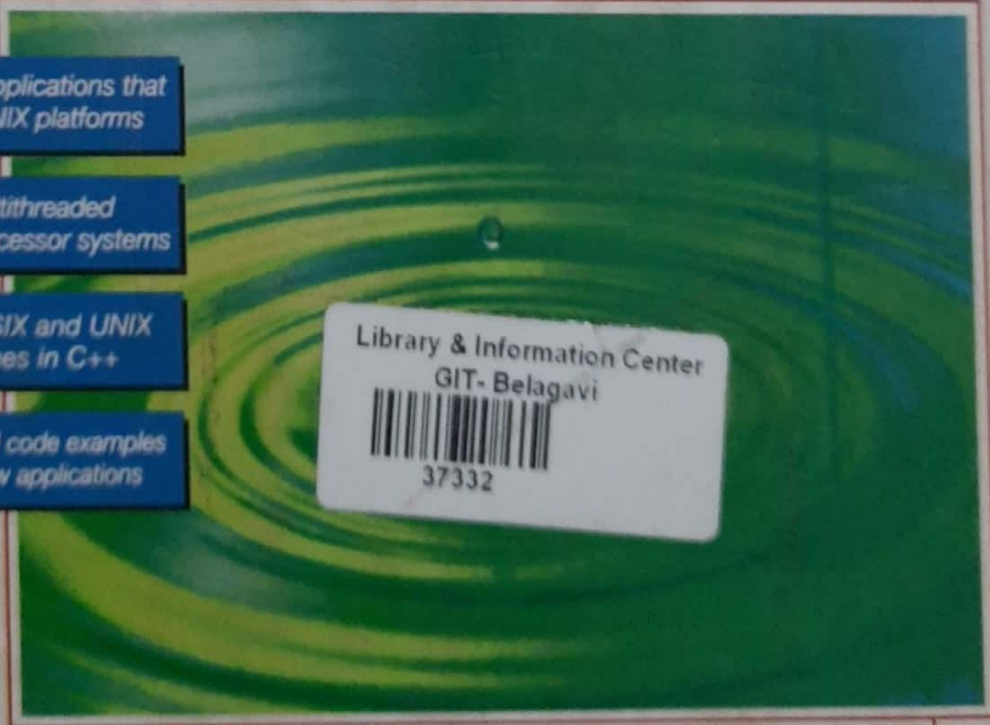
Eastern Economy Edition

UNIX SYSTEM PROGRAMMING USING C++

- Building RPC-based applications that run on heterogenous UNIX platforms
- Creating advanced multithreaded applications for multiprocessor systems
- Advanced ANSI, POSIX and UNIX programming techniques in C++
- Proven C++ classes and code examples to aid development of new applications



Terrence Chan

PHI

CHAPTER 1

UNIX and ANSI Standards

Since the invention of UNIX in the late 1960s, there has been a proliferation of different versions of UNIX on different computer systems. Recent UNIX systems have developed from AT&T System V and BSD 4.x UNIX. However, most computer vendors often add their own extensions to either the AT&T or BSD UNIX on their systems, thus creating the different versions of UNIX. In late 1980, AT&T and Sun Microsystems worked together to create the UNIX System V release 4, which is an attempt to set a UNIX system standard for the computer industry. This attempt was not totally successful, as only a few computer vendors today adopt the UNIX System V.4.

However, in the late 1980s, a few organizations proposed several standards for a UNIX-like operating system and the C language programming environment. These standards are based primarily on UNIX, and they do not impose dramatic changes in vendors' systems; thus, they are easily adopted by vendors. Furthermore, two of these standards, ANSI C and POSIX (which stands for Portable Operating System Interface), are defined by the American National Standard Institute (ANSI) and by the Institute of Electrical and Electronics Engineers (IEEE). They are very influential in setting standards in the industry; thus, most computer vendors today provide UNIX systems that conform to the ANSI C and POSIX.1 (a subset of the POSIX standards) standards.

Most of the standards define an operating system environment for C-based applications. Applications that adhere to the standards should be easily ported to other systems that conform to the same standards. This is especially important for advanced system programmers who make extensive use of system-level application program interface (API) functions (which include library functions and system calls). This is because not all UNIX systems pro-

vide a uniform set of system APIs. Furthermore, even some common APIs may be implemented differently on different UNIX systems (e.g., the *fcntl* API on UNIX System V can be used to lock and unlock files, something that the BSD UNIX version of *fcntl* API does not support). The ANSI C and POSIX standards require all conforming systems to provide a uniform set of standard libraries and system APIs, respectively; the standards also define the signatures (the data type, number of arguments, and return value) and behaviors of these functions on all systems. In this way, programs that use these functions can be ported to different systems that are compliant with the standards.

Most of the functions defined by the standards are a subset of those available on most UNIX systems. The ANSI C and POSIX committees did create a few new functions on their own, but the purpose of these functions is to supplement ambiguity or deficiency of some related constructs in existing UNIX and C. Thus, the standards are easily learned by experienced UNIX and C developers, and easily supported by computer vendors.

The objective of this book is to help familiarize users with advanced UNIX system programming techniques, including teaching users how to write portable and easily maintainable codes. This later objective can be achieved by making users familiar with the functions defined by the various standards and with those available from UNIX so that users can make an intelligent choice of which functions or APIs to use.

The rest of this chapter gives an overview of the ANSI C, draft ANSI/ISO C++, and the POSIX standards. The subsequent chapters describe the functions and APIs defined by these standards and others available from UNIX in more detail.

1.1 The ANSI C Standard

In 1989, the American National Standard Institute (ANSI) proposed C programming language standard X3.159-1989 to standardize the C programming language constructs and libraries. This standard is commonly known as the *ANSI C standard*, and it attempts to unify the implementation of the C language supported on all computer systems. Most computer vendors today still support the C language constructs and libraries as proposed by Brian Kernighan and Dennis Ritchie (commonly known as *K&R C*) as default, but users may install the ANSI C development package as an option (for an extra fee).

The major differences between ANSI C and K&R C 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

Although this book focuses on the C++ programming technique, readers still need to be familiar with the ANSI C standard because many standard C library functions are not covered by the C++ standard classes, thus almost all C++ programs call one or more standard C library functions (e.g., get time of day, or use the *strlen* function, etc.). Furthermore, for some readers who may be in the process of porting their C applications to C++, this section describes some similarities and differences between ANSI C and C++, so as to make it easy for those users to transit from ANSI C to C++.

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

The following example declares a function *foo* and requires that *foo* take two arguments; the first argument *fmt* is of *char** data type, and the second argument is of *double* data type. The function *foo* returns an *unsigned long* value:

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

To create a declaration of the above function, a user simply takes the above function definition, strips off the body section, and replaces it with a semicolon character. Thus, the external declaration of the above function *foo* is:

```
unsigned long foo ( char* fmt, double data );
```

For functions that take a variable number of arguments, their definitions and declarations should have "..." specified as the last argument to each function:

```
int printf( const char* fmt, ...);

int printf( const char* fmt, ... )
{
    /* body of printf */
}
```

The *const* key word declares that some data cannot be changed. For example, the above function prototype 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*.)

The *volatile* key word specifies that the values of some variables may change asynchronously, giving a hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects. For example, the following statements define an *io_Port* variable that contains an address of an I/O port of a system. The two statements that follow the definition are to wait for two bytes of data to arrive from the I/O port and retain only the second byte of data!

```
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 */
}
```

In the above example, if the *io_Port* variable is not declared to be "volatile," when the program is compiled, the compiler may eliminate the 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++.

ANSI C supports internationalization by allowing C programs to use wide characters. Wide characters use more than one byte of storage per character. These are used in countries where the ASCII character set is not the standard. For example, the Korean character set requires two bytes per character. Furthermore, ANSI C also defines the *setlocale* function, which allows users to specify the format of date, monetary, and real number representations. For example, most countries display the date in <day>/<month>/<year> format, whereas the US displays the date in <month>/<day>/<year> format.

The function prototype of the *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 possible values of the *category* argument are:

category value	Effect on standard C functions/macros
LC_CTYPE	Affects the behaviors of the <ctype.h> macros
LC_TIME	Affects the date and time format as returned by the *strftime, ascftime* functions, etc.
LC_NUMERIC	Affects the number representation formats via the *printf* and *scanf* functions
LC_MONETARY	Affects the monetary value format returned by the *localeconv* function
LC_ALL	Combines the effects of all the above

The *locale* argument value is a character string that defines which locale to use. Possible values may be C, POSIX, en_US, etc. The C, POSIX, en_US locales refer to the UNIX, POSIX, and US locales. By default, all processes on an ANSI C or POSIX compliant system execute the equivalent of the following call at their process start-up time:

```
setlocale( LC_ALL, "C" );
```

Thus, all processes start up have a known locale. If a *locale* value is NULL, the *setlocale* function returns the current *locale* value of a calling process. If a *locale* value is "" (a null string), the *setlocale* function looks for an environment variable LC_ALL, an environment variable with the same name as the *category* argument value, and, finally, the LANG environment variable - in that order - for the value of the *locale* argument.

The *setlocale* function is an ANSI C standard that is also adopted by POSIX.1.

ANSI C specifies that a function pointer may be used like a function name. No dereference is needed when calling a function whose address is contained in the pointer. For example, the following statements define a function pointer *funcptr*, which contains the address of the function *foo*:

```
extern void foo ( double xyz, const int* Iptr );
void (*funcptr)(double, const int*) = foo;
```

The function *foo* may be invoked by either directly calling *foo* or via the *funcptr*. The following two statements are functionally equivalent:

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

The K&R C requires *funcptr* be dereferenced to call *foo*. Thus, an equivalent statement to the above, using K&R C syntax, is:

```
(*funcptr)(12.78, "Hello world");
```

Both the ANSI C and K&R C function pointer uses are supported in C++.

In addition to the above, ANSI C also defines a set of *cpp* (C preprocessor) symbols which may be used in user programs. These symbols are assigned actual values at compile time:

***cpp* symbol**	**Use**
__STDC__	Feature test macro. Value is 1 if a compiler is ANSI C conforming, 0 otherwise
__LINE__	Evaluated to the physical line number of a source file for which this symbol is reference
__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

The following *test_ansi_c.c* program illustrates uses of these symbols:

```
#include <stdio.h>
int main()
{
#if __STDC__ == 0
    printf("cc is not ANSI C compliant\n");
#else
    printf(" %s compiled at %s:%s. This statement is at line %d\n",
                __FILE__, __DATE__, __TIME__, __LINE__);
#endif
    return 0;
}
```

Note that C++ supports the __LINE__, __FILE__, __DATE__, and __TIME__ symbols, but not __STDC__.

Finally, ANSI C defines a set of standard library functions and associated headers. These headers are the subset of the C libraries available on most systems that implement K&R C. The ANSI C standard libraries are described in Chapter 4.

1.2 The ANSI/ISO C++ Standard

In early 1980s, Bjarne Stroustrup at AT&T Bell Laboratories developed the C++ programming language. C++ was derived from C and incorporated object-oriented constructs, such as classes, derived classes, and virtual functions, from simula67 [1].The objective of developing C++ is "to make writing good programs earlier and more pleasant for individual programmer" [2]. The name C++ signifies the evolution of the language from C and was coined by Rick Mascitti in 1983.

Since its invention, C++ has gained wide acceptance by software professionals. In 1989, Bjarne Stroustrup published *The Annotated C++ Reference Manual* [3]. This manual became the base for the draft ANSI C++ standard, as developed by the X3J16 committee of ANSI. In early 1990s, the WG21 committee of the International Standard Organization (ISO) joined the ANSI X3J16 committee to develop a unify ANSI/ISO C++ standard. A draft version of such a ANSI/ISO standard was published in 1994 [4]. However, the ANSI/ISO standard is still in the development stage, and it should become an official standard in the near future.

Most latest commercial C++ compilers, which are based on the AT&T C++ language version 3.0 or later, are compliant with the draft ANSI/ISO standard. Specifically, these compilers should 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.

This book will describe the C++ language features as defined by the draft ANSI/ISO C++ standard.

1.3 Differences Between ANSI C and C++

C++ requires that all functions must be declared or defined before they can be referenced. ANSI C uses the K&R C default function declaration for any functions that are referenced before their declaration and definition in a user program.

Another difference between ANSI C and C++ is given the following function declaration:

```
int foo ();
```

ANSI C treats the above function as an old C function declaration and interprets it as declared in the following manner:

```
int foo (...);
```

which means *foo* may be called with any number of actual arguments. However, for C++, the same declaration is treated as the following declaration:

```
int foo ( void );
```

which means *foo* may not accept any argument when it is called.

Finally, C++ encrypts external function names for type-safe linkage. This ensures that an external function which is incorrectly declared and referenced in a module will cause the link editor (*/bin/ld*) to report an undefined function name. ANSI C does not employ the type-safe linkage technique and, thus, does not catch these types of user errors.

There are many other differences between ANSI C and C++, but the above items are the more common ones run into by users (For a detailed documentation of the ANSI C standard, please see [5]).

The next section describes the POSIX standards, which are more elaborate and comprehensive than are the ANSI C standard for UNIX system developers.

1.4 The POSIX Standards

Because many versions of UNIX exist today and each of them provides its own set of application programming interface (API) functions, it is difficult for system developers to create applications that can be easily ported to different versions of UNIX. To overcome this problem, the IEEE society formed a special task force called POSIX in the 1980s to create a set of standards for operating system interfacing. Several subgroups of the POSIX such as POSIX.1, POSIX.1b and POSIX.1c are concerned with the development of a set of standards for system developers.

Specifically, the POSIX.1 committee proposes a standard for a base operating system application programming interface; this standard specifies APIs for the manipulation of files and processes. It is formally known as the IEEE standard 1003.1-1990 [6], and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990. The POSIX.1b committee proposes a set of standard APIs for a real-time operating system interface; these include interprocess communication. This standard is formally known as the IEEE standard

1003.4-1993 [7]. Lastly, the POSIX.1c standard [8] specifies multithreaded programming interface. This is the newest POSIX standard and its details are described in the last chapter of this book.

Although much of the work of the POSIX committees is based on UNIX, the standards they proposed are for a generic operating system that is not necessarily a UNIX system. For example, VMS from the Digital Equipment Corporation, OS/2 from International Business Machines, and Windows-NT from the Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems. Most current UNIX systems, like UNIX System V release 4, BSD UNIX 4.4, and computer vendor-specific operating systems (e.g., Sun Microsystem's Solaris 2.x, Hewlett Packard's HP-UX 9.05 and 10.x, and IBM's AIX 4.1.x, etc.) are all POSIX.1-compliant but they still maintain their system-specific APIs.

This book will discuss the POSIX.1, POSIX.1b and POSIX.1c APIs, and also UNIX system-specific APIs. Furthermore, in the rest of the book, unless stated otherwise, when the word *POSIX* is mentioned alone, it refers to both the POSIX.1 and POSIX.1b standards.

To ensure a user program conforms to the POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before the inclusion of any headers) as:

```
#define _POSIX_SOURCE
```

or specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation:

```
%    CC -D_POSIX_SOURCE *.C
```

This manifested constant is used by *cpp* to filter out all non-POSIX.1 and non-ANSI C standard codes (e.g., functions, data types, and manifested constants) from headers used by the user program. Thus, a user program that is compiled and run successfully with this switch defined is POSIX.1-conforming.

POSIX.1b defines a different manifested constant to check conformance of user programs to that standard. The new macro is _POSIX_C_SOURCE, and its value is a timestamp indicating the POSIX version to which a user program conforms. The possible values of the _POSIX_C_SOURCE macro are:

_POSIX_C_SOURCE value	Meaning
198808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1 compliance
199309L	POSIX.1 and POSIX.1b compliance

Each _POSIX_C_SOURCE value consists of the year and month that a POSIX standard was approved by IEEE as a standard. The *L* suffix in a value indicates that the value's data type is a long integer.

The _POSIX_C_SOURCE may be used in place of the _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition. Thus, readers should browse the *unistd.h* header file on their systems and see which constants, or both, are used in the file.

There is also a _POSIX_VERSION constant that may be defined in the <unistd.h> header. This constant contains the POSIX version to which the system conforms. The following sample program checks and displays the _POSIX_VERSION constant of the system on which it is run.

```
/* show_posix_ver.C */
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE          199309L
#include <iostream.h>
#include <unistd.h>
int main()
{
#ifdef _POSIX_VERSION
      cout << "System conforms to POSIX: "
           << _POSIX_VERSION << endl;
#else
      cout << "_POSIX_VERSION is undefined\n";
#endif
      return 0;
}
```

In general, a user program that must be strictly POSIX.1- and POSIX.1b-compliant may be written as follows:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE   199309L
#include <unistd.h>
/* include other headers here */
int main()
{
      ...
}
```

1.4.1 The POSIX Environment

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

```
#include <header_file_name>
```

This method of referencing header files is adopted in POSIX. However, for each name specified in a *#included* statement, there need not be a physical file of that name existing on a POSIX-conforming system. In fact the data that should be contained in that named object may be builtin to a compiler, or stored by some other means on a given system. Thus, in a POSIX environment, included files are called simply *headers* instead of *header files*. This "headers" naming convention will be used in the rest of the book. Furthermore, in a POSIX-compliant system, the */usr/include* directory does not have to exist. If users are working on a non-UNIX but POSIX-compliant system, please consult the C or C++ programmer's manual to determine the standard location, if any, of the headers on the system.

Another difference between POSIX and UNIX is the concept of *superuser*. In UNIX, a superuser has privilege to access all system resources and functions. The superuser user ID is always zero. However, the POSIX standards do not mandate that all POSIX-conforming systems support the concept of a superuser, nor does the user ID of zero require any special privileges. Furthermore, although some POSIX.1 and POSIX.1b APIs require the functions to be executed in "special privilege," it is up to an individual conforming system to define how a "special privilege" is to be assigned to a process.

1.4.2 The POSIX Feature Test Macros

Some UNIX features are optional to be implemented on a POSIX-conforming system. Thus, POSIX.1 defines a set of feature test macros, which, if defined on a system, means that the system has implemented the corresponding features.

These feature test macros, if defined, can be found in the <unistd.h> header. Their names and uses are:

Feature test macro	Effects if defined on a system
_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 set-GID, so that it can change its effective user ID and group ID to those values via the *seteuid* and *setegid* APIs, respectively

Feature test macro	Effects if defined on a system
_POSIX_CHOWN_RESTRICTED	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 files on a system. If this constant is undefined in <unistd.h> header, users must use the *pathconf* or *fpathconf* function (described in the next section) to check the permission for changing ownership on a per-file basis
_POSIX_NO_TRUNC	If the defined value is -1, any long path name passed to an API is silently truncated to NAME_MAX bytes; otherwise, an error is generated. If this constant is undefined in the <unistd.h> header, users must use the *pathconf* or *fpathconf* function to check the path name truncation option on a per-directory basis
_POSIX_VDISABLE	If the 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. If this constant is undefined in the <unistd.h> header, users must use the *pathconf* or *fpathconf* function to check the disabling character option on a per-terminal device file basis

The following sample program prints the POSIX-defined configuration options supported on any given system:

```
/* show_test_macros.C */
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE    199309L
#include <iostream.h>
#include <unistd.h>
int main()
{
#ifdef _POSIX_JOB_CONTROL
      cout << "System supports job control\n";
#else
      cout << "System does not support job control\n";
#endif
```

```
#ifdef _POSIX_SAVED_IDS
    cout << "System supports saved set-UID and saved set-GID\n";
#else
    cout << "System does not support saved set-UID and "
         << " saved set-GID\n";
#endif

#ifdef _POSIX_CHOWN_RESTRICTED
    cout << "chown_restricted option is: " <<
                _POSIX_CHOWN_RESTRICTED << endl;
#else
    cout << "System does not support chown_restricted option\n";
#endif

#ifdef _POSIX_NO_TRUNC
    cout << "Pathname trunc option is: " << _POSIX_NO_TRUNC
         << endl;
#else
    cout << "System does not support system-wide pathname"
         << " trunc option\n";
#endif

#ifdef _POSIX_VDISABLE
    cout << "Disable char. for terminal files is: "
         << _POSIX_VDISABLE << endl;
#else
    cout << "System does not support _POSIX_VDISABLE\n";
#endif
    return 0;
}
```

1.4.3 Limits Checking at Compile Time and at Run Time

POSIX.1 and POSIX.1b define a set of system configuration limits in the form of manifested constants in the <limits.h> header. Many of these limits are derived from the UNIX systems and they have the same manifested constant names as their UNIX counterparts, plus the _POSIX_ prefix. For example, UNIX systems define the constant CHILD_MAX, which specifies the maximum number of child processes a process may create at any one time. The corresponding POSIX.1 constant is _POSIX_CHILD_MAX. The reason for defining these

constants is that although most UNIX systems define a similar set of constants, their values vary substantially from one UNIX system to another. The POSIX-defined constants specify the minimum values for these constants for all POSIX-conforming systems; thus, it facilitates application programmers to develop programs that use these system configuration limits.

The following is a 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 may be opened simultaneously by a process
_POSIX_STREAM_MAX	8	Maximum number of I/O streams that may be opened simultaneously by a process
_POSIX_ARG_MAX	4096	Maximum size, in bytes, of arguments that may be passed to an *exec* function call
_POSIX_NGROUP_MAX	0	Maximum number of supplemental groups to which a process may belong
_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 from or written to a pipe file
_POSIX_MAX_INPUT	255	Maximum capacity, in bytes, of a terminal's input queue
_POSIX_MAX_CANON	255	Maximum size, in bytes, of a terminal's canonical input queue
_POSIX_SSIZE_MAX	32767	Maximum value that can be stored in a *ssize_t*-typed object
_POSIX_TZNAME_MAX	3	Maximum number of characters in a time zone name

The following is a list of POSIX.1b-defined constants:

Compile time limit	Min. value	Meaning
_POSIX_AIO_MAX	1	Number of simultaneous asynchronous I/O
_POSIX_AIO_LISTIO_MAX	2	Maximum number of operations in one listio
_POSIX_TIMER_MAX	32	Maximum number of timers that can be used simultaneously by a process
_POSIX_DELAYTIMER_MAX	32	Maximum number of overruns allowed per timer
_POSIX_MQ_OPEN_MAX	2	Maximum number of message queues that may be accessed simultaneously per process
_POSIX_MQ_PRIO_MAX	2	Maximum number of message priorities that can be assigned to messages
_POSIX_RTSIG_MAX	8	Maximum number of real-time signals
_POSIX_SIGQUEUE_MAX	32	Maximum number of real time signals that a process may queue at any one time
_POSIX_SEM_NSEMS_MAX	256	Maximum number of semaphores that may be used simultaneously per process
_POSIX_SEM_VALUE_MAX	32767	Maximum value that may be assigned to a semaphore

Note that the POSIX-defined constants specify only the minimum values for some system configuration limits. A POSIX-conforming system may be configured with higher values for these limits. Furthermore, not all these constants must be specified in the <limits.h> header, as some of these limits may be indeterminate or may vary for individual files.

To find out the actual implemented configuration limits system-wide or on individual objects, one can use the *sysconf*, *pathconf*, and *fpathconf* functions to query these limits' values at run time. These functions are defined by POSIX.1; the *sysconf* is used to query general system-wide configuration limits that are implemented on a given system; *pathconf* and *fpathconf* are used to query file-related configuration limits. The two functions do the same thing; the only difference is that *pathconf* takes a file's path name as argument, whereas *fpathconf* takes a file descriptor as argument. The prototypes of these functions are:

```
#include <unistd.h>

long sysconf ( const int limit_name );
long pathconf ( const char* pathname, int flimit_name );
long fpathconf ( const int fdesc, int flimit_name );
```

The *limit_name* argument value is a manifested constant as defined in the <unistd.h> header. The possible values and the corresponding data returned by the *sysconf* function are:

Limit value	*sysconf* return data
_SC_ARG_MAX	Maximum size, in bytes, of argument values that may be passed to an *exec* API call
_SC_CHILD_MAX	Maximum number of child processes that may be owned by a process simultaneously
_SC_OPEN_MAX	Maximum number of opened files per process
_SC_NGROUPS_MAX	Maximum number of supplemental groups per process
_SC_CLK_TCK	The number of clock ticks per second.
_SC_JOB_CONTROL	The _POSIX_JOB_CONTROL value
_SC_SAVED_IDS	The _POSIX_SAVED_IDS value
_SC_VERSION	The _POSIX_VERSION value
_SC_TIMERS	The _POSIX_TIMERS value
_SC_DELAYTIMER_MAX	Maximum number of overruns allowed per timer
_SC_RTSIG_MAX	Maximum number of real time signals
_SC_MQ_OPEN_MAX	Maximum number of message queues per process
_SC_MQ_PRIO_MAX	Maximum priority value assignable to a message
_SC_SEM_MSEMS_MAX	Maximum number of semaphores per process
_SC_SEM_VALUE_MAX	Maximum value assignable to a semaphore
_SC_SIGQUEUE_MAX	Maximum number of real time signals that a process may queue at any one time
_SC_AIO_LISTIO_MAX	Maximum number of operations in one listio
_SC_AIO_MAX	Number of simultaneous asynchronous I/O

As can be seen in the above, all constants used as a *sysconf* argument value have the _SC_ prefix. Similarly, the *flimit_name* argument value is a manifested constant defined in the <unistd.h> header. These constants all have the _PC_ prefix. The following lists some of these constants and their corresponding return values from either *pathconf* or *fpathconf* for a named file object:

Limit value	*pathconf* return data
_PC_CHOWN_RESTRICTED	The _POSIX_CHOWN_RESTRICTED value
_PC_NO_TRUNC	Return the _POSIX_NO_TRUNC value
_PC_VDISABLE	Return the _POSIX_VDISABLE value
_PC_PATH_MAX	Maximum length, in bytes, of a path name
_PC_LINK_MAX	Maximum number of links a file may have
_PC_NAME_MAX	Maximum length, in bytes, of a file name
_PC_PIPE_BUF	Maximum size of a block of data that may be auto-

	matically read from or written to a pipe file
_PC_MAX_CANON	Maximum size, in bytes, of a terminal's canonical input queue
_PC_MAX_INPUT	Maximum capacity, in bytes, of a terminal's input queue

These variables parallel their corresponding variables as defined on most UNIX systems (the UNIX variable names are the same as those of POSIX, but without the _POSIX_ prefix). These variables may be used at compile time, such as the following:

```
char pathname [ _POSIX_PATH_MAX + 1];
for (int i=0; i < _POSIX_OPEN_MAX; i++)
   close (i);                                  // close all file descriptors
```

The following *test_config.C* program illustrates the use of *sysconf*, *pathconf*, and *fpathconf*:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>
int main()
{
        int res;
        if ((res=sysconf(_SC_OPEN_MAX))==-1)
           perror("sysconf");
        else cout << "OPEN_MAX: " << res << endl;

        if ((res=pathconf("/",_PC_PATH_MAX))==-1)
           perror("pathconf");
        else cout << "Max path name: " << (res+1) << endl;

        if ((res=fpathconf(0,_PC_CHOWN_RESTRICTED))==-1)
           perror("fpathconf");
        else
           cout << "chown_restricted for stdin: " << res << endl;
        return 0;
}
```

1.5 The POSIX.1 FIPS Standard

FIPS stands for Federal Information Processing Standard. The POSIX.1 FIPS standard was developed by the National Institute of Standards and Technology (NIST, formerly, the National Bureau of Standards), a department within the US Department of Commerce. The latest version of this standard, FIPS 151-1, is based on the POSIX.1-1988 standard. The POSIX.1 FIPS standard is a guideline for federal agencies acquiring computer systems. Specifically, 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; the _POSIX_JOB_CONTROL symbol must be defined
- Saved set-UID and saved set-GID; the _POSIX_SAVED_IDS symbol must be defined
- Long path name is not supported; the _POSIX_NO_TRUNC should be defined - its value is not -1
- The _POSIX_CHOWN_RESTRICTED must be defined - its value is not -1. This means only an authorized user may change ownership of files, system-wide
- The _POSIX_VDISABLE symbol must be defined - its value is not equal to -1
- 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. Thus, a FIPS 151-1 conforming system is also POSIX.1-1988 conforming, but not vice versa. The FIPS standard is outdated with respect to the latest version of the POSIX.1, and it is used primarily by US federal agencies. This book will, therefore, focus more on the POSIX.1 standard than on FIPS.

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 organization published the *X/Open Portability Guide*, issue 3 (XPG3) in 1989, and issue 4 (XPG4) in 1994. The portability guides specify a set of common facilities and C application program interface functions to be provided on all UNIX-based "open systems." The XPG3 [9] and XPG4 [10] are based on ANSI-C, POSIX.1, and POSIX.2 standards, with additional constructs invented by the X/Open organization.

CHAPTER 5

UNIX and POSIX APIs

UNIX systems provide a set of application programming interface functions (commonly known as *system calls*) which may be called by users' programs to perform system-specific functions. These functions allow users' applications to directly manipulate system objects such as files and processes that cannot be done by using just standard C library functions. Furthermore, many of the UNIX commands, C library functions, and C++ standard classes (e.g., the iostream class) call these APIs to perform the actual work advertised. Thus, users may use these APIs directly to by-pass the overhead of calling the C library functions and C++ standard classes, or to create their own versions of the UNIX commands, C library functions and C++ classes.

Most UNIX systems provide a common set of APIs to perform the following functions:

- Determine system configuration and user information
- Files Manipulation
- Processes creation and control
- Interprocess communication
- Network communication

Most UNIX APIs access their UNIX kernel's internal resources. Thus, when one of these APIs is invoked by a process (a process is a user's program under execution), the execution context of the process is switched by the kernel from a user mode to a kernel mode. A user mode is the normal execution context of any user process, and it allows the process to access its process-specific data only. A kernel mode is a protective execution environment that allows a user process to access kernel's data in a restricted manner. When the API execu-

tion completes, the user process is switched back to the user mode. This context switching for each API call ensures that processes access kernel's data in a controlled manner, and minimizes any chance of a run-away user application may damage an entire system. In general, calling an API is more time-consuming than calling a user function due to the context switching. Thus, for those time-critical applications, users should call their system APIs only if it is absolute necessary.

5.1 The POSIX APIs

Most POSIX.1 and POSIX.1b APIs are derived from UNIX APIs. However, the POSIX committees do create their own APIs when there is perceived deficiency of the UNIX APIs. For example, the POSIX.1b committee creates a new set of APIs for interprocess communication using messages, shared memory, and semaphores. There are equivalent constructs for messages, shared memory, and semaphores in System V UNIX, but the latter constructs use a nonpathname-naming scheme to identify these IPC facilities, and processes cannot use these IPCs to communicate across a LAN. Thus, the POSIX.1b committee created a different version of messages, shared memory, and semaphores that eliminated these short-coming.

In general, POSIX APIs uses and behaviors are similar to those of UNIX APIs. However, users' programs should define the _POSIX_SOURCE (for POSIX.1 APIs) and/or _POSIX_C_SOURCE (for both POSIX.1 and POSIX.1b APIs) in their programs to enable the POSIX APIs declarations in header files that they include.

5.2 The UNIX and POSIX Development Environment

The <unistd.h> header declares some commonly used POSIX.1 and UNIX APIs. There is also a set of API-specific headers placed under the <sys> directory (on a UNIX system it is the /usr/include/sys directory). These <sys/...> headers declare special data types for data objects manipulated by both the APIs and by users' processes. In addition to these, the <stdio.h> header declares the *perror* function, which may be called by a user process whenever an API execution fails. The *perror* function prints a system-defined diagnostic message for any failure incurred by the API.

Most of the POSIX.1, POSIX.1b, and UNIX API object code is stored in the *libc.a* and *libc.so* libraries. Thus, no special compile switch need be specified to indicate which archive or shared library stores the API object code. However, some network communication APIs' object code is stored in special libraries on some systems (e.g., the socket APIs are stored in *libsocket.a* and *libsocket.so* libraries on Sun Mircosystems Solaris 2.x system). Thus, users should consult their system programmer's reference manuals for the special header and library needed for the APIs they use on their systems.

5.3 API Common Characteristics

Although the POSIX and UNIX APIs perform diverse system functions on behalf of users, most of them returns an integer value which indicates the termination status of their execution. Specifically, if a API returns a -1 value, it means the API's execution has failed, and the global variable *errno* (which is declared in the <errno.h> header) is set with an error code. A user process may call the *perror* function to print a diagnostic message of the failure to the standard output, or it may call the *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 (e.g., output to a error log file).

The possible error status codes that may be assigned to *errno* by any API are defined in the <errno.h> header. When a user prints the man page of a API, it usually shows the possible error codes that may be assigned to *errno* by the API, and the reason why. Since this information is readily available to users and they may be different on different systems, this book will not describe the *errno* values for individual API in any details. However, the following is a list of commonly occur error status codes and their meanings:

Error status code	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 privilege
ENOENT	An invalid file name was specified to an API
EBADF	A API was called with an invalid file descriptor
EINTR	A API execution was aborted due to a signal interruption (see Chapter 9 for the explanation of signal interruption)
EAGAIN	A API was aborted because some system resource it requested was temporarily unavailable. The API should be called again later.
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 arguments
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