Unions :-

Union is an user defined datatype in C programming language. It is a collection of variables of different datatypes in the same memory location. We can define a union with many members, but at a given point of time only one member can contain a value.

Need for Union in C programming:-

C unions are used to save memory. To better understand an union, think of it as a chunk of memory that is used to store variables of different types. When we want to assign a new value to a field, then the existing data is replaced with new data. C unions allow data members which are mutually exclusive to share the same memory. This is quite important when memory is valuable, such as in embedded systems. Unions are mostly used in embedded programming where direct access to the memory is needed

You can define a union with many members, but **only one member can contain a value at any given time**. **The memory occupied by a union will be large enough to hold the largest member of the union. So, the size of a union is the size of the biggest component. At a given point in time, only one can exist. All the fields overlap and they have the same offset : 0.**

The format for defining new type using union is as below:

```
union Tag    // union keyword is used
{
     data_type member1;
     data_type member2;
    data_typemember n;
};   // ; is compulsory
```

Example:

```
union car
{
        char name[10];
        float price;
};
```

Now, a variable of car type can store a floating-point number, or a string i.e., **same memory location is used to store multiple types of data**. In the above example, Data type will occupy 10 bytes of memory space. Because this is the maximum space which can be occupied by a character array. Program 1 displays the total memory size occupied by the union.

Program 1:

```
#include <stdio.h>
union car
{                                   //Assumptions
        char name[10];              // 1 byte for char
        float price;                // 4 bytes for float
};
int main( )
{
        union car  c;
        printf( "Memory size occupied by data in bytes : %d\n", sizeof(car));
        return 0;
}
```

Output:

```
Memory size occupied by data in bytes : 10
```

Accessing Union Members using union variable

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access.

Program 2

```
 #include <stdio.h>
#include <string.h>
union car
{
    char name[10];
    float price;
};
int main( )
 {
    union car c;
    strcpy( c.name, " Benz");
    c.price=4000000.00;
    printf( "car name: %s\n", c.name);
    printf( "car price: %f\n", c.price);
    return 0;
}
```

Output:

When the above code is compiled and executed, the value of name gets corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of price member printed well.

Program 3: print all the members of union

```
#include <stdio.h>
#include string.h>
union car
{
        char name[10];
        float price;
};
int main( )
{
        union car c;
        strcpy( c.name, "Benz");
        printf( "car name: %s\n", c.name);
         c.price=4000000.00;
        printf( "car price: %f\n",c.price);
         return 0;
}
```

Output:

```
Car name: Benz
Car price:4000000.00
```

Structure v/s Union

Structure:

1. The keyword struct is used to define a structure.
2. When a variable is associated with a structure, memory is allocated to each member of the structure. The size of structure is greater than or equal to the sum of sizes of its members.

3. Each member within a structure is assigned unique storage area of location

4. In Structure, the address of each member will be in ascending order. This indicates that memory for each member will start at different offset values

5. In Structure, altering the value of a member will not affect other members of the structure.

6. All members can be accessed at a time.

Union:

1. The keyword union is used to define a union

2. When a variable is associated with a union, memory is allocated by considering the size of the largest memory. So, size of union is equal to the size of largest member.

3. Memory allocated for union is shared by individual members of union.

4. For unions, the address is same for all the members of a union. This indicates that every member begins at the same offset value.

5. Altering the value of any of the member will alter other member values

6. In Unions, only one member can be accessed at a time

Similarites between structures and unions

1.Both are user-defined data types used to store data of different types as a single unit.

2.Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.

3.Both structures and unions support only assignment = and sizeofoperators. The two structures or unions in the assignment must have the same members and member types.

4.A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.

5. '.'operator is used for accessing members.

Unions inside structure

In the below code, we have union embedded with in a structure. We know, the fields of a union will share memory, so in main program we ask the user which data he/she would like to store and depending on the user choice the appropriate field will be used. By this way we can use the memory efficiently

Program 4

```
#include<stdio.h>
struct student
 {
        union
          { //anonymous union (unnamed union)
                char name[10];
                int roll;
          };
    int mark;
};
int main()
 {
        struct student stud;
        char choice;
        printf("\n You can enter your name or roll number ");
        printf("\n Do you want to enter the name (y or n): ");
        scanf("%c",&choice);
        if(choice=='y'||choice=='Y') {
```

```
        printf("\n Enter name: ");
        scanf("%s",stud.name);
        printf("\n Name:%s",stud.name);
  }
  else {
        printf("\n Enter roll number");
        scanf("%d",&stud.roll);
        printf("\n Roll:%d",stud.roll);
  }
        printf("\n Enter marks");
        scanf("%d",&stud.mark);
        printf("\n Marks:%d",stud.mark);
        return 0;
}
```

Output –

```
You can enter your name or roll number
Do you want to enter the name (y or n) : y
Enter name: john
Name:john
 Enter marks: 45
Marks:45
```

Structures inside Unions

Program 5

```
#include<stdio.h>
int main() {
struct student
 {
        char name[30];
        int rollno;
        float percentage;
};
union details {
struct student s1;
};
union details set;
printf("Enter details:");
printf("\nEnter name : ");
scanf("%s", set.s1.name);
printf("\nEnter roll no : ");
scanf("%d", &set.s1.rollno);
printf("\nEnter percentage :");
scanf("%f", &set.s1.percentage);
printf("\nThe student details are : \n");
printf("\name : %s", set.s1.name);
printf("\nRollno : %d", set.s1.rollno);
printf("\nPercentage : %f", set.s1.percentage);
return 0;
}
Output –
  Enter details:
  Enter name : jeny
  Enter roll no : 123
  Enter percentage :67
  The student details are :
  Name : jeny
  Rollno : 123
  Percentage : 67.000000
```

Bitfields

In programming terminology, a bit field is a data structure that allows the programmer to allocate memory to structures and unions in bits in order to utilize computer memory in an efficient manner.

Need for Bit Fields in C

Bit fields are of great significance in C programming, because of the following reasons:

- Used to reduce memory consumption.
- Easy to implement.
- Provides flexibility to the code.

In C, we can specify size (in bits) of **structure and union members**. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit.

Bit Field Declaration

```
    struct {

        type [member_name] : width ;

      };
```

type-An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.

member_name -The name of the bit-field.

width -The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Program 6:

```
  #include<stdio.h>
  struct Status
  {
     unsigned int bin1:1; // 1 bit is allocated for bin1. only two digits can be  stored 0 &1.
     unsigned int bin2:1;
      // if it is signed int bin1:1   or int bin1:1,   one bit is used to represent the sign
   };
int main()
{
          printf("Size of structure is %lu\n",sizeof(struct Status));        // 4 bytes
         struct Status s;
          printf("enter the number");
         //scanf("%d",&s.bin1);    // Error
```

```
            // We cannot access the address of bit - field. system is byte addressable.
             return 0;
 }
```

Program 7- Below code demonstrates the maximum value that can be stored in those bits is restricted. Results in warning.

If you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows

```
struct {

  unsigned int age : 3;

} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so.

```
#include <stdio.h>
#include <string.h>
struct {
  unsigned int age : 3;
} Age;
int main( )
{
        Age.age = 4;
        printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
        printf( "Age.age : %d\n", Age.age );
        Age.age = 7;
```

```
            printf( "Age.age : %d\n", Age.age );
            Age.age = 8;
            printf( "Age.age : %d\n", Age.age );
            return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result

```
program1.c: In function 'main':
program1.c:17:14: warning: unsigned conversion from 'int' to 'unsigned char:3' changes value from '8' to '0' [-Woverflow]
    Age.age = 8;
              ^
```

Output –

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Program 8: Demonstration of assigning signed value to an Unsigned variable for which bit fields are specified.

```
#include<stdio.h>
struct Status
{
        unsigned int bin1:4;    // 4 bits is allocated for bin1. 0 to 15, any number can be
                                   used.
        unsigned int bin2:2;    // 2 bits allocated for bin2. 0 to 3, any number can be used
};
int main()
```

```
{
        printf("Size of structure is %lu\n",sizeof(struct Status));// again 4 bytes struct
        Status s1;
        s1.bin1=-7;     // it is unsigned. but sign is given while assigning
        s1.bin2=2;
        printf("%d %d",s1.bin1,s1.bin2);                  // 9 2
        return 0;
}
```

Program 9: Structure having two members with signed and unsigned variable for which bit fields are specified.

```
#include<stdio.h>
struct Status
  {     int bin1:4; // one bit is used for representing the sign. Another 3 bits for data
        unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used
};
int main()
    {       printf("Size of structure is %lu\n",sizeof(struct Status));   // again 4 bytes
            struct Status s1;
            s1.bin1=-7;
            s1.bin2=2;
            printf("%d %d",s1.bin1,s1.bin2);          // -7 2
            return 0;
}
```

Program 10: Array of bit fields not allowed.

Below code results in Error.

```
#include<stdio.h>

 struct Status
 {
              unsigned char bin1[10]:40;// invalid type
 };
 int main()
     {        printf("Size of structure is %lu\n",sizeof(struct Status));
              return 0;
     }
```

Program 11: We cannot have pointers to bit field members as they may not start at a byte boundary. This code results in error.

```
 #include<stdio.h>
 struct Status
 {
              int bin1:4;  // 1 bit is used for representing the sign. Another 3 bits for data
              int *p:2;
 };
 int main()
 {
              printf("Size of structure is %lu\n",sizeof(struct Status));
               return 0;
 }
```

Program 12: Storage class can't be used for bit field. This code results in error.

```
 #include<stdio.h>

 struct Status
 {
         static int bin1:32;// Any storage class not allowed for bit field
         int p:2;
 };
 int main()
 {
         printf("Size of structure is %lu\n",sizeof(struct Status));
          return 0;
 }
```

Program 13: It is implementation defined to assign an out-of-range value to a bit field member. This code results in error.

```
  #include<stdio.h>
 struct Status
 {
         int bin1:33;            // Error: width exceeds its type
         int p:2;
 };
 int main()
 {
         printf("Size of structure is %lu\n",sizeof(struct Status));
         return 0;
 }
```

Program 14: Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field can cause the next field to be aligned on the next container boundary where the container is the same size as the underlying type of the bit field.

Below **c**ode demonstrates the force alignment on next boundary.

```
#include<stdio.h>
struct Status
 {
          int bin1:4;     // one bit is used for representing the sign. Another 3 bits for data
          //int i:0;// you cannot allocate 0 bits for any member inside the structure. Error
           int :0;// A special unnamed bit field of size 0 is used to force alignment on
          // next boundary. observe no member variable. only size is 0 bits
         unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used
 };
 int main()
 {
          // observe the size of the structure. Now 8 bytes
          printf("Size of structure is %lu\n",sizeof(struct Status));       // 8 bytes
          struct Status s1;
          s1.bin1=-7;
          s1.bin2=2;
          printf("%d %d",s1.bin1,s1.bin2);          // -7 2
          return 0;
 }
```

Enumerations

Introduction:

Enumeration (or enum) is a way of creating user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain. It is easy to remember names rather than numbers.

Example scenario: We all know google.com. We don't remember the IP address of website google.com

The keyword 'enum' is used to declare new enumeration types in C. In Essence, Enumerated types provide a symbolic name to represent one state out of a list of states. The names are symbols for integer constants, which won't be stored anywhere in program's memory. **Enums are used to replace #define chains**

Enumeration data type consists of named integer constants as a list. It start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list.

Syntax: enum identifier { enumerator-list }; // semicolon compulsory & identifier optional

Examples:

enum month { Jan, Feb, Mar }; // Jan,Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default

enum month { Jan = 1, Feb, Mar };// Feb and Mar variables will be assigned to 2 and 3 respectively by default

Note: These values are symbols for integer constants, which won't be stored anywhere in program's memory. It doesn't make sense to take its address.

Program 1: Enum variables are automatically assigned values if no value specified.

```
#include<stdio.h>
enum months
{
        jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec
        // symbol jan gets the value 0, all others previous+1
        //If we do not explicitly assign values to enum names, the compiler by default assigns
        //values starting from 0. jan gets value 0, feb gets 1, mar gets 3 and so on.
```

```
            // all constants are separated by comma(,).

            // no memory allocated for these constants. Available in this file anywhere directly.

};          // compulsory semi-colon
int main()
{           enum months m;       // memory is allocated for m.

            printf("sizeof m is %lu\n",sizeof(m));       // same as the size of integer
            m=apr;          // 3 is stored in m
            printf("%d",m);               //3
}
```

Program 2: We can assign values to some of the symbol names in any order. All unassigned names get value as value of previous name plus one.

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5, wednesday, thursday = 10, friday, saturday};
int main()
{       printf("enter the value for monday\n");
        //scanf("%d",&monday);    // error: not legal
          // Memory not allocated for constants in enum. So ampersand(&) can't be used with those.
          //Also value already assigned to monday that can't be changed
        printf("%d %d %d %d %d %d %d", sunday, monday, tuesday, wednesday, thursday,
                                                                  friday, saturday);
        return 0;
}
```

Program 3: Only integer constants are allowed.

Below code results in Error if abc=3.5 is uncommented.

```
#include<stdio.h>
enum examples

{       //abc=3.5, bef; // Value is not an integer constant

            abc="Hello", bef;              // Value is not an integer constant

};
```

```
int main()
   {        enum examples e1;
            return 0;
   }
```

Program 4: Valid arithmetic operators are +, - * and / and %

```
#include<stdio.h>
enum months
{
          jan,feb,mar,apr,may,jun,july,aug,sep,oct,nov,dec
};
int main()
{
          printf("%d\n",mar-jan);                // valid
          printf("%d\n",mar*jan);                // valid
          printf("%d\n",mar&&feb);               // valid
          //mar++;              //error          : mar is a constant
          //printf("after incrementing %d\n",mar);// error
          enum months m=feb;
           m=(enum months)(m + jan); // m can be changed. m is a variable of type enum months.
           // But all constants in enum cannot be changed its value
          printf("m, after incrementing %d\n",m);
          for(enum months i=jan;i<=dec;i++) // loop to iterate through all constants in enum
          {
                 printf("%d\n", i);
          }
          return 0;
}
```

Program 5: Enumerated Types are Not Strings .Two enum symbols/names can have same value.

```
 #include <stdio.h>
 enum State {Working = 1, Failed = 0, Freezed = 0};
 int main()
     {
        printf("%d, %d, %d", Working, Failed, Freezed);          // associated values are printed
        return 0;
 }
```

Program 6: All enum constants must be unique in their scope.

Below code results in Error: Redeclaration of enumerator 'def'.

```
#include<stdio.h>
 enum   example1   {def,   cdt};
 enum example2 {abc, def};
  int main()
 {
            enum example1 e;
            return 0;
  }
```

Program 7: It is not possible to change the constants.

Uncommenting the statement feb=10, results in error.

```
#include<stdio.h>
 enum months
 {
            jan,feb=11,mar,apr,may=23,jun,july
 };
 int main()
 {
            enum months m;
            //feb=10;                // error
            printf("%d",feb);               // no error              // we can access it directly
            return 0;
 }
```

Program 8:Storing the symbol of one enum in another enum variable. It is allowed.

```
 #include<stdio.h>
 enum nation
 {
            India=1, Nepal
 };
 enum States
 {
            Delhi, Katmandu
 };
 int main()
 {
            enum States n=Delhi;
            enum States n1=India;
            // Value 1 is stored in n1 which is of type enum States
            enum nation n2=Katmandu;
            printf("%d\n",n);
            printf("%d\n",n1);
            printf("%d",n2);
            return 0;
 }
```

Program 9:

One of the short comings of Enumerated Types is that they don't print nicely. To print the "String"(symbol) associated with the defined enumerated value, you must use the following cumbersome code

```
#include<stdio.h>
enum example1
{
        abc=123, bef=345,cdt=555
};
void printing(enum example1 e1);
int main()
{
        enum example1 e1;
        // how to print abc, bef and cdt based on user choice?? printf("enter the number");
        scanf("%d",&e1); //enter any number
        printing(e1);       // user defined function to print the symbol name
        return 0;
}
void printing(enum example1 e1)
{
        switch(e1)
        {
              case  abc:printf("abc");break;
              case   bef:printf("bef");break;
              case  cdt:printf("cdt");break;
              default:printf("no symbol in enum with this value");
              break;
        }
}
```

Program 10: The value assigned to enum names must be some integral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

Below code Results in Error: overflow in enumeration values.

```
 #include<limits.h>
#include<stdio.h>
enum examples
{
        abc = INT_MAX, def,cdt
        // INT_MAX is defined in limits.h
        // def must get INT_MAX+1 value which is an error
        // delete def and cdt and compile and run again
};
int main()
{
        enum examples e;
        e=abc;
        printf("%d",e);
        return 0;
}
```

Program 11:Enumerations are actually used to restrict the values to a set of integers within the range of values. But outside the range, if any value is used it is not an error in C standards like C11, C99 and C89. Other languages(ex: Java) doesn't support any value outside the range.

Code to demonstrate why enum supports values outside range values in C.

```
#include<stdio.h>
enum design_flags
  {
        bold =1, italics=4, underline=8

   }df;
```

```
int main()
{
        df=bold;
        df=italics|bold;   // to set bold and italics. Now df contains 5 which is not defined in  enum
        // if you write again df=italics, 5 is replaced with 4. So text is no more bold. It is only italics

        if(df==(bold|italics))
                printf("both\n");
        else if(df==bold)
                printf("bold it is\n");
        else if(df==italics)
                printf("italics it is\n");

        if(df&bold)     // whether the text is bold
                printf("bold\n");
        if(df & italics) // whether the text is italics
                printf("italics\n");
        if(df & underline)      // whether the text is underlined
                printf("underline\n");
        return 0;
}
```

Introduction

Storage Classes are used to describe the features of a variable/function. These features basically include the scope(visibility) and life-time which help us to trace the existence of a particular variable during the runtime of a program.

The following storage classes are most often used in C programming.

Automatic variables (auto)

External variables (extern)

Static variables(static)

Register variables (register)

Global variables

Automatic Variables

A variable declared inside a function without any storage class specification is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function execution is completed. Automatic variables can also be called local variables because they **are local to a function.** By default, they are assigned to undefined values**.**

Program 1:

```
 #include<stdio.h>
  int main()
 {
      int i=90;              // by default auto because defined inside a function main()
      auto float j=67.5;
      printf("%d %f\n",i,j);
 }
 int f1()
 {
             int a;  // by default auto because declared inside a function f1()
                     // Not accessible outside this function.
 }
```

External variables

The extern keyword is used before a variable **to inform the compiler that the variable is declared somewhere else**.The extern declaration does not allocate storage for variables. All functions are of type extern. **The default initial value of external integral type is 0 otherwise null.**

Program 2:

```
int main(){
        extern int i;     // Information saying declaration exist somewhere else and make it available during linking
            // if u comment this line, it throws an error: i undeclared
        printf("%d\n",i);
 }
 int i=23;
```

Note: The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions. When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file

Program 3: Two files share the same global variable

Sample.c

```
        #include <stdio.h>
         int count ;
        extern void write_extern();
         main() {
           count = 5;
           write_extern();
        }
```

Sample1.c

```
#include <stdio.h>
 extern int count;
 void write_extern(void) {
   printf("count is %d\n", count);
}
```

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, Sample.c. Now, compile these two files as follows

gcc Sample.c Sample1.c

It will produce the executable program **a.out**. When this program is executed, it produces the following result −

Output - count is 5

Static variables

A static variable tells the compiler to persist the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static is initialized only once and remains into existence till the end of program. A static variable can either be local or global depending upon the place of declaration.

Scope of local static variable remains inside the function in which it is defined but the life time of local static variable is throughout that program file. Global static variables remain restricted to scope of file in each they are declared and life time is also restricted to that file only. **All static variables are assigned 0 (zero) as default value.**

Program 4: Demo for Life time of Static variable

```
#include<stdio.h>
 int* f1();
int main()
{
		int *res = f1();
```

```
            printf("Res is %p %d\n",res,*res);          //same address in all three outputs. Then 11
            res = f1();
            printf("Res is %p %d\n",res,*res);          // 12
            res = f1();
            printf("Res is %p %d\n",res,*res);          // 13
            return 0;
}
int* f1()
{
    static int a= 10;// memory is shared. This line is ignored after first function call
     a++;
    return &a;    // valid because life time of static variable is through out the file execution
}
```

Program 5: Understand the difference between local and global static variable.

```
#include<stdio.h> void
f1();
static int j=20;                    // global static variable: cannot be used outside this program file int
k=23;                         //global variable: can be used anywhere by linking with this file. By
default   has extern with it.
int main()
    {       f1();                   // 0   20
            f1();                   // 0   21
            f1();                   // 0   22
            return 0;
}
void f1()
{           int i=0;
            printf("%d %d\n",i,j);
```

```
        i++;
        j++;
 }
```

Register variables

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. The keyword register hints to compiler that a given variable can be put in a register. It's compiler's choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register. If a free register is not available, these are then stored in the memory only. If & operator is used with a register variable then compiler may give an error or warning (depending upon the compiler used), because when a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid.

Program - 6

```
#include<stdio.h>
int main()
{
        register int i = 10;
        int* a = &i;
        printf("%d", *a);
        getchar();
        return 0;
}
```

Global variables

The variables declared outside any function are called global variables. They are not limited to any function. **Any function can access and modify global variables**. Global variables are **automatically initialized to 0 at the time of declaration.**

Program 6:

```
#include<stdio.h>
int i = 100;
int j;
int main()
    {       printf("%d\n",i);                        // 100
            i=90;
            printf("%d\n",i);          // 90
            f1();
            printf("%d\n",i);          // 40
            printf("%d\n",j);          // 0 by default, global variable is 0 if just declared
            return 0;
    }
int f1()
{
        i = 40;  // Any function can modify the global variable.
}
```

Think about it!
What if we have int i = 200; in a function f1() ?

Introduction

Qualifiers are keywords which are applied to the data types resulting in Qualified type. Applied to basic data types to alter or modify its sign or size.

Types of Qualifiers are as follows.

- **Size Qualifiers**
- **Sign Qualifiers**
- **Type qualifiers**

Size Qualifiers

Qualifiers are prefixed with data types to **modify the size of a data type** allocated to a variable. Supports two size qualifiers, **short and long. T**he Size qualifier is generally used with an integer type. In addition, double type supports long qualifier.

Rules regarding size qualifier as per ANSI C standard:

short int <= int <=long int

float <= double <= long double

Note: short int may also be abbreviated as short and long int as long. But, there is no abbreviation for long double.

Program 1:

```
#include<stdio.h>
int main()
{           short int i = 100000; // cannot be stored this info with 2 bytes. So warning
            int j = 100000;  // add more zeros and check when compiler results in warning
            long int k = 100000;
            printf("%d %d %ld\n",i,j,k);
            printf("%d %d %d",sizeof(i),sizeof(j),sizeof(k));
            return 0;
}
```

Sign Qualifiers

Sign Qualifiers are used to specify the signed nature of integer types. It specifies whether a variable can hold a negative value or not. It can be used with int and char types

There are two types of Sign Qualifiers in C: s**igned and unsigned**

A signed qualifier specifies a variable which can hold both positive and negative integers

An unsigned qualifier specifies a variable with only positive integers.

Note: In a t-bit signed representation of n, the most significant (leftmost) bit is reserved for the sign, "0" means positive, "1" means negative.

Program 2:

```
#include<stdio.h>
int main()
{
        unsigned int a = 10;
        unsigned int b = -10;   // observe this
        int c = 10;  // change this to -10 and check
        signed int d = -10;
        printf("%u %u %d %d\n",a,b,c,d);
        printf("%d %d %d %d",a,b,c,d);
        return 0;
}
```

Type Qualifiers

A way of expressing additional information about a value through the type system and ensuring correctness in the use of the data. Type Qualifiers consists of two keywords i.e., **const** and **volatile.**

The **const** keyword is like a normal keyword but the only difference is that once they are defined, their values can't be changed. They are also called as literals and their values are fixed.

Syntax: const data_type variable_name

Program 3

```
#include <stdio.h>
void main()
{
        const int height = 100; /*int constant*/
        const float number = 3.14; /*Real constant*/
        const char letter = 'A'; /*char constant*/
        const char letter_sequence[10] = "ABC"; /*string constant*/
        const char backslash_char = '\?'; /*special char cnst*/
        //height++; //error
        printf("value of height :%d \n", height );
        printf("value of number : %f \n", number );
        printf("value of letter : %c \n", letter );
        printf("value of letter_sequence : %s \n", letter_sequence);
        printf("value of backslash_char : %c \n", backslash_char);
}
```

Output:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

Program 4

```
#include<stdio.h>
int main()
{
          /*const int i = 5;
          printf("%d\n",i);
          i = 6;     // error
```

```
printf("%d",i);
*/

/*const int i;
i = 10;   //error
*/

/*
int i = 5;
int j = 6;
const int *p = &i;            // p is a pointer to constant integer
printf("%d\n",*p);
p = &j;   // no error
//*p = j; // error
printf("%d\n",*p);
*/

/*const int i = 5;
int *p = &i;
printf("%d\n",*p);
*p = 6;// only warning. But code works
printf("%d\n",*p);
*/

/*int i = 5;
int j = 6;
int* const p = &i;                // p is a constant pointer to integer
printf("%d\n",*p);
//p = &j; // eror
*p = j;
printf("%d\n",*p);
```

```
	*/

	int i = 5;
	int j = 6;
	const int* const p = &i;	//p is a constant pointer to constant integer
	 //p = &j; // eror
	//*p = j;  // error

	//const const int a; // illegal
	return 0;
}
```

The volatile keyword is intended to prevent the compiler from applying any optimizations. Their values can be changed by the code outside the scope of current code at any time. Also, can be changed by any external device or hardware.

Syntax: volatile data_type variable_name

Program 6:

Version1: No volatile keyword added while declaring the variable a. Run this code using –save-temps and observe the size of .s(assembly file). Optimization is done.

```
#include<stdio.h>
int main()
{
	int a = 0;
	if(a == 0)
		printf("a is 0\n");
	else
		printf("a is not zero\n");
	return 0;
}
```

Version2: volatile keyword added while declaring the variable a. Run this code using –save-temps and observe the size of .s(assembly file). Optimization is not done.

```
#include<stdio.h>
int main()
{
        volatile int a = 0; // observe this
        if(a == 0)
                printf("a is 0\n");
        else
                printf("a is not zero\n");
        return 0;
}
```

Execution steps:

```
gcc program6.c   -save-temps(press enter)         // creates 3 files(.i, .o, .s)
dir program6.i  program6.o  program6.s(press enter)  //windows OS. Check the size of each
ls –l program6.i  program6.o  program6.s(press enter)  // Ubuntu OS
```

Applicability of Qualifiers to Basic Types

The below table helps us to understand which Qualifier can be applied to which basic type of data.

No.	Data Type	Qualifier
1.	char	signed, unsigned
2.	int	short, long, signed, unsigned
3.	float	No qualifier
4.	double	long
5.	void	No qualifier

Introduction

The execution of a ‘C’ program takes place as a multi stage process. The stages are: Pre-processing, Compilation, Loading, Linking and Execution. The first stage is Pre - processing, which is performed by the ‘C’ pre-processor. It is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. **Possible to see the effect of the pre-processor on source files directly by using the -E option of gcc**

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include -> Inserts a particular header from another file.**
3	**#undef -> Undefines a preprocessor macro.**
4	**#ifdef -> Returns true if this macro is defined.**
5	**#ifndef -> Returns true if this macro is not defined.**
6	**#if -> Tests if a compile time condition is true.**
7	**#else -> The alternative for #if.**
8	**#elif -> #else and #if in one statement.**
9	**#endif -> Ends preprocessor conditional.**

10	**#error-> Prints error message on stderr.**
11	**#pragma ->Issues special commands to the compiler, using a standardized method.**

Macro in C

#define: Used to define a Macro. Macro is a piece of code in a program which has been given a name. During preprocessing, when the ‘C’ preprocessor encounters the name, it substitutes the name with the piece of code.

Program 1: Macros doesn't judge anything

```
#include<stdio.h>

#define PI 3.14        // symbolic constant, macro, plain text substitution

// It doesn't judge anything & No memory is allocated for this

int main()

{

 printf("%f",PI);

 return 0;

}
```

Program 2: No Memory allocation for Macro

```
// name given to 10 is MAX. No memory is allocated for MAX. So & MAX is an error.

#include<stdio.h>

#define MAX 10
```

```
int main()

{

 int a[MAX];

 scanf("%d", &MAX);// Error

 return 0;

}
```

Program 3: Defining a string using #define

```
#include<stdio.h>

#define STR "Hello All"     //#define can be used for strings as well

int main()

{

 printf("The string is %s\n",STR); // The string is Hello All

 return 0;

}
```

Program 4: Macro with expression

```
#include<stdio.h>

#define STR 2+5*1  //macro with expression

int main()

{

  printf("Value is %d\n",STR); //STR replaced with 2+5*1 in preprocessing stage.
```

```
    //During execution, expression is evaluated
  return 0;
}
```

Program 5: Macro with parameter

```
#include<stdio.h>
#define SUM(a,b) a+b  // SUM is not a function. It is a macro.
int main()
{
 int a,b;
 printf("Enter two numbers:\n");
 scanf("%d%d",&a,&b);
 printf("The sum of two numbers is %d\n",SUM(a,b));
 return 0;
}
```

Program 6: Demo of plain text substitution

```
#include<stdio.h>
#define sqr(x) (x*x)  //change this to (x)*(x).
int main()
{
 int y=8;
 printf("%d",sqr(2+3));       // 11(2+3*2+3), Not (2+3)*(2+3)

 return 0;

}
```

What changes in Program 6 to perform (2+3)*(2+3) ?

#define sqr(x) (x)*(x) – Think about this.

Program 7: Macro in another macro

```
#include<stdio.h>

#define sqr(x) x*x

#define cube(x) sqr(x)*x  // using sqr in cube

int main()

{

      printf("The cube of 9 is d\n",cube(9));

      return 0;

}
```

Program 8: #define is used to define constants. These constants cannot be changed using assignment operator(=). Code results in error

```
#include<stdio.h>

#define MAX 200

int main()
```

```
{
  if(MAX==20)
          printf("hello");
  else
          {
                  printf("U here??");
              MAX=20;// Error because no memory allocated for MAX.To avoid this,
                          //use #define and check
           }
          printf("MAX is %d",MAX);
          return 0;
}
```

Program 9: Redefining the macro with #define is allowed. But not advisable.//Results in warning

```
#include<stdio.h>
#define MAX 200
int main()
{
        if(MAX==20)
          printf("hello");
                else
                {
                        printf("U here??");
        #define MAX 20
                }
                printf("MAX is %d",MAX);  //20
                return 0;
}
```

Pre-defined Macros in C

Below program demonstrates few pre-defined constants in C. Comments in the program helps you to understand.

Program 10

```
#include<stdio.h>
int main()
{
printf("file name is %s",   FILE   );          // current file name
printf("\nDate is %s",   DATE    ); // current date
printf("\nTime is %s",   TIME   );  // time during Preprocessing
printf("\nC version is %d", STDC_VERSION );      //This macro expands to the C
//Standard version number, a long integer constant of the form yyyymmL where yyyy
//and mm are the year and month of the Standard version.
//Example: 199901L signifies the 1999 revision of the C standard
printf("\nC version is %d",   STDC   );     //In normal operation, this macro expands //to
the constant 1, to signify that this compiler conforms to ISO Standard C
printf("\nLine number is %d",  LINE  );

if( __LINE__== 12)

printf("hello");

      else

        printf("%d",__LINE__);

return 0;

}
```

Enum v/s Macros

- ✓ Macro doesn't have a type and enum constants have a type int.
- ✓ Macro is substituted at pre-processing stage and enum constants are not.
- ✓ Macro can be redefined using #define but enum constants cannot be redefined. However assignment operator on a macro results in error.

File Inclusion Directives

This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files which can be included by the user in the program

- **Header File or Standard files**: These files contain definition of pre-defined functions like printf(), scanf() etc. To work with these functions, header files must be included. Different functions are declared in different header files. These files can be added using **#include<** ***file_name*** > where file_name is the name of the header file to be included. The **'<' and '>' brackets tells the compiler to look for the file in standard directory.**

  Example: Standard I/O functions are in 'stdio.h' file whereas functions which perform string operations are in 'string.h'

- **User- defined files**: When a program becomes very large, it is a good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as **#include"*filename*"**
  Example: #include**"sort.h"**

Conditional Compilation Directives

Implies that in a particular program, all blocks of code will not be compiled. Rather, a few **blocks of code will be compiled based on the result of some condition**. Conditional Compilation in 'C' is performed with the help of the following Preprocessor Directives -> **#ifdef, #ifndef, #if, #else, #elif, #else, #endif**

Program 1: This program demonstrates #ifdef

If the identifier checked by #ifdef is defined, only then, statements followed by #ifdef will be included in the file to be compiled.

```
#include<stdio.h>

#define MAX 20

void main()

{

 #ifdef MAX//No parentheses for #ifdef

 printf("MAX  defined\n");// If #define above is removed, this line is not
            // included for compilation.

 #endif        // compulsory for #ifdef

 printf("Outside the scope of #ifdef\n");

}
```

```
Output: MAX defined

        Outside the scope of #ifdef
```

Program 2: This program demonstrates #ifdef

If the identifier checked by #ifdef is defined, only then, statements followed by #ifdef will be included for compilation.

```
#include<stdio.h>

int main()

{

  #ifdef MAX//No parentheses for #ifdef

  printf("MAX defined\n"); // not included for compilation #endif,compulsory for
                                                            //#ifdef

  printf("Outside the scope of #ifdef\n");

  return 0;

}
```

```
Output:        Outside the scope of #ifdef
```

Program 3: This program demonstrates #ifndef

If the identifier checked by #ifndef is not defined, only then, statements followed by #ifndef will be included for compilation.

```
#include<stdio.h>

int main()

{
 #ifndef MAX        //No parentheses for #ifdef

 printf("MAX not defined\n");     // If #define above is added, this line is not
//included for compilation.

 #endif         // compulsory for #ifndef

 printf("Outside the scope of #ifndef\n");

 return 0;

}
```

Output: MAX not defined

            Outside the scope of #ifndef

Program 4: This program demonstrates #if

```
#include<stdio.h>

int main()

{

 #if (2>1)       //Syntax: #if  Constant_Expression.

                 //#if 2<1. True will not be printed

 printf("True\n");

 #endif

 printf("Outside if\n");

 return 0;

}
```

Output: True

            Outside if

Program 5: This program demonstrates #if and #else

```
#include<stdio.h>

int main()

{

  #if ((2+3) < 2)

   printf("Hello\n");

  #else

   printf("Hii\n");

  #endif

  return 0;

}
```

Output: Hii

Program 6: This program demonstrates #elif

If the result of #if or #ifdef or #ifndef is not true, then the control moves to #elif Preprocessor Directive where it has another constant expression to test. If the constant expression is a non - zero value, then the statements within the scope of #elif will be included for compilation.

```
#include<stdio.h>

 int main()

{

  #if (12%5 == 0)
```

```
   printf("Multiple of 5\n");
  #elif (12 % 3 == 0)
   printf("Multiple of 3\n");
  #endif
}
```

Output: Multiple of 3

Other directives

#undef: Used to undefine an existing macro. This directive works as: #undef LIMIT. Using this statement will undefine the existing macro LIMIT. After this statement every "#ifdef LIMIT" statement will evaluate to false.

Pragma directives:

This directive is a special purpose directive and is used to turn on or off some features. This type of directives is compiler-specific i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below.

1. **#pragma startup and #pragma exit**: These directives helps us to specify the functions that are needed to run before program startup (before the control passes to main ()) and just before program exit (just before the control returns from main()).

   Program 7:

```
#include<stdio.h>
    void func1();
    void func2();
    #pragma startup func1
```

```
#pragma exit func2
void func1()
{
    printf("Inside func1()\n");
}
void func2()
{
    printf("Inside func2()\n");
}
int main()
{
   void func1();
   void func2();
   printf("Inside main()\n");
   return 0;
}
```

Output:

```
Inside func1()
Inside main()
Inside func2()
```

2. **#pragma warn Directive:** This directive is used to hide the warning message which are displayed during compilation. We can hide the warnings as shown below:
   - #pragma warn -rvl: This directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.
   - #pragma warn -par: This directive hides those warning which are raised when a function does not use the parameters passed to it.

- #pragma warn -rch: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the *return* statement in a function is unreachable.

Program 8 - To explain the working of #pragma warn directive

```
#include<stdio.h>
#pragma warn -rvl /* return value */
#pragma warn -par /* parameter never used */
#pragma warn -rch /*unreachable code */

int show(int x)
{
        // parameter x is never used in the function
        printf("PESUNIVERSITY");
        // function does not have a return statement
}
int main ()
{
        show (10);
        return 0;
}
```

Output –

```
        PESUNIVERSITY
```

Program 9 - Program to illustrate the #pragma GCC poison directive.

This directive is supported by the GCC compiler and is used to remove an identifier completely from the program. If we want to block an identifier then we can use the #pragma GCC poison directive.

```
#include<stdio.h>
#pragma GCC poison printf //as explained this cannot be performed
int main()
{
            int a=10;
  if(a==10)
{
    printf("PES University");
}
 else
            printf("bye");
            return 0;
}
```

The above program will give the below error:

```
prog.c: In function 'main':
prog.c:14:9: error: attempt to use poisoned "printf"
         printf("PES University");
         ^
prog.c:17:9: error: attempt to use poisoned "printf"
         printf("bye");
         ^
```

Program 10 -Program to illustrate #pragma pack(n), where n is the alignment in bytes, valid alignment values being 1, 2, 4 and 8. This pragma aligns members of a structure to the minimum of n and their natural alignment. Packed objects are read and written using unaligned accesses.

```
#include<stdio.h>
//pragma
//#pragma pack(n) //n=1 2 or 4
#pragma pack(1)
struct sample
{
  int a; //4 bytes
  char b; //1 byte + 3 padding
  int x;//4 byte
};
int main()
{
  printf("%lu\n",sizeof(struct sample));
  return 0;
}
```

Introduction:

It is possible to pass some values from the command line to C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code. These arguments are provided to the program after the name of the executable in command-line shell of Operating Systems

The complete declaration of main looks like this: **int main (int argc, char *argv[])**

- The function main() must have two arguments, traditionally named as **argc and argv**.
- **argv** is an array of pointers to strings which contains all the arguments passed in the command line.
- **argc** is an int whose value is equal to the number of strings to which argv points including the executable. Means it specifies the number of arguments passed in the command line.
- All the arguments which are passed **in command line are accessed as string inside the program** and must be converted to desired type using different functions.

Program 1: Printing all the command line arguments

```
#include<stdio.h>
int main (int argc, char *argv[])
{
   int i;
   if (argc < 2)
   {
      printf (“The name of the program is %s\n”, argv[0]);
   }
   else
   {
      for (i = 0;i < argc; i ++)
         printf (“Argc %d is %s\n”, i, argv[i]);
}
return 0;
}
```

Program 2: Add three numbers specified in the command line.

```
#include<stdio.h>

#include<string.h>

int main(int argc,char *argv[])          //conventionally argc and argv, Can use any variable

{

    printf("count of arguments are %d\n",argc);

// argc =4

printf("argv[0] is %s\n",argv[0]);

printf("argv[1] is %s\n",argv[1]);

printf("argv[2] is %s\n",argv[2]);

printf("argv[3] is %s\n",argv[3]);

printf("Sum is is %d\n",atoi(argv[1])+ atoi(argv[2])+ atoi(argv[3]));

// The C library function atoi(str) converts the string argument str to an integer

//(type int). If no valid conversion could be performed, it returns 0 return 0;

}
```

Program 3: Find the sum of all numbers provided in the command line.

```
#include<stdio.h> #include<stdlib.h>

int main(int argc,char *argv[])

 {

int i,sum=0;

for(i=0;i<argc;i++)

{

sum=sum+atoi(argv[i]);

}

printf("sum of command line arguments are %d\n",sum);

return 0;

}
```

Programs for you to solve:

Program to print the length of all string provided in the command line.

Sort the strings in the command line in ascending order and print the sorted array.

Sort the strings in the command line based on the ascending order their lengths and print the sorted array.

Introduction

Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are part of the environment in which a program executes and programs executed from the shell inherit all of the environment variables from the shell . The information about the context of the program under execution can be known using two ways:

1) Command line arguments
2) Environment variables

int main(int argc,char*argv[],char*envp[])

Standard environment variables

These are used for information about the user's home directory, terminal type, current locale, and so on. You can define additional variables for other purposes. The set of all environment variables that have values is collectively known as the **environment**. **Names of environment variables are case-sensitive and must not contain the character '='.** The values of environment variables can be anything that can be represented as a string. A value must not contain an embedded null character, since this is assumed to terminate the string.

Environment Access

Let us discuss few functions used to perform read and write Environment Variables. All these functions are available in **stdlib.h**

- **int setenv (const char *name, const char *value, int replace)**
  - The setenv function can be used to add a new definition to the environment. The entry with the name name is replaced by the value '**name=value**'
  - A null pointer for the value parameter is illegal
  - The value of third argument decides whether the name to be overwritten with the given value or not. If the third argument is non-zero, overwrite the name with the value. If it is 0, do not over write.
  - If the function is successful it returns 0. Otherwise the environment is unchanged and the return value is -1 and errno is set

- **char *getenv(const char *name)**
    - The getenv() function searches the environment list to find the environment variable name, and returns a pointer to the corresponding value string
    - The getenv() function returns a pointer to the value in the environment, or NULL if there is no match.

- **int putenv(char *string)**
    - The putenv() function adds or changes the value of environment variables. The argument string is of the form **name=value**. If name does not already exist in the environment, then string is added to the environment. If name does exist, then the value of name in the environment is changed to value. The string pointed to by string becomes part of the environment, so altering the string changes the environment
    - The putenv() function returns zero on success, or nonzero if an error occurs

Program2 : Demo of getenv and setenv functions

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
        char mypath[10000];
        //printf("%s\n", getenv("PATH"));
        char *path = getenv("PATH");
        /*
        printf("Before setting path is %s\n",path);             // the entire PATH
        setenv("PATH", ".", 1);
        printf("Before setting path is %s\n",path);            // new path, that is .
        */
        /*
        // But we should never change the environment variables this way. We need to concatenate
the current directory path with existing PATH.
        path = getenv("PATH");
        printf("Before setting path is %s\n",path);
```

```
	strcpy(mypath, path);
	strcat(mypath, ":.");		// concatenating current directory to mypath
						// all paths are separated with : in ubuntu
						// all paths are separated with ; in windows
	setenv("PATH", mypath, 1); // setting mypath to new mypath
						// last argument can be any non-zero value to overwrite
	path = getenv("PATH");
	printf("After setting path is %s\n",path); // the entire pvalue of PATH with ;. At the end
	*/
	/*
	// Can we create new environment variable?
	setenv("HOME","Sindhu",1);  // HOME=Sindhu
	path = getenv("HOME");
	printf("after setting %s",path);
	*/
	printf("execution is over");
	return 0;
}
```

Program2 : Demo of getenv and putenv functions

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
	char *p = getenv("PATH");
	if(p==NULL)
		printf("NOT AVAILABLE\n");
	else
	{
		/*
		// updating the existing PATH
		printf("before setting %s\n",p);
		strcat(p,";.");   // : used as seperator in ubuntu
```

```
            //putenv(PATH = p);// error   //putenv takes the string in the form of name=value
            char path_v[100000];
            strcpy(path_v,"PATH =");
            strcat(path_v,p);
            putenv(path_v);   // correct way
            p = getenv("PATH");
            printf("after setting %s\n",p);
            */
            //adding new env variable
            /*
            putenv("HOME=sindhu");
            p = getenv("HOME");
            printf("home is %s",p);
            */

        }
        return 0;
 }
```

There are two ways in which we can display all the environment variables.

1. Using an array of character pointers specified in main()

    int main(int argc,char* argv[],**char* envp[]**)

2. Using environ variable declared in stdlib.h.

    char environ** is represented as an array of strings. Each string is of the format **'name=value'.**

Program 3: Using envp

```
 #include <stdio.h>
 int main(int argc, char *argv[], char * envp[])
 {
        int i;
        for (i= 0; envp[i] != NULL; i++)
               printf("\n%d %s", i,envp[i]);
```

```
          return 0;
  }
```

Program 4: Using char environ**

```
  #include<stdio.h>
  #include<stdlib.h>
  int main()
  {
          extern char** environ;  // observe this statement
          char **p = environ;
          int i = 0;
          while((*p)  !=  NULL)
          {
                   printf("%d %s\n", i, *p);
                   p++;
                   i++;
          }
          // adding a new env variable
          putenv("HOME=sindhu");
          char **p1 = environ;
          i = 0;
          while((*p1)!= NULL)
          {
                   printf("%d %s\n",i,*p1);   // printing the environ again
                   i++;p1++;
          }
          return 0;
}
```

Introduction

A programming construct that allows programmers to pass n number of arguments to a function. Also called as var-args. The declaration of a variable length arguments function uses an ellipsis(…)(3 dots) as the last parameter.

Example: int printf(const char* format, ...); //printf takes any numer of arguments during call

Accessing the Variable length Arguments from the function body makes use of below **macros** available in **stdarg.h**

1. **va_start** -> Enables access to Variable length function arguments
2. **va_arg** -> Accesses the next variable length function argument
3. **va_end** -> Eends traversal of the variable length function arguments
4. **va_list** -> Holds the information needed by va_start, va_arg, va_end, and va_copy

Steps to access VLAs

1. Create a variable of type va_list.

   va_list va;

2. Initialize the type to point to the parameter after the last named parameter

   va_start(va, n);

3. Access each element using va_arg. Gets the value of the given type from the location and advances the variable of va_list to the location of the next parameter

   va_arg (va,double); // second argument is type of data passed in the function call

4. Break the relationship between the variable and the stack frame using the macro va_end

   va_end(va);

Program 1: sum function must add all the values passed to it.

```
#include <stdio.h>
#include <stdarg.h>        // must  be included to use  macros  from  this.
//va_start, va_end and va_arg : all these are macros. run using -E. You will get to know
double sum(int n, ...); //  ... ellipsis symbol
                                // The first argument n indicates the number of items following it
int main()
{
        printf("sum is %lf\n",sum(4,1.5,3.5,2.5,2.6));         // 5 arguments to sum
        printf("sum is %lf\n",sum(4,1.5));    // 2 arguments to the same function sum
        return 0;
}

double sum(int n,...) // ... ellipsis symbol
{
        double s = 0;
        va_list va;       // make a variable of type : va_list.
        va_start(va,n);// initialize it to point to the parameter after the last named parameter
        for(int i = 0;i<n;i++)
        {       // get the value at the location by using va_arg
                s+=va_arg(va,double);
                // second argument is type of data passed in the function call
            //va_arg: This macro gets the value of the given type from that location and
advances the variable of va_list to the location of the next parameter
                //printf("address is %p\n",va);
        }
        va_end(va);
      // At the end, break the relationship between the variable and the stack frame using the
macro va_end.
        return s;
}
```

Portable program development

A portable application is a software product designed to be easily moved from one computing environment to another. To make the part of the code to be compiled and executed, we check whether the below macros are set or not.

If **mingw in windows system**, the value of **__MINGW32__** will be 1

If **unix/Linux system** is used, the value of **__unix__** is 1

If **mac system** is used, the value of **__APPLE** is 1

The following program demonstrates the **real use of conditional compilation and pre-defined macros that makes the code portable across different platforms.**

Program:

// Use gcc –E option with this code initially to check which part of the code is provided for compilation

```
#include<stdio.h>
int main()
{
	#ifdef __MINGW32__	// No ( and )	// if mingw in windows system is used, the value of this macro is 1
		printf("windows system");
		char c,d;
		printf("enter the character");
		scanf("%c",&c);
		fflush(stdin);
		printf("enter one more character");
		scanf("%c",&d);
		printf("entered characters are %c %c",c,d);
	#elif __unix__// if unix/Linux system is used, the value of this macro is 1
		char c;
		printf("unix system");
```

```
            #include<stdio_ext.h>
            printf("enter the character");
            scanf("%c",&c);
            __fpurge(stdin);
            printf("enter one more character");
            scanf("%c",&d);
            printf("entered characters are %c %c",c,d);
      #elif __APPLE        // if MAC system is used, the value of this macro is 1
            printf("mac system");
      #else
            printf("other system");
      #endif
      return 0;
}
```