1. Difference between "int main()" and "int main(void)" in C/C++ ?
2. Program statement like below may produce different results in different compilers,so avoid to use:
   printf("%d %d %d\n", ++i, i++, i);
   As a programmer, it is never a good idea to use programming constructs whose behaviour is undefined or unspecified, such programs should always be discouraged. The output of such programs may change with compiler and/or machine.
3. it is never a good idea to use "void main()" or just "main()" as it doesn't confirm standards.
4.

```
#include <stdio.h>
#define LIMIT 100
int main()
{
  printf("%d",LIMIT);
  //removing defined macro LIMIT
  #undef LIMIT
  //Next line causes error as LIMIT is not defined
  printf("%d",LIMIT);
  return 0;
}
```

Below program will execute correctly:

```
#include <stdio.h>
#define LIMIT 1000
int main()
{
  printf("%d",LIMIT);
  //removing defined macro LIMIT
  #undef LIMIT
  //Declare LIMIT as integer again
  int LIMIT=1001;
  printf("\n%d",LIMIT);
  return 0;
}
//OUTPUT:
1000
1001
```

5.

```
#include <stdio.h>
//div function prototype
float div(float, float);
#define div(x, y) x/y
```

```
 int main()
{
//use of macro div
//Note: %0.2f for taking two decimal value after point
printf("%0.2f",div(10.0,5.0));
//removing defined macro div
#undef div
//function div is called as macro definition is removed
printf("\n%0.2f",div(10.0,5.0));
return 0;
}
 //div function definition
float div(float x, float y){
return y/x;
}
//OUTPUT:
2.00
0.50
```

6. compilation stages
    1. Pre-processing -> .i
        a. Removal of Comments
        b. Expansion of Macros
        c. Expansion of the included files.
        d. Conditional compilation
    2. Compilation -> .s (assembly level instructions)
    3. Assembly -> .o (machine level instructions)
    4. Linking -> binary/executable
       This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. *Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends.* For example, there is a code which is required for setting up the environment like passing command line arguments.

    $gcc –Wall –save-temps filename.c –o filename
7. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program. A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.

   Static Linking and Static Libraries is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, **.a** files in Linux and **.lib** files in Windows.

$ ar rcs libmylib.a libmylib.o
$ gcc -o mybin mybin.o -L. -lmylib

Following are some important points about static libraries:

1. For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application.
2. Each process gets its own copy of the code and data. where as in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger.
3. Since library code is connected at compile time, the final executable has no dependencies on the library at run-time.
4. One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.
5. One major advantage of static libraries being preferred even now “is speed”. There will be no dynamic querying of symbols in static libraries.

Dynamic linking and Dynamic Libraries
Dynamic Linking doesn’t require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, **.so** in Linux and **.dll** in Windows.

- command to check which libraries are linked in (runtime or linktime):
  - lsof -p <process_pid> | grep ' mem '

When we link an application against a shared library, the linker leaves some **stubs (unresolved symbols)** to be filled at application loading time. These stubs need to be filled by a tool called, **dynamic linker** at run time or at application loading time. Again loading of a library is of two types, static loading and dynamic loading. Don’t confuse between **static loading** vs **static linking** and **dynamic loading** vs **dynamic linking**.

- objdump :
- nm : nm -u <objfile>, to list all undefined symbols
- readelf
- Linux ELF Object File Format (and ELF Header Structure) Basics

$ gcc -shared -fPIC -o liblibrary.so library.c
$ gcc application.c -L /home/mylib/ -llibrary -o sample
$ export LD_LIBRARY_PATH=/home/mylib/:$LD_LIBRARY_PATH -> at runtime

8. Difference between static loading vs dynamic loading
Static loading:

In static loading, all of those dependent shared libraries are loaded into memory even before the application starts execution. If loading of any shared library fails, the application won't run.
A dynamic loader examines an application's dependency on shared libraries. If these libraries are already loaded into the memory, the library address space is mapped to application virtual address space (VAS) and the dynamic linker does relocation of unresolved symbols.
If these libraries are not loaded into memory (perhaps your application might be first to invoke the shared library), the loader searches in standard library paths and loads them into memory, then maps and resolves symbols.
While resolving the symbols, if the dynamic linker is not able to find any symbol (may be due to older version of shared library), the application can't be started.

Dynamic Loading:
As the name indicates, dynamic loading is about loading libraries on demand.
For example, if you want a small functionality from a shared library. Why should it be loaded at the application load time and sit in the memory? You can invoke loading of these shared libraries dynamically when you need their functionality. This is called dynamic loading. In this case, the programmer is aware of the situation 'when should the library be loaded'. The tool-set and relevant kernel provides API to support dynamic loading, and querying of symbols in the shared library.

9. Signal and interrupt are basically the same but a small distinction exists i.e interrupts are generated by the processor and handled by the kernel but signals are generated by the kernel and handled by the process.
10. Signals :
    - SIGFPE
    - SIGILL
    - Difference between SIGSEGV vs SIGBUS
    - SIGABRT
    - SIGTRAP
    - SIGSYS
11.
```
char* s = "A\0725";
printf("%s", s);
//A:5
char* s = "B\x4a";
printf("%s", s);
//BJ
```
12. What is the difference between declaration and definition of a variable/function
13.
```
#include <stdio.h>

int main()
{
    {
       int x = 10, y = 20;
       {
          printf("x = %d, y = %d\n", x, y);
          {
            int y = 40;
             x++;
             y++;
            printf("x = %d, y = %d\n", x, y);
          }

        printf("x = %d, y = %d\n", x, y);
       }
    }
    return 0;
}
// OUTPUT:
// x = 10, y = 20
// x = 11, y = 41
// x = 11, y = 20
```

14. How Linkers Resolve Global Symbols Defined at Multiple Places?
    At compile time, the compiler exports each global symbol to the assembler as either strong or weak, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols.
    Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiple defined symbols:
    Rule 1: Multiple strong symbols (with same variable name) are not allowed.
    Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.
    Rule 3: Given multiple weak symbols, choose any of the weak symbols.
    1.
    ```
    #include <stdio.h>
    int var;
    int var;
    int main()
    {
            printf("%d ", var);
            return 0;
    }
    //OUTPUT:
    // 0
    ```
    2.
    ```
    #include <stdio.h>
    ```

```
int main()
{
        int var;
        int var;
        printf("%d ", var);
        return 0;
}
//OUTPUT:
// Compilation error:
// redeclaration of ‘var’ with no linkage
```

3.

/* foo3.c */

```
#include <stdio.h>
void f(void);
int x = 15213;
int main()
{
  f();
  printf("x = %d\n", x);
  return 0;
}
```

/* bar3.c */

```
int x;
void f()
{
  x = 15212;
}
//OUTPUT:
// x = 15212
```

4.

/*a.c*/

```
#include <stdio.h>
void b(void);

int x;
int main()
{
    x = 2016;
    b();
    printf("x = %d ",x);
    return 0;
}
```

/*b.c*/

```
#include <stdio.h>

int x;

void b()
{
   x = 2017;

}
//OUTPUT
//x = 2017
```

5.

```
/*a.c*/
#include <stdio.h>
void b(void);

int x = 2016;
int y = 2017;
int main()
{
   b();
   printf("x = 0x%x y = 0x%x \n", x, y);
   return 0;
}
/*b.c*/
double x;

void b()
{
   x = -0.0;
}
//OUTPUT
// x = 0x0 y = 0x80000000
// NOTE: Linker gives "/usr/bin/ld: Warning: alignment 4 of symbol `x' in
/tmp/ccG4NP47.o is smaller than 8 in /tmp/cc29g1AT.o" warning at linking time
```

15. Invoke the linker with a flag such as the **gcc -fno-common** flag, which triggers an error if it encounters multiple defined global symbols.

16.

```
#include <stdio.h>
int var = 20;
int main()
{
    int var = var;
    printf("%d\n", var);
    return 0;
}
//OUTPUT
// <Garbage Integer values>
```

//First var is declared, then value is assigned to it. As soon as var is declared as a local variable, it hides the global variable var.

17. Which variable has the longest scope?

```
int a;
int main()
{
   int b;
   // ..
   // ..
}
int c;
// OUTPUT:
```

// Answer is a , because it is accessible everywhere. b is limited to main() c is accessible only after its declaration.

18. Types of scopes
    - File scope
    - Function scope
    - Function prototype scope
    - Block scope

```
#include <stdio.h>
int main()
{
int i;
for ( i=0; i<5; i++ )
{
   int i = 10;
   printf ( "%d ", i );
   i++;
}
return 0;
}
//OUTPUT:
//10 10 10 10 10
// both int i has different scopes
```

19. How the negative numbers are stored in memory?
20. When an array is passed as parameter to a function, The function can change values in the original array.
21. Complicated declarations in C

    steps to read complicated declarations:
    1. Convert C declaration to postfix format and read from right to left.
    2. To convert expressions to postfix, start from innermost parentheses, If innermost parenthesis is not present then start from declaration name and go right first. When first ending parenthesis encounters then go left. Once the whole parenthesis is parsed then come out from the parenthesis.
    3. Continue until the complete declaration has been parsed.

    1. **int (*fp) ();**
        - Postfix: fp * () int
        - fp is pointer to function returning int
        - e.g

```
#include <stdio.h>
int (*fp) ();
int func(void) { printf("hello\n"); }

int main()
{
        fp = func;
        (*fp)();
        //fp();    // This will also call func
        return 0;
}
```

    2. **int (*daytab) [13]**
        - Postfix: daytab * [13] int
        - daytab is pointer to an array of 13 integer
        - e.g.

```
#include <stdio.h>
int (*daytab) [13];
int arr[13] = { 1,2,3,4,5,6,7,8,9,10,11,12,13 };
int main()
{
        daytab = &arr;
        printf("arr[2] = %d\n", (*daytab)[2]);
        return 0;
}
```

    3. **void (*f[10]) (int, int)**
        - Postfix: f[10] * (int, int) void
        - f is an array of 10 pointers to function (which takes 2 arguments of type int) returning void
        - e.g

```
#include <stdio.h>
void (*f[10]) (int, int);
void func1 (int a, int b) { printf("func1 = %d, %d\n", a,b); }
void func2 (int p, int q) { printf("func2 = %d, %d\n", p,q); }
void func3 (int x, int y) { printf("func3 = %d, %d\n", x,y); }
int main()
{
        f[0] = func1;
        f[1] = func2;
        f[2] = func3;
        (*f[0])(1,2);
        (*f[1])(3,4);
        (*f[2])(5,6);
        return 0;
}
```

4. **char (*(*x())[]) ()**
   - Postfix: x() * [] * () char
   - x is a function returning pointer to array of pointers to function returning char
   - e.g.

```
#include <stdio.h>
char func1 () { return 'a'; }
char func2 () { return 'b'; }
char func3 () { return 'c'; }

char (*xarr[]) () = { func1, func2, func3 };
char (*(*x())[]) () { return &xarr; }

int main()
{
        printf("%c\n", ((*(x()))[0]) () );
        printf("%c\n", ((*(x()))[1]) () );
        printf("%c\n", ((*(x()))[2]) () );
        return 0;
}
```

5. **char (*(*x[3])())[5]**
   - Postfix: x[3] * () * [5] char
   - x is an array of 3 pointers to function returning pointer to an array of 5 char
   - e.g.

```
#include <stdio.h>

typedef char charray5[5];

charray5 carr1 = { 'a', 'b', 'c', 'd', '\0' };
```

```
charray5 carr2 = { 'q', 'w', 'e', 'r', '\0' };
charray5 carr3 = { 'x', 'y', 'z', 'w', '\0' };

charray5* func1 () { return &carr1; }
charray5* func2 () { return &carr2; }
charray5* func3 () { return &carr3; }

char (*(*x[3])())[5] = { func1, func2, func3 };

int main()
{
    printf("func1 = [%c, %c, %c, %c]\n", ((*(x[0]))())[0][0],
    ((*(x[0]))())[0][1],((*(x[0]))())[0][2], ((*(x[0]))())[0][3]);
    printf("func2 = [%c, %c, %c, %c]\n", ((*(x[1]))())[0][0],
    ((*(x[1]))())[0][1],((*(x[1]))())[0][2], ((*(x[1]))())[0][3]);
    printf("func3 = [%c, %c, %c, %c]\n", ((*(x[2]))())[0][0],
    ((*(x[2]))())[0][1],((*(x[2]))())[0][2], ((*(x[2]))())[0][3]);
    return 0;
}
```

6. **int *(*(*arr[5])()) ()**
    - Postfix: arr[5] * () * () * int
    - arr is an array of 5 pointers to function returning pointer to function returning pointer to integer
    - e.g.

```
#include <stdio.h>

int a = 1;
int b = 2;
int* func1 () { return &a; }
int* func2 () { return &b; }

int* (*funcp1()) () { return func1; }
int* (*funcp2()) () { return func2; }
int *(*(*arr[5])()) () = { funcp1, funcp2 };

int main()
{
      printf("%d\n", *(*(*arr[0])())() );
      printf("%d\n", *(*(*arr[1])())() );
      return 0;
}
```

7. **void (*bsd_signal(int sig, void (*func)(int)))(int);**
    - Postfix: bsd_signal(int sig, void (*func)(int)) * (int) void

- bsd_signal is a function that takes integer & a pointer to a function(that takes integer as argument and returns void) and returns pointer to a function(that take integer as argument and returns void)
- e.g.

```
#include <stdio.h>

void on_sig10_exit (int u) { printf("sig10 exit\n"); }
void on_sig20_exit (int u) { printf("sig20 exit\n"); }
void default_exit (int u) { printf("default exit\n"); }
void user_default_exit (int u) { printf("user default exit\n"); }

void (*exit_by) (int);

void (*bsd_signal(int sig, void (*func)(int)))(int)
{
        switch(sig) {
        case 10:
        return on_sig10_exit;
        case 20:
        return on_sig20_exit;
        default:
        if(func==NULL)
                return default_exit;
        else
                return user_default_exit;
        }
}

int main()
{
        (bsd_signal(10, NULL))(0);
        (bsd_signal(20, NULL))(0);
        (bsd_signal(30, NULL))(0);
        (bsd_signal(30, user_default_exit))(0);
        return 0;
}
```

22. **Scope Vs Linkage**
    - **Scope :** Scope of an identifier is the part of the program where the identifier may directly be accessible.
    - **Linkage :** Linkage describes how names can or can not refer to the same entity throughout the whole program or one single translation unit.

- **Translation Unit :** A translation unit is a file containing source code, header files and other dependencies. All of these sources are grouped together in a file for they are used to produce one single executable object.
- In C and C++, a program that consists of multiple source code files is compiled *one at a time*. Until the compilation process, a variable can be described by it's scope. It is only when the linking process starts, that linkage property comes into play. Thus, **scope is a property handled by compiler, whereas linkage is a property handled by linker.**
- Linkage is a property that describes how variables should be linked by the linker. Should a variable be available for another file to use? Should a variable be used only in the file declared? Both are decided by linkage. There are two types of linkage:
- **Internal linkage:** An identifier implementing internal linkage is not accessible outside the translation unit it is declared in. Any identifier within the unit can access an identifier having internal linkage. It is implemented by the keyword static. An internally linked identifier is stored in an initialized or uninitialized segment of RAM.
- **External Linkage:** An identifier implementing external linkage is visible to **every translation unit**. Externally linked identifiers are *shared* between translation units and are considered to be located at the outermost level of the program. In practice, this means that you must define an identifier in a place which is visible to all, such that it has only one visible definition. It is the default linkage for globally scoped variables and functions. Thus, all instances of a particular identifier with external linkage refer to the same identifier in the program. The keyword extern implements external linkage. When we use the keyword extern, we tell the linker to look for the definition elsewhere. Thus, the declaration of an externally linked identifier does not take up any space. Extern identifiers are generally stored in initialized/uninitialized or text segments of RAM.

23. Why do variable names not start with numbers in C ?

24. In C, static and global variables are initialized by the compiler itself. Therefore, they must be initialized with a constant value.
    Below syntax are not allowed in C, but works in C++:
    - static int *p = (int*)malloc(sizeof(p)); // p is a static variable
    - int *p = (int*)malloc(sizeof(p)); // p is a global variable

25. Format specifier
    - **short int = %hd**
    - **unsigned short int = %hu**
    - int = %d
    - unsigned int = %u
    - long int = %ld
    - unsigned long int = %lu
    - long long int = %lld
    - unsigned long long int = %llu
    - double = %lf
    - **Long double = %Lf**

26. Some data types like *char* , *short int* take less number of bytes than *int*, these data types are automatically promoted to *int* or *unsigned int* when an operation is performed on them. This is called **integer promotion.**

    1.

    ```
    #include <stdio.h>
    int main()
    {
        char a = 30, b = 40, c = 10;
        char d = (a * b) / c;
        printf ("%d ", d);
        return 0;
    }
    //OUTPUT:
    120
    ```

    At first look, the expression (a*b)/c seems to cause arithmetic overflow because signed characters can have values only from -128 to 127 (in most of the C compilers), and the value of subexpression ‘(a*b)’ is 1200 which is greater than 128. But integer promotion happens here in arithmetic done on char types and we get the appropriate result without any overflow.

    2.

    ```
    #include <stdio.h>
    int main()
    {
        char a = 0xfb;
        unsigned char b = 0xfb;

        printf("a = %c", a);
        printf("\nb = %c", b);

        if (a == b)
          printf("\nSame");
        else
          printf("\nNot Same");
        return 0;
    }
    //OUTPUT:
    a = ?
    b = ?
    Not same
    ```

a' and 'b' have the same binary representation as *char*. But when comparison operations are performed on 'a' and 'b', they are first converted to int. 'a' is a signed *char*, when it is converted to *int*, its value becomes -5 (signed value of 0xfb). 'b' is an unsigned *char*, when it is converted to *int*, its value becomes 251. The values -5 and 251 have different representations as *int*, so we get the output as "Not Same".

3.

```
#include <stdio.h>
int main()
{
  float c = 5.0;
  printf ("Temperature in Fahrenheit is %.2f", (9/5)*c + 32);
  return 0;
}
```

//OUTPUT:

37

Since 9 and 5 are integers, integer arithmetic happens in subexpression (9/5) and we get 1 as its value. To fix the above program, we can use 9.0 instead of 9 or 5.0 instead of 5 so that floating point arithmetic happens. See below program:

```
#include <stdio.h>
int main()
{
   float c = 5.0;
   printf ("Temperature in Fahrenheit is %.2f", (9.0/5)*c + 32);
   //printf ("Temperature in Fahrenheit is %.2f", (9/5.0)*c + 32); // This will also
work
   //printf ("Temperature in Fahrenheit is %.2f", (9.0/5.0)*c + 32); // This one also
work
  // printf ("Temperature in Fahrenheit is %.2f", ((float)9/5)*c + 32); // This will also
work
   return 0;
}
```

//OUTPUT:

41

27.

```
int main()
{
   void *vptr, v;
   v = 0;
   vptr = &v;
   printf("%v", *vptr);
   getchar();
```

```
    return 0;
}
```
//OUTPUT:

Compilation error

void is not a valid type for declaring variables. void * is valid though.

28.

```
#include <stdio.h>
int main()
{
    if (sizeof(int) > -1)
        printf("Yes");
    else
        printf("No");
    return 0;
}
```
//OUTPUT:

No

In C, when an integer value is compared with an unsigned it, the int is promoted to unsigned. Negative numbers are stored in 2's complement form and unsigned value of the 2's complement form is much higher than the sizeof int.

29.

```
#include<stdio.h>
int main()
{
    float x = 0.1;
    if ( x == 0.1 )
        printf("IF");
    else if (x == 0.1f)
        printf("ELSE IF");
    else
        printf("ELSE");
}
```
//OUTPUT:

ELSE IF

By default, floating point constant values use double data-type. If we declare, x as double then above program will print **IF.**

The values used in an expression are considered as double unless a ‘f’ is specified at the end. So the expression “x==0.1” has a double on the right side and a float which are stored in a single precision floating point format on the left side. In such situations, float is promoted to double. The double precision format uses more bits for precision than single precision format.

The binary equivalent of 0.1_{10} can be written as $(0.00011001100110011\ldots)_2$ which goes up to infinity. Since the precision of the float is less than the double therefore after a certain point(23 in float and 52 in double) it would truncate the result. Hence, after promotion of float into double(at the time of comparison) the compiler will pad the remaining bits with zeroes. Hence, we get different results in which the decimal equivalent of both would be different. For Instance,
In float
=> $(0.1)_{10} = (0.00011001100110011001100)_2$
In double after promotion of float ...(1)
=> $(0.1)_{10} = (0.00011001100110011001100000000000000000...)_2$
^ padding zeroes here
In double without promotion ... (2)
=> $(0.1)_{10} = (0.0001100110011001100110011001100110011001100110011001)_2$

Hence we can see the result of both equations are different.
Therefore 'if' statements can never be executed.

Note that the promotion of float to double can only cause mismatch when a value (like 0.1) uses more precision bits than the bits of single precision. For example, the following C program prints “IF”.

```
#include<stdio.h>
int main()
{
   float x = 0.5;
   if (x == 0.5)
      printf("IF");
   else if (x == 0.5f)
      printf("ELSE IF");
   else
      printf("ELSE");
}
```
//OUTPUT:
```
IF
```

Here binary equivalent of 0.5_{10} is $(0.100000\ldots)_2$

(No precision will be lost in both float and double type). Therefore if the compiler pad the extra zeroes at the time of promotion then we would get the same result in the decimal equivalent of both left and right side in comparison(x == 0.5).

30. Convert the Fractional Portion to Binary
The fractional portion of the number must also be converted to binary, though the conversion process is much different from what you're used to. The algorithm you'll used is based on performing repeated multiplications by 2, and then checking if the result is

>= 1.0. If the result is >= 1.0, then a 1 is recorded for the binary fractional component, and the leading 1 is chopped off the result. If the result is < 1.0, then a 0 is recorded for the binary fractional component, and the result is kept as-is. The recorded builds are built-up left-to-right.The result keeps getting chained along in this way until The result is exactly 1.0, or stops after some repetitive iterations.

e.g.

1. **$(0.5)_{10}$**
   0.5x2 = 1.0 ⇒ $(0.1000000)_2$ we got 1.0, so we stop here
2. **$(0.1)_{10}$**
   0.1x2 = 0.2 ⇒ 0
   0.2x2 = 0.4 ⇒ 00
   0x4x2 = 0.8 ⇒ 000
   0x8x2 = 1.6 ⇒ 0001
   0.6x2 = 1.2 ⇒ 00011
   0.2x2 = 0.4 ⇒ 000110
   0x4x2 = 0.8 ⇒ 0001100
   0x8x2 = 1.6 ⇒ 00011001
   0.6x2 = 1.2 ⇒ 000110011
   0.2x2 = 0.4 ⇒ 0001100110
   0x4x2 = 0.8 ⇒ $(00011001100)_2$ ⇒ we got repetitive pattern, so we can stop here, or repeat 1100 pattern for more bits precision in floating point

31. Some data-types facts:
    - If no data type(int, char, float, double) is given to a variable, the compiler automatically converts it to int data type.
      - e.g. short a = 5; ⇒ compiler interpret as short int a = 5
    - Signed is the default modifier for char and int data types.
      - int a = 5; ⇒ compiler interpret as signed int a = 5
    - We can't use any modifiers (signed, unsigned, short, long) in float data type.
      - signed float a = 5.0; ⇒ compiler will give error
    - Only long modifier is allowed in double data type.
      - long double a = 5.0;

32. Storage classes used to describe the **scope, visibility** and **life-time.**
33. **Register keyword**
    - We cannot obtain the address of a register variable using pointers.

```c
/* below program could give warning / error depending
upon the compiler we are using */

#include<stdio.h>

int main()
{
    register int i = 10;
    int* a = &i;
    printf("%d", *a);
    getchar();
    return 0;
}
```

- register keyword can be used with pointer variables. Obviously, a register can have an address of a memory location. There would not be any problem with the below program.

```c
#include<stdio.h>

int main()
{
    int i = 10;
    // Below statement will work fine
    register int* a = &i;
    printf("%d", *a);
    getchar();
    return 0;
}
```

- register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, register can not be used with static.

```c
#include<stdio.h>

int main()
{
    int i = 10;
    // error
    register static int* a = &i;
    printf("%d", *a);
    getchar();
    return 0;
}
```

- Register can only be used within a block (local), it can not be used in the *global* scope

```c
#include <stdio.h>

// error (global scope)
register int x = 10;

int main()
{
    // works (inside a block)
    register int i = 10;
    printf("%d\n", i);
    printf("%d", x);
    return 0;
}
```

- There is no limit on the number of register variables in a C program, but the point is compiler may put some variables in register and some not.
- As per C standard, "*The only storage-class specifier that shall occur in a parameter declaration is register.*" i.e.
  - int fun(auto int arg); // Invalid Declaration
  - int fun(extern int arg); // Invalid Declaration
  - int fun(static int arg); // Invalid Declaration
  - int fun(register int arg); // Only Valid Declaration

34. **Static variable**

- In C, static variables can only be initialized using constant literals.
  - e.g. static int i = initializer(); // where initializer() returns some constant value. But, this kind of initialization cannot work with static variables, it will give compilation error in c, but will work in c++..
  - auto int i = initializer(); / register int i = initializer(); works well in c.
- Static variables should not be declared inside a structure.
  - struct tmp {

    static int a; // This is not allowed in C whether tmp is global/local, but works in CPP if tmp is declared global

    register int b; // This is also not allowed in c/cpp.

    volatile int c; // This is allowed.

    }
  - The reason is C compiler requires the entire structure elements to be placed together (i.e.) memory allocation for structure members should be contiguous. It is possible to declare structure inside the function (stack segment) or allocate memory dynamically(heap segment) or it can be even global (BSS or data segment). Whatever might be the case, all structure members should reside in the same memory segment because the value for the structure element is fetched by counting the offset of the element from the beginning address of the structure. Separating out one member alone to a data segment defeats the purpose of static variables and it is possible to have an entire structure as static.

35. Declaration Vs Definition

- **Declaration** of a variable or function simply declares that the variable or function exists somewhere in the program, but the memory is not allocated for them. The declaration of a variable or function serves an important role - it tells the program what its type is going to be. In case of *function* declarations, it also tells the program **the arguments, their data types, the order of those arguments, and the return type of the function**. So that's all about the declaration.
- Coming to the **definition**, when we *define* a variable or function, in addition to everything that a declaration does, it also allocates memory for that variable or function. Therefore, **we can think of definition as a superset of the declaration**(or declaration as a subset of definition).
- A variable or function can be *declared* any number of times, but it can be *defined* only once. (Remember the basic principle that you can't have two locations of the same variable or function).

36. extern keyword
    - The extern keyword is used to extend the visibility of variables/functions.
    - Since functions are visible throughout the program by default, the use of extern is not needed in function declarations or definitions. Its use is implicit. i.e. when a function is declared or defined, the extern keyword is implicitly assumed. when we write.

      int foo(int arg1, char arg2);

      The compiler treats it as:

      extern int foo(int arg1, char arg2);
    - When extern is used with a variable, it's only declared, not defined

      Example:

```
// file1.h

int addition_1(int a, int b);
```

```
// file1.c

#include "file1.h"
#include <stdio.h>

int file1_var = 100;

int addition_1(int a, int b) {
    file1_var = file1_var + 100;
    printf("Inside file1.c addition(). file1_var = %d\n", file1_var);
    return a + b;
}
```

```
// file2.h

int addition_2(int a, int b);
```

```
// file2.c

#include "file2.h"
```

```
#include <stdio.h>

extern int file1_var;

int addition_2(int a, int b) {
    file1_var = file1_var + 200;
    printf("Inside file2.c addition(). file1_var = %d\n", file1_var);
    return a + b;
}
```

```
// main.c

#include "file1.h"
#include "file2.h"
#include <stdio.h>

extern int file1_var;

int main() {
    int res1 = addition_1(10, 20);
    int res2 = addition_2(30, 40);
    printf("file1_var = %d\n", file1_var);
    return 0;
}
```

```
output:

Inside file1.c addition(). file1_var = 200
Inside file2.c addition(). file1_var = 400
file1_var = 400
```

- Do you think the below program will work? Well, here comes another surprise from C standards. They say that..if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated–in other words, that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work. i.e. As an exception, when an extern variable is declared with initialization, it is taken as the definition of the variable as well.

```
extern int var = 0;
int main(void)
{
 var = 10;
 return 0;
}
```

37. **type qualifiers:** The keywords which are used to modify the properties of a variable are called type qualifiers. const and volatile.
    - **Volatile**:
        - The volatile keyword is intended to **prevent the compiler from applying any optimizations on objects** that can change in ways that cannot be determined by the compiler. i.e. The volatile keyword was devised to prevent compiler optimizations that might render code incorrect in the presence of certain asynchronous events.

- Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time.
- The system always reads the current value of a volatile object from the memory location rather than keeping its value in a temporary register at the point it is requested.
- So the simple question is, how can the value of a variable change in such a way that the compiler cannot predict.Consider the following cases for answer to this question.

1. **Memory-mapped peripheral registers**

```
/* peripheral registers access in
C without Volatile keyword */

uint8_t * p_reg = (uint8_t *)
0x1234;

// Wait for register to read
non-zero
do { ... } while (0 == *p_reg)
```

```
/* peripheral registers access in
C with Volatile keyword */

uint8_t volatile * p_reg =
(uint8_t *) 0x1234;

// Wait for register to read
non-zero
do { ... } while (0 == *p_reg)
```

```
/* equivalent assembly code with
code optimization ON */

  mov p_reg, #0x1234
  mov a, @p_reg
loop:
  ...
  bz loop
```

```
/* equivalent assembly code with
code optimization ON */

  mov p_reg, #0x1234
loop:
  ...
  mov a, @p_reg
  bz loop
```

2. **Global variables modified by an interrupt service routine outside the scope.**

```
/* Simple global variable in ISR,
may results in unexpected
behavior when optimization turned
ON */

bool gb_etx_found = false;

void main()
{
    ...
    while (!gb_etx_found)
    {
        // Wait
    }
    ...
}

interrupt void rx_isr(void)
{
    ...
    if (ETX == rx_char)
    {
```

```
/* Use Volatile global variable
in ISR, which can fix such
problems when optimization turned
ON */

volatile bool gb_etx_found =
false;

void main()
{
    ...
    while (!gb_etx_found)
    {
        // Wait
    }
    ...
}

interrupt void rx_isr(void)
{
    ...
    if (ETX == rx_char)
```

```c
        gb_etx_found = true;
    }
    ...
}
```

```c
    {
        gb_etx_found = true;
    }
    ...
}
```

3. **Global variables accessed by multiple tasks within a multi-threaded application**

```c
/* Global variable shared by
multiple tasks without Volatile
may ask for trouble when compiler
optimization turned ON */

uint8_t gn_bluetask_runs = 0;

void red_task (void)
{
    while (4 < gn_bluetask_runs)
    {
        ...
    }
    // Exit after 4 iterations of
blue_task.
}

void blue_task (void)
{
    for (;;)
    {
        ...
        gn_bluetask_runs++;
        ...
    }
}
```

```c
/* Global variable shared by
multiple tasks with Volatile to
prevent unexpected behavior when
compiler optimization turned ON
*/

volatile uint8_t gn_bluetask_runs
= 0;

void red_task (void)
{
    while (4 < gn_bluetask_runs)
    {
        ...
    }
    // Exit after 4 iterations of
blue_task.
}

void blue_task (void)
{
    for (;;)
    {
        ...
        gn_bluetask_runs++;
        ...
    }
}
```

- const
    - The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change the value of the const variable by using pointer). The result is implementation-defined if an attempt is made to change a const.

1. **Pointer to Variable :** int *ptr;

```c
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    int *ptr = &i;

    /* pointer to integer */
    printf("*ptr: %d\n", *ptr);

    /* pointer is pointing to another variable */
    ptr = &j;
    printf("*ptr: %d\n", *ptr);

    /* we can change value stored by pointer */
    *ptr = 100;
    printf("*ptr: %d\n", *ptr);

    return 0;
}
```

2. **Pointer to constant** : const int *ptr; / int const *ptr;

```c
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    /* ptr is pointer to constant */
    const int *ptr = &i;

    printf("ptr: %d\n", *ptr);
    /* error: object pointed cannot be modified
    using the pointer ptr */
    *ptr = 100;

    ptr = &j;          /* valid */
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

```c
/* Following is another example where variable i itself is constant.
*/

#include <stdio.h>

int main(void)
{
    /* i is stored in read only area*/
    int const i = 10;
    int j = 20;

    /* pointer to integer constant. Here i
    is of type "const int", and &i is of
    type "const int *".  And p is of type
    "const int", types are matching no issue */
    int const *ptr = &i;

    printf("ptr: %d\n", *ptr);

    /* error */
    *ptr = 100;

    /* valid. We call it up qualification. In
    C/C++, the type of "int *" is allowed to up
    qualify to the type "const int *". The type of
    &j is "int *" and is implicitly up qualified by
    the compiler to "const int *" */

    ptr = &j;
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

```
/* Down qualification is not allowed in C++ and may cause warnings
in C. Following is another example with down qualification. */

#include <stdio.h>

int main(void)
{
       int i = 10;
       int const j = 20;

       /* ptr is pointing an integer object */
       int *ptr = &i;

       printf("*ptr: %d\n", *ptr);

       /* The below assignment is invalid in C++, results in error
       In C, the compiler *may* throw a warning, but casting is
       implicitly allowed */
       ptr = &j;

       /*  In  C++,  it  is  called  'down  qualification'.  The  type  of
       expression &j is "const int *" and the type of ptr is "int *".
       The  assignment  "ptr  =  &j"  causes  to  implicitly  remove
       const-ness  from  the  expression  &j.  C++  being  more  type
       restrictive,  will  not  allow  implicit  down  qualification.
       However,  C++  allows  implicit  up  qualification.  The  reason
       being, const qualified identifiers are bound to be placed in
       read-only  memory  (but  not  always).  If  C++  allows  the  above
       kind of assignment (ptr = &j), we can use 'ptr' to modify the
       value of j which is in read-only memory. The   consequences
       are implementation dependent, the program may fail at runtime.
       So strict type checking helps clean code. */

       printf("*ptr: %d\n", *ptr);

       return 0;
}
```

3. **Constant pointer to variable** : int *const ptr;

```
#include <stdio.h>

int main(void)
{
   int i = 10;
   int j = 20;
/* constant pointer to integer */
   int *const ptr = &i;
```

```
    printf("ptr: %d\n", *ptr);

    *ptr = 100;     /* valid */
    printf("ptr: %d\n", *ptr);

    ptr = &j;         /* error */
    return 0;
}
```

4. **constant pointer to constant** : const int *const ptr;

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;

    /* constant pointer to constant integer */
    const int *const ptr = &i;

    printf("ptr: %d\n", *ptr);

    ptr = &j;     /* error */
    *ptr = 100;   /* error */

    return 0;
}
```

38. **const and volatile are independent i.e. it's possible that a variable is defined as both const and volatile.**
    - In C, const and volatile are type qualifiers and these two are independent. Basically, const means that the value isn't modifiable by the program. And volatile means that the value is subject to sudden change (possibly from outside the program). In fact, C standard mentions an example of valid declaration which is both const and volatile. The example is "extern const volatile int real_time_clock;" where real_time_clock may be modifiable by hardware, but cannot be assigned to, incremented, or decremented. So we should already treat const and volatile separately. Besides, these type qualifier applies for struct, union, enum and typedef as well.
39. **Storage classes questions:**

```
#include <stdio.h>

int main()
{
    static int i=5;
    if(--i){
        main();
        printf("%d ",i);
    }
    return 0;
}

//OUTPUT:
0 0 0 0
```

```c
#include<stdio.h>

int main()
{
  typedef static int *i;
  int j;
  i a = &j;
  printf("%d", *a);
  return 0;
}

//OUTPUT:
Compiler Error -> Multiple Storage classes for a.
```

Yes, typedef is a storage-class-specifier as you found in the standard. In part it's a grammatical convenience, but it is deliberate that you can either have typedef *or* one of the more "obvious" storage class specifiers.
aaa
e.g. typedef static int x;
A typedef declaration creates an alias for a type.

In a declaration static int x; the type of x is int. static has nothing to do with the type.

(Consider that if you take the address of x, &x has type int*. int *y = &x; would be legal as would static int *z = &x but this latter static affects the storage class of z and is independent of the storage class of x.)

If something like this were allowed the static would have no effect as no object is being declared. The type being aliased is just int.

```c
#include<stdio.h>

int main()
{
  typedef int *i;
  int j = 10;
  i *a = &j;
  printf("%d", **a);
  return 0;
}

//OUTPUT:
Compilation error -> Initialization with incompatible pointer type
```

```c
#include <stdio.h>

int fun()
{
  static int num = 16;
  return num--;
}

int main()
{
  for(fun(); fun(); fun())
    printf("%d ", fun());
  return 0;
}

//OUTPUT:
14 11 8 5 2
```

```
#include <stdio.h>

int main()
{
  int x = 10;
  static int y = x;

  if(x == y)
     printf("Equal");
  else if(x > y)
     printf("Greater");
  else
     printf("Less");
  return 0;
}

//OUTPUT:
Compilation error
```

```
#include <stdio.h>

char *fun()
{
    static char arr[1024];
    return arr;
}

int main()
{
    char *str = "geeksforgeeks";
    strcpy(fun(), str);
    str = fun();
    strcpy(str, "geeksquiz");
    printf("%s", fun());
    return 0;
}

//OUTPUT:
geeksquiz
```

```
#include "stdio.h"

void foo(void)
{
 static int staticVar;
 staticVar++;
 printf("foo: %d\n",staticVar);
}

void bar(void)
{
 static int staticVar;
 staticVar++;
 printf("bar: %d\n",staticVar);
}

int main()
{
 foo(), bar(), foo();
 return 0;
}

//OUTPUT:
foo: 1
bar: 1
foo: 2
```

40. printf and scanf

- printf() returns **total number of characters printed**, Or negative value if an output error or an encoding error.

```
#include <stdio.h>

int main()
{
    char st[] = "CODING";
    long int n = 123456789;
    printf("%d\n", printf("%s", st));
    printf("%d\n", printf("%d", n));
    return 0;
}
//OUTPUT:
CODING6
1234567899
```

- scanf() returns the total **number of Inputs Scanned successfully**, or EOF if input failure occurs before the first receiving argument was assigned.

```
#include <stdio.h>

int main()
{
    char a[100], b[100], c[100];

    // scanf() with one input
    printf("\n First scanf() returns : %d",
                        scanf("%s", a));

    // scanf() with two inputs
    printf("\n Second scanf() returns : %d",
                   scanf("%s%s", a, b));

    // scanf() with three inputs
    printf("\n Third scanf() returns : %d",
              scanf("%s%s%s", a, b, c));

    return 0;
}

//OUTPUT:
First scanf() returns : 1
Second scanf() returns : 2
Third scanf() returns : 3
```

41. In C, return type of getchar(), fgetc() and getc() is int (not char). So it is recommended to assign the returned values of these functions to an integer type variable. e.g.

```c
/* getchar() example */
/* syntax : int getchar(void); */

#include <stdio.h>

int in;
while ((in = getchar()) != EOF)
{
   putchar(in);
}
```

```c
/* fgetc() example */
/* syntax : int fgetc(FILE *stream); */

#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   if(fp == NULL) {
      perror("Error in opening file");
      return(-1);
   }
   do {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   } while(1);

   fclose(fp);
   return(0);
}
```

```c
/* getc() example */
/* syntax : int getc(FILE *stream) */

#include <stdio.h>

int main () {
   int c;

   printf("Enter character: ");
   c = getc(stdin);
   printf("Character entered: ");
   putc(c, stdout);

   return(0);
}
```

42. **Scansets in C**

- scanf family functions support scanset specifiers which are represented by %[]. Inside the scanset, we can specify a single character or range of characters. While processing scanset, scanf will process only those characters which are part of scanset. We can define scanset by putting characters inside square brackets. Please note that the scansets are case-sensitive.
- We can also use scanset by providing a comma in between the character you want to add. e.g. scanf(%s[A-Z,_,a,b,c], str);
- If the first character of the scanset is '^', then the specifier will stop reading after the first occurrence of that character.
- some examples:
  - scanf("%[A-Z]s", str);
    - scan A to Z characters only, terminate scanning as soon as it receives different character
    - If input = "HELLOworld", it scan only up to "HELLO"
  - scanf("%[^o]s", str);
    - read all characters but stops after first occurrence of 'o'
    - If input = "Hello world", it scan only up to "Hell"

43. Never use gets().
    - Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

```
void read()
{
   char str[20];
   gets(str);
   printf("%s", str);
   return;
}
```

- The code looks simple, it reads the string from standard input and prints the entered string, but it suffers from Buffer Overflow as gets() doesn't do any array bound testing. gets() keeps on reading until it sees a newline character.
- To avoid Buffer Overflow, fgets() should be used instead of gets() as fgets() makes sure that not more than MAX_LIMIT characters are read.

```
#define MAX_LIMIT 20
void read()
{
   char str[MAX_LIMIT];
   fgets(str, MAX_LIMIT, stdin);
   printf("%s", str);

   getchar();
   return;
}
```

- gets() can read a string with spaces but a normal scanf() with %s can not.
- **NOTE:** fgets() stores the '\n' character if it is read, so removing that has to be done explicitly by the programmer. It is hence, generally advised that your str can store at least (MAX_LIMIT + 1) characters if your intention is to keep the newline character.

This is done so there is enough space for the null terminating character '\0' to be added at the end of the string.

44. puts() can be preferred for printing a string because it is generally less expensive (implementation of puts() is generally simpler than printf()), and if the string has formatting characters like '%s', then printf() would give unexpected results. Also, if str is a user input string, then use of printf() might cause security issues. Also note that puts() moves the cursor to the next line. If you do not want the cursor to be moved to the next line, then you can use fputs().

```
/* int fputc(int c, FILE *stream);          */
/* int fputs(const char *s, FILE *stream);  */
/* int putc(int c, FILE *stream);           */
/* int putchar(int c);                      */
/* int puts(const char *s);                 */

#include <stdio.h>

int main()
{
    puts("HelloWorld");
    fputs("HelloWorld\n", stdout);
    puts("Hello%sWorld%s");
    printf("Hello%sWorld%s");
    getchar();
    return 0;
}

//OUTPUT:
HelloWorld
HelloWorld
Hello%sWorld%s
<may be segmentation fault or some garbage prints>
```

45. Use of %n in printf()

- In C printf(), %n is a special format specifier which instead of printing something causes printf() to load the variable pointed by the corresponding argument with a value equal to the number of characters that have been printed by printf() before the occurrence of %n.

```
#include <stdio.h>

int main()
{
  int c;
  printf("hello%nworld ", &c);
  printf("%d", c);
  getchar();
  return 0;
}

//OUTPUT
helloworld 5
```

46. Amazing printf/scanf

```c
#include <stdio.h>
// Assume base address of "GeeksQuiz" to be 1000
int main()
{
   printf(5 + "HelloWorld");
   return 0;
}

//OUTPUT:
World
```

```c
#include <stdio.h>

int main()
{
    printf("%c ", 5["HelloWorld"]);
    printf("%c ", "HelloWorld"[5]);
    return 0;
}

//OUTPUT:
W
W
```

```c
scanf("%4s", str);  // Read maximum 4 characters from console.
```

```c
#include <stdio.h>

int main()
{
       char *s = "Hello world";
       int n = 7;
       printf("%.*s", n, s);
       return 0;
}

//OUTPUT:
Hello w
```

```c
#include <stdio.h>
int main(void)
{
   int x = printf("HelloWorld");
   printf("%d", x);
   return 0;
}

//OUTPUT:
HelloWorld10
```

```c
#include <stdio.h>

int main()
{
    printf("%d", printf("%d", 1234));
    return 0;
}
//OUTPUT:
12344
```

47. Predict the output:

```
int fun(char *str1)
{
  char *str2 = str1;
  while(*++str1);
  return (str1-str2);
}

int main()
{
  char *str = "GeeksQuiz";
  printf("%d", fun(str));
  return 0;
}

//OUTPUT:
9
```

```
#include <stdio.h>

int main()
{
 int a = 10;
 int b = 15;

 printf("%d",(a+1),(b=a+2));
 printf("%d",b);

 return 0;
}

//OUTPUT:
1112

/* As per C standard C11, all the arguments of printf() are evaluated
irrespective of whether they get printed or not. That's why (b=a+2) would also
be evaluated and value of b would be 12 after first printf(). */
```

```
#include <stdio.h>

int foo(int a)
{
    printf("%d",a);
    return 0;
}

int main()
{
    foo;
    return 0;
}

/* No compile error but foo function wouldn't be executed. The program
wouldn't print anything.*/
```

```
#include <stdio.h>
struct Ournode {
  char x, y, z;
};
```

```
int main() {
  struct Ournode p = {'1', '0', 'a' + 2};
  struct Ournode *q = &p;
  printf("%c, %c", *((char *)q + 1), *((char *)q + 2));
  return 0;
}

//OUTPUT:
0 c
```

48. Difference between %d and %i
- %d specifies signed decimal integer while %i specifies integer
- There is no difference between the %i and %d format specifiers for printf, but **%d and %i behavior is different in scanf.**
- Therefore, both specifiers behave differently while they are used with an input specifier. So, 012 would be 10 with %i but 12 with %d.
  - **%d** takes integer value as signed decimal integer i.e. it takes negative values along with positive values but values should be in decimal otherwise it will print garbage value.
  - **%i** takes integer value as integer value with decimal, hexadecimal or octal type. To enter a value in hexadecimal format – value should be provided by preceding "0x" and value in octal format – value should be provided by preceding "0".

```
#include <stdio.h>

int main()
{
    int d, i;

    printf("Enter value(scanning with %%d): ");
    scanf("%d", &d);
    printf("Enter same value(scanning with %%i): ");
    scanf("%i", &i);

    printf("scanned with %%d, printed with %%d - %d\n", d);
    printf("scanned with %%d, printed with %%i - %i\n", d);
    printf("scanned with %%i, printed with %%d - %d\n", i);
    printf("scanned with %%i, printed with %%i - %i\n", i);

    return 0;
}

OUTPUT:
Enter value(scanning with %d): 012
Enter same value(scanning with %i): 012
scanned with %d, printed with %d - 12
scanned with %d, printed with %i - 12
scanned with %i, printed with %d - 10
scanned with %i, printed with %i - 10
```

49. scanf and fscanf scanning trick

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

```
int main()
{
    FILE* ptr = fopen("abc.txt","r");
    if (ptr==NULL)
    {
        printf("no such file.");
        return 0;
    }

    /* Assuming that abc.txt has content in below
       format
       NAME    AGE   CITY
       abc     12    hyderabad
       bef     25    delhi
       cce     65    bangalore */
    char buf[100];
    while (fscanf(ptr,"%*s %*s %s ",buf)==1)
        printf("%s\n", buf);

    return 0;
}

//OUTPUT:

CITY
hyderabad
delhi
bangalore
```

50. isprint(), isctrl(), rand(),srand()

- **How srand() and rand() are related to each other?**
  - srand() sets the seed which is used by rand to generate "random" numbers. If you don't call srand before your first call to rand, it's as if you had called srand(1) to set the seed to one. In short, srand() — set seed for rand() function.
- The **isprint()** function checks whether a character is a printable character or not. isprint() function takes a single argument in the form of an integer and returns a value of type int. We can pass a char type argument internally; it acts as an int by specifying ASCII value.
- The **iscntrl()** function is used to check whether a character is a control character or not. iscntrl() function also takes a single argument and returns an integer.

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

int main(void)
{
    char ch = 'a';
    if (isprint(ch)) {
        printf("%c is printable character\n", ch);
    } else {
        printf("%c is not printable character\n", ch);
    }

    if (iscntrl(ch)) {
        printf("%c is control character\n", ch);
    } else {
        printf("%c is not control character", ch);
    }
    return (0);
}

//OUTPUT:
a is printable character
a is not control character
```

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

int main(void)
{
    // This program will create same sequence of
    // random numbers on every program run

    for(int i = 0; i<5; i++)
        printf(" %d ", rand());

    return 0;
}

//OUTPUT:
453 1276 3425 89
453 1276 3425 89
```

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

// Driver program
int main(void)
{
    // This program will create different sequence of
    // random numbers on every program run

    // Use current time as seed for random generator
    srand(time(0));

    for(int i = 0; i<4; i++)
        printf(" %d ", rand());

    return 0;
}

//OUTPUT:
453 1432 325 89
8976 21234 45 8975
```

51. Operator precedence and associativity
- **Operator precedence** determines which operator is performed first in an expression with more than one operators with different precedence.
- **Operators Associativity** is used when two operators of the same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.
- Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets.
- Associativity is only used when there are two or more operators of the same precedence.

```
#include <stdio.h>

int x = 0;

int f1()
{
    x = 5;
    return x;
}

int f2()
{
    x = 10;
    return x;
}

int main()
{
    int p = f1() + f2();
    printf("%d ", x);
    return 0;
}

//OUTPUT
10 or 5 (compile dependent)
```

- All operators with the same precedence have the same associativity.
- Comma has the least precedence among all operators and should be used carefully.

```
/* comma as an operator */
int i = (5, 10); /* 10 is assigned to i*/
int j = (f1(), f2()); /* f1() is called (evaluated) first followed by f2().
                      The returned value of f2() is assigned to j */
```

The below program fails in compilation

```
#include<stdio.h>

int main(void)
{
    int a = 1, 2, 3;
    printf("%d", a);
    return 0;
}
```

But, below program compiles successfully,

```
#include <stdio.h>

int main()
{
    int a;
    a = 1, 2, 3; // Evaluated as (a = 1), 2, 3
    printf("%d", a);
    return 0;
}

//OUTPUT
1
```

Now, look at the below program.

```
#include<stdio.h>
int main(void)
{
    int a;
    a = (1, 2, 3);
    printf("%d", a);
    return 0;
}

//OUTPUT:
3
```

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = 15;

    printf("%d", (x, y));
    getchar();
    return 0;
}

//OUTPUT:
15
```

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = (x++, ++x);
    printf("%d", y);
    getchar();
    return 0;
}

//OUTPUT:
12
```

```
#include<stdio.h>

int main()
{
  int a = 10, b = 20;
  (a, b) = 30; // Since b is l-value, this statement is valid in C++, but
not in C.
  printf("b = %d", b);
  getchar();
  return 0;
}

//OUTPUT:
30 (in c++)
```

- There is no chaining of comparison operators in C

```
#include <stdio.h>

int main()
```

```
{
    int a = 10, b = 20, c = 30;

    // (c > b > a) is treated as ((c  > b) > a), associativity of '>'
    // is left to right. Therefore the value becomes ((30 > 20) > 10)
    // which becomes (1 > 20)
    if (c > b > a)
        printf("TRUE");
    else
        printf("FALSE");
    return 0;
}

//OUTPUT:
FALSE
```

52. Precedence and Associativity table

OperatorDescription

Associativity

( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ –	Postfix increment/decrement (see Note 2)	
++ –	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right

53. sizeof()

- When *sizeof()* is used with the expression, it returns size of the expression

```
#include <stdio.h>

int main()
{
    int a = 0;
    double d = 10.21;
    printf("%lu", sizeof(a + d));
    return 0;
}


//OUTPUT:
8
```

- Sizeof can be used to calculate the number of elements of the array automatically.

```
#include <stdio.h>

int main()
{
    int arr[] = { 1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14 };
    printf("Number of elements:%lu ", sizeof(arr) / sizeof(arr[0]));
    return 0;
}

//OUTPUT:
Number of elements: 11
```

54. Precedence of postfix and prefix operator

```
#include <stdio.h>

int main()
{
    char arr[] = "HelloWorld";
    char *p = arr;
    ++*p;
    printf(" %c", *p);
    return 0;
}

//OUTPUT:
I
```

```
#include <stdio.h>

int main()
{
  char arr[] = "geeksforgeeks";
  char *p = arr;
  *p++;
  printf(" %c", *p);
  return 0;
}

//OUTPUT:
e
```

```
#include <stdio.h>

int main()
{
    int a = 10;

    ++a = 20; // works in C++, but error in C
    //a++ = 20; will give error in C and C++
    printf("a = %d", a);
    getchar();
    return 0;
}

//OUTPUT:
In C, above program will give error
But in C++, it will print 20 as output, because In C++, pre-increment (or
pre-decrement) can be used as l-value, but post-increment (or post-decrement)
can not be used as l-value.
```

```
#include <stdio.h>

int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    ++*p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d",
                          arr[0], arr[1], *p);
    return 0;
}

//OUTPUT:
arr[0] = 11, arr[1] = 20, *p = 11
```

```
#include <stdio.h>

int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    *p++;
    printf("arr[0] = %d, arr[1] = %d, *p = %d",
                          arr[0], arr[1], *p);
    return 0;
}

//OUTPUT:
arr[0] = 10, arr[1] = 20, *p = 20
```

```
#include <stdio.h>

int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    *++p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d",
                          arr[0], arr[1], *p);
    return 0;
}

//OUTPUT:
```

```
arr[0] = 10, arr[1] = 20, *p = 20
```

```
/* Execution of printf with ++ operators */

printf("%d %d %d", i, ++i, i++);

//The above invokes undefined behaviour by referencing both 'i' and 'i++' in
the argument list. It is not defined in which order the arguments are
evaluated. Different compilers may choose different orders. A single compiler
can also choose different orders at different times.
```

55. Modulus of negative numbers

```
#include <stdio.h>

int main()
{
   int a = 3, b = -8;
   printf("%d", a % b);
   return 0;
}

//OUTPUT:
3
```

```
#include <stdio.h>

int main()
{
   int a = -3, b = 8;
   printf("%d", a % b);
   return 0;
}

//OUTPUT:
-3
```

```
#include <stdio.h>

int main()
{
   int a = -3, b = -8;
   printf("%d", a % b);
   return 0;
}

//OUTPUT:
-3
```

```
#include <stdio.h>

int main()
{
   int a = -8, b = -3;
   printf("%d", a % b);
   return 0;
}

//OUTPUT:
```

```
-2
```

56. The return type of ternary operator depends on exp_2, and *convertibility* of exp_3 into exp_2 as per usual\overloaded conversion rules. If they are not convertible, the compiler throws an error.

```
#include <stdio.h>

int main()
{
   int test = 0;
   printf("character %d\n", test ? 0 : '1');
   return 0;
}

//OUTPUT:
49
```

```
#include <stdio.h>

int main()
{
   int test = 0;
   printf("string %s\n", test ? "1asdasd": 1);

   return 0;
}

//OUTPUT:
Compilation successful with Warning, but output will be
Some Garbage value, or segfault
```

57.

```
#include <stdio.h>

int main()
{
    int x = 10, y = 10;
    printf("%d \n", sizeof(x == y));
    printf("%d \n", sizeof(x < y));
    return 0;
}

//OUTPUT:
In C, data type of result of comparison operations is int, above program will
print below output.
4
4
Whereas in C++, type of results of comparison operations is bool, so output
will be.
1
1
```

58. To find sum of two numbers without using any operator (*using width field in printf()*)

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

```c
int add(int x, int y)
{
    return printf("%*c%*c", x, '\r', y, '\r');
    //return printf("%*c%*c", x, ' ', y, ' '); //also works but print spaces
to
}

int main()
{
    printf("Sum = %d", add(3, 4));
    return 0;
}

//OUTPUT:
Sum = 7
```

59. Random C programs

```c
#include <stdio.h>

void fun(void)
{
    printf("HelloWorld");
}

int main()
{
    fun();
    while(1);
}

//OUTPUT:
<stuck in loop, no output - press ctrl+c to exit>. Why ? It's because printf
won't print any value on stdout, until it receives "\n" or "End of Program".
```

```c
#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = "November";
    char b[] = {'H','e','l','l','o'};
    printf("Length of String is %u\n", strlen(str));
    printf("Size of String is %u\n", sizeof(str));
    printf("Length of b is %u\n", strlen(b));
    printf("Size of b is %u\n", sizeof(b)); // The strlen function looks for a
null character and behaves abnormally if it doesn't find it.
}

//OUTPUT:
Length of String is 8
Size of String is 9
Length of b is 5
Size of b is <random number until it finds null>

strlen() searches for that NULL character and counts the number of elements
present in the string before the NULL character, here which is 8.

sizeof() operator returns the actual amount of memory allocated for the
operand passed to it. Here the operand is an array of characters which
contains 9 characters including Null character and size of 1 character is 1
```

```
byte. So, here the total size is 9 bytes.
```

60. In C/C++ sizeof() operator is used to find size of a date type or variable. Expressions written in sizeof() are never executed.

```
#include <stdio.h>

int main(){

    // The printf in sizeof is not executed
    // Only the return type of printf is
    // considered and its size is evaluated
    // by sizeof,
    int a = sizeof(printf("HelloWorld\n"));

    printf("%d", a);

    return 0;
}

//OUTPUT:
4
```

```
/* Even if we assign a value inside sizeof(),
the changes are not reflected. */

#include <stdio.h>

int main() {
    int a = 5;
    int b = sizeof(a = 6);
    printf("a = %d,  b = %d\n", a, b);
    return 0;
}

//OUTPUT
a = 5, b = 4
```

61. Stringizing and Taken pasting operator

```
/* Stringizing operator (#) - causes the corresponding actual argument to be
enclosed in double quotation marks (turns the argument it precedes into a
quoted string) */

#include <stdio.h>

#define mkstr(s) #s

int main(void)
{
    printf(mkstr(HelloWorld));
    return 0;
}

//OUTPUT
HelloWorld
```

```
/* Token-pasting operator (##) - allows tokens used as actual arguments to be
concatenated to form other tokens */

#include <stdio.h>

#define concat(a, b) a##b

int main(void)
{
    int xy = 30;
    printf("%d", concat(x, y));
     printf("%d", concat(x     ,     y)); //surrounding whitespaces are removed
    return 0;
}

//OUTPUT
30
30
```

62. Variable length arguments for Macros

```
#include <stdio.h>

#define INFO     1
#define ERR      2
#define STD_OUT  stdout
#define STD_ERR  stderr

#define LOG_MESSAGE(prio, stream, msg, ...) do {\
                        char *str;\
                        if (prio == INFO)\
                            str = "INFO";\
                        else if (prio == ERR)\
                            str = "ERR";\
                        fprintf(stream, "[%s] : %s : %d : "msg" \n", \
                                str, __FILE__, __LINE__, ##__VA_ARGS__);\
                    } while (0)

int main(void)
{
    char *s = "Hello";

        /* display normal message */
    LOG_MESSAGE(ERR, STD_ERR, "Failed to open file");

    /* provide string as argument */
    LOG_MESSAGE(INFO, STD_OUT, "%s Hello World", s);

    /* provide integer as arguments */
    LOG_MESSAGE(INFO, STD_OUT, "%d + %d = %d", 10, 20, (10 + 20));

    return 0;
}

//OUTPUT:
[ERR] : a.c : 23 : Failed to open file
[INFO] : a.c : 26 : Hello Hello World
[INFO] : a.c : 29 : 10 + 20 = 30
```

```
/* Have you noticed the do {...} (while 0) in the above program. Ever wonder
why it is required ? we can’t put simple {...} parenthesis. See below example
*/

#include <stdio.h>

#define MACRO(num, str) {\
            printf("%d", num);\
            printf(" is");\
            printf(" %s number", str);\
            printf("\n");\
           }

int main(void)
{
    int num = 5;

    if (num & 1)
        MACRO(num, "Odd");
    else
        MACRO(num, "Even");

    return 0;
}

//OUTPUT:
error: ‘else’ without a previous ‘if’
i.e.
if (num & 1)
{
    -------------------------
    ---- Macro expansion ----
    -------------------------
};    /* Semicolon at the end of MACRO, and here is ERROR */

We have ended macro with semicolon. When the compiler expands the macro, it
puts a semicolon after the “if” statement. Because of the semicolon between
the “if and else statement” compiler gives compilation error. Above program
will work fine, if we ignore the “else” part.

To overcome this limitation, we can enclose our macro in “do-while(0)”
statement. Our modified macro will look like this.

#define MACRO(num, str) do {\
            printf("%d", num);\
            printf(" is");\
            printf(" %s number", str);\
            printf("\n");\
           } while(0)

Similarly, instead of the “do – while(0)” loop we can enclose multi-line macro
in parenthesis. We can achieve the same result by using this trick

#define MACRO(num, str) ({\
            printf("%d", num);\
            printf(" is");\
            printf(" %s number", str);\
            printf("\n");\
           })
```

63. Asdaasdasd

64.

Questions/topics:

1. How to load dynamic libraries without restarting the application ?
   - https://stackoverflow.com/questions/10001013/update-shared-libraries-without-restarting-processes
   - Application should be designed in a way to use dynamic loading UNIX APIs dlopen, dlsym & dlclose(). But how can app be aware of lib updates?
   - https://backtrace.io/blog/backtrace/elf-shared-library-injection-forensics/
2. What is position independent code (PIC)?
   - Position-independent code (PIC) is code that being placed somewhere in the primary memory, and can execute regardless of its absolute address (i.e. by using relative addressing). PIC is used for shared libraries, allowing library code to be located anywhere in memory
3. What is use of /etc/ld.so.conf.d/* ?
   - By default, the dynamic loader searches through /lib and /usr/lib for dynamic libraries that are needed by programs. /etc/ld.so.conf can be used to configure the dynamic loader to search for other directories (such as /usr/local/lib or /opt/lib) as well.
4.

TODO:

1. Format String Vulnerabilities
   http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Format_String/files/formatstring-1.2.pdf
2. Binary, Octal, Hex or any number format conversions
3.