Unit 6 Storage Classes

Structure:

6.1 Introduction

In the previous unit, you studied about functions. You found out that how functions can be used to break the large problems into small problems and then solve it. You studied how functions can be repeatedly called and used again and again. In this unit, you will study about the types of storage classes that are used in C. You will study how these storage classes are useful in making the C language a very powerful computing language.

Variables are channels of communication within a program. You set a variable to a value at one point in a program, and at another point (or points) you read the value out again. The two points may be in adjoining statements, or they may be in widely separated parts of the program. How long does a variable last? How widely separated can the setting and fetching parts of the program be, and how long after a variable is set does it persist? Depending on the variable and how you're using it, you might want different answers to these questions. For example, in some situations it may be desirable to introduce certain “global” variables that are recognized throughout the entire program (or within major portions of the program, e.g. two or more functions). Such variables are defined differently than the usual “local” variables, which are recognized only within a single function.

We will also consider the issue of static variables which can retain their values, so that the function can be reentered later and the computation resumed.

Finally, we may be required to develop a large, multifunction program in terms of several independent files, with few functions defined within each file. In such programs, the individual functions can be defined and accessed locally within a single file, or globally within multiple files.

Objectives:
After studying this unit, you should be able to:

- implement the concept of storage classes and visibility of variables
- explain the difference between automatic variables, global variables, static variables and external variables.
- compile and execute a program made up of more than one source files.

6.2 Storage Classes and Visibility

There are two ways to categorize variables: by *data type*, and by *storage class*. Data type refers to the type of information represented by a variable, for example, integer number, floating-point number, character etc. Storage class refers to the persistence of a variable and its scope within the program, that is, the portion of the program over which the variable is recognized.

The following types of storage-class specifications in C are discussed in this unit: **global**, **automatic** or **local**, **static,** and **extern**. The exact procedure for establishing a storage class for a variable depends upon the particular storage class, and the manner in which the program is organized, (i.e. single file vs. multiple file).

The *visibility* of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program.

Why would you want to limit the visibility of a variable? For maximum flexibility, wouldn't it be handy if all variables were potentially visible everywhere? As it happens, that arrangement would be *too* flexible: everywhere in a program, you would have to keep track of the names of all the variables declared anywhere else in the program, so that you didn't

accidentally re-use one. Whenever a variable had the wrong value by mistake, you'd have to search the entire program for the bug, because any statement in the entire program could potentially have modified that variable. You would constantly be stepping all over yourself by using a common variable name like i in two parts of your program, and having one snippet of code accidentally overwrite the values being used by another part of the code.

Self Assessment Questions

1. The visibility of a variable determines how much of the rest of the program can access that variable. (True/False)
2. _____ class refers to the persistence of a variable and its scope within the program, that is, the portion of the program over which the variable is recognized.
3. Visibility provides security for your data used in a program. (True/False)

6.3 Automatic or local variables

A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called *local variables*. Their scope is confined to that function. You can use the keyword **auto** to declare automatic variables, but, however it is optional. If another function somewhere else declares a local variable with the same name, it's a different variable entirely, and the two don't clash with each other. If an automatic variable is not initialized in some manner, however, its initial value will be unpredictable and contains some garbage value.

Program 6.1: Program to find factorial of a number

```
#include<stdio.h>
main()
{
        auto int n;       /* Here the keyword auto is optional */
        long int fact(int);
        printf("read the integer n:");
        scanf("%d", &n);
        printf("\nn!=%ld", fact(n) );
}
long int fact(auto int n) /* n is local to the function fact() and auto is optional*/
```

```
{
        auto int i;        /* Here the keyword auto is optional */
        auto long int factorial=1;        /* Here the keyword auto is optional */
        while(n>0)
        {
                factorial=factorial*n;
                n=n-1;
        }
        return factorial;
}
```

An automatic variable doesn't retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited.

Self Assessment Questions

4. The scope of an automatic variable is in _____ in which it is declared.
5. Does an automatic variable retain its value once control is transferred out of its defining function? (Yes/No)
6. The key word **auto** is must in the declaration of automatic variables. (True/False)

6.4 Global Variables

A variable declared outside of any function is a *global variable*, and it is potentially visible anywhere within the program. You use global variables when you *do* want to use the variable in any part of the program. When you declare a global variable, you will usually give it a longer, more descriptive name (not something generic like i) so that whenever you use it you will remember that it's the same variable everywhere. The values stored in global variables persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.)

Program 6.2: Program to find average length of several lines of text

```
#include<stdio.h>
/* Declare global variables outside of all the functions*/
int sum=0;      /* total number of characters */
int lines=0;    /* total number of lines */
main()
{
        int n;   /* number of characters in given line */
        float avg;       /* average number of characters per line */
        void linecount(void);   /* function declaraction */
        float cal_avg(void);
        printf("Enter the text below:\n");
        while((n=linecount())>0) {
                sum+=n;
                ++lines;
        }

        avg=cal_avg();
        printf("\nAverage number of characters per line: %5.2f", avg);
}
 void linecount(void)
{
/* read  a line of text and count the number of characters */
        char line[80];
        int count=0;
        while((line[count]=getchar())!='\n')
                ++count;
        return count;
}
float cal_avg(void)
{
/* compute average and return*/
        return (float)sum/lines;
}
```

In the above program the variables sum and lines are globally declared and hence they could be used in both the functions main() and cal_avg()

Self Assessment Questions

7. The variables declared in the main() function are the global variables. (True/False)
8. The global variables are more secured than the automatic variables in a program. (True/False)

6.5 Static Variables

Static variables are defined within individual functions and therefore have the same scope as automatic variables, i.e. they are local to the functions in which they are declared. Unlike automatic variables, however, static variables retain their values throughout the life of the program. As a result, if a function is exited and then reentered later, the static variables defined within that function will retain their previous values. This feature allows functions to retain information permanently throughout the execution of a program. Static variables can be utilized within the function in the same manner as other variables. They cannot be accessed outside of their defining function.

In order to declare a static variable the keyword **static** is used as shown below:

```
static int count;
```

You can define automatic or static variables having the same name as global variables. In such situations the local variables will take precedence over the global variables, though the values of global variables will be unaffected by any manipulation of the local variables.

Initial values can be included in static variable declaration. The rules associated with the initialization remain same as the initialization of automatic or global variables. They are:

1. The initial values must be constants, not expressions.
2. The initial values are assigned to their respective variables at the beginning of the program execution. The variables retain these values throughout the life of the program, unless different values are assigned during the course of computation.
3. Zeros will be assigned to all static variables whose declarations do not include explicit initial values.

Program 6.3: Program to generate Fibonacci numbers.

```
#include<stdio.h>
main()
{
        int count, n;
        long int fib(int);
        printf("\n How many Fibonacci numbers?");
        scanf("%d\n", &n);
        for(count=1;count<=n;count++)
        {
                printf("\ni=%d        F=%ld", count, fib(count));
        }

        long int fib(int count)
        {
        /* calculate a Fibonacci number using the formula
        if i=1, F=0; if i=2, F=1, and F=F1+F2 for i>=3 */
        static long int f1=0, f2=1;        /* declaration of static variables */
        long int f;
        if (count==1)
                f=0;
        else if (count==2)
                f=1;
        else
                f=f1+f2;
        f2=f1;
        f1=f;       /* f1 and f2 retain their values between different calls of the
function*/
        return f;
}
```

Self Assessment Questions

9. The scope of static variables and automatic variables is the same. (True/False)
10. _____ variables retain their values throughout the life of the program. As a result, if a function is exited and then reentered later, the static variables defined within that function will retain their previous values.
11. By default, a static variable is initialized to _____.

6.6 External Variables

It is possible to split a function up into several source files, for easier maintenance. When several source files are combined into one program the compiler must have a way of correlating the variables which might be used to communicate between the several source files. Furthermore, if a variable is going to be useful for communication, there must be exactly one of it: you wouldn't want one function in one source file to store a value in one variable named **externvar**, and then have another function in another source file read from a *different* variable named **externvar**. Therefore, a variable should have exactly one *defining instance*, in one place in one source file. If the same variable is to be used anywhere else (i.e. in some other source file or files), the variable is declared in those other file(s) with an *external declaration*, which is not a defining instance. The external declaration says the compiler that the variable will be used in this source file but defined in some other source file. Thus the compiler doesn't allocate space for that variable with this source file.

To make a variable as an external declaration, which is defined somewhere else; you precede it with the keyword **extern**:

```
extern int j;
```

Program 6.4: Program to illustrate the concept of external variables.

Type and save the following program in a source file called **externvariables.h**

```
int principle=10000;
float rate=5.5;
int time=2;
float interest;
```

Type and save the following program in a separate source file called **demoexternvar.c**

```
#include<stdio.h>
#include “externvariables.h”  /* the source file where the external variables are defined should be included here.*/

main()
```

```
{
        /* external declarations of the variables which are defined in
externvariables.h */
        extern int principle;
        extern float rate;
        extern int time;
        extern float interest;
        /*compute interest*/
        interest= principle*rate*time/100.0;
        printf("Interest=%f\n", interest);
}
```

Compile demoexternvar.c and execute the program.

The concept of external storage class can be extended to functions also. A source file can access a function defined in any other source file provided the source file is included within the source file where you access that function.

Program 6.5: Program to illustrate the concept of external functions.
Type and save the following program in a file externfunction.h

```
void output(void)
{
        printf(" Hi, Manipal!\n");
        return;
}
```

Type and save the following program in a separate source file called **demoexternfun.c**

```
#include<stdio.h>
#include " externfunction.h"
extern void output(void);
main()
{
        output();
}
```

Compile and execute the above program and observe the result.

However, the keyword **extern** is optional in some C compilers.

Self Assessment Questions

12. The main purpose of using external variables is to access the same variable in different _____ files.
13. Compiler doesn't allocate memory for an external variable where it is accessed. (True/False)
14. Global variables and external variables have the same scope. (True/False)

Example 6.1: Here is an example demonstrating almost everything we've seen so far:

```
int globalvar = 1;
extern int anotherglobalvar;
static int privatevar;
f()
{
        int localvar;
        int localvar2 = 2;
        static int persistentvar;
}
```

Here we have six variables, three declared outside and three declared inside of the function f().

globalvar is a global variable. The declaration we see is its defining instance (it happens also to include an initial value). **globalvar** can be used anywhere in this source file, and it could be used in other source files, too (as long as corresponding external declarations are issued in those other source files).

anotherglobalvar is a second global variable. It is *not* defined here; the defining instance for it (and its initialization) is somewhere else.

privatevar is a "private" global variable. It can be used anywhere within this source file, but functions in other source files cannot access it, even if they try to issue external declarations for it. (If other source files try to declare a global variable called **''privatevar"**, they'll get their own; they won't be sharing this one.) Since it has static duration and receives no explicit initialization, privatevar will be initialized to 0.

localvar is a local variable within the function f(). It can be accessed only within the function f(). (If any other part of the program declares a variable named **“localvar"**, that variable will be distinct from the one we're looking at here.) **localvar** is conceptually “created" each time f() is called, and disappears when f() returns. Any value which was stored in **localvar** last time f() was running will be lost and will not be available next time f() is called. Furthermore, since it has no explicit initializer, the value of **localvar** will in general be garbage each time f() is called.

localvar2 is also local, and everything that we said about **localvar** applies to it, except that since its declaration includes an explicit initializer, it will be initialized to 2 each time f() is called.

Finally, **persistentvar** is again local to f(), but it *does* maintain its value between calls to f(). It has static duration but no explicit initializer, so its initial value will be 0.

The term *declaration* is a general one which encompasses defining instances and external declarations; defining instances and external declarations are two different kinds of declarations. Furthermore, either kind of declaration suffices to inform the compiler of the name and type of a particular variable (or function). If you have the defining instance of a global variable in a source file, the rest of that source file can use that variable without having to issue any external declarations. It's only in source files where the defining instance hasn't been seen that you need external declarations.

You will sometimes hear a defining instance referred to simply as a “definition," and you will sometimes hear an external declaration referred to simply as a “declaration." These usages are mildly ambiguous, in that you can't tell out of context whether a “declaration" is a generic declaration (that might be a defining instance or an external declaration) or whether it's an external declaration that specifically is not a defining instance. Similarly, there are other constructions that can be called “definitions" in C, namely the definitions of preprocessor macros, structures, and typedefs etc.

Program 6.6: Program to illustrate the hiding of variables in blocks

```
/* hiding.c -- variables in blocks */
#include <stdio.h>
int main()
{
   int x = 30;
   printf("x in outer block: %d\n", x);
   {
      int x = 77;  /* new x, hides first x */
      printf("x in inner block: %d\n", x);
   }
   printf("x in outer block: %d\n", x);
   while (x++ < 33)
   {
      int x = 100; /* new x, hides first x */
      x++;
      printf("x in while loop: %d\n", x);
   }
   return 0;
}
```

6.7 Summary

Variables are channels of communication within a program. Storage class refers to the persistence of a variable and its scope within the program, that is, the portion of the program over which the variable is recognized. The scope of a local or automatic variable is confined to the function where it is defined. A global variable is potentially visible anywhere within the program in which it is defined. Static variables retain their values throughout the life of the program. As a result, if a function is exited and then reentered later, the static variables defined within that function will retain their previous values. The external variable declaration says the compiler that the global variable will be used in this source file but defined in some other source file.

6.8 Terminal Questions

1. List some of the storage classes available in C.
2. What is the use of header file? Is the use of header file absolutely necessary?

3. What is the difference between declaration and definition of function?
4. What is the significance of external declaration?
5. How can you justify that variables are channels of communication in a program?

6.9 Answers to Self Assessment Questions

1. True
2. Storage
3. True
4. The function in which it is declared.
5. No
6. False
7. False
8. False
9. True
10. Static.
11. Zero
12. source
13. True
14. False

6.9 Answers for Terminal Questions

1. automatic, global, static, extern
2. Header files are used to define some variables and functions separately in a library. Built-in header files are absolutely necessary if you want to access the variables and functions defined in them.
3. Declaration is nothing but the prototype that contains the type of returned data, name of the function and type of the arguments. But the definition contains the function header and the body of the function.
4. The external declaration says the compiler that the variable will be used in this source file but defined in some other source file.
5. You set a variable to a value at one point in a program, and at another point (or points) you read the value out again. Thus the transfer of information from one point of the program to another is nothing but the communication.

6.10 Exercises

1. Distinguish between the following
   i. Global and local variables
   ii. Automatic and static variables
   iii. Global and extern variables
2. Write a program to count the number of times a function is called using static variables.
3. Write a function prime that returns 1 if its argument is a prime number and returns zero Otherwise.
4. Write a function that will round a floating point number to an indicated decimal place. For example, the number 12.456 would yield the value 12. 46 when it is rounded off to two decimal places.
5. Write a program to illustrate the concept of extern variables.