also true. If we exceed the range from negative side we end up on positive side.

Storage Classes in C

We have already said all that needs to be said about constants, but we are not finished with variables. To fully define a variable one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a 'storage class'.

We have not mentioned storage classes yet, though we have written several programs in C. We were able to get away with this because storage classes have defaults. If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Thus, variables have certain default storage classes.

From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept—Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

Moreover, a variable's storage class tells us:

(a) Where the variable would be stored.
(b) What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
(c) What is the scope of the variable; i.e. in which functions the value of the variable would be available.
(d) What is the life of the variable; i.e. how long would the variable exist.

There are four storage classes in C:

(a) Automatic storage class
(b) Register storage class
(c) Static storage class
(d) External storage class

Let us examine these storage classes one by one.

Automatic Storage Class

The features of a variable defined to have an automatic storage class are as under:

Storage	–	Memory.
Default initial value	–	An unpredictable value, which is often called a garbage value.
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.

Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

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

The output of the above program could be...

```
1211  221
```

where, 1211 and 221 are garbage values of **i** and **j**. When you run this program you may get different values, since garbage values

are unpredictable. So always make it a point that you initialize the automatic variables properly, otherwise you are likely to get unexpected results. Note that the keyword for this storage class is **auto**, and not automatic.

Scope and life of an automatic variable is illustrated in the following program.

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

The output of the above program is:

```
1 1 1
```

This is because, all **printf( )** statements occur within the outermost block (a block is all statements enclosed within a pair of braces) in which **i** has been defined. It means the scope of **i** is local to the block in which it is defined. The moment the control comes out of the block in which the variable is defined, the variable and its value is irretrievably lost. To catch my point, go through the following program.

```
main( )
{
    auto int  i = 1 ;
    {
```

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

The output of the above program would be:

```
3 2 1
```

Note that the Compiler treats the three **i**'s as totally different variables, since they are defined in different blocks. Once the control comes out of the innermost block the variable **i** with value 3 is lost, and hence the **i** in the second **printf( )** refers to **i** with value 2. Similarly, when the control comes out of the next innermost block, the third **printf( )** refers to the **i** with value 1.

Understand the concept of life and scope of an automatic storage class variable thoroughly before proceeding with the next storage class.

Register Storage Class

The features of a variable defined to be of **register** storage class are as under:

Storage	-	CPU registers.
Default initial value	-	Garbage value.
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.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, 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 is loop counters. We can name their storage class as **register**.

```
main( )
{
      register int  i ;

      for ( i = 1 ; i <= 10 ; i++ )
            printf ( "\n%d", i ) ;
}
```

Here, even though we have declared the storage class of **i** as **register**, we cannot say for sure that the value of **i** would be stored in a CPU register. Why? Because the number of CPU registers are limited, and they may be busy doing some other task. What happens in such an event... the variable works as if its storage class is **auto**.

Not every type of variable can be stored in a CPU register.

For example, if the microprocessor has 16-bit registers then they cannot hold a **float** value or a **double** value, which require 4 and 8 bytes respectively. However, if you use the **register** storage class for a **float** or a **double** variable you won't get any error messages. All that would happen is the compiler would treat the variables to be of **auto** storage class.

Static Storage Class

The features of a variable defined to have a **static** storage class are as under:

Storage – Memory.
Default initial value – Zero.

Scope	–	Local to the block in which the variable is defined.
Life	–	Value of the variable persists between different function calls.

Compare the two programs and their output given in Figure 6.3 to understand the difference between the **automatic** and **static** storage classes.

```
main( )                          main( )
{                                {
    increment( ) ;                   increment( ) ;
    increment( ) ;                   increment( ) ;
    increment( ) ;                   increment( ) ;
}                                }

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

The output of the above programs would be:

1	1
1	2
1	3

Figure 6.3

The programs above consist of two functions **main( )** and **increment( )**. The function **increment( )** gets called from **main( )** thrice. Each time it increments the value of **i** and prints it. The only difference in the two programs is that one uses an **auto** storage class for variable **i**, whereas the other uses **static** storage class.

Like **auto** variables, **static** variables are also local to the block in which they are declared. The difference between them is that **static** variables don't 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.

In the above example, when variable **i** is **auto**, each time **increment( )** is called it is re-initialized to one. When the function terminates, **i** vanishes and its new value of 2 is lost. The result: no matter how many times we call **increment( ), i** is initialized to 1 every time.

On the other hand, if **i** is **static**, it is initialized to 1 only once. It is never initialized again. During the first call to **increment( )**, **i** is incremented to 2. Because **i** is static, this value persists. The next time **increment( )** is called, **i** is not re-initialized to 1; on the contrary its old value 2 is still available. This current value of **i** (i.e. 2) gets printed and then **i = i + 1** adds 1 to **i** to get a value of 3. When **increment( )** is called the third time, the current value of **i** (i.e. 3) gets printed and once again **i** is incremented. In short, if the storage class is **static** then the statement **static int i = 1** is executed only once, irrespective of how many times the same function is called.

Consider one more program.

```
main( )
{
      int  *j ;
      int * fun( ) ;
      j = fun( ) ;
      printf ( "\n%d", *j ) ;
}

int *fun( )
{
```

```
    int k = 35 ;
    return ( &k ) ;
}
```

Here we are returning an address of **k** from **fun( )** and collecting it in **j**. Thus **j** becomes pointer to **k**. Then using this pointer we are printing the value of **k**. This correctly prints out 35. Now try calling any function (even **printf( )**) immediately after the call to **fun( )**. This time **printf( )** prints a garbage value. Why does this happen? In the first case, when the control returned from **fun( )** though **k** went dead it was still left on the stack. We then accessed this value using its address that was collected in **j**. But when we precede the call to **printf( )** by a call to any other function, the stack is now changed, hence we get the garbage value. If we want to get the correct value each time then we must declare **k** as **static**. By doing this when the control returns from **fun( )**, **k** would not die.

All this having been said, a word of advice—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.

External Storage Class

The features of a variable whose storage class has been defined as external are as follows:

Storage	–	Memory.
Default initial value	–	Zero.
Scope	–	Global.
Life	–	As long as the program's execution doesn't come to an end.

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

```
int  i ;
main( )
{
      printf ( "\ni = %d", i ) ;

      increment( ) ;
      increment( ) ;
      decrement( ) ;
      decrement( ) ;
}

increment( )
{
      i = i + 1 ;
      printf ( "\non incrementing i = %d", i ) ;
}

decrement( )
{
      i = i - 1 ;
      printf ( "\non decrementing i = %d", i ) ;
}
```

The output would be:

```
i = 0
on incrementing i = 1
on incrementing i = 2
on decrementing i = 1
on decrementing i = 0
```

As is obvious from the above output, the value of **i** is available to the functions **increment( )** and **decrement( )** since **i** has been declared outside all functions.

Look at the following program.

```
int  x = 21 ;
main( )
{
      extern int  y ;
      printf ( "\n%d %d", x, y ) ;
}
int y = 31 ;
```

Here, **x** and **y** both are global variables. Since both of them have been defined outside all the functions both enjoy external storage class. Note the difference between the following:

```
extern int  y ;
int y = 31 ;
```

Here the first statement is a declaration, whereas the second is the definition. When we declare a variable no space is reserved for it, whereas, when we define it space gets reserved for it in memory. We had to declare **y** since it is being used in **printf( )** before it's definition is encountered. There was no need to declare **x** since its definition is done before its usage. Also remember that a variable can be declared several times but can be defined only once.

Another small issue—what will be the output of the following program?

```
int  x = 10 ;
main( )
{
      int  x = 20 ;

      printf ( "\n%d", x ) ;
```

```
    display( ) ;
}
display( )
{
    printf ( "\n%d", x ) ;
}
```

Here **x** is defined at two places, once outside **main( )** and once inside it. When the control reaches the **printf( )** in **main( )** which **x** gets printed? Whenever such a conflict arises, it's the local variable that gets preference over the global variable. Hence the **printf( )** outputs 20. When **display( )** is called and control reaches the **printf( )** there is no such conflict. Hence this time the value of the global **x**, i.e. 10 gets printed.

One last thing—a **static** variable can also be declared outside all the functions. For all practical purposes it will be treated as an **extern** variable. However, the scope of this variable is limited to the same file in which it is declared. This means that the variable would not be available to any function that is defined in a file other than the file in which the variable is defined.

Which to Use When

Dennis Ritchie has made available to the C programmer a number of storage classes with varying features, believing that the programmer is in a best position to decide which one of these storage classes is to be used when. We can make a few ground rules for usage of different storage classes in different programming situations with a view to:

(a) economise the memory space consumed by the variables
(b) improve the speed of execution of the program

The rules are as under:

- Use **static** storage class only if you want the value of a variable to persist between different function calls.
- Use **register** storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of **register** storage class is loop counters, which get used a number of times in a program.
- Use **extern** storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as **extern** would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.
- If you don't have any of the express needs mentioned above, then use the **auto** storage class. In fact most of the times we end up using the **auto** variables, because often it so happens that once we have used the variables in a function we don't mind loosing them.

Summary

(a) We can use different variations of the primary data types, namely **signed** and **unsigned char**, **long** and **short int**, **float**, **double** and **long double**. There are different format specifications for all these data types when they are used in **scanf( )** and **printf( )** functions.

(b) The maximum value a variable can hold depends upon the number of bytes it occupies in memory.

(c) By default all the variables are **signed**. We can declare a variable as **unsigned** to accommodate greater value without increasing the bytes occupied.

(d) We can make use of proper storage classes like **auto**, **register**, **static** and **extern** to control four properties of the variable—storage, default initial value, scope and life.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    int  i ;
    for ( i = 0 ; i <= 50000 ; i++ )
          printf ( "\n%d", i ) ;
}
```

(b)
```
main( )
{
    float  a = 13.5 ;
    double  b = 13.5 ;
    printf ( "\n%f %lf", a, b ) ;
}
```

(c)
```
int  i = 0 ;
main( )
{
    printf ( "\nmain's i = %d", i ) ;
    i++ ;
    val( ) ;
    printf ( "\nmain's i = %d", i ) ;
    val( ) ;
}
val( )
{
    i = 100 ;
    printf ( "\nval's i = %d", i ) ;
    i++ ;
}
```