./

Learning Report –C Programming, Linux Basics and OS Programming



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **To be approved By** | **Remarks/Revision Details** |
| 1 |  | Name/PS No | Name/PS No | Module Owner Name | Comments |
| 2 | 28/02/21 | 99003770  Md. Danish Alam |  |  |  |
| 3 |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

Table of Contents:

1. **Introduction to C Programming**
2. Elements of C language
3. Preprocessor directives
4. Functions
5. Variables
6. Statements
7. Character Set and Keyword in C

8.Operators in C

9. Function in C

10. **Introduction to Iinux OS**

11. Kernel

12. Interrupts

13. Process Life Cycle

14. Signals

15. Threads

16. Inter Process Communication

17. Semaphores and Mutex

18. Fifo/Pipes

19. Shared Memory

20. Message Queues

### Introduction to C Programming:

C is a general-purpose, imperative computer programming language. It was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs. The C language is used to re-implement the Unix operating system. It also has been standardized by the American National Standards Institute (ANSI) since 1989 and subsequently by the International Organization for Standardization (ISO).

The C language was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support.

Despite its low-level capabilities, the language was designed to encourage cross-platform programming. The language has become available on a very wide range of platforms, from embedded microcontrollers to supercomputers.

### Some features of C language:

* Easy to learn and understand.
* A case-sensitive language and obey the scope rule.
* It supports the large number of arithmetical and logical operators, such as +, +=, ++, &, ~, etc.
* It has a small, fixed number of keywords, including a full set of control flow primitives: for, if/else, while, switch, and do/while.
* The user can create own data types using the keyword **enum**, **[struct](https://aticleworld.com/structure-in-c/" \t "_blank)**, and **[union](https://aticleworld.com/union-in-c/" \t "_blank)**.
* Supports preprocessor that executes before the compilation.
* It supports **[pointers](https://aticleworld.com/pointers-in-c/" \t "_blank)**.
* Low-level access to computer memory is possible by converting machine addresses to typed pointers.
* Function return values can be ignored when not needed.
* It does not support the oops concept.

### Why use C language?

There is a lot of reason to select C as the development language. I have mentioned a few points that describe why C is a popular language and why people use it.

* It allows accessing the low level directly and that is the reason it is a good choice for driver and firmware development.
* C is a very simple language and easy to learn and the most important thing is that whatever you learn with C will be directly transferable to future programming languages (depends on language).
* Still, C is a good choice for operating system development (including all of Unix).
* C is used in the creation of language compiler and interpreter.
* Programs that are created with C run very quickly.
* There is a lot of legacy programs are written in C.
* C language is also used in the database (Oracle Database).
* C is a portable language (used in C++ and Android APP using JNI wrapper, etc).

# Elements of c language

Before going in depth, we need to understand the basic elements of c language. It is very necessary to know these elements because it helps you to write c program. In below list, I have listed some important elements of C language.

* Preprocessor directives.
* Functions.
* Variables.
* Statements.
* Comments.

### Preprocessor directives:

A pre-processor directive begins with a special character # (pound ). When you compile the code, then before compilation, directives tell the preprocessor to perform specific actions. These actions can be like replacing tokens in the text (using the macro), insert the contents of other files into the source file (Using file inclusion like #include<stdio.h> ), or perform conditional Compilation (using #if, #else, #endif ..etc ).

### Functions:

A function is a collection of statements that perform a task, like the addition of two numbers. In other languages, a function is called procedure or subroutine. In C language there should be at least one function, this mandatory function is main() function. The main() function is an entry point for c language that means it executes first.

### Variables:

A **[variable](https://aticleworld.com/variable-in-c-language/" \t "_blank)** defines a location name where we can put value and we can use these value whenever required in the program. In other words, we can say that variable is a name (or identifier) which indicate some physical address in the memory, where data will be stored in form of the bits of string. The value of a variable can be changed at different times of executions and it may be chosen by the programmer in a meaningful way.

**Syntax of variable,**

Data\_Type  Variable\_Name;

**E.g.,**  
Data\_Type V1, V2, V3; V1 V2 and V3 are three variable of the same data type.

In above example, Data type should be valid. It can be int, char, float, etc or any user-defined data type like structure, union or enum etc.

***Note:*** Variable name should be unique otherwise you will get a compiler error.

### Statements:

A statement is a command given to the computer that instructs the computer to take a specific action, like display message on the console, performing the mathematical operation and so on. In C program is the collection of statements and each statement must be terminated with a semicolon(;).

**There is a lot of type of statements available in C language these are,**

* Labeled Statements (**[switch and case](https://aticleworld.com/switch-case-in-c/" \t "_blank)**)
* Expression Statements ( optional statement like printf (“welcome”); )
* Compound Statements (function)
* Selection Statements (if, else, ..etc)
* Iteration Statements (for,while, ..etc)
* Jump Statements (goto, ..etc) .

# Character set and keywords in C

Like other languages, C has some character set and keywords (reserved words). It is good to know some important character set and keywords in C because it will help during the programming.

### C language character set:

* Lowercase and uppercase letters of ISO Basic Latin Alphabet: a–z A–Z
* Decimal digits: 0–9
* 29 graphic characters

**!  ”  #  %  &  ‘ (  )  \*  +  ,  – .  /  :**  
**;  <  =  >  ?  [  \  ]  ^  \_  {  |  }  ~**

* Whitespace characters: space, horizontal tab, vertical tab, form feed, newline (Newline indicates the end of a text line), carriage return, backspace,..etc.

**OPERATORS IN C**

Operators are the backbone of any programming language. So C/C++ programming language is incomplete without the operators. The C/C++ programming language support mainly three types of operators unary operators, binary operators, and ternary operator (Conditional-expression operator).

# Function in C: Types, Advantages and Use

A function is a set of statements that together perform a specific task.  Every C program consists of one or more functions. The main() function is mandatory for the C program because it is the entry point of your C code from where your program is executed.

Before starting the function let’s see the advantage of the function. It helps us to understand why the function is important for any programming language.

## Advantages of function:

* The function increases the modularity of the program. A large problem can be divided into subproblems and then solved by using functions.
* The function increases the reusability because functions are reusable. Once you have created a function you can call it anywhere in the program without copying and pasting entire logic. So you don’t need to write the same code again and again.
* Because function increases the modularity of your program, so the program becomes more maintainable. If you want to modify the program sometimes later, you only need to update your function without changing the base code.

Now l think you able to understand the advantages of the function if you are not able to understand then don’t worry. I am going to explain the function step by step. So let’s started with the type of function.

## Types of function:

At a broad level, we can categorize function in two types.

1. Library function.
2. User-defined function.

**Note:** We can also categorize function on their inputs and return types.

### Library function:

Like other languages, C has many built-in library functions to perform various operations. for example for input-out operation, scanf and the printf function are used. Similarly for string manipulation strings functions are available like strcpy, strcmp, etc.

You need to remember before using any library function you must include the corresponding header file. For example, if you are going to use string functions, then you must include string.h header file using a pre-processor directive.

Let’s see an example code,

In this example code, I am using strcpy() to copy the string in an array and printf() function to print the array on the console.

**Introduction to Linux OS:**

Operating systems are there from the very first computer generation and they keep evolving with time .Operating System is an interface between the user and the system hardware.

**Types of Operating system**

• Batch OS

• Time-sharing OS (Linux)

• Distributed OS

• Network OS

• Real-time OS

Linux OS Architecture

• Kernel

• Core of OS, responsible for all major OS activities, interacts with hardware, provides abstraction to

hardware from system / application programs

• Drivers

• Used for interaction for additional hardware & I/O

• System Libraries

• Special programs used by system / application

programs access kernel’s features, implement

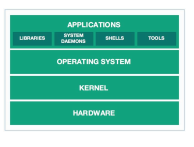
most of the functionalities of OS

• Multimedia library, Network library

• System Utilities

• Used to do specialized, individual level tasks

• Shell, , Terminal



**KERNEL**

• Mandate component of

Operating System

• Resides in memory all the time,

rest all depending on kernel

• Provides basic services including

memory management, IO

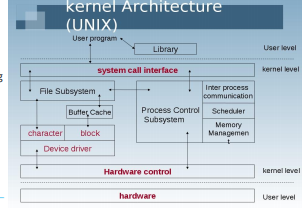
management & other

management services

• Provides services to application

and libraries in the form of

SYSTEM CALLS



**Types of Kernel**

Micro

CPU, memory and IPC in kernel

mode, everything else is

accessory and runs in user

mode.

**Advantages**

• Portability, Small install footprint

• Small memory footprint, Security

**Disadvantages**

• Hardware is more abstracted

through drivers, may react

slower because drivers are in

user mode

• Processes have to wait in a

queue to get information, can’t

get access to other processes

without waiting

Monolithic

Modular

CPU, memory, IPC + device drivers,

file system management, and system

server calls in kernel mode

**Advantages**

• More direct access to hardware for

programs

• Easier for processes to communicate

between each other

• If your device is supported, it should work

with no additional installations

• Processes react faster because there isn’t

a queue for processor time

**Disadvantages**

• Large install footprint

• Large memory footprint

• Less secure because everything runs in

supervisor mode

Linux is modular kernel type is

combination of both

monolithic & micro kernel

**Advantages**

• Has collection of both statically

loaded & dynamically loadable

modules

• No need to load everything on

boot, less boot time, less size,

new need to recompile to add

new module

**Disadvantages**

• Chances of losing stability

• Security Compromise with

modules

• Coding can be difficult

**Linux OS Kernel**

• Compressed Kernel is stored at /boot/vmlinu\*

• Dynamic modules of kernel /lib/modules

• uname –r

• 5.4.0-33-generic

(major.minor.release-tagname)

• Versions of kernel

• 2.x, 2.4, 2.5, 2.6, 3.x, 4.x, 5.x

• 5.x is current version

**Interrupts**

Asynchronous events

• IRQ (Interrupt Request), Interrupt Vector Table,

ISR (Interrupt Service Routine)

• Interrupts must be serviced with utmost priority

• ISR should be as short as possible with

no/minimal blocking calls

• Maskable & Non-maskable Interrupts

• Types

• Hardware Interrupts

• Software Interrupts

**System calls**

Types of System calls

• Process Control

• File Management

• Device Management

• Information Maintenance

• Communications

• Protection

• Write system call:

• printf in C, echo in shell, cout in C++

• Read system call:

• scanf in C, cin in C++

• Trace the system calls:

• strace man, echo, cp, cat

• man strace

• list of system call numbers

**System calls**

• Example folder **“Intro”**

• Standard file descriptors used by any process

• fd = 0 (stdin) (read)

• fd = 1 (stdout) (write)

• fd = 2 (stderr)

• “perror” – always appends error message based on return value of system call

**Process Management**

• **Process is a Program under execution**

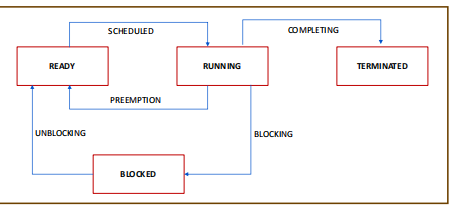
• Program is passive entity and process is active entity

• Every process has its own independent stack

• Kernel maintains process list table in the form of doubly linked list

• Each process has a unique id (pid)

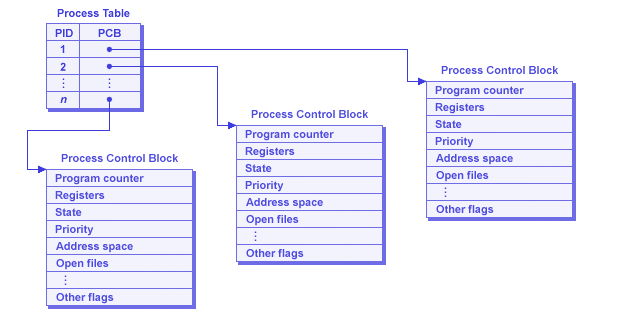
**Process Life Cycle**



**Process Table and Process Control Block**

• **Process table is maintained by kernel**

• **Process Control Block**



**SIGNALS**

• Signals always operate at process level

• Signals communicate between applications at user level

• Used for communication of abnormal termination, illegal memory access & events that go

wrong

• Signals are considered as software interrupts, but there is no interrupt vector table

• Signals between processes

• SENDER send / triggers signals from one process to other process

• TARGET will set the corresponding bit based on sender’s signal bit

• Target will lookup in the signal handler table for handler addresses for each of signal handler

• Process descriptor (PD) / process control block (PCB) has signal related fields

• Most of the default signal handlers will cause abnormal termination.

**Signals in common actions**

• Commands

• kill –l, will list all the signals

• kill –SIGxxxx <pid>

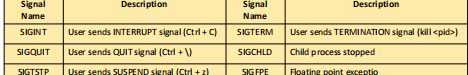
• kill -<signo> <pid>

• kill <pid>

• kill -9 <pid>  terminate is SURE KILL

• kill (pid, signal number)  system call

• pkill, killall, pgrep  process kill



**Default & Custom handlers**

• Signal has default handler

• Custom handler can override the default handler

• APIs – signal, raise, pause, kill, alarm

• Modern APIs – sigaction, sigprocmask, sigsuspend

• Non maskable signals

• Have NO custom handlers

• SIGKILL, SIGSTOP

**Threads**

**Basics of Threads**

• Path of execution within a process

• Various sub-activities within applications are referred as threads

• Referred as Light Weight Process (LWP)

• Significance of threads

• Concurrent execution (parent – child process / multiple child process)

• RESOURCE SHARING across threads

• Child process will have own resources, but threads will have shared resources

• Scheduled threads interchangeably use CPU based on time sharing

• Every process is run initially as a single thread, then multiple threads spawn

• Firefox browser initially will be a single thread, on need basis multiple threads spawn

• Threads are faster than fork

• Common resources during execution run independently

**Advantage of Thread over Process**

• Concurrent execution and faster response, less time for context switch

• Effective use of multiprocessor system

• Resource sharing: code, global data, files can be shared among threads

• PC, Stack and Registers is separate for each thread

• Private / local data is not shared

• Easier communication between threads

• Enhanced throughput of the system

• Number of jobs completed per unit time

Note:

If one thread makes a blocking call, whole process gets blocked.Restricted Circulation | L&T Technology Services | www.LTTS.com

34

**Thread Models**

• Types of threads

• User threads

• Threads used by application programmers, are above kernel and without kernel support

• Kernel threads

• Supported within kernel, perform multiple simultaneous tasks to serve multiple kernel system calls

• Models

• Used to map user threads to kernel threads

• Many to One model

• Many user-level threads are mapped to single kernel thread, thread management is handled by thread

library in user space

• One to One model

• Separate kernel thread is created to handle each and every thread, limitation is the count of threads that

can be created

• Many to Many

• Many user-level threads are mapped to multiple kernel level threadsRestricted Circulation | L&T Technology Services | www.LTTS.com

35

**Commands**

• ps –e –L –o pid,ppid,lwp,nlwp,stat,cmd

• ps –eLf

• To create threads, POSIX thread library is used

• pthread\_create

• pthread\_join

• pthread\_self

• pthread\_equal

• pthread\_yield

• pthread\_cancel

• gcc psample.c -lpthread

**Inter Process Communication(IPC)**

**IPC**

Requirement of IPC

• Data exchange

• Synchronization

• Dependency / Sequencing

• Mutual Exclusion

• Data exchange  shared memory, message queues, FIFOs/pipes

• Mutual exclusion  semaphore, mutex, spinlocks

• Dependency semaphores, condition variables / event flags

Process that writes/updates data is **PRODUCER** and process that reads is **CONSUMER**

**Critical section & Mutual Exclusion**

**Critical Section:** Code/Instructions in a Process/Thread using shared resources

• During process execution in critical section, no switching should be allowed

• Only one Process/Thread can be in a related critical section at any given time.

• Should be as short as possible & no blocking calls

**Mutual exclusion:** Preventing simultaneous access to shared resources

• Disable interrupts (for very shorter duration)

• - User space cannot have access

• - For longer duration, inconsistency occurs

• - Other CPU can access the resources

• Hardware support instructions

• Atomic operation: Programming operations independent of other processes

• - Resources can't be accessed by other process

• Data bus locking techniques: CPU level bus locking techniques

• Above techniques have limitations and not scalable

• Software level solution for Mutual exclusion is semaphore & mutex

**Semaphore and Mutex**

**Mutex**

• Mutual Exclusion

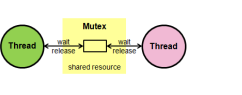
• Only locked Process(es)/Threads can unlock the resources

• Any other Process/Threads trying to unlock is referred as “unauthorized operation”

• Unlocking twice or unlocking before locking is not allowed

• Strictly lock & unlock in the same thread only

• Mutex will have "ownership" as compared to semaphore



**Mutex API’s**

• #include <pthread.h>

• pthread\_mutex\_t m1=PTHREAD\_MUTEX\_INITIALIZER

(declare & initialize)

• pthread\_mutex\_init(&m1)

• pthread\_mutex\_lock(&m1)

(lock)

• pthread\_mutex\_unlock(&m1)

(unlock)

• pthread\_mutex\_destroy (&m1)

(destroy)

Always check return value for Success or Failure

**Semaphores**

• Sequencing, Signaling mechanism, used for process/thread synchronization

• Manage and protect access to shared resources

• Kernel level data structure

Types of usage

• Binary Semaphore

• Value of semaphore ranges between 0 & 1

• Mutual Exclusion / Access to a single resource

• Counting Semaphore

• Value of semaphore can be 0 (zero) & any positive value

• Accessing/sharing multiple similar resources

Two (2) varieties of semaphores

• Traditional System V semaphores

• POSIX semaphores.

Two (2) types of POSIX semaphores

• Named

• Unnamed

**Named Semaphore**

Name is given to semaphore and can be access by parent & child or different processes

• Uses internal shared memory for resources access

**POSIX API’s**

#include <semaphore.h>

#include <errno.h>

• sem\_t \*ps;

(declare a semaphore variable)

• ps = sem\_open(“/s1", O-CREAT, 0666, 1)

(internal shared memory)

• sem\_wait(ps)

(lock the semaphore)

• sem\_post(ps)

(unlock the semaphore)

• sem\_close(ps)

(close semaphore from process)

• sem\_unlink(ps)

(remove named semaphore)

All calls return 0 on success, -1 on error and ‘**errno’** variable is set to error number

**Unnamed Semaphores**

No name is given to the Semaphore.

• Memory is allocated in the program address space

**POSIX Unnamed Semaphore API’s**

#include <semaphore.h>

#include <errno.h>

• sem\_init(sem\_t \*sem, int pshared, unsigned int value)

(Initialize unnamed semaphore)

• sem\_wait(sem\_t \*sem)

(Lock the semaphore )

• Check sem\_trywait & sem\_timedwait

• sem\_post(sem\_t \*sem)

(Unlock the semaphore)

• sem\_destroy(sem\_t \*sem)

(Destroy the semaphore )

All calls return 0 on success, -1 on error and ‘**errno’** variable is set to error number

**Produce and Consumer Problem**

Producer and Consumer scenario

• A Process/Thread will add data – Producer

• A Process/Thread will remove data – Consumer

• Common Buffer/Data Source

• Either Producer or Consumer only can access common data at a time (Shared resource)

• Consumer should block if buffer empty

• Producer should block if Buffer full

**Deadlock**

Two or more processes infinitely blocked (forever) due to circular dependency of resources

• Digital Copy – Printer(s1), Scanner(s2) Problem

• Arbitrary locking of multiple semaphores

• Parent & child - unlocking semaphore after waitpid

• Producer consumer problem - order of locking

Avoid deadlock

• If multiple locks are required, lock all of them at once (atomic locking)

• Don't apply mutual exclusion, before resolving dependency

**Limitations of Semaphore and Mutex as a method of IPC**

• Semaphores & Mutex can never carry data

• Processes / threads need to carry data or exchange the data

**Data Exchange**

**Limitations of Semaphore and Mutex as a method of IPC**

• Semaphores & Mutex can never carry data

• Processes / threads need to carry data or exchange the data

**Need for other IPC Mechanisms**

• Pipes/FIFO

• Message Queue

• Shared Memory

**FIFO/Pipes**

**FIFO/Pipes**

Pipe is a connection between two related processes

• Pipe is one-way communication only

• If a process tries to read before something is written to the pipe, the process is suspended

until something is written.

• For two way communication using pipes, two pipes should be used.

• Process-1 writes to Pipe-1 & reads from Pipe-2

• Process-2 reads from Pipe-1 & writes to Pipe-2

**Named Pipe/ FIFO**

• Connection between two unrelated processes

int mkfifo(const char \*pathname, mode\_t mode)

• mkfifo mypipe, tail -f mypipe

**Pipes**

System Calls related to pipe

#include <unistd.h>

• int pipe(int pipedes[2])

(Create unnamed pipe)

• ssize\_t write(int fd, void \*buf, size\_t count)

(Write to pipe)

• ssize\_t read(int fd, void \*buf, size\_t count)

(Read from pipe)

• int close(int fd)

(Close pipe)

• Advantages and Disadvantages

**Shared Memory**

Memory Segment is created by the kernel and mapped to the data segment of the address

space of a requesting process

Can be used like a global variable in address space

• int shm\_open (const char \*name, int oflag, mode\_t mode);

Create, or gain access to, a shared memory object.

• void \*mmap (void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

Map a shared memory object into its address space.

Do operations on shared memory (read, write, update).

• int munmap (void \*addr, size\_t length);

Delete mappings of the shared memory object.

• int shm\_unlink (const char \*name);

Destroy a shared memory object when no references to it remain open.

**Message Queues**

he messages from Producer are stored on queue & provided on-demand to Consumer

• Typically FIFO based, can also be priority based

• Messages with same priority are read in FIFO order

Synchronization

• On read, if queue is empty, the receiver is blocked

• On write, if the queue is full, sender will be blocked

• Messages are discrete

**Message Queues**

#include <fcntl.h> /\* For O\_\* constants \*/

#include <sys/stat.h> /\* For mode constants \*/

#include <mqueue.h>

• mqd\_t mq\_open(const char \*name, int oflag)

• mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode, struct mq\_attr \*attr)

• int mq\_send(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned int msg\_prio)

• ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned int \*msg\_prio)

• int mq\_close(mqd\_t mqdes)

• int mq\_unlink(const char \*name)

**Thank You!**