C++ Storage Classes

- We have seen that every variable has a data type.
- To fully define a variable, we need a storage class as well apart from data type.
- But we have not introduced the storage class so far.
- When we do not define a storage class, compiler automatically assigns a default storage class to it.
- A variable's storage class gives information about
    - the storage location of variable in memory,
    - default initial value,
    - scope of the variable and it's lifetime.

- Storage class specifiers supported in C++ are **auto, register, static, extern** and **mutable**.

1. ' auto ' storage class

It applies to local variables only and the variable is visible only inside the function in which it is declared and it dies as soon as the function execution is over. If not initialized, variables of class auto contains garbage value.

Following example shows how variables of auto storage class are declared :

```
int var;      // by default, storage class is auto
auto int var;
```

2. ' register ' storage class

Variables of class 'register' are stored in CPU registers instead of memory which allows faster access. The scope of the variable is local to the function in which it is defined and it dies as soon as the function execution is over. It contains some garbage value if not initialized. It is declared as :

```
register int var;
```

3. ' static ' storage class

The scope of static variable is local to the function in which it is defined but it doesn't die when the function execution is over. The value of a static variable persists between function calls. The default initial value of static variable is 0. Following program illustrates the behavior of static variables :

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

void func() {
  static int count = 0;     /* value of 'count' persist between function calls. count doesn't become 0
                               every time func() is called; previous value of count remains alive
  count++;         cout << "Function is called " << count << " times " << endl;
}
```

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

Output :

```
Function is called 1 times
Function is called 2 times
Function is called 3 times
```

4. extern ' storage class

- Variables of extern storage class have a global scope.
- It is useful when we share a variable between a few modules. We define it in one module, and declare using extern in the others. It is visible outside the file in which it is declared. So, an extern variable can be shared across multiple files.
- An extern variable remains alive as long as program execution continues.
  - A static global variable is visible only in the file it is declared but
  - An extern global variable is visible across all the files of a program.

- We define the *existence* of global variables in a source file so that other source file that includes that file knows about it, but we only need to "declare" it once in one of our source files.

In simple words :

```
extern int count; // declaration of variable 'count'
int count;        // definition of variable 'count'
```

- Using extern int x; tells the compiler that an object of type int called x exists *somewhere*. It's not the compilers job to know where it exists, it just needs to know the type and name so it knows how to use it.
- Once all of the source files have been compiled, the linker will resolve all of the references of x to the one definition that it finds in one of the compiled source files.
- For it to work, the definition of the x variable needs to have what's called "external linkage", which basically means that it needs to be declared outside of a function (at what's usually called "the file scope") and without the static keyword.

```
/* File : extern_var_util.cpp   */

int count = 7;  // definition of variable
                   count

void incrementCount() {
  ++count;
}
```

The variable count is declared in one file and it is defined in other file. It can be accessed and modified in any of the files and the updated value of count is reflected in both the files.

```
/* File : extern_var.cpp    */

#include<iostream>
#include "extern_var_util.cpp" // including the contents of a
                                  user defined file
using namespace std;

extern int count; // declaration of variable 'count'

int main() {
  cout << "count : " << count << endl;
  count = 1;
  incrementCount();
  cout << "count : " << count << endl;
  return 0;
}
```

```
Output :

Count : 7
Count : 2
```

5. mutable ' storage class

mutable storage class is applicable to only class data members. We know that a constant variable cannot be modified in the program. This is applicable to objects also. If a data member of a class is declared as mutable, then it can be modified by an object which is declared as constant. See the program below to get more clarity :

```
#include<iostream>
using namespace std;
class X {
   public :
         X (int a, int b) { // constructor
            m = a;      n = b;
         }
  // data members
  int m;
  mutable int n;
};
```

```
int main() {
  // declare a const variable 'obj' of type 'X'
  const X obj(5, 2);     // m = 5 and n = 2
  cout << "m : " << obj.m << "  n : " << obj.n << endl;

  // obj.m = 7;      // illegal since 'obj' is constant
  obj.n = 8;          // legal since 'n' is mutable
  cout << "m : " << obj.m << "  n : " << obj.n << endl;
  return 0;  }
```

We will discuss only the static keyword in more detail.

Usages of static keyword in C++

- static keyword can be applied to local variables, functions, class' data members and objects in C++.
- static local variables (declared static within the body of a function) retain their values between function call and initialized only once.
- A variable declared static within a module (but outside the body of a function) is accessible by all functions within that module. However, it is not accessible by functions from other modules.

- static members exist as members of the class rather than as an instance in each object of the class.

    - There is only a single instance of each static data member for the entire class.
    - Static member function can only access static member data, or other static member functions although Non-static member functions can access all data members of the class: static and non-static.

Static objects in C++

An object become static when static keyword is used in its declaration.

```
Test t;              // Stack based object also called automatic objects or local objects
                        Local object is created each time its declaration is encountered in the
                        execution of program.
static Test t1;    // Static object initialized only once and live until the program terminates
```

File scope: it can be seen only within a file where it's defined.

Visibility: if it is defined within a function/block, it's scope is limited to the function/block. It cannot be accessed outside of the function/block.

In Java there is no equivalent syntax. A static object in Java is considered as a class all of whose members (functions and data) are declared static.

Some important rules of static Class Member

1. Only single instance of each static data member for the entire class exist.
2. **this** keyword is not available in a static member function.
3. **Static member functions** can only access static member data, or other static member functions
4. It should be initialized, usually in the **source file** that implements the class member functions. Because the member is initialized outside the class definition, we must use fully qualified name when we initialize it: class_name::static_member_name = value;

When we declare a static member variable inside a class, we're simply telling the class that a static member variable exists. Because static member variables are not part of the individual class objects (they are treated similarly to global variables, and get initialized when the program starts), you must explicitly define the static member outside of the class, in the global scope.

If the class is defined in a .h file, the static member definition is usually placed in the associated code file for the class (e.g. Something.cpp). If the class is defined in a .cpp file, the static member definition is usually placed directly underneath the class.

Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you'll end up with multiple definitions, which will cause a compile error).

In the example below, we do so via this line:

Car.h

```
class Car
{
    private:
        static int id;
    public:
        Car();
    ...
};
```

Declaring the variable inside class but it doesn't allocate memory.

Implementation file, Car.cpp:

```
#include <iostream>
int Car::id = 100;
...
```

The declaration line serves two purposes:

- it instantiates the static member variable (just like a global variable), and
- optionally initializes it. In this case, we're providing the initialization value 100. If no initializer is provided, C++ initializes the value to 0.

Note that this static member definition is not **subject to access controls**: you can define and initialize the value even if it's declared as private (or protected) in the class.

Initialize a static member variable inside the class declaration. It allocates memory.

```
Car::Car() {}
....
```

If the static member variables are **public**, we can access them directly using the class name and the scope resolution operator. But for **private** static variable we need to define a public static method to access it.

4. The **exception for the initialization** : Initialize inside class if the static data member is a const of integral or enumeration type.

```
#include <iostream>
class Car
{
        enum Color {silver = 0, maroon, red };
        int year;
        int mileage = 34289;   // error: not-static data members
                               // only static const integral data members  can be initialized within a class

        static int vin = 12345678;       // error: non-constant data member
                               // only static const integral data members can be initialized within a class

        static const string model = "Sonata";  // error: not-integral type
                                               // cannot have in-class initializer

        static const int engine = 6;         // allowed: static const integral type
};
int Car::year = 2013;    // error: non-static data members cannot be defined out-of-class

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

The following example shows an illegal access to non-static member from a static function.

```
class X
{
public:
        int x;
        static void fstat(int);
};
void X::fstat(int z) {x=z;}
```

In function fstat(), x=z is an error because it is a static function and trying to access non-static member x.

To fix it we should declare x as static like: static int x;

3. Inline initialization of static member variables

There are a few shortcuts to the above. First, when the static member is a **const** integer (which includes char and bool) or a const enum, the static member can be initialized inside the class definition:

```
class Whatever
{
   public:
      static const int s_value = 4; // a static const int can be declared and initialized directly
};
```

Note : constexpr specifier as of **C++11 –** By using static constexpr members of any type can be initialized inside the class definition:

```
#include <array>
class Whatever
{
   public:
        static constexpr double s_value = 2.2; // ok
        static constexpr std::array<int, 3> s_array = { 1, 2, 3 }; // this even works for classes
};
```

4. Non-static members cannot be used as **default arguments.** The following code shows that a non-static members cannot be used as default arguments.

```
#include <iostream>
int xGlobal = 7;

struct Foo
{
    int xMember;          static int xStatic;
    Foo(int x) : xMember(x)  {}

    int a(int x = xGlobal)   {    return x;    }
    // wrong: won't compile as xMember is not static
    int b(int x = xMember)   {    return x;   }
    int c(int x = xStatic)   {    return x;   }
};
int Foo::xStatic = 1;
int main()
{
   Foo f(911);          std::cout << f.a() << std::endl;    std::cout << f.b() << std::endl;
   std::cout << f.c() << std::endl;
   return 0;
}
```

Static Member Functions

- We can define class members static using static keyword.
- No matter how many objects of the class are created, there is only one copy of the static member. A static member is shared by all objects of the class.
- All static data is initialized to zero when the **first object is created**, if no other initialization is

present.

- We can't put it in the class definition but it can be initialized outside the class using the scope resolution operator :: to identify which class it belongs to.

- A static member function can only access :
    1. static member data,
    2. static member functions,
    3. data and functions outside the class.

```
class Something
{
    private:
        static int s_value;
    public:
        static int getValue() {
            return s_value;
        } // static member function
};
int Something::s_value = 1;

int main()  {
    std::cout << Something::getValue() << '\n';
}
```

In this case, we can't access Something::s_value directly from main(), because it is **private**. So Normally we access private members through public member functions.

- If we create a normal public member function to access s_value, we'd then need to instantiate an object of the class type to use the function!
- If we define the static function we can access it with the help of class. Like static member variables, static member functions are not attached to any particular object.

Note : Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator. Like static member variables, they can also be called through objects of the class type, though this is not recommended.

- A non-static member function can be called only after instantiating the class as an object. A static member function can be called, even when a class is not instantiated.

- A static member function cannot have access to the **this** pointer of the class. This makes sense -- the *this* pointer always points to the object that the member function is working on. Static member functions do not work on an object, so the *this* pointer is not needed.

- Static member functions can only access static member variables. They cannot access non-static member variables. This is because non-static member variables must belong to a class object, and static member functions have no class object to work with!

- A non-static member function can be declared as virtual but care must be taken not to declare a static member function as virtual.

When are static objects destroyed?

An object become static when static keyword is used in its declaration. See the following two statements for example in C++.

```
Test t;          // Stack based object memory allocated to stack
static Test t1;  // Static object
```

Stack based objects are also called automatic objects or local objects.

- static object are initialized only once and live until the program terminates.
- static objects are allocated storage in static storage area.
- static object is destroyed at the termination of program.
- C++ supports both local static object and global static object.

Following is example that shows use of local static object.

```
#include <iostream>
class Test
{
    public:
        Test()    {    std::cout << "Constructor is executed\n";   }
        ~Test()  {    std::cout << "Destructor is executed\n";    }
};
void myfunc()
{
   static Test obj;
} // obj is still not destroyed because it is static

int main()
{
   std::cout << "main() starts\n";
   myfunc();   // Destructor will not be called here
   std::cout << "main() terminates\n";
   return 0;
}
```

Output:
main() starts
Constructor is executed
main() terminates
Destructor is executed

From output we can see that the destructor for the local object named obj is not called after it's constructor is executed because the local object is static so it has scope till the lifetime of program so it's destructor will be called when main() terminates.

What happens when we remove static in above program?

We get the output as below:
main() starts
Constructor is called
Destructor is called
main() terminates

This is because the object is now stack based object and it is destroyed when it is goes out of scope and its destructor will be called.

How about global static objects?

The following program demonstrates use of global static object.

```
#include <iostream>
class Test
{
    public:
        int a;
        Test()    {      a = 10;      std::cout << "Constructor is executed\n";    }
        ~Test()   {      std::cout << "Destructor is executed\n";    }
};
static Test obj;   //  Global static object
int main()  {
   std::cout << "main() starts\n";
   std::cout << obj.a;
   std::cout << "\n main() terminates\n";
   return 0;     }
```

Output:
Constructor is executed
main() starts
10
main() terminates
Destructor is executed

C++ doesn't have any facility for initializing static class members like that of Java using static block

Keyword const

When passing objects by using the keyword const, as a parameter to a function that possibly calls some of the object's member functions, one has to tell C++ that these methods will not change the passed object. To do this one has to insert

The keyword const in the methods' declarations.
Here is a short example.

It is always advisable to use the keyword const for member functions that will not change the object's data. For instance, it makes debugging easier and it helps to avoid some errors already at the compilation stage.

```
class myclass {
  int a;
  public:
     void showvalues(const myclass &A) { this->print();    A.print(); }
     void print() const { cout << a << endl; } // const needed!
};
```

C++ final specifier

In Java, we can use final for a function to make sure that it cannot be overridden. We can also use final in Java to make sure that a class cannot be inherited. Similarly, the latest C++ standard C++ 11 added final.

final in C++ 11 vs in Java

Note that use of final specifier in C++ 11 is same as in Java but Java uses final before the class name while final specifier is used after the class name in C++ 11. Same way Java uses final keyword in the beginning of method definition (Before the return type of method) but C++ 11 uses final specifier after the function name.

```
class Test
{
   final void fun()// use of final in Java
   { }
}
```

```
class Test
{
   public:
        virtual void fun() final     //use of final in C++ 11
        {}
};
```

Unlike Java, final is not a keyword in C++ 11. final has meaning only when used in above contexts, otherwise it's just an identifier.

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

int main()
{
   int final = 20;
   cout << final;
   return 0;
}
```

Output:
20

In java, final can also be used with variables to make sure that a value can only be assigned once. This use of final is not there in C++ 11.