Lesson 7 - Storage Classes in C++

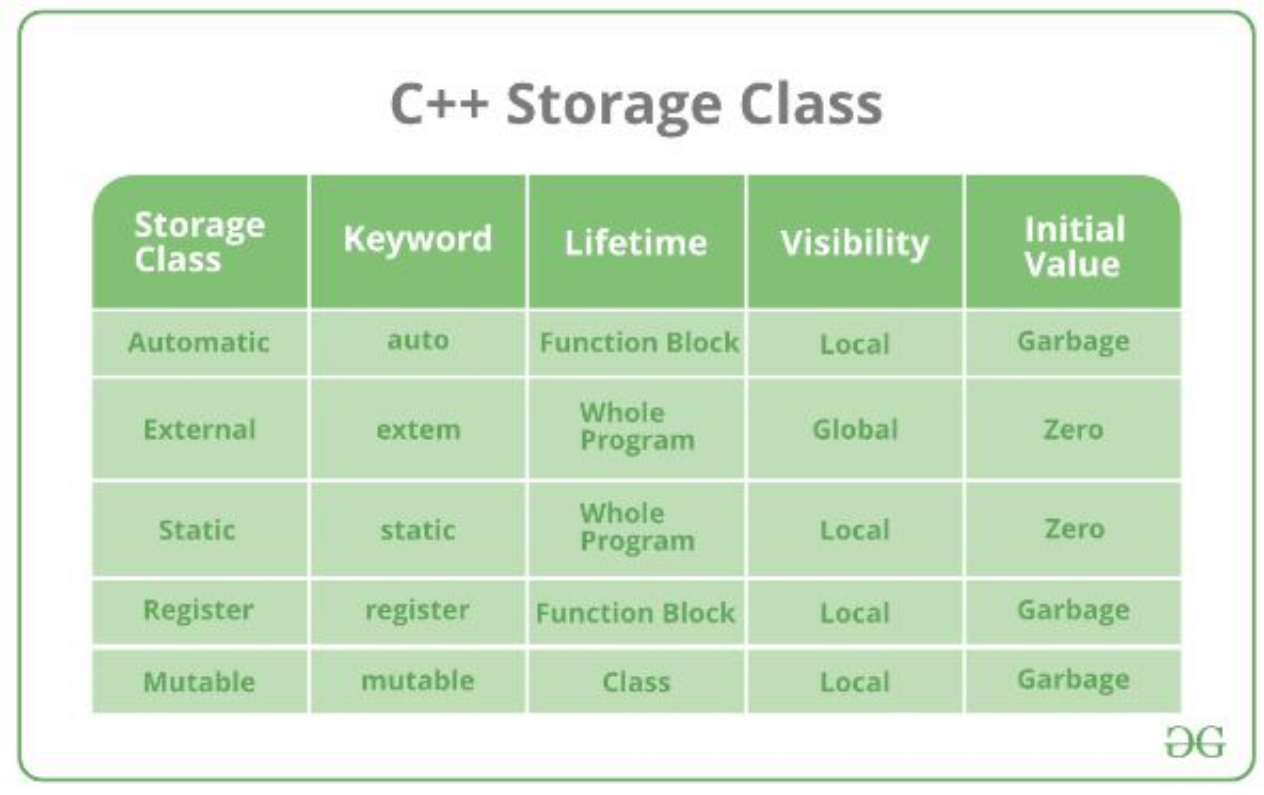
CSC/ITS – 101(PROGRAMMING 1)

What is a Storage Class in C++?

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

What is a Storage Class in C++?



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).

The register Storage Class

Computer Registers

-Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

-A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

-The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

(https://www.javatpoint.com/computer-registers, Ret. 9/24/2023)

The register Storage Class

```
{
   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' does 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

1. The static storage class instructs the compiler to keep a local variable in existence during the life-time 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.
2. 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.
3. 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 static Storage 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;
}
```

The extern Storage Class

1. 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.
2. 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 file.
3. The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained on the next page. Try it!

The extern Storage Class

First File: main.cpp

```
#include <iostream>
int count ;
extern void write_extern();

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

The extern Storage Class

Second File: support.cpp

```
#include <iostream>

extern int count;

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

The mutable Storage Class

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

Lesson 8 - Operators in C++

CSC/ITS – 101(PROGRAMMING 1)

What are Operators in C++?

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language –

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Relational Operators

There are following relational operators supported by C++ language
Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is false.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is false.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is false.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

There are following logical operators supported by C++ language.
Assume variable A holds 1 and variable B holds 0, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows –

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Bitwise Operators

Assume if A = 60; and B = 13; now in binary format they will be as follows –

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Bitwise Operators

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. (Sometimes used as **insertion operator** when preceded with cout object of ostream library)	A << 2 will give 240 which is 1111 0000 (cout<<"A"<<endl; //will print the 240)
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. (Sometimes used as **extraction operator** when preceded with cin object of istream library)	A >> 2 will give 15 which is 0000 1111 (cin>>A; //will get the value of A from the user upon input)

Assignment Operators

There are following assignment operators supported by C++ language –

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A

Assignment Operators

There are following assignment operators supported by C++ language –

Operator	Description	Example
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where ‘a’ is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	*** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operators Precedence in C++

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

QUESTIONS?