Programming in C

Sai Krishna

Contents

1 Basic Terminology

- **Computer program** is a set of instructions that the computer can perform in order to perform some task
- **Programming** is the task of creating computer programs
- **Source code** is a list of commands typed into one or more text files.
- **Execution** of a program means loading the program into memory and the hardware sequentially executing each instruction.
- **Machine code** is a set of instructions that a CPU can understand directly
- **Machine Language** - Each instruction is represented by a binary code
- **Assembly Language** - Each instruction is represented by an abbreviation
  - **Assembler** converts Assembly level language to machine language
  - Assembly language programs are not portable
  - Many instructions are required to even execute simple programs
- **High level language** - Programs that are readable and portable
  - **Compiler** reads the entire source code and produces a stand-alone executable program that can be run. Once the executable file is generated the compiler is no longer required to execute the program.
  - **Interpreter** is a program that directly executes the instruction in the source code without requiring them to be compiled into an executable first.
  - Different compilers can be used for the same program to make it compatible to different hardware (like 32 bits, x86 executable etc)

COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	PHP, Perl, Python, Ruby uses an interpreter.

1.1 Execution of a program

1.1.1 Pre-processing

The following tasks are performed

- Expansion of included headers
- Removal of comments from the osurce code
- Expansion of macros
- The output of this phase is a file with the extension $program_name.i$ that contains the preprocessed code.

1.1.2 Compiling a Source code

The compiler sequentially goes through $program_name.i$ file generated in the previous phase and converts it to assembly level instructions and saves those

instructions to $program_name.s$ file

- Checks the code to make sure it follows the rules of the programming language. If it does not, the compiler will give an error and further compilation is aborted.
- Translates the source code into a assembly language file. If the program has n source code files then n object files are generated.

1.1.3 Assembly

This phase of compilation reads the contents of assembly instructions from $program_name.s$ file and generates machine level instructions that are saved to another file called $program_name.o$. This is the object file.

- This program converts only program source excluding all the external functions used in the program.
  Ex: printf() function from *stdio* library is not converted to machine level instructions.

1.1.4 Linking object files and libraries

Once the object files are ready, then another program called **linker** kicks in. It does the following job

- Combine all the object files into a single executable program.
- Links the library files to the executable program.
  Library file is a collection of pre compiled code that has been packaged up for reuse in other programs.
- Makes sure that all cross-file dependencies are resolved properly.
  If the linker is unable to connect a reference to something with its definition, then a linker error will abort the linking process.
- The linker also adds some extra code to our program which is required when the program starts and ends.
  GCC by default does dynamic linking, so the printf function used in the program is dynamically linked to the final executable binary that gets generated.

1.1.5 Testing and Debugging

Once the executable file is generated, it can then be executed to test if the output of the program meets the expectation. If it does not then the program must be debugged for possible errors.

1.1.6 Integrated development environments (IDEs)

Separate programs can be used for compiling, linking, debugging or a software package known as an *integrated development environment* (IDE) can be used, which bundles and integrates all the features together.

1.2 Example

The source code file is as follows

1.2.1 Source Code - basicprogram.c

```
/* To see the outputs at all intermediate stages of compilation */
#include<stdio.h>
int main()
{
printf("Hello World");
}
```

The above file was saved as *basicProgram.c*, the contents of *basicProgram.i* file is

1.2.2 Pre-processed code - basicprogram.i

```
# 1 "basicProgram.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "basicProgram.c"

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
```

```
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4




# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 34 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
# 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4


typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;


typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;
```

```
typedef long int __quad_t;
typedef unsigned long int __u_quad_t;
```

```
typedef long int __intmax_t;
typedef unsigned long int __uintmax_t;
# 130 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/typesizes.h" 1 3 4
# 131 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4


typedef unsigned long int __dev_t;
typedef unsigned int __uid_t;
typedef unsigned int __gid_t;
typedef unsigned long int __ino_t;
typedef unsigned long int __ino64_t;
typedef unsigned int __mode_t;
typedef unsigned long int __nlink_t;
typedef long int __off_t;
typedef long int __off64_t;
typedef int __pid_t;
typedef struct { int __val[2]; } __fsid_t;
typedef long int __clock_t;
typedef unsigned long int __rlim_t;
typedef unsigned long int __rlim64_t;
typedef unsigned int __id_t;
typedef long int __time_t;
typedef unsigned int __useconds_t;
typedef long int __suseconds_t;

typedef int __daddr_t;
typedef int __key_t;


typedef int __clockid_t;


typedef void * __timer_t;
```

```
typedef long int __blksize_t;



typedef long int __blkcnt_t;
typedef long int __blkcnt64_t;


typedef unsigned long int __fsblkcnt_t;
typedef unsigned long int __fsblkcnt64_t;


typedef unsigned long int __fsfilcnt_t;
typedef unsigned long int __fsfilcnt64_t;


typedef long int __fsword_t;

typedef long int __ssize_t;


typedef long int __syscall_slong_t;

typedef unsigned long int __syscall_ulong_t;



typedef __off64_t __loff_t;
typedef char *__caddr_t;


typedef long int __intptr_t;


typedef unsigned int __socklen_t;




typedef int __sig_atomic_t;
# 36 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/types/__FILE.h" 1 3 4
```

```
struct _IO_FILE;
typedef struct _IO_FILE __FILE;
# 37 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/types/FILE.h" 1 3 4



struct _IO_FILE;


typedef struct _IO_FILE FILE;
# 38 "/usr/include/stdio.h" 2 3 4



# 1 "/usr/include/x86_64-linux-gnu/bits/libio.h" 1 3 4
# 35 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/_G_config.h" 1 3 4
# 19 "/usr/include/x86_64-linux-gnu/bits/_G_config.h" 3 4
# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 20 "/usr/include/x86_64-linux-gnu/bits/_G_config.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h" 1 3 4
# 13 "/usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h" 3 4
typedef struct
{
  int __count;
  union
  {
    unsigned int __wch;
    char __wchb[4];
  } __value;
} __mbstate_t;
# 22 "/usr/include/x86_64-linux-gnu/bits/_G_config.h" 2 3 4



typedef struct
{
  __off_t __pos;
  __mbstate_t __state;
} _G_fpos_t;
typedef struct
{
  __off64_t __pos;
```

```
  __mbstate_t __state;
} _G_fpos64_t;
# 36 "/usr/include/x86_64-linux-gnu/bits/libio.h" 2 3 4
# 53 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stdarg.h" 1 3 4
# 40 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stdarg.h" 3 4
typedef __builtin_va_list __gnuc_va_list;
# 54 "/usr/include/x86_64-linux-gnu/bits/libio.h" 2 3 4
# 149 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
struct _IO_jump_t; struct _IO_FILE;




typedef void _IO_lock_t;




struct _IO_marker {
  struct _IO_marker *_next;
  struct _IO_FILE *_sbuf;



  int _pos;
# 177 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
};


enum __codecvt_result
{
  __codecvt_ok,
  __codecvt_partial,
  __codecvt_error,
  __codecvt_noconv
};
# 245 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
struct _IO_FILE {
  int _flags;



  char* _IO_read_ptr;
```

```
  char* _IO_read_end;
  char* _IO_read_base;
  char* _IO_write_base;
  char* _IO_write_ptr;
  char* _IO_write_end;
  char* _IO_buf_base;
  char* _IO_buf_end;

  char *_IO_save_base;
  char *_IO_backup_base;
  char *_IO_save_end;

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;



  int _flags2;

  __off_t _old_offset;



  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];



  _IO_lock_t *_lock;
# 293 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
  __off64_t _offset;







  void *__pad1;
  void *__pad2;
  void *__pad3;
  void *__pad4;
```

```
  size_t __pad5;
  int _mode;

  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];

};


typedef struct _IO_FILE _IO_FILE;


struct _IO_FILE_plus;

extern struct _IO_FILE_plus _IO_2_1_stdin_;
extern struct _IO_FILE_plus _IO_2_1_stdout_;
extern struct _IO_FILE_plus _IO_2_1_stderr_;
# 337 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
typedef __ssize_t __io_read_fn (void *__cookie, char *__buf, size_t __nbytes);







typedef __ssize_t __io_write_fn (void *__cookie, const char *__buf,
     size_t __n);







typedef int __io_seek_fn (void *__cookie, __off64_t *__pos, int __w);


typedef int __io_close_fn (void *__cookie);
# 389 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
extern int __underflow (_IO_FILE *);
extern int __uflow (_IO_FILE *);
extern int __overflow (_IO_FILE *, int);
# 433 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
extern int _IO_getc (_IO_FILE *__fp);
extern int _IO_putc (int __c, _IO_FILE *__fp);
```

```
extern int _IO_feof (_IO_FILE *__fp) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_ferror (_IO_FILE *__fp) __attribute__ ((__nothrow__ , __leaf__));

extern int _IO_peekc_locked (_IO_FILE *__fp);




extern void _IO_flockfile (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
extern void _IO_funlockfile (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_ftrylockfile (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
# 462 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
extern int _IO_vfscanf (_IO_FILE * __restrict, const char * __restrict,
   __gnuc_va_list, int *__restrict);
extern int _IO_vfprintf (_IO_FILE *__restrict, const char *__restrict,
    __gnuc_va_list);
extern __ssize_t _IO_padn (_IO_FILE *, int, __ssize_t);
extern size_t _IO_sgetn (_IO_FILE *, void *, size_t);

extern __off64_t _IO_seekoff (_IO_FILE *, __off64_t, int, int);
extern __off64_t _IO_seekpos (_IO_FILE *, __off64_t, int);

extern void _IO_free_backup_area (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
# 42 "/usr/include/stdio.h" 2 3 4




typedef __gnuc_va_list va_list;
# 57 "/usr/include/stdio.h" 3 4
typedef __off_t off_t;
# 71 "/usr/include/stdio.h" 3 4
typedef __ssize_t ssize_t;




typedef _G_fpos_t fpos_t;
# 131 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdio_lim.h" 1 3 4
# 132 "/usr/include/stdio.h" 2 3 4
```

```
extern struct _IO_FILE *stdin;
extern struct _IO_FILE *stdout;
extern struct _IO_FILE *stderr;




extern int remove (const char *__filename) __attribute__ ((__nothrow__ , __leaf__));

extern int rename (const char *__old, const char *__new) __attribute__
((__nothrow__ , __leaf__));



extern int renameat (int __oldfd, const char *__old, int __newfd,
       const char *__new) __attribute__ ((__nothrow__ , __leaf__));







extern FILE *tmpfile (void) ;
# 173 "/usr/include/stdio.h" 3 4
extern char *tmpnam (char *__s) __attribute__ ((__nothrow__ , __leaf__)) ;




extern char *tmpnam_r (char *__s) __attribute__ ((__nothrow__ , __leaf__)) ;
# 190 "/usr/include/stdio.h" 3 4
extern char *tempnam (const char *__dir, const char *__pfx)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__malloc__)) ;







extern int fclose (FILE *__stream);
```

```
extern int fflush (FILE *__stream);
# 213 "/usr/include/stdio.h" 3 4
extern int fflush_unlocked (FILE *__stream);
# 232 "/usr/include/stdio.h" 3 4
extern FILE *fopen (const char *__restrict __filename,
      const char *__restrict __modes) ;




extern FILE *freopen (const char *__restrict __filename,
        const char *__restrict __modes,
        FILE *__restrict __stream) ;
# 265 "/usr/include/stdio.h" 3 4
extern FILE *fdopen (int __fd, const char *__modes) __attribute__((__nothrow__ , __leaf__))
# 278 "/usr/include/stdio.h" 3 4
extern FILE *fmemopen (void *__s, size_t __len, const char *__modes)
  __attribute__ ((__nothrow__ , __leaf__)) ;



extern FILE *open_memstream (char **__bufloc, size_t *__sizeloc) __attribute__
((__nothrow__ , __leaf__)) ;




extern void setbuf (FILE *__restrict __stream, char *__restrict __buf) __attribute__
((__nothrow__ , __leaf__));



extern int setvbuf (FILE *__restrict __stream, char *__restrict __buf,
      int __modes, size_t __n) __attribute__ ((__nothrow__ , __leaf__));



extern void setbuffer (FILE *__restrict __stream, char *__restrict __buf,
        size_t __size) __attribute__ ((__nothrow__ , __leaf__));
```

```
extern void setlinebuf (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));




extern int fprintf (FILE *__restrict __stream,
      const char *__restrict __format, ...);




extern int printf (const char *__restrict __format, ...);

extern int sprintf (char *__restrict __s,
      const char *__restrict __format, ...) __attribute__ ((__nothrow__));




extern int vfprintf (FILE *__restrict __s, const char *__restrict __format,
       __gnuc_va_list __arg);




extern int vprintf (const char *__restrict __format, __gnuc_va_list __arg);

extern int vsprintf (char *__restrict __s, const char *__restrict __format,
       __gnuc_va_list __arg) __attribute__ ((__nothrow__));



extern int snprintf (char *__restrict __s, size_t __maxlen,
       const char *__restrict __format, ...)
     __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3, 4)));

extern int vsnprintf (char *__restrict __s, size_t __maxlen,
        const char *__restrict __format, __gnuc_va_list __arg)
     __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3, 0)));
# 365 "/usr/include/stdio.h" 3 4
extern int vdprintf (int __fd, const char *__restrict __fmt,
```

```
       __gnuc_va_list __arg)
     __attribute__ ((__format__ (__printf__, 2, 0)));
extern int dprintf (int __fd, const char *__restrict __fmt, ...)
     __attribute__ ((__format__ (__printf__, 2, 3)));







extern int fscanf (FILE *__restrict __stream,
     const char *__restrict __format, ...) ;




extern int scanf (const char *__restrict __format, ...) ;

extern int sscanf (const char *__restrict __s,
     const char *__restrict __format, ...) __attribute__ ((__nothrow__ , __leaf__));
# 395 "/usr/include/stdio.h" 3 4
extern int fscanf (FILE *__restrict __stream, const char *__restrict __format, ...)
__asm__ ("" "__isoc99_fscanf")

                               ;
extern int scanf (const char *__restrict __format, ...) __asm__ ("" "__isoc99_scanf")
                              ;
extern int sscanf (const char *__restrict __s, const char *__restrict __format, ...)
__asm__ ("" "__isoc99_sscanf") __attribute__ ((__nothrow__ , __leaf__))

                      ;
# 420 "/usr/include/stdio.h" 3 4
extern int vfscanf (FILE *__restrict __s, const char *__restrict __format,
      __gnuc_va_list __arg)
     __attribute__ ((__format__ (__scanf__, 2, 0))) ;




extern int vscanf (const char *__restrict __format, __gnuc_va_list __arg)
     __attribute__ ((__format__ (__scanf__, 1, 0))) ;


extern int vsscanf (const char *__restrict __s,
```

```
      const char *__restrict __format, __gnuc_va_list __arg)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__
     ((__format__ (__scanf__, 2, 0)));
# 443 "/usr/include/stdio.h" 3 4
extern int vfscanf (FILE *__restrict __s, const char *__restrict __format,
__gnuc_va_list __arg) __asm__ ("" "__isoc99_vfscanf")



     __attribute__ ((__format__ (__scanf__, 2, 0))) ;
extern int vscanf (const char *__restrict __format, __gnuc_va_list __arg)
__asm__ ("" "__isoc99_vscanf")

     __attribute__ ((__format__ (__scanf__, 1, 0))) ;
extern int vsscanf (const char *__restrict __s, const char *__restrict __format,
__gnuc_va_list __arg) __asm__ ("" "__isoc99_vsscanf")
__attribute__ ((__nothrow__ , __leaf__))



     __attribute__ ((__format__ (__scanf__, 2, 0)));
# 477 "/usr/include/stdio.h" 3 4
extern int fgetc (FILE *__stream);
extern int getc (FILE *__stream);




extern int getchar (void);
# 495 "/usr/include/stdio.h" 3 4
extern int getc_unlocked (FILE *__stream);
extern int getchar_unlocked (void);
# 506 "/usr/include/stdio.h" 3 4
extern int fgetc_unlocked (FILE *__stream);
# 517 "/usr/include/stdio.h" 3 4
extern int fputc (int __c, FILE *__stream);
extern int putc (int __c, FILE *__stream);




extern int putchar (int __c);
# 537 "/usr/include/stdio.h" 3 4
extern int fputc_unlocked (int __c, FILE *__stream);
```

```
extern int putc_unlocked (int __c, FILE *__stream);
extern int putchar_unlocked (int __c);
```

```
extern int getw (FILE *__stream);
```

```
extern int putw (int __w, FILE *__stream);
```

```
extern char *fgets (char *__restrict __s, int __n, FILE *__restrict __stream)
     ;
# 603 "/usr/include/stdio.h" 3 4
extern __ssize_t __getdelim (char **__restrict __lineptr,
          size_t *__restrict __n, int __delimiter,
          FILE *__restrict __stream) ;
extern __ssize_t getdelim (char **__restrict __lineptr,
        size_t *__restrict __n, int __delimiter,
        FILE *__restrict __stream) ;
```

```
extern __ssize_t getline (char **__restrict __lineptr,
       size_t *__restrict __n,
       FILE *__restrict __stream) ;
```

```
extern int fputs (const char *__restrict __s, FILE *__restrict __stream);




extern int puts (const char *__s);




extern int ungetc (int __c, FILE *__stream);




extern size_t fread (void *__restrict __ptr, size_t __size,
       size_t __n, FILE *__restrict __stream) ;




extern size_t fwrite (const void *__restrict __ptr, size_t __size,
        size_t __n, FILE *__restrict __s);
# 673 "/usr/include/stdio.h" 3 4
extern size_t fread_unlocked (void *__restrict __ptr, size_t __size,
         size_t __n, FILE *__restrict __stream) ;
extern size_t fwrite_unlocked (const void *__restrict __ptr, size_t __size,
          size_t __n, FILE *__restrict __stream);
```

```
extern int fseek (FILE *__stream, long int __off, int __whence);



extern long int ftell (FILE *__stream) ;



extern void rewind (FILE *__stream);
# 707 "/usr/include/stdio.h" 3 4
extern int fseeko (FILE *__stream, __off_t __off, int __whence);



extern __off_t ftello (FILE *__stream) ;
# 731 "/usr/include/stdio.h" 3 4
extern int fgetpos (FILE *__restrict __stream, fpos_t *__restrict __pos);



extern int fsetpos (FILE *__stream, const fpos_t *__pos);
# 757 "/usr/include/stdio.h" 3 4
extern void clearerr (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern int feof (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;

extern int ferror (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;



extern void clearerr_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int feof_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int ferror_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;







extern void perror (const char *__s);
```

```
# 1 "/usr/include/x86_64-linux-gnu/bits/sys_errlist.h" 1 3 4
# 26 "/usr/include/x86_64-linux-gnu/bits/sys_errlist.h" 3 4
extern int sys_nerr;
extern const char *const sys_errlist[];
# 782 "/usr/include/stdio.h" 2 3 4



extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;



extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
# 800 "/usr/include/stdio.h" 3 4
extern FILE *popen (const char *__command, const char *__modes) ;




extern int pclose (FILE *__stream);




extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 868 "/usr/include/stdio.h" 3 4

# 3 "basicProgram.c" 2
```

```
# 3 "basicProgram.c"
int main()
{
 printf("Hello World");
}
```

In the pre-processed code

- Comments are removed from the program
- include statement is replaced by a lot of code

1.2.3 Assembly level code - basicprogram.s

```
.file "basicProgram.c"
.text
.section .rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
leaq .LC0(%rip), %rdi
movl $0, %eax
call printf@PLT
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
.section .note.GNU-stack,"",@progbits
```

1.2.4 Machine level code - basicprogram.o

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0001 003e 0001 0000 0000 0000 0000 0000
```

```
0000020 0000 0000 0000 0000 02d8 0000 0000 0000
0000030 0000 0000 0040 0000 0000 0040 000d 000c
0000040 4855 e589 8d48 003d 0000 b800 0000 0000
0000050 00e8 0000 b800 0000 0000 c35d 6548 6c6c
0000060 206f 6f57 6c72 0064 4700 4343 203a 5528
0000070 7562 746e 2075 2e37 2e35 2d30 7533 7562
0000080 746e 3175 317e 2e38 3430 2029 2e37 2e35
0000090 0030 0000 0000 0000 0014 0000 0000 0000
00000a0 7a01 0052 7801 0110 0c1b 0807 0190 0000
00000b0 001c 0000 001c 0000 0000 0000 001c 0000
00000c0 4100 100e 0286 0d43 5706 070c 0008 0000
00000d0 0000 0000 0000 0000 0000 0000 0000 0000
00000e0 0000 0000 0000 0000 0001 0000 0004 fff1
00000f0 0000 0000 0000 0000 0000 0000 0000 0000
0000100 0000 0000 0003 0001 0000 0000 0000 0000
0000110 0000 0000 0000 0000 0000 0000 0003 0003
0000120 0000 0000 0000 0000 0000 0000 0000 0000
0000130 0000 0000 0003 0004 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0000 0000 0003 0005
0000150 0000 0000 0000 0000 0000 0000 0000 0000
0000160 0000 0000 0003 0007 0000 0000 0000 0000
0000170 0000 0000 0000 0000 0000 0000 0003 0008
0000180 0000 0000 0000 0000 0000 0000 0000 0000
0000190 0000 0000 0003 0006 0000 0000 0000 0000
00001a0 0000 0000 0000 0000 0010 0000 0012 0001
00001b0 0000 0000 0000 0000 001c 0000 0000 0000
00001c0 0015 0000 0010 0000 0000 0000 0000 0000
00001d0 0000 0000 0000 0000 002b 0000 0010 0000
00001e0 0000 0000 0000 0000 0000 0000 0000 0000
00001f0 6200 7361 6369 7250 676f 6172 2e6d 0063
0000200 616d 6e69 5f00 4c47 424f 4c41 4f5f 4646
0000210 4553 5f54 4154 4c42 5f45 7000 6972 746e
0000220 0066 0000 0000 0000 0007 0000 0000 0000
0000230 0002 0000 0005 0000 fffc ffff ffff ffff
0000240 0011 0000 0000 0000 0004 0000 000b 0000
0000250 fffc ffff ffff ffff 0020 0000 0000 0000
0000260 0002 0000 0002 0000 0000 0000 0000 0000
0000270 2e00 7973 746d 6261 2e00 7473 7472 6261
0000280 2e00 6873 7473 7472 6261 2e00 6572 616c
0000290 742e 7865 0074 642e 7461 0061 622e 7373
00002a0 2e00 6f72 6164 6174 2e00 6f63 6d6d 6e65
00002b0 0074 6e2e 746f 2e65 4e47 2d55 7473 6361
00002c0 006b 722e 6c65 2e61 6865 665f 6172 656d
00002d0 0000 0000 0000 0000 0000 0000 0000 0000
*
0000310 0000 0000 0000 0000 0020 0000 0001 0000
```

```
0000320 0006 0000 0000 0000 0000 0000 0000 0000
0000330 0040 0000 0000 0000 001c 0000 0000 0000
0000340 0000 0000 0000 0000 0001 0000 0000 0000
0000350 0000 0000 0000 0000 001b 0000 0004 0000
0000360 0040 0000 0000 0000 0000 0000 0000 0000
0000370 0228 0000 0000 0000 0030 0000 0000 0000
0000380 000a 0000 0001 0000 0008 0000 0000 0000
0000390 0018 0000 0000 0000 0026 0000 0001 0000
00003a0 0003 0000 0000 0000 0000 0000 0000 0000
00003b0 005c 0000 0000 0000 0000 0000 0000 0000
00003c0 0000 0000 0000 0000 0001 0000 0000 0000
00003d0 0000 0000 0000 0000 002c 0000 0008 0000
00003e0 0003 0000 0000 0000 0000 0000 0000 0000
00003f0 005c 0000 0000 0000 0000 0000 0000 0000
0000400 0000 0000 0000 0000 0001 0000 0000 0000
0000410 0000 0000 0000 0000 0031 0000 0001 0000
0000420 0002 0000 0000 0000 0000 0000 0000 0000
0000430 005c 0000 0000 0000 000c 0000 0000 0000
0000440 0000 0000 0000 0000 0001 0000 0000 0000
0000450 0000 0000 0000 0000 0039 0000 0001 0000
0000460 0030 0000 0000 0000 0000 0000 0000 0000
0000470 0068 0000 0000 0000 002a 0000 0000 0000
0000480 0000 0000 0000 0000 0001 0000 0000 0000
0000490 0001 0000 0000 0000 0042 0000 0001 0000
00004a0 0000 0000 0000 0000 0000 0000 0000 0000
00004b0 0092 0000 0000 0000 0000 0000 0000 0000
00004c0 0000 0000 0000 0000 0001 0000 0000 0000
00004d0 0000 0000 0000 0000 0057 0000 0001 0000
00004e0 0002 0000 0000 0000 0000 0000 0000 0000
00004f0 0098 0000 0000 0000 0038 0000 0000 0000
0000500 0000 0000 0000 0000 0008 0000 0000 0000
0000510 0000 0000 0000 0000 0052 0000 0004 0000
0000520 0040 0000 0000 0000 0000 0000 0000 0000
0000530 0258 0000 0000 0000 0018 0000 0000 0000
0000540 000a 0000 0008 0000 0008 0000 0000 0000
0000550 0018 0000 0000 0000 0001 0000 0002 0000
0000560 0000 0000 0000 0000 0000 0000 0000 0000
0000570 00d0 0000 0000 0000 0120 0000 0000 0000
0000580 000b 0000 0009 0000 0008 0000 0000 0000
0000590 0018 0000 0000 0000 0009 0000 0003 0000
00005a0 0000 0000 0000 0000 0000 0000 0000 0000
00005b0 01f0 0000 0000 0000 0032 0000 0000 0000
00005c0 0000 0000 0000 0000 0001 0000 0000 0000
00005d0 0000 0000 0000 0000 0011 0000 0003 0000
00005e0 0000 0000 0000 0000 0000 0000 0000 0000
00005f0 0270 0000 0000 0000 0061 0000 0000 0000
```

```
0000600 0000 0000 0000 0000 0001 0000 0000 0000
*
0000618
```

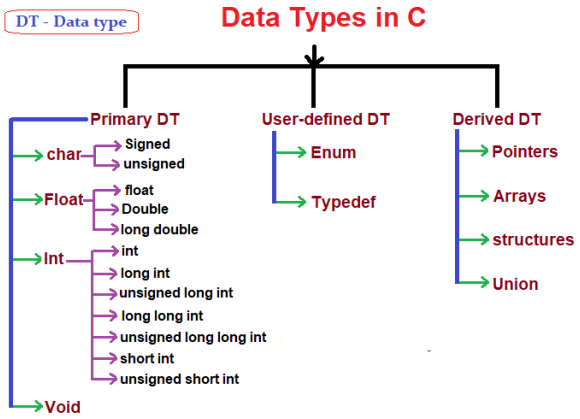
2 Nuts and Bolts of C

The C language was developed by Dennis Ritchie in 1972 at Bell Telephone laboratories. C++ was developed by Bjarne Stroustrup at Bell Labs as an extension to C, starting in 1979. C++ adds many new features to the C language.

2.1 Comments

- Single line comment - //
- Multiple line comment - /* ... */

2.2 Data Types



Type	Size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-2^{15} to $2^{15} - 1$ -2^{31} to $2^{31} - 1$
unsigned int	2 or 4 bytes	0 to 2^{16} 0 to 2^{32}
short	2 bytes	-2^{15} to $2^{15} - 1$
unsigned short	2 bytes	0 to $2^{15} - 1$
long	8 bytes	-2^{63} to $2^{63} - 1$
unsigned long	8 bytes	0 to 2^{64}

Type	Size	Value range	Precision
float	4 bytes	1.2e-38 to 3.4e+38	6 decimal places
double	8 bytes	2.3e-308 to 1.7e+308	15 decimal places
long double	10 bytes	3.4e-4932 to 1.1e+4932	19 decimal places

2.2.1 Modifiers

A modifier is a keyword prefixed to a declaration to change its storage size and properties of data types. Available modifiers in C

- short
  - Only for int datatype
  - Changes the size to 2 bytes
- long
  - Allows to increase the size of *int* and *double* datatypes
  - long int - 4 bytes (32 bit system)
  - long double - 12 bytes (32 bit system)
  - Sizes are relative underlying processor
- signed
  - Can store positive oe negative values
  - By default a variable declared without any modifier is prefixed with *signed*
- unsigned
  - Can store positive values only
- long long
  - Can only be used with *int* datatype
  - long long int - 8 bytes
  - Size may vary based on the underlying processor

2.2.2 Lvalues and Rvalues in C

- Certain fixed values which may not alter during the execution of the program are called **literals**.
  - 85 : decimal literal
  - 0213 : octal literal
  - 0x4a : hexadecimal literal
  - 30u : unsigned int literal
  - 30l : long literal
  - 30ul : unsigned long literal
- **lvalue** - Expressions that refer to a memory location are called "lvalue" expressions.
  May appear on the left hand side or the right hand side of the expression.

Variables are *lvalues* hence they can appear on LHS or RHS of an expression.

- **rvalue** - Refers to the data value that is stored at some address in memory. May appear on the right hand side but not on the left hand side of the expression

  Numeric literals are *rvalues* hence can appear only on RHS of an expression.

2.3 Keywords

keyword is a reserved word in C, and cannot be used for a variable name. There are 32 keywords in C.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

2.4 Identifiers

The names given to variables, functions, structures etc are called **identifiers**. 52 alphabets (Lower and Upper case), 10 numerical digits (0-9) and an underscore can be used to form an identifier.

Rules for constructing Identifiers

- First character should be alphabet or underscore
  (However using underscore is discouraged as it would be difficult to differentiate between user defined variables and system variables)
- Should not begin with a numerical digit
- It is case sensitive
- Comma and blank spaces cannot be used
- Keywords cannot be used
- Length of the identifier should not be more than 31 characters

2.5 Defining Constants

2.5.1 #define Preprocessor

Using the following syntax it is possible to define a constant.

```
#define identifier value
```

2.5.2 const Keyword

const prefix can be used to declare constants in a program.

```
const type variable = value;
```

If the value of a variable declared as a constant is modified in the program then the compiler throws a compile time error denoting *assignment to read only variable*

2.6 Escape Sequence

Using backslash causes "escape" from the normal way the characters are interpreted by the compiler.

- \ b - Backspace
- \ t - Horizontal Tab
- \ v - Vertical Tab
- \ n - New line
- \ r - Carriage Return
- \ 0 - NULL
- \ \ - Backslash
- \ " - Double Quote
- \ f - Form feed
- \ ' - Single Quote
- \ ? - Question mark

2.7 Storage Class

A storage class defines the scope (visibility) and life-time of the variables and/or functions within a C program. They precede the type that they modify.

2.7.1 auto Storage Class

The **auto** storage class is the default storage class for all local variables. It can be used only within functions where these local variables are defined.

2.7.2 register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM.

- Maximum size of the variable is equal to the register size
- It does not have a memory location associated with it.
- Usually variables that require quick access are stored in registers
- The system does not necessarily store the variable in a register, it MIGHT be stored based on the hardware and implementation restrictions

2.7.3 static Storage Class

The **static** storage class instructs the compiler to not delete the variable once it is out of scope. That is once declared its life time is equal to the duration of the program.

- Useful to make variables that have to maintain their values between function calls.
- static modifier for global variables, will restrict the scope of the variable to the file in which it is declared.

2.7.4 extern Storage Class

The **extern** storage class is used when two or more files share the same global variables or functions

- When "extern" is used, it is not possible to initialize the variable.
- "extern" is used in file2 to provide the reference of defined variable in file1.
- To use an external variable you must declare it with extern keyword prefix inside a function that wants to access it.

2.8 Operators

Logical Operators

Operator	Description
&&	Logical AND operator If both operands are non zero, then the condition becomes true.
||	Logical OR operator If any of the two operands is non zero then condition becomes true.
!	Logical NOT operator Reverses the logical state of the operand

The other operators like Arithmetic, Relational, Assignment and bitwise are straightforward.

Misc Operators

Operator	Description
sizeof()	Returns the size of a variable
&	Returns the address of a variable
*	Pointer to a variable
?:	Conditional Expression condition ? true : false;

Increment and decrement operators

These operators are used to increment or decrement the value of the operand by 1. These are unary operators.
Types:

- Pre decrement
- Pre increment
- Post decrement
- Post increment

Understanding with examples

```
int i = 10;
int j = i++, k = ++i;
printf("The value of j is: %d\n",j);
printf("The value of k is: %d",k);

OUTPUT
The value of j is: 10
The value of k is: 12
```

While initializing the variable j post increment is used, so the value of i is assigned first and then incremented.
While initializing the variable k pre increment is used, so the value of i is incremented first and then assigned.

```
int i = 10;
printf("%d\t%d\t%d\n",i++,i++,i);

OUTPUT
11 10 12
```

Execution of the printf statement, happens from right to left.

- The third assignment requires the value of i
- The second assignment assigns i, and then increments the value of i, same is the case with the first assignment
- After all the incrementing and decrementing is done, the final value of i is used in the third assignment.

```
int i = 10;
printf("%d\t%d\t%d\n",i++,i++,--i);
OUTPUT
10 9 11


int i = 10;
printf("%d\t%d\t%d\n",--i,--i,--i);
OUTPUT
7 7 7

int i = 10;
printf("%d\t%d\t%d\t%d\t%d\n",--i,++i,i,i++,i--);
OUTPUT
10 10 10 9 10
```

NOTE: Remember that the execution within the printf function is from right to left.

comma operator

It is used to link an expression together

```
int i,j = 10, k;
printf("%d\n%d\n%d\n",i,j,k);

OUTPUT
0
10
0
```

2.8.1 Operator Precedence

Level	Operators	Description	Associativity
15	() [] -> . ++ --	Function Call Array Subscript Member Selectors Postfix Increment/Decrement	Left to Right
14	++ -- + - ! ~ (type) * & sizeof	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement Casting Dereferencing Address of Find size in bytes	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== !=	Equality Inequality	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

2.8.2 Associativity

The associativity of operands indicates the order in which same precedence operators appearing in an expression are evaluated.

2.9 Format specifier

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output.
%p	It is used to print the address in a hexadecimal form.
%c	It is used to print the unsigned character.
%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value.

2.10 Typecasting

Typecasting is a way to convert a variable from one data type to another data type.

```
float x = 7/5;
printf("%f",x);

OUTPUT: 1.000000
```

To avoid loss of data typecasting can be done.

```
float x = (float)7/5;
printf("%f",x);

OUTPUT: 1.400000
```

There are two types of typecasting

- Implicit
  That is done automatically in the program
- Explicit
  Need to be explicitly mentioned, as in the above example.

Certain inbuilt typecasting functions are

Typecast Function	Description
atoi()	Convert string to int
atof()	Convert string to float
atol()	Convert string to long
itoa()	Convert int to string
ltoa()	Convert long to string

2.11 Decision Making statements

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	nested switch statements You can use one **switch** statement inside another **switch** statement(s).

2.12 Loops

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is more like a while statement, except that it tests the condition at the end of the loop body.
4	nested loops You can use one or more loops inside any other while, for, or do..while loop.

2.12.1 Loop control statements

- **break**: Terminates the loop, to exit the switch statement.
- **continue**: Transfers the execution to the first statement (condition) of the loop
- **goto**: Transfers the control to the labelled statement

2.12.2 Infinite loop

- ```
  for( ; ; )
  ```
- ```
  while(True)
  ```

2.13 Functions

A group of statements that together perform a task is called a *function*.

- **Function Declaration** tells the compiler about a function's name, return type and parameters.

  ```
  return_type function_name( parameters_list);
  ```

  - **Return type** is the datatype of the value the function returns. If a function does not return anything return type is **void**.
  - **Parameters** are the data passed to a function when it is called. The type, order and number of parameters is called *Parameter list*.

  - Parameter name are not important in function declaration, only parameter type are important.
  - **function name** and **parameter list** constitute the **function signature**

- **Function Definition** provides the actual body of the function

2.13.1 Function Arguments

- **Call by Value**
  This method sends a copy of the actual arguments to the function. So any changes made to these within the function are not reflected outside the function.

- **Call by reference**
  This method sends a copy of the address of the actual arguments to the function. The address is used to access the actual arguments. So any changes made to these within the function are reflected outside the function.

By default C uses ***call by value***

2.14 Scope of Variables

A part of the program where a particular variable is accessible and beyond which it is not accessible is called **scope of a variable**.

- **local variables**
  - Defined within a function
  - Scope of the variable - Function where it is defined
  - When defined the system does not initialize it. Some garbage value is stored

- **global variables**
  - Defined outside all functions
  - Scope of the variable - Entire program
  - When defined the system automatically initializes it to the default value
    - int - 0
    - char - ‘ 0’
    - float - 0
    - double - 0
    - pointer - NULL

- **formal parameters** Defined in the function definition

NOTE: Same name can be given to both global and local variables, but the value of the local variable inside a function will take preference.

2.15 Enumerations

Enumerations or enum is a user defined datatype in C. These provide a convenient way of associating constant values with names.

```
enum week{Mon = 10, Tues, Wed, Thur, Fri, Sat, Sun};

int main()
{
int i;
i = Wed;
printf("%d\n",i);
return 0;
}
OUTPUT
12
```

2.16 Input Output

A **stream** is a logical interface to interact with various devices. Usually a stream is backed by a buffer allocated onto the memory since reads and writes to memory are fast.

- **getchar()** function
  Reads one character from the screen and returns it as an integer.

  ```
  int c;
  c = getchar();
  ```

- **putchar()** function
  Writes one character to the screen and returns the same character.

  ```
  putchar(c);
  ```

- **gets()** function
  Reads a line from *stdin* into the buffer pointed by the argument until a terminating newline or EOF.

  ```
  char str[100];
  gets(str);
  ```

- **puts()** function
  Writes the argument with a trailing newline to *stdout*.

```
puts(str);
```

- **scanf()** function
  Reads the input from *stdin* and scans the input according to the $format$ (%s, %d, %x, %f) provided.

```
char str[100];
int i;
scanf("%s %d", str, &i);
```

- **printf()** function
  Writes the output to *stdout* in the format specified.

```
int i=10;
printf("%d",i);
```

NOTE: **scanf()** function stops reading as soon as it encounters a space.

3 Pointers

A pointer is a variable whose value is the address of another variable. The storage space to store a pointer variable depends upon the processor architecture, for 32bit processor a pointer would take 4 bytes and on a 64bit processor it would take 8 bytes. Pointer variable can be declared as

```
type *var-name;
```

type : pointer's base type
var-name : name of pointer variable
* : Dereferencing/ indirection operator

The actual datatype of the value of all pointers is *long hexadecimal number*. Type simply refers to the type of data that the pointer points to.

3.1 NULL pointer

A pointer that is assigned NULL is called a NULL pointer.

```
int *ptr = NULL;
printf( "The value of ptr is: %x", ptr );

OUTPUT
The value of ptr is: 0
```

3.2 Wild Pointer

A Wild pointer is a pointer which is declared but not yet initialized to a value.

3.3 Pointer Arithmetic

Depends on the datatype of the pointer variable and not the datatype of data it is pointing to.
It is best explained with examples.

Incrementing and Decrementing a Pointer

```
int *a;
double *b;
char *c;

a = 1000;
b = 2000;
c = 3001;
```

```
a++;
b++;
c--;

printf( "The value of a is: %x", a );
printf( "The value of b is: %x", b );
printf( "The value of c is: %x", c );

OUTPUT
The value of a is: 1004
The value of b is: 2008
The value of c is: 3000
```

To store an integer requires 4 bytes and for a double 8 bytes are required. Hence incrementing the respective pointers will increment it by the size of the respective variable.

```
int a[] = {11,13,17,19,23};
int *p = a;
int *q = a+1;
p = p + 2;
printf("%d\n",*p);
printf("%d\n",*q);

OUTPUT
17
13
```

Since the pointers are of type int, adding 1 means moving the pointer by 4 bytes, adding 2 means moving the pointer by 8 bytes. Note that just the name of the array is also like a pointer.

Pointer Comparisons
It is possible to compare pointers pointing to variables that are related.

```
int a[] = {2,3,5};
int i, *ptr;

ptr = a;
i = 0;

while (ptr <= &a[2] )
{
   printf("Address of a[%d] = %x",i, ptr);
   i++;
```

```
    ptr++;
}

OUTPUT
Address of a[0] = 1000
Address of a[1] = 1004
Address of a[2] = 1008
```

ptr is incremented to traverse the array, this is compared with the address of the last element to finish the task.

3.4 Indirection Operator with increment/decrement operators

NOTE: The precedence of indirection operator and increment/decrement operator are same and they associate from right to left.

```
int a[] = {23,29,31,37,41};
int *p = a;
int *q = a;
int *r = a;
printf("%d\n",*p++);
printf("%d\n",*p);

printf("%d\n",++*q);
printf("%d\n",*q);

printf("%d\n",*++r);
printf("%d\n",*r);

OUTPUT
23
29
24
24
29
29
```

- In the first pair printf function, as post-increment is used, first the pointer is dereferenced and printed. Next it is incremented to point to the next variable.
- In the second pair printf function, as pre-increment is used, first the pointer is dereferenced and incremented, the incremented value is printed. The increment is done using the memory location of the variable hence this change will be permanently reflected in the array.

- In the third pair of printf functions, the pointer is first incremented to point to the next element and then it is dereferenced.

3.5 Array of Pointers

Each element of the array is a pointer to some other variable.

```
int *ptr[2];
int a[] = {1,3,5};
int b[] = {2,4,6};
ptr[0] = a;
ptr[1] = b;
printf("%d\n",*ptr[0]);
printf("%d",*ptr[1]);

OUTPUT
1
2
```

3.6 Pointer to a pointer or Double Pointer

Syntax to declare a pointer to a pointer.

```
int **ptr;
```

An example program to better understand this concept

```
int a = 10;
int *p = &a;
int **q = &p;
printf("%d\n",*p);
printf("%d\n",**q);
printf("%p\n",p);
printf("%p\n",*q);
printf("%p\n",q);

OUTPUT
10
10
0x7ffe573086a4
0x7ffe573086a4
0x7ffe573086a8
```

Dereferencing the pointer to pointer variable once, will return the address of the pointer, dereferencing the pointer to pointer twice, will return the data stored in the location pointed to by the pointer.

3.7 Passing pointer to a function

```
 void ReceivePointer(int *ptr)
{
    printf("The data in the received variable %d\n",*ptr);
    printf("The address received is %p\n",ptr);
}
int main()
{
    int a = 10;
    ReceivePointer(&a);
    printf("The address of the variable in main function %p\n",&a);
    return 0;
}

OUTPUT
The data in the received variable 10
The address received is 0x7fffcae636a4
The address of the variable in main function 0x7fffcae636a4
```

3.8 Returning pointer from a function

- Do not return address of local variable. Declare it as **static** in case a local variable has to be returned
- Address of global variables can be returned.

3.9 Dangling Pointer

If the pointer is not initialized with a valid address it is called **dangling pointer**

- Usually when an object is deleted, the pointer that was previously pointing to it will be a dangling pointer.
- When variable goes out of scope then the pointer pointing to it will become a dangling pointer.
- **free()** function can be used to deallocate the memory assigned to a pointer.
- Dangling pointers error can be avoided by initializing the pointer to NULL value.

3.10 Void Pointer

void pointer is a special kind of pointer supported in C to which we can associate any data type.

- A void pointer cannot be dereferenced
- size is 1 byte by default
- Pointer arithmetic can be performed

3.11 const Pointer

A constant pointer cannot change the address of the variable it is pointing to i.e. Address will remain constant. However the value can be changed.

```
int *const ptr;
```

Ex:

```
int *const ptr;
int a = 10, b = 20;
ptr = &a;
ptr = &b;
printf("Content of ptr %d\n",*ptr);

OUTPUT
ptr = &b;
error: assignment of read-only variable ‘ptr’
```

3.12 Pointer to const

The value of the variable to which pointer is pointing to cannot be changed. However the address the pointer is pointing to can be changed.

```
const int *ptr;
```

Ex:

```
int b = 10, a = 20;
const int *ptr;
ptr = &a;
ptr = &b;
*ptr = 30;

OUTPUT
*ptr = 30;
error: assignment of read-only location ‘*ptr’
```

3.13 const pointer to a const

Neither the value of the variable pointed to by the pointer nor the address of the variable the pointer is pointing to can be changed.

```
const int *const ptr;
```

3.14 Function Pointers

A function pointer is a feature which makes it easy to store address of a function into a pointer variable. Once an address of a function is stored in a variable, the function can be invoked from any part of the code which has access to the function pointer.

- A function pointer points to code and not data
- function's name can also be used to get function address.

```
void fun(int a)
{
printf("%d\n",a);
}
int main()
{
void (*fun_ptr)(int) = fun;
void (*f_ptr)(int) = &fun;

(*f_ptr)(9);
(*fun_ptr)(8);
}
OUTPUT
9
8
```

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

3.14.1 Array of function pointers

```
void add(int a, int b)
{
    printf("Sum is %d\n", a+b);
}
void subtract(int a, int b)
{
```

```
    printf("Difference is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Product is %d\n", a*b);
}

int main()
{
    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);
}
OUTPUT
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Product is 150
```

3.14.2 Function pointers can be passed as arguments to functions

```
void sum(int a, int b) { printf("Sum is: %d\n",a+b);}
void call_fun( void (*fun)())
{
int a = 7, b = 11;
fun(a,b);
}
int main()
{
call_fun(sum);
}
OUTPUT
Sum is: 18
```

Notice that in the definition of call_fun, the argument is a function pointer, however the arguments for the function to be called cannot be passed from the main program as we are only passing the address to call_fun from the main program.

3.14.3 Function pointers can be returned from a function

```
typedef void (*function_ptr)(int,int);
void add(int a, int b) { printf("Sum is %d\n", a+b); }

void subtract(int a, int b) { printf("Difference is %d\n", a-b); }

void multiply(int a, int b) { printf("Product is %d\n", a*b); }
function_ptr choice(int a)
{
if(a>2)
printf("Invalid choice");
else if(a==0)
return add;
else if(a==1)
return subtract;
else
return multiply;
}
int main()
{
int a;
printf("Enter Choice: 0 for add, 1 for subtract and 2 for multiply\n");
scanf("%d",&a);
function_ptr fun = NULL;
fun = choice(a);
fun(10,5);
}
OUTPUT
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Product is 50
```

typedef has to be used to define the function pointer of the required type, and an instance of it can be used to achieve the desired functionality.

4 Derived Datatypes

4.1 Arrays

Arrays is used to store a collection of data of same type.

Declaring arrays Requires datatype and size of the array

```
type arrayName [arraySize];
```

Initializing arrays

```
int a[5] = {9,8,98,14,12};\\
int a[] = {9,6,3,12,21};
```

Passing arrays to function
Array can be passed to the function as a pointer by specifying the array's name without an index.

4.2 Strings

1-d array of characters terminated by a **null** character.

- The size of the string is 1 greater than the number of characters in the string. (because of null character at the end)
- The null character is automatically placed at the end of the string by the C compiler.

4.2.1 Inbuilt String functions

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

4.3 Structures

A user-defined data type that allows to combine data items of different types.
Defining a Structure

```
struct [structure tag] {

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

The **structure tag** is optional. It is used to instantiate objects of this type wherever necessary in the program.

4.3.1 Accessing Structure members

member access operator (.) is used to access members of a structure.

```
struct records {
     char name[50];
     int runs_scored;
     int wickets_taken;
     };
```

```
int main()
{
 struct records P1, P2;
 strcpy(P1.name, "Sachin");
 P1.runs_scored = 18426;
 P1.wickets_taken = 154;

 strcpy(P2.name, "Kallis");
 P2.runs_scored = 11579;
 P2.wickets_taken = 273;

 printf("Player1 name: %s\n",P1.name);
 printf("Player1 runs scored: %d\n",P1.runs_scored);
 printf("Player1 wickets taken: %d\n",P1.wickets_taken);
 return 0;
}

OUTPUT
Player1 name: Sachin
Player1 runs scored: 18426
Player1 wickets taken: 154
```

4.3.2 Passing Structures to Functions

The function declaration is as follows

```
return_type function_name (struct structure_tag struct_object);
```

4.3.3 Pointers to Structures

Syntax to declare pointer to a structure is same as declaring a pointer to any other variable.

```
     struct records *struct_pointer;
```

To access the members of a structure → is used.

```
struct records {
     char name[50];
     int runs_scored;
     int wickets_taken;
     };
void printRecords(struct records *ptr);
int main()
```

```
{
 struct records P1, P2;
 strcpy(P1.name, "Sachin");
 P1.runs_scored = 18426;
 P1.wickets_taken = 154;

 strcpy(P2.name, "Kallis");
 P2.runs_scored = 11579;
 P2.wickets_taken = 273;
 printRecords(&P2);
 return 0;
}

void printRecords(struct records *ptr)
{
 printf("Player name: %s\n",ptr->name);
 printf("Runs scored: %d\n",ptr->runs_scored);
 printf("Wickets taken: %d\n",ptr->wickets_taken);
}

OUTPUT
Player name: Kallis
Runs scored: 11579
Wickets taken: 273
```

4.3.4 Bit Fields

Bit fields allow the packing of data in a structure. i.e. The user can define variables with non standard size (Ex: int of size 9 bits). C automatically packs the bit fields as compactly as possible. This is done to utilize computer memory efficiently.

Declaration of bit fields:

```
struct
{
data_type variable_name : size_in_bits;
};
```

Since structures are user defined variables, user will have an idea about the memory requirements of each element. And bit fields provide a way to allocate only the required amount of memory to each member and hence prevents the wastage of memory.

```
struct time
{
unsigned int hours: 5; // Size restricted to 5 bits
unsigned int mins:6; // Size restricted to 6 bits
```

```
unsigned int seconds:6; // Size restricted to 6 bits
};
int main()
{
struct time t = {10,30,45};
printf("The time is %d:%d:%d\n",t.hours,t.mins,t.seconds);
return 0;
}

OUTPUT
The time is 10:30:45
```

4.3.5 Polymorphism of function pointers

```
typedef void (*salary)(void);
typedef void (*designation)(void);

typedef struct employee
{
salary Salary;
designation Post;
} Employee;

typedef struct manager
{
salary Salary;
designation Post;
} Manager;

void employeeSalary()
{ printf("Employee salary is: %d\n",1000); }
void managerSalary()
{ printf("Manager salary is: %d\n",5000); }
void managerDesignation()
{ printf("Designation is Manager\n"); }
void employeeDesignation()
{ printf("Designation is Employee\n"); }
void Designation(Manager *m)
{ m->Post(); }
void Earning(Manager *m)
{ m->Salary(); }

int main()
{
Employee e = {.Salary = employeeSalary, .Post = employeeDesignation};
Manager m = {.Salary = managerSalary, .Post = managerDesignation};
```

```
Designation(&e);
Earning(&e);

Designation(&m);
Earning(&m);
}
OUTPUT
Designation is Employee
Employee salary is: 1000
Designation is Manager
Manager salary is: 5000
```

Since both the structures have the similar members (function pointers), polymorphism is possible.

4.4 Unions

A **union** is a special data type, that allows to store different data types in the same memory location.

- Only one member of the union can contain a value at any given time.

Defining a Structure

```
union [union tag] {

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

The **structure tag** is optional. It is used to instantiate objects of this type wherever necessary in the program.

- The memory occupied by a union is equal to the memory requirement of its largest member.

4.4.1 Accessing Union members

member access operator (.) is used to access members of a union.

```
union sales
{
 char product_name[20];
 int tax;
```

```
 int profit;
}p1;
int main()
{
 union sales p2;
 printf("Memory occupied by instantiations of Union is %ld\n",sizeof(p2));
 strcpy(p1.product_name, "LG Washing machine");
 p1.tax = 14;
 p1.profit = 1000;
 printf("Product name: %s\n",p1.product_name);
 printf("Tax: %d %%\n",p1.tax);
 printf("Profit: %d\n",p1.profit);

 p2.tax = 20;
 printf("Tax: %d %%\n",p2.tax);
 p2.profit = 5000;
 printf("Profit: %d\n",p2.profit);
 strcpy(p2.product_name, "Samsung TV");
 printf("Product name: %s\n",p2.product_name);
 return 0;
}

OUTPUT
Memory occupied by instantiations of Union is 20
Product name:
Tax: 1000 %
Profit: 1000
Tax: 20 %
Profit: 5000
Product name: Samsung TV
```

4.5 typedef

typedef is a keyword used in C programming to provide some meaningful names to the already existing data type name.
Ex:

```
typedef unsigned char BYTE;
```

After this, the identifier BYTE can be used as an abbreviation for the data type **unsigned char**.

```
BYTE b1,b2;
```

4.5.1 typedef for Structures

typedef can be used with structure to define a new data type, and then use that data type to define structure variables directly.

```
typedef struct Books
{
 char title[50];
 char author[50];
 char subject[50];
 int book_ID;
} BOOK;

int main()
{
 BOOK b1;
 strcpy(b1.title,"Mind At Play");
 strcpy(b1.author,"Rob Goodman");
 strcpy(b1.subject,"Biography of Claude Shannon");
 b1.book_ID = 1;

 printf("Book title: %s\n",b1.title);
 printf("Author: %s\n",b1.author);
 printf("Subject: %s\n",b1.subject);
 printf("Book ID: %d\n",b1.book_ID);
 return 0;
}

OUTPUT
Book title: Mind At Play
Author: Rob Goodman
Subject: Biography of Claude Shannon
Book ID: 1
```

BOOK is the alias used for declaring objects for struct Books

4.5.2 typedef vs #define

#define is a C-directive which is used to define aliases for various data types similar to typedef

- **typedef** is limited to giving symbolic names to types, where as **#define** can be used to define alias for values as well
- **typedef** interpretation is performed by compiler **#define** statements are processed by pre-processor.

5 Recursion and Dynamic Programming

Sometimes when the problem is too complex to solve because it is too big. There are two approaches that can be used to tackle this situation.

- Top-Down approach
- Bottom-Up approach

5.0.1 Top-Down approach

The first step is to create an overview of the system without specifying much details of the subsystems.
It's like starting with an outline and then filling things in as you go until you have a story.
Example: To write a program that plays chess

STEP 1

```
PlayChess()
```

Call a function that plays Chess, though the function definition is not yet available. This is the first step where you have defined the problem.

STEP 2

```
PlayChess()
{
    setupboard();
    do till end of game
    {
        player1Move();
        player2Move();
    }
}
```

Though the function definitions are not availbale for the function calls in the above program. You have defined an outline of how the program must look like.

STEP 3
The next step is to fill in the function body.
Advantages:

- Breaking problems into parts help us to identify what needs to be done.
- At each step of refinement new parts will become less complex and therefore easier to solve.
- Parts of solution may turn out to be reusable.
- Breaking problems into parts allows more than one person to solve the problem.

5.0.2 Bottom-Up approach

The first step is to design the individual parts of the main program in detail. And then these designs for the smaller problems are combined together to eventually obtain the solution to the entire problem.
Example: To write a program that plays chess

STEP 1

- Define the conditions for check
- Define how various pieces move
- Define conditions for end of game *checkmate* or *stalemate*

STEP 2

- Define the function for a player to move

```
if(check)
    avoid check and protect the king
else
    make any valid move
```

STEP 3

```
PlayChess()
{
    setupboard();
    do till end of game
    {
        player1Move();
        player2Move();
    }
}
```

5.1 Recursion

Solving a complex problem by first breaking down it into smaller versions of itself and finding solutions to these smaller problems to eventually build the solution to the entire problem. This is the main idea behind **recursion**.

Recursion is a computer programming technique involving the use of a procedure, subroutine, function or algorithm that calls itself in a step having a termination condition so that successive repetitions are processed up to the critical step where the condition is met at which the rest of each repetition is processed from the last one called to the first.

The important questions to be answered before using recursion to solve the given problem

- What is the base case, and can it be solved
- What is the general case
- Does the recursive call make the problem smaller and approach the base case.

Base Case

The base case, or halting case of a function is the problem that we know the answer to, that can be solved without any more recursive calls. The base case is what stops the recursion from continuing on forever. Every sub problem to which the solution is known can be used as a base case.

General Case

The general case is what happens most of the time, and is where the recursive call takes place.

Diminishing Size of Problem

After every recursive call, the problem must be approaching the base case. If this is not satisfied recursion will never end.

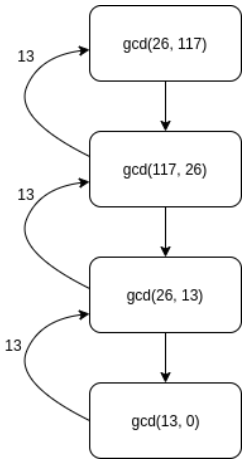
Recursion might not be the most efficient way to implement an algorithm. Each time a function is called, there is a certain amount of *overhead* that takes up memory and system resources. This is because when a function($sayf2$) is called from another function ($sayf1$), all the information about $f1$ must be stored so that the computer can return to it after executing $f2$.

5.1.1 Example programs for Recursion

1. Using Recursion to find GCD of two numbers

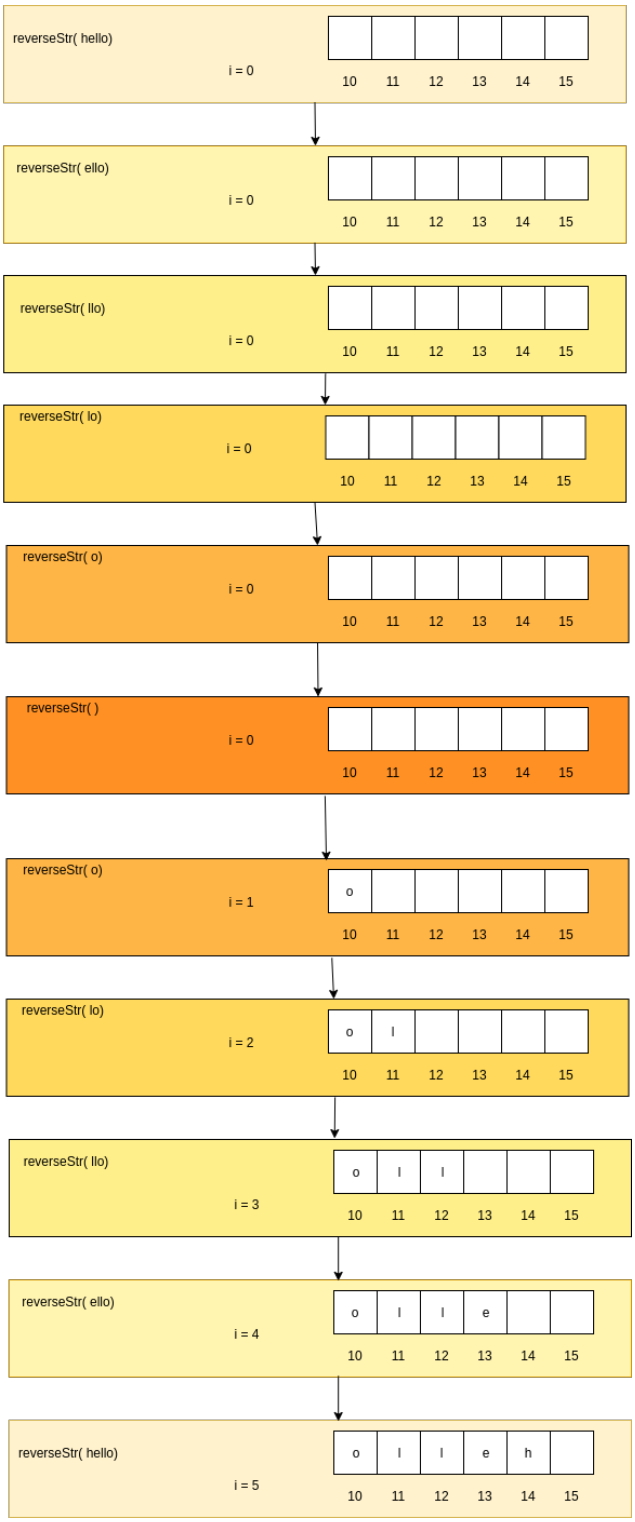
```
int GCD(int a, int b)
{
    if(b!=0)
        return GCD(b,a%b);
    else
        return a;
}
int main()
{
    int a,b;
    printf("Enter two numbers: ");
    scanf("%d %d",&a,&b);
```

```
    int val = GCD(a,b);
    printf("GCD is: %d\n",val);
    return 0;
}
```



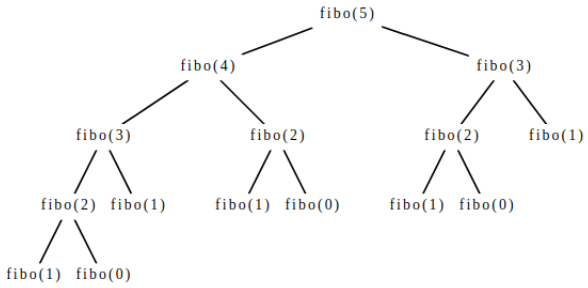
2. Using Recursion to reverse a string

```
char* reverseStr(char str[])
{
    static char reverse[MAX];
    static int i = 0;
    if(*str)
    {
        reverseStr(str+1);
        reverse[i++] = *str;
    }
    return reverse;
}
int main()
{
    char s[25];
    printf("Enter String: ");
    scanf("%s",s);
    char *rev;
    rev = reverseStr(s);
    printf("%s",rev);
    return 0;
}
```


reverseStr(hello)
i = 0
10 11 12 13 14 15
reverseStr(ello)
i = 0
10 11 12 13 14 15
reverseStr(llo)
i = 0
10 11 12 13 14 15
reverseStr(lo)
i = 0
10 11 12 13 14 15
reverseStr(o)
i = 0
10 11 12 13 14 15
reverseStr( )
i = 0
10 11 12 13 14 15
reverseStr(o)
i = 1
o
10 11 12 13 14 15
reverseStr(lo)
i = 2
o l
10 11 12 13 14 15
reverseStr(llo)
i = 3
o l l
10 11 12 13 14 15
reverseStr(ello)
i = 4
o l l e
10 11 12 13 14 15
reverseStr(hello)
i = 5
o l l e h
10 11 12 13 14 15

3. Using Recursion to generate the nth number of Fibonacci sequence

```
int fibo(int n)
{
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
    {
        return fibo(n-1) + fibo(n-2);
    }
}
int main()
{
        int n = 0;
        printf("Enter the required index of the fibonacci sequence to be printed: ");
        scanf("%d",&n);
        int num = fibo(n);
        printf("The %dth number in the Fibonacci sequence is: %d",n,num);
}
```



From the above flowchart it can be seen that $fibo(1)$ is called 4 times. This increases the time and space complexity significantly. An alternative approach would be to save results to such smaller problem and reuse it to solve the bigger problems or similar problems. This is the main idea behind **Dynamic Programming**.

NOTE:

- The main risk of using recursive methods to solve a problem is that we solve identical sub problems multiple times.
- Every recursive solution can also be implemented in a iterative manner.

5.2 Iteration vs Recursion vs Dynamic Programming

- Recursive algorithm is often simple and elegant as compared to the iterative algorithm used to solve the same problem.
- Recursion increases time and space complexity.
- Dynamic programming addresses the main drawback of recursive methods.

5.2.1 Fibonacci - Iterative

```
int fibo(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int x = 0;
    int y = 1;
    for (int i = 2; i < n; i++) {
        int tmp = x + y;
        x = y;
        y = tmp;
    }
    return x + y;
}
```

Time Complexity : O(n)
Space Complexity: O(1)

5.2.2 Fibonacci - Recursive

```
int fibo(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibo(n-1) + fibo(n-2);
}
```

Time Complexity : $O(2^n)$
Space Complexity: O(n)

5.2.3 Fibonacci - Dynamic Programming (Top-Down)

```
int fibo(int n) {
    return fiboRec(n, new int[n+1]);
}

int fiboRec(int n, int[] memo) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (memo[n] == 0) memo[n] = fiboRec(n-1, memo) + fiboRec(n-2, memo);
    return memo[n];
}
```

Time Complexity : O(n)
Space Complexity: O(n)

5.3 Dynamic Programming

From the previous example we observe that, with dynamic programming, the code is simple and at the same time, the space and time complexity is comparable to that of iterative methods.

5.3.1 When to use Dynamic Programming

Dynamic programming cannot be used with every recursive solution. The problem must contain two properties in order to apply Dynamic Programming

- **Overlapping sub problems**
  There are sub problems that arise repeatedly while solving the overall problem
  Ex: fibo(4) is required in computation of both fibo(5), fibo(6) and every fibo(n) where n $>$ 4
- **Optimal sub structure**
  optimal solution of the given problem can be obtained by using the optimal solutions of its sub problems.

Usually Dynamic Programming uses the bottom-up approach to solve a problem. i.e. The simplest possible case of the problem at hand is solved first and the solution is saved. Consequently more complex problems are considered, and the solution to these are obtained using the solutions to the simpler problems.

The top-down approach with Dynamic programming is just recursion with caching intermediate results. This is known as **memoization**.

6 Dynamic Memory Allocation

Allocating the memory at run-time is referred to as **Dynamic Memory Allocation**. The memory allocated is from the free memory block of RAM.

The functions that deal with Dynamic Memory Allocation are

- Malloc
- Calloc
- Realloc
- Free

1. With dynamic memory allocation it is not necessary to specify the size of the memory block as in the case of static allocation where you have to specify the size at the time of compilation.
2. The user must free the memory after it has been used.

6.1 malloc

malloc stands for memory allocator, the function when called reserves block of memory specified as the size argument to the function and returns a void pointer.

If the allocator finds free memory block, then that would be reserved and the address of the beginning of the block is returned as void pointer. If no such memory block which meets the requirement is found it returns NULL value

NOTE: Always initialize the memory to zeros, as it may contain garbage values.

Syntax:

```
datatype *ptr = (datatype*)malloc(size);
```

6.2 calloc

calloc stands for contiguous memory allocator, the function when called reserves multiple blocks of memory each of same size and sets all bytes in the memory to zeros and returns a void pointer pointing to the beginning of that memory block.

If it fails to find a memory block that matches the requirement it returns NULL.

Syntax:

```
datatype *ptr = (datatype*)calloc(number, size);
```

6.3 free

Releases the memory alllocated with malloc, calloc or realloc.

Syntax:

```
free(void *p);
```

6.3.1 memset

A handy function used to initialize the dynamically allocated memory to a particular value.

Syntax:

```
    memset(ptr, value, size);
```

Understanding the working of Dynamic Memory Allocation with the help of a program

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct _student
{
char name[20];
float marks;
} Student;

//Function to dynamically allocate memory of the required size
void* alloc_mem(size_t size)
{
void* ptr = (void*)malloc(size);
if(ptr)
memset(ptr,0,size);

return ptr;
}

//Function to free the dynamically allocated memory
void free_mem(void *ptr)
{
if(ptr)
{
free(ptr);
ptr = NULL;       //To avoid dangling pointers
}
}
```

```
int main()
{
int sections = 0;
int students = 0;

int secIndex = 0;
int studIndex = 0;

float avgALLsections = 0.0;

printf("Enter the number of sections:");
scanf("%d",&sections);

printf("Enter the number of students per section:");
scanf("%d",&students);

Student **records = (Student*)alloc_mem(sections * sizeof(Student*));
//records is an array of double pointers
//Each element in this array points to starting address of one section of students

//Memory block is dynamically allocated for each section
for(int sec = 0; sec<sections; sec++)
{
records[sec] = (Student*)alloc_mem(students*sizeof(Student));
}

//To store the average marks of each section
float *avgsections = (float*)alloc_mem(sections * sizeof(float));

//Collect marks for students per each section
while(secIndex < sections)
{
printf("\nSection %d\n",secIndex+1);
while(studIndex < students)
{
printf("Enter Student name: ");
scanf("%s",records[secIndex][studIndex].name);
printf("Enter Student marks: ");
scanf("%f", &(records[secIndex][studIndex].marks));
studIndex++;
}
studIndex = 0;
secIndex++;
}
```

```
//Calculate avg marks sectionwise
for(secIndex = 0;secIndex<sections;secIndex++)
{
studIndex = 0;
while(studIndex < students)
{
avgsections[secIndex] += records[secIndex][studIndex].marks;
studIndex++;
}
avgsections[secIndex] = (float)avgsections[secIndex]/students;
avgALLsections += avgsections[secIndex];
}

printf("--------------------------------------------------------------------------------\n");
for(int i =0;i<sections;i++)
{
printf("Avg marks for section %d is: %.02f\n", i+1, avgsections[i]);
}
printf("Avg marks across all sections %.02f\n",(float)avgALLsections/sections);

//Releasing the dynamically allocated memory
free_mem(avgsections);
for(int i =0;i<sections;i++)
{
free_mem(records[i]);
}
free_mem(records);
}
OUTPUT
Enter the number of sections:2
Enter the number of students per section:2

Section 1
Enter Student name: John
Enter Student marks: 67.67
Enter Student name: Smith
Enter Student marks: 86.77

Section 2
Enter Student name: Gabriel
Enter Student marks: 89
Enter Student name: Richards
Enter Student marks: 76.56
--------------------------------------------------------------------------------
Avg marks for section 1 is: 77.22
Avg marks for section 2 is: 82.78
```

```
Avg marks across all sections 80.00
```

6.4 realloc

realloc changes the size of memory block pointed to by the beginning address of memory block passed to it by the number of bytes supplied to it.

- The new memory is allocated just after the end of the original memory or the memory contents of original memory are copied elsewhere where there is enough memory available to accomodate the new size.(This is an expensive computation)

Syntax:

```
datatype *ptr = (datatype*)realloc(*ptr, new_size);
```

7 File I/O

A file represents a sequence of bytes.

7.1 Opening Files

The function used to create a new file or open an existing file is **fopen()**. This call will initialize an object of the type FILE, which contains the necessary information to control the stream.

```
FILE *fopen(const char * filename, const char * mode);
```

Sr.No.	Mode & Description
1	**r** Opens an existing text file for reading purpose.
2	**w** Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
3	**a** Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
4	**r+** Opens a text file for both reading and writing.
5	**w+** Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
6	**a+** Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Incase of binary files the access modes used are as follows
{ rb, wb, ab, rb+, r+b, wb+, w+b, ab+, a+b }

7.2 Reopening a File

A function *freopen* is used to associate a new filename with the given open stream at the same time closes the old file opened in the stream.

Syntax:

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

An example for the usage of the above function is given below

```
int main () {
   FILE *fp;

   printf("This text is redirected to stdout\n");

   fp = freopen("file.txt", "w+", stdout);

   printf("This text is redirected to file.txt\n");

   fclose(fp);
 }
```

7.3 Closing a File

```
int fclose(FILE *fp);
```

- fclose returns zero on success, or EOF if there is an error in closing the file.
- All the buffers associated with the stream are disassociated and flushed
- The unwritten content in the output buffer is written before flushing it. The unread content in the input buffer is discarded.

7.4 Check End of File

The function used to check if we have reached the end of file is $feof$.

Syntax:

```
int feof(FILE *fp)
```

This function returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.

7.5 Writing to a File

- To write a character stored in a variable c to the output stream referenced by fp.

```
int fputc( int c, FILE *fp);
```

  Returns zero on success, EOF on failure

- To write a string to the output stream referenced by fp.

```
int fputs( const char *s, FILE *fp);
```

  Returns a non-negative value on success, EOF on failure

- To write a formatted string to the output stream refernenced by fp

```
fprintf(FILE *fp, const char *format, ..);
```

7.6 Reading from a File

- To read a character from the input file referenced by fp.

```
int fgetc( FILE *fp);
```

  The return value is the character read, in case of any error EOF is returned

- To read (n-1)characters from the input file referenced by fp.

```
char *fgets(char *buf, int n, FILE *fp);
```

  - It copies the read string into the buffer **buf**, appending a null character to terminate the string.
  - If a newline character or EOf is encountered before reading (n-1) characters it returns the characters read up to that point including the newline character.

- To read strings until the first space character that is encountered.

```
int fscanf( FILE *fp, const char *format,..)
```

```
int main()
{
 FILE *fp;
 char buf[255];

 fp = fopen("FileIObasics.txt", "w");
 fprintf(fp, "This was written using fprintf\n");
 fputs("This was written using fputs\n",fp);
 fclose(fp);
```

```
 fp = fopen("FileIObasics.txt", "a+");
 fscanf(fp, "%s", buf);
 printf("Read using fscanf: %s\n",buf);
 fgets(buf, 255, fp);
 printf("Read using fgets: %s",buf);
 fprintf(fp,"With one call to fprintf and fgets the file was read till here\n");
 return 0;
}

OUTPUT
Read using fprintf: This
Read using fgets:  was written using fprintf

fileIObasics.txt
This was written using fprintf
This was written using fputs
With one call to fprintf and fgets the file was read till here
```

Since "a+" was the mode used to open the file for the second time, though only the first line of contents were read. When we tried to write something to the file, automatically the file pointer pointed to the last character and started appending the content to be written from there.
Now when the same is tried with just one change, that is using the mode "r+" when the file is opened for the second time. The file contents were as follows

```
fileIObasics.txt
This was written using fprintf
With one call to fprintf and fgets the file was read till here
```

Since only one line was read. And then we started writing to the file, the second line that was previously present in the file got overwritten.

Trying the same by opening the file in "w+" mode instead of "r+".

```
fileIObasics.txt
With one call to fprintf and fgets the file was read till here
```

This is because opening the file in "w+" mode, will first erase all the content and then start writing to it.

Read after write in a+ mode

```
int main()
{
 FILE *fp;
 char buf[255];

 fp = fopen("FileIObasics.txt", "w");
```

```
 fprintf(fp, "This was written using fprintf\n");
 fputs("This was written using fputs\n",fp);
 fclose(fp);

 fp = fopen("FileIObasics.txt", "a+");
 fgets(buf, 255, fp);
 printf("Read using fgets: %s",buf);
 fprintf(fp,"With one call to fgets the file was read till here\n");
 fgets(buf, 255, fp);
 printf("Content read after writing in a+ mode: %s",buf);
 return 0;
}

OUTPUT
Read using fgets: This was written using fprintf
Content read after writing in a+ mode: This was written using fprintf
```

7.7 Reading from a binary file

To read data from the given stream into the array pointed to, by ptr.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

- ptr This is the pointer to a block of memory with a minimum size of size*nmemb bytes.
- size This is the size in bytes of each element to be read.
- nmemb This is the number of elements, each one with a size of size bytes.
- stream This is the pointer to a FILE object that specifies an input stream.

The total number of elements successfully read are returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.

7.8 Writing to a binary file

To write data from the array pointed to, by ptr to the given stream.

Syntax:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

- ptr This is the pointer to the array of elements to be written.
- size This is the size in bytes of each element to be written.

- nmemb This is the number of elements, each one with a size of size bytes.
- stream This is the pointer to a FILE object that specifies an output stream.

This function returns the total number of elements successfully returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, it will show an error.

7.9 Important functions involving File pointers

7.9.1 fseek()

This function is used to set the file pointer to the specified offset.

```
int fseek(FILE *fp, long int offset, int whence);
```

- **offset**: Number of bytes to be offset
- **whence**: Specifies from where the offset is added.
  - **SEEK_SET**: The beginning of the file
  - **SEEK_CUR**: The current position of the file pointer
  - **SEEK_END**: End of file

7.9.2 ftell()

This functions returns the current position of the file pointer.

Syntax:

```
long int ftell(FILE *stream)
```

This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

7.9.3 rewind()

Sets the file pointer to the beginning of the file.

```
rewind(FILE *fp);
```

7.9.4 fgetpos()

Gets the current position of the file pointer and writes it to pos.

Syntax:

```
int fgetpos(FILE *fp, fpos_t *pos);
```

This function returns zero on success, else non-zero value in case of an error.

7.9.5 fsetpos()

Sets the position of the file pointer to the given position. The argument pos is a position given by the function fgetpos.

Syntax:

```
int fsetpos(FILE *fp, const fpos_t *pos)
```

This function returns zero value if successful, or else it returns a non-zero value and sets the global variable errno to a positive value, which can be interpreted with perror.

8 Advance Concepts

8.1 Preprocessor

A text substitution tool that instructs the compiler to do the required the pre-processing before the actual compilation.

Pre-processor Directive	Description
#define	Substitutes a pre-processor macro
#include	Inserts a particular header file from another file
#undef	Undefines a pre-processor macro
#ifdef	Returns True if this macro is defined
#ifndef	Returns True if this macro is not defined
#if	Tests if a compile time condition is True.
#else	The alternative of #if
#elif	#else and #if in one statement.
#endif	Ends pre-processor conditional
#error	Prints error message on *stderr*
#pragma	Issues special commands to the compiler.

8.2 Macros

A macro is a segment of code which is replaced by the value of macro. Macro is defined by define directive. There are two types of macros:

- Object-like Macro
- Function-like Macro

One can also define a macro without a value or a piece of code attached to it.

8.2.1 Object-like Macro

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants.
Ex:

```
\#define PI 3.14
```

8.2.2 Function-like Macro

The function-like macro looks like function call.
Ex:

```
\#define MIN(a,b) ((a)<(b)?(a):(b))
```

MIN is the Macro name.

8.2.3 Pre-defined Macros

Macro	Description
DATE	Represents the current date in "MMM DD YYYY" format.
TIME	Represents the current time in "HH:MM:SS" format.
FILE	Represents the current file name
LINE	Represents the current line number
STDC	It is defined as 1 when compiler complies with the ANSI standard.

8.2.4 Passing compile time parameter to define a macro

To define a macro in a program during compilation the following needs to be used

```
gcc test.c -DMACRO_DEFN -o test
```

Using the above to the compile the code written in *test.c*,

- The name of the executable file created is test
  To run the executable file use

  ```
  ./test
  ```

- MACRO_DEFN is the macro defined in the program *test.c*

8.3 Variable number of Arguments to functions

C allows the programmer to define a function that takes in variable number of arguments.

- In the function definition for such a function, the first argument passed must be an integer which specifies the number of arguments being passed.

- The last argument passed must be **...**
- **stdarg.h** library is required to use this functionality
- Create a va_list type variable in the function definition
- Use int parameter and va_start macro to initialize the va_list variable to an argument list
- Use va_arg macro and va_list variable to access each item in argument list.
- Use a macro va_end to clean up the memory assigned to va_list variable.

```
double average(int num,...) {
 va_list valist;
 double sum = 0.0;
 int i;

 /* initialize valist for num number of arguments */
 va_start(valist, num);

 /* access all the arguments assigned to valist */
 for (i = 0; i < num; i++) {
  sum += va_arg(valist, int);
 }

 /* clean memory reserved for valist */
 va_end(valist);

 return sum/num;
}

int main() {
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}

OUTPUT
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

8.4 Command Line Arguments

When a C program is invoked two command line arguments can be supplied. Syntax:

```
return_type main(int argc, void *argv[])
```

- *argc* denotes the number of arguments supplied to the program

- $*argv[]$ is an array of string pointers

Consider the following exmaple code $CLA.c$

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, void *argv[])
{
if(argc < 3)
{
printf("Please specify a range to calculate a sum of even numbers");
exit(0);
}
int start = atoi(argv[1]);
int end = atoi(argv[2]);
int sum = 0;
for(int i = start;i<= end; i++)
{
sum += (i%2) == 0? i:0;
}
printf("The sum of even numbers in the given range is: %d\n",sum);
}


In Terminal:
gcc CLA.c
./a.out 1 10
The sum of even numbers in the given range is: 30
```

8.5 Makefile

Makefile is a tool to simplify or to organize code for compilation. When the source code is spread across multiple files, compiling each of them would be a cumbersome and a repetitive task. Even if there is a small change in one of the files, we have to go through this cumbersome procedure of compiling all files again.

- Make process keeps track of changed files in the source code.
- Make compiles only changed source code files keeping other precompiled code intact to save on total compilation time
- With Makefile you can define perform conditional compilation for example *OS dependent compilation*

Lets look at an example
All the files needs to be in the same directory

```
/* main.c */
#include<stdio.h>
#include "operations.h"

int main()
{
int a=0,b=0;
printf("Enter two numbers:");
scanf("%d %d",&a,&b);

printf("Sum of the numbers: %d\n",ADD(a,b));
printf("Product of the numbers: %d\n",PROD(a,b));
}

/* operations.h */
int ADD(int,int);
int PROD(int,int);

/* ADD.c */
#include<stdio.h>
#include "operations.h"

int ADD(int a, int b)
{
return (a+b);
}

/* PROD.c */
#include<stdio.h>
#include "operations.h"

int PROD(int a, int b)
{
return (a*b);
}

/* Makefile */
all:
gcc main.c ADD.c PROD.c -o main

OUTPUT
make
  gcc main.c ADD.c PROD.c -o main
./main
Enter two numbers:2 3
Sum of the numbers: 5
```

```
Product of the numbers: 6
```

As it can be seen in the above example, the files *main.c*, *ADD.c* and *PROD.c* are compiled using a single command. The definitions for the functions used in *main.c* are included in a header file in the local directory named *operations.h*. **all** used in the Makefile is the name of the target, we can have many such targets in the make file to compile different combinations of files.

Further variables can be used in the make file to generalise it.

```
/* Makefile */
CC = gcc
TARGET = main
all:
    $(CC) main.c ADD.c PROD.c -o $(TARGET)
clean:
    rm $(TARGET)
```

In the above make files there are two targets.

```
In terminal

make all
  gcc main.c ADD.c PROD.c -o main
./main
Enter two numbers:2 3
Sum of the numbers: 5
Product of the numbers: 6

make clean
rm main
./main
bash: ./main: No such file or directory
```

The target *all* is used to create the object file.
The targte *clean* is used to delete the created object file *main.o*

Compiling only those files whose source code has been modified

```
/* Makefile */
CC = gcc
TARGET = main
all:    main.o ADD.o PROD.o
    $(CC) main.c ADD.c PROD.c -o $(TARGET)
clean:
    rm $(TARGET)
/* ADD.c */
int ADD(int a, int b)
```

```
{
return (b+a);
}
```

The makefile has been modified, to create object files for each of the source file. All the files added in the line where the target is defined inform the compiler about the dependencies of the target. So if there is a change in any of the dependencies, only those are compiled again.

After modifying the makefile, the only source file which has been updated is *ADD.c* where in the return statement instead of $a + b$, $b + a$ is used. Now running the make command we get the following

```
In terminal

make all
gcc    -c -o ADD.o ADD.c
gcc main.c ADD.c PROD.c
```

It can be seen that only *ADD.c* file is compiled again and not all the files.

8.5.1 Special MACROs used in makefile

- $@ is the name of the file to be made.
- $? is the names of the changed dependents.
- $< the name of the related file that caused the action.

The make file can be updated using these

```
CC = gcc
main: main.o ADD.o PROD.o
    $(CC) main.c ADD.c PROD.c -o $@
```

In the above makefile $@ gets replaced with *main*

8.6 Compiler Switches

GNU Compiler Collection supports variety of switches that direct the compiler about how it should compile the program source code.

Compiler flags	Description
-Wall	Turn on all warnings
-Werror	Convert all warnings to errors
-D¡macro¿	Supply macros to the code being build
-Wshadow	warn whenever local variable shadows another local variable, parameter or global variable or whnever a built-in function is shadowed
-save-temps	Output all stages of compilation
-c	Turn linking off, the compier only compiles the program and produces compiled object files
-E	Produce only pre-processed output the destination can also be specified gcc -E main.c ¿ main.i
-S	Produce only assembly code the destination can also be specified gcc -S main.c ¿ main.s
-o	to specify the name of the object file
-l	link with shared libraries gcc main.c -luserlib links main.c with the library userlib

GCC also allows certain optimizations to be performed on the code.

gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

8.7 Libraries

- Library is an archive of object files
- Library introduces code modularity
- A library is just like an executable

There are two types of libraries

1. Static Library
   A static library is simply an archive file that contains one or more object files that are combined with the programs object file during the linking stage to produce the executable file.
2. Shared Library
   It is similar to static library but the only difference is that these are linked to the program when they are executed.
   - If multiple files use the same library, copies of the library are not created, a single copy loaded onto RAM is used by all the programs
   - As linking happens during execution time, the code takes more time to run.

8.7.1 Creating static library

- Create a C file that contains the definitions to all functions in your library. (similar to having ADD.c and PROD.c in a single file operations.c)
- Create a header file for the library (operations.h file used in the makefile example)
- Compile library files

```
gcc -c liboperations.c -o liboperations.o
```

- Create static library

```
ar rcs liboperations.a liboperations.o
```

- Create a C file that has the main function which calls functions from the created library
- Compile this program

```
gcc -c main.c -o main.o
```

- Link the static library to the compiled program

```
gcc -o main main.o -L. -loperations
```

- Run the driver program

```
./main
```

9 Debugging

GDB, short for GNU Debugger, is the most popular debugger for UNIX systems to debug C and C++ programs.

To let GDB be able to read all that information line by line from the symbol table, we need to compile it a bit differently.

```
gcc -g operations.c -o operations
```

-g flag must be used to enable debugging using gdb.

```
gdb operations
```

9.1 gdb - Commands

Commands	Action performed
print or p	print value of expression
breakpoint or b	Set a breakpoint by specifying line number, function_name, file_name:line_number, file_name:function_name or address of a function
file_name	function_name or address of a function
info breakpoints or i b	Lists breakpoints that are currently set
info locals or i l	Lists all variables in the current function stack
run or r	To run the program being debugged
next [N] or n [N]:	Step over current statement and runs all the statement till N reaches
step [N] or s [N]:	Step into the statement after N statements if it is a function
continue [N] or c	Continue normal execution
set args	Set input arguments for the program being debugged
delete [N]	Delete a breakpoint identified by break point number N
delete breakpoints	Delete all breakpoints

Commands	Action performed
enable [N]	Enable a breakpoint identified by number N
disable [N]	Disable a breakpoint identified by number N
info registers or i r	Lists integer registers and their contents for selected stack frame
info registers-all	Lists all registers including platform specific registers and their contents for selected stack frame
backtrace or bt	Shows backtrace of all stack frames, the frames are listed by frame numbers
backtrace full	Shows backtrace of all stack frames include local variables
frame [N]	Shows frame details identified by frame number
info args or i ar	: Print the arguments supplied to the function of the current stack frame
list	Show ten source lines by default
set variable <variable_name> <value>	Used to manually set / override value of a variable in the current scope
call <function_name>	Call a function in the program
disassemble or disas	Disassemble a specified section of memory, defaults to the function surrounding the current instruction in the selected stack frame

Let ADD and PROD be functions in the file operations. To set break points at these functions

```
break operations.c:ADD
break operations.c:PROD
```

To view source code from a particular line in file

```
list operations.c:2
```

The above prints the contents of *operations.c* from line 2.

- Debugger provides a function called *finish* which terminates from the present function and returns to the caller and reports the return value of the function.

9.2 Removing debug information from the generated binary file

Create the executable for debugging

```
gcc -g operations.c -o operations
```

Make a copy of the executable

```
cp operations operations.debug
```

To remove the debug information from the original file

```
strip -g operations
```

If the system crashes while executing operations file. Then a file called *core* is created which contains the backtrace of the code. This along with *operations.debug* is used together to debug *coredump*.

10 Parallel processing

Process

A process is the execution of a program that allows you to perform the appropriate actions specified in a program.

The other processes created by the main process are called child process.

Thread

Thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time. It is a unit of execution in concurrent programming. A thread is lightweight and can be managed independently by a scheduler. It helps you to improve the application performance using parallelism.

Threads/ Processes are the mechanism by which you can run multiple code segments at a time, threads appear to run concurrently; the kernel schedules them asynchronously, interrupting each thread from time to time to give others chance to execute.

Identifying a thread

- Each thread identified by an ID, which is known as Thread ID
- A Thread ID is unique in the current process, while a Process ID is unique across the system.
- Thread ID is represented by type pthread_t
- The corresponding header file to be included is pthread.h

10.1 Creating a thread

pthread_create is the function of pthread.h header file, which is used to create a thread.

Syntax:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

- pthread_t *thread
  Pointer to a pthread_t variable which is used to store thread id of new created thread.
- const pthread_attr_t *attr
  Pointer to a thread attribute object which is used to set thread attributes, NULL can be used to create a thread with default arguments

- void *(*start_routine) (void *)
  Pointer to a thread function; this function contains the code segment which is executed by the thread.

- void *arg
  It is the thread functions argument of the type void*, you can pass what is necessary for the function using this parameter.

- If thread created successfully, return value will be 0 (Zero) otherwise pthread_create will return an error number of type integer.

Example:

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

/*thread function definition*/
void* threadFunction(void* args)
{
    static int i = 0;
    while(1)
    {
       printf("%d\n",i);
       i++;
    }
}
int main()
{
    /*creating thread id*/
    pthread_t id;
    int ret;
    static int j = 0;
    /*creating thread*/
    ret=pthread_create(&id,NULL,&threadFunction,NULL);
    if(ret==0){
        printf("Thread created successfully.\n");
    }
    else{
        printf("Thread not created.\n");
        return 0; /*return from main*/
    }

    while(1)
    {
        printf("In main: %d\n",j);
        j++;
    }
```

```
    return 0;
}
OUTPUT
gcc thread.c -lpthread
./a.out
Thread created successfully.
In main: 0
0
In main: 1
1
In main: 2
2
In main: 3
3
In main: 4
4
In main: 5
5
In main: 6
6
In main: 7
7
In main: 8
8
In main: 9
9
10
11
12
13
14
15
16
17
18
19
In main: 10
20
21
In main: 11
22
In main: 12
23
In main: 13
24
```

From the above example we can clearly see the difference between a normal function and a thread. In the above program if a normal function was used then the control of execution would have never entered the infinite while loop inside the function as there is no function call. But since we have created a thread the above program now has two active threads (main, threadfunction), so during execution certain timeslots are given to each of them, and they are executed parallely.

10.2 Creating a process

To create a process we can use a function called $fork()$. This function is available in the *unistd.h* header file. Whenever the compiler comes across this function a child process is created.
Example:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
int id = fork();
printf("Hello World from id: %d\n",id);
return 0;
}
OUTPUT
Hello World from id: 23352
Hello World from id: 0
```

When the fork function is called a child process is created. As fork is called before printing the statement, there are two processes (main. child) that execute the printf statement. So in the output it is printed twice.

On successsful forking, the PID of the child process is returned in the parent, and 0 is returned in the child. The process ID should always be greater than zero. Hence the value of *id* can never be 0 in the parent process. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

If the $fork()$ function is called n times then 2^n - 1 child processes are created. The ID can be used inside an if condition to create m number of processes where $(m \neq 2^n)$

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
int id = fork();
int id1 = 10;
```

```
if(id!= 0)
id1 = fork();
if(id1 == 10)
printf("Hello World from id: %d\n",id);
else
printf("HELLO World from id: %d\n",id1);
return 0;
}
OUTPUT
HELLO World from id: 23467
HELLO World from id: 0
Hello World from id: 0
```

It can be seen that the id of all child process is 0. Both the main and child2 processes execute the printf in else statement. Child1 executes the print statement in if block.

10.2.1 Orphan process

When the execution of the parent process gets completed before the execution of the child process, then the child process is termed as an **orphan process**. The main process then adopts this child process.
Ex:

```
    #include<stdio.h>
#include<unistd.h>
/* For orphan child process systemd becomes the parent in linux OS*/

int main()
{
    int i;
    printf("Before fork\n");
    i = fork();
    if(i == 0){
        sleep(10);
        printf("In child process, child process ID = %d\n", getpid());
        printf("In child process, parent process ID = %d\n", getppid());
    }
    else{
        printf("In parent process, child process ID = %d\n", i);
        printf("In child process, parent process ID = %d\n", getpid());
    }
    printf("After fork\n");
}

OUTPUT
krishna@krishna:~/Desktop$ ./a.out
```

```
Before fork
In parent process, child process ID = 6223
In child process, parent process ID = 6222
After fork
krishna@krishna:~/Desktop$ In child process, child process ID = 6223
In child process, parent process ID = 1491
After fork
```

In the above example the parent process (else part) gets executed first as the child process (if part) is asleep for 10 seconds. So after the execution of the parent process, the systemd process adopts the child process.

To avoid this the *wait*() function is used.

10.2.2 Waiting for processes to finish their work

When we have multiple processes the order in which they are executed is chaotic.

```
int id = fork();
int n;
if(id == 0)
n = 1;
else
n = 6;
int i;
for(i=n;i<n+5;i++)
{
printf("%d\n",i);
}
OUTPUT
6
7
8
9
10
1
2
3
4
5

OUTPUT
6
1
7
2
8
```

```
3
9
4
10
5
```

The order of the output keeps changing. To avoid such a mess we can use *wait*() function.

wait() stops the execution until a child process has finished executing.

```
int id = fork();
int n;
if(id == 0)
n = 1;
else
n = 6;
if(id != 0)
wait();
int i;
for(i=n;i<n+5;i++)
printf("%d\n",i);

OUTPUT
1
2
3
4
5
6
7
8
9
10
```

By using *wait*() we ensure that the child process finishes its execution first. And hence the order at the output is no longer chaotic. *wait*() function only waits for one of the child to finish execution, if there are multiple (n) child process for a parent process then *wait*() function should be called n times.

10.2.3 Process IDs

Every process is recognised by its unique ID.
getpid() is the function used to the get the ID of a particular process. This function is available in the *sys/wait.h* header file.
NOTE: Every process except the main process will have a parent.

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

```
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{
int id = fork();
printf("Current ID: %d, parent ID: %d\n", getpid(), getppid());
}
OUTPUT
Current ID: 23734, parent ID: 3128
Current ID: 23735, parent ID: 23734
```

The ID for the process that creates the child process is 23734. The ID of the child process is 23735. However the ID of some global main process which created the parent process is 3128.

In order to avoid orphan processes, *wait* function can be used. On success, *wait*() returns the process ID of the terminated child; If there are no child processes to wait for then -1 is returned.

```
int id = fork();
printf("Current ID: %d, parent ID: %d\n", getpid(), getppid());
int res = wait(NULL);
if(res == -1)
printf("No children to wait for\n");
else
printf("%d finished execution\n",res);

OUTPUT
Current ID: 23825, parent ID: 3128
Current ID: 23826, parent ID: 23825
No children to wait for
23826 finished execution
```

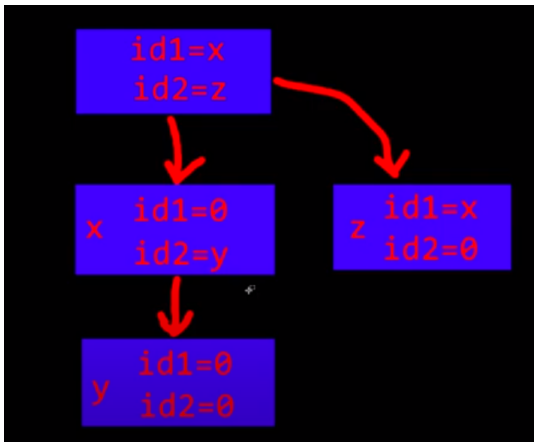
Though wait function is called from both process, only the execution of the parent process is halted.

When called from the child process *wait*() returns -1. When called from the parent process it returns the ID of child because of which the parent process has been halted.

When a parent has more than one child, *wait*() halt the parent process only until one of its child has finished executing. Consider a program where fork is called 2 times. Hence creating two childs for the parent process.

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

```
#include<unistd.h>
#include<sys/wait.h>
#include<errno.h>
int main()
{
    int id1 = fork();
    int id2 = fork();
    if(id1 == 0){
        if(id2 == 0)
            printf("We are in process y\n");
        else
            printf("We are in process x\n");
    }
    else{
        if(id2 == 0)
            printf("We are in process z\n");
        else
            printf("We are in parent process \n");
    }
    while(wait(NULL) != -1 || errno != ECHILD){
        printf("Waited for child to finish\n");
    }
}
```



The parent process is the top left blue rectangle (with two children x,z)

OUTPUT
We are in parent process
We are in process z
We are in process x
We are in process y

Waited for child to finish
Waited for child to finish
Waited for child to finish

As there were 3 child process it can be seen that the while loop gets executed three times.

10.3 Communicating between process

pipe() is a function that helps in establishing communication between process. Basically a *pipe* is like a in-memory file, it only has a buffer that is saved in memory and it is possible to write and read from it.

To declare a *pipe* we need an array of two integers and these two integers are the file descriptors for the pipe. File descriptors are like the keys to either read/write to the pipe.

- fd[0] is the read end
- fd[1] is the write end

Once the pipe has been declared, then we can call the function $fork()$ to create a child process. By doing it in this order the file descriptors declared get copied over to the child process as well. But file descriptors in both the processes are independent. Closing a file descriptor in one process does not close it in the other process as well. So the two file descriptors get copied over and are inherited as well.

The following example shows how to achieve communication between parent process and a child process. A number is sent from the child process and the parent process receives this number and prints the square of the number.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
#include<errno.h>
int main()
{
    int fd[2];
    if(pipe(fd) == -1)
    {
        printf("An error occured with opening the pipe\n");
    }
    int id = fork();
    if(id == -1)
```

```
        printf("An error occured while creating a child process");
    if(id == 0)
    {
        //In child process

        close(fd[0]);
        //child process is only writing, so no need to keep read end open

        int x;
        printf("Enter a number: ");
        scanf("%d",&x);

        //Write the data stored in &x to the parent process using the write end of the pipe

        if(write(fd[1], &x, sizeof(int)) == -1)
        {
            printf("An error occured while writing to the pipe\n");
        }

        close(fd[1]);
        //As there is nothing else to be written close the pipe
    }
    else
    {
        //In parent process

        close(fd[1]);
        //parent process is only reading, so no need to keep write end open

        int y;

        //Read data into the variable y
        if(read(fd[0], &y, sizeof(int)) == -1)
        {
            printf("An error occured while reading to the pipe\n");
        }


        close(fd[0]);
        //As there is nothing else to write

        printf("Number received from child process is: %d\n",y);
        printf("It's square is: %d\n",y*y);
    }
}
OUTPUT
```

```
Enter a number: 5
Number received from child process is: 5
It's square is: 25
```

10.4 Communication between processes of different hierarchy

Using the *pipe*() function it is possible to only set up the communication between a parent and its child process. However to establish communication between two different processes *pipe* cannot be used. For this purpose **Named Pipes** are used.

Named pipes are special kind of file on local storage.

10.4.1 Creating a named pipe

The required header files are *sys/stat.h* and *sys/types.h* The function $mkfifo$ is used to create a named pipe.

Syntax:

```
int mkfifo(const char *pathname, mode_t mode);
```

- *pathname* is the desired location of the file and its name
- *mode* specifies the permissions given to the file.
  0777 is the mode to be used which gives read and write permissions to all users.

Alternatively you can directly create the FIFO file from the terminal using the command

```
mkfifo <FIFO_filename>
```

10.4.2 Opening a named pipe

The required header file is $fcntl.h$.

Syntax:

```
int fd = open(const char *pathname, int flags, mode_t mode);
```

- The argument flags must include one of the following access modes:
  - O_RDONLY (Read only)
  - O_WRONLY (Write only)
  - O_RDWR (Read/Write)

NOTE: Opening the read or write end (not in O_RDWR mode) of a FIFO blocks until the other end is also opened (by another process or thread). So if both the ends are not open the program will just hang there.

Lets look at an example where an array of numbers are sent from one program to another through a FIFO file. The second program calculates the sum and returns it to the first file.

STEP 1: Create a FIFO file

```
Open a terminal and type
mkfifo FIFOSUM
```

STEP 2 Create a file that sends an array of numbers to FIFOSUM

```
/* FIFO1.c */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<errno.h>

int main()
{
int a[] = {11,13,17,19,23};
int sum = 0;
//Opening the FIFO file FIFOSUM
int fd = open("FIFOSUM", O_WRONLY);

if(fd == -1)
printf("Error in opening FIFOSUM\n");

//This part of the code is not reached if the reading end of the FIFOSUM is not opened
if(write(fd,a,sizeof(a)) == -1)
printf("Could not write to FIFOSUM\n");
printf("The contents of the array have been written\n");

//close FIFOSUM
close(fd);

//Reopen the file to read the sum
fd = open("FIFOSUM", O_RDONLY);
read(fd, &sum, sizeof(int));
close(fd);
```

```
printf("The received sum is: %d\n",sum);
}
```

STEP 3: Create a file that reads the numbers from FIFOSUM and calculates it's sum.

```
/* FIFO2.c */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<errno.h>

int main()
{
int a[5];
int sum = 0;
int fd = open("FIFOSUM", O_RDONLY);

if(fd == -1)
printf("Error in opening FIFOSUM\n");

if(read(fd, a, sizeof(a)) == -1)
printf("Could not read from FIFOSUM\n");
printf("Received the contents of the array\n");

close(fd);
for(int i=0;i<5;i++)
{
sum+=a[i];
}
fd = open("FIFOSUM", O_WRONLY);
write(fd, &sum, sizeof(sum));
close(fd);
printf("The sum of numbers received is sent back\n");
}
```

STEP 4: Open two terminals and create object files for both the above source codes.

```
Terminal 1
gcc FIFO1.c -o FIFO1

Terminal 2
gcc FIFO2.c -o FIFO2
```

STEP 5: Execute the generated object files.

```
Terminal 1
./FIFO1

Terminal 2
./FIFO2
```

STEP 6: Once both the object files are executed, the following will be seen.

```
Terminal 1
The contents of the array have been written
The received sum is: 83

Terminal 2
Received the contents of the array
The sum of numbers received is sent back
```

10.5 Two way communication using pipes

To achieve this both the ends of the pipe must be open, but since the file descriptors are inherited, chances are pretty high that the same pipe that writes immediately reads this content. Whereas the other process keeps waiting forever to read the data that was sent. To avoid such a scenario use multiple pipes.

- pipe1 Parent → Child
  Parent uses this pipe to only write data
  Child uses this pipe to only read data
- pipe2 Child → Parent
  Parent uses this pipe to only read data
  Child uses this pipe to only write data

Lets look at an example, where parent sends a number, child returns the square of the number received, to the parent

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

int main()
{
    int p1[2];  // P -> C
    int p2[2]; // C -> P
    if(pipe(p1) == -1)
    {
        printf("An error occured with opening the pipe\n");
    }
```

```
if(pipe(p2) == -1)
{
    printf("An error occured with opening the pipe\n");
}
int id = fork();
if(id == -1)
    printf("An error occured while creating a child process");
if(id == 0)
{
    //child process
    close(p1[1]);
    close(p2[0]);
    int x;

    if(read(p1[0],&x,sizeof(x)) == -1)
        printf("Could not read data sent from parent\n");
    printf("Received %d from parent\n",x);

    x*=4;
    if(write(p2[1],&x,sizeof(x)) == -1)
        printf("Could not write back to parent\n");

    printf("Wrote %d to parent\n",x);

    close(p1[0]);
    close(p2[1]);

}
else
{
    //parent process
    close(p1[0]);
    close(p2[1]);

    srand(time(NULL));
    int y = rand() % 10;

    if(write(p1[1], &y,sizeof(y)) == -1)
        printf("Could not write to child\n");

    printf("Wrote %d to child\n",y);

    if(read(p2[0],&y,sizeof(y)) == -1)
        printf("Could not read data sent from child\n");

    printf("Result is: %d\n",y);
```

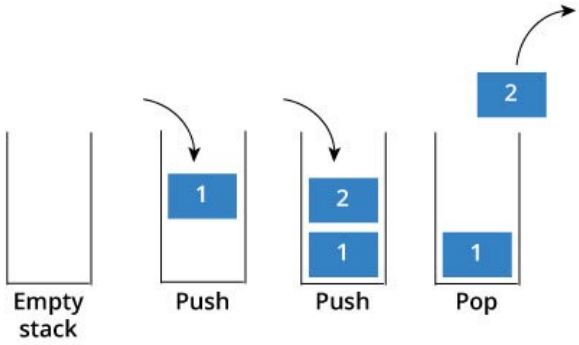
```
        close(p1[1]);
        close(p2[0]);
        wait(NULL);
    }
}
OUTPUT
Wrote 9 to child
Received 9 from parent
Wrote 36 to parent
Result is: 36
```

11 Abstract Datatypes

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

11.1 Stack

- Linear datastructure
- Last In First Out (LIFO)
- Inserting (PUSH) or deleting (POP) elements from the stack can happen only at one end (Usually termed as top of the stack)



11.1.1 Implementation

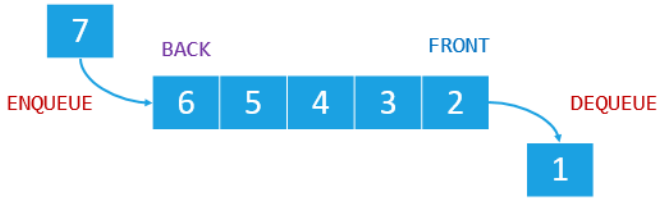
```
//A structure representing the stack
typedef struct Stack{
int top;
unsigned size;
int* array;
} STACK;

//Function to create a stack of given capacity
STACK* create_stack(unsigned capacity)
{
STACK* stack = (STACK*)malloc(sizeof(STACK));
stack->size = capacity;
stack->top = -1;
stack->array = (int*)calloc(stack->size, sizeof(int));
```

```
}

//Function to check if stack is full
int isFull(STACK* stack)
{
if(stack->top == stack->size - 1)
return 1;
return 0;
}

//Function to check if stack is empty
int isEmpty(STACK* stack)
{
if(stack->top == -1)
return 1;
return 0;
}

//Function to check how many slots are free
int getAvailable(STACK* stack)
{
if(stack->top == -1)
return stack->size;
else
return (stack->size - 1) - stack->top;
}

//Function to add an element to the stack
void push(STACK* stack, int element)
{
if(isFull(stack))
printf("Stack is full can't push more items\n");
else
{
stack->top = stack->top + 1;
stack->array[stack->top] = element;
printf("Pushed %d into stack\n",element);
}
}

//Function to remove element from the top of the stack
int pop(STACK* stack)
{
if(isEmpty(stack))
{
printf("stack is empty can't pop anymore items\n");
```

```
return -1;
}
else
{
int item = stack->array[stack->top];
stack->top = stack->top - 1;
return item;
}
}
```

11.2 Queue

- Linear datastructure
- First In First Out (FIFO)
- Inserting (ENQUEUE) and deleting (DEQUEUE) from the queue takes place at REAR and FRONT ends respectively.



11.2.1 Implementation

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

//A structure representing the queue
typedef struct Queue{
int front,rear,size;
unsigned capacity;
int* array;
} QUEUE;

//Function to create a queue of given capacity
QUEUE* create_queue(unsigned capacity)
{
QUEUE* queue = (QUEUE*)malloc(sizeof(QUEUE));
queue->capacity = capacity;
queue->front = queue->size = 0;
queue->rear = -1;
```

```
queue->array = (int*)calloc(queue->capacity, sizeof(int));
}

//Function to check if queue is full
int isFull(QUEUE* queue)
{
if(queue->size == queue->capacity)
return 1;
return 0;
}

//Function to check if queue is empty
int isEmpty(QUEUE* queue)
{
if(queue->size == 0)
return 1;
return 0;
}

//Function to check how many slots are free
int getAvailable(QUEUE* queue)
{
return (queue->capacity - queue->size);
}

//Function to add an element to the queue
void Enqueue(QUEUE* queue, int element)
{
if(isFull(queue))
printf("Queue is full, enqueue not possible\n");
else
{
queue->rear = queue->rear + 1;
queue->array[queue->rear] = element;
queue->size += 1;
printf("%d enqueued to queue\n",element);
}
}

//Function to remove an element from the queue
int Dequeue(QUEUE* queue)
{
if(isEmpty(queue))
{
printf("Queue is empty dequeue not possible\n");
return -1;
```

```
}
else
{
int item = queue->array[queue->front];
queue->front = queue->front + 1;
queue->size -= 1;
return item;
}
}

//Function to get the data pointed by front pointer of queue
int front(QUEUE* queue)
{
if(isEmpty(queue))
{
printf("Queue is empty\n");
return -1;
}
else
return queue->array[queue->front];
}

//Function to get the data pointed by rear pointer of queue
int rear(QUEUE* queue)
{
if(isEmpty(queue))
{
printf("Queue is empty\n");
return -1;
}
else
return queue->array[queue->rear];
}
```

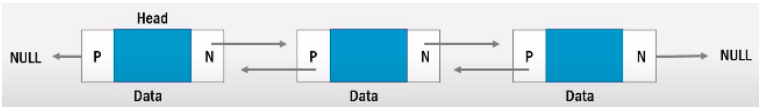
11.3 Linked List

- Linear datastructure
- Elements are not stored as arrays and are linked using pointers
- Each item of a linked list has two members data, address field (which points to next element in the linked list)
- First element of the linked list is called head or the root of the linked list

Types:

- Singly Linked list

- Doubly Linked list



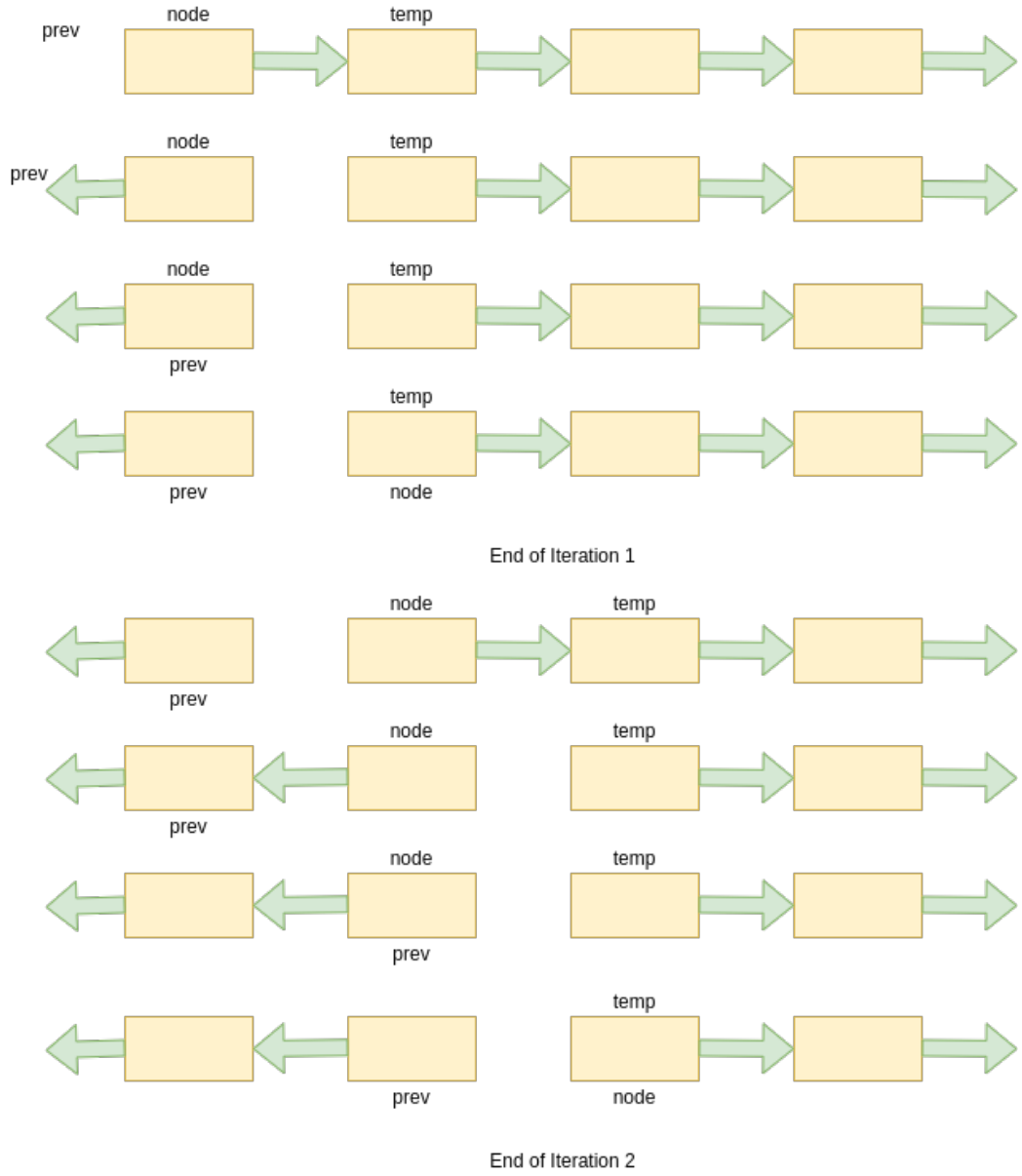
- Circular Linked list

11.3.1 Implementation of Singly Connected Linkedlist

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

typedef struct node
{
int data;
struct node* next;
}NODE;

//Initialize the linked list
void init(NODE** head)
{
*head = NULL;
}

void print_list(NODE* head)
{
NODE* temp;
for(temp = head;temp;temp=temp->next)
printf("%d ->",temp->data);
printf("\n");
}

//Adds a new node at the beginning of the linked list
```

```
NODE* add(NODE* node, int data)
{
NODE* temp = (NODE*)malloc(sizeof(NODE));
temp->data = data;

//This is the first node
if(node == NULL)
{
temp->next = NULL;
node = temp;
return node;
}

temp->next = node;
node = temp;
return node;
}

NODE* reverse(NODE* node)
{
NODE* temp;
NODE* prev = NULL;
while(node!= NULL)
{
temp = node->next;
node->next = prev;
prev = node;
node = temp;
}
return prev;
}

NODE* free_list(NODE *head)
{
NODE* temp = head;
NODE* toDelete = NULL;

while(temp!=NULL)
{
toDelete = temp;
temp = temp->next;
free(toDelete);
}
return NULL;
}
```

Visualisation of Reversing the linked list



It can be seen that at the end of every iteration, prev points to the index upto which the linkedlist is reversed properly.

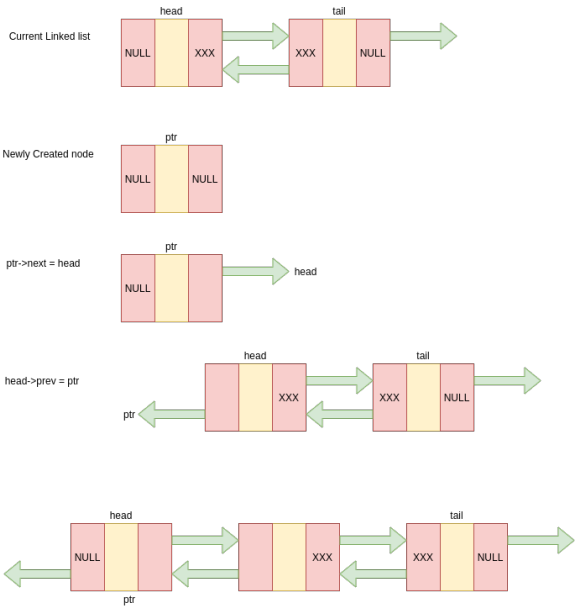
11.3.2 Implementing Doubly connected linked list

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

typedef struct node
{
int data;
struct node *next, *prev;
} node;

node *head, *tail;

node* make_node(int item)
{
node *ptr = (node*)malloc(sizeof(node));
ptr->data = item;
ptr->prev = ptr->next = NULL;
}

void insertAtHead(int item)
{
node *ptr = make_node(item);

if(NULL == head)
{
head = tail = ptr;
return;
}
ptr->next = head;
head->prev = ptr;
head = ptr;
}

void insertAtTail(int item)
{
node *ptr = make_node(item);

if(NULL == head)
{
head = tail = ptr;
return;
}
tail->next = ptr;
ptr->prev = tail;
tail = ptr;
```

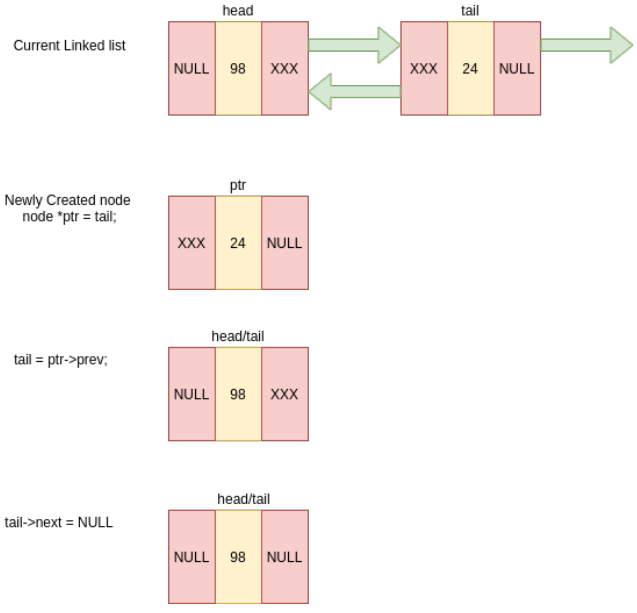
```
}

int deleteFromHead()
{
if(head == NULL)
{
printf("Empty list\n");
exit(1);
}
node *ptr = head;
int item = head->data;
head = ptr->next;

if(head == NULL)
tail = head;
else
head->prev = NULL;

free(ptr);
ptr = NULL;
return item;
}

int deleteFromTail()
{
if(tail == NULL)
{
printf("Empty list\n");
exit(1);
}

node *ptr = tail;
int item = tail->data;
tail = ptr->prev;

if(tail == NULL)
head = tail;
else
tail->next = NULL;

free(ptr);
ptr = NULL;
return item;
}

int deleteList()
```

```
{
if(head == NULL)
{
printf("Empty list\n");
exit(1);
}
node *ptr = NULL;
while(head!=NULL)
{
ptr = head->next;
printf("Deleting %d\n",head->data);
free(head);
head = ptr;
}
head = tail = NULL;
}

void print()
{
node *ptr = head;
while(ptr!=NULL)
{
printf(" %d ->",ptr->data);
ptr = ptr->next;
}
printf("\n");
}

void printReverse()
{
node *ptr = tail;
while(ptr!= NULL)
{
printf(" %d ->",ptr->data);
ptr = ptr->prev;
}
printf("\n");
}
```

Visualisation of inserting at Head



Visualisation of deleting at Tail



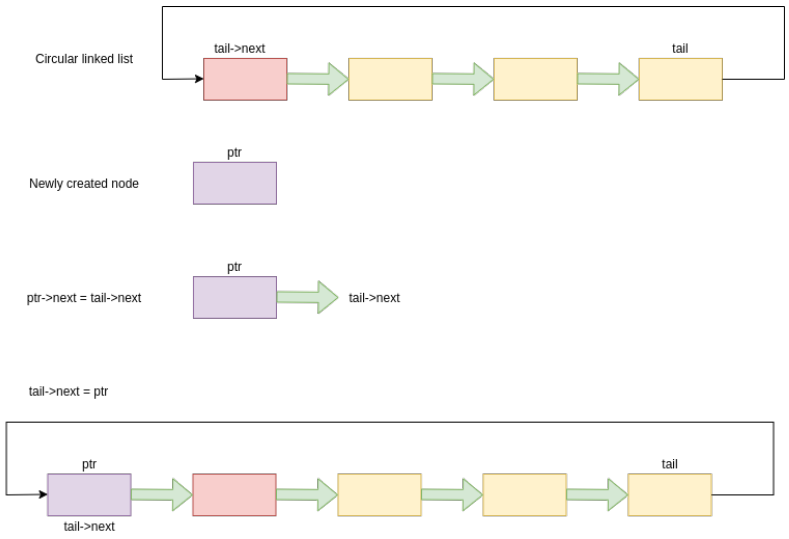
Inserting at tail and deleting at head can be intuitively understood by making corresponding modifications.

11.3.3 Implementation of circular linked list

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

typedef struct node
{
int data;
struct node *next;
} node;

node* make_node(int item)
{
node *ptr = (node*)malloc(sizeof(node));
ptr->data = item;
return ptr;
}

void display(node *tail)
{
/*Use the link connecting the tail to the head, with do while
so you first move to the head before checking the condition*/
node *cur = tail;
if(tail != NULL)
{
do{
cur = cur->next;
printf(" %d ->",cur->data);
}while(cur!=tail);
}
printf("\n");
}

int length(node *tail)
{
int len = 0;
node *cur = tail;
if(tail != NULL)
{
do{
cur = cur->next;
len+=1;
}while(cur!=tail);
}
return len;
}
```

```
node* insertAtHead(node *tail, int item)
{
node *ptr = make_node(item);
if(tail == NULL)
{
tail = ptr;
ptr->next = tail;
}
else
{
ptr->next = tail->next;
tail->next = ptr;
}
return tail;
}

/*
Since it is a circular linked list, insert at head and tail mean the same,
instead of returning tail, return next->tail, so the newly added node
gets printed at the end
*/
node* insertAtTail(node *tail, int item)
{
return insertAtHead(tail,item)->next;
}
```

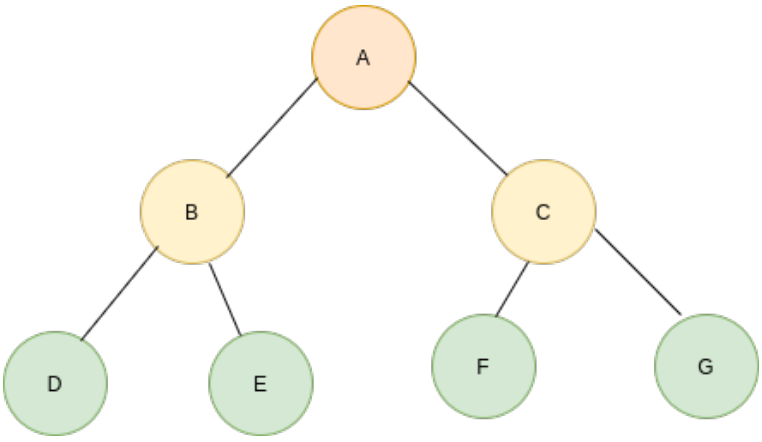
Visualisation of inserting at head



11.4 Binary Trees

- Hierarchial datastructure
- Top-most node is called root of the tree.
- Elements directly under a node are called children
- Elements directly above a node are called parents
- Elements with no children are called leaf nodes
- Each node has at most 2 children in a binary tree.
- Maximum number of nodes at level n cannot exceed 2^n.

11.4.1 Binary Tree traversal



- Depth Order Traversal
  - Inorder
    D, B, E, A, F, C, G
  - Preorder
    A, D, E, B, F, G, C
  - Postorder
    D, E, B, F, G, C, A
- Level Order Traversal Breadth first Traversal A, B, C, D, E, F, G

11.4.2 Binary Search Tree

- Each node has a data member to store a key, each key should be unique
- Left subtree of a node contains only nodes with keys lesser than the node's key

- Right subtree of a node contains only nodes with keys greater than the node's key.
- Left and Right subtree must also be a binary tree.

Insertion in a Binary Search Tree

- A new element is always inserted as a leaf node
- If node to be inserted has a key smaller than that of root node, the element is inserted into left subtree of root otherwise added as a right subtree.

Deletion of a node in a Binary Search Tree

Possible cases

- Node to be deleted has to two children

  Deleted node is replaced with in-order predecessor with maximum valued key in left subtree, or in-order successor with least valued key in the right subtree.

- Node to be deleted has a single child

  Deleted node is replaced by the child directly

- Node to be deleted is a leaf node

  Directly deleted by removing its link from the parent node.

Implementation of binary search tree

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

typedef struct node
{
int data;
struct node* left;
struct node* right;
}Node;

Node* make_node(int item)
{
Node *ptr = (Node*)malloc(sizeof(Node));
ptr->data = item;
ptr->left = NULL;
ptr->right = NULL;

return ptr;
}
```

```
void printPostOrder(Node *node)
{
if(node == NULL)
return;

//first recur on left subtree
printPostOrder(node->left);

//then recur on right subtree
printPostOrder(node->right);

//now deal with the node
printf("%d ", node->data);
}

void printInOrder(Node *node)
{
if(node == NULL)
return;

//first recur on left subtree
printInOrder(node->left);

//now deal with the node
printf("%d ", node->data);

//then recur on right subtree
printInOrder(node->right);
}

void printPreOrder(Node *node)
{
if(node == NULL)
return;

//first deal with the node
printf("%d ", node->data);

//then recur on left subtree
printPostOrder(node->left);

//then recur on right subtree
printPostOrder(node->right);
}
```

```
int findHeight(Node* node)
{
if(node == NULL)
return -1;

int lHeight = findHeight(node->left);
int rHeight = findHeight(node->right);

if(lHeight > rHeight)
return lHeight + 1;
else
return rHeight + 1;
}

int main()
{
Node *root = make_node(1);
root->left = make_node(2);
root->right = make_node(3);
root->left->left = make_node(4);
root->left->right = make_node(5);
root->right->left = make_node(6);
root->right->right = make_node(7);

printf("Pre-order traversal of binary tree\n");
printPreOrder(root);

printf("\nPost-order traversal of binary tree\n");
printPostOrder(root);

printf("\nIn-order traversal of binary tree\n");
printInOrder(root);

printf("\nHeight of the tree: %d\n",findHeight(root));
}

OUTPUT
Pre-order traversal of binary tree
1 4 5 2 6 7 3
Post-order traversal of binary tree
4 5 2 6 7 3 1
In-order traversal of binary tree
4 2 5 1 6 3 7
Height of the tree: 2
```

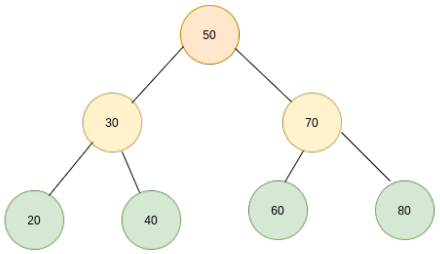
Inserting and Deleting nodes from Binary search Tree

```
typedef struct node
{
int key;
struct node* left;
struct node* right;
}Node;

Node* make_node(int item)
{
Node *ptr = (Node*)malloc(sizeof(Node));
ptr->key = item;
ptr->left = NULL;
ptr->right = NULL;

return ptr;
}

void printInOrder(Node *node)
{
if(node != NULL)
{
printInOrder(node->left);
printf("%d ", node->key);
printInOrder(node->right);
}
}

Node* insert(Node *node, int key)
{
if(node == NULL)
return make_node(key);

if(key < node->key)
node->left = insert(node->left, key);
else
node->right = insert(node->right, key);

return node;
}

//Returns the node with the least value, does not require the search of the entire tree
Node* minValueNode(Node* node)
{
Node* ptr = node;

while(ptr->left != NULL)
```

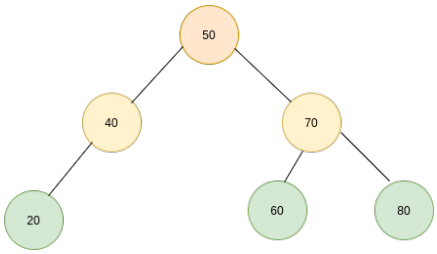
```
ptr = ptr->left;

return ptr;
}

Node* deleteNode(Node* root, int key)
{
//base case
if(root == NULL)
return root;

//if key to be deleted is smaller than that of root node it lies in left subtree
if(key < root->key)
root->left = deleteNode(root->left, key);

//if key to be deleted is greater than that of root node it lies in right subtree
else if(key > root->key)
root->right = deleteNode(root->right, key);

//if key to be deleted is equal to root key, then this node must be deleted.
else
{
//node with only one child
if(root->left == NULL)
{
Node *temp = root->right;
free(root);
return temp;
}
else if(root->right == NULL)
{
Node *temp = root->left;
free(root);
return temp;
}

//node with two children, go for inorder processor
Node* temp = minValueNode(root->right);

root->key = temp->key;
root->right = deleteNode(root->right, temp->key);
}
return root;
}

int main()
```

```
{
Node *root = NULL;
root = insert(root,50);
root = insert(root,30);
root = insert(root,20);
root = insert(root,40);
root = insert(root,70);
root = insert(root,60);
root = insert(root,80);

printf("Inorder traversal of the tree \n");
printInOrder(root);

printf("\nDelete 20\n");
root = deleteNode(root,20);

printf("Inorder traversal of the tree \n");
printInOrder(root);
}

OUTPUT
Inorder traversal of the tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the tree
30 40 50 60 70 80
```

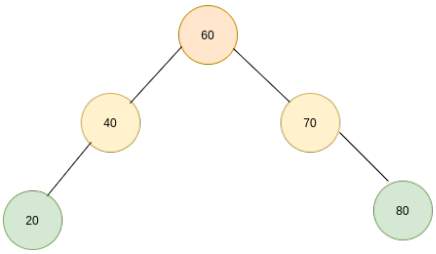
Visualisation of the node deletion



Node to be deleted 30



Node to be deleted 50



12 Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithm is a set of instructions to be followed in problem solving operation. It has a definite beginning, definite end and a finite number of steps. Algorithm can be implemented in more than one programming language. From Data structure point of view, important categories of algorithms:

- Search - Algorithm to search an item in a data structure.
- Sort - Algorithm to sort the items in a certain order.
- Insert - Algorithm to insert item in a data structure.
- Update - Algorithm to update an existing item in a data structure.
- Delete - Algorithm to delete an existing item from a data structure.

12.1 Characteristics of Algorithm

An algorithm should have the following characteristics:

- Input
  The input is the data to be transformed during the computation to produce the output. An algorithm should have 0 or more well-defined inputs.
- Output
  The output is the data resulting from the computation. An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- Definiteness / Unambiguous
  Algorithms must specify every step and the order the steps must be taken in the process. Definiteness means specifying the sequence of operations for turning input into output. Algorithm should be clear and unambiguous.
- Effectiveness / Feasibility
  All the steps mentioned in the algorithm should be feasible with the available resources. It should not contain any unnecessary and redundant steps which could make an algorithm ineffective.
- Finiteness
  Algorithm must have a break condition. It should terminate after a finite number of steps.
- Independent
  An algorithm should have step-by-step directions which are independent of any any programming code.

12.2 Algorithm Analysis

Efficiency of an algorithm can be analysed at different stages - before implementation and after implementation.

- Priori Analysis
  Also called as performance analysis. It is a theoretical analysis of an algorithm. Efficiency is measured by assuming that all other factors are constant and have no effect on implementation.

- Posterior Analysis
  Also called as performance measurement. It is an empirical analysis of an algorithm. The algorithm is implemented using a programming language and is executed on target computer machine. In this analysis, actual statistics are collected.

Analysis of algorithms is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them. Algorithm analysis deals with the execution or running time of various operations involved. The running time can be defined as the number of computer instructions executed per operation. The results of the analysis allows us to make quantitative judgements about the value of one algorithm over the other.

12.3 Algorithm complexity

The complexity of an algorithm is a function $f(n)$ which measures the time and space used by an algorithm in terms of input size n. In computer science, the complexity of an algorithm is a way to classify how efficient an algorithm is, compared to alternative ones. The focus is on how execution time increases with the data set to be processed. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures. Two main factors which decide the efficiency of algorithm are: Time factor and Space factor.

12.3.1 Space complexity

Space complexity of an algorithm is the amount of memory it needs to run to completion. Space complexity includes both Auxiliary space and space used by input. The space required by an algorithm is equal to the sum of the following two components:

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

The space requirement S(P) of any algorithm P can be written as $S(P) = c + S_p$ where c is a constant and S_p is instance characteristics. When analysing the space complexity of an algorithm, we concentrate on estimating the S_p. For any problem, we need to determine which instance characteristics to use to measure the complexity. This is very problem specific. Generally speaking it is limited to quantities like number and magnitude of inputs and outputs of the algorithm.

12.3.2 Time complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time. The time T(P) taken by a program P is the sum of the compile time and the execution time. Compile time doesn't depend on the instance characteristics. Run time is denoted by t_p(instance characteristics). The value of $t_p(n)$ for any input n can be obtained only experimentally. The program is typed, compiled and run on a machine and is clocked physically. This process is not efficient.

The next best alternative is to calculate the number of program steps. Program step is loosely defined as syntactically or semantically meaningful segment of a program. There are 3 kinds of step counts: best case, worst case, and average. The *best-case* step count is the minimum number of steps that can be executed for the given parameters. The *worst-case* step count is the maximum number of steps that can be executed for the given parameters. The *average* step count is the average number of steps executed on instances with the given parameters.

Number of steps can be determined in one of the two ways. First method is to introduce a new variable called *count*. It is a global variable with initial value 0. Statements to increment *count* by appropriate amount are introduced. Each time a statement of original program is executed, *count* is incremented.

Example:

Algorithm Sum(a,n) $sum := 0$; i:=1n $sum := sum + a[i]$; **return** sum;

The problem instances for this algorithm are characterised by n, the number of elements to be summed. The space needed by n is one word and the space needed by a is the space needed by variables of type array. This is at least n words. So, $S_S um(n) \geq (n + 3)$. (n for $a[]$, one each for n,i and s).

To calculate the time complexity we need the step count. This can be determined by introducing *count* statements.

Algorithm Sum(a,n) $sum := 0$; $count := count + 1$; *count* is global; It is initially zero.i:=1 to n $count := count + 1$; for **for** $sum := sum + a[i]$; $count := count + 1$; For last time of **for** $count := count + 1$; For the **return** **return** sum;

With this, it is easy to see that in the **for** loop, the value pf *count* will increase by a total of $2n$. And if *count* is zero to start with, then on termination it will be $2n + 3$.

12.4 Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance. Using this we can conclude the best case, average case and worst case scenario of an algorithm. Asymptotic analysis is input bound, if there is no input for an algorithm it is concluded that it works in constant time. Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. Usually, the time required falls under one of the three types -

- Best Case
  Minimum time required for the execution
- Average Case
  Average time required for the execution
- Worst Case
  Maximum time required for the execution

12.5 Asymptotic Notations

Commonly used Asymptotic notations to calculate the running time complexity are:

12.5.1 Big Oh Notation O

Definition : The function $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$.

The notation $O(n)$ is the formal way to express upper bound of an algorithm's running time. Measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

12.5.2 Omega Notation Ω

Definition : The function $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$.

The notation $\Omega(n)$ is the formal way to express lower bound of an algorithm's run time. Measures the best case time complexity or the least amount of time an algorithm can possibly take to complete.

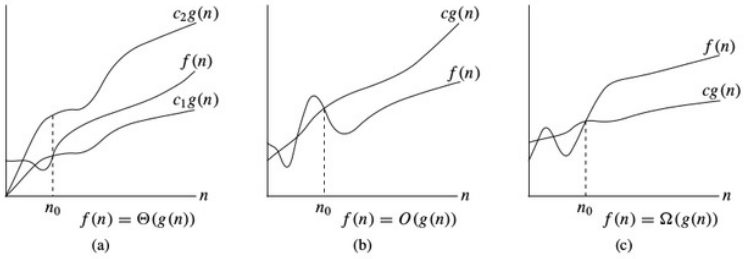


Figure 1: Example functions of notations

12.5.3 Theta Notation Θ

Definition : The function $f(n) = \Theta(g(n))$ iff there exist positive constants c_1,c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 * g(n)$ for all n, $n \geq n_0$.

The notation $\Omega(n)$ is the formal way to express lower bound of an algorithm's run time. Measures the best case time complexity or the least amount of time an algorithm can possibly take to complete.

12.5.4 Examples

1. `printf("Hello World..!!\n");`

   Time Complexity of the above program is **O(1)**

2. ```
   for(int i=0;i<N;i++)
       printf("Hello World..!!\n");
   ```

   Time Complexity of the above program is **O(N)**

3. ```
   for(int i=0;i<N:i++)
       for(int j=0;j<N;j++)
           printf("Hello World..!!\n");
   ```

   Time Complexity of the above program is $\mathbf{O(N^2)}$

4. ```
   int sum(int x, int y, int z)
   {
       int r = x + y + z;
       return r;
   }
   ```

   The Space Complexity of the above program is **O(1)**

5. ```
   int sum(int a[], int n){
       int r = 0;
       for(int i=0;i<n;i++)
           r += a[i];
   ```

```
return r; }
```

The Space Complexity of the above program is **O(N)**

NOTE:

- If a certain function which requires N bytes of memory is called m times in a program, then the space complexity is still **O(N)**. As memory is reused every time the function is called. However this is not true in case of recursive functions.
- Space Complexity changes based on *Call by Value* or *Call by Reference*

12.6 Sorting Algorithms

12.6.1 Bubble Sort

- Brute force application to the sorting problem.
- Comparison=based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order.
- Not suitable for large data sets.
- Time Complexity
  - Worst-case: $O(n^2)$
  - Best-case: $O(n)$
  - Average-case: $O(n^2)$
- Space Complexity: $O(1)$

Pseudocode for Insertion sort

Algorithm BubbleSort(A$[0..n-1]$**)**
//Input: An array of n orderable elements
//Output: Array sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ **then**
      Swap $A[j]$ and $A[j+1]$
    end if
  end for
end for

Implementation in C

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
    int i, j, temp, flag=0;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                // if swapping happens update flag to 1
                flag = 1;
            }
        }
        // if value of flag is zero after all the iterations of inner loop
        // then break out
        if(flag==0)
        {
            break;
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
```

```
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

Visualization

12.6.2 Insertion Sort

- In-place comparison-based sorting algorithm.
- It is considered to be an application of decrease-by-one technique which is a type of divide-and-conquer technique
- Elements are processed sequentially
- One by one sorted elements are placed in a sorted sub array into the same array
- The number of key comparisons in this algorithm depends on the nature of the input.
  Worst case input is an array of strictly decresing values. The number of key comparisons for such input is:

  $$C_{worst}(n) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

  Best case input is when the array is already sorted in ascending order. The number of key comparisons for such input is:

  $$C_{best}(n) = (n-1) \in \Theta(n)$$

  Average case input is when the array is randomly ordered. The number of key comparisons for such input is:

  $$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

Pseudocode for Insertion sort

Algorithm InsertionSort(A$[0..n-1]$**)**
//Input: An array of n orderable elements
//Output: Array sorted in non-decreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[j]$
  $j \leftarrow i-1$
  while $j \geq 0$ **and** $A[j] > v$ **do**

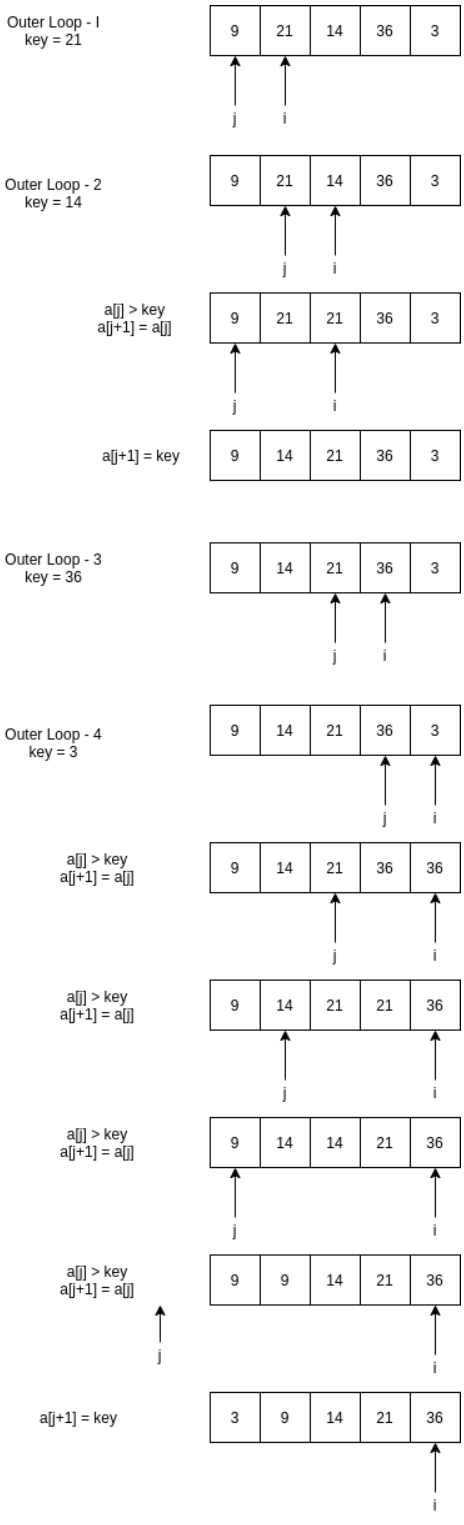
$A[j+1] \leftarrow A[j]$
$j \leftarrow j-1$
end while
$A[j+1] \leftarrow v$
end for

Implementation in C

```
void insertionSort(int a[], int n)
{
    int i,j,key;
    for(i=1;i<n;i++)
    {
        key = a[i];
        j = i-1;

        /* Move elements a[0,..i-1] that are greater than key,
        to one position ahead of  their current position */
        while(j>=0 && a[j] > key)
        {
            a[j+1] = a[j];
            j-=1;
        }
        a[j+1] = key;
}
}
```

Time Complexity: $\mathbf{O(N^2)}$
Space Complexity: $\mathbf{O(N)}$

Visualisation



Insertion Sort

12.6.3 Selection Sort

- Repeatedly finds minimum element in a set of elements from unsorted part and puts it at the beginning
- The algorithm maintains two arrays one with sorted elements the other with unsorted elements
- In every iteration an element from the unsorted array is moved to sorted array

Pseudocode for Selection sort

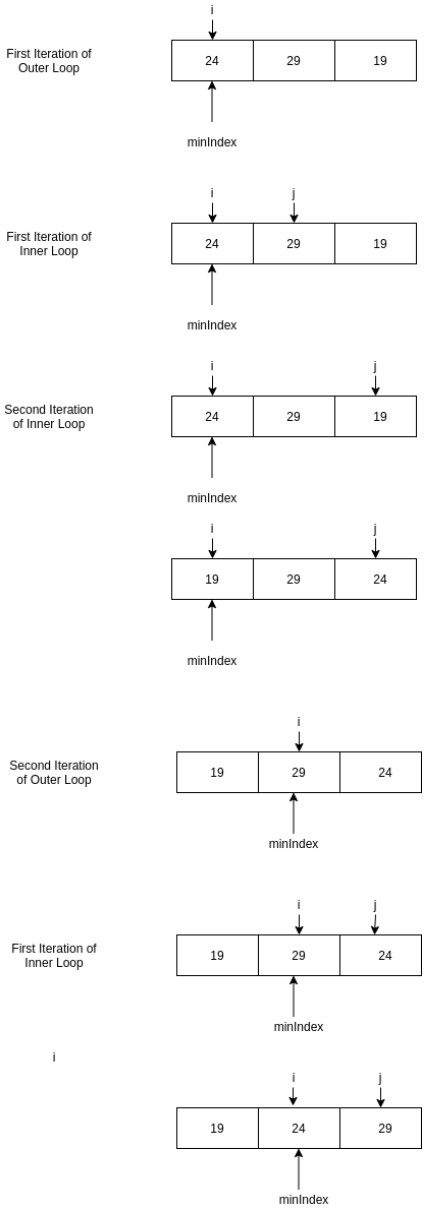
Algorithm SelectionSort(A$[0..n-1]$**)**
//Input: An array of n orderable elements
//Output: Array sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ **then**
      $min \leftarrow j$
    end if
  end for
  Swap $A[i] and A[min]$
end for

Implementation in C

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void selectionSort(int a[], int n)
{
    int i,j,minIndex;

    for(i=0;i<n;i++)
    {
        minIndex = i;
        for(j=i+1;j<n;j++)
         if(a[j] < a[minIndex])
         minIndex = j;

        swap(&a[minIndex], &a[i]);
    }
}
```

Time Complexity: $\mathbf{O(N^2)}$
Space Complexity: $\mathbf{O(1)}$

Visualisation



Selection Sort

12.6.4 Merge Sort

- Divide and conquer technique

- Array is split into two halves and each array is sorted independently by recursively calling the function
- Finally the two sorted sub arrays are merged.

Pseudocode for Merge sort

ALGORITHM Mergesort(A[0 .. n - 1])
//Sorts array A[0..n - 1] by recursive mergesort
//Input: An array A[0..n - 1] of order-able elements
//Output: Array A[0..n - 1] sorted in non-decreasing order
if $n \geq 1$ **then**
  $copyA[0..\lfloor n/2 \rfloor - 1] toB[0..\lfloor n/2 \rfloor - 1]$
  $copyA[\lfloor n/2 \rfloor..n - 1] toC[0..\lceil n/2 \rceil - 1]$
  $Mergesort(B[0..\lceil n/2 \rceil - 1])$
  $Mergesort(C[0..\lfloor n/2 \rfloor - 1])$
  $Merge(B, C, A)$
end if

ALGORITHM Merge(B$[0..p-1]$**,C**$[0..q-1]$**,A**$[0..p+q-1]$**)**
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 .. p -1] and C[0 .. q -1] both sorted
//Output: Sorted array A[0 .. p + q -1] of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ and $j < q$ **do**
  if $B[i] \leq C[j]$ **then**
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else
    $A[k] \leftarrow C[j]$; $j \leftarrow j + 1k \leftarrow k + 1$
  end if
end while
if $i = p$ **then**
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else
  copy $B[i..p-1]$ to $A[k..p+q-1]$
end if

Implementation

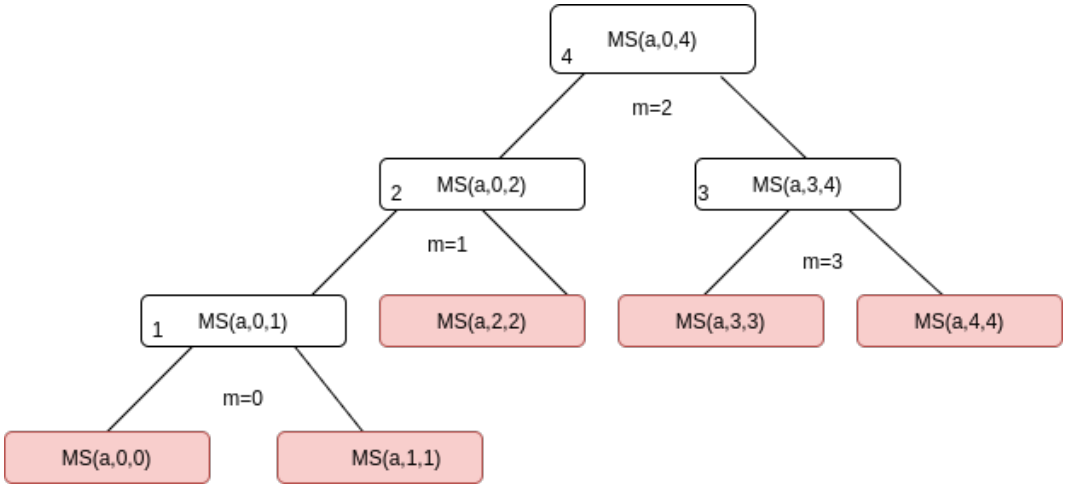
```
//left, middle, right
void merge(int a[], int l, int m, int r)
{
    int i,j,k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];
```

```
    //Copy data into two temp arrays
    for(i=0;i<n1;i++)
        L[i] = a[l+i];
    for(j=0;j<n2;j++)
        R[j] = a[m+1+j];

    //Merge two arrays back into 1
    i=0; j=0; k=l;
    while(i < n1 && j < n2)
    {
        if(L[i] <= R[j])
        {
            a[k] = L[i];
            i++;
        }
        else
        {
            a[k] = R[j];
            j++;
        }
        k++;
    }

    //Copy the remaining elements of L if any are available
    while(i < n1)
    {
        a[k] = L[i];
        i++;
        k++;
    }

    //Copy the remaining elements of R if any are available
    while(j < n2)
    {
        a[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int l ,int r)
{
    if(l<r)
    {
        int m = l + (r-l)/2;
```

```
        //Sort the first and second half
        mergeSort(a,l,m);
        mergeSort(a,m+1,r);
        merge(a,l,m,r);
    }
}
```

Time Complexity: **O(Nlog(N))**
Space Complexity: **O(N)**

Visualisation
Consider an array {9,21,14,36,3}



Merge Sort

From the above flowchart we observe the Recursive calling that occurs while executing the mergeSort. The number inside the rectangle indicates in which order the functions are executed. The ones shaded in red, do not satisfy the condition at the beginning of the mergeSort function.

1. MS(a,0,1) → merge(a,0,0,1)
   n1 = 1
   n2 = 1
   L = { 9 }
   R = { 21 }
   a = { 9, 21, 14, 36, 3}
2. MS(a,0,2) → merge(a,0,1,2)
   n1 = 2
   n2 = 1
   L = { 9, 21 }
   R = { 14 } a = { 9, 14, 21, 36, 3}

3. MS(a,3,4) → merge(a,3,3,4)
   n1 = 1
   n2 = 1
   L = {36 }
   R = { 3 }
   a = { 9, 14, 21, 3, 36 }

4. MS(a,0,4) → merge(a,0,2,4)
   n1 = 3
   n2 = 2
   L = { 9,14,21 }
   R = { 3, 36 }
   a = { 3, 9, 14, 21, 36 }

12.6.5 Quick Sort

- Divide and Conquer technique
- Works by picking a pivot element from a given array
- Arranges all elements around the selected pivot
- Partitions the data by putting array elements smaller than pivot element to the left and greater elements on the right of the array.

Pseudocode for Quick sort

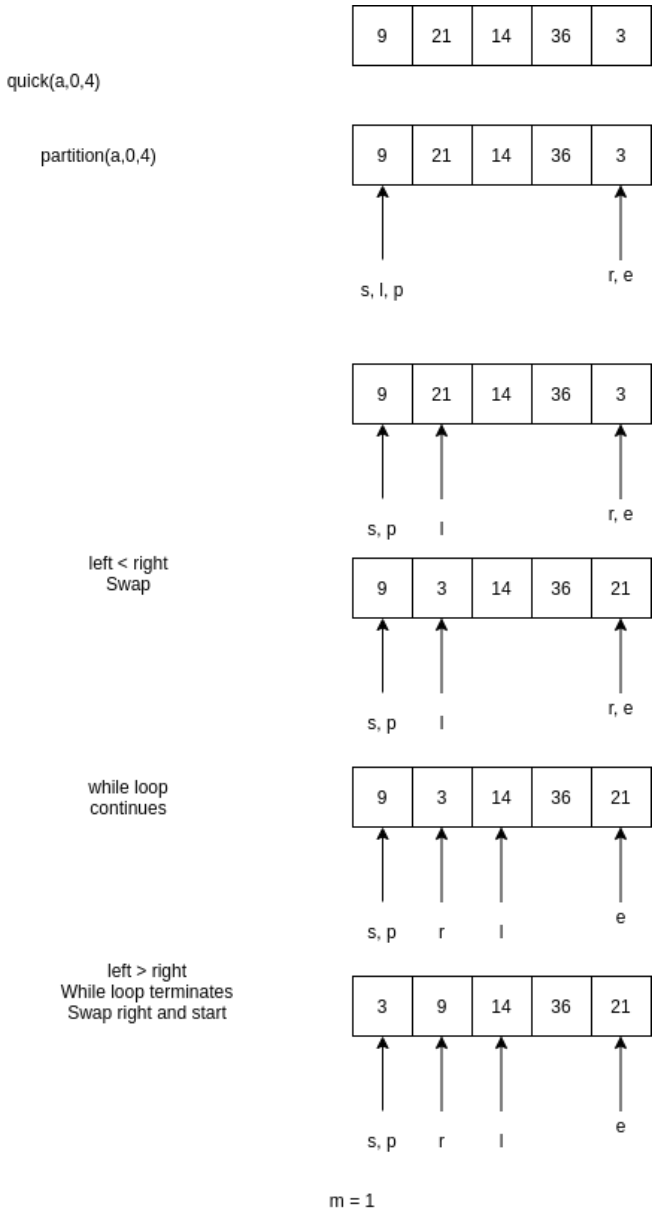
Algorithm Quicksort(A$[l..r]$**)**
//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of n elements, defined by its left and right indices l and r
//Output: Subarray sorted in ascending order
if $l < r$ **then**
  $s \leftarrow Parition(A[l..r])$ {s is a split position}
  $Quicksort(A[l..s-1])$
  $Quicksort(A[s+1..r])$
end if

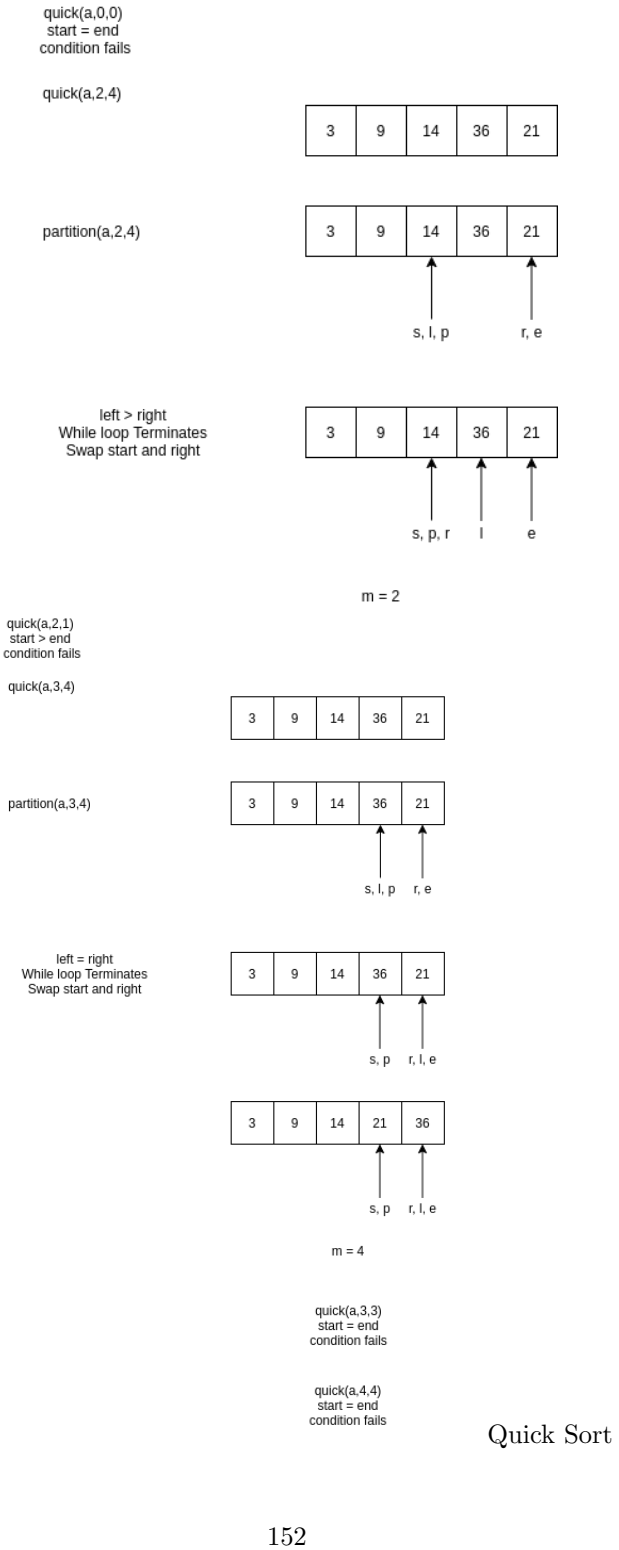
Implementation in C

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int a[], int start, int end)
{
    int pivot = start;
```

```
    int left = start;
    int right = end;
    int pivotVal = a[pivot];

    while(left < right)
    {
        while(a[left] <= pivotVal && left <= end)
            left++;
        while(a[right] > pivotVal && right >= 0)
            right--;
        if(left < right)
         swap(&a[left],&a[right]);
    }

    a[start] = a[right];
    a[right] = pivotVal;
    //print(a,8);
    return right;
}

void quick(int a[], int start, int end)
{
    int m;
    if(start < end)
    {
        m = partition(a,start,end);
        quick(a,start,m-1);
        quick(a,m+1,end);
    }
}
```

Time Complexity: $\mathbf{O(N^2)}$
Space Complexity: $\mathbf{O(\log(N))}$
Visualisation


9 21 14 36 3
quick(a,0,4)
partition(a,0,4)
9 21 14 36 3
s, l, p
r, e
9 21 14 36 3
s, p
l
r, e
left < right
Swap
9 3 14 36 21
s, p
l
r, e
while loop
continues
9 3 14 36 21
s, p
r
l
e
left > right
While loop terminates
Swap right and start
3 9 14 36 21
s, p
r
l
e
m = 1


quick(a,0,0)
start = end
condition fails
quick(a,2,4)
3 9 14 36 21
partition(a,2,4)
3 9 14 36 21
s, l, p
r, e
left > right
While loop Terminates
Swap start and right
3 9 14 36 21
s, p, r
l
e
m = 2
quick(a,2,1)
start > end
condition fails
quick(a,3,4)
3 9 14 36 21
partition(a,3,4)
3 9 14 36 21
s, l, p
r, e
left = right
While loop Terminates
Swap start and right
3 9 14 36 21
s, p
r, l, e
3 9 14 21 36
s, p
r, l, e
m = 4
quick(a,3,3)
start = end
condition fails
quick(a,4,4)
start = end
condition fails

Quick Sort

12.6.6 Heap Sort

- Performed on heap data structure.
- Heap is a complete binary tree. It can be of two types:
  - Min-heap: The root element is minimum.
  - Max-heap: The root element is maximum.
- Heap sort is very fast and is widely used for sorting.
- Time Complexity: $O(n \log(n))$
- Space Complexity: $O(1)$

Pseudocode for Heap Sort

Algorithm Heapify($array, size$)
for $i \leftarrow 1$ **to** $size$ **do**
  $node \leftarrow i$
  $par \leftarrow \lfloor node/2 \rfloor$
  while $par \geq 1$ **do**
    if $array[par] < array[node]$ **then**
      swap array[par] with array[node]
    end if
    $node \leftarrow par$
    $par \leftarrow \lfloor node/2 \rfloor$
  end while
end for

Algorithm heapSort($array, size$)
for $i \leftarrow n$ **to** n **do**
  Heapify(array,i)
  Swap $array[1]$ with $array[i]$
end for

Implementation in C

```
#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    // if left child is larger than root
```

```
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // if right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // if largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    // build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // one by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << "\n";
}


int main()
```

```
{
    int arr[] = {121, 10, 130, 57, 36, 17};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Visualization

12.6.7 Radix Sort

- Digit by digit sort starting from least significant digit to most significant digit.
- Uses counting sort as a subroutine to start.
- Counting sort algorithm can sort larger, multi-digit numbers.
- Time Complexity: $O(d(n+k))$ where d is the number cycle, n is number of digits, k is the base.
- Space Complexity: $O(n+2^d)$

Pseudocode for Radix Sort

Algorithm RadixSort $(list, n)$
$shift \leftarrow 1$
for $loop \leftarrow 1$ **to** $keysize$ **do**
  for $entry \leftarrow 1$ **to** n **do**
    $bucketnumber \leftarrow (list[entry].key/shift) \mod 10$
    $append(bucket[bucketnumber], list[entry])$
  end for
  $list = combinebuckets()$
  $shift = shift * 10$
end for

Implementation in C

```
#include <stdio.h>
// Function to get the largest element from an array
int getMax(int array[], int n)
{
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
```

```
  return max;
}

// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place)
{
  int output[size + 1];
  int max = (array[0] / place) % 10;

  for (int i = 1; i < size; i++)
  {
    if (((array[i] / place) % 10) > max)
      max = array[i];
  }
  int count[max + 1];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  // Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;

  // Calculate cummulative count
  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i--)
  {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }

  for (int i = 0; i < size; i++)
    array[i] = output[i];
}

// Main function to implement radix sort
void radixsort(int array[], int size)
{
  // Get maximum element
  int max = getMax(array, size);

  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
```

```
    countingSort(array, size, place);
}

// Print an array
void printArray(int array[], int size)
{
  for (int i = 0; i < size; ++i)
  {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

// Driver code
int main()
{
  int array[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n);
  printArray(array, n);
}
```

Visualization

12.7 Searching Algorithms

12.7.1 Linear Search

- Simple search algorithm
- All items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise search continues till the end of the data collection.
- Time Complexity:
  - Worst Case: $O(n)$
  - Best Case: $O(1)$
  - Space Complexity:

Pseudocode

Algorithm SequentialSearch($A[0..n-1], K$**)**
//Searches for a given value in a given array by sequential search
//Input: Array $A[0..n-1]$ and a search key K
//Output: The index of the first element of A that matches K
$i \leftarrow 0$

while $i < n$ and $A[i] \neq K$ **do**
  $i \leftarrow i + 1$
end while
if $i < n$ **then**
  return i
else
  return -1
end if

Implementation in C

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);

  printf("Enter %d integer(s)\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter a number to search\n");
  scanf("%d", &search);

  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);

  return 0;
}
```

Visualization

12.7.2 Binary Search

- Requires a sorted array

- Starts by comparing the search element with the middle element in the array:
  - If match is found returns the index of middle element
  - If value of search element is less, the search narrows down to first half of the array.
  - If value of search element is more, the search narrows down to second half of the array.
- This process continues until the search element is reached or if there are no more elements to compare with.

Pseudocode

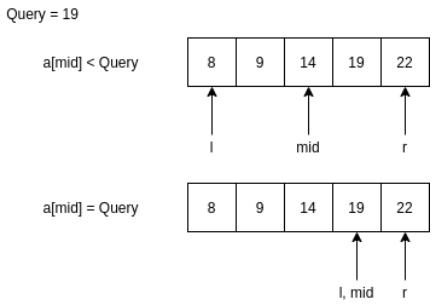
Algorithm BinarySearch($A[0..n-1], K$**)**
//Implements nonrecursue binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
//a search key K
//Output: Index of array's elements that is equal to K
//or -1 if no such element
$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ **then**
    return m
  else if $K < A[m]$ **then**
    $r \leftarrow m-1$
  else
    $l \leftarrow m+1$
  end if
end while
return -1

Implementation in C

```
int binarySearch(int a[], int l, int r, int query)
{
    if(r>= l)
    {
        int mid = l + (r-l)/2;
        if(a[mid] == query)
         return mid;
        else if(a[mid] > query)
         return binarySearch(a,l,mid-1,query);
        else
         return binarySearch(a,mid,r,query);
    }
```

```
    //To indicate element is not present in the array
    return -1;
}
```

Visualisation



Binary Search

12.7.3 Interpolation Search

- Similar to Binary Search.
- Searches for a given target value in a sorted array.
- In Interpolation Search, we move to different element than middle element always in order to reduce the search space further.
- Time Complexity:
  - Worst case: $O(N)$
  - Average case: $O(\log \log N)$
  - Best case: $O(1)$
- Space complexity: $O(1)$

Pseudocode

Algorithm InterpolationSearch()
$L_O \leftarrow 0$; $Mid - 1$; $Hi \leftarrow N - 1$
while X does not match **do**
  if $Lo == Hi$ **or** $A[Lo] == A[Hi]$ **then**
    TARGET NOT FOUND
  end if
  $Mid = Lo + ((Hi - Lo)/(A[Hi] - A[Lo])) * (X - A[Lo])$
  if $A[Mid] = X$ **then**
    Success, Target found at Mid
  else
    if $A[Mid] < X$ **then**
      $Lo \leftarrow Mid + 1$
    else if $A[Mid] > X$ **then**

$Hi \leftarrow Mid - 1$
end if
end if
end while

Implementation in C

```
#include <stdio.h>
int interpolationSearch(int* array, int n, int key){
    int high = n-1;
    int low = 0;
    int mid;
    while ( (array[high] != array[low]) && (key >= array[low]) && (key <= array[high])) {
        mid = low + (int)( ((float)(high-low) / (array[high] - array[low])) * (key - array[l
        if(array[mid] == key)
            return mid;
        else if (array[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
void test(){
    // Array of items on which search will be conducted.
    int array[10] = {1, 5, 9, 12, 16, 18, 25, 56, 76, 78};
    int n = sizeof(array)/sizeof(array[0]);
    // Element to be searched
    int key = 25;
    int index = interpolationSearch(array, n, key);
    if(index !=-1)
        printf("Element found at index %d\n", index);
    else
        printf("Element not found.\n");
}
int main() {
    test(25);
    return 0;
}
```

Visualization

12.7.4 Exponential Search

- Also called Doubling Search, Galloping Search, Struzik Search
- Used for searching sorted, unbounded/infinite arrays.

- Time Complexity:
  - Worst case: $O(\log i)$ where i is the index of element being searched
  - Average case: $O(\log i)$
  - Best case: $O(1)$
- Space complexity: $O(1)$

Pseudocode

Algorithm BinarySearch(array,start,end,key)
if $start \leq end$ **then**
  $mid \leftarrow start + (end - start)/2$
  if $array[mid] = key$ **then**
    return mid location
  end if
  if $array[mid] > key$ **then**
    $binarySearch(array, mid + 1, end, key)$
  else if $array[mid] < key$ **then**
    $binarySearch(array, start, mid - 1, key)$
  end if
else
  return *invalid* location
end if

Algorithm ExponentialSearch(array,start,end,key)
if $(end–start) \leq 0$ **then**
  return *invalid* location
  $i \leftarrow 1$
  while $i < (end - start)$ **do**
    if $array[i] < key$ **then**
      $i \leftarrow i * 2$ {increase i as power of 2}
    else
      terminate the loop
    end if
  end while
  $binarySearch(array, i/2, i, key)$

Implementation in C

```
#include <stdio.h>
int min(int a, int b)
{
    return (a < b ? a : b);
}
int binarySearch(int arr[], int l, int r, int x)
{
```

```
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return (mid);
        if (arr[mid] > x)
            return (binarySearch(arr, l, mid - 1, x));
         return (binarySearch(arr, mid + 1, r, x));
    }
    return (-1);
}
int exponentialSearch(int arr[], int n, int x)
{
    if (arr[0] == x)
        return (0);
    int i = 1;
    while (i < n && arr[i] <= x)
        i = i*2;
    return (binarySearch(arr, i / 2, min(i, n), x));
}
int main()
{
   int n;
   printf("Enter size of Array \n");
   scanf("%d", &n);
   int arr[n], i;
   printf("Enter %d integers in ascending order \n", n);
   for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);
   int x;
   printf("Enter integer to be searched \n");
   scanf("%d", &x);
   int result = exponentialSearch(arr, n, x);
   if (result == -1)
        printf("%d is not present in array \n", x);
   else
        printf("%d is present at index %d \n", x, result);
   return (0);
}
```

Visualization

12.7.5 Jump Search

- Searching algorithms for sorted arrays.
- Checks fewer elements than linear search by jumping ahead by fixed steps or skipping some lements in place of searching all elements

- Optimal block size to be skipped is $m = \sqrt{n}$
- Time Complexity:
  - Worst case: $O(\sqrt{n})$
  - Average case: $O(\sqrt{n})$
  - Best case: $O(1)$
- Space complexity: $O(1)$

Pseudocode

Algorithm JumpSearch(array, size, key)
$blockSize \leftarrow \sqrt{size}$
$start \leftarrow 0$
$end \leftarrow blockSize$
while $array[end] \leq key$ **and** $end < size$ **do**
  $start \leftarrow end$
  $end \leftarrow end + blockSize$
  if $end > size$–1 **then**
    $end \leftarrow size$
  end if
end while
for $i \leftarrow start$ **to** $end - 1$ **do**
  if $array[i] = key$ **then**
    return i
  end if
end for
return *invalid* location

Implementation in C

```
#include <stdio.h>
#include <math.h>
#define MAX 100
int find_element(int element);
int arr[MAX],n;
int main()
{
   int i,element,result;
   printf("\nEnter the number of elements: ");
   scanf("%d",&n);
   printf("\nEnter the elements of array: \n");
   for(i=0;i<n;i++)
   {
         scanf("%d",&arr[i]);
   }
```

```
      printf("\n\nEnter the element you want to search: ");
      scanf("%d",&element);
      result=find_element(element);
      if(result==-1)
      {
            printf("\nElement is not found in the array !\n");
      }
      else
      {
            printf("\nElement %d is present at position %d",element,result);
      }
      return 0;
}
int find_element(int element)
{
   int jump_step,prev=0;
   jump_step=floor(sqrt(n));
   /* Finding block in which element lies, if it is present */
   while(arr[prev]<element)
   {
       if(arr[jump_step]>element || jump_step>=n)
       {
            break;
       }
       else
       {
                prev=jump_step;
                jump_step=jump_step+floor(sqrt(n));
       }
   }
   /*Finding the element in the identified block */
   while(arr[prev]<element)
   {
        prev++;
   }
   if(arr[prev]==element)
   {
        return prev+1;
   }
   else
   {
        return -1;
   }
}
```

Visualization

12.7.6 Ternary Search

- Divide-and-conquer search algorithm.
- Mandatory for the array to be sorted.
- This algorithm divides the whole array into 3 almost equal parts and after each iteration it neglects $\frac{1}{3}$ part of the array and repeats the same operations on the remaining $\frac{2}{3}$
- Time Complexity:
  - Worst case: $O(\log n)$
  - Average case: $O(\log n)$
  - Best case: $O(1)$
- Space complexity: $O(1)$

Pseudocode

if $start \leq end$ **then**
  $midFirst := start + (end - start)/3$
  $midSecond := midFirst + (end - start)/3$
  if $array[midFirst] = key$ **then**
    return $midFirst$
  end if
  if $array[midSecond] = key$ **then**
    return $midSecond$
  end if
  if $key < array[midFirst]$ **then**
    $ternarySearch(array, start, midFirst - 1, key)$
  end if
  if $key > array[midSecond]$ **then**
    $ternarySearch(array, midFirst + 1, end, key)$
  else
    $ternarySearch(array, midFirst + 1, midSecond - 1, key)$
  end if
else
  return *invalid* location
end if

Implementation in C

```
#include <stdio.h>
int ternarySearch(int array[], int left, int right, int x)
{
   if (right >= left) {
     int intvl = (right - left) / 3;
     int leftmid = left + intvl;
```

```
      int rightmid = leftmid + intvl;
      if (array[leftmid] == x)
         return leftmid;
      if (array[rightmid] == x)
         return rightmid;
      if (x < array[leftmid]) {
        return ternarySearch(array, left, leftmid, x);
      }
      else if (x > array[leftmid] && x < array[rightmid]) {
        return ternarySearch(array, leftmid, rightmid, x);
      }
      else {
        return ternarySearch(array, rightmid, right, x);
      }
    }
    return -1;
}
int main(void)
{
    int array[] = {1, 2, 3, 5};
    int size = sizeof(array)/ sizeof(array[0]);
    int find = 3;
    printf("Position of %d is %d\n", find, ternarySearch(array, 0, size-1, find));
    return 0;
}
```

Visualization

13 Error Handling in C

Error handling is the most important part of any program. Error handling is often divided into two parts detection and recovery.

- Detection refers to discovering that an error has occured
- Recovery refers to the process of determination as to how the error should be handled.

13.1 Global Variable - errno

When a function is called in C, a variable named as errno is automatically assigned a code (value) which can be used to identify the type of error that has been encountered. It is defined in the header file *errno.h*.

```
errno value       Error
1             /* Operation not permitted */
2             /* No such file or directory */
3             /* No such process */
4             /* Interrupted system call */
5             /* I/O error */
6             /* No such device or address */
7             /* Argument list too long */
8             /* Exec format error */
9             /* Bad file number */
10            /* No child processes */
11            /* Try again */
12            /* Out of memory */
13            /* Permission denied */
```

13.2 strerror()

Converts the errno to the corresponding text that explains the error.

```
printf("%s\n",strerror(errno));
```

13.3 perror()

It displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

```
void perror (const char *str)
```

str is a string that contains the message to be printed before the actual error message.

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

```
{
    FILE *fp = fopen("ErorHandling.txt","r");
    if (fp == NULL)
    {
        printf("Value of errno: %d\n", errno);
        printf("Error is: %s\n", strerror(errno));
        perror("Oops..!! There is an error in your program");
    }
}

OUTPUT
Value of errno: 2
Error is: No such file or directory
Oops..!! There is an error in your program: No such file or directory
```

13.4 Exit status

The C standard specifies two constants: EXIT_SUCCESS and EXIT_FAILURE, that may be passed to exit() to indicate successful or unsuccessful termination, respectively. The macros are defined in *stdlib.h*

14 Object Oriented Programming

Not possible in C

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Class

- User defined datatype
- Encapsulates member variables and functions
- An object is an instance of a class
  Whenever an object is instantiated the constructor of the class is called.
- Access to member variables and functions of a class is controlled by access modifiers that are private, public and protected.
- Similar to *structure* in C, but it not only encapsulates different datatypes but also functions

14.1 Compile-time binding vs run-time binding

Compile-time binding

```
#include<iostream>
#include<cstdlib>

using namespace std;
class Bird
{
    public:
        void sing()
        { cout << "Bird Singing" << endl; }

        void fly()
        { cout << "Bird flying" << endl;  }
};
class Sparrow : public Bird
{
    public:
        void sing()
        { cout << "Sparrow Singing" << endl; }

        void fly()
        { cout << "Sparrow flying" << endl;  }
};
class Pigeon : public Bird
{
    public:
        void sing()
        { cout << "Pigeon Singing" << endl; }

        void fly()
        { cout << "Pigeon flying" << endl;  }
};
int main()
{
    Bird *bird;
    Sparrow s;
    Pigeon p;

    bird = &s;
    s->sing();
    s->fly();

    bird = &p;
    p->sing();
```

```
    p->fly();
}

OUTPUT
Bird Singing
Bird flying
Bird Singing
Bird flying
```

- Sparrow and Pigeon are subclasses of the class Bird. i.e. they inherit features of the class Bird.
- Compile-time binding resolves the function call based on the type of the source pointer.

Run-time binding

```
#include<iostream>
#include<cstdlib>

using namespace std;
class Bird
{
    public:
        virtual void sing()
        { cout << "Bird Singing" << endl; }

        virtual void fly()
        { cout << "Bird flying" << endl;  }
};
class Sparrow : public Bird
{
    public:
        void sing()
        { cout << "Sparrow Singing" << endl; }

        void fly()
        { cout << "Sparrow flying" << endl;  }
};
class Pigeon : public Bird
{
    public:
        void sing()
        { cout << "Pigeon Singing" << endl; }

        void fly()
        { cout << "Pigeon flying" << endl;  }
};
```

```
int main()
{
    Bird *bird;
    Sparrow s;
    Pigeon p;

    bird = &s;
    s->sing();
    s->fly();

    bird = &p;
    p->sing();
    p->fly();
}

OUTPUT
Sparrow Singing
Sparrow flying
Pigeon Singing
Pigeon flying
```

- Using the keyword **virtual** to define functions in the parent class Bird, instructs the compiler to resolve the function calls at run-time. So the type of the target object is deciding which function is to be called.

14.2 Function-overloading and Function-overriding

```
#include<iostream>
#include<cstdlib>

using namespace std;

//function overloading
class Math_utils
{
    public:
        int add(int x, int y)
            return x+y;
        float add(float x, float y)
            return x+y;
};

//function overriding
class AdderInt
{
    public:
```

```
    int add(int x, int y)
        return x+y;
};
class AdderFloat : public AdderInt
{
    public:
    float add(float x, float y)
        return x+y;
};

int main()
{
    Math_Utils m;
    cout << "Sum of (10,20) = " << m.add(10,20) <<endl;
    cout << "Sum of (10.3,20.5) = " << m.add(10.3,20.5) <<endl;

    AdderFloat a;
    cout << "Sum of (10,20) = " << a.add(10,20) <<endl;
    cout << "Sum of (10.3,20.5) = " << a.add(10.3,20.5) <<endl;
}

OUTPUT
Sum of (10,20) = 30.8
Sum of (10.3,20.5) = 30.8
Sum of (10,20) = 30.8
Sum of (10.3,20.5) = 30.8
```

- **Function overloading** refers to different functions of the same class having the same name but the return type and type of parameters passed are different.
- **Function overriding** refers to functions of the parent-child classes having same name but return type and type of parameters passed are different.