically allocated on the **Heap**.
- The non-static memory and local variables get memory allocated on **Stack**.

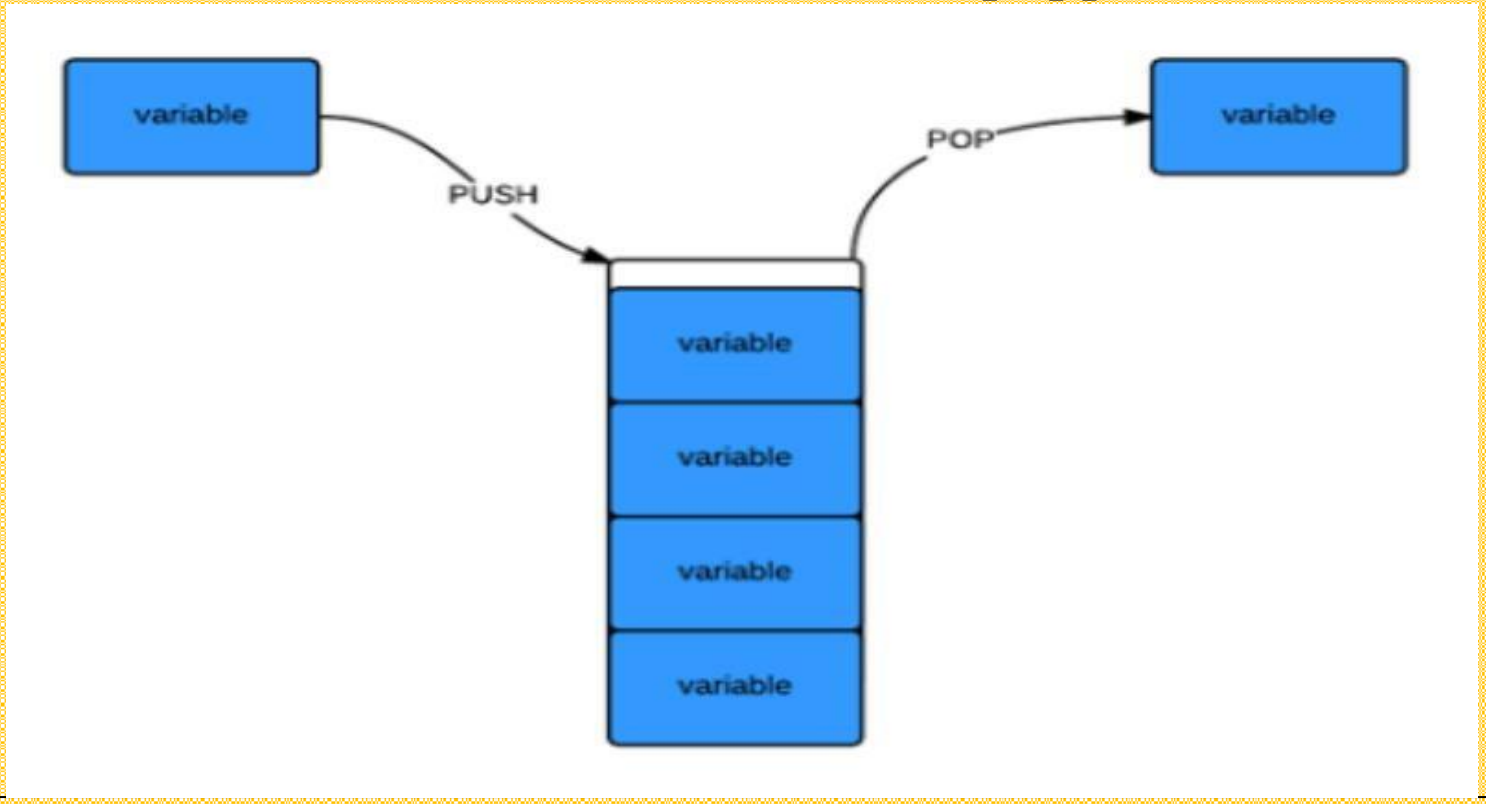
Dynamic memory allocation



Dynamic memory allocation

Stack

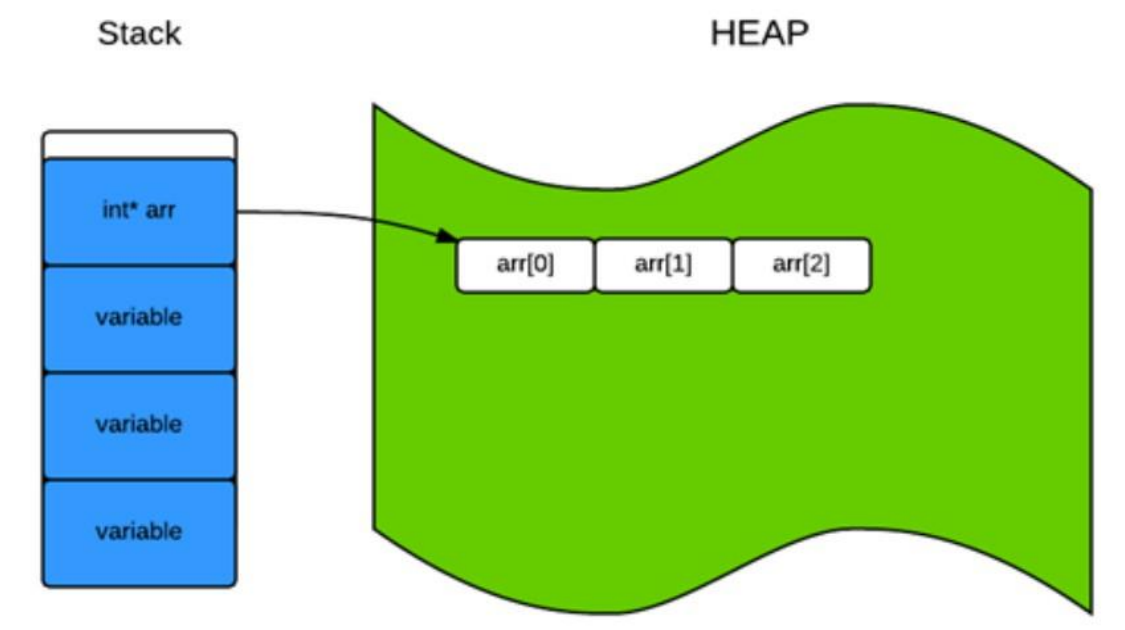
- A stack is the special region of the computer's memory, where **temporary variables** are stored.
- Stack follows the **First In Last Out** (FILO) data structure.
- When a function declares a variable, this variable is **pushed** into the stack.
- Once the function **exits,** the variable is popped from the stack**.**



Dynamic memory allocation

Heap

- A heap is a region of a computer's memory, used for **dynamic memory allocation**.
- When you use dynamic allocation, all the created variables are stored in a **heap**.
- Heap memory is not **managed automatically**.
- When you use dynamic memory allocation, **a pointer** that is located in the stack **points to the region** of the allocated memory in the **heap.**



New Operator in C++

- The **new operator** is used to allocate memory for a variable or any other entity like objects or arrays on a **heap memory area**.
- Operator **new** ***returns the pointer to the newly allocated space***.
- If a sufficient amount of memory is available on the heap, the new operator will initialize the memory and return the address of the newly allocated memory and you can use **pointers** to store the **address of that memory location**.

The syntax for the new operator:

<data_type pointer_name> = new <data_type>

Here in the above syntax,

- data_type can be any **inbuilt data type of C++** or any user data type data-type.
- pointer_name is a pointer variable of the type 'data_type'.

Eg: int* scaler = **new** int;

- In the given example, the **scaler** is a pointer of type **int** that points to a **new** memory block created at run time.

What if enough memory is not available in the heap:

- If sufficient memory is not available in the heap to allocate, it is indicated by **throwing an exception** of type **std::bad_alloc** and a pointer is returned unless the exception is handled with a new operator in which case it returns a NULL pointer.
- So it is good practice to check if the new operator is returning the NULL pointer and take appropriate action accordingly.

```
int* scaler = NULL;
//check whether after assigning a dynamic variable to scaler pointer, it is null or not
if (!(scaler = new int))
 {
cout << "Out of memory.";
 return;
}
```

Dynamic memory allocation

- Another way of using the **new** keyword is;

data_type *pointer_variable = new data_type;

For instance,

```
int *pointer = NULL;
pointer = new int;

or

int *pointer = new int;
```

- In case you would like to allocate memory **in arrays,** follow the order as follows:

Syntax:

new data_type[size_of_array];

Initialize memory

You can initialize the memory using new operator with the following syntax:

```
pointer_variable = new data_type(value);
```

Eg: int* scaler = new int(7);

In the above example scaler pointer is pointing to a dynamically allocated variable which is initialized by value 7.

Delete Operator in C++

- Since the programmer has allocated memory at runtime, it's the responsibility of the programmer to **delete that memory** when not required.
- So at any point, when programmers feel a variable that has been dynamically allocated is no anymore required, they can **free up** the memory that it occupies in the free store or heap with the "delete" operator.
- It returns the memory to the operating system. This is also known as memory deallocation.
- Also, memory will be deallocated automatically once the program ends.
- We use the "delete" operator in C++ for dynamic memory deallocation.
- Just like the "new" operator, the **"delete"** operator is also used by the programmer to manage computer memory
- **Key takeaway:** Both new and delete operators go hand in hand.

Syntax

```
delete pointer_variable;
```

For instance,

```
delete Scalar;
```

In the given example, the memory that is pointed by the variable 'scaler' will be

Note:

- If the program uses a large amount of unwanted memory using the “new” operator then the system may crash because there will be no memory available.
- In this case, the delete operator can help the system from crashing or from memory leap.
- It is good to use dynamic memory allocation for flexibility and other usages but if the programmer doesn’t deallocate memory, it can cause a memory leak (in which memory is not deallocated until the program terminates).
- So programmers should carefully explicitly delete the dynamically allocated memory for good coding practice and to avoid memory leaks.

C++ "new" & "delete" Operators for Arrays

- You can use **new and delete operators** for dynamic memory allocation and deallocation.
- Suppose you want to allocate memory for **an integer array**, i.e., an array of 7 integers.
- Using the **same syntax** that you have used above we can allocate memory dynamically as shown below
- int* scaler = new int[7]; // Request memory for the array

Here in the above example, the pointer 'scaler' is pointing to the first element of a dynamically created array of 7 integers.

- As the programmer has dynamically created the array so to **delete that array** programmer can use the delete operator as shown below:

```
delete [] scaler;          // Delete array pointed to by scaler
```

C++ "new" & "delete" Operators for Arrays

- **Similarly, you can allocate and deallocate memory for a multi-dimensional array**.

```
int** pointer_val = new int[7][7]; // Allocate memory for a 7*7 array
```

- In the above example, pointer_val is pointing to the memory that has been allocated dynamically to a 7*7 multi-dimensional array of the **type int.**
- The syntax to release the memory for a multi-dimensional array will remain the same as it is for **1-D arrays** :

```
delete [] pointer_val;  // Delete array pointed to by pointer_val
```

C++ "new" & "delete" Operators for Objects

- In C++, you can also dynamically allocate o**bjects**. It is just the same as a simple **data type.** Again use pointers while dynamically allocating **memory to objects**.
- In the below example, as you are allocating memory to an array of five **Scaler class objects,** the constructor of the Scaler class would be called **five times.** Similarly, while deleting these objects, the **destructor** will also be **called five times**.

C++ "new" & "delete" Operators for Objects

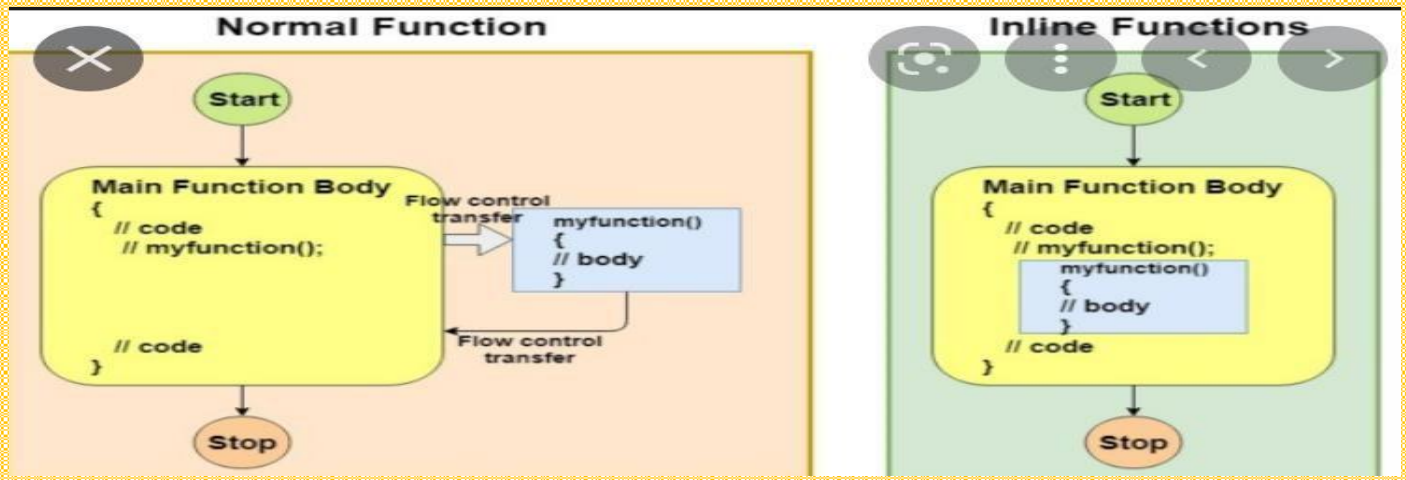
```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {

  // declare an int pointer
  int* scaler_int;

  // declare a float pointer
  float* scaler_float;

  // memory allocation of array with 7 elements
  int* scaler_array = new int[7];

  // dynamically allocate memory to both the pointers
  scaler_int = new int;
  scaler_float = new float;

  // assign value to the memory
  *scaler_int = 7;
  *scaler_float = 7.75 f;
  for (int i = 0; i < 7; i++) {
    *(scaler_array + i) = i;
  }

  //printing values stored at memory location
  cout << *scaler_int << endl;
  cout << *scaler_float << endl;
  for (int i = 0; i < 7; i++) {
    cout << *(ptr + i) << "";
  }

  // deallocating the memory
  delete scaler_int;
  delete scaler_float;
  delete[] scaler_array;

  return 0;
}
```

Output of the programme:

```
7
7.75
0 1 2 3 4 5 6
```

Inline Function

- **Inline function is introduced which is an optimization technique used by the compilers especially to reduce the execution time.**
- **" Inline functions are those whose *function body* is inserted in the place of the function call statement during the compilation process."**
- **Functions that are present inside a class are implicitly inline. Functions that are present outside class are considered normal functions.**



What is inline function :

- **The inline functions are a C++ enhancement feature to decrease the execution time of a program.**
- **Functions can be instructed to the compiler to make them inline so that the compiler can replace those function definitions wherever they are called.**
- **Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.**
- **if the function is big (in terms of executable instruction etc) then, the compiler can ignore the "inline" request and treat the function as normal function.**

Syntax of an inline function

```
inline returnType functionName(parameters)
{
// code
}
```

- **The return-type is the return-type of an inline function.**
- **The function name is the name of the inline function.**
- **The Parameters are the name and type of arguments allowed to be passed to this function.**

Difference between inline and Normal function

Inline Function

```
#include <iostream>
using namespace std;
inline int square(int s)
{
    return s*s;
}
void main()
{
    cout << square(5) << "\n";
}
```

Normal Function:

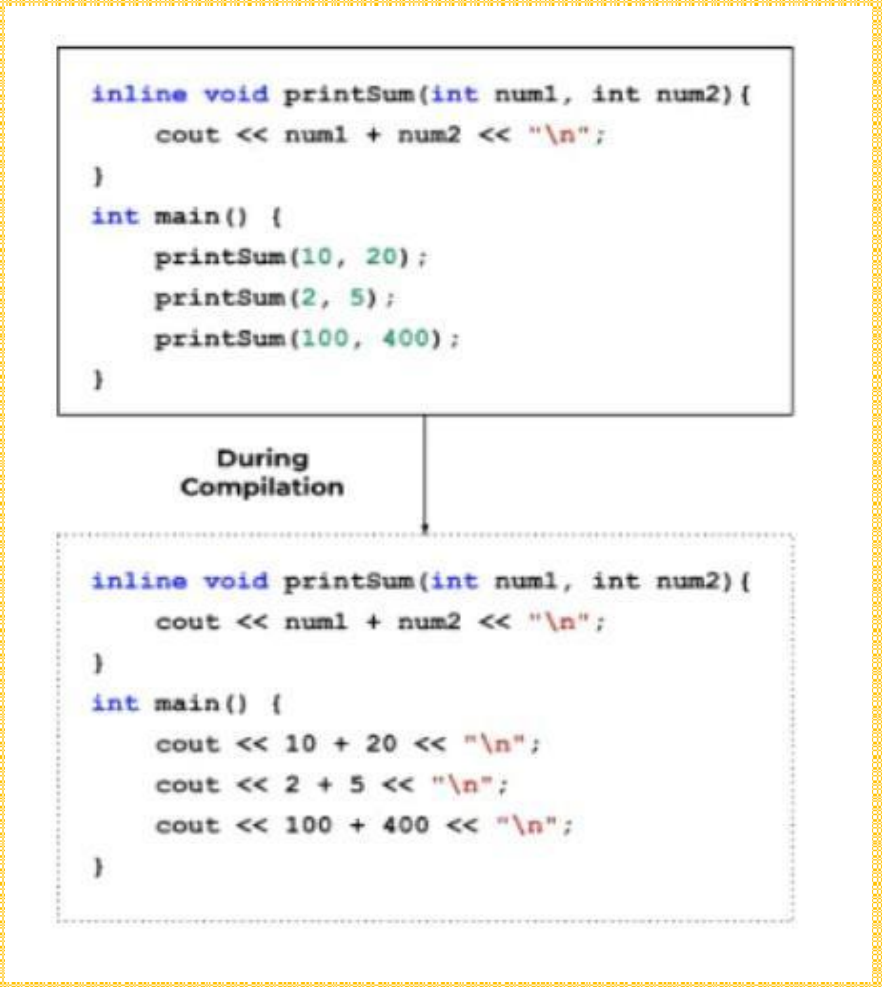
```
1.  #include <iostream>
2.  using namespace std;
3.  int square(int s)
4.  {
5.      return s*s;
6.  }
7.  void main()
8.  {
9.      cout << square(5) << "\n";
10. }
```

Example Inline with compilation

```
// define an inline function
// that prints the sum of 2 integers
inline void printSum(int num1,int num2) {
    cout << num1 + num2 << "\n";
}
int main() {
    // call the inline function
    // first call
    printSum(10, 20);
    // second call
    printSum(2, 5);
    // third call
    printSum(100, 400);
    return 0;
}
```

Example Inline with compilation

- The following diagram illustrates how the compiler works while executing the above program with an inline function:



Pros

1. It speeds up your program by avoiding function calling overhead.

2. It save overhead of variables push/pop on the stack, when function calling happens.

3. It save overhead of return call from a function.

4. It increases locality of reference by utilizing instruction cache.

5. By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)

When to use ?

The function can be made inline as per the programmer's need. Some useful recommendations are mentioned below-

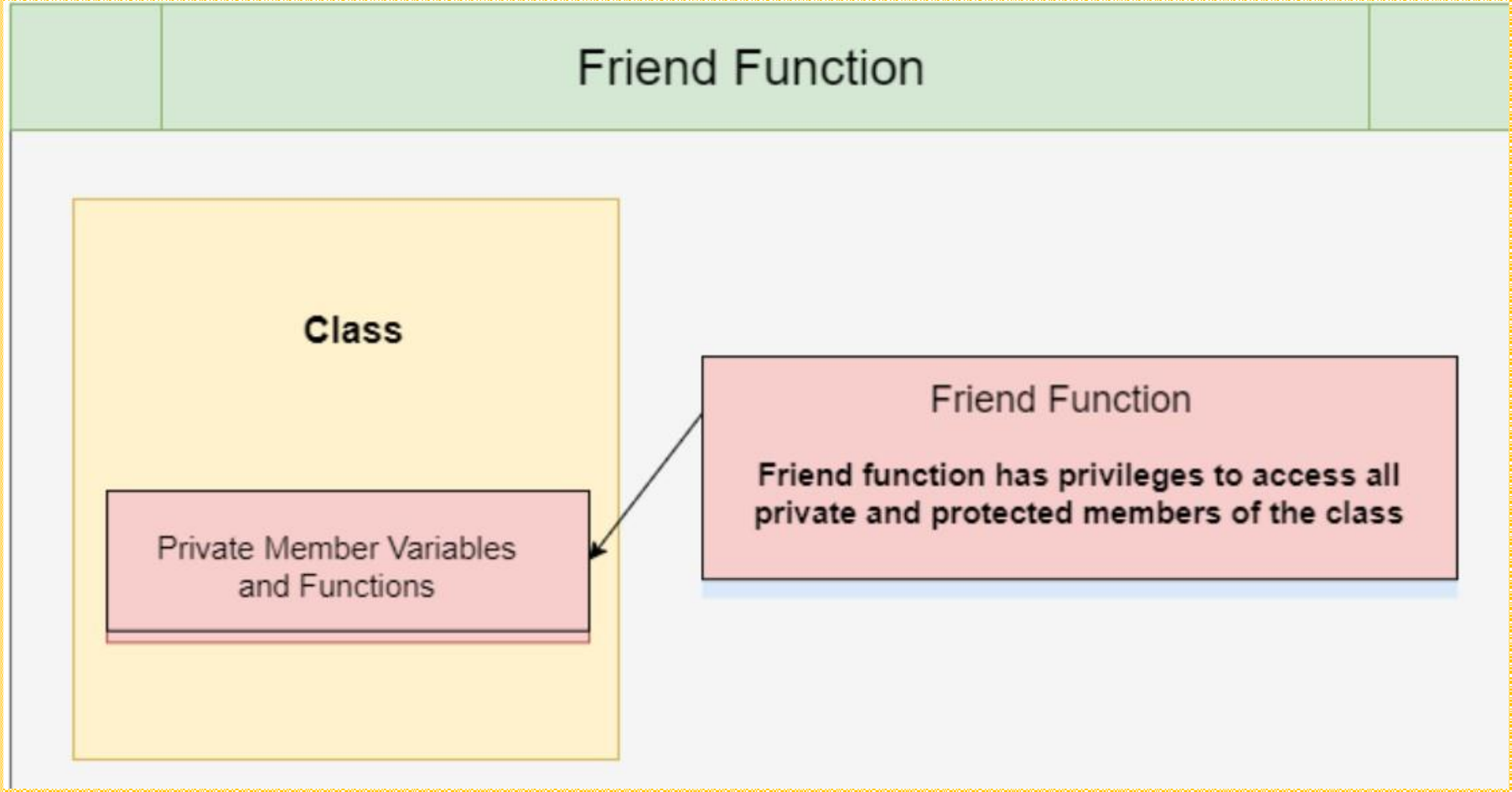
1. Use inline function when performance is needed.
2. Use inline function over macros.
3. Prefer to use inline keywords outside the class with the function definition to hide **implementation details**

Key Points

1. It's just a suggestion, not a compulsion. The compiler may or may not inline the functions you marked as inline. It may also decide to inline functions not marked as inline at compilation or linking time.
2. Inline works like a copy/paste controlled by the compiler, which is quite different from a preprocessor macro.
3. All the member functions declared and defined within the class are Inline by default. So no need to define explicitly.
4. Virtual methods are not supposed to be inlinable. Still, sometimes, when the compiler can know for sure the type of the object (i.e. the object was declared and constructed inside the same function body), even a virtual function will be inlined because the compiler knows exactly the type of the object.

Friend Function

Introduction



What is Friend Function?

- A non member function
- Has access to all the Private and Protected Members of a class

How is it different from normal function?

- A member is access "through" the object

Ex : sample object;
object.getdata();

- Whereas A Friend function requires object to be passed by value or by reference as a parameter

Ex : void getdata(sample object);

Why using Friend?

- A Friend function is a non-member function of the class that has been granted access to all private members of the class.
- We simply declare the function within the class by a prefixing its declaration with keyword friend.
- Function definition must not use keyword friend.
- Definition of friend function is specified outside the class body and is not treated as a part of the class.
- The major difference b/w member function and friend function is that the member function is accessed through the object while friend function requires object to be passed as parameter.

Syntax

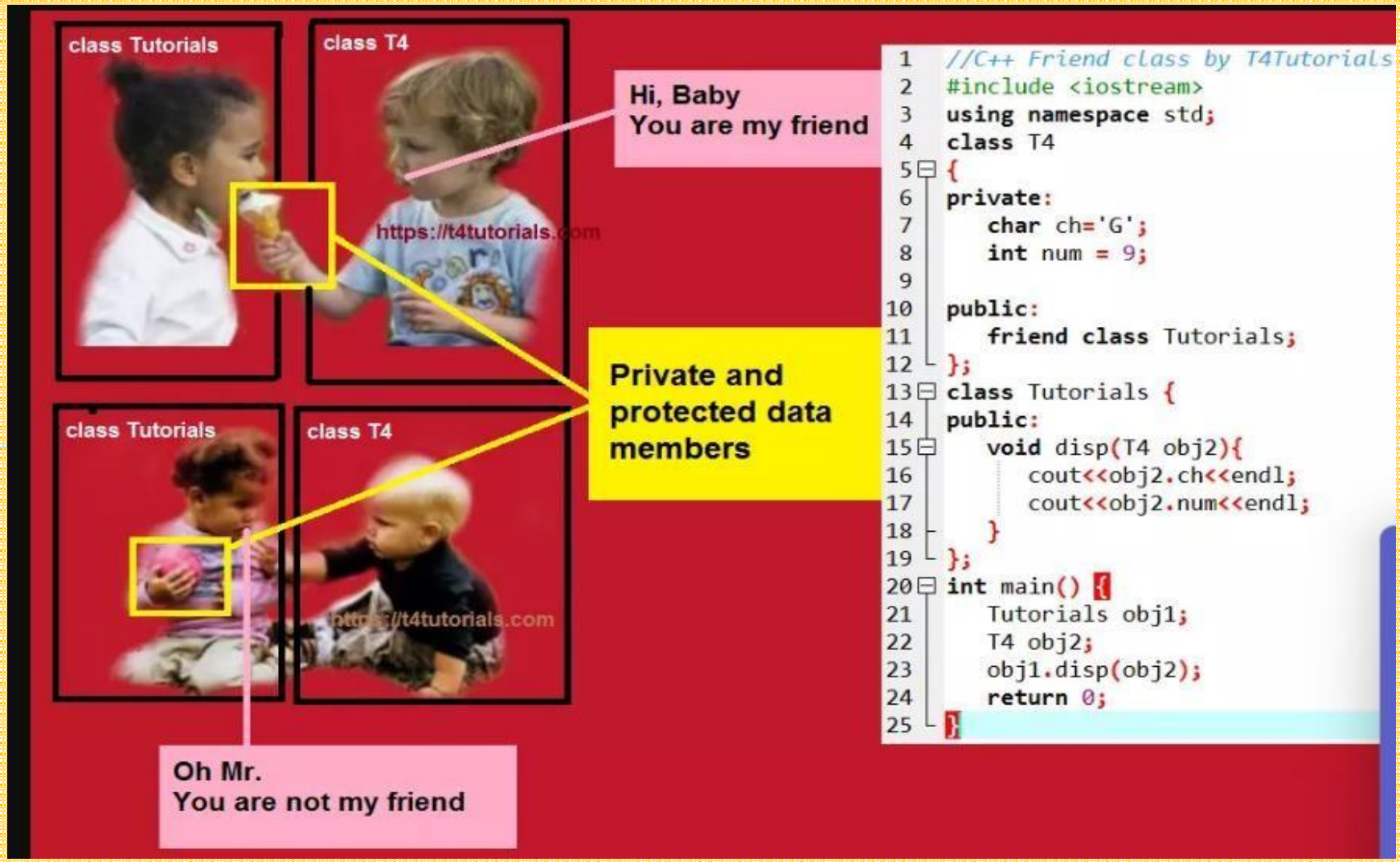
```
class ABC
{
.............
public:
friend void xyz(object  of class);
};
```

- Friend function characteristics
- It is not in the scope of the class.
- It cannot be called using the object of that class.
- It can be invoked like a normal function.
- It should use a dot operator for accessing members.
- It can be public or private.
- It has objects as arguments.
- The most common use of friend functions is overloading << and >> for I/O.

What are its Disadvantages?

What are its Disadvantages?

- Violates Data Hiding
- Violates encapsulation


class Tutorials
class T4
https://t4tutorials.com
Hi, Baby
You are my friend
Private and
protected data
members
class Tutorials
class T4
https://t4tutorials.com
Oh Mr.
You are not my friend
1 //C++ Friend class by T4Tutorials
2 #include <iostream>
3 using namespace std;
4 class T4
5 {
6 private:
7 char ch='G';
8 int num = 9;
9
10 public:
11 friend class Tutorials;
12 };
13 class Tutorials {
14 public:
15 void disp(T4 obj2){
16 cout<<obj2.ch<<endl;
17 cout<<obj2.num<<endl;
18 }
19 };
20 int main() {
21 Tutorials obj1;
22 T4 obj2;
23 obj1.disp(obj2);
24 return 0;
25 }

Friend class

- In the previous section of the class, we declared **only one function** as a friend of another class. But it is possible that all members of the one class can be friends of another class.
- Friendship is not inherited, transitive, or reciprocal.
- Derived classes don't receive the privileges of friendship (more on this when we get to inheritance in a few classes).
- The privileges of friendship aren't transitive. If class A declares class B as a friend, and class B declares class C as a friend, class C doesn't necessarily have any special access rights to class A.
- If class A declares class B as a friend (so class B can see class A's private members), class A is not automatically a friend of class B (so class A cannot necessarily see the private data members of class B).

This Operator

Introduction to This Pointer

- C++ this pointer actually points at the **current object** of the class i.e. it stores and represents the address of the **current object** of the **particular class.**
- Moreover, this pointer also enables us to refer to the instance variable of the current class being called.
- Every object in C++ has access to its own address through an important pointer called this pointer.
- The this pointer is an implicit parameter to all **member functions.** Therefore, inside a member function, this may be used to refer to the invoking object. this pointer can also be used to **return its own reference.**

- Friend functions do not have this pointer, because friends are not members of a class.
- Only member functions have this' pointer.
- this' pointer is not available in static member functions as static member functions can be called without any object (with class name).
- Following are the situations where 'this' pointer is used:
  - When the local variable's name is the same as the member's name.
  - To return a reference to the calling object.

What is this pointer

- **What is *this* pointer?**
- Every object has a special pointer "this" which points to the object itself.
- This pointer is accessible to all members of the class but not to any static members of the class.
- Can be used to find the address of the object in which the function is a member.
- Presence of this pointer is not included in the sizeof calculations.

Syntax This Pointer

Into C++ programming language, '**this**' is a keyword that refers to the current instance of the class.

There can be 3 main usages of this keyword in C++.

- This can be used to pass the current object as a parameter to another method.
- This can be used to refer to the current class instance variable.
- This can be used to declare indexers.

Syntax This Pointer

```
this->variable=value; //syntax of referring to the instance variable of the
class'

this* // syntax for referring to the current object of the class/function
```

served by this(->) pointer:

- Point to the current object in the class/function.
- Refer to the instance variables of the current class in process/use.

Example of this pointer

- In the below example, we have created a class 'Point' wherein we have defined a private data member as 'Val'.
- Further, we have created two functions – 'show', and 'set' which consist of an instance variable 'val'.

Syntax This Pointer

```
#include <iostream>
using namespace std;
class Point {
private:
  int val;

public:
  void set(int val){
    this->val = val;
  }
  void show(){
    cout<<"Value of variable: "<<val<<endl;
  }
};
int main(){
  Point ob;
  ob.set(500);
  ob.show();
  return 0;
}
```

Output:

```
Value of variable: 500
```

- So, as we can clearly understand, the above program contains a data member and a local variable having the same name – **'Val'.**
- In this case, this pointer enables us to assign a value to the class's **instance variable** through the local data member.

When the local variable's name is the same as the member's name.

```
#include<iostream>
using namespace std;
/* local variable is the same as a member's name */
class Test
{
private:
int x;
public:
void setX (int x)
 {
// The 'this' pointer is used to retrieve the object's x
 // hidden by the local variable 'x'
this->x = x;
}
void print() { cout << "x = " << x << endl; }
};
int main()
{
Test obj;
int x = 20;
obj.setX(x);
obj.print();
 return 0;
```

```
Output

x = 20
```

To return a reference to the calling object.

```
#include<iostream>
using namespace std;
class Max
{
    int a;
    public:
        void getdata()
        {
            cout<<"Enter the Value :";
            cin>>a;
        }
        Max &greater(Max &x)
        {
            if(x.a>=a)
                return x;
            else return *this;
        }
        void display()
        {
            cout<<"Maximum No. is : "<<a<<endl;
        }
};
int main()
{
    Max one,two,three;
    one.getdata();
    two.getdata();
    three=one.greater(two);
    three.display();

    return 0;
}
```

Output

```
Enter the Value :5
Enter the Value :8
Maximum No. is : 8
```