C Storage Classes

Storage classes define the **lifetime**, **visibility**, and **memory location** of variables.

There are four main storage class specifiers in C:

- auto
- static
- register
- extern

Difference Between Scope and Storage Classes

Scope defines where a variable can be used, and **storage classes** define how long it lasts and where it's stored. This chapter continues from the C Scope chapter.

---

auto

The auto keyword is used for local variables. It is default for variables declared inside functions, so it's rarely used explicitly.

Example

```
int main() {
  auto int x = 50;  // Same as just: int x = 50;
  printf("%d\n", x);
  return 0;
}
```

---

static

The static keyword changes how a variable or function behaves in terms of **lifetime** and **visibility**:

- **Static local variables** keep their value between function calls.
- **Static global variables/functions** are not visible outside their file.

Example

```
void count() {
  static int myNum = 0; // Keeps its value between calls
  myNum++;
  printf("num = %d\n", myNum);
}

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

Result:

```
num = 1
num = 2
num = 3
```

Try to remove the static keyword from the example to see the difference.

---

register

The register keyword suggests that the variable should be stored in a CPU register (for faster access).

You cannot take the address of a register variable using &.

Note: The register keyword is mostly obsolete - modern compilers automatically choose the best variables to keep in registers, so you usually don't need to use it.

Example

```
int main() {
```

```c
  register int counter = 0;

  printf("Counter: %d\n", counter);

  return 0;

}
```

---

extern

The extern keyword tells the compiler that a variable or function is defined in another file.

It is commonly used when working with **multiple source files**.

File 1: main.c

```c
#include <stdio.h>


extern int shared;  // Declared here, defined in another file


int main() {

  printf("shared = %d\n", shared);

  return 0;

}
```

File 2: data.c

```c
int shared = 50;  // Definition of the variable
```

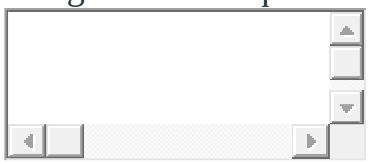
File Handling in C

File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such

as **fopen()**, **fwrite()**, **fread()**, **fseek()**, **fprintf()**, etc. to perform input, output, and many different C file operations in our program.

Open a File in C

For opening a file in C, the fopen() function is used with the filename or file path along with the required **access modes**.



```c
#include <stdio.h>
#include <stdlib.h>

int main() {

    // File pointer to store the
    // value returned by fopen
    FILE* fptr;

    // Opening the file in read mode
    fptr = fopen("filename.txt", "r");

    // checking if the file is
    // opened successfully
    if (fptr == NULL) {
        printf("The file is not opened.");
    }
    return 0;
}
```

Output

```
The file is not opened.
```

The file is not opened because it does not exist in the source directory. But the fopen() function is also capable of creating a file if it does not exist.

Note: If is essential to check for NULL values that might be returned by the fopen() function to avoid any errors.

Syntax of fopen()

```
FILE* fopen(*file_name, *access_mode);
```

Parameters

- ***file_name:*** name of the file when present in the same directory as the source file. Otherwise, full path.
- ***access_mode:*** Specifies for what operation the file is being opened.

Return Value

- If the file is opened successfully, returns a file pointer to it.
- If the file is not opened, then returns NULL.

File Opening Modes

File opening modes or access modes specify the allowed operations on the file to be opened. They are passed as an argument to the **fopen()** function. Some of the commonly used file access modes are listed below:

Opening Modes	Description
r	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
w	Open for writing in text mode. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
a	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens only in the append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
r+	Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file.
w+	Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.
a+	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens the file in both reading and append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.

Create a File

The fopen() function can not only open a file but also can create a file if it does not exist already.

```
#include <stdio.h>

#include <stdlib.h>


int main() {


    // File pointer

    FILE* fptr;


    // Creating file using fopen()

    // with access mode "w"

    fptr = fopen("file.txt", "w");


    // checking if the file is created

    if (fptr == NULL)

        printf("The file is not opened.");

    else

        printf("The file is created Successfully.");

    return 0;

}
```

Output

```
The file is created Successfully.
```

Write to a File
The file write operations can be performed by the functions fprintf() and fputs(). C programming also provides some other functions that can be used to write data to a file such as:

Function	Description
fprintf()	Similar to printf(), this function uses formatted string and variable arguments list to print output to the file.
fputs()	Prints the whole line in the file and a newline at the end.
fputc()	Prints a single character into the file.
fputw()	Prints a number to the file.

```
#include <stdio.h>

#include <stdlib.h>


int main() {


    // File pointer

    FILE* fptr;


    // Get the data to be written in file

    char data[50] = "GeeksforGeeks-A Computer "

                    "Science Portal for Geeks";


    // Creating file using fopen()
```

```
    // with access mode "w"
    fptr = fopen("file.txt", "w");

    // Checking if the file is created
    if (fptr == NULL)
        printf("The file is not opened.");
    else{
        printf("The file is now opened.\n");
        fputs(data, fptr);
        fputs("\n", fptr);

        // Closing the file using fclose()
        fclose(fptr);
        printf("Data successfully written in file "
               "file.txt\n");
        printf("The file is now closed.");
    }
    return 0;
}
```

Output

```
The file is now opened.
Data successfully written in file file.txt
The file is now closed.
```

Reading From a File

The file read operation in C can be performed using functions **fscanf()** or **fgets()**. Both the functions performed the same operations as that of **scanf()** and gets but

with an additional parameter, the file pointer. There are also other functions we can use to read from a file. Such functions are listed below:

Function	Description
fscanf()	Use formatted string and variable arguments list to take input from a file.
fgets()	Input the whole line from the file.
fgetc()	Reads a single character from the file.
fgetw()	Reads a number from a file.
fread()	Reads the specified bytes of data from a binary file.

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

int main() {
   FILE* fptr;

   // Declare the character array
   // for the data to be read from file
   char data[50];
   fptr = fopen("file.txt", "r");

   if (fptr == NULL) {
      printf("file.txt file failed to open.");
   }
   else {

      printf("The file is now opened.\n");

      // Read the data from the file
      // using fgets() method
      while (fgets(data, 50, fptr)
            != NULL) {

         // Print the data
         printf("%s", data);
      }

      // Closing the file using fclose()
      fclose(fptr);
   }
```

```
    return 0;
}
```

Output

```
The file is now opened.
GeeksforGeeks-A Computer Science Portal for Geeks
```

The getc() and some other file reading functions return **EOF** (**E**nd **O**f **F**ile) when they reach the end of the file while reading. EOF indicates the end of the file, and its value is implementation-defined. Reading more after EOF results in undefined error so, it is always recommended to check for EOF while reading a file.

Note: One thing to note here is that after reading a particular part of the file, the file pointer will be automatically moved to the end of the last read character.

Closing a File

The **fclose()** function is used to close the file. After successful file operations, you must always close a file to remove it from the memory.

Syntax:

```
fclose(file_pointer);
```

Move File Pointer

File pointer generally points to the position according to the mode or last read/write operation. We can manually move this pointer to any position in the file using **fseek()** function.

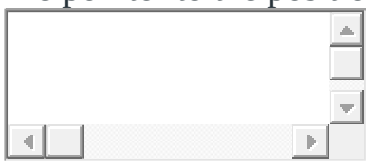
Syntax:

```
fseek(fptr, offset, pos);
```

where, ***pos*** is the position from where offset is counted and ***offset*** is the number of positions to shift from pos (it can be negative or positive)**.**

Example:

While writing to a file opened in **rw+** mode, the file pointer moves to the end of the file. In case where we want to replace a word, then first we have to move the file pointer to the position where that word starts.



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

int main() {

    // File pointer
    FILE* fptr;

    // Get the data to be written in file
    char data[50] = "GeeksforGeeks-A Computer "
                    "Science Portal for Geeks";

    // Creating file using fopen()
    // with access mode "w"
    fptr = fopen("file.txt", "w");
```

```
    // Checking if the file is created
    if (fptr == NULL)
        printf("The file is not opened.");
    else{
        printf("The file is now opened.\n");
        fputs(data, fptr);
        fputs("\n", fptr);
        fseek(fptr, -6, SEEK_END);

        fputs("GeeksforGeeks", fptr);

        // Closing the file using fclose()
        fclose(fptr);
        printf("Data successfully written in file "
               "file.txt\n");
        printf("The file is now closed.");
    }
    return 0;
}
```

Output

```
The file is now opened.

Data successfully written in file file.txt

The file is now closed.
```

Now, imagine you want to read this file after writing. We can use **fseek()** here too, but these is one more function specifically for this purpose which is **rewind().**

More Functions for C File Operations

The following table lists some more functions that can be used to perform file operations or assist in performing them.

Functions	Description
fopen()	It is used to create a file or to open a file.
fclose()	It is used to close a file.

Functions	Description
fgets()	It is used to read a file.
fprintf()	It is used to write blocks of data into a file.
fscanf()	It is used to read blocks of data from a file.
getc()	It is used to read a single character to a file.
putc()	It is used to write a single character to a file.
fseek()	It is used to set the position of a file pointer to a mentioned location.
ftell()	It is used to return the current position of a file pointer.
rewind()	It is used to set the file pointer to the beginning of a file.
putw()	It is used to write an integer to a file.
getw()	It is used to read an integer from a file.

Command Line Arguments in C

The most important function of C is the main() function. It is mostly defined with a return type of int and without parameters.

```
int main() {
   ...
}
```

We can also give command-line arguments in C. Command-line arguments are the values given after the name of the program in the command-line shell of Operating Systems. Command-line arguments are handled by the main() function of a C program.

To pass command-line arguments, we typically define main() with two arguments: the first argument is the **number of command-line arguments** and the second is a **list of command-line arguments.**

Syntax

```
int main(int argc, char *argv[]) { /* ... */ }
        or
int main(int argc, char **argv) { /* ... */ }
```

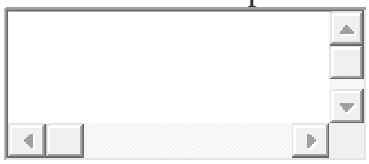
Here,

- **argc (ARGument Count)** is an integer variable that stores the number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, the value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non-negative.
- **argv (ARGument Vector)** is an array of character pointers listing all the arguments.
- If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

For better understanding run this code on your Linux machine.

Example

The below example illustrates the printing of command line arguments.



```
// C program named mainreturn.c to demonstrate the working
// of command line arguement
#include <stdio.h>

// defining main with arguments
int main(int argc, char* argv[])
{
    printf("You have entered %d arguments:\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Output

```
You have entered 4 arguments:
./main
geeks
for
geeks
```

for Terminal Input

```
$ g++ mainreturn.cpp -o main $ ./main geeks for geeks
```

Note: *Other platform-dependent formats are also allowed by the C standards; for example, Unix (though not POSIX.1) and Microsoft Visual C++ have a third argument giving the program's environment, otherwise accessible through getenv in stdlib.h. Refer C program to print environment variables for details.*

Properties of Command Line Arguments in C

1. They are passed to the main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control programs from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[argc-1] points to the last argument.

Note: *You pass all the command line arguments separated by a space, but if the argument itself has a space, then you can pass such arguments by putting them inside double quotes "" or single quotes ''.*

Example

The below program demonstrates the working of command line arguments.

```
// C program to illustrate
// command line arguments
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Program name is: %s", argv[0]);

    if (argc == 1)
        printf("\nNo Extra Command Line Argument Passed "
               "Other Than Program Name");

    if (argc >= 2) {
        printf("\nNumber Of Arguments Passed: %d", argc);
        printf("\n----Following Are The Command Line "
               "Arguments Passed----");
        for (int i = 0; i < argc; i++)
            printf("\nargv[%d]: %s", i, argv[i]);
```

```
    }
    return 0;
}
```

Output in different scenarios:

1. Without argument: When the above code is compiled and executed without passing any argument, it produces the following output.

Terminal Input

```
$ ./a.out
```

Output

```
Program Name Is: ./a.out
No Extra Command Line Argument Passed Other Than Program Name
```

2. Three arguments: When the above code is compiled and executed with three arguments, it produces the following output.

Terminal Input

```
$ ./a.out First Second Third
```

Output

```
Program Name Is: ./a.out
Number Of Arguments Passed: 4
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First
argv[2]: Second
argv[3]: Third
```

3. Single Argument: When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

Terminal Input

```
$ ./a.out "First Second Third"
```

Output

```
Program Name Is: ./a.out
Number Of Arguments Passed: 2
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First Second Third
```

4. A single argument in quotes separated by space: When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

Terminal Input

```
$ ./a.out 'First Second Third'
```

Output

```
Program Name Is: ./a.out
Number Of Arguments Passed: 2
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First Second Third
```

Garphics:

In C graphics, the graphics.h functions are used to draw different shapes like circles, rectangles, etc, display text(any message) in a different format (different fonts and colors). By using the functions in the header graphics.h, programs, animations, and different games can also be made.

Functions used:

- **line(x1, y1, x2, y2):** It is a function provided by graphics.h header file to draw a line. Here x1, y1 is the first coordinates of the line, and x2, y2 are the second coordinates of the line respectively.
- **circle(x, y, r):** It is a function provided by graphics.h header file to draw a circle. The x, y are the center points of the circle and r is the radius of the circle.
- **rectangle(X1, Y1, X2, Y2):** It is employed in the creation of a rectangle. The rectangle must be drawn using the coordinates of the left top and right bottom corners. The X-coordinate and Y-coordinate of the top left corner are **X1** and **Y1** and the X-coordinate and Y-coordinate of the bottom right corner are **X2** and **Y2** respectively.
- **delay(n):** It is used to hold the program for a specific time period. Here n is the number of seconds you want to hold the program.
- **cleardevice():** It is used to clear the screen in graphic mode. It sets the position of the cursor to its initial position, that is, (0, 0) coordinates.
- **closegraph():** It is used to close the graph.

Syntax of Graphics Program in C

The general syntax structure for a graphics program in C using the *graphics.The* library looks like this:

1. **#include <graphics.h>**: Adds the graphics library for drawing functions.
2. **initgraph(&gd, &gm, NULL)**: Initializes graphics mode; gd detects the driver, gm sets the mode.
3. **Graphics Functions:** Use functions like *circle()*, *line()*, or *rectangle()* to draw shapes.
4. **getch():** Waits for a key press before exiting.
5. **closegraph():** Closes the graphics mode and deallocates resources.

Drawing A Circle

A basic **graphics program in C** involves setting up a graphics mode, drawing basic shapes, and then closing the graphics mode after the work is done. Below is a simple program that initializes the graphics mode and draws a circle:

```
#include <graphics.h>

#include <conio.h>   // For getch()



int main() {

    int gd = DETECT, gm;



    // Initialize graphics mode

    initgraph(&gd, &gm, NULL);



    // Draw a circle with center (300, 300) and radius 50

    circle(300, 300, 50);



    // Wait for user input before closing the window

    getch();



    // Close the graphics mode

    closegraph();



    return 0;

}
```

Output:

Applications of Graphics Programming in C

1. **Game Development:** Graphic programming is essential for games, as it enables the development of 2-D and 3-D graphics and the implementation of interactivity.
2. **Computer-Aided Design (CAD):** In CAD applications, graphics programming is used to model and control 2D and 3D objects.
3. **Data Visualisation:** Graphics programming transforms data into graphical information such as charts, graphs, or diagrams.
4. **Image Processing:** C for graphics programming helps in activities such as image Enhancement, Filtering and manipulation.
5. **Graphical User Interfaces (GUIs):** Graphic programming involves developing interactive, user-friendly interfaces for software programs.
6. **Simulations and Visualisations:** It creates simulations and visualises real-world situations using physical simulation, scientific simulations, and more.
7. **Embedded Systems:** Graphics programming is used in embedded systems to deliver practical graphical user interfaces and inputs for devices such as smart appliances and medical instruments.

Pros and Cons of Graphics Program

Here are some of the pros and cons of **graphics programs in C**:

Pros:

- **Enhanced Visual Appeal:** Creates engaging, interactive visuals for applications such as games and GUIs.
- **Real-Time Interactivity:** Enables real-time user interaction with graphics, improving user experience.
- **Wide Application:** Useful in diverse fields like gaming, CAD, simulations, and data visualisation.
- **Creativity:** Offers endless possibilities for creative design and animation.
- **Hardware Access:** Graphics libraries like OpenGL and SDL provide direct access to hardware for optimised performance.

Cons:

- **Complexity:** Requires advanced programming skills and understanding of graphical concepts.
- **Performance Demands:** Can be resource-intensive, requiring powerful hardware for smooth performance.
- **Platform Dependency:** Some libraries are platform-specific, limiting cross-platform compatibility.
- **Steep Learning Curve:** Learning the intricacies of different graphics libraries can be challenging for beginners.
- **Debugging Challenges:** Visual bugs can be harder to diagnose than traditional programming bugs.

In C programming for graphics, various data types are utilized to represent graphical elements and properties, and global variables can be employed to manage shared data across different graphics functions.

Common Data Types in C Graphics:

- `int`:

Used for integer values, such as coordinates (x, y), dimensions (width, height), color components (0-255), and loop counters.

- `float` / `double`:

  Used for floating-point numbers, especially when dealing with precise calculations for transformations, scaling, or color blending. `double` offers higher precision than `float`.

- `char`:

  Used for single characters, such as text to be displayed or input from the user. Arrays of `char` are used to store strings.

- `struct`:

  User-defined data types are frequently used to group related data for graphical objects. Examples include:

- `Point`: `struct Point { int x; int y; };`
- `Color`: `struct Color { unsigned char r, g, b; };`
- `Rectangle`: `struct Rectangle { int x1, y1, x2, y2; };`
- `unsigned char`:
  Often used for individual color components (Red, Green, Blue, Alpha) as their values typically range from 0 to 255.

Global Variables in C Graphics:

Global variables are declared outside of any function and can be accessed and modified by any function in the program. In graphics programming, they are useful for:

- **Screen dimensions**: Storing the width and height of the graphics window or drawing area.

```
int screenWidth = 800;
int screenHeight = 600;
```

- **Current drawing color**: Maintaining the active color for drawing operations.

```
struct Color currentColor = {255, 0, 0}; // Red
```

- **Graphics mode or state**: Keeping track of the current graphics mode (e.g., drawing lines, filling shapes) or other rendering states.

```
int drawingMode = LINE_MODE; // A defined constant
```

- **Pointers to graphics context**: In some graphics libraries, a global pointer to a graphics context or display surface might be used.

```
SDL_Surface* screen = NULL; // Example with SDL library
```

- **Shared resources**: Managing access to shared resources like textures, fonts, or other assets that are used across multiple drawing functions.

Graphics Example Using Colours of C Language Graphics

A Coloured Circle and a Rectangle:

```
#include <graphics.h>

#include <conio.h>



int main() {

    int gd = DETECT, gm;



    // Initialize graphics mode

    initgraph(&gd, &gm, NULL);



    // Set color and draw a circle

    setcolor(RED);

    circle(200, 200, 100);



    // Set color and draw a rectangle

    setcolor(BLUE);

    rectangle(150, 150, 250, 250);
```

```
    // Wait for user input
    getch();


    // Close graphics mode
    closegraph();


    return 0;
}
```

Output: