Data Types Revisited

Integers, long and short

- For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767
- For a 32-bit compiler the range would be –2147483648 to +2147483647
- the highest bit (16th/32nd bit) is used to store the sign of the integer
- This bit is 1 if the number is negative, and 0 if the number is positive
- C offers a variation of the integer data type that provides what are called short and long integer values
- Though not a rule, short and long integers would usually occupy two and four bytes respectively

Integers, long and short

- Each compiler can decide appropriate sizes depending on the operating system and hardware for which it is being written, subject to the following rules:
- shorts are at least 2 bytes big
- longs are at least 4 bytes big
- shorts are never bigger than ints
- ints are never bigger than longs

Compiler	short	int	long
16-bit (Turbo C/C++)	2	2	4
32-bit (Visual C++)	2	4	4

Long integers

- long variables which hold long integers are declared using the keyword long, as in,
- long int i ;
- long int abc ;
- long integers cause the program to run a bit slower, but the range of values that we can use is expanded tremendously
- The value of a long integer typically can vary from -2147483648 to +2147483647

Short integers

- shorts—integers that need less space in memory and thus help speed up program execution.
- short integer variables are declared as,
- short int j ;
- short int height ;
- C allows the abbreviation of short int to short and of long int to long
- So the declarations made above can be written as
- long i ;
- long abc ;
- short j ;
- short height ;

Adding suffix l or L

- Sometimes we come across situations where the constant is small enough to be an int, but still we want to give it as much storage as a long
- In such cases we add the suffix ‘L’ or ‘l’ at the end of the number, as in 23L

Integers, signed and unsigned

- Sometimes, we know in advance that the value stored in a given integer variable will always be positive
- when it is being used to only count things, for example.
- In such a case we can declare the variable to be unsigned, as in,
- unsigned int num_students ;
- declaring an integer as unsigned almost doubles the size of the largest possible value that it can otherwise take
- because on declaring the integer as unsigned, the left-most bit is now free and is not used to store the sign of the number

Integers, signed and unsigned

- Note that an unsigned integer still occupies two bytes.
- This is how an unsigned integer can be declared:
- unsigned int i ;
- unsigned i ;
- Like an unsigned int, there also exists a short unsigned int and a long unsigned int.
- By default a short int is a signed short int and a long int is a signed long int.

Chars, signed and unsigned

- Parallel to signed and unsigned ints (either short or long), similarly there also exist signed and unsigned chars, both occupying one byte each, but having different ranges
- Consider the statement
- char ch = 'A’ ;
- Here what gets stored in ch is the binary equivalent of the ASCII value of ‘A’ (i.e. binary of 65)
- And if 65’s binary can be stored, then -54’s binary can also be stored (in a signed char)
- A signed char is same as an ordinary char and has a range from -128 to +127; whereas, an unsigned char has a range from 0 to 255.

Chars, signed and unsigned

- This will not print the character equivalent of 291 because the range of char is only upto +127.
- Second one is an infinite loop

```
main( )
{
    char  ch = 291 ;
    printf ( "\n%d %c", ch, ch ) ;
}
```

```
main( )
{
    char  ch ;

    for ( ch = 0 ; ch <= 255 ; ch++ )
        printf ( "\n%d %c", ch, ch ) ;
}
```

Chars, signed and unsigned

- Would declaring ch as an unsigned char solve the problem?
- Even this would not serve the purpose since when ch reaches a value 255, ch++ would try to make it 256 which cannot be stored in an unsigned char.
- Thus the only alternative is to declare ch as an int.
- However, if we are bent upon writing the program using unsigned char, it can be done as shown below.

```
main( )
{
    unsigned char  ch ;

    for ( ch = 0 ; ch <= 254 ; ch++ )
        printf ( "\n%d %c", ch, ch ) ;

    printf ( "\n%d %c", ch, ch ) ;
}
```

Floats and Doubles

- A float occupies four bytes in memory and can range from -3.4e38 to +3.4e38
- If this is insufficient then C offers a double data type that occupies 8 bytes in memory and has a range from -1.7e308 to +1.7e308.
- A variable of type double can be declared as,
- double a, population ;
- beyond the range offered by double data type, then there exists a long double that can range from -1.7e4932 to +1.7e4932.
- A long double occupies 10 bytes in memory.
- most of the times in C programming one is required to use either chars or ints and cases where floats, doubles or long doubles would be used are indeed rare

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.

Storage Classes in C

- To fully define a variable one needs to mention not only its ‘type’ but also its ‘storage class’
- not only do all variables have a data type, they also have a ‘storage class’
- 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.

Storage Classes in C

- 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

What does storage class tell us

- a variable’s storage class tells us:
- Where the variable would be stored.
- What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
- What is the scope of the variable; i.e. in which functions the value of the variable would be available.
- What is the life of the variable; i.e. how long would the variable exist.

Storage Classes in C

- There are four storage classes in C:
- Automatic storage class
- Register storage class
- Static storage class
- External storage class

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.

Example

- 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 ) ;
}
```

May contain any value as garbage values are unpredictable

The output of the above program could be...

1211 221

Scope and Life of automatic variable

- 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

Scope and Life of automatic variable

- 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.
- The output of the above program would be:
- 3 2 1

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

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

Scope and Life of automatic variable

- 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.

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.

Register Storage Class

- 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
- 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.
- 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.

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

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

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

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

The output of the above programs would be:

```
1
1
1
```

```
1
2
3
```

Static Storage Class

- 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
```

Declaration and Definition

- 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

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

Declaration and 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
- Also remember that a variable can be declared several times but can be defined only once

Variables with same name

- Here x is defined at two places, once outside main( ) and once inside it
- 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

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

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


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

Static variable as extern

- 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

- We can make a few ground rules for usage of different storage classes in different programming situations with a view to:
- economise the memory space consumed by the variables
- 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

Register Storage class

- 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.

Extern Storage Class

- 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.

Auto Storage Class

- 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.