MODULE – 1

UNIX and ANSI Standards

UNIX is a computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs, including **Ken Thompson, Dennis Ritchie, Douglas McElroy and Joe Ossanna.** Today UNIX systems are split into various branches, developed over time by AT&T as well as various commercial vendors and non-profit organizations.

1.1 The ANSI C Standard:

In 1989, **American National Standard Institute** (ANSI) proposed C programming language standard X3.159-1989 to standardise the language constructs and libraries. This is termed as ANSI C standard. This attempt to unify the implementation of the C language supported on all computer system. The major differences between ANSI C and K&R C [Kernighan and Ritchie] are as follows:

- Function prototyping
- Support of the const and volatile data type qualifiers
- Support wide characters and internationalization
- Permit function pointers to be used without dereferencing

1.1.1 Function prototyping:

ANSI C adopts C++ function prototype technique where function definition and declaration include function names, arguments' data types, and return value data types. This enables ANSI C compilers to check for function calls in user programs that pass invalid number of arguments or incompatible arguments' data type. These fix a major weakness of K&R C compilers: invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

Eg: unsigned long foo(char * fmt, double data)
{
/*body of foo*/
}

External declaration of this function foo is
unsigned long foo(char * fmt, double data);

eg: int printf(const char* fmt,.....);
specify variable number of arguments

1.1.2 Support of the const and volatile data type qualifiers:

- The **const** keyword declares that some data cannot be changed

Eg: **int printf(const char* fmt,.....);**

Declares a fmt argument that is of a const char * data type, meaning that the function printf cannot modify data in any character array that is passed as an actual argument value to fmt

- **Volatile** keyword specifies that the values of some variables may change asynchronously, giving an hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects

eg**: char get_io()**

```
{
volatile char* io_port = 0x7777;
char ch = *io_port; /*read first byte of data*/
ch = *io_port; /*read second byte of data*/
}
```

If io_port variable is not declared to be volatile when the program is compiled, the compiler may eliminate second ch = *io_port statement, as it is considered redundant with respect to the previous statement.

- The **const** and **volatile** data type qualifiers are also supported in C++.

1.1.3 Support wide characters and internationalization:

- ANSI C supports internationalisation by allowing C-program to use wide characters. Wide characters use more than one byte of storage per character.

- ANSI C defines the **setlocale** function, which allows users to specify the format of date, monetary and real number representations.
  For eg: most countries display the date in dd/mm/yyyy format whereas US displays it in mm/dd/yyyy format

- Function prototype of setlocale function is:

```
#include<locale.h>
char setlocale (int category, const char* locale);
```

- The setlocale function prototype and possible values of the category argument

are declared in the <locale.h> header. The category values specify what format class(es) is to be changed

• Some of the possible values of the category argument are:

category value effect on standard C functions/macros
LC_CTYPE Affects behavior of the <ctype.h> macros
LC_TIME Affects date and time format
LC_NUMERIC Affects number representation format
LC_MONETARY Affects monetary values format
LC_ALL combines the affect of all above

1.1.4 Permit function pointers without dereferencing

ANSI C specifies that a function pointer may be used like a function name. No referencing is needed when calling a function whose address is contained in the pointer. For Example, the following statement given below defines a function pointer funptr, which contains the address of the function foo.

```
extern void foo(double xyz,const int *ptr);
   void (*funptr)(double,const int *)=foo;
```

The function foo may be invoked by either directly calling foo or via the funptr

```
foo(12.78,"Hello world");
funptr(12.78,"Hello world");
```

ANSI C also defines a set of C processor(cpp) symbols, which may be used in user programs. These symbols are assigned actual values at compilation time

cpp Symbol USE
STDC Feature test macro. Value is 1 if a compiler is ANSI C, 0 otherwise
LINE Evaluated to the physical line number of a source file _FILE_ Value is the file name of a module that contains this symbol _DATE_ Value is the date that a module containing this symbol is compiled
TIME value is the time that a module containing this symbol is compiled

1.2 The ANSI/ISO C++ Standards:

These compilers support C++ classes, derived classes, virtual functions, operator overloading. Furthermore, they should also support template classes, template functions, exception handling and the iostream library classes

1.3 Difference between ANSI C and C++:

ANSI C	C++
Uses K&R C default function declaration for any functions that are referred before their declaration in the program **int foo();** ANSI C treats this as old C function declaration & interprets it as declared in following manner. **int foo(……..);** meaning that foo may be called with any number of arguments. Does not employ type_safe linkage technique and does not catch user errors	Requires that all functions must be declared / defined before they can be referenced **int foo();** C++ treats this as int foo(void); Meaning that foo may not accept any arguments Encrypts external function names for type_safe linkage. Thus reports any user errors

1.4 The POSIX Standards:

- POSIX or "Portable Operating System Interface" is the name of a family of related standards specified by the IEEE to define the application-programming interface (API), along with shell and utilities interface for the software compatible with

variants of the UNIX operating system.

- Because many versions of UNIX exist today and each of them provides its own set of API functions, it is difficult for system developers to create applications that can be easily ported to different versions of UNIX.
- Some of the subgroups of POSIX are POSIX.1, POSIX.1b & POSIX.1c are concerned with the development of set of standards for system developers.
- **POSIX.1**
  - This committee proposes a standard for a base operating system API; this standard specifies APIs for the manipulating of files and processes. It is formally known as IEEE standard 1003.1-1990 and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990.
- **POSIX.1b**
  - This committee proposes a set of standard APIs for a real time OS interface; these include IPC (inter-process communication).
  - This standard is formally known as IEEE standard 1003.4-1993.
- **POSIX.1c**
  - This standard specifies multi-threaded programming interface. This is the newest POSIX standard.

  - These standards are proposed for a generic OS that is not necessarily be UNIX system.
  - E.g.: VMS from Digital Equipment Corporation, OS/2 from IBM, & Windows NT from Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems.
  - To ensure a user program conforms to POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before inclusion of any header) as;
    #define _POSIX_SOURCE
    Or specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation;
    % CC -D_POSIX_SOURCE *.C
  - POSIX.1b defines different manifested constant to check conformance of user program to that standard. The new macro is _POSIX_C_SOURCE and its value indicates POSIX version to which a user program conforms. Its value can be:
  - _POSIX_C_SOURCE may be used in place of _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition.
  - There is also a _POSIX_VERSION constant defined in <unistd.h> header. It contains the POSIX version to which the system conforms

1.4.1 The POSIX Environment:

Although POSIX was developed on UNIX, a POSIX complaint system is not necessarily a UNIX system. A few UNIX conventions have different meanings according to the POSIX standards. Most C and C++ header files are stored under the /usr/include directory in any UNIX system and each of them is referenced by

#include<header-file-name>

This method is adopted in POSIX. There need not be a physical file of that name existing on a POSIX conforming system.

1.4.2 The POSIX Feature Test Macros:

POSIX.1 defines a set of feature test macro's which if defined on a system, means that the system has implemented the corresponding features. All these test macros are defined in **<unistd.h>** header

Feature test macro Effects if defined

_POSIX_JOB_CONTROL The system supports the BSD style job control

_POSIX_SAVED_IDS Each process running on the system keeps the saved set UID and the set-GID, so that they can change its effective user-ID and group-ID to those values via seteuid and setegid API's

_POSIX_CHOWN_RESTRI CTED If the defined value is -1, users may change ownership of files owned by them, otherwise only users with special privilege may change ownership of any file on the system

_POSIX_NO_TRUNC If the defined value is -1, any long pathname passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated

_POSIX_VDISABLE If defined value is -1, there is no disabling character for special characters for all terminal device files. Otherwise the value is the disabling character value

1.4.3 Limits checking at Compile time and at Run time

POSIX.1 and POSIX.1b defines a set of system configuration limits in the form of manifested constants in the <limits.h> header

List of POSIX.1 – defined constants in the <limits.h> header

Compile time limit Min. Value **Meaning**

_POSIX_CHILD_MAX 6 Maximum number of child processes that may be created at any one time by a process

_POSIX_OPEN_MAX 16 Maximum number of files that a process can open simultaneously

_POSIX_STREAM_MAX 8 Maximum number of I/O streams opened simultaneously

_POSIX_ARG_MAX 4096 Maximum size, in bytes of arguments that may be passed to

_POSIX_NGROUP_MA X groups
0 Maximum number of supplemental

_POSIX_PATH_MAX 255 Maximum number of characters allowed in a path name
_POSIX_NAME_MAX 14 Maximum number of characters allowed in a file name
_POSIX_LINK_MAX 8 Maximum number of links a file may have
_POSIX_PIPE_BUF 512 Maximum size of a block of data that may be atomically read
_POSIX_MAX_INPUT 255 Maximum capacity of a terminal's input queue (bytes)
_POSIX_MAX_CANON 255 Maximum size of a terminal's canonical input queue
_POSIX_SSIZE_MAX 32767 Maximum value that can be stored in a ssize_t typed
_POSIX_TZNAME_MAX 3 Maximum number of characters in a time zone name

1.5 The POSIX.1 FIPS Standard:

FIPS stands for Federal Information Processing Standard. The FIPS standard is a restriction of the POSIx.1 – 1988 standard, and it requires the following features to be implemented in all FIPS-conforming systems:

- Job control
- Saved set-UID and saved set-GID
- Long path name is not supported
- The _POSIX_CHOWN_RESTRICTED must be defined
- The _POSIX_VDISABLE symbol must be defined
- The NGROUP_MAX symbol's value must be at least 8
- The read and write API should return the number of bytes that have been transferred after the APIs have been interrupted by signals
- The group ID of a newly created file must inherit the group ID of its containing directory. The FIPS standard is a more restrictive version of the POSIX.1 standard

1.6 The X/Open Standards:

The X/Open organization was formed by a group of European companies to propose a common operating system interface for their computer systems. The portability guides specify a set of common facilities and C application program interface functions to be provided on all UNIX based open systems. In 1973, a group of computer vendors initiated a project called "*common open software environment*" (COSE). The goal of the project was to define a single UNIX programming interface specification that would be supported by all type vendors. The applications that conform to ANSI C and POSIX also conform to the X/Open standards but not necessarily vice-versa.

1.7 UNIX and POSIX APIs:

API: A set of application programming interface functions that can be called by user programs to perform system specific functions.

Most UNIX systems provide a common set of API's to perform the following functions:

- Determine the system configuration and user information.
- Files manipulation.
- Processes creation and control.
- Inter-process communication.
- Signals and daemons
- Network communication.

1.8 The POSIX APIs:

In general POSIX API's uses and behaviours' are similar to those of Unix API's. However, user's programs should define the _POSIX_SOURCE or _POSIX_C_SOURCE in their programs to enable the POSIX API's declaration in header files that they include.

1.9 The UNIX and POSIX Development Environment:

POSIX provides portability at the source level. This means that you transport your source program to the target machine, compile it with the standard C compiler using conforming headers and link it with the standard libraries.

Some commonly used POSIX.1 and UNIX API's are declared in <unistd.h> header. Most of POSIX.1, POSIX>1b and

UNIX API object code is stored in the libc.a and lib.so libraries.

1.10 API Common Characteristics:

- Many APIs returns an **integer value** which indicates the termination status of their execution
- API **return -1** to indicate the **execution has failed**, and the global variable **errno** is set with an error code.
- A user process may call **perror** function to print a diagnostic message of the failure to the std o/p, or it may call strerror **function** and gives it errno as the actual argument value; the strerror function returns a diagnostic message string and the user process may print that message in its preferred way
- The possible error status codes that may be assigned to errno by any API are defined in the <errno.h> header.

List of Commonly occurring error codes and their meanings

Code

Error Status Meaning

EACCESS A process does not have access permission to perform an operation via a API.

EPERM A API was aborted because the calling process does not have the superuser

ENOENT An invalid filename was specified to an API BADF A API was called with invalid file descriptor EINTR A API execution was aborted due to a signal interruption EAGAIN A API was aborted because some system resource it requested was temporarily

ENOMEM A API was aborted because it could not allocate dynamic memory EIO I/O error occurred in a API execution

EPIPE A API attempted to write data to a pipe which has no reader EFAULT A API was passed an invalid address in one of its argument ENOEXEC A API could not execute a program via one of the exec API ECHILD A process does not have any child process which it can wait on