# KCA102- Lecture

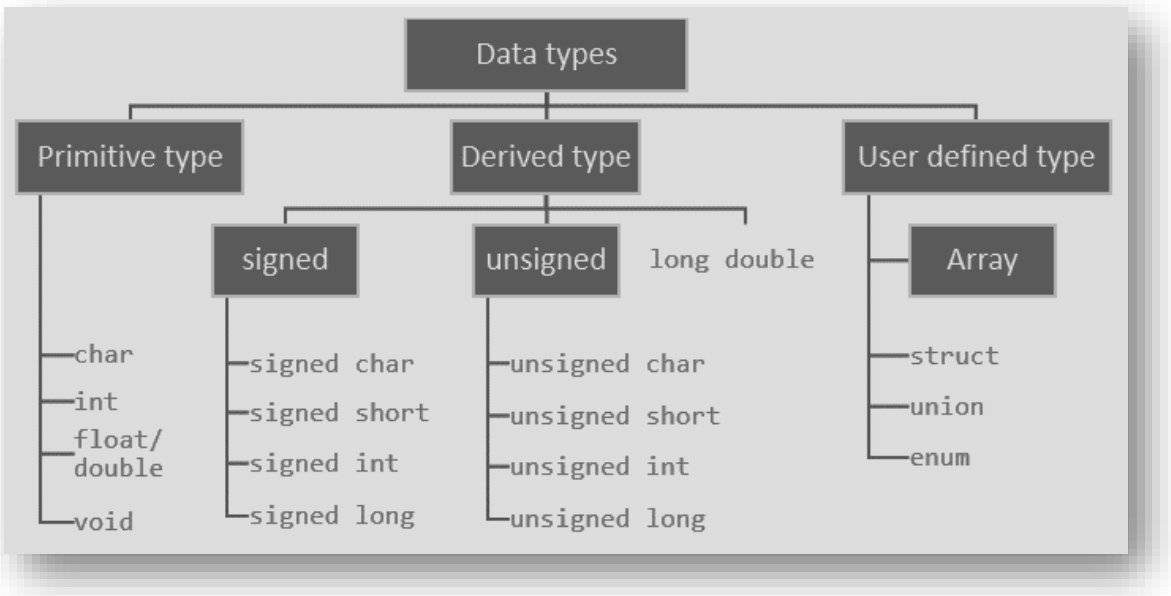
## Data Types in C

- **Primitive data types** are the first form – the basic data types (char, int, float, double).
- **Derived data types** are a derivative of primitive data types known as arrays, pointer and function.
- **User defined** data types are those data types which are defined by the user/programmer himself.

| S.N. | TYPES & DESCRIPTIONS |
|---|---|
| 1. | **Basic Types**<br>They are arithmetic types and are further classified into: (a) integer types (b) character types and (c) floating-point types. |
| 2. | **Enumerated types**<br>They are again arithmetic types and they are used to define variables that can only assign a certain discrete integer value throughout the program. |
| 3. | **The type void**<br>The type specifier void indicates that no value is available. |
| 4. | **Derived types**<br>They include (a) pointer types (b) Array types (c) structure types (d) union types and (e) function types |

Following are the primitive data types in C language. They are:-

- **int** – This data type is used to define an integer number (-….-3,-2,-1,0,1,2,3….). A single integer occupies 2 bytes.
- **char** – Used to define characters. A single character occupy 1 byte.
- **float** – Used to define **floating point numbers** (***single precision***). Occupies 4 bytes.
- **double** – Used for double precision floating point numbers(***double precision***). Occupies 8 bytes.



## Data Types and Range

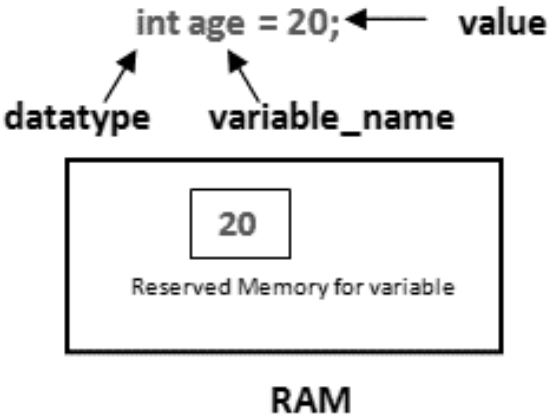
| Type | Size | Range | Precision for real numbers |
|---|---|---|---|
| char | 1 byte | -128 to 127 | |
| unsigned char | 1 byte | 0 to 255 | |
| signed char | 1 byte | -128 to 127 | |
| short int or short | 2 bytes | -32,768 to 32,767 | |
| unsigned short or unsigned short int | 2 bytes | 0 to 65535 | |
| int | 2 bytes | -32,768 to 32,767 | |
| unsigned int | 2 bytes | 0 to 65535 | |
| Long or long int | 4 bytes | -2147483648 to 2147483647 (2.1 billion) | |
| unsigned long or unsigned long int | 4 bytes | 0 to 4294967295 | |
| float | 4 bytes | 3.4 E–38 to 3.4 E+38 | 6 digits of precision |
| double | 8 bytes | 1.7 E-308 to 1.7 E+308 | 15 digits of precision |
| long double | 10 bytes | +3.4 E-4932 to 1.1 E+4932 | provides between 16 and 30 decimal places |

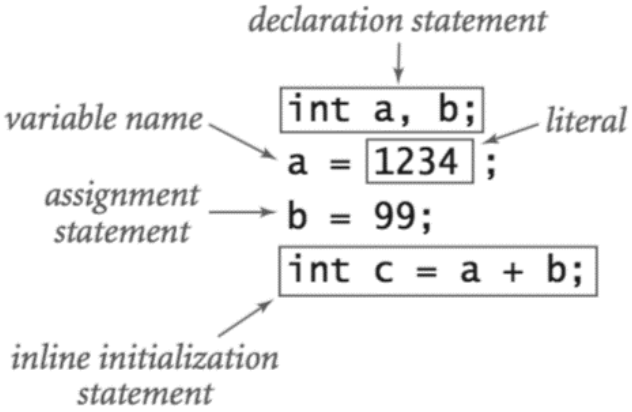
## Format Specifier/ Type Modifier used in C with Standard Input/Output functions ( printf()/scanf()

| 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.

## Variable Declaration and Assignment




## %d and %i Specifier

```
#include <stdio.h>
void main(void)
{
  int x = 100;
  printf("decimal = %d” , x);
  printf("dec = %i",  x);
  printf("octal = %o; hex = %x\n",  x, x);
  }
```

## Float and Double Data

Real number, analogous to scientific notation

Storage area divided into three areas:

Sign (0 for positive, 1 for negative)

Exponent (repeated multiplication)

Mantissa (binary fraction between 0.5 and 1)

type `double` format


The mantissa and exponent are chosen such that the following formula is correct

**Example**

```
#include<stdio.h>
void main ()
{
double average = 679999999.454;
float fnum = 679999999.454;
printf(“\naverage is %lf", average);
printf(“\n Fractional Number is %.2f", fnum);
)
```

# Floating Point Representation

type name

```
float myFloat; // Declaration only

int x = 10; // Declaration with initial value
char aLetter = 'q';
```

**Example**

| sign | | sign | |
|---|---|---|---|
| 0 | .1234567 | 0 | 04 |
| mantissa | | exponent | |

==> $+.1234567 \times 10^{+04}$

**Note:**

In Floating Point Number representation, only Mantissa(M) and Exponent(E) are explicitly represented. The Radix(R) and the position of the Radix Point are implied.

**Example**

**A binary number +1001.11 in 16-bit floating point number representation (6-bit exponent and 10-bit fractional mantissa)**

| | Sign | Exponent | Mantissa |
|---|---|---|---|
| | 0 | 000100 | 100111000 |
| or | 0 | 000101 | 010011100 |

In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

where M = *mantissa*, B = *base* and E = *exponent*

In C, we have two types that represent real numbers:

| C – data type | Bits Used | Bits used - Exponent | Bits used - Mantissa |
|---|---|---|---|
| `float` | 32 | 8 | 23 |
| `double` | 64 | 11 | 52 |

There are many ways to represent a floating-point number. Here is one way to represent the number 284:

**284** = $100011100_2$ = $1.000111 \times 2^8$

| 1-bit sign | 8-bit exponent | 23-bit mantissa |
|---|---|---|
| 0 | 00001000 | 100 0111 0000 0000 0000 0000 |

Since the leading digit in the mantissa is always 1 (for non-zero values), we can assume that this is implied in an improved representation as follows:

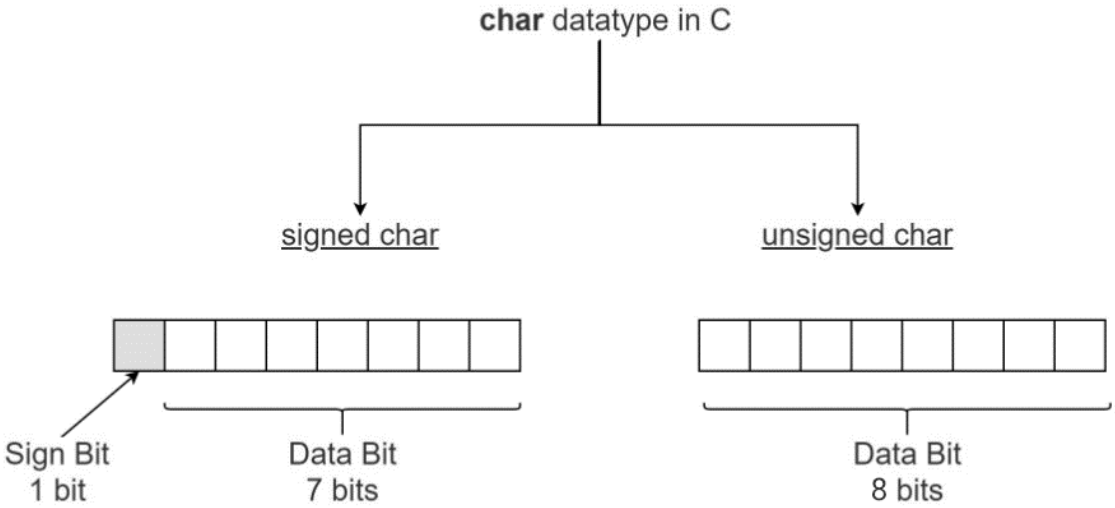
| 1-bit sign | 8-bit exponent | 23-bit mantissa |
|---|---|---|
| 0 | 00001000 | 000 1110 0000 0000 0000 0000 |

In the IEEE 754 32-bit floating-point standard, we add a bias of **127** to the exponent as follows:

| 1-bit sign | 8-bit biased exponent | 23-bit mantissa |
|---|---|---|
| 0 | 10000111 | 000 1110 0000 0000 0000 0000 |

```
#include <stdio.h>
void main(void)
{
 short int snum = 10000;
 int num = 121113991;
 long num1 = 49929929991;
 long num2 = 23009090909909933322;
 long long sum = num1 + num2;
 printf("\nShort Number is %hd, \nNumber is %d",snum, num );
 printf("\n Long Num1 is %ld, \n Large Long %ld", num1, num2)
printf("\n Sum of 2 Long Numbers, %lld", sum);
}
```

## Char Type Representation




Example
char ch =’A’; // Single char
char ch1=”A”; // String type
char Name[]=”Delhi”;// String as char array

```
int main()
 {
int x, y,z;
float num = 13.48;
char ch = 'G';
char city_name[] = “Greater Noida”;
x = 125;
y = 134;
z = x+y;
printf("%d \n", z);
printf("%f \n", num);
printf("%.2f \n", num);
printf("%07.2f \n", num);
```

```
printf("%c \n", ch);
printf("%d \n", ch);
return 0;
}
```

```
#includ<stdio.h>
int main()
{
short s = 10;
int i = 1000;
unsigned int ui = 45555;
long l = 1234567;
unsigned long ul = 1234567898;
float f = 3.5f;
double d = 23.9999;
long double ld = 23.239;
char ch1 = -128;
char ch2 = 'a';
unsigned char uc='b';
char str[10]=’Welcome”
printf("unsigned numerical value of char : %hhu \n", uc);
printf("short value : %hi \n", s);
printf("int value : %d \n",i);
printf("unsigned int value : %u \n", ui);
printf("long value : %ld \n", l);
printf("unsigned long value : %ul \n", ul);
printf("float value : %f \n ", f);
printf("double value : %lf \n", d);
printf("char value : %c \n", ch1);
printf("signed numerical value of char : %hhi \n", ch1);
printf("char value : %c \n", ch2);
printf("unsigned char value : %c \n", uc);
printf("Cgar type as  String value : %s \n", str);
return 0;
}
```

**Derived Data Types:** These are the data types derived from the primary data type.
Example: Array, Function, Pointer
**User Defined Data Type:** C allows programmers to create user defined types using the typedef and enum.
**typedef:** keyword used to give new identifier name to a exisiting data type.
The general format to use typedef is given below.
typedef type identifier
Where type is any existing data type and identifier is the new name given to it.

**Example typedef int num;**

In the above line we are giving integer int data type a new name num.
In the following example we have used typedef to give int a new name num. And then created a new variable and assigned integer value 10 to it.

## Literals and Constants

**Literals:**
The values assigned to each constant variables are referred to as the *literals*. Generally, both terms, constants and literals are used interchangeably.
**Example,** "*const int = 5;*", is a constant expression and
The value 5 is refered to as constant integer literal.

**Constant:**
**It an identifier whose value cannot be changed while executing a program**
**How to use constant**

1- **Using *const* keyword**
2- **Using pre-processor #define (macros)**

- The ***const*** keyword is used to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

- The keyword ***const*** can be declared before or after the data type.

**Syntax**

1. **Using const Keyword**

**const datatype identifier name = value;**

**example**
**const int plotlength=100;**

~~**const int plotlength;**~~

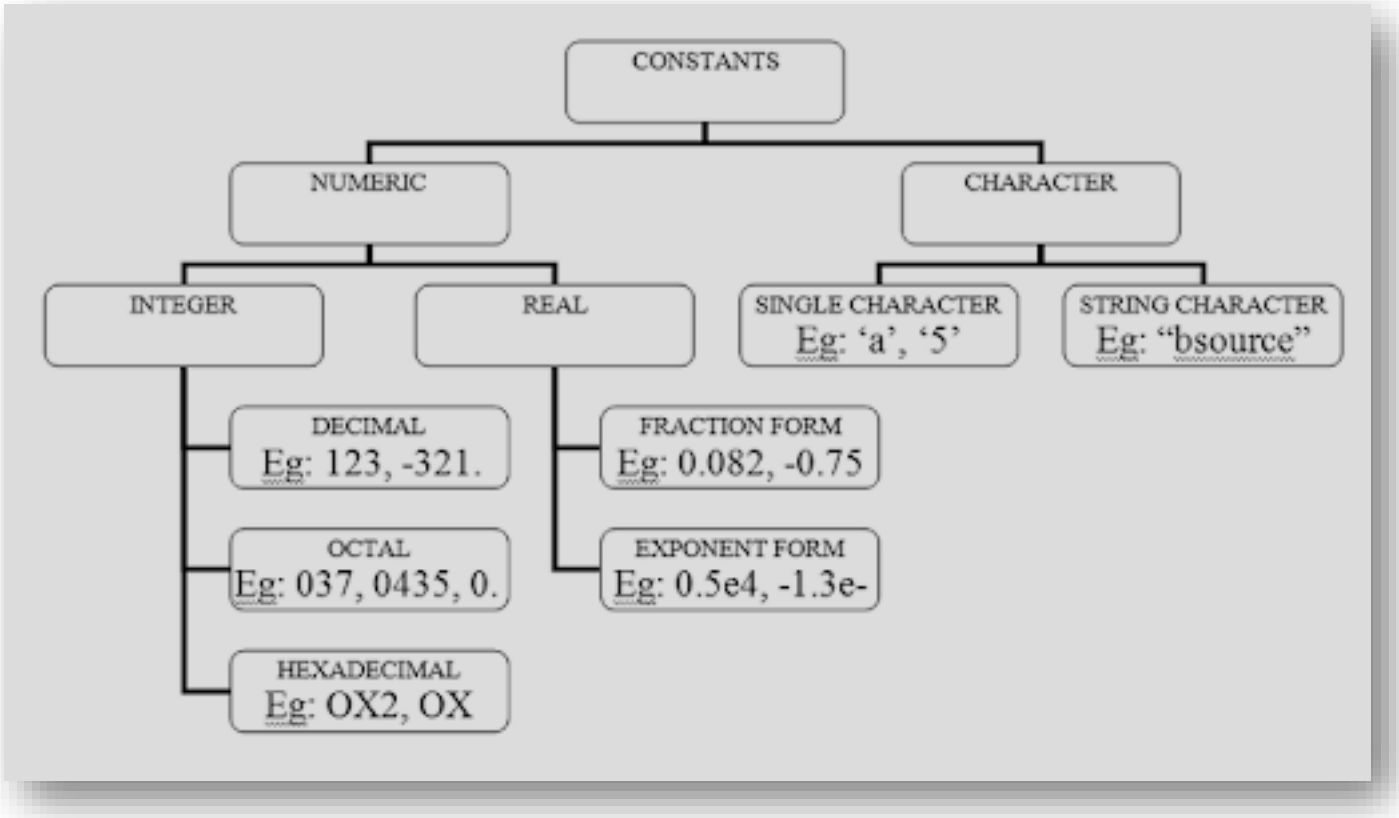
~~**plotlength=100;**~~

2. **Using #define**
   **Syntax**

   **#define identifier name value;**
   **example**
   **# define listsize 10**
   **# define pi  3.14**

```
#include <stdio.h>
#define pi 3.14
void main()
{
const int radius=12;
float area
area = 2 * pi*radius;
printf("Area with constant radius : %.2f",area);
}
```

**Types of Constants**




- Character constants
  A character enclosed in a single quotation mark
  Example:
  const char letter = 'n';
  const char number = '1';

```
        printf("%c", 'S');
```

```
#include <stdio.h>
#define pi 3.412
void main(void)
{
    double ht, radius, base, vol;
    printf("Enter the height and radius of the cone:");
    scanf("%lf %lf",&ht,&radius);
    base = pi * radius * radius;
    volume = (1.0/3.0) * base * height;
    printf("\nThe volume of a cone is %f ", vol;
}
```

◦

**Numeric Constants**

**Integer Constants**

As the name itself suggests, an integer constant is an integer with a fixed value, that is, it cannot have fractional value like 10, -8, 2019.

For example: **const signed int limit = 20;**

We may use different combinations of U and L suffixes to denote unsigned and long modifiers respectively, keeping in mind that its repetition does not occur.

We can further classify it into three types, namely:

- **Decimal number system constant:** It has the base/radix 10. ( 0 to 9)
  For example, 55, -20, 1.
  In the decimal number system, no prefix is used.
- **Octal number system constant:** It has the base/radix 8. ( 0 to 7 )
  For example, 034, 087, 011.
  In the octal number system, 0 is used as the prefix.
- **Hexadecimal number system constant:** It has the base/radix 16. (0 to 9, A to F)
  In the hexadecimal number system, 0x is used as the prefix. C language gives you the provision to use either uppercase or lowercase alphabets to represent hexadecimal numbers.

**2 Floating or Real Constants**

We use a floating-point constant to represent all the real numbers on the number line, which includes all fractional values.

**const long float pi = 3.14159;**

We may represent it in 2 ways:

- **Decimal form:** The inclusion of the decimal point ( . ) is mandatory.
  For example, 2.0, 5.98, -7.23.
- **Exponential form:** The inclusion of the signed exponent (either e or E) is mandatory.
  For example, the universal gravitational constant G = 6.67 x 10-11 is represented as 6.67e-11 or 6.67E-11.

**3 Character Constants**

Character constants are used to assign a fixed value to characters including alphabets and digits or special symbols enclosed within single quotation marks**( ' ' )**.

Each character is associated with its specific numerical value called the ASCII (American Standard Code For Information Interchange) value.
Apart from these values, there is a set in C known as ***Escape Sequences***
For example, '+', 'A', 'd'.

**4 String Constants**

A string constant is an array of characters that has a fixed value enclosed within double quotation marks **( " " )**.

**"Constant Flair", "Hello Friends!"**

***Example for using Constant in C?***

Here is a code in C that illustrates the use of some constants:

```
#include<stdio.h>
int main()
{
printf("Welcome to DataFlair tutorials!\n\n");
const int value = 4;
const float marks = 98.98;
const char grade = 'A';
const char name[30] = "DataFlair";
printf("The constant int value is: %d\n",value);
printf("The constant floating-point marks is: %f\n", marks);
printf("The constant character grade is: %c\n", grade);
printf("The constant string name is: %s\n",name);
return 0;
}
```

1. **Using *#define* preprocessor directive:** This directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:
   ```
   #define identifierName value
   ```
   - **identifierName:** It is the name given to constant.
   - **value:** This refers to any value assigned to identifierName.

   **Example:**
   ```
   #include<stdio.h>
   #define val 10
   #define floatVal 4.5
   #define charVal 'G'

   int main()
   {
      printf("Integer Constant: %d\n",val);
      printf("Floating point Constant: %.1f\n",floatVal);
      printf("Character Constant: %c\n",charVal);

      return 0;
   }
   ```
2. **using a *const* keyword**: Using *const* keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a *const* keyword.

   ```
   #include <stdio.h>
    int main()
   ```

```
{
   const int intVal = 10;  // int constant
   const float floatVal = 4.14; // Real constant
  const char charVal = 'A';     // char constant
   const char stringVal[10] = "ABC";     // string constant
   printf("Integer constant:%d \n", intVal );
   printf("Floating point constant: %.2f\n", floatVal );
   printf("Character constant: %c\n", charVal );
   printf("String constant: %s\n", stringVal);
  return 0;
}
```

**Macros and its types in C**

A **macro** is a slice of code in a program that is replaced by the value of the macro. Macro is defined by **#define** directive.

- Whenever a micro name is encountered by the compiler, it replaces the name with the definition of the macro.
- Macro definitions need not be terminated by semi-colon(**;**).

**Example**

```
#include <stdio.h>
 #define NUM 5
 void main()
{
   // Print the value of macro defined
   printf("The value of NUM"" is %d", NUM);
    }
```

```
#include <stdio.h>
 #define AREA(l, b) (l * b) //A  Macro definition

void main()
{
  int len = 10, bth = 5, area;
   area = AREA(len, bth);      // Find the area using macros
  printf("Area of rectangle" " is: %d", area);
 }
```

**Macros Vs. Constants**

- constant variable follows scoping rules, a #define macro does not
- A constant variable is type safe, a #define macro is not.

For example, suppose we want to change a constant "M" from 1.2 to 1, in either of the scenarios:

Using **const:**

```
const float M 1.2;
float num = M / 2;  // OUPUT  0.6
const float M 1;
float num = M / 2;  // WILL Be now0.5
```

**Refernces**

1. Hanly J. R. and Koffman E. B.,“Problem Solving and Program Design in C”, Pearson Education.
2. Schildt H., “C- The Complete Reference”, McGraw-Hill.
3. Kanetkar Y., “Let Us C”, BPB Publications.
4. Gottfried B., “Schaum’s Outlines- Programming in C”, McGraw-Hill Publications.
5. Kochan S.G., “Programming in C”, Addison-Wesley.
6. Dey P. and Ghosh M., “Computer Fundamentals and Programming in C”, Oxford University Press.
7. Goyal K. K., Sharma M. K. and Thapliyal M. P. “Concept of Computer and C Programming”, University Science Press.
8. Goyal K. K. and Pandey H.M., Trouble Free C”, University Science Press