# C-BASIC DATA TYPES

## C BASIC DATA TYPES


Every variable and function in C programming has two properties: type and storage class. Data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. Storage class defines the scope (visibility) and life time of variables and/or functions within a C Program. This specifies precede the type that they modify.

The types in C can be classified as follows:

| S.no | Types | Description |
|---|---|---|
| 1 | Basic types | They are arithmetic types and consist of the two types:<br>(a) integer types and<br>(b) Floating-point types. |
| 2 | Enumerated types | They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program. |
| 3 | The type void | The type specifier void indicates that no value is available. |
| 4 | Derived types | They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types. |

# TOPIC 1 INTEGER TYPE

Following table gives you detail about standard integer types with its storage sizes and value ranges:

| Type | Storage size | Value range |
|---|---|---|
| char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions sizeof (type) yields the storage size of the object or type in bytes. Following is an example to get the size of int type on any machine:

## Example program

```
#include<stdio.h>
#include<limits.h>
int main()
{
        printf("%d",sizeof(int));
        return 0;
}
```

When you compile the above program you get the output as, Storage size for int: 4

### (1) Explanation related to range of an integer and the size of compiler

We had seen earlier that the range of an Integer constant depends upon the compiler. 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. Here a 16-bit compiler means that when it compiles a C program it generates machine language code that is targeted towards working on a 16-bit microprocessor like Intel 8086/8088. As against this, a 32-bit compiler like VC++ generates machine language code that is targeted towards a 32-bit microprocessor like Intel Pentium.

Note that this does not mean that a program compiled using Turbo C would not work on 32-bit processor. It would run successfully but at that time the 32-bit processor would work as if it were a 16-bit processor. This happens because a 32-bit

processor provides support for programs compiled using 16-bit compilers. If this backward compatibility support is not provided the 16-bit program would not run on it. This is precisely what happens on the new Intel Itanium processors, which have withdrawn support for 16-bit code.

Remember that out of the two/four bytes used to store an integer, the highest bit ($16^{th}$/$32^{nd}$ 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.

## (2) Signed integer and unsigned integer

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 ;
```

With such a declaration, the range of permissible integer values (for a 16-bit OS) will shift from the range -32768 to +32767 to the range 0 to 65535. Thus, declaring an integer as **unsigned** almost doubles the size of the largest possible value that it can otherwise take. This so happens 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. 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**.

## (3)Long int:-

C offers a variation of the integer data type that provides what are called **short** and **long** integer values. The intention of providing these variations is to provide integers with different ranges wherever possible. Though not a rule, **short** and **long** integers would usually occupy two and four bytes respectively. Each compiler can decide appropriate sizes depending on the operating system and hardware for which it is being written, subject to the following rules:

- **short**s are at least 2 bytes big
- **long**s are at least 4 bytes big
- **short**s are never bigger than **int**s
- **int**s are never bigger than **long**s

**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. More than this you should not need unless you are taking a world census.

If there are such things as **long**s, symmetry requires **short**s as well—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 ;
```

Naturally, most C programmers prefer this short-cut.

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.

## TOPIC 2 FLOATING POINT TYPES

### (1) Float, double

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;
```

If the situation demands usage of real numbers that lie even 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.

# TOPICS 3 CHAR TYPES

## (1) Signed char and unsigned char

**Signed** and **unsigned chars**, both occupying one byte each, but having different ranges. To begin with it might appear strange as to how a **char** can have a sign. 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.
- **Unsigned char** has a range from 0 to 255.

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

What output do you expect from this program? Possibly, 291 and the character corresponding to it. Well, not really. Surprised? The reason is that **ch** has been defined as a **char**, and a **char** cannot take a value bigger than +127. Hence when value of **ch** exceeds +127, an appropriate value from the other side of the range is picked up and stored in **ch**. This value in our case happens to be 35, hence 35 and its corresponding character # gets printed out.