STORAGE CLASSES

1

- To fully define a variable one needs to mention not only its ‘type’ but also its ‘storage class’.
- There are basically two kinds of locations in a computer where the value of a variable may be stored: Memory and CPU Registers.
- It is the variable’s storage class which determines in which of the two locations the value will be stored.

2

A variable's storage class tells :

- Where the variable would be stored.
- What will be the initial value of the variable, if the initial value is not specifically defined.
- What is the scope of the variable; i.e. in which parts of the program the value of the variable would be available.
- What is the life of the variable; i.e. the period during which a variable retains a given value during execution of the program.

3

There are four storage classes in C:

1. **Automatic**
2. **Register**
3. **Static**
4. **External**

4

Automatic Storage Class

- The features of a variable defined to have an **auto** storage class are as under:
- Storage – Memory
- Default Value – Garbage
- Scope – Local to the block in which the variable is defined.
- Life – Till the control remains within the block in which the variable is defined.

5

```
main()
{
   auto int i=1;
   {
       {
              {
                     printf("%d\n",i);
              }
               printf("%d\n",i);
       }
        printf("%d\n",i);
   }
}
```

OUTPUT

1

1

1

6

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

OUTPUT

3

2

1

7

- One important feature of **auto** variables is that their values can not be changed accidentally by what happens in some other function in the program.
- This assures that we may declare and use the same variable name in different functions in the same program.

8

Register Storage Class

- The features of a variable defined to have a **register** storage class are as under:
- Storage – CPU Registers
- Default Value – Garbage
- Scope – Local to the block in which the variable is defined.
- Life – Till the control remains within the block in which the variable is defined.

9

- A value stored in a CPU Register can always be accessed faster than the one which is stored in memory.
- If a variable is used at many places in a program, it is better to declare its storage class as register.
- A good example of frequently used variables are loop counters.

10

```
main()
{
   register int i;
   for(i=0;i<10;i++)
        printf("%d\n",i);
// printf("Address of i=%u",&i); // Gives error
}
```

- Only variables stored in memory have addresses.
- It's not always necessary that a variable with register storage class is stored in a CPU register, because the number of CPU registers are very limited and they might be busy doing some other task. In that case the variable works as if its storage class is auto.

11

- We can not use **register** storage class for all types of variables. Following declarations are wrong.

  register float f;

  register double d;

  register long a;

  This is because the CPU registers are usually 16 bit registers therefore can not hold float, long or double value.

  However, no error message will be displayed. The compiler would treat them as **auto**

12

Static Storage Class

- The features of a variable defined to have a **static** storage class are as under:
- Storage – Memory
- Default Value – **Zero**
- Scope – Local to the block in which the variable is defined.
- Life – Value of the variable **persists** between different function calls.

13

```
main()
{
        increment();
        increment();
        increment();
}
increment()
{
        auto int i =1;
        printf("%d\n",i);
        i = i+1;
}
```

Output:

1
1
1

```
main()
{
        increment();
        increment();
        increment();
}
increment()
{
        static int i =1;
        printf("%d\n",i);
        i = i+1;
}
```

Output:

1
2
3

14

- **static** variables do not disappear when the function is no longer active. Their values persist.
- If the control comes back to the same function again the **static** variables have the same values they had last time around.
- When the variable i is **auto**, each time increment() is called it is re-initialised to 1.
- On the other hand, if i is **static**, it is initialised to 1 only once.It is never initialised again.

15

- Avoid using **static** variables unless you really need them.
- Because their values are kept in memory when the variables are not active, which means they take up space in memory that could otherwise be used by other variables.

16

External Storage Class

- The features of a variable defined to have an **external** storage class are as under:
- Storage – Memory
- Default Value – Zero
- Scope – Global.
- Life – As long as the program execution doesn't come to an end.

17

```
#include<stdio.h>
int x;
main()
{
        int fun1(),fun2(),fun3();
        x=10;
        printf("x=%d\n",x);
        printf("x=%d\n",fun1());
        printf("x=%d\n",fun2());
        printf("x=%d\n",fun3());
}
fun1()
{
        x=x+10;
        return(x);
}
```

```
fun2()
{
        int x;
        x=1;
        return(x);
}
fun3()
{
        x=x+10;
        return(x);
}
```

Output:

X=10

X=20

X=1

X=30

18

- Once a variable has been declared as global, any function can use it and change its value.
- Then subsequent functions can reference only that new value.
- In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared.

19

- One other aspect of a global variable is that it is visible only from the point of declaration to the end of the program.

```
main()
{
        y=5;
        ----
        ----
}
int y;
fun1()
{
        y=y+1;
}
```

- As far as main is concerned, y is not defined. So the compiler will issue an error.
- Unlike local variables, global variables are initialized to 0 by default.
- The statement y=y+1 in fun1() will therefore assign 1 to y.

20

```
main()
{
        extern int y;
        ----
        ----
}

fun1()
{
        extern int y;
        ----
        ----
}
int y;
```

- In the above program, main() cannot access the variable y, as it is declared after the main function.
- This problem can be solved by declaring the variable with the storage class **extern**.
- The external declaration of y inside the functions informs the compiler that y is an integer type defined somewhere else in the program.
- **extern** declaration doesn't allocate storage space for variables.

21

In case of arrays, the definition should include their size as well.

```
main()
{
        int i;
        void print_out();
        extern float height[];
        ----
        ----
        print_out();
}

print_out()
{
        extern float height[];
        int i;
        ----
        ----
}
float height[20];
```

22

• An **extern** within a function provides the type information to just that one function.

• We can provide type information to all functions within a file by placing external declaration before them.

• Functions are external by default .

•Therefore the declaration:

void print_out(); is equivalent to

extern void print_out();

•That's why even if we define a function after main(), it is accessible in main if we have given the function prototype even without extern keyword.

```
extern float height[];
main()
{
        int i;
        void print_out();
        ----
        ----
        print_out();
}

print_out()
{
        int i;
        ----
        ----
}
float height[20];
```

23

Multifile Programs

- In real-life programming environment, we use more than one source files which may be compiled separately and linked later to form an executable code.
- This approach is useful because any change in one file doesn't affect other files thus eliminating the need for recompilation of the entire program.
- Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and then explicitly define them with **extern** in other files.

24

// file1.c

```
extern int m;
void fun1();
extern void fun2();
extern void fun3();
main()
{
    fun1();  fun2();  fun3();
}
void fun1()
{
     printf("\nIn fun1(): m=%d",m);
}
```

25

// file2.c

```
int m;
void fun2()
{
    m=m+1;
    printf("\nIn fun2():m=%d",m);

}
void fun3()
{
    m=m+1;
    printf("\nIn fun3():m=%d",m);
}
```

26

OUTPUT:

In fun1():m=0
In fun2():m=1
In fun3():m=2Press any key to continue

27

- The **extern** specifier tells the compiler that the following variable types and names have already been declared elsewhere and no need to create storage space for them.
- It is the responsibility of the linker to resolve the reference problem.
- It is important to note that a multifile global variable should be declared without **extern** in one (and only one) of the files.
- The **extern** declaration is done in places where secondary references are made.

28

- When a function is defined in one file and accessed in another, the later file must include a function declaration.
- The declaration identifies the function as an external function whose definition appears elsewhere.
- Although all functions are assumed to be external, it would be a good practice to explicitly declare such functions with the storage class **extern** .

29

Point to Remember

- The difference between a **static** external variable and a simple external variable is that the **static** external variable is available only within the file where it is defined while the simple external variable can be accessed by other files also.
- If in the previous example we write **static int m;** in file2.c instead of **int m;** then the program will not run.

30

- We can use the storage classes to:
  - Economize the memory space consumed by the variables
  - Improve the speed of execution of the program.

The rules are:

- Use **static** storage class only if you want the value of a variable to persist between different function calls. A typical application of this storage class is recursive functions.

31

- Use **register** storage class for those variables which are being used very often in a program. e.g. loop counters, which get executed a number of times in a program.
- Use **extern** for those variables which are being used by almost all the functions in the program.
- Declaring all the variables as **extern** would amount to a lot of memory wastage because these variables remain active throughout the life of the program.
- If you don't have any specific need, then use **auto** storage class.

32