C Programming Short Notes

Part 4

1. Storage Classes

A storage class defines the scopes and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a c program

Auto:

- The auto storage class is the **default for all variables declared within a function or a block.**
- Auto variables are accessible **only within the block or function where they are declared.**
- They are assigned a **garbage value** by default when declared.
- Although rarely used explicitly, auto variables can be accessed within nested blocks within the parent block or function.
- Pointers can also access them outside their scope by pointing to their exact memory location.

Example:

```
void exampleFunction() {
    auto int x = 10;
    // 'x' is an auto variable
}
```

Extern:

- The extern storage class indicates that a variable is defined elsewhere (not within the same block where it is used).
- It allows **global variables to be initialized with a legal value where they are declared, making them accessible across functions or blocks**.
- **Normal global variables can also be made extern by using the extern keyword before their declaration.**
- Useful for accessing variables between different files in a large program.

Example:

```
extern int globalVar; // Declaration
int main() {
    globalVar = 42; // Accessing the global variable
}
```

Static:

- Static variables **preserve their value even after going out of scope.**
- They are **initialized only once and exist until program termination.**
- Local to the function where they are defined.
- Global static variables can be accessed anywhere in the program.
- Compiler assigns them a **default value of 0.**

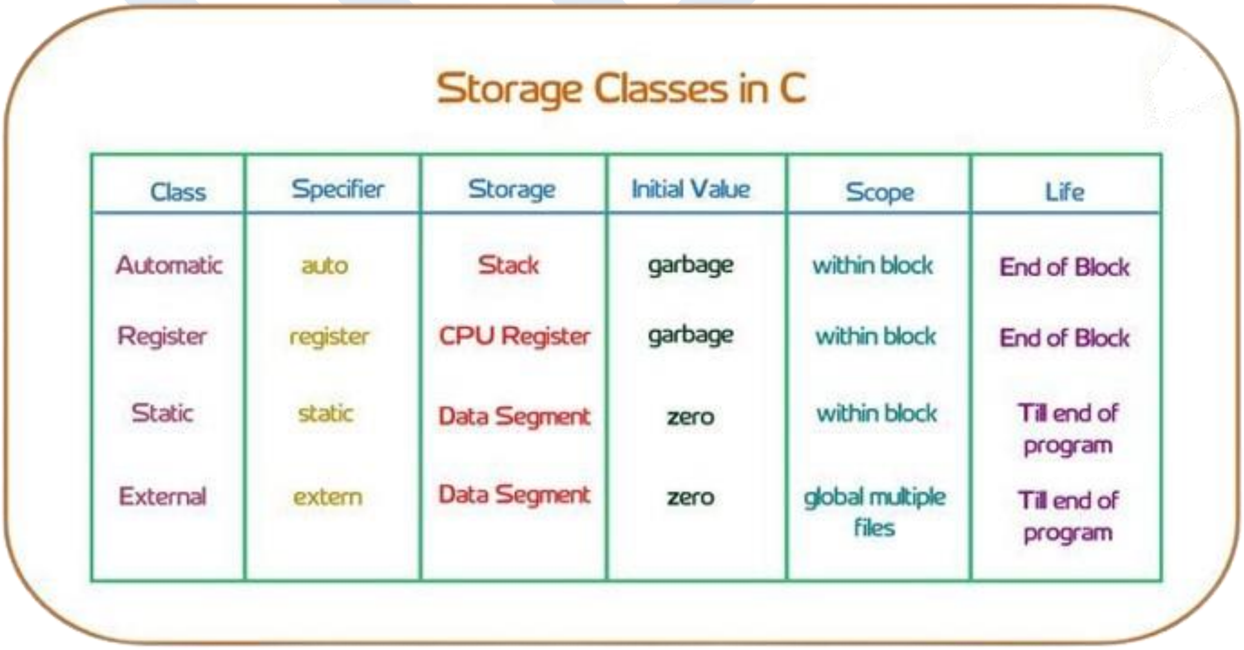
Example:

```
void counter() {
    static int count = 0; // Initialized only once
    count++;
    printf("Count: %d\n", count);
}
```

Register:

- **Register variables have the same functionality as auto variables.**
- Compiler attempts **to store them in CPU registers for faster access during program runtime**.
- If **no free register is available, they are stored in memory.**

Example: register int speed = 100; // May be stored in a register



Storage Classes in C

Class	Specifier	Storage	Initial Value	Scope	Life
Automatic	auto	Stack	garbage	within block	End of Block
Register	register	CPU Register	garbage	within block	End of Block
Static	static	Data Segment	zero	within block	Till end of program
External	extern	Data Segment	zero	global multiple files	Till end of program

2. Scope of the variable

a. Global Scope:

- **Variables declared outside any function or block have global scope.**
- These global variables are **visible throughout the entire program.**
- They are also known as **file-scope variables because their scope extends from the beginning to the end of the file.**

Example:

```
#include <stdio.h>

int global = 5; // Global variable

void display() {
   printf("%d\n", global);
}


int main() {

   printf("Before change within main: ");

   display();

   global = 10;

   printf("After change within main: ");

   display();

   return 0;

}
```

Output:

```
Before change within main: 5
After change within main: 10
```

Linkage of Global Variables:

- By default, global variables have external linkage, meaning they can be accessed in other C source files.
- To restrict access to the current file only, global variables can be marked as static.

b. Local Scope:

- Variables **declared inside a function or block have local scope**.
- They are **visible only within the block** where they are defined.
- Local variables are also called **block-scope variables.**

Example:

```
#include <stdio.h>

int main() {
    {
        int x = 10, y = 20;
        {
            printf("x = %d, y = %d\n", x, y);
            {
                int y = 40;
                x++;
                y++;
                printf("x = %d, y = %d\n", x, y);
            }
            printf("x = %d, y = %d\n", x, y);
        }
    }
    return 0;
}
```

Output:

```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

- **Local variables have internal linkage.**
- If an inner block declares a variable with the same name as an outer block variable, the visibility of the outer block variable ends at the point of the inner declaration.

3. Strings in c

A string is a sequence of characters **terminated with a null character** ('\0').

1. String Basics:

- **A string in C is stored as an array of characters.**
- The difference between a character array and a C string lies in the termination character. A C string is always terminated with a unique character '\0'.
- To declare a string, you can use the following basic syntax:

```
char string_name[size];
```

- string_name is any name given to the string variable.
- size defines the length of the string (i.e., the number of characters it will store).

2. String Initialization:

You can initialize a C string in several ways:

- Assigning a String Literal without Size:

```
char str[] = "RasimRockers";
```

- Assigning a String Literal with a Predefined Size:

```
char str[50] = "RasimRockers";
```

  - Always account for one extra space for the null character.

- Assigning Character by Character with Size:

```
char str[14] = {'R', 'a', 's', 'i', 'm', 'R', 'o', 'c', 'k', 'e', 'r', 's', '\0'};
```

- Assigning Character by Character without Size:

```
char str[] = {'R', 'a', 's', 'i', 'm', 'R', 'o', 'c', 'k', 'e', 'r', 's', '\0'};
```

Example Program:

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Rasim";
    printf("%s\n", str);

    int length = strlen(str);
    printf("Length of string str is %d", length);

    return 0;
      }
```

Output:

```
Rasim
Length of string str is 5
```

- In the example program, we print the string using printf, and unlike arrays, we don't need to print the string character by character.
- The null character ('\0') is crucial for indicating the end of a C string.

In C, there are several ways to handle string input and output. Here are a few common methods:

1. Using **'printf()'** and **'scanf()'** for input and output respectively:

   Example:

```
#include <stdio.h>
int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);
    printf("You entered: %s\n", str);
    return 0;
}
```

Output :

```
Enter a string: Hello, World!
You entered: Hello,
```

2. Using **‘printf()’** and **‘gets()’**(not recommended due to security risks) for input and output respectively:

   Example:

```
#include <stdio.h>
  int main() {
      char str[100];
      printf("Enter a string: ");
      gets(str);
      printf("You entered: %s\n", str);
      return 0;
  }
```

   Output :

```
Enter a string: Hello, World!
You entered: Hello, World!
```

3. Using **‘printf()’** and **‘puts()’** for output (if input is not required):

   Example:

```
#include <stdio.h>
int main() {
    char str[] = "Hello, World!";
    printf("The string is: %s\n", str);
    puts("This is another way to output a string.");
    return 0;
}
```

   Output :

```
The string is: Hello, World!
This is another way to output a string.
```

4. Using **‘gets()’** and **‘puts()’** for input and output respectively:

```
#include <stdio.h>
int main() {
    char str[100];
    printf("Enter a string: ");
    gets(str);
    puts("You entered:");
    puts(str);
    return 0;
}
```

Output :

```
Enter a string: Hello, World!
You entered:
Hello,World!
```

5. Using **‘printf()’** and **‘fgets()’** for input and output respectively (preferred for safer input):

Example:

```
#include <stdio.h>
int main() {
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    printf("You entered: %s\n", str);
    return 0;
}
```

Output :

```
Enter a string: Hello, World!
You entered: Hello, World!
```

Using Function with String :-

Example :-

```
#include <stdio.h>
#include <string.h>

void printLength(const char *str);
void concatenateStrings(char *dest, const char *src);

int main() {
    char str1[50] = "Hello";
    char str2[] = "World!";
    printf("Length of str1: ");
    printLength(str1);
    printf("Before concatenation:\n");
    printf("str1: %s\n", str1);
    printf("str2: %s\n", str2);
    concatenateStrings(str1, str2);
    printf("After concatenation:\n");
    printf("str1: %s\n", str1);
    printf("str2: %s\n", str2);
    return 0;
}
void printLength(const char *str) {
    printf("%zu\n", strlen(str));
}
void concatenateStrings(char *dest, const char *src) {
    strcat(dest, src);
}
```

Output

```
Length of str1: 5
Before concatenation:
str1: Hello
str2: World!
After concatenation:
str1: HelloWorld!
str2: World!
```