

# C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

## Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>

int main () {

    /* local variable declaration */
    int a, b;
    int c;

    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;

    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

    return 0;
}
```

## OUTPUT

```
value of a = 10, b = 20 and c = 30
```

## Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

    /* local variable declaration */
    int a, b;

    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;

    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

    return 0;
}
```

## OUTPUT

value of a = 10, b = 20 and g = 30

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example –

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

    /* local variable declaration */
    int g = 10;

    printf ("value of g = %d\n", g);

    return 0;
}
```

## OUTPUT

value of g = 10

## Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example –

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

    /* local variable declaration in main function */
    int a = 10;
    int b = 20;
    int c = 0;

    printf ("value of a in main() = %d\n",  a);
    c = sum( a, b);
    printf ("value of c in main() = %d\n",  c);

    return 0;
}

/* function to add two integers */
int sum(int a, int b) {

    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

## OUTPUT

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

## Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows –

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

## Compile time vs Runtime

Compile-time and Runtime are the two programming terms used in the software development. Compile-time is the time at which the source code is converted into an executable code while the run time is the time at which the executable code is started running. Both the compile-time and runtime refer to different types of error.

### Compile-time errors

Compile-time errors are the errors that occurred when we write the wrong syntax. If we write the wrong syntax or semantics of any programming language, then the compile-time errors will be thrown by the compiler. The compiler will not allow to run the program until all the errors are removed from the program. When all the errors are removed from the program, then the compiler will generate the executable file.

The compile-time errors can be:

- Syntax errors
- Semantic errors

### Syntax errors

When the programmer does not follow the syntax of any programming language, then the compiler will throw the syntax error.

For example,

```
int a, b:
```

The above declaration generates the compile-time error as in C, every statement ends with the semicolon, but we put a colon (:) at the end of the statement.

### Semantic errors

The semantic errors exist when the statements are not meaningful to the compiler.

For example,

```
a+b=c;
```

The above statement throws a compile-time error. In the above statement, we are assigning the value of 'c' to the summation of 'a' and 'b' which is not possible in C programming language as it can contain only one variable on the left of the assignment operator while right of the assignment operator can contain more than one variable.

The above statement can be re-written as:

```
c=a+b;
```

## Runtime errors

The runtime errors are the errors that occur during the execution and after compilation. The examples of runtime errors are division by zero, etc. These errors are not easy to detect as the compiler does not point to these errors.

### Let's look at the differences between compile-time and runtime:

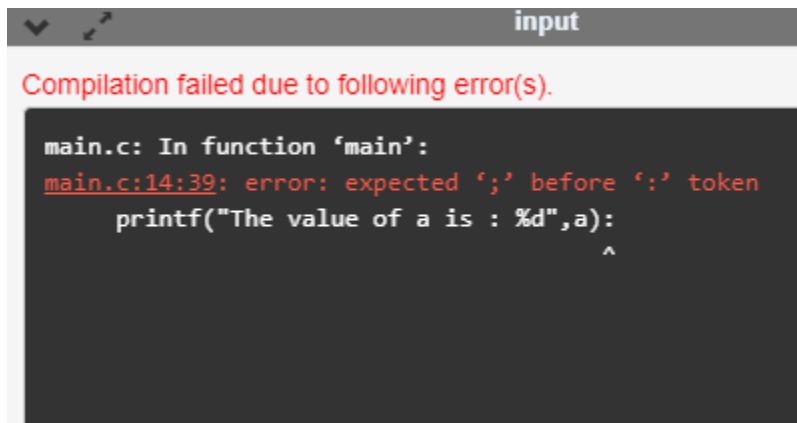
Compile-time	Runtime
The compile-time errors are the errors which are produced at the compile-time, and they are detected by the compiler.	The runtime errors are the errors which are not generated by the compiler and produce an unpredictable result at the execution time.
In this case, the compiler prevents the code from execution if it detects an error in the program.	In this case, the compiler does not detect the error, so it cannot prevent the code from the execution.
It contains the syntax and semantic errors such as missing semicolon at the end of the statement.	It contains the errors such as division by zero, determining the square root of a negative number.

### Example of Compile-time error

```
#include <stdio.h>
int main()
{
    int a=20;
    printf("The value of a is : %d",a):
    return 0;
}
```

OUTPUT

In the above code, we have tried to print the value of 'a', but it throws an error. We put the colon at the end of the statement instead of a semicolon, so this code generates a compile-time error.



input

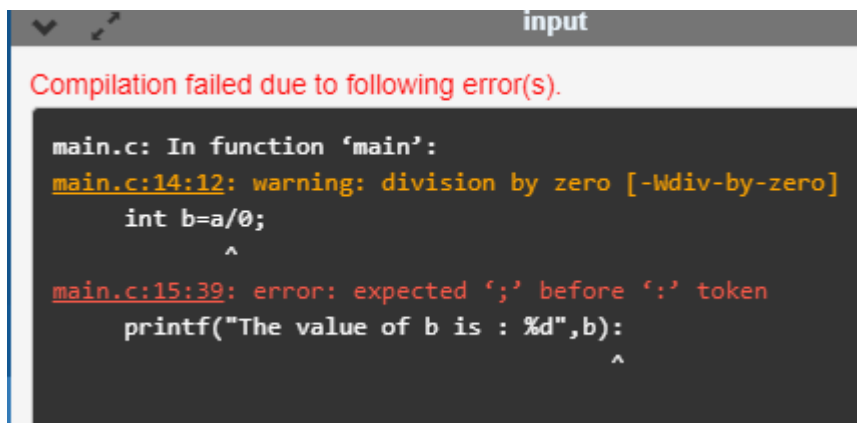
Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:14:39: error: expected ';' before ':' token
    printf("The value of a is : %d",a):
                                   ^
```

### Example of runtime error

```
#include <stdio.h>
int main()
{
    int a=20;
    int b=a/0; // division by zero
    printf("The value of b is : %d",b);
    return 0;
}
```

### OUTPUT



input

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
    int b=a/0;
           ^
main.c:15:39: error: expected ';' before ':' token
    printf("The value of b is : %d",b):
                                   ^
```

## Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

(type)value;

*Note: It is always recommended to convert the lower value to higher for avoiding data loss.*

### Without Type Casting:

```
int f= 9/4;  
printf("f : %d\n", f );
```

Output: 2

### With Type Casting:

```
float f=(float) 9/4;  
printf("f : %f\n", f );
```

Output: 2.250000

## C - Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

### Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.

- Every local variable is automatic in C by default.

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 in the same storage class. 'auto' can only be used within functions, i.e., local variables.

```
#include <stdio.h>
int main()
{
    int a; //auto
    char b;
    float c;
    printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
    return 0;
}
```

OUTPUT

Garbage value Garbage value Garbage value

```
#include <stdio.h>
int main()
{
    int a = 10,i;
    printf("%d ",++a);
    {
        int a = 20;
        for (i=0;i<3;i++)
        {
            printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
        }
    }
    printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

OUTPUT

11 20 20 20 11

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

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



```

int main( )
{
    auto int j = 1;
    {
        auto int j= 2;
        {
            auto int j = 3;
            printf ( " %d ", j);
        }
        printf ( "\t %d ",j);
    }
    printf( "%d\n", j);
}

```

## OUTPUT

3 2 1

## External

- extern variable is a programmer's shorthand to represent external variable.
- Extern storage class is used when we have global functions or variables which are shared between two or more files.
- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- We can also access extern variables of one file to another file. But make sure both files are in same folder.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we cannot initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Let's work out a program to demonstrate extern storage class in C. First create a file named **variable.h** where you put all your variables with **extern** keyword which can be used by any program by simply including the file name in it.

### variable.h

```

extern int num1 = 9;
extern int num2 = 1;

```

### extern.c

```

#include <stdio.h>
#include "variable.h"
int main()
{
    int add = num1 + num2;
    printf("%d + %d = %d ", num1, num2, add);
    return 0;
}

```

```
}
```

## OUTPUT

9+1=10

Though extern variable is declared above the main function. We can redeclare the extern variable in the main function which will overwrite the global extern variable only within the function.

```
#include <stdio.h>
extern int num1 = 1;
extern int num2 = 2;
int main()
{
    int num1 = 3;
    int num2 = 4;
    int add = num1 + num2;
    printf("%d + %d = %d ", num1, num2, add);
    return 0;
}
```

## OUTPUT

3+4=7

```
#include <stdio.h>
int main()
{
    extern int a;
    printf("%d",a);
}
```

## OUTPUT

main.c:(.text+0x6): undefined reference to `a'  
collect2: error: ld returned 1 exit status

```
#include <stdio.h>
int a;
int main()
{
    extern int a; // variable a is defined globally, the memory will not be allocated to a
    printf("%d",a);
}
```

## OUTPUT

0

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

```

int a;
int main()
{
extern int a = 0; // this will show a compiler error since we can not use extern and initializer at same time
printf("%d",a);
}

```

## OUTPUT

Compilation failed due to following error(s).

```

main.c: In function 'main':
main.c:13:16: error: 'a' has both 'extern' and initializer
    extern int a = 0; // this will show a compiler error since we can not use extern and initializer at same time
               ^

```

```

#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the program or not.
printf("%d",a);
}

```

```
int a = 20;
```

## OUTPUT

20

```

extern int a;
int a = 10;
#include <stdio.h>
int main()
{
printf("%d",a);
}

```

```
int a = 20; // compiler will show an error at this line
```

## OUTPUT

Compilation failed due to following error(s).

```
main.c:16:5: error: redefinition of 'a'
  int a = 20; // compiler will show an error at this line
      ^
main.c:10:5: note: previous definition of 'a' was here
  int a = 10;
      ^
```

## Static

- A static variable is a variable that tells the compiler to retain the value until the program terminates.
- They are created once when the function is called, even though the function gets repeated it retains the same value and exists until the program terminates.
- The default value of static variable is zero(0).
- The keyword for a variable to declared under static storage class is **static**

## Syntax - Static Storage Class

```
static int st;
```

## Example

```
#include <stdio.h>
//function declaration
void subfun();
int main()
{
  subfun();
  subfun();
  subfun();
  return 0;
}
//function definition
void subfun()
{
  static int st = 1; //static variable declaration
  printf("\nst = %d ", st);
  st++;
}
```

## OUTPUT

```
st = 1
st = 2
st = 3
```

```
#include <stdio.h> /* function declaration */
void next(void);
```

```

static int counter = 7; /* global variable */
main() {
    while(counter<10) {
        next();
        counter++;    }
return 0;}
void next( void ) {    /* function definition */
    static int iteration = 13; /* local static variable */
    iteration ++;
    printf("iteration=%d and counter= %d\n", iteration, counter);}

```

## OUTPUT

```

iteration=14 and counter= 7
iteration=15 and counter= 8
iteration=16 and counter= 9

```

Global static variables are accessible throughout the file whereas local static variables are accessible only to the particular part of a code.

```

#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}

```

## OUTPUT

```

0 0 0.000000 (null)

```

## 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 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 <stdio.h>
int main()
{
    register int a; // variable a is allocated memory in the CPU register.
    printf("%d",a);

}
```

## OUTPUT

0

```
#include <stdio.h>
int main()
{
    register int a = 0;
    printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
}
```

## OUTPUT

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:13:5: error: address of register variable 'a' requested
    printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
    ^~~~~~
```

## A C program to demonstrate different storage classes

```
#include <stdio.h>

// declaring the variable which is to be made extern
// an initial value can also be initialized to x
int x;

void autoStorageClass()
{

    printf("\nDemonstrating auto class\n\n");

    // declaring an auto variable (simply
    // writing "int a=32;" works as well)
    auto int a = 32;

    // printing the auto variable 'a'
    printf("Value of the variable 'a'")
```

```

        " declared as auto: %d\n",
        a);

    printf("-----");
}

void registerStorageClass()
{
    printf("\nDemonstrating register class\n\n");

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

    // printing the register variable 'b'
    printf("Value of the variable 'b'"
        " declared as register: %d\n",
        b);

    printf("-----");
}

void externStorageClass()
{
    printf("\nDemonstrating extern class\n\n");

    // telling the compiler that the variable
    // z is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;

    // printing the extern variables 'x'
    printf("Value of the variable 'x'"
        " declared as extern: %d\n",
        x);

    // value of extern variable x modified
    x = 2;

    // printing the modified values of
    // extern variables 'x'
    printf("Modified value of the variable 'x'"
        " declared as extern: %d\n",
        x);

    printf("-----");
}

void staticStorageClass()
{
    int i = 0;

```

```

printf("\nDemonstrating static class\n\n");

// using a static variable 'y'
printf("Declaring 'y' as static inside the loop.\n"
      "But this declaration will occur only"
      " once as 'y' is static.\n"
      "If not, then every time the value of 'y' "
      "will be the declared value 5"
      " as in the case of variable 'p'\n");

printf("\nLoop started:\n");

for (i = 1; i < 5; i++) {

    // Declaring the static variable 'y'
    static int y = 5;

    // Declare a non-static variable 'p'
    int p = 10;

    // Incrementing the value of y and p by 1
    y++;
    p++;

    // printing value of y at each iteration
    printf("\nThe value of 'y', "
          "declared as static, in %d "
          "iteration is %d\n",
          i, y);

    // printing value of p at each iteration
    printf("The value of non-static variable 'p', "
          "in %d iteration is %d\n",
          i, p);
}

printf("\nLoop ended:\n");

printf("-----");
}

int main()
{

    printf("A program to demonstrate"
          " Storage Classes in C\n\n");

    // To demonstrate auto Storage Class
    autoStorageClass();

    // To demonstrate register Storage Class
    registerStorageClass();
}

```



```

// To demonstrate extern Storage Class
externStorageClass();

// To demonstrate static Storage Class
staticStorageClass();

// exiting
printf("\n\nStorage Classes demonstrated");

return 0;
}

```

## OUTPUT

A program to demonstrate Storage Classes in C

Demonstrating auto class

Value of the variable 'a' declared as auto: 32

---

Demonstrating register class

Value of the variable 'b' declared as register: 71

---

Demonstrating extern class

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

Modified value of the variable 'x' declared as extern: 2

---

Demonstrating static class

Declaring 'y' as static inside the loop.

But this declaration will occur only once as 'y' is static.

If not, then every time the value of 'y' will be the declared value 5 as in the case of variable 'p'

Loop started:

The value of 'y', declared as static, in 1 iteration is 6

The value of non-static variable 'p', in 1 iteration is 11

The value of 'y', declared as static, in 2 iteration is 7

The value of non-static variable 'p', in 2 iteration is 11

The value of 'y', declared as static, in 3 iteration is 8

The value of non-static variable 'p', in 3 iteration is 11

The value of 'y', declared as static, in 4 iteration is 9  
The value of non-static variable 'p', in 4 iteration is 11

Loop ended:

---

Storage Classes demonstrated