C Programming Skill Test

TABLE OF CONTENTS

1. <u>Various compilation stages, compiler errors</u>

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the below-mentioned errors, and if the source code is error-free, then it generates the object code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for successful execution of the program.

There are mainly five types of errors that exist in C programming:

- Syntax error
- Run-time error
- Linker error
- Logical error
- Semantic error

Syntax error

Syntax errors are also known as compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors mainly occurred due to mistakes while typing or do not follow the syntax of the specified programming language. These errors can be easily debugged or corrected.

Commonly occurred syntax errors are:

- If we miss the parenthesis (}) while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon (;) at the end of the statement.
- If we use c keywords without considering its case sensitiveness.

Run-Time error

Sometimes the errors exist during the execution time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is a common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

Linker error

Linker errors are mainly generated when the executable file of the program is not created. This can happen either due to the wrong function prototyping or usage of the wrong header file. For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in the func.c file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main().**

Logical error

The logical error is an error that leads to an undesired output. These errors produce incorrect output, but they are error-free, known as logical errors. If we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

Semantic error

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

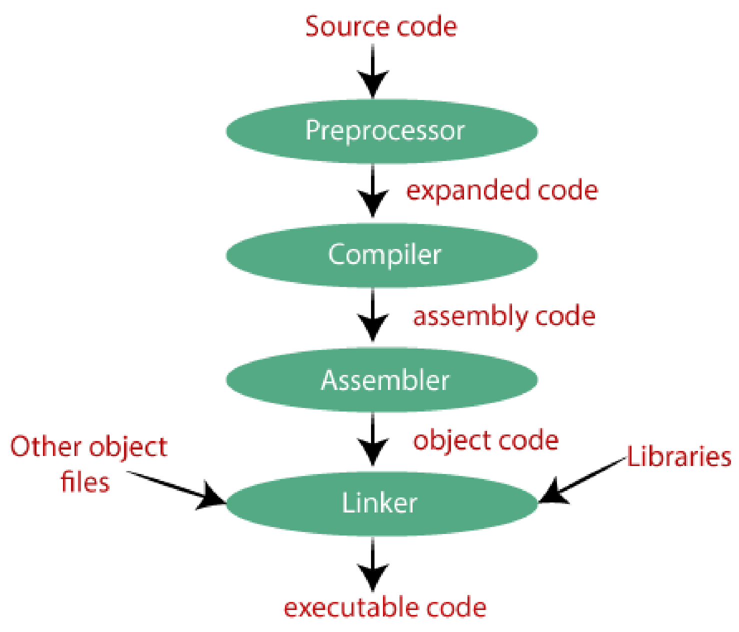
- Array index out of bound

```
int a[10];

a[10] = 34;
```

- Errors in expressions(we use the statement a+b =c, which is incorrect as we cannot use the two operands on the left-side.)

```
int a, b, c;
a+b = c;
```

2. Compiling overview and process



By executing the below command, We get all intermediate files in the current directory along with the executable.

$gcc –Wall –save-temps filename.c –o filename

The option **-Wall** enables all compiler's **warning** messages. This option is recommended to generate better code. The option **-o** is used to specify the **output file** name. If we do not use this option, then an output file with the name a.out is generated.

Preprocessor

The source code is the code that is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler. The preprocessed output is stored in the **filename.i**.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code. The next step is to compile **filename.i** and produce an; intermediate compiled output file **filename.s**. This file is in assembly-level instructions.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is **'filename.c',** then the name of the object file would be '**filename.obj**'.In this phase the **filename.s** is taken as input and turned into **filename.o** by an assembler. This file contains machine-level instructions.

Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with the '.lib' (or '.a') extension. The main work of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'.

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. The 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 that is required for setting up the environment like passing command-line arguments. This task can be easily verified by using **$size filename.o** and **$size filename**. Through these commands, we know how the output file increases from an object file to an executable file. This is because of the extra code that the linker adds to our program.

3. Compiling multiple source files from the command line

Here is an example of the one-step process, which will compile and link test1.c, test2.c, and test3.c into an executable file called test; the -o stands for *output*:

```
gcc -o test1 test1.c test2.c test3.c
```

This series of commands will compile the files individually, then link the results into an executable file; the -c stands for *compile only*, and causes an object file (.o) to be produced:

```
gcc -c test1.c

gcc -c test2.c

gcc -c test3.c

gcc -o test1 test1.o test2.o test3.o
```

4. Makefiles (Understand how to write your own Makefile)

If you have multiple source files in c, c++, and other languages and want to compile them from Terminal Command, it is hard to write every time. To solve such a problem, we use **Makefile** because during the compilation of large projects we need to write numbers of source files as well as linker flags that are not so easy to write again and again.

Makefile is a tool to simplify or organize code for compilation. **Makefile** is a set of commands (similar to terminal commands) with variable names and targets to create object files and to remove them. In a single make file, we can create multiple targets to compile and to remove the object, binary files. You can compile your project (program) any number of times by using **Makefile**.

Syntax of makefiles:- The difference between a space and a tab. *All rules must be on lines that start with a tab*; a space won't do. Because several spaces and a tab look much the same and because almost everywhere else in Linux programming there's little distinction between spaces and tabs, this can cause problems. Also, a space at the end of a line in the makefile may cause a make command to fail.

Comments in a Makefile:- A comment in a makefile starts with # and continues to the end of the line.

Macros in a Makefile:- Even if this was all there was to make and makefiles, they would be powerful tools for managing multiple source file projects. However, they would also tend to be large and inflexible for projects consisting of a very large number of files. Makefiles, therefore, allow you to use macros so that you can write them in a more generalized form. You define a macro in a makefile by writing MACRONAME=value, then accessing the value of MACRONAME by writing either $(MACRONAME) or ${MACRONAME}. Some versions of make may also accept $MACRONAME. You can set the value of a macro to a blank (which expands to nothing) by leaving the rest of the line after the = blank.

Macro Definition

$?	List of prerequisites (files the target depends on) changed more recently than the current targe
$@	Name of the current target
$<	Name of the current prerequisite
$*	Name of the current prerequisite, without any suffix

Multiple Targets:-

It's often useful to make more than a single target file or to collect several groups of commands into a single place. You can extend your makefile to do this. You add a clean option that removes unwanted objects and an install option that moves the finished application to a different directory.

```
#Multiple targets

all: myapp
# Which compiler
CC = gcc
# Where to install
INSTDIR = /usr/local/bin
```

```
# Where are include files kept
INCLUDE = .
# Options for development
CFLAGS = -g -Wall –ansi
# Options for release
# CFLAGS = -O -Wall –ansi
myapp: main.o 2.o 3.o
        $(CC) -o myapp main.o 2.o 3.o
main.o: main.c a.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
2.o: 2.c a.h b.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
3.o: 3.c b.h c.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
clean:
        -rm main.o 2.o 3.o
install: myapp
@if [ -d $(INSTDIR) ]; \
        then \
        cp myapp $(INSTDIR);\
        chmod a+x $(INSTDIR)/myapp;\
        chmod og-w $(INSTDIR)/myapp;\
        echo “Installed in $(INSTDIR)“;\
else \
        echo “Sorry, $(INSTDIR) does not exist”;\
fi
```

- make all – It compiles everything so that you can do local testing before installing applications.
- make install – It installs applications in the right places.
- make clean – It cleans applications, gets rid of the executables, any temporary files, object files, etc.

<u>5. Communication between files</u>

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */

/**
```

```
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

main.c

```
#include "foo.h"

int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Compile and Link

First, we compile both foo.c and main.c to object files. Here we use the gcc compiler.

```
$ gcc -c foo.c
$ gcc -c main.c
```

Now we link them together to produce our final executable:

```
$ gcc -o testprogram foo.o main.o
```

```
##########################################
MAKEFILE :
    CC = gcc
    testprogram: main.o foo.o
        $(CC) main.o foo.o -o testprogram
    main.o: main.c foo.h
        $(CC) -c main.c
    foo.o: foo.c foo.h
        $(CC) -c foo.c
##########################################
```

6. Using Header files effectively

Rule #1. Each module with its .h and .c file should correspond to a clear piece of functionality. The Standard Library modules math.h and string.h are good examples of clearly distinct modules.

Rule #2. Always use “include guards” in a header file. The most compact form uses #ifndef. For example “abc_xyz.h” would start with:

```
    #ifndef ABC_XYZ_H
    #define ABC_XYZ_H
```

and end with:

```
    #endif
```

Rule #3. All of the declarations needed to use a module must appear in its header file, and this file is always used to access
the module.

Rule #4. The header file contains only declarations and is included by the .c file for the module.

Rule #5. Set up program-wide global variables with an extern declaration in the header file, and a defining declaration in the .c file. For global variables that will be known throughout the program, place an **extern declaration in the .h file**, as in:

```
extern int g_number_of_entities;
```

The other modules #include only the .h file. The .c file for the module must include this same .h file, and near the beginning of the file, a defining declaration should appear - this declaration both defines and initializes the global variables, as in:

```
int g_number_of_entities = 0;
```

Of course, some other value besides zero could be used as the initial value, and static/global variables are initialized to zero by default; but initializing explicitly to zero is customary because it marks this declaration as to the defining declaration, meaning that this is the unique point of definition. Note that different C compilers and linkers will allow other ways of setting up global variables, but this is the accepted C++ method for defining global variables and it works for C as well to ensure that the global variables obey the One Definition Rule.

Rule #6. Keep a module's internal declarations out of the header file. Sometimes a module uses strictly internal components that are not supposed to be accessed by other modules. If you need structure declarations, global variables, or functions that are used only in the code in the .c file, put their definitions or declarations near the top of the .c file and do not mention them in the .h file.

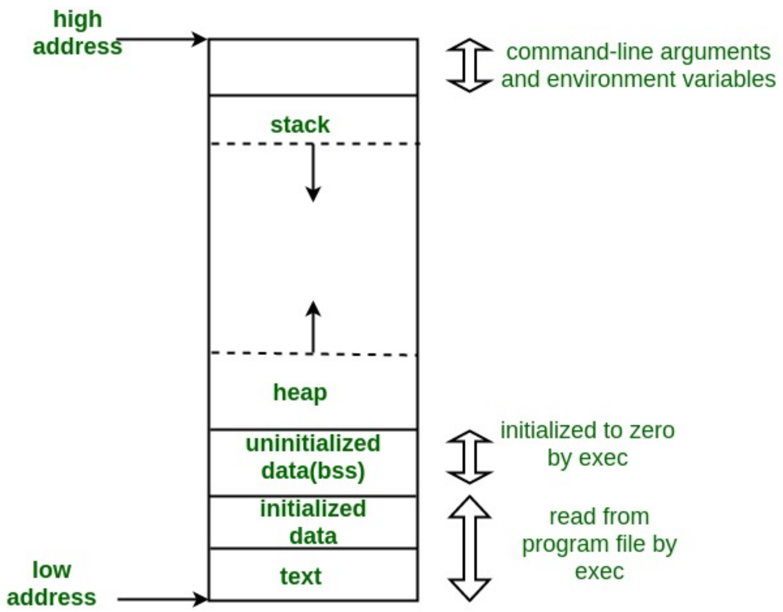
Rule #7. Every header file A.h should #include every other header file that A.h requires to compile correctly.

Rule #8. The content of a header file should compile correctly by itself.

Rule #9. The A.c file should first #include its A.h file, and then any other headers required for its code.

Rule #10. Never #include a.c file for any reason! This happens occasionally and it is always a mess. This causes instant confusion for other programmers and interferes with convenience in using IDEs, because .c files are normally separately compiled, so you have to somehow tell people not to compile this one .c file out of all the others. Furthermore, if they miss this hard-to-document point, they get really confused because compiling this sort of odd file typically produces a million error messages, making people think something mysterious is fundamentally wrong with your code or how they installed it. Conclusion: It can't be treated like a normal header file.

7. Stack, Heap, Data and Code Memory Allocation


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
low address
text

7.1. Text Segment:

A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains **executable instructions.**

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the **text segment is often read-only,** to prevent a program from accidentally modifying its instructions.

This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. It is represented by .text section. This defines an area in memory that stores the instruction codes. This is also a fixed area.

7.2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that the data segment is **not read-only**, since the values of the variables can be **altered at run time.**

This segment can be further classified into the initialized read-only area and the initialized read-write area.

For instance, the global string defined by char s[] = "hello world" in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in the initialized read-write area. And a global C statement like const char* string = "hello world" makes the string literal "hello world" to be stored in the initialized read-only area and the character pointer variable string in the initialized read-write area.

Ex: **static int i = 10** will be stored in the data segment and **global int i = 10** will also be stored in the data segment

7.3. Uninitialized Data Segment:

An uninitialized data segment often called the "**bss**" segment, was named after an ancient assembler operator that stood for "**block started by symbol**." Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are **initialized to zero** or do not have explicit initialization in the source code.

For instance, a variable declared **static int i**; would be contained in the BSS segment.

For instance, a **global** variable declared **int j**; would be contained in the BSS segment.

7.4. Stack:

The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow in opposite directions.)

The stack area contains the program stack, a **LIFO structure**, typically located in the higher parts of memory. On the standard PC x86 computer architecture, it grows toward address zero; on some other architectures, it grows in the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at a minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its **automatic and temporary variables.** This is how **recursive functions** in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

7.5. Heap:

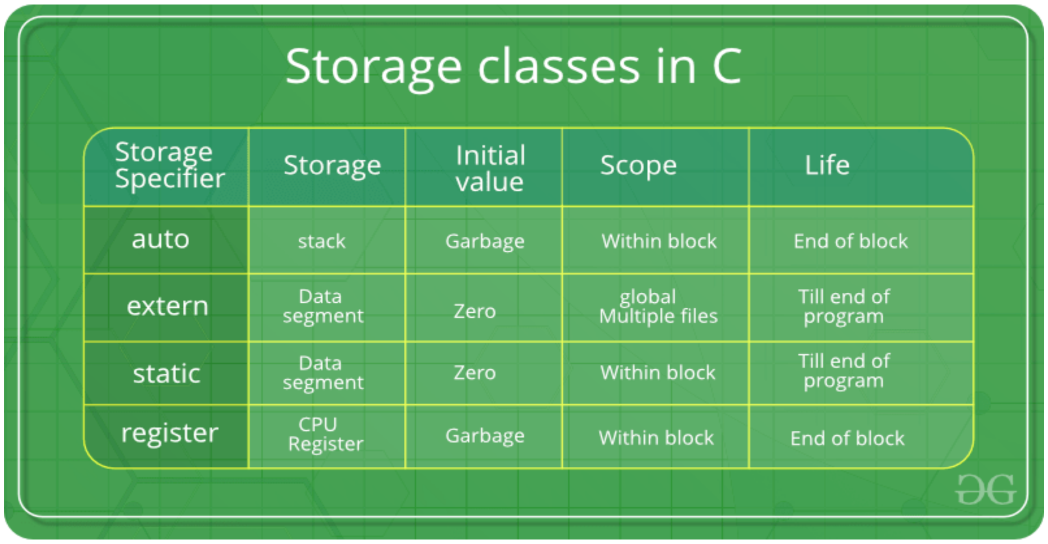
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by **malloc, realloc, and free**, which may use the brk and sbrk system calls to adjust its size. The Heap area is shared by all shared libraries and **dynamically loaded modules** in a process.

7.6. Key difference between STACK and HEAP memory :

- Stack is a linear data structure whereas Heap is a hierarchical data structure.
- Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed.
- Stack accesses local variables only while Heap allows you to access variables globally.
- Stack variables can’t be resized whereas Heap variables can be resized.
- Stack memory is allocated in a contiguous block whereas Heap memory is allocated in any random order.
- Stack doesn’t require to deallocate variables whereas in Heap deallocation is needed.
- Stack allocation and deallocation are done by compiler instructions whereas Heap allocation and deallocation are done by the programmer.

8. Storage Classes:



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

8.1. Automatic Variables

This is the default storage class for all the variables declared **inside a function or a block**. Hence, the keyword auto is rarely used while writing programs in the C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables reside. They are assigned a **garbage value by default** whenever they are declared. The auto variables are always local and are stored **on the stack**.

8.2. Extern

Extern storage class simply tells us that the **variable is defined elsewhere** and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a **global variable initialized with a legal value where it is declared in order to be used elsewhere**. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. The **extern** variables are stored in the data segment. The extern modifier tells the compiler that a different compilation unit is actually declaring the variable, so don't create another instance of it or there will be a name collision at link time.

8.3. Static

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of **preserving their value even after they are out of their scope**! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. **Global static variables can be accessed anywhere in the program.** By default, they are assigned the value 0 by the compiler. The **static** variables are stored in a data segment (again, unless the compiler can optimize them away) and have visibility from the point of declaration to the end of the enclosing scope.

8.4. Register

This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the **register of the microprocessor if a free register is available.** This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers. The **register** modifier tells the compiler to do its best to **keep the variable in a register** if at all possible. Otherwise, it is stored on the stack.

EXAMPLE :

```
// A C program to demonstrate different storage
// classes
#include <stdio.h>

// declaring the variable which is to be made extern an initial value can also be initialized to x
int x;
void autoStorageClass()
{
	printf("\nDemonstrating auto class\n\n");

	// declaring an auto variable (simply writing "int a=32;" works as well)
	auto int a = 32;

	// printing the auto variable 'a'
	printf("Value of the variable 'a'"
		" declared as auto: %d\n", a);

	printf("--------------------------------");
}
void registerStorageClass()
{

	printf("\nDemonstrating register class\n\n");

	// declaring a register variable
	register char b = 'G';

	// printing the register variable 'b'
	printf("Value of the variable 'b'"
		" declared as register: %d\n", b);

	printf("--------------------------------");
}
void externStorageClass()
{

	printf("\nDemonstrating extern class\n\n");

	// telling the compiler that the variable z is an extern variable and has been
	// defined elsewhere (above the main function)
	extern int x;

	// printing the extern variables 'x'
	printf("Value of the variable 'x'"
		" declared as extern: %d\n", x);

	// value of extern variable x modified
	x = 2;

	// printing the modified values of extern variables 'x'
	printf("Modified value of the variable 'x'"
		" declared as extern: %d\n", x);

	printf("--------------------------------");
}
void staticStorageClass()
```

```
{
        int i = 0;

        printf("\nDemonstrating static class\n\n");

        // using a static variable 'y'
        printf("Declaring 'y' as static inside the loop.\n"
                "But this declaration will occur only"
                " once as 'y' is static.\n"
                "If not, then every time the value of 'y' "
                "will be the declared value 5"
                " as in the case of variable 'p'\n");

        printf("\nLoop started:\n");

        for (i = 1; i < 5; i++) {

                // Declaring the static variable 'y'
                static int y = 5;

                // Declare a non-static variable 'p'
                int p = 10;

                // Incrementing the value of y and p by 1
                y++;
                p++;

                // printing value of y at each iteration
                printf("\nThe value of 'y', "
                        "declared as static, in %d "
                        "iteration is %d\n",
                        i, y);

                // printing value of p at each iteration
                printf("The value of non-static variable 'p', "
                        "in %d iteration is %d\n",
                        i, p);
        }
        printf("\nLoop ended:\n");
        printf("--------------------------------");
}
int main()
{
        printf("A program to demonstrate"
                " Storage Classes in C\n\n");

        // To demonstrate auto Storage Class
        autoStorageClass();
        // To demonstrate register Storage Class
        registerStorageClass();

        // To demonstrate extern Storage Class
        externStorageClass();

        // To demonstrate static Storage Class
        staticStorageClass();

        printf("\n\nStorage Classes demonstrated");
        return 0;
    }
```

OUTPUT :

```
        A program to demonstrate Storage Classes in C
        Demonstrating auto class
        Value of the variable ‘a’ declared as auto: 32
        ———————————–
        Demonstrating register class
        Value of the variable ‘b’ declared as register: 71
        ———————————–
        Demonstrating extern class
        Value of the variable ‘x’ declared as extern: 0
        Modified value of the variable ‘x’ declared as extern: 2
        ———————————–
        Demonstrating static class
        Declaring ‘y’ as static inside the loop.
        But this declaration will occur only once as ‘y’ is static.
        If not, then every time the value of ‘y’ will be the declared value 5 as in the case of variable
        ‘p’
        Loop started:
        The value of ‘y’, declared as static, in 1 iteration is 6
        The value of non-static variable ‘p’, in 1 iteration is 11
        The value of ‘y’, declared as static, in 2 iteration is 7
        The value of non-static variable ‘p’, in 2 iteration is 11
        The value of ‘y’, declared as static, in 3 iteration is 8
        The value of non-static variable ‘p’, in 3 iteration is 11
        The value of ‘y’, declared as static, in 4 iteration is 9
        The value of non-static variable ‘p’, in 4 iteration is 11
        Loop ended:
        ———————————–
        Storage Classes demonstrated
```

9. Advance data types

9.1. The #define statement (constants)

In the C Programming Language, the #define directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code.
Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions.

#define CNAME value

#define CNAME (expression)

CNAME : The name of the constant. Most C programmers define their constant names in uppercase, but it is not a requirement of the C Language.

value : The value of the constant.

expression : Expression whose value is assigned to the constant. The *expression* must be enclosed in parentheses if it contains operators.

Note : Do NOT put a semicolon character at the end of #define statements.

```
#define AGE 10
#define AGE_COUNT ( 30 / 2 )
#define NAME "Nidhi Parmar"
```

9.2. Using Typedef

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program.

For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use **a typedef** keyword.

```
#include <stdio.h>
int main()
{
typedef unsigned int unit;
unit i,j;
i=10;
printf("Value of i is :%d",i);
return 0;
}
```

Output:

```
Value of i is :10
```

In the above code, we have declared the **unit** variable of type **unsigned int** by using **a typedef** keyword.

```
typedef struct student
{
char name[20];
int age;
} stud;
stud s1,s2;
```

In the above code, we have declared the variable **stud** of type struct student. Now, we can use the **stud** variable in a program to create the variables of type **struct student**.We conclude that **typedef** keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

typedef vs #define

typedef is limited to giving **symbolic names** to types only whereas **#define** can be used to define **aliases** for values as well, you can define 1 as ONE etc.

typedef interpretation is performed by the **compiler** whereas **#define** statements are processed by the **pre-processor**.

9.3. Variable length arrays

In a **variable length array** in C programming, length or size is evaluated at execution time. We can easily declare one dimensional, two dimensional and multidimensional arrays.

Note: **sizeof** operator when used in variable length array operates at run time instead of at compile time.

```
#include <stdio.h>

void oneDArray( int length, int a[ length ]); //function prototype
void twoDArray( int row, int col, int a[ row ][ col ]); //function prototype

void oneDArray( int length, int a[ length ])
{
  int i;
  for ( i = 0; i < length; ++i)
  printf("a[%d] : %d\n", i, a[ i ]);
 }  //end of oneDArray

void twoDArray( int row, int col, int a[ row ][ col ])
{
  int i, j;
  for ( i = 0; i < row; ++i )
  {
    printf("\n");
    for ( j = 0; j < col; ++j )
      printf("%5d", a[i][j]);
  }
}  //end twoDArray

int main()
{
  int i, j; //counter variable
  int size; //variable to hold size of one dimensional array
  int row1, col1, row2, col2; //number of rows & columns of two D array

  printf("Enter size of one-dimensional array: ");
  scanf("%d", &size);

  printf("Enter number of rows & columns of 2-D array:\n");
  scanf("%d %d", &row1, &col1);

  printf("Enter number of rows & columns of multi-D array:\n");
  scanf("%d %d", &row2, &col2);

  //declaring arrays
  int arr[ size ];
  int arr2D[ row1 ][ col1 ]; //2-D array
  int arrMD[ row2 ][ col2 ]; //multi-dimensional array

  //one dimensional array
  for ( i = 0; i < size; ++i)
```

```
    {
      arr[ i ] = 2 * i;
    }

    //two dimensional array
    for ( i = 0; i < row1; ++i)
    {
      for ( j = 0; j < col1; ++j)
      {
      arr2D[ i ][ j ] = i + j;
      }
    }

    //multi-dimensional array
    for ( i = 0; i < row2; ++i)
    {
      for ( j = 0; j < col2; ++j)
      {
        arrMD[ i ][ j ] = i + j;
      }
    }

    //printing arrays
    printf("One-dimensional array:\n");
    oneDArray( size, arr );

    printf("Two-dimensional array:\n");
    twoDArray( row1, col1, arr2D );

    printf("\nMulti-dimensional array:\n");
    twoDArray( row2, col2, arrMD );

    return 0;
  }  //end main
```

Output :

```
Enter size of one-dimensional array: 5
Enter number of rows & columns of 2-D array:
2
4
Enter number of rows & columns of multi-D array:
4
3
One-dimensional array:
a[0] : 0
a[1] : 2
a[2] : 4
```

```
a[3] : 6
a[4] : 8
Two-dimensional array:

 0  1  2  3
 1  2  3  4
Multi-dimensional array:

 0  1  2
 1  2  3
 2  3  4
 3  4  5
```

In the above example, we see that function parameters of oneDArray and twoDArray are declared with variable length array type.

The size of a variable length array in c programming must be of integer type and it cannot have an initializer.

9.4. Flexible array members

A structure *with at least one member* may additionally contain a single array member of unspecified length at the end of the structure. This is called a flexible array member:

A flexible array member is the official C99 name for what used to (usually) be called the "struct hack". The basic idea is that you define a struct something like this:

```
struct x {
int a; // whatever members you want here.
size_t size;
int x[ ]; // no size, last member only
};
```

This is used primarily (or exclusively) with dynamic allocation. When you want to allocate an object of this type, you allocate enough extra space for whatever size of array you need:

```
struct x *a = malloc(sizeof(struct x) + 20 * sizeof(int));
a->size = 20;
```

The **size** member isn't strictly necessary, but often handy to keep track of the size allocated for an item. The one above has space for 20 int's, but the main point of this is that you might have several mallocs around, each with its own size.

9.5. Designated initializers

In C90 standard we have to initialize the arrays in the fixed order, like initializing indexes at positions 0, 1, 2 and so on. From the C99 standard, they have introduced designated initializing features in C. Here we can initialize elements in random order. Initialization can be done using the array index or structure members. This extension is not implemented in GNU C++.

To specify an **array index**, write **'[index] =' or '[index]'** before the element value. For example,

int a[6] = {[4] = 29, [2] = 15 }; or

int a[6] = {[4]29 , [2]15 };

is equivalent to

int a[6] = { 0, 0, 15, 0, 29, 0 };

Note:- *The index values must be constant expressions.*
To initialize a range of elements to the same value, write **'[first ... last] = value'**.The triple dots need to be separated from the both numbers. For example,

```
// C program to demonstrate designated initializers
// with arrays.
#include <stdio.h>
void main(void)
{
 int numbers[100] = {1, 2, 3, [3 ... 9] = 10,
          [10] = 80, 15, [70] = 50, [42] = 400 };

 int i;
 for (i = 0; i < 20; i++)
          printf("%d ", numbers[i]);

 printf("\n%d ", numbers[70]);
 printf("%d", numbers[42]);
}
```

Output :

```
1 2 3 10 10 10 10 10 10 10 80 15 0 0 0 0 0 0 0 0
50 400
```

Explanation:

- In this example, the first three elements are initialized to 1, 2, and 3 respectively.
- 4th to 10th elements have value 10.
- Then element 80 (the 11th element) is initialized to 10
- The next element (12th) is initialized to 15.
- Element number 70 (the 71st) is initialized to 50, and number 42 (the 43rd) to 400.
- As with Normal initialization, all uninitialized values are set to zero.

Note:-

- These initializers do not need to appear in order.
- The length of the array is the highest value specified plus one.

In structure or union: In a structure initializer, specify the name of a field to initialize with **'.fieldname =' or 'fieldname:'** before the element value. For example, given the following structure,

```
// C program to demonstrate designated
// initializers with structures
#include <stdio.h>
struct Point
{
        int x, y, z;
};
int main()
{
        // Examples of initialization using
        // designated initialization
        struct Point p1 = {.y = 0, .z = 1, .x = 2};   // equivalent to struct Point p = { 2, 0, 1 };
        struct Point p2 = {.x = 20};                  // equivalent to struct Point p = { 20 };
        printf("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
        printf("x = %d", p2.x);
        return 0;
}
```

Output :

```
x = 2, y = 0, z = 1
x = 20
```

10. Type qualifiers

10.1. 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 const variable by using pointer). The result is implementation-defined if an attempt is made to change a const.

10.1.1. Pointer to variable.

int *ptr;

We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in the read-write area. See the following code fragment.

```
#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;
}
```

Output:

```
*ptr: 10
*ptr: 20
*ptr: 100
```

10.1.2. Pointer to constant.

Pointer to constant can be declared in the following two ways.

const int *ptr;   OR   **int const *ptr;**

We can change the pointer to point to any other integer variable, but cannot change the value of the object (entity) pointed using pointer ptr. The pointer is stored in the read-write area (stack in the present case). The object pointed may be in the read-only or read-write area. See the following code fragment.

```
#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;
}
```

Output :

```
main.c:11:7: error: assignment of read-only location '*ptr'
*ptr = 100;
     ^
```

10.1.3. Constant Pointer to variable.

int *const ptr;

Above declaration is a constant pointer to an integer variable, which means we can change the value of an object pointed by a pointer, but cannot change the pointer to point to another variable.

```
#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;
}
```

Output :

```
main.c:14:5: error: assignment of read-only variable ‘ptr’
ptr = &j;  /* error */
    ^
```

10.1.4. Constant Pointer to constant.

const int *const ptr;

Above declaration is a constant pointer to a constant variable which means we cannot change the value pointed by the pointer as well as we cannot point the pointer to other variables. Let us see with an example.

```
#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;
}
```

Output:

```
main.c:12:6: error: assignment of read-only variable ‘ptr’
  ptr = &j;  /* error */
      ^
main.c:13:7: error: assignment of read-only location ‘*ptr’
  *ptr = 100; /* error */
       ^
```

10.2. Volatile

volatile tells the compiler not to optimize anything that has to do with the volatile variable.

There are at least three common reasons to use it, all involving situations where the value of the variable can change without action from the visible code:

1. When you interface with hardware that changes the value itself.
2. When there's another thread running that also uses the variable.
3. When there's a signal handler that might change the value of the variable.

For example, a little piece of hardware that is mapped into RAM somewhere and that has two addresses: a command port and a data port:

```
typedef struct {
        int command;
        int data;
        int isBusy;
} MyHardwareGadget;
```

Now you want to send some command:

```
void SendCommand (MyHardwareGadget * gadget, int command, int data)
{
        // wait while the gadget is busy:
        while (gadget->isbusy)
        {
                // do nothing here.
        }
        // set data first:
        gadget->data = data;
        // writing the command starts the action:
        gadget->command = command;
}
```

Looks easy, but it can fail because the compiler is free to change the order in which data and commands are written. This would cause our little gadget to issue commands with the previous data-value. Also take a look at the wait while busy loop. That one will be optimized out. The compiler will try to be clever, read the value of isBusy just once and then go into an infinite loop. That's not what you want.

The way to get around this is to declare the pointer gadget as **volatile**. This way the compiler is forced to do what you wrote. It can't remove the memory assignments, it can't cache variables in registers and it can't change the order of assignments either
This is the correct version:

```
void SendCommand (volatile MyHardwareGadget * gadget, int command, int data)
{
        // wait while the gadget is busy:
        while (gadget->isBusy)
        {
                // do nothing here.
        }
        // set data first:
        gadget->data = data;
        // writing the command starts the action:
        gadget->command = command;
}
```

Use of “volatile”

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables within a multi-threaded application

Basically, C standard says that “**volatile**” variables can change from outside the program and that’s why **compilers aren’t supposed to optimize their access**. Now, you can guess that having too many ‘**volatile**‘ variables in your program would also result in lesser & lesser compiler optimization.

10.3. Restrict

- This is used only with pointers. It **indicates that the pointer is only an initial way to access the dereference data**. It provides more help to the compiler for optimization.

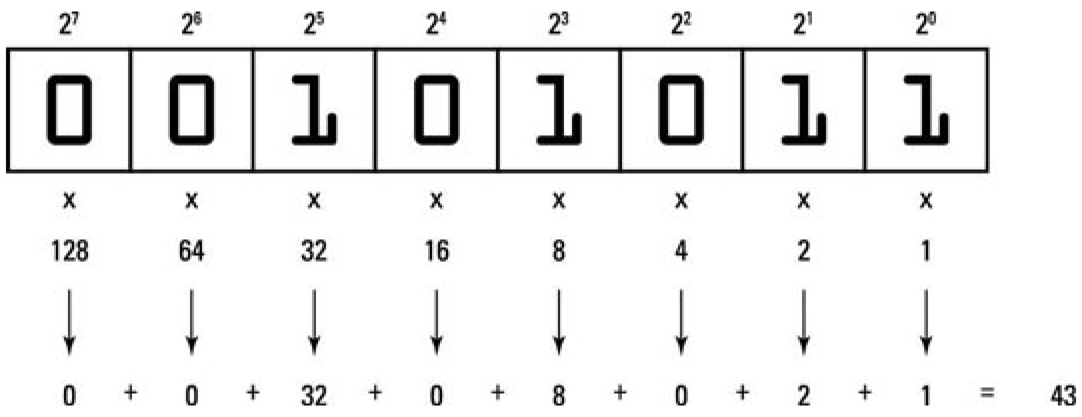
- **restrict** keyword is mainly used in pointer declarations as a type qualifier for pointers.
- It doesn't add any new functionality. It is only a way for programmers to inform about optimizations that compilers can make.
- When we use restrict with a pointer ptr, it tells the compiler that ptr is the only way to access the object pointed by it and the compiler doesn't need to add any additional checks.
- restrict is not supported by C++. It is a C only keyword.

11. Bit manipulation

11.1. Binary numbers and bits

Decimal to binary in C: We can convert any decimal number (base-10 (0 to 9)) into binary number(base-2 (0 or 1)) by c program.

Expression	Decimal Value	Binary Value
2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	10000
2^5	32	100000
2^6	64	1000000
2^7	128	10000000



Each bit that's set, or has the value 1, represents a power of two: 2^5, 2^3, 2^1, and 2^0. When you multiply these values by their decimal counterparts and then total them up, you get the decimal representation of binary 00101011, which is 43.

Example :

- Step 1: Divide the number by 2 through % (modulus operator) and store the remainder in array
- Step 2: Divide the number by 2 through / (division operator)
- Step 3: Repeat the step 2 until number is greater than 0

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

int main()
{
        int a[10], n, i;
        printf("Enter the number to convert: ");
        scanf("%d",&n);

        for(i=0; n>0; i++)
        {
                a[i]=n%2;
                n=n/2;
        }

        printf("\nBinary of Given Number is=");
        for(i=i-1;i>=0;i--)
        {
                printf("%d",a[i]);
        }
        return 0;
    }
```

Output:

```
    Enter the number to convert: 5
    Binary of Given Number is=101
```

11.2. Bitwise Operators (Logical)

BITWISE OPERATORS are used for manipulating data at the bit level, also called bit level programming. Bitwise operates on one or more bit patterns or binary numerals at the level of their individual bits. They are used in numerical computations to make the calculation process faster.
Bitwise operators cannot be directly applied to primitive data types such as float, double, etc. The bitwise operators are mostly used with the integer data type because of its compatibility.

The bitwise logical operators work on the data bit by bit, starting from the least significant bit, i.e. LSB bit which is the rightmost bit, working towards the MSB (Most Significant Bit) which is the leftmost bit.

The result of the computation of bitwise logical operators is shown in the table given below.

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise AND

This is one of the most commonly used logical bitwise operators. It is represented by a single ampersand sign (&). Two integer expressions are written on each side of the (&) operator.

The result of the bitwise AND operation is 1 if both the bits have the value as 1; otherwise, the result is always 0.

Bitwise OR

It is represented by a single vertical bar sign (|). Two integer expressions are written on each side of the (|) operator.

The result of the bitwise OR operation is 1 if at least one of the expressions has the value as 1; otherwise, the result is always 0.

Bitwise EXCLUSIVE OR

It is represented by a symbol (^). Two integer expressions are written on each side of the (^) operator.

The result of the bitwise Exclusive-OR operation is 1 if only one of the expressions has the value as 1; otherwise, the result is always 0.

A simple program that demonstrates bitwise logical operators :

```
#include <stdio.h>
int main()
{
        int a = 20;      /* 20 = 010100 */
        int b = 21;      /* 21 = 010101 */
        int c = 0;

        c = a & b;     /* 20 = 010100 */
        printf("AND - Value of c is %d\n", c );

        c = a | b;     /* 21 = 010101 */
        printf("OR - Value of c is %d\n", c );

        c = a ^ b;     /* 1 = 0001 */
        printf("Exclusive-OR - Value of c is %d\n", c );

        getch();
}
```

Output:

```
AND - Value of c is 20
OR - Value of c is 21
Exclusive-OR - Value of c is 1
```

11.3. Bitwise Operators (Shifting)

The bitwise shift operators are used to move/shift the bit patterns either to the left or right side. Left and right are two shift operators provided by 'C' which are represented as follows:

Operand << n (Left Shift)
Operand >> n (Right Shift)

Here,

- an operand is an integer expression on which we have to perform the shift operation.
- 'n' is the total number of bit positions that we have to shift in the integer expression.

The left shift operation will shift the 'n' number of bits to the left side. The leftmost bits in the expression will be popped out, and n bits with the value 0 will be filled on the right side.

The right shift operation will shift the 'n' number of bits to the right side. The rightmost 'n' bits in the expression will be popped out, and the value 0 will be filled on the left side.

Example: x is an integer expression with data 1111. After performing shift operation the result will be:

x << 2 (left shift) = 1111<<2 = 1100
x>>2 (right shift) = 1111>>2 = 0011

Shift operators can be combined then it can be used to extract the data from the integer expression. A program to demonstrate the use of bitwise shift operators.

EXAMPLE :

```
#include <stdio.h>
int main() {
  int a = 20;      /* 20 = 010100 */
  int c = 0;

  c = a << 2;      /* 80 = 101000 */
  printf("Left shift - Value of c is %d\n", c );

  c = a >> 2;      /*05 = 000101 */
  printf("Right shift - Value of c is %d\n", c );
  return 0;
}
```

Output:

```
Left shift - Value of c is 80
Right shift - Value of c is 5
```

After performing the left shift operation the value will become 80 whose binary equivalent is 101000.

After performing the right shift operation, the value will become 5 whose binary equivalent is 000101.

11.4. Bitmasks

A mask defines which bits you want to keep, and which bits you want to clear.
Masking is the act of applying a mask to a value. This is accomplished by doing:

- Bitwise ANDing in order to extract a subset of the bits in the value
- Bitwise ORing in order to set a subset of the bits in the value
- Bitwise XORing in order to toggle a subset of the bits in the value

Below is an example of extracting a subset of the bits in the value:

Mask: 00001111b

Value: 01010101b

Applying the mask to the value means that we want to clear the first (higher) 4 bits, and keep the last (lower) 4 bits. Thus we have extracted the lower 4 bits. The result is:

Mask: 00001111b

Value: 01010101b

```
Result: 00000101b
```

Masking is implemented using AND, so in C we get:

```
uint8_t stuff(...)
{
	uint8_t mask = 0x0f; // 00001111b
	uint8_t value = 0x55; // 01010101b
	return mask & value;
}
```

11.5. Using Bit Operators to pack data

```
int age, gender, height, packed_info;
// Assign values
// Pack as AAAAAAA G HHHHHHH
using shifts and "or"
packed_info = (age << 8) | (gender << 7) | height;
```

The left shift operator means "multiply by two, this many times". In binary, multiplying a number by two is the same as adding a zero to the right side.

The right shift operator is the reverse of the left shift operator.

The pipe operator is "or", meaning overlay two binary numbers on top of each other, and where there is a 1 in either number the result in that column is a 1.

So, let's extract the operation for packed_info:

```
// Create age, shifted left 8 times:
// AAAAAAA00000000
age_shifted = age << 8;

// Create gender, shifted left 7 times:
// 0000000G0000000
gender_shifted = gender << 7;

// "Or" them all together:
// AAAAAAA00000000
// 0000000G0000000
// 00000000HHHHHHH
// ---------------
// AAAAAAAGHHHHHHH
packed_info = age_shifted | gender_shifted | height;
```

11.6. Setting and Reading Bits

11.6.1. Setting Bits

We use bitwise OR "|" operator to set any bit of a number. Bitwise OR operator evaluates each bit of the resultant value to 1, if any of the operand corresponding bit value is 1.

Step by step descriptive logic to set the nth bit of a number.

- Input number from user. Store it in some variable say **num**.
- Input bit position you want to set. Store it in some variable say **n**.
- To set a particular bit of number. Left shift **1 n** times and perform bitwise OR operation with num. Which is **newNum = (1 << n) | num;.**

Example :

```
// C program to set the nth bit of a number

#include <stdio.h>
int main()
{
   int num, n, newNum;

   /* Input number from user */
   printf("Enter any number: ");
   scanf("%d", &num);

   /* Input bit position you want to set */
   printf("Enter nth bit to set (0-31): ");
   scanf("%d", &n);

   /* Left shift 1, n times and perform bitwise OR with num */
   newNum = (1 << n) | num;

   /* Clearing Bits :  Left shifts 1 to n times, Perform complement of above, finally
   perform  bitwise AND with num and result of above */
   // newNum = num & (~(1 << n)); // clearing bit

   printf("Bit set successfully.\n\n");
   printf("Number before setting %d bit: %d (in decimal)\n", n, num);
   printf("Number after setting %d bit: %d (in decimal)\n", n, newNum);

   return 0;
}
```

Output:

```
Enter any number: 12
Enter nth bit to set (0-31): 0
Bit set successfully.

Number before setting 0 bit: 12 (in decimal)
Number after setting 0 bit: 13 (in decimal)
-----------------------------------------------------------------------
Enter any number: 4
Enter nth bit to set (0-31): 1
Bit set successfully.

Number before setting 1 bit: 4 (in decimal)
Number after setting 1 bit: 6 (in decimal)
```

11.6.2. Reading Bits

To read a specific bit from a byte. The value that we want to test is 0 or 1.To read n bit from the buffer, If it's 0 or 1. Example if 7th bit from byte is 0 or 1.

EXAMPLE:

```
#include <stdio.h>

int main()
{
   int number, bit_position_to_read;

   printf("Enter number for reading bit value : ");
   scanf("%d", &number);

   printf("Enter position for reading bit value : ");
   scanf("%d", &bit_position_to_read);

   if (((number >> bit_position_to_read) & 0x01) == 1)
   {
      printf("Bit at position %d in number %d is : 1 ", bit_position_to_read, number);
   }
   else
   {
      printf("Bit at position %d in number %d is : 0 ", bit_position_to_read, number);
   }
   return 0;
}
```

OUTPUT:

```
Enter number for reading bit value : 1
Enter position for reading bit value : 0
Bit at position 0 in number 1 is : 1
--------------------------------------------------------
Enter number for reading bit value : 4
Enter position for reading bit value : 2
Bit at position 2 in number 4 is : 1
--------------------------------------------------------
Enter number for reading bit value : 4
Enter position for reading bit value : 1
Bit at position 1 in number 4 is : 0
```

11.7. Using Bit Fields to pack data

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows –

```
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;
```

This structure status1 requires 8 bytes of memory space but in actuality, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only that number of bytes. For example, the above structure status1 can be rewritten as follows –

```
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status2;
```

The above structure requires 4 bytes of memory space for the status2 variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status2 structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. The following example to understand the concept –

EXAMPLE :

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

/* define simple structure */
struct {
  unsigned int widthValidated;
```

```
    unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;

int main( ) {
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
    return 0;
}
```

OUTPUT :

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

The declaration of a bit-field has the following form inside a structure –

```
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.

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, 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.

In the below example, structure Age 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. When the below code is compiled, it will compile with a warning and when executed, it produces the following output as mentioned.

EXAMPLE :

```
#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; // Shows warning as “width” of int age is 3
          printf( "Age.age : %d\n", Age.age );

          return 0;
}
```

OUTPUT :

```
main.c:11:30: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type
‘long unsigned int’ [-Wformat=]
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
                            ^
main.c:17:14: warning: large integer implicitly truncated to unsigned type [-Woverflow]
   Age.age = 8;
            ^
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Use of bit field in embedded C

Suppose a microcontroller GPIO Port has 8 pins and each pin is connected to the led. In that scenario using the bitfield, we can easily change the status of the led. Let’s see a small example where I am trying to explain how to access GPIO Pin using the bit-field.

So first we need to create a bit-field structure for mapping with the GPIO Port of a given micro-controller.

```
typedef union
{
          struct
          {
                    uint8_t LED1 : 1;
                    uint8_t LED2 : 1;
```

```
                uint8_t LED3 : 1;
                uint8_t LED4 : 1;
                uint8_t LED5 : 1;
                uint8_t LED6 : 1;
                uint8_t LED7 : 1;
                uint8_t LED8 : 1;
        };

        uint8_t AllLedState;
} LED_BAR_STATE;

/* Create a pointer to the above-created bit-field 'LED_BAR_STATE' and
assign the address of the GPIO Port. */

volatile LED_BAR_STATE *pLedState = (volatile LED_BAR_STATE *)0xE002C000;

// Access to the individual led using the pointer.

pLedState->LED1 = 1;
pLedState->LED2 = 0;
```

Note: It is not a Good suggestion to use bit-field in the mapping of a hardware register because the allocation of bit-field depends upon the compiler. Might be the result of one compiler can be different from another compiler. So we should avoid the compiler-dependent code. In simple words, avoid using bit fields for the mapping of the hardware register.

We can not create a **pointer** to the bit-field and also not use the address-of operator (&) to the bit-field member.

EXAMPLE :

```
#include <stdio.h>
struct sData
{
  unsigned int a: 2;
  unsigned int b: 2;
  unsigned int c: 2;
};
int main()
{
  struct sData data;
  data.a = 2;
  printf("Address of data.a =  %p", &data.a );
 return 0;
}
```

OUTPUT :

```
main.c:12:39: error: cannot take address of bit-field ‘a’
   printf("Address of data.a =  %p", &data.a );
                                     ^
```

We can not create an **array** of a bit field in c.

EXAMPLE :

```c
#include <stdio.h>

struct sData
{
  unsigned int a: 2;
  unsigned int b[5]: 2;
};
int main()
{
  struct sData data;
  data.a = 2;
  return 0;
}
```

OUTPUT :

```
Compilation failed due to following error(s).
  unsigned int b[5]: 2;
              ^
```

We can not calculate the size of the bit field in c using the **sizeof operator**.

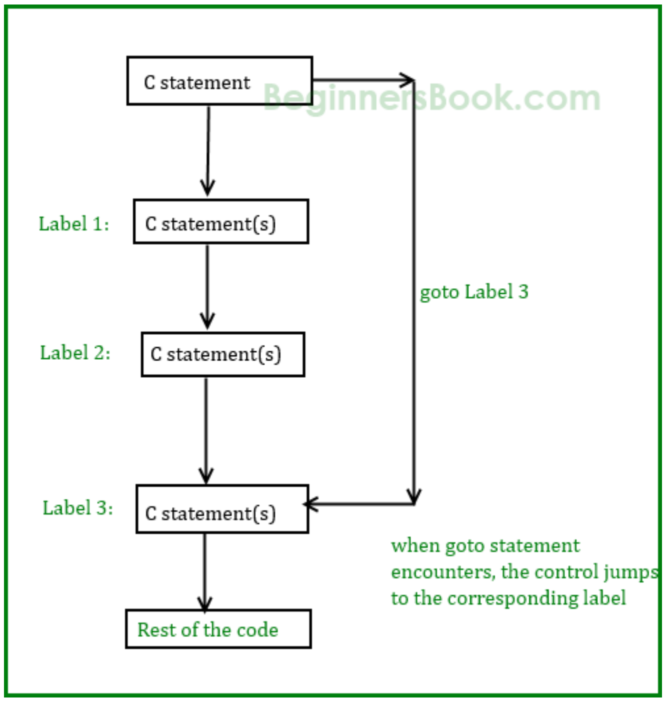
```c
#include <stdio.h>
struct sData
{
  unsigned int a: 2;
  unsigned int b: 2;
  unsigned int c: 2;
};
int main()
{
  struct sData data;
  printf("Sizeof of data.a =  %d", sizeof(data.a));
  return 0;
}
```

OUTPUT :

```
main.c:11:44: error: 'sizeof' applied to a bit-field
    printf("Sizeof of data.a =  %d", sizeof(data.a));
                                     ^
```

12. Advanced control flow

12.1. The goto statement (Why it is not preferable in embedded systems)



The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated.

When should we use goto?

1) The goto statement allows us to transfer control of the program to the specified label.

EXAMPLE :

```
// Program to calculate the sum and average of positive numbers
// If the user enters a negative number, the sum and average are displayed.
```

```
#include <stdio.h>

int main() {

  const int maxInput = 100;
  int i;
  double number, average, sum = 0.0;

  for (i = 1; i <= maxInput; ++i) {
    printf("%d. Enter a number: ", i);
    scanf("%lf", &number);

    // go to jump if the user enters a negative number
    if (number < 0.0) {
      goto jump;
    }
    sum += number;
  }

jump:
  average = sum / (i - 1);
  printf("Sum = %.2f\n", sum);
  printf("Average = %.2f", average);

  return 0;
}
```

OUTPUT :

```
1. Enter a number: 1
2. Enter a number: 2
3. Enter a number: 5
4. Enter a number: 8
5. Enter a number: -2
Sum = 16.00
Average = 4.00
```

2) The condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. Consider the following example.

EXAMPLE :

```
#include <stdio.h>
int main()
{
        int i, j, k;
        for(i=0;i<10;i++)
         {
```

```
                    for(j=0;j<5;j++)
                    {
                            for(k=0;k<3;k++)
                            {
                                    printf("%d %d %d\n",i,j,k);
                                    if(j == 3)
                                    {
                                            goto out;
                                    }
                            }
                    }
            }
     out:
            printf("came out of the loop");
        }
```

OUTPUT :

```
0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0
came out of the loop
```

Why is goto not preferable in embedded systems?

Confusing Code : Code can get confusing very quickly when it takes arbitrary paths and jumps from place to place. The more GOTOs in the same routine, the worse it gets. Debugging is enough of a challenge when code runs sequentially. When it jumps around as you are trying to step through it, debugging can become a nightmare.

Infinite Loop : It is easy to get caught in an infinite loop if the goto point is above the goto call. This is almost guaranteed if there is no reliable escape from the code, such as a RETURN statement within a conditional statement.

12.2. The null statement

If a statement has only a semicolon, then it is called a null statement.

Syntax:

;

What is the purpose of the null statement?
do nothing

Example 1:

```
for (i = 0; i < 5; i++)
;   // null statement
```

Here, the body of the for loop is a null statement.

Example 2:

```
goto label;
........
label:
;
```

12.3. The comma operator

In the code below, the value of **b** will be 30, **because 10, 20, 30 are enclosed in braces, and braces have more priority than assignment (=) operator**. When multiple values are given with comma operator within the braces, then the right most value is considered as the result of the expression. Thus, 30 will be assigned to the variable **b.**

EXAMPLE :

```
#include <stdio.h>
int main()
{
        int a,b;
        // a = 10,20,30; // error, It is a declaration with an initialization statement and here
        comma is a separator and we cannot use values like this.

        b = (10,20,30);

        //printing the values
        printf("b = %d\n",b);
        return 0;
}
```

OUTPUT :

```
b = 30
```

12.4. Setjmp & longjmp functions

“Setjmp” and “Longjmp” are defined in **setjmp.h**, a header file in the C standard library.

- **setjmp(jmp_buf buf)** : uses buf to remember current position and returns 0.
- **long jump(jmp_buf buf, i)** : Go back to place buf is pointing to and return i .

EXAMPLE :

```
// A simple C program to demonstrate working of setjmp() and longjmp()
#include<stdio.h>
#include<setjmp.h>
jmp_buf buf;
void func()
{
	printf("Welcome to func()\n");

	// Jump to the point setup by setjmp
	longjmp(buf, 1);

	printf("Sample2\n");
}

int main()
{
	// Setup jump position using buf and return 0
	if (setjmp(buf))
		printf("Sample3\n");
	else
	{
		printf("Sample4\n");
		func();
	}
	return 0;
}
```

OUTPUT :

```
Sample4
Welcome to func()
Sample3
```

The main feature of these functions is to provide a way that deviates from standard call and return sequence. This is mainly used to implement exception handling in C. **setjmp** can be used like **try** (in languages like C++, Java and Python). The call to **longjmp** can be used like **throw** (Note that longjmp() transfers control to the point set by setjmp()).

13. Input and output

13.1 char functions (input / output)

getchar() : It is used to get or read the input (i.e a single character) at runtime.

EXAMPLE :

```
#include <stdio.h>

void main()

{

   printf("Enter Char : ");

   char ch = getchar();

   printf("Input Char is : %c",ch);

}
```

OUTPUT :

```
Enter Char : nidhi

Input Char is : n
```

getche() : It is used to get a character from the console and echoes to the screen.

This is a simple Hello World! C program. After displaying Hello World! In the output screen, this program waits for any character input from the keyboard. After any single character is typed/pressed, this program returns 0. But, please note that, the getche() function will wait for any keyboard input (and press ENTER) and it will display the given input character on the output screen immediately after keyboard input is entered.To use getch(), getche() functions, you need to include #include <conio.h> header file which is a non-standard header file.

EXAMPLE :

```
#include <stdio.h>

 #include <conio.h>
```

```
int main()
{
   char flag;
   /* Our first simple C basic program */
   printf("Do you want to continue Y or N");

   printf("Hello World! ");    flag = getche(); // It waits for keyboard input.
   if (flag == 'Y')
   {
          printf("You have entered Yes");
   }
   else
   {
          printf("You have entered No");
   }
   return 0;
}
```

OUTPUT :

```
Hello World!
Do you want to continue Y or N
Y
You have entered Yes
```

getch() is used to get a character from the console but does not echo to the screen.

EXAMPLE

```
    #include <stdio.h>

    #include <conio.h>

    void main()

    {

        char ch;

        ch = getch();

        printf("Input Char Is :%c",ch);

    }
```

OUTPUT :

```
    Input Char Id: n
```

During the program execution, a single character gets or reads through the getch(). The given value is not displayed on the screen and the compiler does not wait for another character to be typed.And then, the given character is printed through the printf function.

putch() : It displays any alphanumeric characters to the standard output device. It displays only one character at a time.

EXAMPLE :

```
    #include<stdio.h>

    #include<conio.h>

    void main()
```

```
{
        char ch;
        clrscr(); /* Clear the screen */
        printf("Press any character: ");
        ch = getch();
        printf("\nPressed character is:");
        putch(ch);
        getch(); /* Holding output */
}
```

OUTPUT :

```
Press any character:
Pressed character is: e
```

Note: while using getch() pressed character is not echoed. Here the pressed character is e and it is displayed by putch().

putchar() : This method in C is used to write a character, of unsigned char type, to stdout. This character is passed as the parameter to this method.

EXAMPLE :

```
// C program to demonstrate putchar() method
#include <stdio.h>
int main()
{
        // Get the character to be written
```

```
                char ch = '1';

                // Write the Character to stdout

                for (ch = '1'; ch <= '9'; ch++)

                        putchar(ch);

                return (0);

        }
```

OUTPUT :

```
        123456789
```

13.2 string functions

gets() : It accepts single or multiple characters of string including spaces from the standard Input device.
puts() : It displays single or multiple characters of string including spaces to the standard Input device.

EXAMPLE :

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

void main()
{
  char a[20];
  gets(a);
  puts(a);
}
```

OUTPUT :

```
nidhi parmar

nidhi parmar
```

13.3 Formatting functions

Formatted console input/output functions are used to take one or more inputs from the user at the console and it also allows us to display one or multiple values in the output to the user at the console.

Some of the most important formatted console input/output functions are -

Functions	Description
scanf()	This function is used to read *one or multiple* inputs from the user at the console.
printf()	This function is used to display *one or multiple* values in the output to the user at the console.
sscanf()	This function is used to read the characters from a string and store them in variables.
sprintf()	This function is used to read the values stored in different variables and store these values in a character array.

EXAMPLE :

```
#include<stdio.h>

int main()

{

        char ar[20] = "Nidhi M 23 1.75";

        char str[10];

        char ch;

        int i;
        float f;

        /* Calling sscanf() to read multiple values from a char[] array and store each
        value in matching variable */

        sscanf(ar, "%s %c %d %f", str, &ch, &i, &f);

        printf("The value in string is : %s \n", str);

        printf("The value in char is : %c \n", ch);

        printf("The value in int is : %d \n", i);

        printf("The value in float is : %f ", f);

        return 0;

    }
```

OUTPUT :

```
    The value in string is : Nidhi

    The value in char is : F

    The value in int is : 23

    The value in float is : 1.750000
```

EXAMPLE :

```
    #include<stdio.h>

    int main()

    {

        char target[20];

        char name[10] = "Nidhi";

        char gender  = 'F';

        int age = 23;
        float height = 1.70;

        printf("The name is : %s \n", name);

        printf("The gender is : %c \n", gender);

        printf("The age is : %d \n", age);

        printf("The height is : %f \n", height);

        /* Calling sprintf() function to read multiple variables and store their values in a
        char[] array i.e. string.*/

        sprintf(target, "%s %c %d %f", name, gender, age, height);

        printf("The value in the target string is : %s ", target);

        return 0;

    }
```

OUTPUT :

```
    The name is : Nidhi

    The gender is : F

    The age is : 23

    The height is : 1.700000

    The value in the target string is : Nidhi F 23 1.700000
```

14. Advance function concept

14.1. Variadic Functions

Variadic functions are functions that can take a variable number of arguments. In C programming, a variadic function adds flexibility to the program. It takes one fixed argument and then any number of arguments can be passed. The variadic function consists of at least one fixed variable and then an ellipsis(…) as the last parameter.

Syntax:

int function_name(data_type variable_name, ...);

Values of the passed arguments can be accessed through the header file named as:

#include <stdarg.h>

<stdarg.h> includes the following methods:

- **va_start(va_list ap, argN)** : This enables access to variadic function arguments.
- **va_arg(va_list ap, type)** : This one accesses the next variadic function argument.
- **va_copy(va_list dest, va_list src)** : This makes a copy of the variadic function arguments.
- **va_end(va_list ap)** : This ends the traversal of the variadic function arguments.
- **va_list** holds the information needed by **va_start**, **va_arg**, **va_end**, and **va_copy**.

The following simple C program will demonstrate the working of the variadic function **AddNumbers()**:

EXAMPLE :

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

// Variadic function to add numbers
int AddNumbers(int n, ...)
{
        int Sum = 0;

        // Declaring pointer to the argument list
        va_list ptr;

        // Initializing argument to the list pointer
        va_start(ptr, n);

        for (int i = 0; i < n; i++)

                // Accessing current variable and pointing to next one
                Sum += va_arg(ptr, int);

        // Ending argument list traversal
        va_end(ptr);

        return Sum;
}

// Driver Code
int main()
{
        printf("\n\n Variadic functions: \n");
        // Variable number of arguments
        printf("\n 1 + 2 = %d ",
                AddNumbers(2, 1, 2));

        printf("\n 3 + 4 + 5 = %d ",
                AddNumbers(3, 3, 4, 5));

        printf("\n 6 + 7 + 8 + 9 = %d ",
                AddNumbers(4, 6, 7, 8, 9));

        printf("\n");

        return 0;
    }
```

OUTPUT:

```
    Variadic functions:

     1 + 2 = 3
     3 + 4 + 5 = 12
     6 + 7 + 8 + 9 = 30
```

14.2. va_copy

The va_copy macro copies src to dest.

va_end should be called on dest before the function returns or any subsequent re-initialization of dest (via calls to va_start or va_copy).

Parameters

dest - an instance of the va_list type to initialize
src - the source va_list that will be used to initialize dest

EXAMPLE :

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

    double sample_stddev(int count, ...)
    {
       /* Compute the mean with args1. */
       double sum = 0;
       va_list args1;
       va_start(args1, count);
       va_list args2;
       va_copy(args2, args1);            /* copy va_list object */
       for (int i = 0; i < count; ++i) {
           double num = va_arg(args1, double);
           sum += num;
       }
       va_end(args1);
       double mean = sum / count;

       /* Compute standard deviation with args2 and mean. */
       double sum_sq_diff = 0;
       for (int i = 0; i < count; ++i) {
           double num = va_arg(args2, double);
           sum_sq_diff += (num-mean) * (num-mean);
       }
       va_end(args2);
       return sqrt(sum_sq_diff / count);
    }

    int main(void)
    {
       printf("%f\n", sample_stddev(4, 25.0, 27.3, 26.9, 25.7));
    }
```

OUTPUT :

```
Standard deviation is : 0.920258
```

14.3. Recursion

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

EXAMPLE (Number Factorial) :

```
#include <stdio.h>

int factorial(unsigned int i) {
  if(i <= 1) {
    return 1;
  }
  return i * factorial(i - 1);
}

int  main() {
  int num;
  printf("Enter Number to find Factorial : ");
  scanf("%d",&num);
  printf("Factorial of %d is %d\n", num, factorial(num));
  return 0;
}
```

OUTPUT :

```
Enter Number to find Factorial : 5
Factorial of 5 is 120
```

EXAMPLE (Fibonacci Series) :

```
#include <stdio.h>

int fibonacci(int i) {

  if(i == 0) {
    return 0;
  }

  if(i == 1) {
    return 1;
  }
  return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

  int cnt;
  printf("Enter number for fibonacci count : ");
  scanf("%d", &cnt);

  for (cnt = 0; cnt < 10; cnt++) {
    printf("%d  ", fibonacci(cnt));
  }

  return 0;
}
```

OUTPUT :

```
Enter number for fibonacci count : 10
0  1  1  2  3  5  8  13  21  34
```

14.4. Inline Functions

Inline Function are those functions whose definitions are small and can be substituted at the place where its function call happens. Function substitution is totally compiler choice.

EXAMPLE :

```
#include <stdio.h>

// Inline function in C
inline int result_value()
{
        return 2;
}

// Driver code
int main()
{

        int ret;

        // inline function call
        ret = result_value();

        printf("Output is: %d\n", ret);
        return 0;
}
```

OUTPUT :

```
main.c:(.text+0xe): undefined reference to `result_value'
collect2: error: ld returned 1 exit status
```

This is one of the side effects of GCC the way it handles inline functions. When compiled, **GCC performs inline substitution as the part of optimisation**. So there is no function call present (*result_value*) inside main.
Normally GCC's file scope is "not extern linkage". That means inline function is never ever provided to the linker which is causing linker error, mentioned above.

How to remove this error?

To resolve this problem use "static" before inline. Using a **static** keyword forces the compiler to consider this inline function in the linker, and hence the program compiles and runs successfully.

EXAMPLE :

```
#include <stdio.h>

// Inline function in C
static inline int result_value()
{
        return 2;
}

// Driver code
int main()
{

        int ret;

        // inline function call
        ret = result_value();

        printf("Output is: %d\n", ret);
        return 0;
}
```

OUTPUT :

```
main.c:(.text+0xe): undefined reference to `result_value'
collect2: error: ld returned 1 exit status
```

14.5. _Noreturn Functions

After the removal of “noreturn” keyword, C11 standard (known as final draft) of C programming language introduce a new “_Noreturn” function specifier that specify that the function does not return to the function that it was called from. If the programmer tries to return any value from that function which is declared as _Noreturn type, then the compiler automatically generates a compile time error.

EXAMPLE :

```
// C program to show how _Noreturn type function behaves if it has a return
statement.
#include <stdio.h>
#include <stdlib.h>

// With return value
_Noreturn void view()
{
   return 10;
}

int main(void)
{
   printf("Ready to begin...");
   view();

   printf("NOT over till now");
   return 0;
}
```

OUTPUT :

```
main.c:8:12: warning: function declared ‘noreturn’ has a ‘return’ statement
    return 10;
           ^~
main.c:8:12: warning: ‘return’ with a value, in function returning void
main.c:6:16: note: declared here
 _Noreturn void view()
               ^~~~
main.c:8:12: warning: ‘noreturn’ function does return
    return 10;
           ^~
```

EXAMPLE :

```
// C program to illustrate the working
// of _Noreturn type function.
#include <stdio.h>
#include <stdlib.h>

// Nothing to return
_Noreturn void show()
{
        printf("BYE BYE");
}
int main(void)
{
        printf("Ready to begin...\n");
        show();

        printf("NOT over till now\n");
        return 0;
}
```

EXAMPLE :

```
Ready to begin...
BYE BYE
```

<u>15. Structure/Unions (define/access member)</u>

<u>15.1. Structure</u>

Arrays allow to define types of variables that can hold several data items of the same kind. Similarly structure is another user defined data type available in C that allows combining data items of different kinds.

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct [structure tag] {
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure –

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type. The following example shows how to use a structure in a program –

EXAMPLE :

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

struct Books {
        char  title[50];
        char  author[50];
        char  subject[100];
        int   book_id;
};

int main( )
{
        struct Books Book1;      /* Declare Book1 of type Book */
        struct Books Book2;      /* Declare Book2 of type Book */

        /* book 1 specification */
        strcpy( Book1.title, "C Programming");
        strcpy( Book1.author, "Nuha Ali");
        strcpy( Book1.subject, "C Programming Tutorial");
        Book1.book_id = 6495407;

        /* book 2 specification */
        strcpy( Book2.title, "Telecom Billing");
        strcpy( Book2.author, "Zara Ali");
        strcpy( Book2.subject, "Telecom Billing Tutorial");
        Book2.book_id = 6495700;
```

```
            /* print Book1 info */
            printf( "Book 1 title : %s\n", Book1.title);
            printf( "Book 1 author : %s\n", Book1.author);
            printf( "Book 1 subject : %s\n", Book1.subject);
            printf( "Book 1 book_id : %d\n", Book1.book_id);

            /* print Book2 info */
            printf( "Book 2 title : %s\n", Book2.title);
            printf( "Book 2 author : %s\n", Book2.author);
            printf( "Book 2 subject : %s\n", Book2.subject);
            printf( "Book 2 book_id : %d\n", Book2.book_id);

            return 0;
}
```

OUTPUT :

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

EXAMPLE :

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

struct Books {
            char  title[50];
            char  author[50];
            char  subject[100];
            int   book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( )
{
            struct Books Book1;       /* Declare Book1 of type Book */
            struct Books Book2;       /* Declare Book2 of type Book */
            /* book 1 specification */
            strcpy( Book1.title, "C Programming");
            strcpy( Book1.author, "Nuha Ali");
            strcpy( Book1.subject, "C Programming Tutorial");
            Book1.book_id = 6495407;

            /* book 2 specification */
            strcpy( Book2.title, "Telecom Billing");
            strcpy( Book2.author, "Zara Ali");
            strcpy( Book2.subject, "Telecom Billing Tutorial");
            Book2.book_id = 6495700;

            /* print Book1 info */
            printBook( Book1 );

            /* Print Book2 info */
            printBook( Book2 );

            return 0;
}

void printBook( struct Books book )
{
            printf( "Book title : %s\n", book.title);
            printf( "Book author : %s\n", book.author);
            printf( "Book subject : %s\n", book.subject);
            printf( "Book book_id : %d\n", book.book_id);
}
```

OUTPUT :

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable –

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

```
struct_pointer->title;
```

Let us rewrite the above example using a structure pointer.

EXAMPLE :

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

struct Books {
        char  title[50];
        char  author[50];
        char  subject[100];
        int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( )
{
        struct Books Book1;      /* Declare Book1 of type Book */
        struct Books Book2;      /* Declare Book2 of type Book */

        /* book 1 specification */
        strcpy( Book1.title, "C Programming");
        strcpy( Book1.author, "Nuha Ali");
        strcpy( Book1.subject, "C Programming Tutorial");
        Book1.book_id = 6495407;

        /* book 2 specification */
        strcpy( Book2.title, "Telecom Billing");
        strcpy( Book2.author, "Zara Ali");
        strcpy( Book2.subject, "Telecom Billing Tutorial");
        Book2.book_id = 6495700;

        /* print Book1 info by passing address of Book1 */
        printBook( &Book1 );

        /* print Book2 info by passing address of Book2 */
        printBook( &Book2 );
```

```
		return 0;
	}

	void printBook( struct Books *book )
	{
		printf( "Book title : %s\n", book->title);
		printf( "Book author : %s\n", book->author);
		printf( "Book subject : %s\n", book->subject);
		printf( "Book book_id : %d\n", book->book_id);
	}
```

OUTPUT :

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

15.2. Unions

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union [union tag] {
  member definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str –

```
union Data {
		int i;
```

```
            float f;
            char str[20];
    } data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the **maximum space** which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

EXAMPLE :

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

    union Data {
            int i;
            float f;
            char str[20];
    };

    int main( ) {

            union Data data;

            printf( "Memory size occupied by data : %d\n", sizeof(data));

            return 0;
    }
```

OUTPUT :

```
    Memory size occupied by data : 20
```

Accessing Union Members

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. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program

EXAMPLE :

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

```
union Data {
        int i;
        float f;
        char str[20];
};

int main( ) {

        union Data data;

        data.i = 10;
        data.f = 220.5;
        strcpy( data.str, "C Programming");

        printf( "data.i : %d\n", data.i);
        printf( "data.f : %f\n", data.f);
        printf( "data.str : %s\n", data.str);

        return 0;
}
```

OUTPUT :

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

EXAMPLE :

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

union Data {
        int i;
        float f;
        char str[20];
};

int main( )
{
        union Data data;
```

```
                data.i = 10;
                printf( "data.i : %d\n", data.i);

                data.f = 220.5;
                printf( "data.f : %f\n", data.f);

                strcpy( data.str, "C Programming");
                printf( "data.str : %s\n", data.str);

                return 0;
        }
```

OUTPUT :

```
        data.i : 10
        data.f : 220.500000
        data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

16. Dynamic memory allocation

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of the stdlib.h header file.

16.1. malloc()

malloc() allocates a single block of requested memory.It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

EXAMPLE :

```
#include<stdio.h>

#include<stdlib.h>

int main()

{

  int n,i,*ptr,sum=0;

  printf("Enter number of elements: ");

  scanf("%d",&n);
```

```
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
       printf("Sorry! unable to allocate memory");
       exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
       scanf("%d",ptr+i);
       sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
 return 0;
 }
```

OUTPUT:

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

16.2. calloc()

calloc() allocates multiple blocks of requested memory.It initially initializes all bytes to zero.It returns NULL if memory is not sufficient.

EXAMPLE :

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
```

```
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
       printf("Sorry! unable to allocate memory");
       exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
       scanf("%d",ptr+i);
       sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
 return 0;
 }
```

OUTPUT:

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

16.3. realloc()

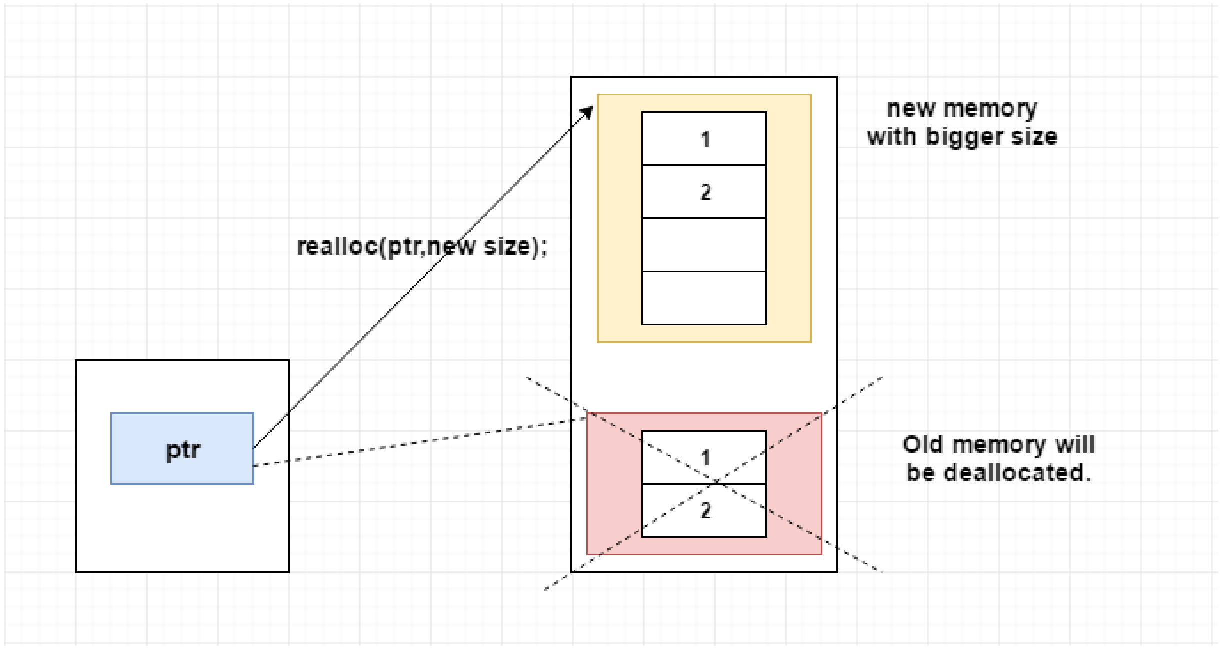
realloc() reallocates the memory occupied by malloc() or calloc() functions.If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

EXAMPLE :

```
#include<stdio.h>
//To use realloc in our program
#include<stdlib.h>

int main()
{
   int *ptr,i;

   //allocating memory for only 1 integer
   ptr = malloc(1*sizeof(int));

   ptr[0] = 1;
   //realloc memory size to store 3 integers
   ptr = realloc(ptr, 3 * sizeof(int));
   ptr[1] = 2;
   ptr[2] = 3;

   //printing values
   for(i = 0; i < 3; i++)
        printf("%d ",ptr[i]);

   return 0;
}
```

OUTPUT :

```
1 2 3
```



16.4. free()

free() frees the dynamically allocated memory.The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until the program exits.

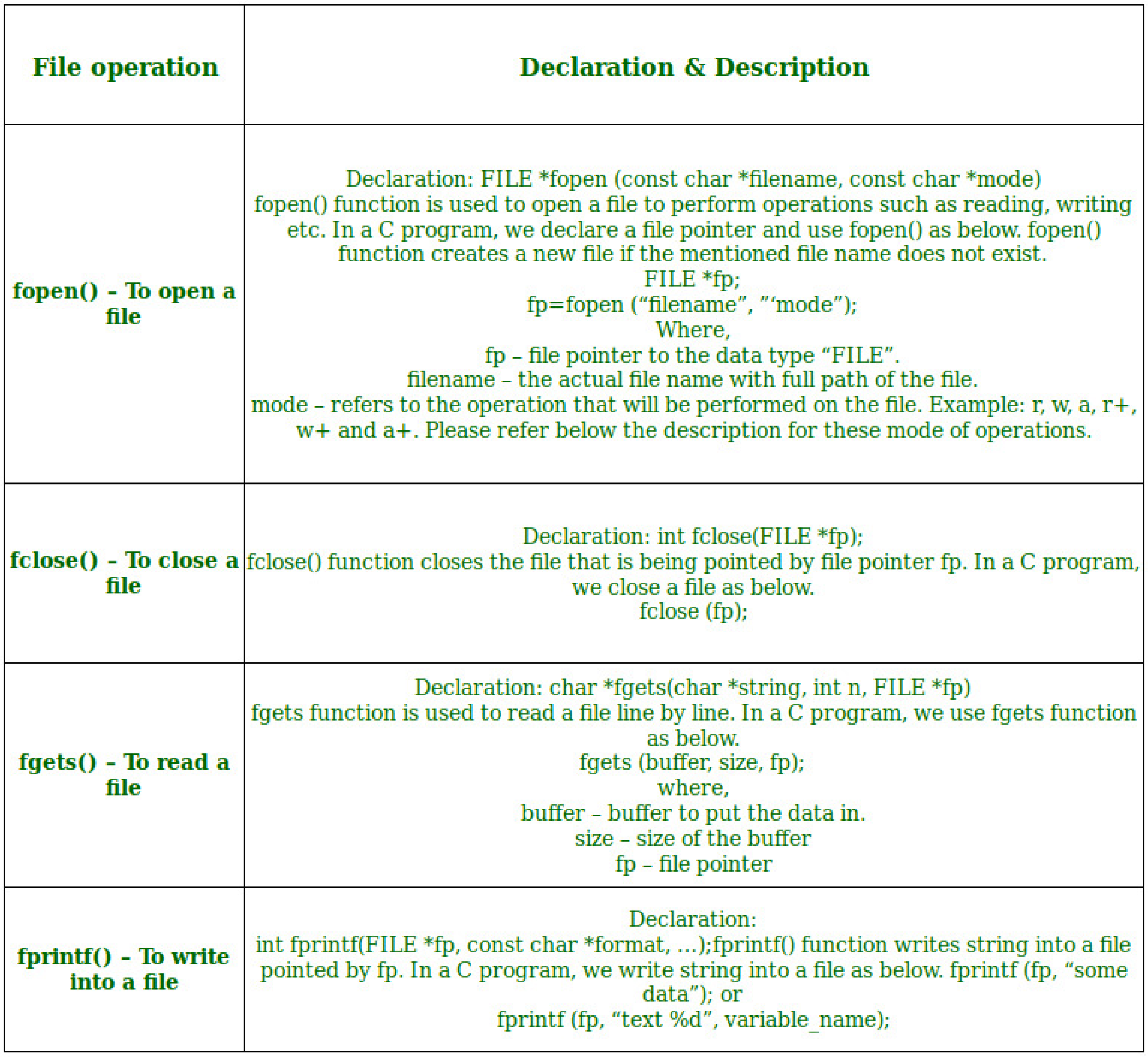
static memory allocation	**dynamic memory allocation**
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing a program.	memory can be increased while executing a program.
used in an array.	used in the linked list.

17. File handling in C

So far the operations using C program are done on a prompt / terminal which is not stored anywhere. But in the software industry, most of the programs are written to store the information fetched from the program. One such way is to store the fetched information in a file. Different operations that can be performed on a file are:

1. Creation of a new file (**fopen with attributes as “a” or “a+” or “w” or “w++”)**
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf or fgets**)
4. Writing to a file (**fprintf or fputs**)
5. Moving to a specific location in a file (**fseek, rewind**)
6. Closing a file (**fclose**)

The text in the brackets denotes the functions used for performing those operations.



File operation	Declaration & Description
fopen() - To open a file	Declaration: FILE *fopen (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=fopen (“filename”, ”‘mode”); Where, fp – file pointer to the data type “FILE”. filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.

fclose() - To close a file	Declaration: int fclose(FILE *fp);
	fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below.
	fclose (fp);

fgets() - To read a file	Declaration: char *fgets(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. fgets (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
fprintf() - To write into a file	Declaration: int fprintf(FILE *fp, const char *format, ...);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, “some data”); or fprintf (fp, “text %d”, variable_name);

Opening or creating file :

For opening a file, the fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

File opening modes in C:

Mode	**Description**
r	opens a text file in read mode
w	opens a text file in write mode

a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

For performing the operations on the file, a special pointer called File pointer is used which is declared as

```
FILE *filePointer;
```

So, the file can be opened as

```
filePointer = fopen(“fileName.txt”, “w”)
```

The second parameter is mode , which can be changed to contain all the attributes listed in the above table.

Reading from a file –

The file read operations can be performed using functions fscanf or fgets. Both the functions performed the same operations as that of scanf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.And the code snippet for reading a file is as:

```
FILE * filePointer;
```

```
filePointer = fopen(“fileName.txt”, “r”);
fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);
```

Writing a file –:

The file write operations can be performed by the functions fprintf and fputs with similarities to read operations. The snippet for writing to a file is as :

```
FILE *filePointer ;

filePointer = fopen(“fileName.txt”, “w”);

fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);
```

Closing a file –:

After every successful file operation, you must always close a file. For closing a file, you have to use the fclose function. The snippet for closing a file is given as :

```
FILE *filePointer ;
filePointer= fopen(“fileName.txt”, “w”);

---------- Some file Operations -------

fclose(filePointer)
```

Moving in a file –:

ftell() : Declaration is, long int ftell(FILE *fp). ftell function is used to get the current position of the file pointer. In a C program, we use ftell(fp); where fp is a file pointer**.**

fseek() : Declaration: int fseek(FILE *fp, long int offset, int whence)fseek() function is used to move file pointer position to the given location.

Where, fp – file pointer
offset – Number of bytes/characters to be offset/moved from whence/the current file pointer position
whence – This is the current file pointer position from where offset is added.

Below 3 constants are used to specify this.

SEEK_SET – It moves the file pointer position to the beginning of the file.
SEEK_CUR – It moves the file pointer position to a given location.
SEEK_END – It moves the file pointer position to the end of file.

rewind() : Declaration: void rewind(FILE *fp).rewind function is used to move file pointer position to the beginning of the file. In a C program, we use rewind() as below. rewind(fp);

18. The Pre-processor

18.1. Conditional Compilation

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- The **#if and #elif directives**, which conditionally include or suppress portions of source code, depending on the result of a constant expression
- The **#ifdef directive**, which conditionally includes source text if a macro name is defined
- The **#ifndef directive**, which conditionally includes source text if a macro name is not defined
- The **#else directive**, which conditionally includes source text if the previous #if, #ifdef, #ifndef, or #elif test fails
- The **#endif directive**, which ends conditional text

For each *#if, #ifdef*, and *#ifndef* directive, there are zero or more *#elif* directives, zero or one *#else* directive, and one matching *#endif* directive. All the matching directives are considered to be at the same nesting level.

Conditional compilation directives can be nested. A *#else*, if present, can be matched unambiguously because of the required *#endif*.

18.2. Include guards and #undef

18.2.1. Include guards

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file
```

```
#endif
```

This ensures that when you *#include "my-header-file.h"* in multiple places, you don't get duplicate declarations of functions, variables, etc. Imagine the following hierarchy of files:

header-1.h

```
typedef struct {
   ...
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);

main.c

#include "header-1.h"
#include "header-2.h"

int main() {
   // do something
}
```

This code has a serious problem: the detailed contents of *MyStruct* is defined twice, which is not allowed. This would result in a compilation error that can be difficult to track down, since one header file includes another. If you instead did it with header guards:

header-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
   ...
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

header-2.h

```
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif

main.c

#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

This would then expand to:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #defined before.
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif
```

```
int main() {
   // do something
}
```

When the compiler reaches the second inclusion of header-1.h, *HEADER_1_H* was already defined by the previous inclusion. Ergo, it boils down to the following:

```
#define HEADER_1_H

typedef struct {
   …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
   // do something
}
```

And thus there is no compilation error.

18.2.1. #undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax: #undef token

Let's see a simple example to define and undefine a constant.

EXAMPLE :

```
#include <stdio.h>
#define PI 3.14
#undef PI
main() {
        printf("%f",PI);
}
```

OUTPUT :

```
main.c:5:17: error: ‘PI’ undeclared (first use in this function)
   printf("%f",PI);
               ^~
```

The #undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variables. But before being undefined, it was used as a square variable.

EXAMPLE :

```
#include <stdio.h>

#define number 5

int square=number*number;

#undef number

int main() {

        printf("%d",square);

        return 0;

}
```

OUTPUT :

```
25
```

18.3. #pragma and #error

18.3.1. #pragma

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

#pragma startup and #pragma exit: These directives help 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()).
Note: Below program will not work with GCC compilers.

EXAMPLE :

```
#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()
{
	printf("Inside main()\n");

	return 0;
}
```

OUTPUT :

```
Inside main()
```

This happens because GCC does not support #pragma startup or exit. However, you can use the below code for a similar output on GCC compilers.

EXAMPLE :

```
#include<stdio.h>

void func1();
void func2();

void __attribute__((constructor)) func1();
void __attribute__((destructor)) func2();

void func1()
{
	printf("Inside func1()\n");
}

void func2()
{
	printf("Inside func2()\n");
}

int main()
{
	printf("Inside main()\n");
```

```
        return 0;
}
```

OUTPUT :

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

#pragma GCC poison: 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.

EXAMPLE :

```
// Program to illustrate the
// #pragma GCC poison directive

#include<stdio.h>

#pragma GCC poison printf

int main()
{
        int a=10;

        if(a==10)
        {
                printf("Pragma poison example");
        }
        else
                printf("Terminate");

        return 0;
}
```

OUTPUT :

```
prog.c: In function 'main':
prog.c:14:9: error: attempt to use poisoned "printf"
      printf("Pragma poison example");
      ^
prog.c:17:9: error: attempt to use poisoned "printf"
      printf("Terminate");
      ^
```

18.3.2. #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

EXAMPLE :

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

OUTPUT :

```
Compile Time Error: First include then compile
```

But, if you include math.h, it does not give errors.

EXAMPLE :

```
#include<stdio.h>
#include<math.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

OUTPUT :

```
2.645751
```

19. Macros

19.1. Macros vs. Functions

Macros are **pre-processed** which means that all the macros would be processed before your program compiles. However, functions are **not preprocessed but compiled**.

In macros, no type checking(incompatible operand, etc.) is done and thus use of macros can lead to errors/side-effects in some cases. However, this is not the case with functions. Also, macros do not check for compilation errors (if any). Consider the following two codes:

Macros:

EXAMPLE :

```
#include<stdio.h>
#define CUBE(b) b*b*b
int main()
{
        printf("%d", CUBE(1+2));
        return 0;
}
```

OUTPUT :

```
7
```

Note: This macro is expanded as below

CUBE(1+2) === 1+2*1+2*1+2 which is equal to 7 [correct but unexpected result]

To fix this we need to replace #define CUBE(b) b*b*b as #define CUBE(b) (b)*(b)*(b). With updated macro, CUBE(1+2) will be expanded as

CUBE(1+2) === (1+2)*(1+2)*(1+2) which is equal to 27 [correct and expected result]

Functions:

EXAMPLE :

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

OUTPUT :

```
27
```

Macros are no longer recommended as they cause the following issues. There is a better way in modern compilers that is inline functions and const variables.

Disadvantages of macros :

a) There is no type checking

b) Difficult to debug as they cause simple replacement.

c) Macro don't have a namespace, so a macro in one section of code can affect another section.

d) Macros can cause side effects as shown in above CUBE() example.

19.2. Creating your own Macros

A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive.
Here's an example.

```
#define c 299792458  // speed of light
```

Here, when we use **c** in our program, it is replaced with 299792458

19.3. Preprocessor operators

The C preprocessor offers the following operators to help create macros –

The Macro Continuation (\) Operator :

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example –

```
#define  message_for(a, b)  \
  printf(#a " and " #b ": We love you!\n")
```

The Stringize (#) Operator :

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list.

EXAMPLE :

```
#include <stdio.h>

#define  message_for(a, b)  \
  printf(#a " and " #b ": We love you!\n")

int main(void) {
  message_for(Nidhi, Vidhi);
  return 0;
}
```

OUTPUT :

Nidhi and Vidhi: We love you!

The Token Pasting (##) Operator :

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example –

EXAMPLE :

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
  int token34 = 40;
  tokenpaster(34);
  return 0;
}
```

OUTPUT :

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor –

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting.

The Defined() Operator :

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows –

EXAMPLE :

```
#include <stdio.h>

#if !defined (MESSAGE)
  #define MESSAGE "Good Morning!"
#endif

int main(void) {
  printf("Here is the message: %s\n", MESSAGE);
  return 0;
}
```

OUTPUT :

```
  Here is the message: Good Morning!
```

19.4. Predefined Macros

ANSI C defines many predefined macros that can be used in C programs.

No.	Macro	Description
1	_DATE_	represents the current date in "MMM DD YYYY" format.
2	_TIME_	represents the current time in "HH:MM:SS" format.
3	_FILE_	represents the current file name.
4	_LINE_	represents the current line number.
5	_STDC_	It is defined as 1 when the compiler complies with the ANSI standard.

EXAMPLE :

```
#include<stdio.h>
 int main(){
          printf("File :%s\n", __FILE__ );
          printf("Date :%s\n", __DATE__ );
          printf("Time :%s\n", __TIME__ );
          printf("Line :%d\n", __LINE__ );
          printf("STDC :%d\n", __STDC__ );
          return 0;
 }
```

OUTPUT :

```
  File :main.c
  Date :Aug  2 2021
  Time :09:51:14
```

```
Line :6
STDC :1
```

20. GCC Compiler Options

20.1. Debugging with preprocessor

The output of the preprocessing stage can be produced using the -E option.

```
$ gcc -E main.c > main.i
```

The gcc command produces the output on stdout so you can redirect the output in any file. In our case(above), the file main.i would contain the preprocessed output.

20.2. Debugging with GDB

Gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface.It helps you to poke around inside your C programs while they are executing and also allows you to see what exactly happens when your program crashes. GDB operates on executable files which are binary files produced by the compilation process.

To prepare your program for debugging with gdb, you must compile it with the -g flag. So, if your program is in a source file called main.c and you want to put the executable in the file test_output, then you would compile with the following command:

```
gcc -g -o test_output main.c
```

To start gdb, just type gdb at the unix prompt. Gdb will give you a prompt. From that prompt you can run your program, look at variables, etc., using the commands listed below. Or, you can start gdb and give it the name of the program executable you want to debug by saying

```
gdb ./test_output
```

To exit the program just type quit at the (gdb) prompt (actually just typing q is good enough).This is a brief description of some of the most commonly used features of gdb.

- run or r –> executes the program from start to end.
- break n or b n –> sets breakpoint on a particular line number n.
- disable -> disable a breakpoint.
- enable –> enable a disabled breakpoint.
- next or n -> executes the next line of code, but doesn't dive into functions.
- step or s –> go to the next instruction, diving into the function.
- list or l –> displays the code.
- print or p –> used to display the stored value.
- quit or q –> exits out of gdb.
- clear –> to clear all breakpoints.
- continue or c –> continue normal execution.
- info b –> to see the breakpoints
- Enter –> execute the previously executed command again.
- bt –> print backtrace of all stack frames, or innermost COUNT frames.
- quit –> exit from gdb debugger.

20.3. core files (crash analysis by mapping register with elf file)

A core dump is the in-memory image of the process when it crashed. It includes the program segments, the stack, the heap and other data. You'll still need the original program in order to make sense of the contents: the symbol tables and other data make the raw addresses and structures in the memory image meaningful.

By using binutils tools like readelf and objdump, we can bulk dump information contained in the core file such as the memory state. Most/all of it must also be visible through GDB, but those binutils tools offer a more bulk approach which is convenient for certain use cases, while GDB is more convenient for a more interactive exploration.
First:

```
file core
```

tells us that the core file is actually an ELF file:

```
core: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from './main.out'
```

This is why we are able to inspect it more directly with usual binutils tools.
A quick look at the ELF standard shows that there is actually an ELF type dedicated to it:

```
Elf32_Ehd.e_type == ET_CORE
```

Further format information can be found at:

```
man 5 core
```

Then:

```
readelf -Wa core
```

gives some hints about the file structure. Memory appears to be contained in regular program headers:

```
Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  NOTE           0x000468 0x0000000000000000 0x0000000000000000 0x000b9c 0x000000
0
  LOAD           0x002000 0x0000000000400000 0x0000000000000000 0x001000 0x001000
R E 0x1000
  LOAD           0x003000 0x0000000000600000 0x0000000000000000 0x001000 0x001000
R   0x1000
  LOAD           0x004000 0x0000000000601000 0x0000000000000000 0x001000 0x001000
RW  0x1000
```

objdump can easily dump all memory with:

```
objdump -s core
```

which contains:

```
Contents of section load1:

 4007d0 01000200 73747269 6e672069 6e207465  ....string in te
 4007e0 78742073 65676d65 6e740074 65787420  xt segment.text

Contents of section load15:

 7ffec6739220 73747269 6e672069 6e206461 74612073  string in data s
 7ffec6739230 65676d65 6e740000 00a8677b 9c6778cd  egment....g{.gx.

Contents of section load4:

 1612010 73747269 6e672069 6e206d6d 61702073  string in mmap s
 1612020 65676d65 6e740000 11040000 00000000  egment..........
```

20.4. Profiling

The GNU profiler gprof is a useful tool for measuring the performance of a program--it records the number of calls to each function and the amount of time spent there, on a per-function basis. Functions which consume a large fraction of the run-time can be identified easily from the output of gprof. Efforts to speed up a program should concentrate first on those functions which dominate the total run-time.

The first step in generating profile information for your program is to compile and link it with profiling enabled.

To compile a source file for profiling, specify the `-pg' option when you run the compiler. (This is in addition to the options you normally use.)

To link the program for profiling, if you use a compiler such as cc to do the linking, simply specify `-pg' in addition to your usual options. The same option, `-pg', alters either compilation or linking to do what is necessary for profiling. Here are examples:

```
gcc -g -c myprog.c utils.c -pg
cc -o myprog myprog.o utils.o -pg
```

The `-pg' option also works with a command that both compiles and links:

```
gcc -o myprog myprog.c utils.c -g -pg
```

20.5. Static Analysis (MISRA Coding standard)

MISRA provides coding standards for developing safety-critical systems. MISRA is made up of manufacturers, component suppliers, and engineering consultancies. Experts from Perforce's static code analysis team (formerly PRQA) are members of MISRA, too. MISRA first developed coding guidelines in 1998. These were specific to the C programming language. Since then, MISRA has added a coding standard for C++.

You can use MISRA standards to ensure your code is:

- Safe
- Secure
- Reliable
- Portable

MISRA C is the most widely used set of coding guidelines for C around the world. There have been three releases of the MISRA C standard.

MISRA C:1998
MISRA C:1998 was published in 1998 and remains widely used today. It was written for C90. There are 127 coding rules, including:

Rule 59

The statement forming the body of an "if", "else if", "else", "while", "do ... while", or "for" statement shall always be enclosed in braces

MISRA C:2004
MISRA C:2004 is the second edition of MISRA C, published in 2004. It was written for C90. There are 142 coding rules, including:

Rule 14.9

An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Rule 14.10

All if … else if constructs shall be terminated with an else clause.

MISRA C:2012
MISRA C:2012 is the third edition of MISRA C, published in 2012. It was written for C99. There are 143 rules, including:

Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

20.6. Memory leak test (memcheck, valgrind memory leak check)

Valgrind Memcheck is a tool that detects memory leaks and memory errors. Some of the most difficult C bugs come from mismanagement of memory: allocating the wrong size, using an uninitialized pointer, accessing memory after it was freed, overrunning a buffer, and so on. These types of errors are tricky, as they can provide little debugging information, and tracing the observed problem back to the underlying root cause can be challenging. Valgrind is helpful here.

Memory Errors Vs. Memory Leaks

Valgrind reports two types of issues: memory *errors* and memory *leaks*. When a program dynamically allocates memory and forgets to later free it, it creates a leak. A memory leak generally won't cause a program to misbehave, crash, or give wrong answers, and is not an urgent situation. A memory error, on the other hand, is a red alert. Reading uninitialized memory, writing past the end of a piece of memory, accessing freed memory, and other memory errors can have significant consequences. Memory errors should never be treated casually or ignored. Although this guide describes how to use Valgrind to find both, keep in mind that errors are by far the primary concern, and memory leaks can generally be resolved later.

Running A Program Under Valgrind

Like the debugger, Valgrind runs on your executable, so be sure you have compiled an up-to-date copy of your program. Run it like this, for example, if your program is named memoryLeak:

```
$ valgrind ./memoryLeak
```

Valgrind will then start up and run the specified program inside of it to examine it. If you need to pass command-line arguments, you can do that as well:

```
$ valgrind ./memoryLeak red blue
```

When it finishes, Valgrind will print a summary of its memory usage. If all goes well, it'll look something like this:

```
==4649== ERROR SUMMARY: 0 errors from 0 contexts
==4649== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4649== malloc/free: 10 allocs, 10 frees, 2640 bytes allocated.
==4649== For counts of detected errors, rerun with: -v
==4649== All heap blocks were freed -- no leaks are possible.
```

This is what you're shooting for: no errors and no leaks. Another useful metric is the number of allocations and total bytes allocated. If these numbers are in the same ballpark as our sample (you can run the solution under valgrind to get a baseline), you'll know that your memory efficiency is right on target.

20.7. Cpp checker

Cppcheck is a command-line tool that tries to detect bugs that your C/C++ compiler doesn't see. It is versatile, and can check non-standard code including various compiler extensions, inline assembly code, etc. Its internal preprocessor can handle includes, macros, and several preprocessor commands. While Cppcheck is highly configurable, you can start using it just by giving it a path to the source code.

Cppcheck is an analysis tool for C/C++ code. It provides unique code analysis to detect bugs and focuses on detecting undefined behaviour and dangerous coding constructs. The goal is to detect only real errors in the code, and generate as few false positives (wrongly reported warnings) as possible. Cppcheck is designed to analyze your C/C++ code even if it has non-standard syntax, as is common in for example embedded projects.

Supported code and platforms:

- Cppcheck checks non-standard code that contains various compiler extensions, inline assembly code, etc.
- Cppcheck should be compilable by any compiler that supports C++11 or later.
- Cppcheck is cross platform and is used in various posix/windows/etc environments.

21. Advanced Pointers

21.1. Double pointers (pointer to a pointer)

We already know that a pointer points to a location in memory and thus used to store the address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as double pointers.

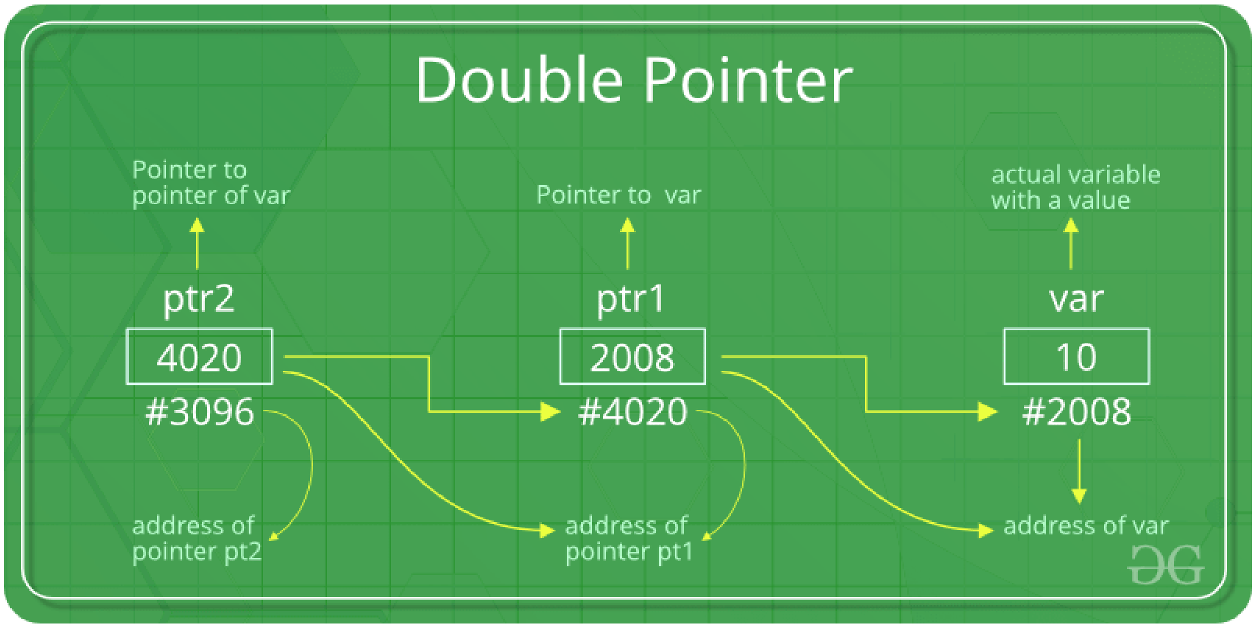
How to declare a pointer to pointer in C?

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

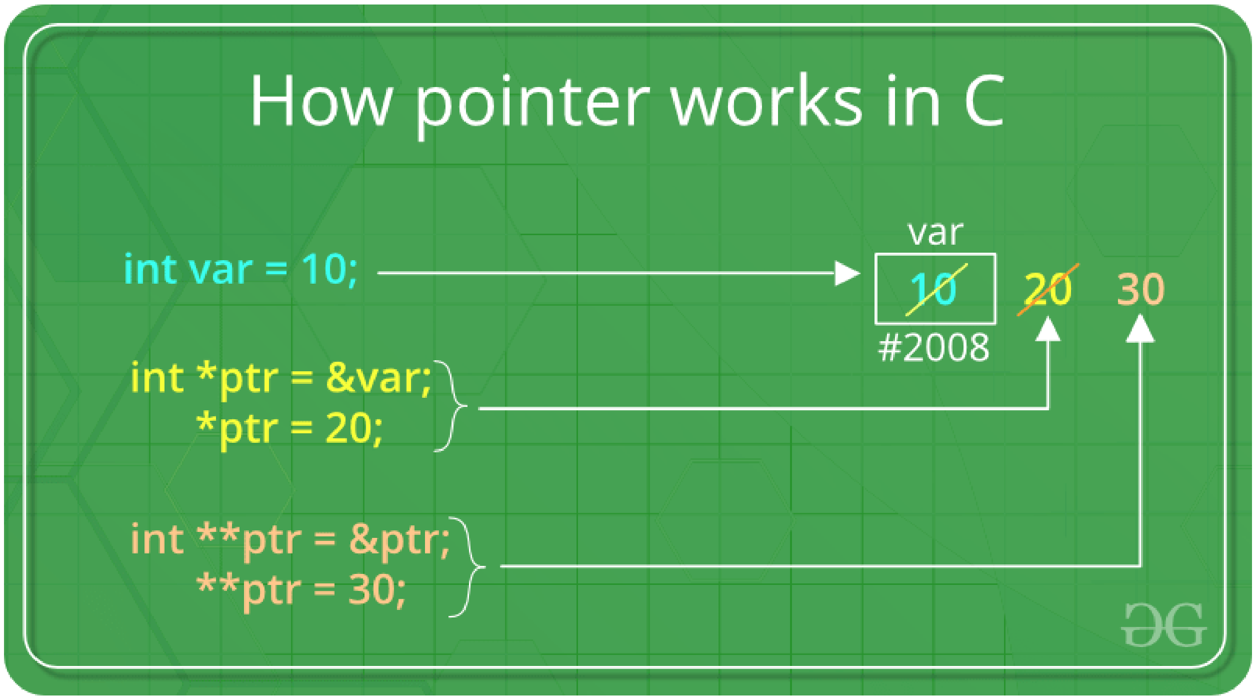
Syntax:

```
int **ptr;    // declaring double pointers
```

Below diagram explains the concept of Double Pointers:



The above diagram shows the memory representation of a pointer to pointer. The first pointer ptr1 stores the address of the variable and the second pointer ptr2 stores the address of the first pointer.



EXAMPLE :

```
#include <stdio.h>

// C program to demonstrate pointer to pointer
int main()
{
	int var = 789;

	// pointer for var
	int *ptr2;

	// double pointer for ptr2
	int **ptr1;

	// storing address of var in ptr2
	ptr2 = &var;
	// Storing address of ptr2 in ptr1
	ptr1 = &ptr2;

	// Displaying value of var using
	// both single and double pointers
	printf("Value of var = %d\n", var );
	printf("Value of var using single pointer = %d\n", *ptr2 );
	printf("Value of var using double pointer = %d\n", **ptr1);

return 0;
}
```

OUTPUT :

```
Value of var = 789
Value of var using single pointer = 789
Value of var using double pointer = 789
```

21.2. Function pointers

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

EXAMPLE :

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
	printf("Value of a is %d\n", a);
}

int main()
{
	// fun_ptr is a pointer to function fun()
	void (*fun_ptr)(int) = &fun;
	/* The above line is equivalent of following two
	void (*fun_ptr)(int);
	fun_ptr = &fun;
	*/

	// Invoking fun() using fun_ptr
	(*fun_ptr)(10);

	return 0;
}
```

OUTPUT :

```
Value of a is 10
```

If we remove bracket, then the expression “void (*fun_ptr)(int)” becomes “void *fun_ptr(int)” which is a declaration of a function that returns a void pointer.

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate deallocate memory using function pointers.

3) A function’s name can also be used to get functions’ addresses. For example, in the below program, we have removed the address operator ‘&’ in assignment. We have also changed function call by removing *, the program still works.

EXAMPLE :

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
	printf("Value of a is %d\n", a);
}
```

```
int main()
{
        void (*fun_ptr)(int) = fun; // & removed

        fun_ptr(10); // * removed

        return 0;
}
```

OUTPUT :

```
Value of a is 10
```

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of a switch case. For example, in the below program, the user is asked for a choice between 0 and 2 to do different tasks.

EXAMPLE :

```
#include <stdio.h>
void add(int a, int b)
{
        printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
        printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
        printf("Multiplication is %d\n", a*b);
}

int main()
{
        // fun_ptr_arr is an array of function pointers
        void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
        unsigned int ch, a = 15, b = 10;

        printf("Enter Choice: 0 for add, 1 for subtract and 2 "
                "for multiply\n");
        scanf("%d", &ch);

        if (ch > 2) return 0;

        (*fun_ptr_arr[ch])(a, b);

        return 0;
}
```

OUTPUT :

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

EXAMPLE :

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
```

```
void wrapper(void (*fun)())
{
        fun();
}

int main()
{
        wrapper(fun1);
        wrapper(fun2);
        return 0;
}
```

OUTPUT :

```
Fun1
Fun2
```

21.3. Void pointers

The void pointer in C is a pointer which is not associated with any data types. It points to some data location in the storage means points to the address of variables. It is also called a general purpose pointer. In C, malloc() and calloc() functions return void * or generic pointers.

It has some limitations −

1) Pointer arithmetic is not possible with void pointer due to its concrete size.

2) It can’t be used as dereferenced.

Algorithm

```
Begin
   Declare a of the integer datatype.
      Initialize a = 7.
   Declare b of the float data type.
      Initialize b = 7.6.
   Declare a pointer p as void.
   Initialize p pointer to a.
   Print “Integer variable is”.
      Print the value of a using pointer p.
   Initialize p pointer to b.
   Print “Float variable is”.
      Print the value of b using pointer p
```

```
End.
```

Here is a simple example –

EXAMPLE :

```
#include<stdlib.h>
int main() {
   int a = 7;
   float b = 7.6;
   void *p;
   p = &a;
   printf("Integer variable is = %d", *( (int*) p) );
   p = &b;
   printf("\nFloat variable is = %f", *( (float*) p) );
   return 0;

}
```

OUTPUT :

```
Integer variable is = 7
Float variable is = 7.600000
```

21.4. Null pointers

A Null Pointer is a pointer that does not point to any memory location. It stores the base address of the segment. The null pointer basically stores the Null value while void is the type of the pointer.

A null pointer is a special reserved value which is defined in a **stddef** header file. Here, Null means that the pointer is referring to the 0th memory location.

If we do not have any address which is to be assigned to the pointer, then it is known as a null pointer. When a NULL value is assigned to the pointer, then it is considered as a **Null pointer**.

Applications of Null Pointer

Following are the applications of a Null pointer:

- It is used to initialize a pointer variable when the pointer does not point to a valid memory address.
- It is used to perform error handling with pointers before dereferencing the pointers.

- It is passed as a function argument and to return from a function when we do not want to pass the actual memory address.

Examples of Null Pointer

```
int *ptr=(int *)0;

float *ptr=(float *)0;

char *ptr=(char *)0;

double *ptr=(double *)0;

char *ptr='\0';

int *ptr=NULL;
```

Let's look at the situations where we need to use the null pointer.

- **When we do not assign any memory address to the pointer variable.**

```
#include <stdio.h>
int main()
{
         int *ptr;
        printf("Address: %d", ptr); // printing the value of ptr.
        printf("Value: %d", *ptr); // dereferencing the illegal pointer
        return 0;
}
```

In the above code, we declare the pointer variable *ptr, but it does not contain the address of any variable. The dereferencing of the uninitialized pointer variable will show the compile-time error as it does not point to any variable. According to the stack memory concept, the local variables of a function are stored in the stack, and if the variable does not contain any value, then it shows the garbage value. The above program shows some unpredictable results and causes the program to crash.

Therefore, we can say that keeping an uninitialized pointer in a program can cause serious harm to the computer.

How to avoid the above situation?

We can avoid the above situation by using the Null pointer. A null pointer is a pointer pointing to the 0th memory location, which is a reserved memory and cannot be dereferenced.

EXAMPLE :

```
#include <stdio.h>
int main()
{
   int *ptr=NULL;
   if(ptr!=NULL)
   {
      printf("value of ptr is : %d",*ptr);
   }
   else
   {
      printf("Invalid pointer");
   }
  return 0;
}
```

OUTPUT :

```
Invalid pointer
```

In the above code, we create a pointer *ptr and assigns a NULL value to the pointer, which means that it does not point any variable. After creating a pointer variable, we add the condition in which we check whether the value of a pointer is null or not.

- **When we use the malloc() function.**

```
#include <stdio.h>

int main()

{

   int *ptr;

   ptr=(int*)malloc(4*sizeof(int));

   if(ptr==NULL)

   {

      printf("Memory is not allocated");

   }

   else

   {

      printf("Memory is allocated");

   }

   return 0;

}
```

OUTPUT :

```
Memory is allocated
```

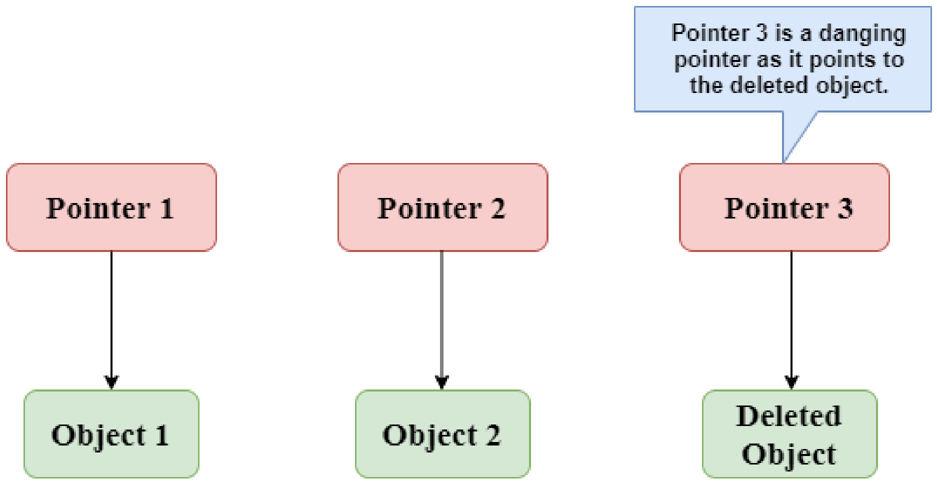
In the above code, we use the library function, i.e., malloc(). As we know, that malloc() function allocates the memory; if malloc() function is not able to allocate the memory, then it returns the NULL pointer. Therefore, it is necessary to add the condition which will check whether the value of a pointer is null or not, if the value of a pointer is not null means that the memory is allocated.

21.5. Dangling pointers

The most common bugs related to pointers and memory management is dangling/wild pointers. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system. If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then dereferenced the dangling pointer will cause the segmentation faults.

Let's observe the following examples.


Pointer 3 is a danging pointer as it points to the deleted object.
Pointer 1
Pointer 2
Pointer 3
Object 1
Object 2
Deleted Object

In the above figure, we can observe that the **Pointer 3** is a dangling pointer. **Pointer 1** and **Pointer 2** are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. **Pointer 3** is a dangling pointer as it points to the de-allocated object.

Let's understand the dangling pointer through some C programs.

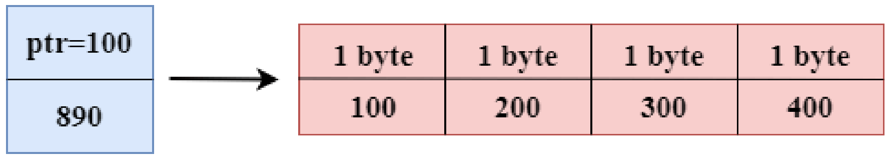
Using free() function to deallocate the memory.

```
#include <stdio.h>

int main()
```

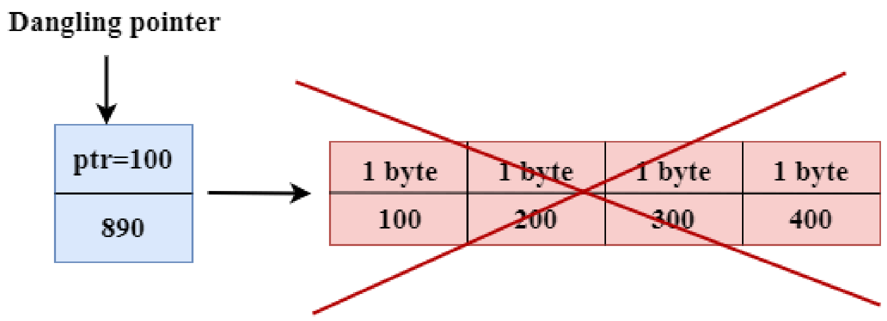
```
{

  int *ptr=(int *)malloc(sizeof(int));

  int a=560;

  ptr=&a;

  free(ptr);

  return 0;

}
```

In the above code, we have created two variables, i.e., *ptr and a where 'ptr' is a pointer and 'a' is an integer variable. The *ptr is a pointer variable which is created with the help of **malloc()** function. As we know that malloc() function returns void, so we use int * to convert void pointer into int pointer.

The statement **int *ptr=(int *)malloc(sizeof(int));** will allocate the memory with 4 bytes shown in the below image:



The statement **free(ptr)** de-allocates the memory as shown in the below image with a cross sign, and the 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:

ptr=NULL
890
ptr is not a dangling pointer

Variable goes out of the scope

When the variable goes out of the scope then the pointer pointing to the variable becomes a **dangling pointer.**

22. Static libraries and shared objects

22.1. Creating a Static Library (archive)

When a C program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. 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.

Steps to create a static library Let us create and use a Static Library in UNIX or UNIX like OS.

1. Create a C file that contains functions in your library.

```
/* Filename: lib_mylib.c */
#include <stdio.h>
void fun(void)
{
printf("fun() called from a static library");
}
```

We have created only one file for simplicity. We can also create multiple files in a library.

2. Create a header file for the library

```
/* Filename: lib_mylib.h */
void fun(void);
```

3. Compile library files.

```
gcc -c lib_mylib.c -o lib_mylib.o
```

4. Create a static library. This step is to bundle multiple object files in one static library. The output of this step is a static library.

```
ar rcs lib_mylib.a lib_mylib.o
```

5. Now our static library is ready to use. At this point we could just copy lib_mylib.a somewhere else to use it. For demo purposes, let us keep the library in the current directory.

Let us create a driver program that uses the above created static library.

1. Create a C file with main function

```
/* filename: driver.c */
#include "lib_mylib.h"
void main()
{
        fun();
}
```

2. Compile the driver program.

```
gcc -c driver.c -o driver.o
```

3. Link the compiled driver program to the static library. Note that -L**.** is used to tell that the static library is in current folder (See this for details of -L and -l options).

```
gcc -o driver driver.o -L. -l_mylib
```

4. Run the driver program

```
./driver
```

```
fun() called from a static library
```

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. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.

3. Since library code is connected at compile time, the final executable has no dependencies on the library at run time i.e. no additional run-time loading costs, it means that you don't need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.

4. In static libraries, once everything is bundled into your application, you don't have to worry that the client will have the right library (and version) available on their system.

5. One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.

6. One major advantage of static libraries being preferred even now "is speed". There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

How to generate a static library (object code archive file):

- Compile: gcc -Wall -c ctest1.c ctest2.c
  Compiler options:
  - -Wall: include warnings. See man page for warnings specified.
- Create library "libctest.a": ar -cvq libctest.a ctest1.o ctest2.o
- List files in library: ar -t libctest.a
- Linking with the library:
  - gcc -o *executable-name* prog.c libctest.a
  - gcc -o *executable-name* prog.c -L/path/to/library-directory -lctest
- Example files:
  - ctest1.c
    ```
    void ctest1(int *i)
    ```

```
{
   *i=5;
}
```

- ctest2.c

```
void ctest2(int *i)
{
   *i=100;
}
```

prog.c

```
#include <stdio.h>

void ctest1(int *);

void ctest2(int *);


int main()

{

   int x;

   ctest1(&x);

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


   return 0;

}
```

Note: After creating the library it was once necessary to run the command: ranlib ctest.a. This created a symbol table within the archive. Ranlib is now embedded into the "ar" command.

22.2. Creating a Dynamic Library (Shared object)

Dynamic Linking doesn't require the code to be copied, it is done by just placing the 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.

How to generate a shared object: (Dynamically linked object library file.) Note that this is a two step process.

1. Create object code
2. Create library
3. Optional: create default version using a symbolic link.

Library creation example:
```
  gcc -Wall -fPIC -c *.c
   gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0   *.o
   mv libctest.so.1.0 /opt/lib
   ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1
   ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so
```

This creates the library libctest.so.1.0 and symbolic links to it.
It is also valid to cascade the linkage:

```
  ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1
   ln -sf /opt/lib/libctest.so.1   /opt/lib/libctest.so
```

If you look at the libraries in /lib/ and /usr/lib/ you will find both methodologies present. Linux developers are not consistent. What is important is that the symbolic links eventually point to an actual library.
Compiler options:

- -Wall: include warnings. See man page for warnings specified.
- -fPIC: Compiler directive to output position independent code, a characteristic required by shared libraries. Also see "-fpic".
- -shared: Produce a shared object which can then be linked with other objects to form an executable.
- -Wl,*options*: Pass options to linker.
  In this example the options to be passed on to the linker are: "-soname libctest.so.1". The name passed with the "-o" option is passed to gcc.
- Option -o: Output of operation. In this case the name of the shared object to be output will be "libctest.so.1.0"

Library Links:

- The link to /opt/lib/libctest.so allows the naming convention for the compile flag -lctest to work.
- The link to /opt/lib/libctest.so.1 allows the run time binding to work. See dependency below

22.3. Dynamically loading a shared object

Compile main program and link with shared object library:

Compiling for run-time linking with a dynamically linked libctest.so.1.0:

```
  gcc -Wall -I/path/to/include-files -L/path/to/libraries prog.c -lctest -o prog
```

Use:

```
  gcc -Wall -L/opt/lib prog.c -lctest -o prog
```

Where the name of the library is libctest.so. (This is why you must create the symbolic links or you will get the error "/usr/bin/ld: cannot find -lctest".)
The libraries will NOT be included in the executable but will be dynamically linked during run-time execution.
List Dependencies:

The shared library dependencies of the executable can be listed with the command: ldd *name-of-executable*

Example: ldd prog

```
     libctest.so.1 => /opt/lib/libctest.so.1 (0x00002aaaaaac000)
      libc.so.6 => /lib64/tls/libc.so.6 (0x0000003aa4e00000)
      /lib64/ld-linux-x86-64.so.2 (0x0000003aa4c00000)
```

Unresolved errors within a shared library may cause an error when the library is loaded.
Example:
Error message at run-time:

```
     ERROR: unable to load  libname-of-lib.so
      ERROR: unable to get function address
```

Investigate error:

```
  [prompt]$ ldd libname-of-lib.so
      libglut.so.3 => /usr/lib64/libglut.so.3 (0x00007fb582b74000)
      libGL.so.1 => /usr/lib64/libGL.so.1 (0x00007fb582857000)
      libX11.so.6 => /usr/lib64/libX11.so.6 (0x00007fb582518000)
      libIL.so.1 (0x00007fa0f2c0f000)
      libcudart.so.4 => not found
```

The first three libraries show that there is a path resolution. The last two are problematic.

The fix is to resolve dependencies of the last two libraries when linking the library lib*name-of-lib*.so:

- Add the unresolved library path in /etc/ld.so.conf.d/*name-of-lib*-x86_64.conf and/or /etc/ld.so.conf.d/*name-of-lib*-i686.conf
  Reload the library cache (/etc/ld.so.cache) with the command: sudo ldconfig
  or
- Add library and path explicitly to the compiler/linker command: -l*name-of-lib* -L/path/to/lib
  or
- Add the library path to the environment variable to fix run-time dependency:
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/lib

Run Program:

- Set path: export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH
- Run: prog

Example with code:

Using the example code above for ctest1.c, ctest2.c and prog.c

1. Compile the library functions: gcc -Wall -fPIC -c ctest1.c ctest2.c
2. Generate the shared library: gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0 ctest1.o ctest2.o
   This generates the library libctest.so.1.0
3. Move to lib/ directory:
   - sudo mv libctest.so.1.0 /opt/lib
   - sudo ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1
   - sudo ln -sf /opt/lib/libctest.so.1 /opt/lib/libctest.so

Compile program for use with a shared library: gcc -Wall -L/opt/lib prog.c -lctest -o prog

If the symbolic links are not created (above), you will get the following error:
/usr/bin/ld: cannot find -lctest
collect2: error: ld returned 1 exit status

4.
   The reference to the library name -lctest refers to /opt/lib/libctest.soConfigure the library path (see below and choose one of three mechanisms).
   In this example we set the environment variable: export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/lib

Run the program: ./prog
Valx=5

You get the following error if the library path is not set:
./prog: error while loading shared libraries: libctest.so.1: cannot open shared object file: No such file or directory

23. Useful C libraries

23.1. Assert

The assert.h header file of the C Standard Library provides a macro called assert which can be used to verify assumptions made by the program and print a diagnostic message if this assumption is false.

The defined macro assert refers to another macro NDEBUG which is not a part of <assert.h>. If NDEBUG is defined as a macro name in the source file, at the point where <assert.h> is included, the assert macro is defined as follows −

```
#define assert(ignore) ((void)0)
```

Following is the only function defined in the header assert.h −

void assert(int expression)

This is actually a macro and not a function, which can be used to add diagnostics in your C program.

Description

The C library macro void assert(int expression) allows diagnostic information to be written to the standard error file. In other words, it can be used to add diagnostics in your C program.

Declaration

Following is the declaration for assert() Macro.

void assert(int expression);

Parameters

- expression − This can be a variable or any C expression. If expression evaluates to TRUE, assert() does nothing. If expression evaluates to FALSE, assert() displays an error message on stderr (standard error stream to display error messages and diagnostics) and aborts program execution.

Return Value

This macro does not return any value.

Example

The following example shows the usage of assert() macro –

```
#include <assert.h>

#include <stdio.h>

int main () {

   int a;

   char str[50];


   printf("Enter an integer value: ");

   scanf("%d", &a);

   assert(a >= 10);

   printf("Integer entered is %d\n", a);


   printf("Enter string: ");

   scanf("%s", str);

   assert(str != NULL);

   printf("String entered is: %s\n", str);


   return(0);

}
```

OUTPUT :

```
Enter an integer value: 11

Integer entered is 11

Enter string: NIDHI PARMAR

String entered is: NIDHI PARMAR
```

```
Enter an integer value: 5
a.out: main.c:9: main: Assertion `a >= 10' failed.
```

23.2. General Utilities (stdlib.h)

The <stdlib.h> header file declares four types and several functions of general use, and defines several macros. The functions perform string conversion, random number generation, searching and sorting, memory management, and similar tasks.

Types

size_t : An unsigned integral type of the result of the sizeof operator.

wchar_t : An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

div_t : A structure type that is the type of the value returned by the div function.

ldiv_t : A structure type that is the type of the value returned by the ldiv function.

Macros

NULL

Expands to an implementation-defined null pointer constant.

EXIT_FAILURE /EXIT_SUCCESS

Expand to integral expressions for use as the argument to the exit function to return unsuccessful or successful termination status, respectively, to the host environment. These macros are useful as return values from the main function as well.

RAND_MAX

Expands to an integral constant expression whose value is the maximum value returned by the rand function.

MB_CUR_MAX

Expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_ TYPE), and whose value is never greater than MB_ LEN_MAX .

String Conversion Functions

double atof(const char *nptr*);

Converts the string pointed to by *nptr* to double representation and returns the converted value. Except for its behavior when an error occurs, this function is equivalent to: strtod(nptr, (char **)NULL)

int atoi(const char *nptr*);

Converts the string pointed to by *nptr* to int representation and returns the converted value. Except for its behavior when an error occurs, this function is equivalent to:
(int)strtol(nptr, (char **)NULL, 10)

long int atol(const char *nptr*);

Converts the string pointed to by *nptr* to long int representation and returns the converted value. Except for its behavior when an error occurs, this function is equivalent to: strtol(nptr, (char **)NULL, 10)

double strtod(const char *nptr*, char **endptr*);

Converts the string pointed to by *nptr* to double representation.

long int strtol(const char *nptr*, char **endptr*, int *base*);

Converts the string pointed to by *nptr* to long int representation.

unsigned long int strtoul(const char *nptr*, char **endptr*, int *base*);

Converts the string pointed to by *nptr* to unsigned long int representation.

Pseudo-Random Sequence Generation Functions

int rand(void);

Returns a sequence of pseudo-random integers in the range 0 to RAND_MAX .

void srand(unsigned int *seed*);

Uses the argument as a seed for a new sequence of pseudo- random integers to be returned by subsequent calls to rand . If srand is then called with the same seed value, the sequence of pseudo-random integers is repeated. If rand is called before any calls to srand are made, the sequence generated is the same as when srand is first called with a seed value of 1. The srand function returns no value.

Memory Management Functions

void *calloc(size_t *nmemb*, size_t *size*);

Allocates an area in memory for an array of *nmemb* items, each with size *size*. The area is initialized to all bits 0. The calloc function returns either a null pointer if unable to allocate, or a pointer to the allocated area.

void free(void **ptr*);

Deallocates the memory area pointed to by *ptr* that was allocated by a previous calloc , malloc , or realloc . If *ptr* is null, no action occurs. No value is returned.

void *malloc(size_t *size*);

Allocates a contiguous area in memory for an object of size *size*. The area is not initialized. This function returns a pointer to the allocated area, or it returns a null pointer if unable to allocate.

void *realloc(void **ptr*, size_t *size*);

Changes the size of the area pointed to by *ptr* to the number of bytes specified by *size*. If *ptr* is null, the behavior of realloc is identical to malloc . The contents of the area are unchanged up to the lesser of the old and new sizes. This function returns either a null pointer if unable to resize, or a pointer to the possibly moved reallocated area.

Communication with the Environment

void abort(void);

Causes abnormal program termination to occur, unless the SIGABRT signal is being caught and the signal handler does not return. The abort function cannot return to its caller.

int atexit(void (**func*)(void));

Registers the function pointed to by *func* to be called without arguments at normal program termination. Up to 32 functions can be registered. The atexit function returns 0 if the registration succeeds; otherwise, it returns nonzero.

void exit(int *status*);

Causes normal program termination to occur. If a program executes more than one call to exit , the behavior is undefined. Upon execution, the following occurs:

1. All functions registered by atexit are called in the reverse order of their registration.
2. All open output streams are flushed, all open streams are closed, and all files created by tmpfile are removed.
3. Control is returned to the host environment. The value of *status* corresponds to an errno value:
    - If the value *status* is 0 or EXIT_ SUCCESS , a *successful termination* status is returned.
    - If the value *status* is EXIT_ FAILURE , an *unsuccessful termination* status is returned.
    - Otherwise, an *unsuccessful termination* status is returned.

char *getenv(const char **name*);

Searches an environment list provided by the host environment.

int *system(const char **string*);

Passes the string pointed to by *string* to the host environment for execution by a command processor. A null pointer can be specified to inquire whether a command processor exists. If the argument is a null pointer, the system function returns nonzero if a command processor is available or 0 if one is not available. If the argument is not a null pointer, the return value is the status returned by the command processor or 0 if a command processor is not available.

Searching and Sorting Utilities

void *bsearch(const void **key*, const void **base*,
 size_t *nmemb*, size_t *size*, int (**compar*)
 (const void *, const void *));
Searches an array of *nmemb* objects for an element that matches the object pointed to by *key*. The first element of the array is pointed to by *base*; the size of each element is specified by *size*.

You must first sort the array in ascending order according to the function pointed to by *comparison*. The bsearch function calls the specified comparison function pointed to by *comparison* with two arguments that point to the objects being compared (the *key* object and an array element). The comparison function returns:

- An integer less than 0, if the first argument is less than the second argument
- An integer greater than 0, if the first argument is greater than the second argument
- An integer equal to 0, if the first argument equals the second argument

The bsearch function returns a pointer to the matching element of the array, or a null pointer if no match is found.

void qsort(void **base*, size_t *nmemb*,
size_t *size*, int (**compar*) (const void *,
const void *));

Sorts an array of *nmemb* objects in place. The first element of the array is pointed to by *base*; the size of each element is specified by *size*.

The contents of the array are sorted in ascending order according to a comparison function pointed to by compare , which is called with two arguments that point to the objects being compared. The comparison function returns:

- An integer less than 0, if the first argument is less than the second argument
- An integer greater than 0, if the first argument is greater than the second argument
- An integer equal to 0, if the first argument equals the second argument

If two compared elements are equal, their order in the sorted array is unspecified.

The qsort function returns no value.

Integer Arithmetic Functions

int abs(int *j*);

Returns the absolute value of an integer *j*.

div_t div(int *number*, int *denom*);

Computes the quotient and remainder of the division of *number* by *denom*. The div function returns a structure of type div_t containing the quotient and remainder:
int quot;   /* quotient */

int rem;   /* remainder */

long int labs(long int *j*);

Returns the absolute value of a long integer *j*.

ldiv_t ldiv(long int *number*, long int *denom*);

Similar to the div function, except that the arguments and the members of the returned structure (which has type ldiv_t) all have type long int .

23.3. Date and Time functions

Time functions in C are used to interact with system time routine and formatted time outputs are displayed. Example programs for the time functions are given below.

Reference link : https://fresh2refresh.com/c-programming/c-time-related-functions/

Function	Description
setdate()	This function used to modify the system date
getdate()	This function is used to get the CPU time
clock()	This function is used to get current system time
time()	This function is used to get current system time as structure
difftime()	This function is used to get the difference between two given times
strftime()	This function is used to modify the actual time format
mktime()	This function interprets tm structure as calendar time
localtime()	This function shares the tm structure that contains date and time informations
gmtime()	This function shares the tm structure that contains date and time informations
ctime()	This function is used to return string that contains date and time informations

asctime()	Tm structure contents are interpreted by this function as calendar time. This time is converted into string.

24. Data Structures

24.1. Abstract Data Types

The Data Type is basically a type of data that can be used in different computer programs. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

The abstract data type is a special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these data types, we can perform different operations. But how those operations are working is totally hidden from the user. The ADT is made of primitive data types, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

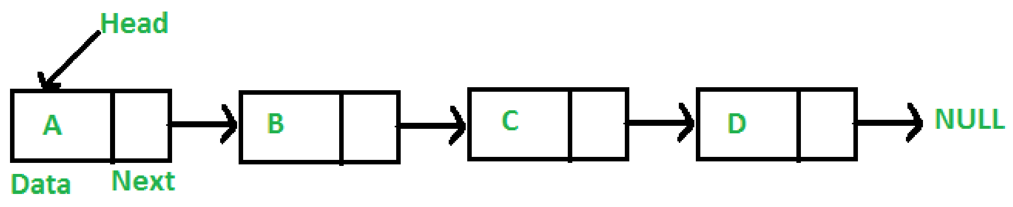
Let us see some operations of those mentioned ADT −

- Stack −
    - isFull(), This is used to check whether stack is full or not
    - isEmpty(), This is used to check whether stack is empty or not
    - push(x), This is used to push x into the stack
    - pop(), This is used to delete one element from top of the stack
    - peek(), This is used to get the top most element of the stack
    - size(), this function is used to get number of elements present into the stack
- Queue −
    - isFull(), This is used to check whether queue is full or not
    - isEmpty(), This is used to check whether queue is empty or not
    - insert(x), This is used to add x into the queue at the rear end
    - delete(), This is used to delete one element from the front end of the queue
    - size(), this function is used to get number of elements present into the queue
- List −
    - size(), this function is used to get number of elements present into the list
    - insert(x), this function is used to insert one element into the list

- remove(x), this function is used to remove given element from the list
- get(i), this function is used to get element at position i
- replace(x, y), this function is used to replace x with y value

24.2. Linked Lists (Overview)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Why Linked List?
Arrays can be used to store linear data of similar types, but arrays have the following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.
For example, in a system, if we maintain a sorted list of IDs in an array id[].
id[] = [1000, 1010, 1050, 2000, 2040].
And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

3) Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

Drawbacks of linked list:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it here.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.
Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node
In C, we can represent a node using structures. Next topic shows an example of a linked list node with integer data.

24.3. Linked Lists (Implementation)

24.3.1. Linked List Creation and Traversal

EXAMPLE :

```
// A simple C program to introduce a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
        int data;
        struct Node* next;
};

// This function prints contents of linked list
// starting from the given node
void printList(struct Node* n)
{
   while (n != NULL) {
      printf(" %d ", n->data);
      n = n->next;
   }
}
```

```
// Program to create a simple linked list with 3 nodes
int main()
{
        struct Node* head = NULL;
        struct Node* second = NULL;
        struct Node* third = NULL;

        // allocate 3 nodes in the heap
        head = (struct Node*)malloc(sizeof(struct Node));
        second = (struct Node*)malloc(sizeof(struct Node));
        third = (struct Node*)malloc(sizeof(struct Node));

        /* Three blocks have been allocated dynamically.
        We have pointers to these three blocks as head,
        second and third

        head            second              third
                |                   |                   |
                |                   |                   |
        +---+-----+      +----+----+     +----+----+
        | # | # |  | # | # |          | # | # |
        +---+-----+      +----+----+     +----+----+

  # represents any random value.
  Data is random because we haven't assigned
  anything yet */

        head->data = 1; // assign data in first node
        head->next = second; // Link first node with the second node

        /* data has been assigned to the data part of the first
        block (block pointed by the head). And next
        pointer of first block points to second.
        So they both are linked.

        head            second              third
                |                   |                   |
                |                   |                   |
        +---+---+        +----+----+     +-----+----+
        | 1 | o----->| # | # |       | # | # |
        +---+---+        +----+----+     +-----+----+
  */
```

```
        second->data = 2;       // assign data to second node
        second->next = third;  // Link second node with the third node

        /* data has been assigned to the data part of the second
        block (block pointed by second). And next
        pointer of the second block points to the third
        block. So all three blocks are linked.

        head              second                  third
                |                        |                       |
                |                        |                       |
        +---+---+         +---+---+        +----+----+
        | 1 | o----->| 2 | o-----> | # | # |
        +---+---+         +---+---+        +----+----+       */

        third->data = 3; // assign data to third node
        third->next = NULL;

        /* data has been assigned to data part of third
        block (block pointed by third). And next pointer
        of the third block is made NULL to indicate
        that the linked list is terminated here.

        We have the linked list ready.

                head
                        |
                        |
                +---+---+         +---+---+        +----+------+
                | 1 | o----->| 2 | o-----> | 3 | NULL |
                +---+---+         +---+---+        +----+------+


        Note that only head is sufficient to represent
        the whole list. We can traverse the complete
        list by following next pointers. */

    printList(head);

        return 0;
}
```

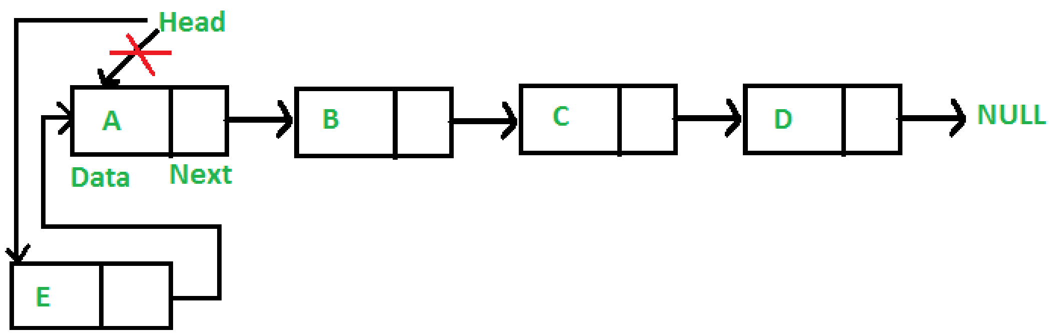
OUTPUT :

```
 1  2  3
```

24.3.2. Inserting a Node in Linked List

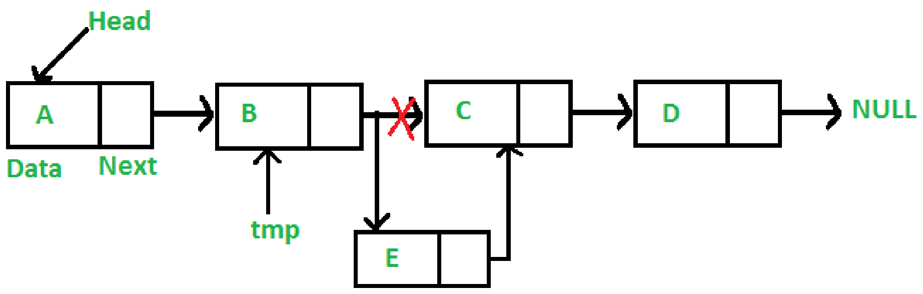
The methods to insert a new node in the linked list in three below mentioned ways :
1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

Add a node at the front: (4 steps process)
The new node is always added before the head of the given Linked List. And the newly added node becomes the new head of the Linked List. For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node.



Add a node after a given node: (5 steps process)

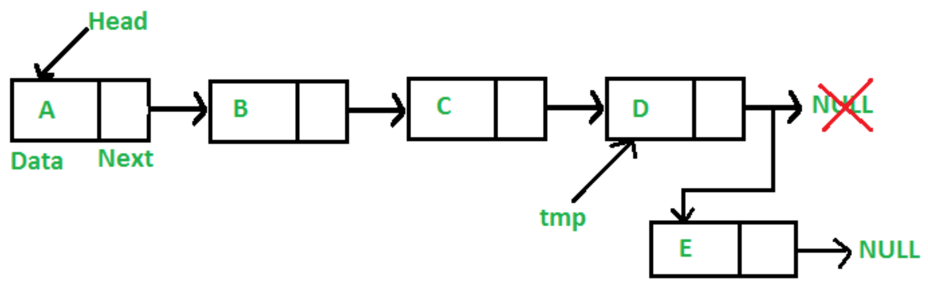
We are given a pointer to a node, and the new node is inserted after the given node.



Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.



Following is a complete program that uses all of the above methods to create a linked list.

EXAMPLE :

```
// A complete working C program to demonstrate all insertion methods
// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
int data;
struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and
an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
        /* 1. allocate node */
        struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

        /* 2. put in the data */
        new_node->data = new_data;
        /* 3. Make next of new node as head */
        new_node->next = (*head_ref);

        /* 4. move the head to point to the new node */
        (*head_ref) = new_node;
}

/* Given a node prev_node, insert a new node after the given
prev_node */
void insertAfter(struct Node* prev_node, int new_data)
{
        /*1. check if the given prev_node is NULL */
        if (prev_node == NULL)
        {
        printf("the given previous node cannot be NULL");
        return;
        }

        /* 2. allocate new node */
        struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

        /* 3. put in the data */
        new_node->data = new_data;

        /* 4. Make next of new node as next of prev_node */
        new_node->next = prev_node->next;

        /* 5. move the next of prev_node as new_node */
        prev_node->next = new_node;
}

/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
        /* 1. allocate node */
        struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

        struct Node *last = *head_ref; /* used in step 5*/

        /* 2. put in the data */
        new_node->data = new_data;
```

```
	/* 3. This new node is going to be the last node, so make next of
		it as NULL*/
	new_node->next = NULL;

	/* 4. If the Linked List is empty, then make the new node as head */
	if (*head_ref == NULL)
	{
	*head_ref = new_node;
	return;
	}

	/* 5. Else traverse till the last node */
	while (last->next != NULL)
		last = last->next;

	/* 6. Change the next of last node */
	last->next = new_node;
	return;
}

// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
while (node != NULL)
{
	printf(" %d ", node->data);
	node = node->next;
}
}

/* Driver program to test above functions*/
int main()
{
/* Start with the empty list */
struct Node* head = NULL;

// Insert 6. So linked list becomes 6->NULL
append(&head, 6);

// Insert 7 at the beginning. So linked list becomes 7->6->NULL
push(&head, 7);

// Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
push(&head, 1);
```

```
// Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
append(&head, 4);

// Insert 8, after 7, as head is at 1, so head->next is 7. So linked list becomes
1->7->8->6->4->NULL
insertAfter(head->next, 8);

printf("\n Created Linked list is: ");
printList(head);

return 0;
}
```

OUTPUT :

```
Created Linked list is:  1  7  8  6  4
```

<u>24.3.3. Deleting a Node in Linked List</u>

Given a 'key', delete the first occurrence of this key in the linked list.

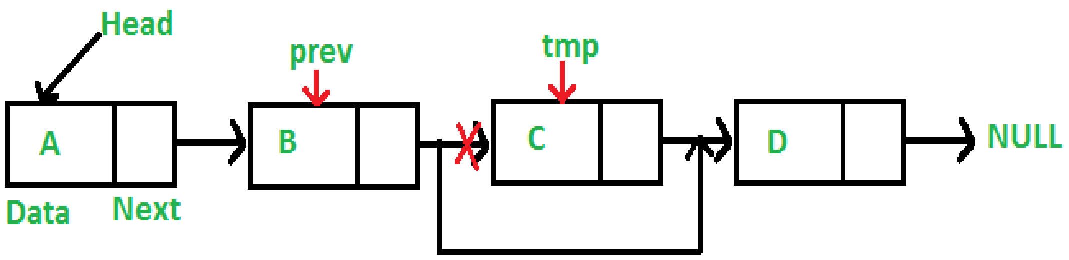
<u>Iterative Method:</u>

To delete a node from the linked list, we need to do the following steps.

1) Find the previous node of the node to be deleted.

2) Change the next of the previous node.

3) Free memory for the node to be deleted.



Since every node of the linked list is dynamically allocated using malloc() in C, we need to call free() to free memory allocated for the node to be deleted.

EXAMPLE :

```
// A complete working C program
// to demonstrate deletion in
// singly linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
	int data;
	struct Node* next;
};

/* Given a reference (pointer to pointer) to the head of a
list and an int, inserts a new node on the front of the
list. */
void push(struct Node** head_ref, int new_data)
{
	struct Node* new_node
		= (struct Node*)malloc(sizeof(struct Node));
	new_node->data = new_data;
	new_node->next = (*head_ref);
	(*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a
list and a key, deletes the first occurrence of key in
linked list */
void deleteNode(struct Node** head_ref, int key)
{
	// Store head node
	struct Node *temp = *head_ref, *prev;

	// If head node itself holds the key to be deleted
	if (temp != NULL && temp->data == key) {
		*head_ref = temp->next; // Changed head
		free(temp); // free old head
		return;
	}

	// Search for the key to be deleted, keep track of the
	// previous node as we need to change 'prev->next'
	while (temp != NULL && temp->data != key) {
		prev = temp;
		temp = temp->next;
	}
	// If key was not present in linked list
	if (temp == NULL)
		return;

	// Unlink the node from linked list
	prev->next = temp->next;

	free(temp); // Free memory
}

// This function prints contents of linked list starting
// from the given node
void printList(struct Node* node)
{
	while (node != NULL) {
		printf(" %d ", node->data);
		node = node->next;
	}
}

// Driver code
int main()
{
	/* Start with the empty list */
	struct Node* head = NULL;

	push(&head, 7);
	push(&head, 1);
	push(&head, 3);
	push(&head, 2);

	puts("Created Linked List: ");
	printList(head);
	deleteNode(&head, 1);
	puts("\nLinked List after Deletion of 1: ");
	printList(head);
	return 0;
}
```

OUTPUT :

```
Created Linked List:
 2  3  1  7
Linked List after Deletion of 1:
 2  3  7
```

24.3.4. Deleting a Node at given position in Linked List

Given a singly linked list and a position, delete a linked list node at the given position.

Example:

Input: position = 1, Linked List = 8->2->3->1->7

Output: Linked List = 8->3->1->7

Input: position = 0, Linked List = 8->2->3->1->7

Output: Linked List = 2->3->1->7

If the node to be deleted is the root, simply delete it. To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if positions are not zero, we run a loop position-1 times and get a pointer to the previous node.

EXAMPLE :

```
// A complete working C program to delete a node in a linked list
// at a given position
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
	int data;
	struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
and an int inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
	struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
	new_node->data = new_data;
	new_node->next = (*head_ref);
```

```
          (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
and a position, deletes the node at the given position */
void deleteNode(struct Node **head_ref, int position)
{
// If linked list is empty
if (*head_ref == NULL)
          return;

// Store head node
struct Node* temp = *head_ref;

          // If head needs to be removed
          if (position == 0)
          {
                    *head_ref = temp->next; // Change head
                    free(temp);                         // free old head
                    return;
          }

          // Find previous node of the node to be deleted
          for (int i=0; temp!=NULL && i<position-1; i++)
                    temp = temp->next;

          // If position is more than number of nodes
          {
                    printf("Position Out of range! List unchanged.\n");
                    return;
          }

          // Node temp->next is the node to be deleted
          // Store pointer to the next of node to be deleted
          struct Node *next = temp->next->next;

          // Unlink the node from linked list
          free(temp->next); // Free memory

          temp->next = next; // Unlink the deleted node from list
}

// This function prints contents of linked list starting from
// the given node
```

```
void printList(struct Node *node)
{
        while (node != NULL)
        {
                printf(" %d ", node->data);
                node = node->next;
        }
}

/* Driver program to test above functions*/
int main()
{
        /* Start with the empty list */
        struct Node* head = NULL;
   int position;

        push(&head, 7);
        push(&head, 1);
        push(&head, 3);
        push(&head, 2);
        push(&head, 8);

        puts("Created Linked List: ");
        printList(head);

        puts("\nEnter position to delete node : ");
        scanf("%d", &position);

        deleteNode(&head, position);
        printf("Linked List after Deletion at position %d: \n", position);
        printList(head);
        return 0;
}
```

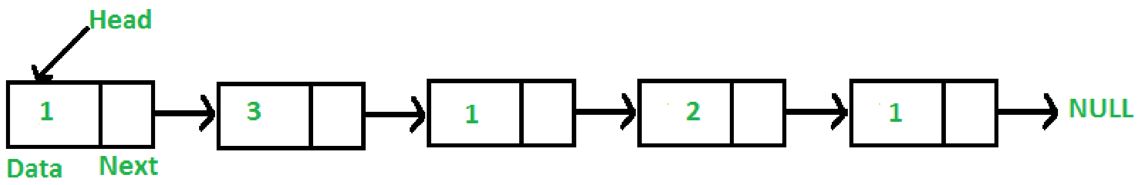
OUTPUT :

```
Created Linked List:
 8  2  3  1  7
Enter position to delete node :
2
Linked List after Deletion at position 2:
 8  2  1  7
--------------------------------------------------------------------

 Created Linked List:
 8  2  3  1  7
```

```
Enter position to delete node :
0
Linked List after Deletion at position 0:
 2  3  1  7
--------------------------------------------------------------------

Created Linked List:
 8  2  3  1  7
Enter position to delete node :
6
Position Out of range! List unchanged.
Linked List after Deletion at position 6:
 8  2  3  1  7
```

24.3.5. Find length of Linked List

For example, the function should return 5 for linked list 1->3->1->2->1.



Output: 5

1) Initialize count as 0

2) Initialize a node pointer, current = head.

3) Do following while current is not NULL

    a) current = current -> next

    b) count++;

4) Return count

EXAMPLE :

```
// Iterative C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
	int data;
	struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct Node** head_ref, int new_data)
{
	/* allocate node */
	struct Node* new_node =
			(struct Node*) malloc(sizeof(struct Node));

	/* put in the data */
	new_node->data = new_data;

	/* link the old list off the new node */
	new_node->next = (*head_ref);

	/* move the head to point to the new node */
	(*head_ref) = new_node;
}

/* Counts no. of nodes in linked list */
int getCount(struct Node* head)
{
	int count = 0; // Initialize count
	struct Node* current = head; // Initialize current
	while (current != NULL)
	{
		count++;
		current = current->next;
	}
	return count;
}

/* Driver program to test count function*/
int main()
{
	/* Start with the empty list */
	struct Node* head = NULL;

	/* Use push() to construct below list
```

```
        1->2->1->3->1 */
        push(&head, 1);
        push(&head, 3);
        push(&head, 1);
        push(&head, 2);
        push(&head, 1);

        /* Check the count function */
        printf("count of nodes is %d", getCount(head));
        return 0;
}
```

OUTPUT :

```
count of nodes is 5
```

24.3.6. Search an Element in Linked List

Write a function that searches a given key 'x' in a given singly linked list. The function should return true if x is present in the linked list and false otherwise.

```
  bool search(Node *head, int x)
```

For example, if the key to be searched is 15 and the linked list is 14->21->11->30->10, then the function should return false. If the key to be searched is 14, then the function should return true.

Iterative Solution

```
1) Initialize a node pointer, current = head.

2) Do following while current is not NULL

     a) current->key is equal to the key being searched return
true.

    b) current = current->next

3) Return false
```

Following is an iterative implementation of the above algorithm to search a given key.

EXAMPLE :

```
// Iterative C program to search an element in linked list
#include<stdio.h>
#include<stdlib.h>
```

```
#include<stdbool.h>

/* Link list node */
struct Node
{
        int key;
        struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct Node** head_ref, int new_key)
{
        /* allocate node */
        struct Node* new_node =
                        (struct Node*) malloc(sizeof(struct Node));

        /* put in the key */
        new_node->key = new_key;

        /* link the old list off the new node */
        new_node->next = (*head_ref);

        /* move the head to point to the new node */
        (*head_ref) = new_node;
}

/* Checks whether the value x is present in linked list */
bool search(struct Node* head, int x)
{
        struct Node* current = head; // Initialize current
        while (current != NULL)
        {
                if (current->key == x)
                        return true;
                current = current->next;
        }
        return false;
}

/* Driver program to test count function*/
int main()
```

```
{
        /* Start with the empty list */
        struct Node* head = NULL;
        int x;

        /* Use push() to construct below list
        14->21->11->30->10 */
        push(&head, 10);
        push(&head, 30);
        push(&head, 11);
        push(&head, 21);
        push(&head, 14);

        printf("Enter element value to search : ");
        scanf("%d", &x);
   printf("Value %d present in list ? " ,x);

        search(head, x)? printf("Yes") : printf("No");
        return 0;
}
```
OUTPUT :

```
Enter element value to search : 21

Value 21 present in list ? Yes

----------------------------------------------------

Enter element value to search : 15

Value 15 present in list ? No
```

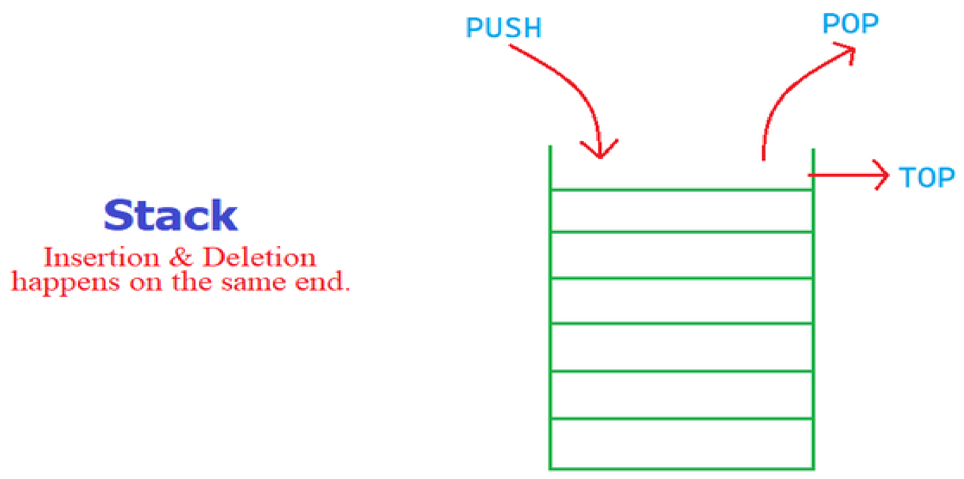
24.4. Stacks (Overview)

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.



How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Practical Application : As soon as the compiler encounters a function call, it gets pushed into the stack. In the case of nested functions, the inner functions get executed before the outer functions. This is totally managed by stacks.

24.5. Stacks (Implementation)

24.5.1. Stacks using Arrays

Initially, we set a pointer Peek/Top to keep the track of the topmost item in the stack. Initialize the stack to -1. Then, we check whether the stack is empty through the comparison of Peek to -1 i.e. **Top == -1**

As we add the elements to the stack, the position of the Peek element keeps updating every time.

As soon as we pop or delete an item from the set of inputs, the top-most element gets deleted and thus the value of Peek/Top gets reduced.

EXAMPLE using ARRAY :

```
#include<stdio.h>

#include<stdlib.h>

#define Size 4

int Top=-1, inp_array[Size];
void Push();
void Pop();
void show();

int main()
{
   int choice;

   while(1)
   {
      printf("\nOperations performed by Stack");
      printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");
      printf("\n\nEnter the choice:");
      scanf("%d",&choice);

      switch(choice)
      {
         case 1: Push();
               break;
         case 2: Pop();
```

```
                break;
            case 3: show();
                break;
            case 4: exit(0);

            default: printf("\nInvalid choice!!");
        }
    }
}

void Push()
{
    int x;

    if(Top==Size-1)
    {
        printf("\nOverflow!!");
    }
    else
    {
        printf("\nEnter element to be inserted to the stack:");
        scanf("%d",&x);
        Top=Top+1;
        inp_array[Top]=x;
    }
}

void Pop()
{
    if(Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element:  %d",inp_array[Top]);
        Top=Top-1;
    }
}

void show()
{
    if(Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
        for(int i=Top;i>=0;--i)
            printf("%d\n",inp_array[i]);
    }
}
```

OUTPUT :

```
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:1

Enter element to be inserted to the stack:11

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3

Elements present in the stack:
11

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:1

Enter element to be inserted to the stack:12
```

```
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3

Elements present in the stack:
12
11

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:1

Enter element to be inserted to the stack:13

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3

Elements present in the stack:
13
12
11

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:2
```

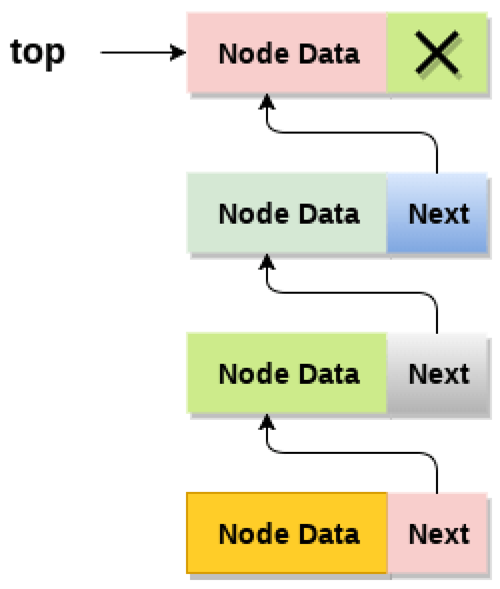
```
Popped element:  13
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3

Elements present in the stack:
12
11

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:2

Popped element:  12
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3

Elements present in the stack:
11

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:2

Popped element:  11
Operations performed by Stack
1.Push the element
```

```
2.Pop the element
3.Show
4.End

Enter the choice:3

Underflow!!
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:2

Underflow!!
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:4
```

24.5.2. Stacks using Linked List

Instead of using an array, we can also use a linked list to implement a stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

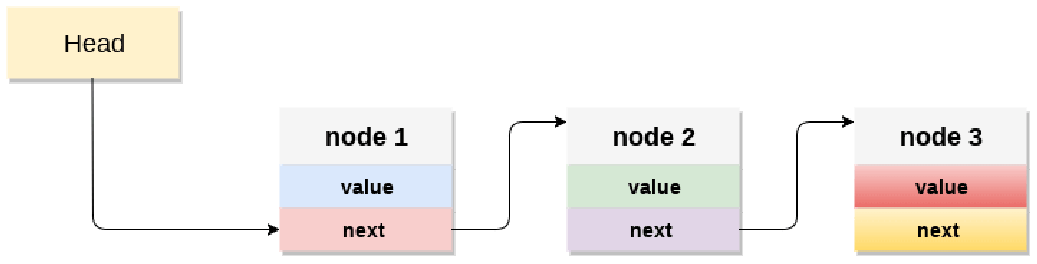
The top most node in the stack always contains null in its address field. Let's discuss the way in which each operation is performed in the linked list implementation of stack.

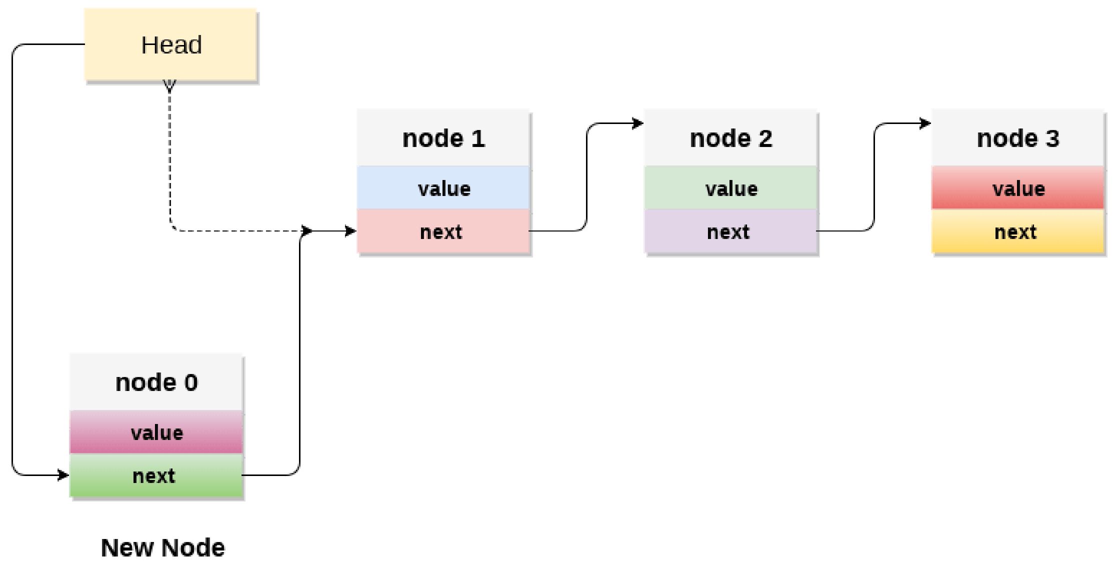
Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assigning null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)





<u>Deleting a node from the stack (POP operation)</u>

Deleting a node from the top of the stack is referred to as a pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

EXAMPLE :

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;

void main ()
{
   int choice=0;
   printf("\n*********Stack operations using linked list*********\n");
   printf("\n----------------------------------------------\n");
   while(choice != 4)
   {
      printf("\nChoose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit\n");
      printf("\nEnter your choice : ");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
```

```
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice : ");
            }
    };
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("Not able to push the element");
    }
    else
    {
        printf("Enter the value : ");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed : %d\n" ,val);

    }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped : %d\n", item);

    }
}
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```

OUTPUT :

```
*********Stack operations using linked list*********

----------------------------------------------

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 1
Enter the value : 11
Item pushed : 11

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 1
Enter the value : 12
Item pushed : 12

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 1
Enter the value : 13
Item pushed : 13
```

```
Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 3
Printing Stack elements
13
12
11

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 2
Item popped : 13

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 3
Printing Stack elements
12
11

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 3
Printing Stack elements
```

```
12
11

Choose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 2
Item popped : 12

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 2
Item popped : 11

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 3
Stack is empty

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 2
Underflow
Choose one from the below options...
```

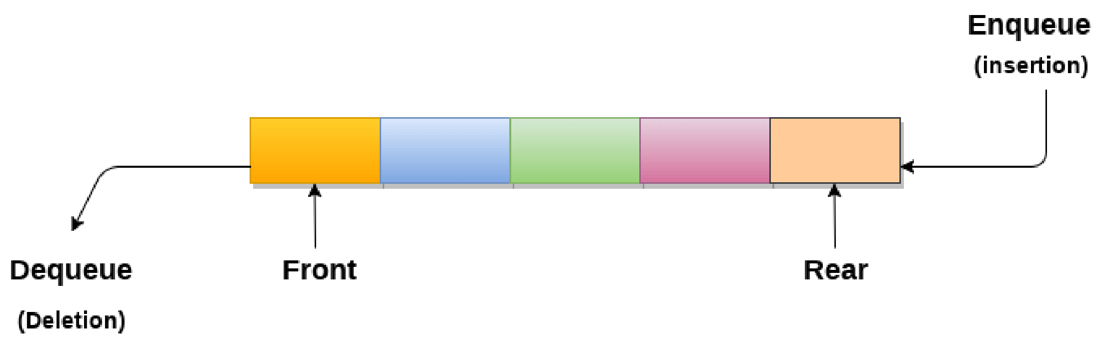
```
1.Push
2.Pop
3.Show
4.Exit

Enter your choice : 4
Exiting....
```

24.6. Queues (Overview)

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on a first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queues are used to maintain the playlist in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

24.7. Queues (Implementation)

One of the alternatives of array implementation is linked list implementation of queue.

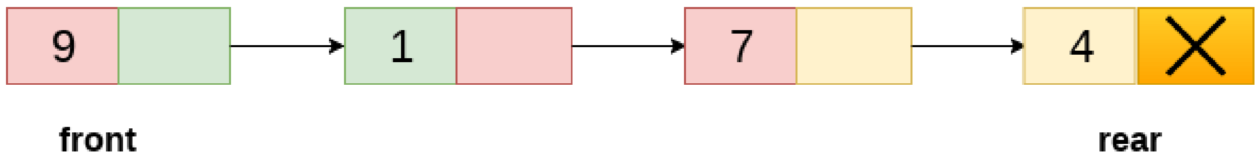
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

1. Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenarios of inserting this new node ptr into the linked queue.

In the first scenario, we insert an element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
      if(front == NULL)
      {
         front = ptr;
         rear = ptr;
         front -> next = NULL;
         rear -> next = NULL;
      }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of the rear point to NULL.

```
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

```
• Step 1: Allocate the space for the new node PTR
• Step 2: SET PTR -> DATA = VAL
• Step 3: IF FRONT = NULL
      SET FRONT = REAR = PTR
      SET FRONT -> NEXT = REAR -> NEXT = NULL
      ELSE
      SET REAR -> NEXT = PTR
      SET REAR = PTR
      SET REAR -> NEXT = NULL
      [END OF IF]
• Step 4: END
```

C Function

```
void insert(struct node *ptr, int item; )
{


   ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
      return;
   }
```

```
    else
    {
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check whether the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;

front = front -> next;

free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

- **Step 1:** IF FRONT = NULL

  Write " Underflow "

Go to Step 5
[END OF IF]

- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

C Function

```
void delete (struct node *ptr)
{
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
   }
}
```

EXAMPLE :

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
    int choice;
    printf("\n************************Main Menu****************************\n");

printf("\n=================================================================
==\n");
    while(choice != 4)
    {
                        printf("\n1.Insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
        printf("\nEnter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice : ");
        }
    }
}
void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value : ");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        printf("\nDeleted from queue is :%d\n",ptr -> data);
        front = front -> next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        while(ptr != NULL)
        {
            printf("\n%d\n",ptr -> data);
            ptr = ptr -> next;
        }
    }
}
```

OUTPUT :

```
*************************Main Menu*****************************

==============================================================

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit
```

```
Enter your choice : 1

Enter value : 11

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 1

Enter value : 12

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 1

Enter value : 13

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 3

printing values .....

11

12

13

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 2
```

```
Deleted from queue is :11

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 3

printing values .....

12

13

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 2

Deleted from queue is :12

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 3

printing values .....

13

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 2

Deleted from queue is :13
```

```
1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 3

Empty queue

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 2

UNDERFLOW

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 3

Empty queue

1.Insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice : 4
```

24.8. Binary Trees (Overview)

https://www.programiz.com/dsa/binary-tree

24.9. Binary Trees (Implementation)

https://www.thegeekstuff.com/2013/02/c-binary-tree/

24.10. Sorting algorithms

1. Binary search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array.

EXAMPLE :

```
// Binary Search in C

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 5;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

OUTPUT :

```
Element is found at index 2
```

2. Linear Search

Linear Search Algorithm

```
LinearSearch(array, key)
  for each item in the array
    if item == value
      return its index
```

EXAMPLE :

```
// Linear Search in C

#include <stdio.h>

int search(int array[], int n, int x) {

  // Going through array sequentially
  for (int i = 0; i < n; i++)
    if (array[i] == x)
      return i;
  return -1;
}

int main() {
  int array[] = {2, 4, 0, 1, 9};
  int x = 1;
  int n = sizeof(array) / sizeof(array[0]);

  int result = search(array, n, x);

  (result == -1) ? printf("Element not found") : printf("Element found at index: %d", result);
}
```

OUTPUT:

```
Element found at index: 3
```

25. Interprocess Communication and signals

25.1. Interprocess Communication

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Let's discuss an example of communication between processes using the shared memory method.

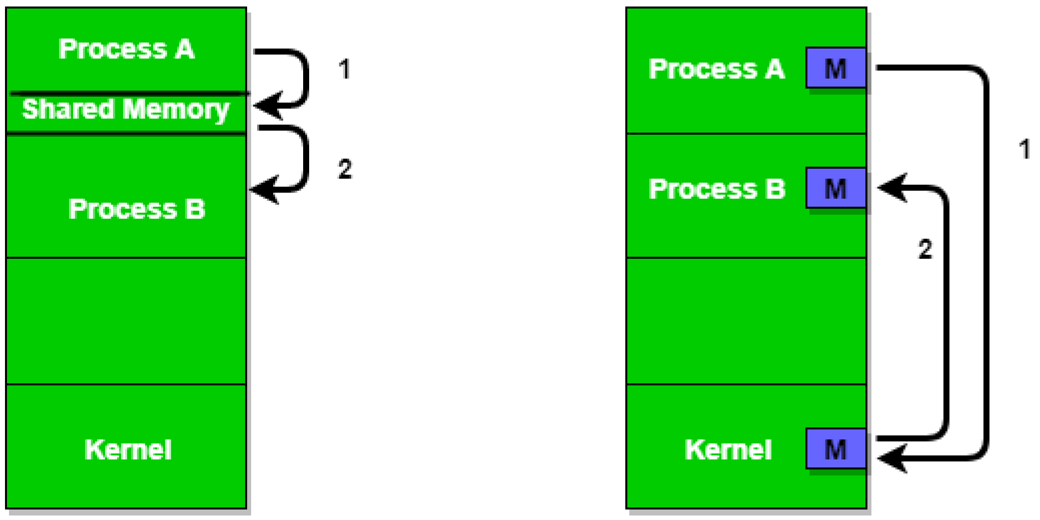


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memories, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, consumers will consume them.

ii) Messaging Passing Method

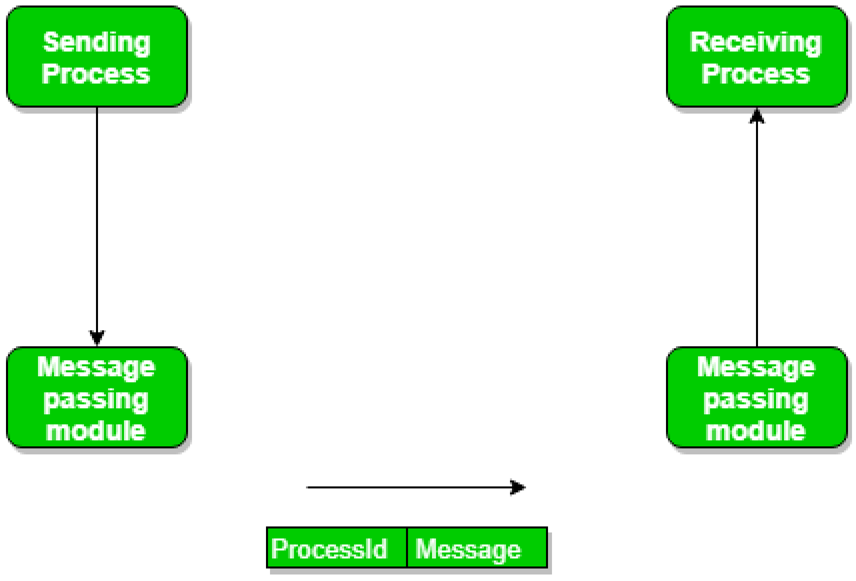
Now, The communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.

  We need at least two primitives:

  – **send**(message, destination) or **send**(message)

  – **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if it runs out of buffer space, sequence number, priority. Generally, messages are sent using FIFO style.

25.2. Signals (Overview)

A signal is a software generated interrupt that is sent to a process by the OS because of when user press ctrl-c or another process tell something to this process.

There are fix set of signals that can be sent to a process. signal are identified by integers.

Signal number have symbolic names. For example **SIGCHLD** is number of the signal sent to the parent process when child terminates.

Examples:

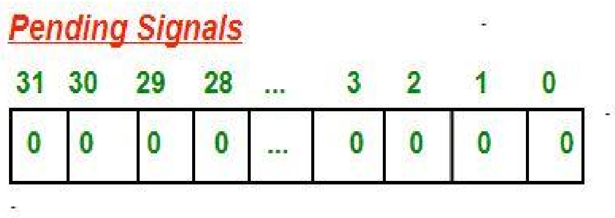
```
#define SIGHUP  1   /* Hangup the process */
#define SIGINT  2   /* Interrupt the process */
#define SIGQUIT 3   /* Quit the process */
#define SIGILL  4   /* Illegal instruction. */
#define SIGTRAP 5   /* Trace trap. */
#define SIGABRT 6   /* Abort. */
```

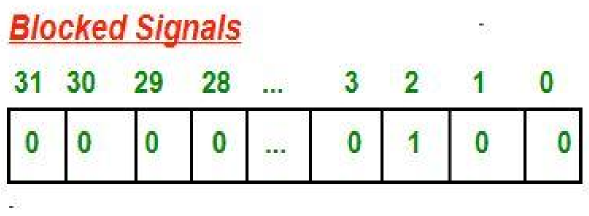
OS Structures for Signals

- For each process, the operating system maintains 2 integers with the bits corresponding to a signal number.
- The two integers keep track of: **pending signals and blocked signals**
- With 32 bit integers, up to 32 different signals can be represented.

Example :

In the example below, the SIGINT (= 2) signal is blocked and no signals are pending.





A signal is sent to a process setting the corresponding bit in the pending signals integer for the process. Each time the OS selects a process to be run on a processor, the pending and blocked integers are checked. If no signals are pending, the process is restarted normally and continues executing at its next instruction.

If 1 or more signals are pending, but each one is blocked, the process is also restarted normally but with the signals still marked as pending. If 1 or more signals are pending and NOT blocked, the OS executes the routines in the process's code to handle the signals.

Default Signal Handlers

There are several default signal handler routines. Each signal is associated with one of these default handler routine. The different default handler routines typically have one of the following actions:

- Ign: Ignore the signal; i.e., do nothing, just return
- Term: terminate the process
- Cont: unblock a stopped process
- Stop: block the process

EXAMPLE :

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>


void sighandler(int);
```

```
int main () {
  signal(SIGINT, sighandler);

  while(1) {
    printf("Going to sleep for a second...\n");
    sleep(1);
  }
  return(0);
}

void sighandler(int signum) {
  printf("Caught signal %d, coming out...\n", signum);
  exit(1);
}
```

OUTPUT:

Print hello world infinite times. If user presses ctrl-c to terminate the process because of **SIGINT** signal sent and its default handler to terminate the process.

```
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
Going to sleep for a second...
^CCaught signal 2, coming out...
```

25.3. Raising a Signal

The C library function int raise(int sig) causes signal sig to be generated. The sig argument is compatible with the SIG macros.

Declaration

Following is the declaration for signal() function.

```
int raise(int sig)
```

Parameters

- sig – This is the signal number to send.

Returns

The raise function returns zero if successful and a nonzero value if unsuccessful.

EXAMPLE :

```
/* Example using raise by TechOnTheNet.com */

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

void signal_handler(int signal)
{
   /* Display a message indicating we have received a signal */
   if (signal == SIGUSR1)
      printf("Received a SIGUSR1 signal\n");

   /* Exit the application */
   exit(0);
}

int main(int argc, const char * argv[])
{
   /* Display a message indicating we are registering the signal handler */
   printf("Registering the signal handler\n");

   /* Register the signal handler */
   signal(SIGUSR1, signal_handler);

   /* Display a message indicating we are raising a signal */
   printf("Raising a SIGUSR1 signal\n");

   /* Raise the SIGUSR1 signal */
   raise(SIGUSR1);

   /* Display a message indicating we are leaving main */
   printf("Finished main\n");
```

```
    return 0;
}
```

OUTPUT :

```
Registering the signal handler
Raising a SIGUSR1 signal
Received a SIGUSR1 signal
```

25.4. Handling a Signal using the signal function

In the C Programming Language, the **signal function** installs a function as the handler for a signal.

Syntax

The syntax for the signal function in the C Language is:

*void (*signal(int sig, void (*func)(int)))(int);*

Parameters or Arguments

sig : The numeric value of the signal.

func : The function to install as the signal handler.

Returns

The signal function returns a pointer to the previous handler for this signal. If the handler can not be installed, the signal function returns SIG_ERR.

Required Header

In the C Language, the required header for the signal function is:

#include <signal.h>

The Above example, signal function is used

25.5. Handling a Signal using sigaction

The sigaction(2) function is a better way to set the signal action. It has the prototype:

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
```

As you can see you don't pass the pointer to the signal handler directly, but instead a struct sigaction object. It's defined as:

```
struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

For a detailed description of this structure's fields see the sigaction(2) manual page. Most important fields are:

- sa_handler - This is the pointer to your handler function that has the same prototype as a handler for signal(2).
- sa_sigaction - This is an alternative way to run the signal handler. It has two additional arguments beside the signal number where the siginfo_t * is the most interesting. It provides more information about the received signal, I will describe it later.
- sa_mask allows you to explicitly set signals that are blocked during the execution of the handler. In addition if you don't use the SA_NODEFER flag the signal which triggered will be also blocked.
- sa_flags allow you to modify the behavior of the signal handling process. For the detailed description of this field, see the manual page. To use the sa_sigaction handler you must use the SA_SIGINFO flag here.

EXAMPLE :

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static int exit_flag = 0;

static void hdl (int sig)
{
        exit_flag = 1;
}

int main (int argc, char *argv[])
{
        struct sigaction act;

        memset (&act, '\0', sizeof(act));
        act.sa_handler = &hdl;
        if (sigaction(SIGTERM, &act, NULL) < 0) {
                perror ("sigaction");
                return 1;
        }

        while (!exit_flag)
                ;

        return 0;
}
```

OUTPUT :

```
^C
```

It depends on compiler optimization settings. Without optimization it executes a loop that ends when the process receives SIGTERM or other sgnal that terminates the process and was not handler. When you compile it with the *-O3* gcc flag it will not exit after receiving SIGTERM. Because the while loop is optimized in such a way that the exit_flag variable is loaded into a processor register once and not read from the memory in the loop. The compiler isn't aware that the loop is not the only place where the program accesses this variable while running the loop. In such cases - modifying a variable in a signal handler that is also accessed in some other parts of the program you must remember to instruct the compiler to always access this variable in memory when reading or writing them. You should use the volatile keyword in the variable declaration:

```
static volatile int exit_flag = 0;
```

After this change everything works as expected.

25.5. The fork() system call

Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains the process ID of the newly created child process.

EXAMPLE :

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

        // make two process which run same program after this instruction

        fork();

        printf("Hello world!\n");

        return 0;

}
```

OUTPUT :

```
Hello world!

Hello world!
```

EXAMPLE :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
        // child process because return value zero
        if (fork() == 0)
                printf("Hello from Child!\n");

        // parent process because return value non-zero.
        else
                printf("Hello from Parent!\n");
}
```

```
int main()
{
        forkexample();
        return 0;
}
```

OUTPUT :

```
1.
Hello from Child!
Hello from Parent!
     (or)
2.
Hello from Parent!
Hello from Child!
```

In the above code, a child process is created. fork() returns 0 in the child process and positive integer in the parent process.

Here, two outputs are possible because the parent process and child process are running concurrently. So we don’t know whether the OS will first give control to the parent process or the child process.

Important: Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different.

26. Threads

26.1. Overview

What is a Thread?

A thread is a single sequence stream within a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads share with other threads their code section, data section and OS resources like open files and signals. But, like a process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

Threads are a popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

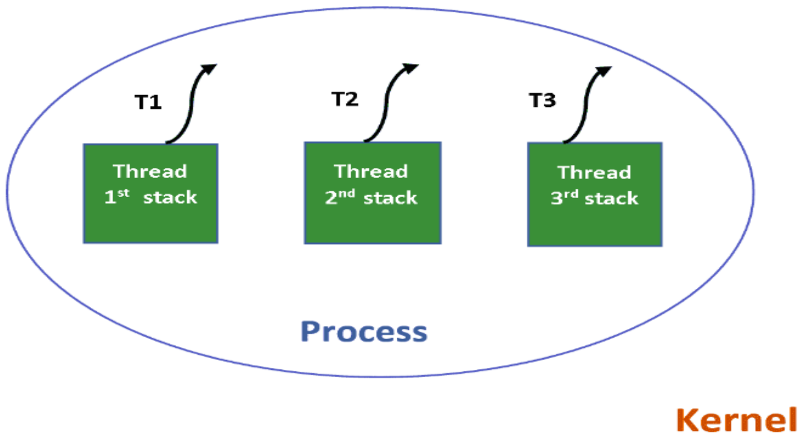
Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.

2) Context switching between threads is much faster.

3) Threads can be terminated easily

4) Communication between threads is faster.



26.2. Creating a thread

POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

A simple C program to demonstrate use of pthread basic functions

Please note that the below program may compile only with C compilers with pthread library.

EXAMPLE :

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

```
#include <unistd.h> //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
	sleep(1);
	printf("Printing from Thread Function \n");
	return NULL;
}

int main()
{
	pthread_t thread_id;
	printf("Before Thread\n");
	pthread_create(&thread_id, NULL, myThreadFun, NULL);
	pthread_join(thread_id, NULL);
	printf("After Thread\n");
	exit(0);
}
```

OUTPUT :

```
Before Thread
Printing from Thread Function
After Thread
```

Explanation of the above code is mentioned in the next topic.

26.3. Passing arguments and returning values

In main() we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.

pthread_create() takes 4 arguments.

- The first argument is a pointer to thread_id which is set by this function.
- The second argument specifies attributes. If the value is NULL, then default attributes shall be used.

- The third argument is the name of the function to be executed for the thread to be created.
- The fourth argument is used to pass arguments to the function, myThreadFun.

The **pthread_join()** function for threads is the equivalent of **wait()** for processes. A call to pthread_join blocks the calling thread until the thread with an identifier equal to the first argument terminates.

How to compile the above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

```
gcc multithread.c -lpthread
```

26.4. Common Thread functions

In a **Unix/Linux operating system**, the **C/C++ languages** provide the POSIX thread(pthread) standard API(Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be implemented on a kernel-level for achieving the speed of execution. Gains can also be found in uni-processor systems by exploiting the latency in IO or other system functions that may halt a process.

We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library. To execute the c file, we have to use the -pthread or -lpthread in the command line while compiling the file.

```
cc -pthread file.c or
cc -lpthread file.c
```

The **functions** defined in the **pthreads library** include:

pthread_create: used to create a new thread
Syntax:
```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```
Parameters:

- **thread**: pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr**: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine**: pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg**: pointer to void that contains the arguments to the function defined in the earlier argument.

pthread_exit: used to terminate a thread

Syntax:

```
void pthread_exit(void *retval);
```

Parameters: This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the terminated thread. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

pthread_join: used to wait for the termination of a thread.

Syntax:

```
int pthread_join(pthread_t th, void **thread_return);
```

Parameter: This method accepts following parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

pthread_self: used to get the thread id of the current thread.

Syntax:

```
pthread_t pthread_self(void);
```

pthread_equal: compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.

Syntax:

```
int pthread_equal(pthread_t t1, pthread_t t2);-
```

Parameters: This method accepts following parameters:

- t1: the thread id of the first thread
- t2: the thread id of the second thread

pthread_cancel: used to send a cancellation request to a thread

Syntax:

```
int pthread_cancel(pthread_t thread);
```

Parameter: This method accepts a mandatory parameter **thread** which is the thread id of the thread to which cancel request is sent.

pthread_detach: used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.

Syntax:

```
int pthread_detach(pthread_t thread);
```

Parameter: This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

Note: If we use exit() instead of **pthread_exit()** to end a thread, the whole process with all associated threads will be terminated even if some of the threads may still be running.

26.5. Thread Synchronization Concepts

POSIX Threads Synchronization

POSIX Threads provide multiple flows of execution within a process. The threads have their own stacks but share the global data and the heap. So the global variables are visible to multiple threads. Also, the threads need to synchronize their actions so that they jointly realize the overall objectives of the process they belong to. The core problems of concurrent programming, mutual exclusion and synchronization are relevant for threads just like these problems are relevant for multi-process systems.

Mutual Exclusion

The global variables are visible to multiple threads. For Pthreads, we have a special locking mechanism for mutual exclusion known as a **mutex** object. If there are calls analogous to P (mutex) and V (mutex) at the start and end of the critical section of code, only one thread would execute the critical section at any time.

Pthread mutex creation

The simplest way to initialize a mutex is to define and initialize it as a global variable.

```
pthread_mutex_t new_mutex = PTHREAD_MUTEX_INITIALIZER;
```

This can only be done for global variables. For automatic and dynamically allocated variables, it is necessary to initialize the mutex with the `pthread_mutex_init` call.

The basic calls for using mutex are the `pthread_mutex_lock` and `pthread_mutex_unlock` calls.

pthread_mutex_lock

```
#include <pthread.h>
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

`pthread_mutex_lock` locks the mutex identified by the pointer passed as the argument. If the mutex is already locked, the call blocks till the time mutex becomes available for locking.

pthread_mutex_unlock

```
#include <pthread.h>
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

`pthread_mutex_unlock` unlocks the mutex identified by the pointer passed as the argument.

Condition Variables

Mutexes are like binary semaphores that help threads in mutually excluding each other from executing critical sections of code concurrently. There is another class of synchronization problem where a counting semaphore value represents the number of instances of resource available and P (semaphore) represents acquiring an instance of resource and V (semaphore) represents releasing an instance. That way processes or threads can work easily with the available number of instances of that resource concurrently. Pthreads provide **condition variables** that help in solving this problem in an easy way. Actually, we are interested in the condition, resource_is_available. If an instance of that resource is available, we can go ahead and take it. If no instance of that resource is available, we wait for a resource to become available.

To solve the problem of synchronized usage of a number of instances of a resource by multiple threads, we need a condition variable, a mutex and a predicate. A predicate could be something like, a resource instance is available. **A condition variable is used for signalling the state of predicate.** The mutex provides the mutual exclusion protection for data between multiple threads.

A thread releasing an instance of a resource has code like this.

```
//Thread 1: Release a resource instance
...
lock (mutex);
release resource instance
update resource control data
cond_signal (cond, mutex);
unlock (mutex);
...
```

A thread acquiring an instance of a resource has code similar to this.

```
// Thread 2: Acquire a resource instance
...
lock (mutex);
while (!resource_instance_is_available)
    cond_wait (cond, mutex);
acquire resource instance
update resource control data
unlock (mutex);
...
```

The code for the first thread is quite straightforward. It first locks the mutex. It releases an instance of resource and unlocks the mutex. The code for the second thread is interesting. It locks the mutex. Then it does a *cond_wait* on condition variable, *cond* using *mutex*. As soon as the conditional wait starts, the *mutex* is released. This ensures that the first thread can lock the *mutex* and do a conditional signal on *cond*. When *cond_wait* returns, the *mutex* is locked and is owned by Thread 2. Thread 2 can acquire the resource, update the resource database and, then, unlock the *mutex*. *cond_wait* is put in a while loop because a signal might interrupt it, causing it to return without a resource instance becoming available.

Pthread condition variable creation

The simplest way to initialize a condition variable is to define and initialize it as a global variable.

```
pthread_cond_t new_cond = PTHREAD_COND_INITIALIZER;
```

This can only be done for global variables. For automatic and dynamically allocated variables, it is necessary to initialize with the `pthread_cond_init` call.

The basic calls for using condition variables are `pthread_cond_wait` and `pthread_cond_signal`.

pthread_cond_wait

```
#include <pthread.h>
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

`pthread_cond_wait` blocks on the condition variable pointed by *cond*. The *mutex* is released at the start of the block. Some other thread can lock the mutex, issue `pthread_cond_signal` and unlock the *mutex*. `pthread_cond_wait`, then, returns with the *mutex* locked and owned by the calling thread.

pthread_cond_signal

```
#include <pthread.h>
int pthread_cond_signal (pthread_cond_t *cond);
```

`pthread_cond_signal` unblocks a thread waiting on the condition variable pointed by *cond*.

Now the task is to synchronize a number of threads using the pthread standard library present with the gcc compiler. The idea is to take thread count and print 1 in the first thread, print 2 in the second thread, print 3 in the third thread till the 10th thread. The output will contain numbers from 1 to 10 based upon the priorities of the threads.

Algorithm

```
Start
Step 1 -> Declare global variables as int MAX=10 and count=1
Step 2 -> declare variable thr of pthread_mutex_t and cond of pthread_cond_t
Step 3 -> Declare Function void *even(void *arg)
  Loop While(count < MAX)
    Call pthread_mutex_lock(&thr)
    Loop While(count % 2 != 0)
      Call pthread_cond_wait(&cond, &thr)
    End
    Print count++
    Call pthread_mutex_unlock(&thr)
    Call pthread_cond_signal(&cond)
  End
  Call pthread_exit(0)
Step 4 -> Declare Function void *odd(void *arg)
  Loop While(count < MAX)
    Call pthread_mutex_lock(&thr)
    Loop While(count % 2 != 1)
      Call pthread_cond_wait(&cond, &thr)
    End
    Print count++
    Call pthread_mutex_unlock(&thr)
    Call pthread_cond_signal(&cond)
  End
  Set pthread_exit(0)
Step 5 -> In main()
  Create pthread_t thread1 and pthread_t thread2
  Call pthread_mutex_init(&thr, 0)
  Call pthread_cond_init(&cond, 0)
  Call pthread_create(&thread1, 0, &even, NULL)
  Call pthread_create(&thread2, 0, &odd, NULL)
  Call pthread_join(thread1, 0)
  Call pthread_join(thread2, 0)
  Call pthread_mutex_destroy(&thr)
  Call pthread_cond_destroy(&cond)
Stop
```

EXAMPLE :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int MAX = 10;
int count = 1;
pthread_mutex_t thr;
pthread_cond_t cond;
void *even(void *arg){
  while(count < MAX) {
    pthread_mutex_lock(&thr);
    while(count % 2 != 0) {
      pthread_cond_wait(&cond, &thr);
    }
```

```
        printf("%d ", count++);
        pthread_mutex_unlock(&thr);
        pthread_cond_signal(&cond);
    }
    pthread_exit(0);
}
void *odd(void *arg){
    while(count < MAX) {
        pthread_mutex_lock(&thr);
        while(count % 2 != 1) {
            pthread_cond_wait(&cond, &thr);
        }
        printf("%d ", count++);
        pthread_mutex_unlock(&thr);
        pthread_cond_signal(&cond);
    }
    pthread_exit(0);
}
int main(){
    pthread_t thread1;
    pthread_t thread2;
    pthread_mutex_init(&thr, 0);
    pthread_cond_init(&cond, 0);
    pthread_create(&thread1, 0, &even, NULL);
    pthread_create(&thread2, 0, &odd, NULL);
    pthread_join(thread1, 0);
    pthread_join(thread2, 0);
    pthread_mutex_destroy(&thr);
    pthread_cond_destroy(&cond);
    return 0;

}
```

OUTPUT :

```
1 2 3 4 5 6 7 8 9 10
```

26.6. Mutexes

The most popular way of achieving thread synchronization is by using Mutexes.

A Mutex is a lock that we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of code. So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code. So this ensures a synchronized access of shared resources in the code.

Internally it works as follows :

- Suppose one thread has locked a region of code using mutex and is executing that piece of code.
- Now if the scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
- Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
- Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
- Mutex lock will only be released by the thread who locked it.
- So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.
- Hence, this system ensures synchronization among the threads while working on shared resources.

A mutex is initialized and then a lock is achieved by calling the following two functions :

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

The first function initializes a mutex and through the second function any critical region in the code can be locked.

The mutex can be unlocked and destroyed by calling following functions :

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The first function above releases the lock and the second function destroys the lock so that it cannot be used anywhere in future

26.7. Condition Variables

When you want to sleep a thread, a condition variable can be used. In C under Linux, there is a function pthread_cond_wait() to wait or sleep.

On the other hand, there is a function pthread_cond_signal() to wake up a sleeping or waiting thread.

Threads can wait on a condition variable.

Syntax of pthread_cond_wait() :

```
int pthread_cond_wait(pthread_cond_t *restrict cond,

               pthread_mutex_t *restrict mutex);
```

Parameter :

cond : condition variable

mutex : is mutex lock

Return Value :

On success, 0 is returned ; otherwise, an error

number shall be returned to indicate the error.

The pthread_cond_wait() releases a lock specified by mutex and waits on the condition cond variable.

Syntax of pthread_cond_signal() :

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Parameter :

cond : condition variable

Return Value :

On success, 0 is returned ; otherwise, an error number

shall be returned to indicate the error.

The pthread_cond_signal() wakes up threads waiting for the condition variable.

Note : The above two functions work together.

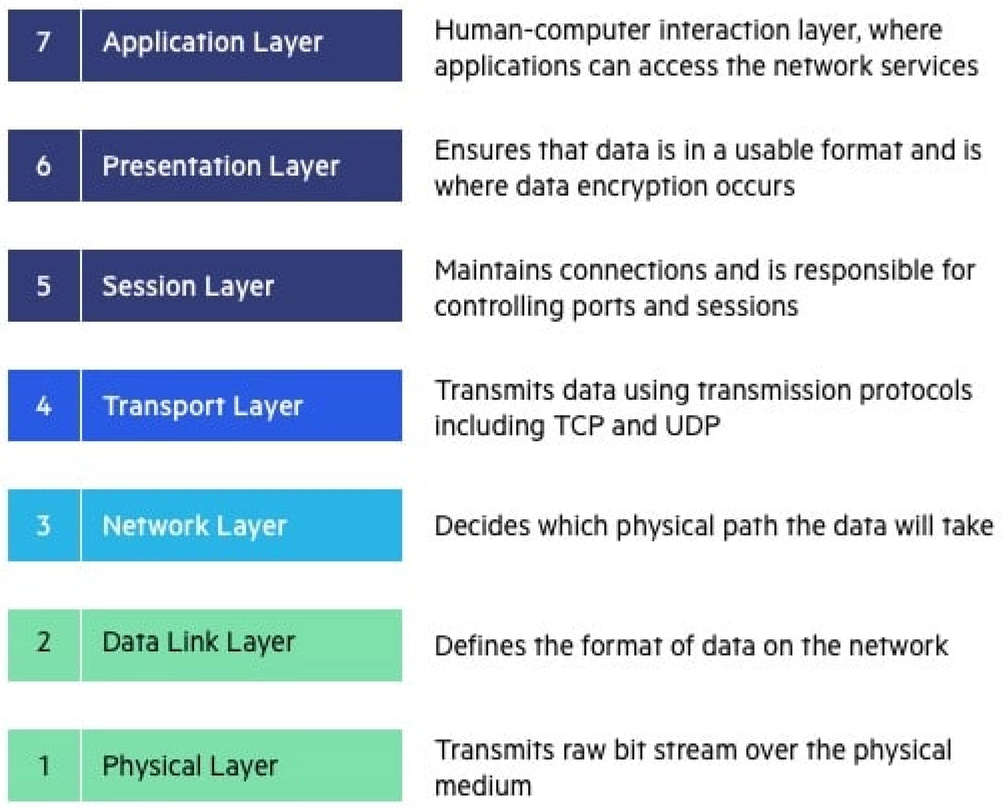
27. Networking (Sockets)

27.1. Overview (OSI layers, TCP/IP)

The Open Systems Interconnection (OSI) model describes seven layers that computer systems use to communicate over a network. It was the first standard model for network communications, adopted by all major computer and telecommunication companies in the early 1980s

The modern Internet is not based on OSI, but on the simpler TCP/IP model. However, the OSI 7-layer model is still widely used, as it helps visualize and communicate how networks operate, and helps isolate and troubleshoot networking problems.

OSI was introduced in 1983 by representatives of the major computer and telecom companies, and was adopted by ISO as an international standard in 1984.



7	Application Layer	Human-computer interaction layer, where applications can access the network services
6	Presentation Layer	Ensures that data is in a usable format and is where data encryption occurs
5	Session Layer	Maintains connections and is responsible for controlling ports and sessions
4	Transport Layer	Transmits data using transmission protocols including TCP and UDP
3	Network Layer	Decides which physical path the data will take
2	Data Link Layer	Defines the format of data on the network
1	Physical Layer	Transmits raw bit stream over the physical medium

We'll describe OSI layers "top down" from the application layer that directly serves the end user, down to the physical layer.

7. Application Layer

The application layer is used by end-user software such as web browsers and email clients. It provides protocols that allow software to send and receive information and present meaningful data to users. A few examples of application layer protocols are the Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Post Office Protocol (POP), Simple Mail Transfer Protocol (SMTP), and Domain Name System (DNS).

6. Presentation Layer
The presentation layer prepares data for the application layer. It defines how two devices should encode, encrypt, and compress data so it is received correctly on the other end. The presentation layer takes any data transmitted by the application layer and prepares it for transmission over the session layer.

5. Session Layer
The session layer creates communication channels, called sessions, between devices. It is responsible for opening sessions, ensuring they remain open and functional while data is being transferred, and closing them when communication ends. The session layer can also set checkpoints during a data transfer—if the session is interrupted, devices can resume data transfer from the last checkpoint.

4. Transport Layer
The transport layer takes data transferred in the session layer and breaks it into "segments" on the transmitting end. It is responsible for reassembling the segments on the receiving end, turning it back into data that can be used by the session layer. The transport layer carries out flow control, sending data at a rate that matches the connection speed of the receiving device, and error control, checking if data was received incorrectly and if not, requesting it again.

3. Network Layer
The network layer has two main functions. One is breaking up segments into network packets, and reassembling the packets on the receiving end. The other is routing packets by discovering the best path across a physical network. The network layer uses network addresses (typically Internet Protocol addresses) to route packets to a destination node.

2. Data Link Layer
The data link layer establishes and terminates a connection between two physically-connected nodes on a network. It breaks up packets into frames and sends

them from source to destination. This layer is composed of two parts—Logical Link Control (LLC), which identifies network protocols, performs error checking and synchronizes frames, and Media Access Control (MAC) which uses MAC addresses to connect devices and define permissions to transmit and receive data.

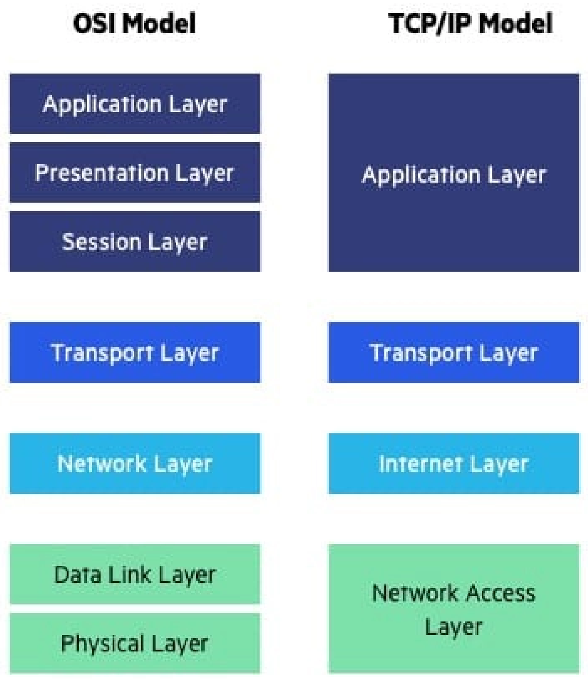
1. Physical Layer

The physical layer is responsible for the physical cable or wireless connection between network nodes. It defines the connector, the electrical cable or wireless technology connecting the devices, and is responsible for transmission of the raw data, which is simply a series of 0s and 1s, while taking care of bit rate control.

The OSI model helps network device manufacturers and networking software vendors:

- Create devices and software that can communicate with products from any other vendor, allowing open interoperability
- Define which parts of the network their products should work with.
- Communicate to users at which network layers their product operates – for example, only at the application layer, or across the stack.

Difference between OSI Model and TCP/IP Model :



Here are some important differences between the OSI and TCP/IP model:

OSI Model	**TCP/IP Model**
It is developed by ISO (International Standard Organization)	It is developed by ARPANET (Advanced Research Project Agency Network).
The OSI model provides a clear distinction between interfaces, services, and protocols.	TCP/IP doesn't have any clear distinguishing points between services, interfaces, and protocols.
OSI refers to Open Systems Interconnection.	TCP refers to the Transmission Control Protocol.
OSI uses the network layer to define routing standards and protocols.	TCP/IP uses only the Internet layer.
OSI follows a vertical approach.	TCP/IP follows a horizontal approach.
OSI layers have seven layers.	TCP/IP has four layers.
In the OSI model, the transport layer is only connection-oriented.	A layer of the TCP/IP model is both connection-oriented and connectionless.
In the OSI model, the data link layer and physical are separate layers.	In TCP, physical and data links are both combined as a single host-to-network layer.
Session and presentation layers are a part of the OSI model.	There is no session and presentation layer in the TCP model.
It is defined after the advent of the Internet.	It was defined before the advent of the internet.
The minimum size of the OSI header is 5 bytes.	The minimum header size is 20 bytes.

Advantages of the OSI Model

Here are the major benefits/pros of using the OSI model:

- It helps you to standardize router, switch, motherboard, and other hardware
- Reduces complexity and standardizes interfaces

- Facilitates modular engineering
- Helps you to ensure interoperable technology
- Helps you to accelerate the evolution
- Protocols can be replaced by new protocols when technology changes.
- Provide support for connection-oriented services as well as connectionless service.
- It is a standard model in computer networking.
- Supports connectionless and connection-oriented services.
- It offers flexibility to adapt to various types of protocols.

Advantages of TCP/IP

Here, are pros/benefits of using the TCP/IP model:

- It helps you to establish/set up a connection between different types of computers.
- It operates independently of the operating system.
- It supports many routing-protocols.
- It enables the internetworking between the organizations.
- TCP/IP model has a highly scalable client-server architecture.
- It can be operated independently.
- Supports several routing protocols.
- It can be used to establish a connection between two computers.

Disadvantages of OSI Model

Here are some cons/ drawbacks of using OSI Model:

- Fitting of protocols is a tedious task.
- You can only use it as a reference model.
- It doesn't define any specific protocol.
- In the OSI network layer model, some services are duplicated in many layers such as the transport and data link layers
- Layers can't work in parallel as each layer needs to wait to obtain data from the previous layer.

Disadvantages of TCP/IP

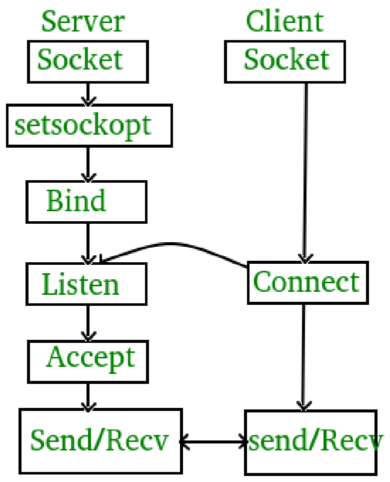
Here, are few drawbacks of using the TCP/IP model:

- TCP/IP is a complicated model to set up and manage.
- The shallow/overhead of TCP/IP is higher-than IPX (Internetwork Packet Exchange).
- In this model the transport layer does not guarantee delivery of packets.
- Replacing protocol in TCP/IP is not easy.
- It has no clear separation from its services, interfaces, and protocols.

27.2. The Socket API

If we are creating a connection between client and server using TCP then it has few functionality like, TCP is suited for applications that require high reliability, and transmission time is relatively less critical. It is used by other protocols like HTTP, HTTPs, FTP, SMTP, Telnet. TCP rearranges data packets in the order specified. There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent. TCP does Flow Control and requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control. It also does error checking and error recovery. Erroneous packets are retransmitted from the source to the destination.

The entire process can be broken down into following steps:



The entire process can be broken down into following steps:

TCP Server –

1. using create(), Create TCP socket.
2. using bind(), Bind the socket to the server address.
3. using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. using accept(), At this point, connection is established between client and server, and they are ready to transfer data.
5. Go back to Step 3.

TCP Client –

1. Create a TCP socket.
2. connect the newly created client socket to the server.

<u>**27.3. Creating a Server Socket**</u>

Stages for server

<u>**Socket creation:**</u>

______int sockfd = socket(domain, type, protocol)

sockfd: socket descriptor, an integer (like a file-handle)

domain: integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol)

type: communication type

    SOCK_STREAM: TCP(reliable, connection oriented)

    SOCK_DGRAM: UDP(unreliable, connectionless)

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

<u>**Setsockopt**</u>**:**

```
int setsockopt(int sockfd, int level, int optname,
    const void *optval, socklen_t optlen);
```

This helps in manipulating options for the socket referred by the file descriptor sockfd.

This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

<u>Bind</u>:

```
int bind(int sockfd, const struct sockaddr *addr,
          socklen_t addrlen);
```

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

<u>Listen</u>:

```
int listen(int sockfd, int backlog);
```

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

<u>Accept</u>:

```
int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

EXAMPLE :

```
// Server side C/C++ program to demonstrate Socket programming
```

```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
	int server_fd, new_socket, valread;
	struct sockaddr_in address;
	int opt = 1;
	int addrlen = sizeof(address);
	char buffer[1024] = {0};
	char *hello = "Hello from server";

	// Creating socket file descriptor
	if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
	{
		perror("socket failed");
		exit(EXIT_FAILURE);
	}

	// Forcefully attaching socket to the port 8080
	if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
		&opt, sizeof(opt)))
	{
		perror("setsockopt");
```

```
        exit(EXIT_FAILURE);
}
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons( PORT );

// Forcefully attaching socket to the port 8080
if (bind(server_fd, (struct sockaddr *)&address,
                                                    sizeof(address))<0)
{
        perror("bind failed");
        exit(EXIT_FAILURE);
}
if (listen(server_fd, 3) < 0)
{
        perror("listen");
        exit(EXIT_FAILURE);
}
if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                (socklen_t*)&addrlen))<0)
{
        perror("accept");
        exit(EXIT_FAILURE);
}
valread = read( new_socket , buffer, 1024);
printf("%s\n",buffer );
send(new_socket , hello , strlen(hello) , 0 );
printf("Hello message sent\n");
```

```
	return 0;
}
```

27.4. Creating a Client Socket

Stages for Client

Socket connection: Exactly same as that of server's socket creation

Connect:

```
int connect(int sockfd, const struct sockaddr *addr,
                                  socklen_t addrlen);
```

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

EXAMPLE :

```
// Client side C/C++ program to demonstrate Socket programming
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
	int sock = 0, valread;
	struct sockaddr_in serv_addr;
	char *hello = "Hello from client";
	char buffer[1024] = {0};
	if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
	{
		printf("\n Socket creation error \n");
		return -1;
	}

	serv_addr.sin_family = AF_INET;
	serv_addr.sin_port = htons(PORT);
```

```
        // Convert IPv4 and IPv6 addresses from text to binary form
        if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
        {
                printf("\nInvalid address/ Address not supported \n");
                return -1;
        }

        if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
        {
                printf("\nConnection Failed \n");
                return -1;
        }
        send(sock , hello , strlen(hello) , 0 );
        printf("Hello message sent\n");
        valread = read( sock , buffer, 1024);
        printf("%s\n",buffer );
        return 0;
}
```

OUTPUT :

```
Client:Hello message sent
Hello from server
Server:Hello from client
Hello message sent
```