# Storage Classes in C++ with Examples

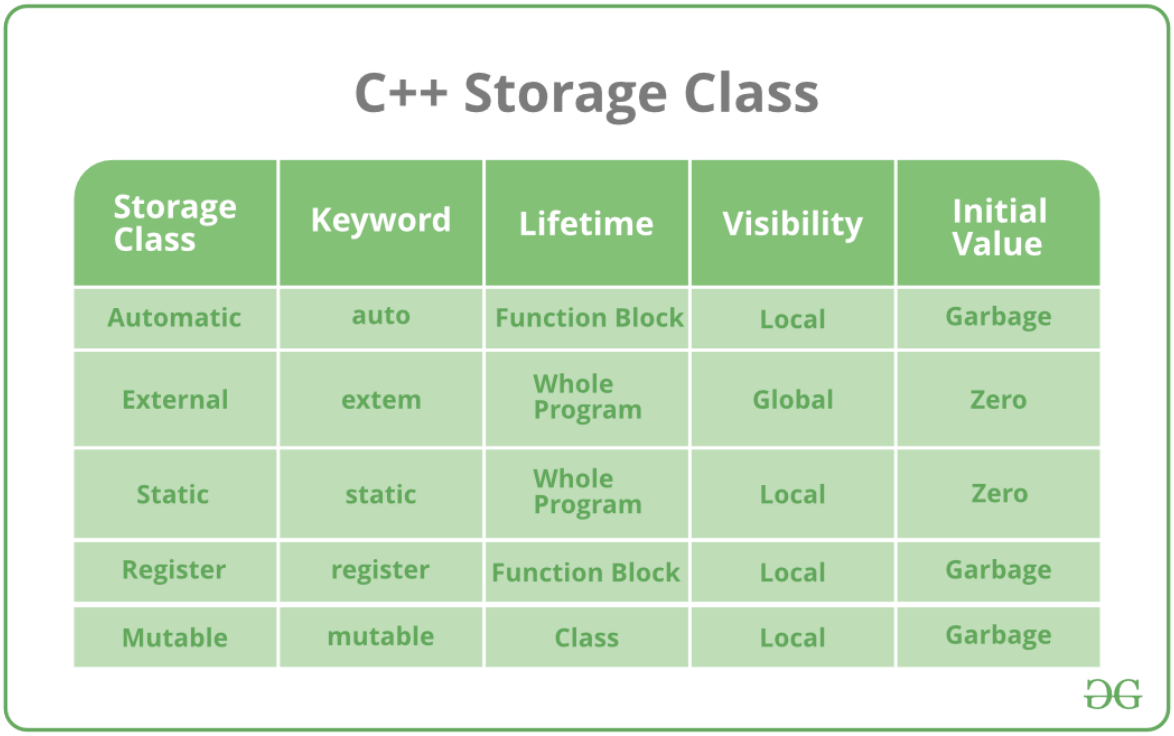
**Storage Classes** are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program. To specify the storage class for a variable, the following syntax is to be followed:

**Syntax:**

```
storage_class var_data_type var_name;
```

C++ uses 5 storage classes, namely:

1. auto
2. register
3. extern
4. static
5. mutable




Below is the detailed explanation of each storage class:

**auto**: The auto keyword provides type inference capabilities, using which automatic deduction of the data type of an expression in a programming language can be done. This consumes less time having to write out things the compiler already knows. As all the types are deduced in compiler phase only, the time for compilation increases slightly but it does not affect the run time of the program. This feature also extends to functions and non-type template parameters. Since C++14 for functions, the return type will be deduced from its return statements. Since C++17, for non-type template parameters, the type will be deduced from the argument.

**Example:**

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

void autoStorageClass()
{

    cout << "Demonstrating auto class\n";

    // Declaring an auto variable
    // No data-type declaration needed
    auto a = 32;
    auto b = 3.2;
    auto c = "GeeksforGeeks";
    auto d = 'G';

    // printing the auto variables
    cout << a << " \n";
    cout << b << " \n";
    cout << c << " \n";
    cout << d << " \n";
}

int main()
{

    // To demonstrate auto Storage Class
    autoStorageClass();

    return 0;
}
```

**Output:**

```
Demonstrating auto class

32

3.2

GeeksforGeeks

G
```

**extern**: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this link.

**Example:**

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

// declaring the variable which is to
// be made extern an initial value can
// also be initialized to x
int x;
void externStorageClass()
{
    cout << "Demonstrating extern class\n";
    // telling the compiler that the variable
    // x is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;
    // printing the extern variables 'x'
    cout << "Value of the variable 'x'"
         << "declared, as extern: " << x << "\n";
    // value of extern variable x modified
    x = 2;
    // printing the modified values of
    // extern variables 'x'
    cout
        << "Modified value of the variable 'x'"
        << " declared as extern: \n"
        << x;
}
int main()
{
    // To demonstrate extern Storage Class
    externStorageClass();
    return 0;
}
```

**Output:**

```
Demonstrating extern class

Value of the variable 'x'declared, as extern: 0

Modified value of the variable 'x' declared as extern:

2
```

**static**: This storage class is used to declare static variables which are popularly used while writing programs in C++ language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist until the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

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

// Function containing static variables
// memory is retained during execution
int staticFun()
{
    cout << "For static variables: ";
    static int count = 0;
    count++;
    return count;
}
// Function containing non-static variables
// memory is destroyed
int nonStaticFun()
{
    cout << "For Non-Static variables: ";
    int count = 0;
    count++;
    return count;
}

int main()
{
    // Calling the static parts
    cout << staticFun() << "\n";
    cout << staticFun() << "\n";
    ;
    // Calling the non-static parts
    cout << nonStaticFun() << "\n";
    ;
    cout << nonStaticFun() << "\n";
    ;
    return 0;
}
```

**Output:**

```
For static variables: 1

For static variables: 2

For Non-Static variables: 1

For Non-Static variables: 1
```

**register**: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

**Example:**

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

void registerStorageClass()
{

    cout << "Demonstrating register class\n";

    // declaring a register variable
    register char b = 'G';

    // printing the register variable 'b'
    cout << "Value of the variable 'b'"
         << " declared as register: " << b;
}
int main()
{

    // To demonstrate register Storage Class
    registerStorageClass();
    return 0;
}
```

**Output:**

```
Demonstrating register class

Value of the variable 'b' declared as register: G
```

**mutable:** Sometimes there is a requirement to modify one or more data members of class/struct through const function even though you don't want the function to update other members of class/struct. This task can be easily performed by using the mutable keyword. The keyword mutable is mainly used to allow a particular data member of const object to be modified. When we declare a function as const, this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.

**Example:**

```
#include <iostream>
using std::cout;

class Test {
public:
    int x;

    // defining mutable variable y
    // now this can be modified
    mutable int y;

    Test()
    {
        x = 4;
        y = 10;
    }
};

int main()
{
    // t1 is set to constant
    const Test t1;

    // trying to change the value
    t1.y = 20;
    cout << t1.y;

    // Uncommenting below lines
    // will throw error
    // t1.x = 8;
    // cout << t1.x;
    return 0;
}
```

**Output:**

```
20
```