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 provide 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* lptr );
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 time-stamp 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.

In addition to the above, in 1993 a group of computer vendors (e.g., Hewlett-Packard, International Business Machines, Novell, Open Software Foundation, and Sun Microsystems, Inc.) 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 the vendors. This specification is known as *Spec 1170* and has been incorporated into XPG4 as part of the X/Open Common Application Environment (CAE) specifications.

The X/Open CAE specifications have a much broader scope than do the POSIX and ANSI-C standards. This means applications that conform to ANSI-C and POSIX also conform to the X/Open standards, but not necessarily vice versa. Furthermore, though most computer vendors and independent software vendors (ISVs) adopted POSIX and ANSI-C, some of them have yet to conform to the X/Open standards. Thus, this book will focus primarily on the common UNIX system programming interface and the ANSI-C and POSIX standards. Readers may consult more detailed publications [4,5] for further information on the X/Open CAE specifications.

1.7 Summary

This chapter gave an overview of the various standards that are applicable to UNIX system programmers. The objective is to familiarize readers with these standards and to help readers understand the benefits they provide. The details of these standards and their corresponding functions and APIs, as provided on most UNIX systems, are described in the rest of the book.

1.8 References

[1]. O-J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Common Base Language*, 1970.

[2]. Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, 1991.

[3]. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[4]. Andrew Koenig, *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++ (Committees: WG21/N0414, X3J16/94-0025)*, 1994.

[5]. American National Standard Institute, *American National Standard for Information Systems - Programming Language C, X3.159 - 1989*, 1989.

[6]. Institute of Electrical and Electronics Engineers, *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language], IEEE 1003.1.* 1990.

[7]. Institute of Electrical and Electronics Engineers, *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language] - Amendment: Real-Time Extension, IEEE 1003.1b.* 1993.

[8]. Institute of Electrical and Electronics Engineers, *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language] - Amendment: Thread Extension, IEEE 1003.1c.* 1995.

[9]. X/Open, *X/Open Portability Guide,* Prentice Hall, 1989.

[10]. X/Open, *X/Open CAE Specification, Issue 4*, Prentice Hall, 1994.

CHAPTER 2

C++ Language Review

This chapter reviews the essential constructs of the C++ language, which is based on the draft ANSI/ISO C++ standard [1]. The readers are assumed to be familiar with the C++ language, at least at the beginner's level. This chapter gives a quick review of the C++ programming techniques, so as to refresh the reader's memory. It also describes the template classes and exception handling. These latter subjects are new features to the C++ language and defined by the ANSI/ISO C++ standard. Readers who need a more detailed reference on the C++ language programming may consult [2,3].

Besides describing the C++ language constructs, this chapter also covers the C++ standard I/O classes and object-oriented design techniques. The standard I/O classes are powerful and rich in functionality. They essentially replace the C stream I/O functions and the strings function. It is important for readers to know these I/O classes to maximize code reuse and to reduce their application development time and costs.

Object-oriented programming techniques enable users to move from algorithmic program designs to object-based program designs. When doing object-oriented programming, users are more concerned with the types of objects that their programs have to deal with, the properties of these objects, and how they interact with each other and with users. Object-oriented programming techniques are valuable for database and GUI applications and are also useful in encapsulating low-level network communication protocol to provide a higher level interface for network-based application developers. The latter chapters in this book will show how this is done.

2.1 C++ Features for Object-Oriented Programmıng

C++ supports class declarations. Classes are used to construct user-defined data types. Each class encapsulates the data storage and legal operations of any object of that data type. Thus, C++ programs spend less time in the traditional algorithmic design for their applications but put more effort in designing classes and management of class objects and their interactions.

A class provides data hiding such that the internal data can be classified as "public," "private," and "protected." Class "public" data are accessible by any user functions which are not defined inside the class. Class "private" data cannot be accessed by anyone except member functions defined in the same class. Finally, class "protected" data are "private" to all user functions, but "public" to all its own and subclasses' member functions. The elaborate scheme of classifying class data is to allow developers to control the access and manipulation of class objects' data. This prevents class objects' data being changed anywhere in user programs. Furthermore, any changes to a class private and protected data will have minimum impact on user functions, provided the member functions used to access those data remain the same.

Furthermore, a class imposes a well-defined interface for objects of that type to interact with the rest of the world. This allows users to change the internal implementation of any class while maintaining the working order of the rest of the program (as long as the interface of the class remains unchanged). This renders well-designed C++ programs that are easy to maintain or change.

Another advantage of classes is that they promote code sharing. Specifically, a new class may be "derived" from one or more existing classes, and the new class contains all the data storage and functions of its derived class(es). Furthermore, the new class may define additional data and functions that are unique to objects of that new type, and it may even redefine the functions it inherits from its base class(es). Thus, class inheritance provides maximum flexibiiity in generating new classes that are similar, but not identical, to existing classes.

Like other object-oriented languages, C++ supports *constructor* and *destructor* functions for classes. These ensure that objects are properly initialized when created and that data is cleaned up when being discarded. Moreover, C++ defines the *new* and *delete* operators for objects to allocate and deallocate dynamic memory. However, unlike other object-oriented languages, there is no built-in garbage collection to manage dynamic memory used by objects, and *constructor* and *destructor* functions are not mandated for all defined classes. These relaxations are done to reduce the overhead of C++ programs' run-time performance but require developers to be disciplined in crafting their programs.

ANSI/ISO C++ supports template classes and functions. These allow users to create and debug some generic classes and functions. Once these are done, they may safely derive real classes and functions that work with different data types. This saves substantial program development and debug time. In addition to this, C++ defines a formal method of exception handling. This maintains consistent methods for all C++ applications to handle exceptions that may occur in their programs.

All in all, object-oriented programming strives to achieve the following goals:

- Data abstraction to ensure a well-defined interface for all objects
- Class inheritance to promote code reuse
- Polymorphism such that classes derived from other classes may have different data and functions
- Modeling of objects and their interactions after real-life situations

All the above objectives are supported by C++, and it is also backward-compatible with C. Thus it allows C programmers to start using C++ with their C programs, then migrate toward using C++ and object-oriented constructs in their programs at a pace at which they are comfortable.

2.2 C++ Class Declaration

A C++ class represents an abstract data type. It consists of data members and functions. These, in turn, may be classified as private, public, and protected. Private data members and functions are accessible via the same class member functions only, whereas public data members and functions are accessible by any other objects. These public data members and functions form the external interface to the world for objects of the class. Protected data members and functions are like their private counterparts but are also accessible to subclasses' member functions.

When a class data member or function is referenced anywhere outside the class declaration, the "::" scope resolution operator should be used to qualified their names. Specifically, the name appearing on the left of the "::" operator is a class name, and the name appearing on the right is a variable or function defined in the class. For example, *menu::num_fields* refers to the *num_fields* data member of the class *menu*. In addition to this, if no name appears on the left of a "::", it means that the name specified on the right is a global variable or function. For example, if there is a global variable called *x*, and in a class declaration there is also a data member called *x*, the member functions of that class may reference the global variable *x* by *::x*, and the data member *x* by either *x* or *<class_name>::x*.

The following *menu.h* header declares a class called *menu*:

```
#ifndef MENU_H
#define MENU_H
```

```
class menu
{
    private:
        char* title;
    protected:
        static int num_fields;
    public:
        // constructor function
        menu( const char* str )
        {
            title = new char[strlen(str)+1];
            strcpy (title, str );
            num_fields = 0;
        };
        // constructor function
        menu()
        {
            title = 0;
            num_fields = 0;
        };
        // destructor function
        ~menu()
        {
            delete title;
        };
        void incr_field( int size=1 )
        {
            num_fields+= size
        };
        static int fields()
        {
            return num_fields;
        };
        char* name()
        {
            return title;
        };
    };
    #endif              /* menu.h */
```

In the above, *menu::num_fields* and *menu::fields()* are declared as static. Unlike non-static data members where each object of a class has its own private copy of those members, there is only one instance of each static data member for all objects. Static data members are used like global variables by all objects of the same type. The accessibility of static data members by objects of other classes is determined by whether the data members are declared as private, protected, or public.

Static data members must be defined in one of the C source modules of a program if they are being used in a program. For example, the following module defines the *menu::num_fields* data member with an initial value of zero:

```
// module name: a.C
#include <string.h>
#include "menu.h"
int menu::num_fields = 0;
int main()                {... .}
```

Note that in the above example, although *menu::num_fields* is a protected data member, it is legal to define and initialize it in a program scope. However, further modification of that variable in user programs must be done only via the *menu* class member functions, subclass functions, or friend functions. The same also applies for private static data members.

Static member functions can access only static data members in a class. Thus in the *menu* class, the *menu::fields()* cannot access the *menu::title* data member. Whereas nonstatic member functions must be called via objects of a class, static functions have no such restriction:

```
menu abc ("Example");
abc.incr_field( 5 );
cout << "static func. called independent of objects: "
     << menu::fields() << endl;
cout << "Static func. can also be called via object: "
     << abc.fields() << endl;
```

Static data members and functions are commonly used to track how many objects of a class have been created as well as other general statistics, They can also be used to manage dynamic memory created by all objects of a class and to do garbage collection.

The *menu::menu* functions are constructors. A constructor function is called when an object of a class is created, and it initializes the data members of a newly created object. A constructor function may be overloaded, which means multiple constructor functions with different signatures may be defined in a same class. For example, the following object definitions use different constructor functions:

```
menu abc;                // use menu::menu()
menu xyz ("Example");    // use menu::menu( const char* str )
```

The *menu()::~menu()* is a destructor function. It is called when an object of a class is going out of scope to ensure proper cleanup of the object's data (e.g., deallocate dynamic memory used by the object). A destructor function cannot be overloaded, and it accepts no argument.

In the menu class example, all member function definitions are placed in the class declaration. This means these member functions are to be used as inline functions, or like macros, in C terminology. The advantage of using inline functions is to improve program performance by eliminating the overhead of function calls. The disadvantage of inline functions is that any changes made in an inline function require source modules that reference the function to be recompiled.

A user may declare class member functions as non-inline by placing their definitions in a separate source module. For example, *menu.h* may be changed to the following:

```
#ifndef MENU_H
#define MENU_H
class menu
{
    private:
        char* title;
    protected:
        static int num_fields;
    public:
        // constructor function
        menu( const char* str );
        menu();
        // destructor function
        ~menu();
        void incr_field( int size=1 );
        static int fields();
        char* name() ;
};
#endif
```

Then a separate C++ module, for example, *menu.C*, must be created to define the actual member functions:

```
// source file name: menu.C
#include <string.h>
#include "menu.h"
// a constructor with an argument
menu::menu ( const char* str )
{
        title = new char[strlen(str)+1];
        strcpy (title, str );
        num_fields = 0;
}
// a constructor with no argument
menu::menu()
{
        title = 0;
        num_fields = 0;
}
// a destructor
menu::~menu()
{
        delete title;
}
// Note:the siz argument cannot have default value here
void menu::incr_field ( int size )
{
        num_fields += size;
}
// a static member function
int menu::fields()
{
        return num_fields;
}
// another non-static member function.
char*menu:: name()
{
        return title;
}
```

Any program that uses the menu class must be compiled with *menu.C* to create an executable object. For example, given the following *test_menu.C* file:

```
// source test_menu.C
#include <iostream.h>
#include "menu.h"
int menu::num_fields = 0;
int main()
{
      menu abc ("Test");
      cout << abc.name() << endl;
      return menu::fields();
}
```

the *test_menu.C* is compiled as shown below to create an executable program *a.out*:

```
%     CC test_menu.C menu.C
%     a.out
Test
```

Finally, notice that the *menu::incr_field* function declaration in the *menu.h* has a default value for the *size* argument, but in the *menu.C* file the *menu::incr_field* function definition is not allowed to specify a default value for the *size* argument. In C++ 1.0, this was allowed, but it may lead to inconsistency in assigning default values to function arguments. Thus, this practice is no longer allowed.

2.3 Friend Functions and Classes

The friend construct in C++ allows designated functions and member functions of other classes to directly access private and protected data members of a class. This is for special occasions where it is more efficient for a function to directly access an object's private data than to go through its class member function. For example, the << operator function is commonly declared as a friend function to classes so that one can print objects of these classes as if they were of the basic data types (e.g., *int, double*).

Since friend functions can directly access and change private data of objects of a class, the compiler must be told that these are special, authorized functions. Thus, their names must be listed as *friend* in the class declaration. Furthermore, it serves as a reminder to users that whenever the class declaration is changed, all of its friend functions may require modification accordingly.

The following example illustrates the use of a friend function and a friend class:

```
// source module: friend.C
#include <iostream.h>
int year;
class foo;
class dates
{
        friend ostream& operator<<(ostream&,dates&);
        int year, month, day;
  public:
        friend class foo;
        dates() { year=month=day = 0; };
        ~dates() {};
        int sameDay(int d) const { return d==day; };
        void set(int y) const { ::year = y; };
        void set(int y) { year = y; };
};

class foo
{
  public:
        void set(dates& D, int  year) {D.year = year; };
};

ostream& operator<<(ostream& os, dates& D)
{
  os << D.year << " ," << D.month << " ," << D.day;
  return os;
}

int main()
{
        dates Dobj;
        foo Fobj;
        Fobj.set(Dobj, 1998);
        clog << "Dobj: " << Dobj << '\n';
}
```

In the above example, the << operator and class *foo* are declared as friends of the class *dates*. This means that the << function and class *foo*'s member functions can access the private data members of any objects of class *dates*. The compilation and sample output of the program is:

```
%    CC friend.C
%    a.out
Dobj: 1998, 0, 0
```

2.4 Const Member Functions

Const member functions are special member functions that cannot modify any data members in their class. They are designed to accommodate class objects that are defined as *const*. Specifically, a const object can invoke only its class's const member functions. This guarantees that the object's data is not being modified.

The C++ compiler flags an error when a const object invokes a nonconst member function. The only exception to this is that nonconst constructors and destructors may be applied to const objects. Finally, const and nonconst member functions with the same signatures may be overloaded.

The following example illustrates the definition of const member functions and their usage:

```
// source module: const.C
static int year = 0;
class dates
{
          int year, month, day;
     public:
          dates() { year=month=day = 0; };
          ~dates() {};
          void set(int y) { year = y; };
          // const member functions
          void print( ostream& = cerr ) const;
          int sameDay(int d) const { return d==day; };
          // note: this function is overloaded
          void set(int y) const { ::year = y; };
};
```

```
void dates::print( ostream& os ) const
{
        os << year << "," << month << "," << day << '\n';
}

int main()
{
        const dates foo;                // const object
        dates foo1;                     // non-const object
        foo.set(1915);                  // ::year = 1915
        foo.print();                    // year=0, month=0, day=0
        foo1.set(25);                   // foo1.year=25
        foo1.print();                   // year=25,month=0,day=0
}
```

In the above example, *foo* is a const object and *foo1* is a nonconst object. Both objects are initialized via the *dates::dates* constructor. The *foo.set(1915)* statement invokes the const member function *dates::set*, whereas the *foo1.set(25)* statement invokes the nonconst member function *dates::set*. The *foo.print()* and *foo1.print()* statements both invoke the const member function *dates::print*. This is fine, as nonconst objects can always invoke const member functions, but not vice versa.

The compilation and sample output of the program is:

```
%   CC const.C
%   a.out
0, 0, 0
25, 0, 0
```

2.5 C++ Class Inheritance

Class inheritance allows a class to be derived from one or more existing classes. The new class is called a subclass, and the class(es) it derived from is called a *base class* or *super class*.

A subclass inherits all data members and functions of its base class(es), and it may access all protected and public data members and functions of its base class. Furthermore, a subclass may define additional data members and functions that are unique to itself.

The following *window.h* header declares a *window* class, which is a subclass of *menu*:

```
#ifndef WINDOW_H
```

```
#define WINDOW_H
#include "menu.h"
#include <iostream.h>
class window : public menu
{
    private:
        int xcord, ycord;
    public:
        // constructor function
        window( const int x, const int y, const char* str ) : menu(str)
        {
            xcord = x;
            ycord = y;
        };
        // destructor
        ~window() {};
        // window-specific function
        void show (ostream& os )
        {
            os << xcord << ',' << ycord << " => " << name() << endl;
        };
};
#endif
```

In a subclass declaration, a base class name may be preceded by a "public" or "private" key word, which means that the base class public data members and functions are to be treated as "public" or "private," respectively, in the subclass. If no such key word is specified, the default is "private."

A subclass member function can directly access only the protected and public data members of its base class(es). A subclass can explicitly mark selected "public" or "protected" data members and/or functions of "private" base classes to be "public" or "protected," respectively, in the subclass.

For example, the following *class window2* makes all the menu data members and functions it inherits private except the *menu::num_fields*, which is treated as protected in this subclass:

```
class window2 : private menu
{
    private:
            int xcord, ycord;
        protected:
            menu::num_fields;          // make menu::num_field protected
        public:
            ...
    };
```

When an object of a subclass is defined, the calling sequence of the subclass and base class(es) constructor function is in the order shown below:

- Base class constructors, in the order listed in the subclass declaration
- Data member constructors, in the order declared in the subclass
- Subclass constructors

For example, given the following subclass declaration:

```
class a, b;
class base1, base2;
class sub : public base1, private base2
{
      a var1;
      b var2;
   public:
      ...
};
sub foo;
```

the invocation order of constructor functions for the variable *foo* is: *base1::base1, base2::base2, a::a, b::b*, and, finally, *sub::sub.*

When an object of a subclass is out of scope, the calling sequence of the subclass and base class(es) destructor functions is in the order shown below:

- Subclass destructor
- Data member destructors, in reverse order as to that declared in the subclass
- Base class destructors, in the reverse order to that listed in the subclass declaration

In the above example, if the variable *foo* is going out of scope, the invocation order of destructor functions for it is: *sub::~sub, b::~b, a::~a, base2::~base2,* and, finally, *base1::~base1.*

When a subclass's constructor is called, the data to be passed on to its base classes' constructors and its data members' constructors are specified in a subclass constructor's initialization list. The list is specified after the subclass constructor function argument list but before the function body definition:

```
<subclass>::<subclass> ( <arg_list> ) : <initialization list>
{
    /* body */
}
```

and <initlialization list> is:

```
<class_name> ( <arg> ) [ , <class_name> ( <arg> ) ]+
```

For example, the *class sub* constructor function may be written as:

```
sub::sub( int x, int y, int z ) : base1(a), base2(b), a(z), b(z=1)
{
    /* body */
}
```

An initialization list can be specified only in a subclass constructor function definition, not at its declaration. Furthermore, one can skip specifying a base class or a data member name in an initialization list if that class does not have a constructor defined or it has a constructor defined that does not take any argument.

2.6 Virtual Functions

A class member function may be declared with a "virtual" key word. This means that any sub-classes of this class may redefine this virtual function. This is C++'s way of supporting polymorphism in object-oriented programming. A subclass may or may not redefine virtual functions that it inherits from its base classes. If it does redefine any virtual function, it cannot change the signature of these functions.

Virtual functions are used to define common operations for a set of related classes. The interface of these operations is the same for all of those classes, but the actual behavior (or implementation) of these operations may be changed per class. For example, a base class menu may define a draw operation that draws a form on a console, and its subclass window may redefine the draw operation to draw a window and then a menu on a console.

Constructor functions may not be declared as virtual functions. Destructor and overloaded operator functions can and should be declared as virtual.

The following example illustrates uses of virtual functions:

```
// source module: virtual.C
#include <iostream.h>
class date
{
        int year, month, day;
    public:
        date(int y, int m, int d)   { year=y; month=m; day=d; };
        virtual ~date() {};
        virtual void print() {cerr << year << '/' << month << '/'
                                 << day << "\n";};
        virtual void set (int a, int b, int c) { year=a;  month=b;  day=c; };
};
class derived : public date
{
        int x;
    public:
        derived (int a,int b,int c,int d): date(a,b,c), x(d) {};
        ~derived() {};
        void print() { date::print(); cout << "derived: x=" << x << "\n"; };
        virtual void set(int a, int b, int c) {x=a;};
};

int main()
{
        date foo(1991,5,4);
        derived y(1,2,3,4);
        date* p = &y;
        p->print();                // derived::print()
        p = & foo;
        p->print();                // date::print()
}
```

In the above example, the *date::~date, date::print*, and *date::set* are all declared as virtual functions. The *class derived* redefines the *print* and *set* virtual functions. In the *main* function, when the variable *p* is pointing to *y*, the *p->print()* statement actually invokes the *derived::print* function, and when *p* is pointing to *foo*, the *p->print()* statement invokes the *date::print* function. Had the *date::print* function not been declared as virtual, then both the *p->print*() statements in the *main* function would have invoked only the *date::print* function, and the *derived::print* function would have been treated as an overloaded function of the *date::print* function.

The compilation and sample output of the program is:

```
%    CC virtual.C
%    a.out
1/2/3
derived: x=4
1991/5/4
```

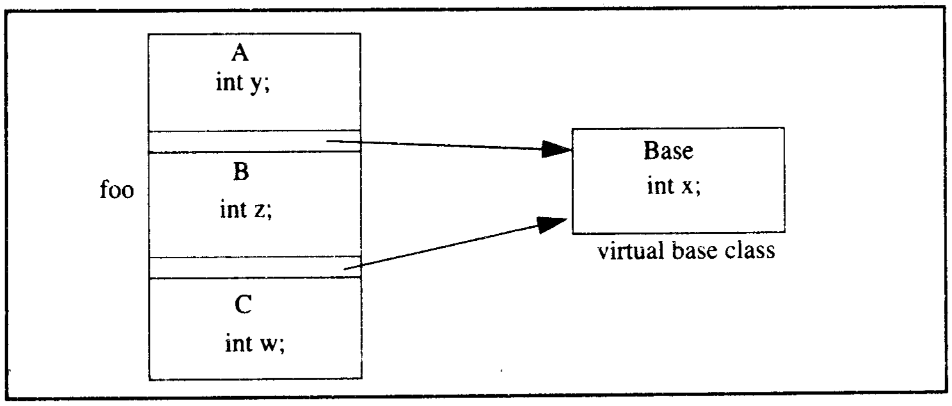
2.7 Virtual Base Classes

Assume that a class *A* and a class *B* are both derived from a class *Base*. If then a class *C* is derived from both class *A* and class *B*, each object of class *C* has two copies of data members of class *Base*, and this may not be desirable for an application. To ensure, in such a multiple inheritance situation, that only one copy of class *Base* data members are kept for every object of class *C*, class *Base* must be declared as "virtual" in both class *A* and class *B* declarations, as illustrated below:

```
class Base
{
     int x;
     ...
};

class A : virtual public Base
{
     int y;
     ...
};

class B: virtual public Base
{
     int z;
```

```
          ...
    };

    class C : public A, private B
    {
          int w;
          ....
    };
```

A layout of the storage of an object (for example *foo*) of class *C* is similar to the following:



A virtual base class must define a constructor function that either takes no argument or has default values for all its arguments. A virtual base class is initialized by its most derived class (e.g., class *Base* is initialized via class *C*, not by class *A* or by class *B*). If the most derived class does not explicitly initialize the virtual base class, then the virtual base class's constructor, which does not require function arguments, is invoked to initialize objects of the most derived class. Furthermore, virtual base class constructors are invoked before nonvirtual base class constructors, and virtual base class destructors are invoked after nonvirtual base class destructors.

If there is a public virtual base class and a private virtual base class in a subclass, then the virtual base class is treated as public in the most derived class. For example, the class *Base* is treated as a public base class in class *C* in the above example.

The following shows the calling sequences of virtual and nonvirtual base class constructor and destructor functions, and the invocation of a virtual base class function (*base::print*) in a most derived class object:

```
// source module: virtual_base.C
#include <stream.h>

class base
{
   public:
      int x;
      base(int xa=0) : x(xa) { cerr << "base(" << xa << ") called\n"; };
      virtual void print()      { cerr << "x=" << x << "\n"; };
      virtual ~base()           {cerr << "~base() called\n";}
};

class d1 : virtual public base
{
   public:
      int y;
      d1(int xa, int ya) : base(xa)
      {
        cerr<<"d1("<<xa<<","<<ya<<") called\n";   y = ya;
      };
      ~d1()                     { cerr << "~d1() called\n"; };
      void print()              { cerr << "y=" << y << "\n"; };
};

class d2 : virtual public base
{
   public:
      int z;
      d2(int xa, int za) : base(xa)
      {
         cerr<<"d2("<<xa<<","<<za<<") called\n";   z = za;
      };
      ~d2()                     { cerr << "~d2() called\n"; };
};


class derived : public d1, public d2
{
   public:
          int all;
          derived(int a, int b, int c, int d) : base(a), d1(a,b), d2(a,c), all(d)
                                    {   cerr << "derived(" << all << ") called\n";};
          void prints()             { base::print();cerr << "all=" << all << "\n";};
};

int main()
{
          derived foo(1,2,3,4);
          foo.prints();
}
```

The compilation and sample output of the program is:

```
%     CC virtual_base.C
%     a.out
base(1) called
d1(1,2) called
d2(1,3) called
derived(4) called
x=1
all=4
~d2() calleo
~d1() called
~base() called
```

2.8 Abstract Classes

An abstract class is designed to be a framework for derivation of subclasses. It has an incomplete specification of its operations; thus, no objects should be defined for any abstract class. The C++ compiler enforces such an restriction.

An abstract class declares one or more pure virtual functions; these functions have no definitions, and all sub-classes of the abstract class must redefine these functions. A pure virtual function is declared as shown below:

```
class abstract_base
{
          public:
            virtual void draw() = 0;          // a pure virtual function
    };
```

An abstract class must contain at least one pure virtual function declaration. Users may define abstract class-typed pointer or reference variables, but these variables must reference subclass objects only.

The following example illustrates the uses of an abstract class. The program is an interactive program that displays a menu for the user to select operations. Each operation is encapsulated by one *menu_obj*-typed object. The *menu_obj* class is derived from the *abstract_base* class. The latter is an abstract base class and contains two pure virtual functions: *info* and *opr*. These pure virtual functions are re-defined in the *menu_obj* class, and the *info* function is called to display the use of a *menu_obj* class object, while the *opr* function is called to actually execute the function of a *menu_obj*-typed object.

In the example, there are three *menu_obj* class objects defined: one is to display a local date/time, the second one is to display Greenwich Mean Time date/time, and the last one is to terminate the program. These objects are stored in a dispatch table called *menu*. The *main* function iteratively calls the *menu_obj::info* functions of all objects stored in *menu* to display a menu to console. It then gets an input selection from a user and invokes the *menu_obj::opr* function of a corresponding object. Note that the program is extensible, in that users may define more *menu_obj* class objects and store them in *menu*, and the program will automatically include those objects in action:

```
// source module: abstract.C
#include <iostream.h>
#include <time.h>
#include <string.h>
typedef void (*PF)();

class abstract_base                  // abstract base class
{
  protected:
    PF     fn_ptr;
    char   *info_msg ;               // for displaying information
  public:
    abstract_base(PF fn=0, char* msg=0) {fn_ptr=fn,  info_msg=msg; };
    virtual void info(int) =0;
    virtual void opr() =0;
};
```

```
class menu_obj : public abstract_base // derived class
{
   public:
      menu_obj(PF fn, char *msg) : abstract_base(fn,msg) {};
      void info(int menu_idx)
      {
            cout << menu_idx << ": " << info_msg << "\n";
      };
      void opr()                  { this->fn_ptr(); };
};

inline void fn0()
{
      long tim = time(0);
      cerr<<"Local: "<<asctime(localtime(&tim))<<"\n";
}

inline void fn1()
{
      long tim = time(0);
      cerr<<"GMT: "<<asctime(gmtime(&tim))<<"\n";
}

inline void fn2()                 { exit(0); }

// dispatch table
 menu_obj menu[] =
{
      menu_obj(fn0, "Local data:time"),
      menu_obj(fn1,"GMT date:time"),
      menu_obj(fn2,"Exit program")
};

#define     MENU_SIZ      sizeof(menu)/sizeof(menu[0])

inline void display_menu()
{
      for (int i=0; i<MENU_SIZ; i++) menu[i].info(i);
}
```

```
int main()
{
     for (int idx; 1; )
     {
          display_menu();
          cout << "Select (0-" << (MENU_SIZ-1) << ")> ";
          cin>>idx;
          if (idx >=0 && idx<MENU_SIZ)
             menu[idx].opr();
          else cerr << "Illegal input: " << idx << "\n";
     }
}
```

The compilation and sample output of the program is:

```
%    CC abstract_base.C
%    a.out
0: Local date:time
1: GMT date:time
2: Exit Program
Select (0-2): 0
Local: Fri Apr 12 19:38:21 1991

0: Local date:time
1: GMT date:time
2: Exit Program
Select (0-2): 1
GMT: Sat Apr 13 02:38:22 1991

0: Local date:time
1: GMT date:time
2: Quit
Select (0-2): 2
```

2.9 The *new* and *delete* Operators

C++ defines the *new* and *delete* operators for dynamic memory management. These operators are supposed to be more efficient than the C *malloc*, *calloc*, and *free* functions for dynamic memory management.

The argument to a *new* operator is a data type (or class) name and, optionally, a parentheses-enclosed initialization data list for the new object's constructor function. If no initialization data is specified, either the new object's constructor function, which does not take argument, is invoked -- if it is defined, or the new object is not initialized.

For example, given the following class declaration:

```
class date {
      int year, month, day;
   public:
      date( int a, int b, int c ) { year=a, month=b, day=c; };
      date() { year = month = day = 0; };
      ~date() {};
};
```

The following two statements use two dıtterent *class date* constructors:

```
date *date1p = new date (1995,7,1);      // use date::date(int,int,int);
date *date2p = new date;                 // use date::date();
```

An array of objects may be allocated via the *new* operator. This is done by specifying a class name followed by the number of objects in the array and enclosing that number in brackets. For example, the following statement allocates an array of ten *class date*-type objects:

```
date *dateList = new date [ 10 ];
```

To initialize objects in an array that are allocated via *new*, the object's class should have a constructor that requires no arguments, and this constructor is used to initialize every object in the array. If no such constructor is defined, the objects in the array are not initialized.

There is a global variable, *_new_handler*, defined in the standard C++ library. If this *_new_handler* variable is set to a user-defined function, then whenever the *new* operator fails, it calls this routine to do user-defined error recovery actions. It then returns a NULL pointer value to its caller. If the *_new_handler* is set as its default value NULL, then when the *new* operator fails, it simply returns a NULL pointer to its caller.

The *_new_handler* declaration is defined in the <new.h> header as:

```
extern void (*_new_handler)();
```

It can be set either by direct assignment in users' programs or by the *set_new_handler* macro, as defined in the <new.h>:

```
#include <new.h>
extern void error_handler();                // user-defined function
main()
{
        _new_handler = error_handler;       // direct assignment
        set_new_handler ( error_handler );  // assigned via a macro
}
```

Finally, the *new* operator may be instructed to use a preallocated memory region to place "dynamic" objects on it. In this case, a user takes over the memory allocation task, and the *new* operator is used to initialize the new objects that are placed on the user-specified memory region. The following example shows how this is done:

```
#include <new.h>
#include "date.h"

const NUM_OBJ = 1000;
date *pool = new char[sizeof(DATE) * NUM_OBJ];
int main()
{
        date *p = new (pool) date [NUM_OBJ];
        delete [NUM_OBJ] p;
}
```

In the above example, a user allocates a memory region pointed to by the *pool* variable. The user then "allocates" NUM_OBJ objects of *class date*. This array is placed in the memory region referenced by *pool,* and the variable *p* points to the array. Finally, the array is deallocated via the *delete* operator.

A dynamic object allocated via the *new* operator should be deallocated via the *delete* operator. For example, to delete a *date* class object whose address is pointed to by a variable called *p*:

```
date *p = new date;
...
delete p;
```

If the object to be deleted is an array, the *delete* operator should be followed by a specification of the number of entries in the array, and then the array name. For example:

```
date *arrayP = new date[10];
...
delete [10] arrayP;
```

The above syntax is needed, as it causes the destructor function of objects in the array to be invoked for each of the objects. If the above array is deallocated as:

```
delete arrayP:
```

Then the destructor function is called for the first object in the array only.

The *new* and *delete* operators may be overloaded in a class; then, whenever an object of such a class is allocated via the *new* and deallocated via the *delete* operators, the class instance of these operators is used instead.

The overloaded *new* and *delete* operators must be declared as class member functions. They are treated as static member functions, in that they cannot modify any data member of objects in their classes.

The following example illustrates the declaration of overloaded *new* and *delete* operators in the *class date*:

```
class date
{
    int year, month, day;
  public:

    date( int a, int b, int c )  { year=a, month=b, day=c; };

    ~date() {};

    // overloaded new operator
    void* operator new (size_t siz )
    {
        return ::new char [siz ];
    };
```

```
        // overloaded delete operator
        void operator delete (void* ptr )
        {
                ::delete ptr;
        };
};
```

An overloaded *new* member function must take a *size_t*-type argument, which specifies the size, in bytes, of an object to be allocated. The function then returns the address of newly allocated object.

An overloaded *delete* member function must take a *void** argument, which points to an object to be deallocated. The function does not return any value.

Users are free to implement the body of the *new* and *delete* member functions in any way they like.

2.10 Operator Overloading

C++ allows users to define standard built-in operators to work on classes. This enables class objects to be used as if they were of the built-in data type. The built-in '+', '-', '*', and "[]" operators have no meaning to class objects, unless users explicitly overload these operators in their classes.

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	<<=
>>=	[ ]	( )	->	->*	new	delete	

The C++ operators that may be overloaded in classes are shown in the above table. Specifically, the meanings, precedence, and arity of operators cannot be overridden by overloading. This means that the "+", "-","*" and "&" operators may be defined as unary or binary operators. Furthermore, overloading treats all prefix and suffix instances of "++" and "--" operators as prefix operators.

All overloaded operator functions must take at least one class object as argument. They may be declared as friend or member functions. However, operators such as "<<" and ">>," which do not require a left operand to be a class object, should be defined as friend functions, and operators such as "[]", '=', "->", "()" and "+=," which require a left operand to be a class object, should be declared as member functions.

The following example illustrates the use of operator overloading:

```
// source module: overload.c
#include <iostream.h>
#include <string.h>
class sarray
{
        int num;
        char *data;
    public:
        sarray( const char* str )              // constructor
        {
                num = strlen(str)+1;
                data = new char[num];
                strcpy( data, str );
        };

        ~sarray() { delete data; };

        char& operator[] (int idx )            // destructor
        {
                if (idx>=0 && idx < num)
                        return data[idx];
                else
                {
                        cerr << "Invalid index: " << idx << endl;
                        return data[0];
                };
        };
```

```
const char* operator=( const char* str )
{
        if (strlen(str))
        {
                delete data;
                num = strlen(str)+1;
                data = new char[num];
                strcpy (data, str );
        }
        return str;
};

const sarray& operator=( const sarray& obj )
{
        if (strlen(obj.data))
        {
                delete data;
                num = strlen(obj.data)+1;
                data = new char[num];
                strcpy (data, obj.data );
        }
        return obj;
};

int operator == ( const char* str )
{
        return (str && data) ?  strcmp(data,str) : -1;
};

int operator < ( const char* str)
{
        return (strcmp(data,str) < 0) ? 1 : 0;
};

int operator > ( const char* str )
{
        return (strcmp(data,str) > 0) ? 1 : 0;
};
```

```
        friend ostream& operator << (ostream& os, sarray& obj)
        {
            return os << (obj.data ? obj.data : "Nil") ;
        };
};

int main()
{
    sarray A("Hello"), B("world");
    cout << "A: " << A << endl;         // << operator
    A = "Bad";                          // = operator
    A[0] = 'T';                         // [] operator
    cout << "A: " << A << endl;
    A = B;                              // = array& operator
    cout << "A: " << A[1] << endl;
    cout << "A < B: " << (A < "Two") << endl;
    return 0;
}
```

The above example defines a class *sarray*, which manages a character array per its object. The advantage of using the class is that each of its objects adjusts its buffer dynamically to fit any character string stored in it. Furthermore, the objects can be operated on using the "<<", "[]", "=," and "<" operators, due to the declaration of the overloaded operator functions. All of these make the *sarray* objects more intuitive to understand and use, and they hide the low-level array manipulation coding from users.

The compilation and sample output of the program is:

```
% CC overload.c
% a.out
A: Hello
A: Tad
A: o
A < B: 0
```

2.11 Template Functions and Template Classes

Template functions and template classes enable users to create generic functions and classes that work with different data types and classes. After these functions and classes are coded and tested, users can create different instances of these functions and classes for specific data types and/or classes. Thus, template functions and classes significantly save application development time. Another advantage of these is the reduction, per program, of unique functions and class names that require definition. This speeds up the compilation process and reduces potential name conflicts in users' programs.

A template function or a template class is instantiated when it is first used or when its address is being taken. These instances have no names, and they last only as long as they are in use.

2.11.1 Template Functions

The formal syntax of a template function declaration is:

```
template <formal param. list> <return_value> <function_name>
     ( <arg_list> )
{
     <body>
}
```

For example, to declare a template function that swaps the content of two objects:

```
template <class T> void swap ( T& left, T& right )
{
     T temp = left;
     left = right;
     right = tenp;
}
```

Once the above template function is declared, users can create objects of specialized instances of that class for various data types. The following example shows how this is done:

```
main()
{
     int a = 1, b = 2;
     swap (a, b) ;            // creates a swap(int,int) instance
     double aa = 101.0, bb = 25.0;
```

```
        swap( aa, bb );          // creates a swap(double,double) instance
    }
```

A formal parameter list is enclosed by the "<" and ">" characters. The list consists of a comma separated list of formal type parameters. The list cannot be empty. Each formal type parameter begins with a *class* key word and is followed bv a type identifier. For example, the following declaration is correct:

```
template <class T, class U>  void foo( T*, U);
```

whereas the following declaration is wrong, as the type U does not have the *class* key word preceding it:

```
template <class T, U> void foo1(T&);          // Error. Must be "class U"
```

A type parameter may appear only once in a formal type parameter list. Thus, the following declaration is incorrect, as it specifies class *T* twice:

```
template <class T, class T> void foo1(T&);       // Error
```

Each type parameter must appear at least once in a template function argument list. Thus, the following declaration is incorrect, as the type *U* is not used in the function argument list:

```
template <class T, class U> U  foo1(T&);         // Error
```

A correct version of the declaration is to use type *U* in the function argument list, such as the following:

```
template <class T, class U> U foo1(T&,U*);       // OK.
```

A template function can be declared *extern, static,* or *inline*. The specifier is placed after the formal parameter list and before the return value type specification. The following examples declare an inline function and an external function:

```
template <class T> inline void foo( T* tobj, int size) { ... }
template <class T> extern int fooA( T&  tobj) ;
```

A template function can be overloaded, provided that the signature of each declaration is distinguished, either by argument type or by number. For example, the following declarations are all legal:

```
template <class T> T min(T  t1, T t2);
template <class T> T min(T* t1, T t2, T t3);
template <class T> T min(T  t1, int t2);
```

However, the two declarations below are incorrect, as type parameter identifiers cannot be used to differentiate overloaded template functions:

```
template <class T> T min(T  t1, T t2);
template <class U> U min(U t1, U t2);            // Error!
```

Finally, specialized template functions can be defined. Specialized functions are used in higher precedence than are generic template functions in resolving function references. For example, in the following program, the first invocation of *min* in the *main* function uses the specialized version of *min* that takes *char**-typed arguments, and the second invocation of *min* creates an instance of the template function *min* for the double data type:

```
template <class T> T min(T t1, T t2)
{
     return (t1 < t2) ? t1 : t2;
}
// specialized version of min()
char * min(char* t1, char* t2)
{
     return (strcmp(t1,t2) < 0) ? t1 : t2;
}

int main()
{
     char* ptr = min ("C++","UNIX");             // min(char*,char*)
     double x = min(2.0, 3.0);                   // min(T,T)
}
```

2.11.2 Template Classes

The formal syntax of a template class declaration is:

```
template <formal parameter list> class <class_name>
{
     <declaration>
}
```

In a template class declaration, the *template* key word is followed by a formal parameter list, then a class name and its body. A formal parameter list is enclosed by the "<" and ">" characters. The list consists of a comma-separated list of formal parameters. A formal parameter may be a type parameter or a constant expression parameter. The following is an example of a template class declaration:

```
template <class T, int len> class foo
{
     T list[len];
     ...
};
```

As in a template function declaration, each parameter declared in a formal parameter list should be used at least once in the associated template class declaration. Moreover, whenever a template class name is referenced, it must always be specified along with its parameter list enclosed by angle brackets, except inside the class declaration. The following example depicts a declaration of a template class *Array*, which contains an array of type *T* with *len* entries. The *T* and *len* are parameters to be supplied when the template class instances are created. Note also that the constructor function definition has the class name and parameter list specified:

```
template <class T, int len>   class Array
{
   public:
      Array();
      ~Array() {};
   protected:
      T list[len];
};


template <class TT, int len> inline
Array<TT,len> :: Array()
{
     ...
}
```

To create an object for a specialized instance of a template class, users specify the class name, followed by the actual data for the parameter list enclosed in "<" and ">." Thus to create an object of the *Array* class that contains an integer array with 100 entries, the object definition is:

```
Array<int, 100> foo;                    // foo is an object
```

Template classes can be derived from template or nontemplate base classes. The type/subtype relationship between derived and public base template classes holds, provided they are of the same actual parameter types.

The following example declares a template subclass *Array_super* which is derived from the template classes *b1* and *b2*. The *foo* variable is defined as the int-instance of the *Array_super* class, and *ptr* is a pointer to the int-instance of the class *b1*. Since *b1<int>* and *Array_super<int>* have the same parameter type, they are compatible and, thus, *ptr* can be assigned the address of *foo*. However, *b1<double>* and *Array_super<int>* are incompatible; thus, it is an error to assign the address of *foo* to *ptr2*.

```
template <class Type>
     class Array_super :  public b1<Type>, public b2<Type> {...};
Array_super<int> foo;
b1<int> *ptr = &foo;                  // Correct
b1<double> *ptr2 = &foo;              // Error
```

Friend functions and classes can be declared inside template classes. Each of these friend functions and classes may be one of the following:

- Nontemplate function or class
- Template function or class
- Specialized instance of a template function or class

If a friend function or class is a general template, this means that all instances of that function or class are friends to the template class. On the other hand, if a friend function or class is a specialized instance of a template, only that instance is a friend to the template class being declared. The following example illustrates these concepts:

```
template <class U> class Container
{
     // general template friend class
     template <class T> friend class general_class;
```

```
        // general template friend function
        template <class UT> friend general_func ( <UT&, int );

        // Only the same type U instance of class Array is friend
        friend class Array<U>;

        // Only the same type U instance of the function is friend
        friend ostream& operator<< (ostream&, Container<U>&);

        // non-template friend class
        friend class dates;

        // non-template friend function
        friend void foo ();
};
```

In the above example, the template class *Container* has a formal parameter of type *U*. The *Container* class has a few friend functions and classes; of these, class *dates* and *foo* are friends for all specialized instances of the *Container* class. The template *Array* class and the overloaded operator "<<" are friends for specialized instances of the *Container* class as long as they use the same parameter type. Thus, *Array<int>* is a friend class for the *Container<int>*, but A*rray<double>* is not a friend for the *Container<int>*. Finally, all specialized instances of the template *general_class* class and *general_func* function are friends for all specialized instances of the *Container* class.

Specialized member functions or template classes may be defined for a template class. However, all these specialized instances can be defined only after the template class is declared. Furthermore, a specialized template class must define all member functions of a template class on which it is based. The following shows an example of a declaration of a template class *Array*, then a specialized constructor of *Array* for the *double* data type, and, lastly, a definition of a specialized class *Array* for the *char** type:

```
template <class T> class Array
{
   public:
      Array(int sz) { ar=new T[size=sz]; };
      ~Array() { delete [siz] ar; };
      T& operator[](int i) { return ar[i]; };
   protected:
      T* ar;
```

```
        int size;
    };

    // Specialized constructor for double type
    Array<double>::Array( int size ) { ... }

    // Specialized Array class definition
    class Array<char*>
    {
      public:
        Array(int sz)            { ar=new char[size=sz]; };
        ~Array()                 { delete [] ar; };
        char& operator[](int i) { return ar[i]; };
      protected:
        char* ar;
        int size;
    };
```

A template class can declare static data members. Each specialized instance of the template class has its own set of static data members. The following example declares a template *Array* class with two static data members: *Array<T>::pool* and *Array<T>::pool_sz*. These static variables are defaulted to be initialized to 0 and 100, respectively, for all specialized instances of class *Array*:

```
    template <class T> class Array
    {
      public:
        Array(int sz)            { ar=new char[size=sz]; };
        ~Array()                 { delete [sz] ar; };
        void *operator new(size_t);
        void operator delete(void*,size_t);
      protected:
        char*            ar;
        int              size;
        static Array *   pool;
        static const int pool_sz;
    };
    template <class T> Array<T>* Array<T>::pool = 0;
    template<class T> const int Array<T>::pool_size = 100;
```

One can define a specialized instance of static class data members and specify unique initial values for them. Thus one can define the *Array<char>::pool* and *Array<char>::pool_sz* variables for the *char* instance of the class *Array* as:

```
Array<char>* Array<char>::pool = new char[1000];
Array<char>* Array<char>::pool_sz = 1000;
```

In addition to all the above, static class data members of a template class can be accessed only through a specialized instance of the class. Thus the first statement below, which directly references the *Array<T>::pool,* is illegal, whereas the references of *Array<char>::pool* and *Array<int>::pool_sz* are correct:

```
cout << Array<T>::pool << endl;              // Error
Array<char>* ptr = Array<char>::pool;        // Correct
int x = Array<int>::pool_sz;                 // Correct
```

Finally, a nontemplate function can manipulate objects of specialized instances of template classes, whereas a template function can use objects of either a specific instance or of a general parameterized template class. In the following, *foo* is a nontemplate function; thus, it may work with objects of a specialized instance (in this case, *Array<int>*) of the class *Array*. On the other hand, *foo2* is a template function and it can manipulate objects of any instance of the class *Array,* as long they have the same parameter type (i.e., *foo2<int>* function can take a *Array<int>* type object as argument):

```
void foo( Array<int>& Aobj, int size )
{
    Array<int> *ptr = &Aobj;
    ...
}

template <class T> extern void foo2 ( Array<T>&, Array<int>& );
```

2.12 Exception Handling

ANSI/ISO C++ provides a standard exception handling method for all applications to respond to run-time program anomalies. This simplifies application development efforts and ensures consistency in behavior of applications.

An exception is an error condition detected in a program at run time. An exception is "raised" via a *throw* statement, and an exception handler function is "caught" by a user-defined *catch*-block, which is in the same program. If a catch-block does not terminate the program, the program control flow continues at the code right after the catch-block and is not after the *throw* statement. Furthermore, only by code executed directly or indirectly within a *try* block can exception be thrown. Thus, exception handling requires special structuring of users' C++ program to anticipate code region where exception may occur and to specify one or more catch-blocks to handle exception.

C++ exception handling mechanisms are synchronous, in that exceptions are triggered in users' applications via the explicit *throw* statements. This differs from asynchronous exceptions caused by events like keyboard interruptions from users. The latter exceptions are unpredictable as to when and where they will occur, and they can be handled via the *signal* function (see the chapter on Signals).

The following example illustrates a simple exception handling in C++:

```
// source module: simple_exception.C
#include <iostream.h>
main( int argc, char* argv[])
{
    try {
      if (argc==1) throw "Insufficient no. of argument";
      while (--argc > 0)
            cout << argc << ": " << argv[argc] << endl;
      cout << "Finish " << argv[0] << endl;
      return 0;
    }
    catch (const char* msg ) {
      cerr << "exception: " << msg << endl;
    }
    catch (int unused ) {
      cerr << "This catch block is un-used\n";
      return 99;
    }
    cout << "main: continue here after exception\n";
    return 1;
}
```

In the above example, the normal function code for *main* is enclosed in the try block. In this block, a test is made on the *argc* value. If its value is 1, an exception occurs and the throw statement is executed to raise an exception. However, if *argc* value is greater than 1, the *while* loop is executed to dump out all command line arguments, in reverse order, and the program terminates via the *return 0* statement.

The syntax of a *throw* statement is:

```
throw <expression> ;
```

where <expression> is any C++ expression that evaluates to a C++ basic data type value or a class object. The <expression> value is used to select a catch-block that the "argument" type matches or that is compatible with the <expression> data type. If a *throw* statement is executed, the rest of the statements in the same *try* block are skipped, and the program flow continues in a selected catch-block.

One or more catch-blocks may be specified in a function. The catch-blocks must be specified right after a try block. Each catch block begins with the catch key word, followed by a object tag specification enclosed in "(" and ")". After that, one or more statements for the block are enclosed in "{" and "}". Note that although a catch-block looks like a function definition, it is not a function. It is really just a set of C++ statements collected together and given an object tag. The object tag is used by a *throw* statement to select which catch-block to execute, and the object contains the <expression> value of the *throw* statement. This value usually conveys more information about an exception that was raised.

After a catch-block is executed, if it does not terminate the program, the program flow continues at the statement after the catch-blocks.

In the above example, a sample compilation and sample run of the program, with no exceptions, are:

```
%    CC simple_exception.C
%    a.out hello
1: hello
Finish a.out
```

If the program is rerun without any argument, an exception occurs and the program's output is:

```
%    a.out
exception: Insufficient no. of argument
main: continue here after exception
```

Note that in the above run, the *throw* statement includes a *char** expression, thus the *catch (const char* msg)* block is selected, but not the *catch(int unused)* block. However, if users add a statement like *throw 5* in the *try* block, this *new* statement, if executed, selects the *catch(int unused)* block only.

A *throw* statement can also specify a class object name as argument. This allows it to pass more information to a catch-block for better error diagnostics and reporting. The following program is a revised version of the previous example:

```
// source module: simple2.C
#include <iostream.h>

// special class for error reporting
class errObj
{
   public:
      int line;
      char* msg;
      // constructor function
      errObj( int lineNo, char* str )
      {
         line = lineNo;
         msg = str;
      };
      // destructor function
      ~errObj() {};
};

main( int argc, char* argv[])
{
      try {
         if (argc==1) throw errObj(__LINE__,
                                    "Insufficient no. of arguments");
         while (--argc > 0)
               cout << argc << ": " << argv[argc] << endl;
         cout << "Finish " << argv[0] << endl;
         return 0;
      }
      catch (errObj& obj )  {
         cerr  << "exception at line: " << obj.line << ", msg: "
```

```
                  << obj.msg << endl;
        }
        cout << "main: continue here after exception\n";
        return 1;
}
```

The compilation and sample run of the program, with an exception, are:

```
%    CC simple2.C
%    a.out
exception at line: 23, msg: Insufficient no. of arguments
main: continue here after exception
```

If an exception is raised but no catch-blocks in the same function match the *throw* statement's argument, then the function is "returned" to its calling function(s), and the catch-blocks in each of these functions are searched for a match to the *throw*'s argument. The process stops when either a catch-block is matched and the program flow continues in that block and in the code after that, or no match is found and the built-in function *terminate* is called. The *terminate* function in turns calls the *abort* function, which aborts the program.

When a function returns to its caller(s) due to a *throw* statement, any objects local to an exiting function are deallocated via their destructors, and the run-time stack is rewound to deallocate the stack frame reserved for the exiting function.

The following example illustrates these concepts:

```
// source module: simple3.C
#include <iostream.h>
void f2 ( int x )
{
      try   {
            switch (x) {
               case 1: throw "exception from f2.";
               case 2: throw 2;
            }
            cout << "f2: got " << x << " arguments.\n";
            return;
      }
      catch (int no_arg ) {
```

```
                cerr << "f2 error: need at least " << no_arg << " arguments\n";
        }
        cerr << "f2 returns after an exception\n";
}

main( int argc, char* argv[])
{
        try         {
           f2(argc);
           cout << "main: f2 returns normally\n";
           return 0;
        }
        catch (const char* str ) {
           cerr << "main: " << str << endl;
        }
        cerr << "main: f2 returns via an exeception\n";
        return 1;
}
```

In the above example, if the program is invoked with no argument, the *f2* function executes the *throw "exception from f2"* statement and this causes the *catch (const char* str)* block in *main* to be executed. The program output is:

```
%    CC simple3.C
%    a.out
main: exception from f2.
main: f2 returns via an execution
```

If, however, the program is invoked with one argument (and *argc* value is 2), the *f2* function executes the *throw 2* statement and this causes the *catch(int no_arg)* block in *f2* to be executed. The program output for this is:

```
%    a.out hello
f2 error: need at least 2 arguments
f2 returns after an exception
main: f2 returns normally
```

Finally, if the program is invoked with two or more arguments, then no exception is raised by *f2*, and the program output is:

```
%    a.out hello world
f2: got 3 arguments.
main: f2 returns normally
```

2.12.1 Exceptions and Catch-Blocks Matching

When a *throw* statement with an argument of data type *T* is executed, the following rules are used to match a catch-block to catch the exception. Assuming the data type of a catch block is of type *C*, then the catch-block is selected if any one of the following conditions is true:

- *T* is same as *C*
- *T* is a const or volatile of *C*, or vice versa
- *C* is a reference of *T*, or vice versa
- *C* is a public or protected base class of *T*
- Both *C* and *T* are pointers, and *T* can be converted to *C* by a standard pointer conversion

2.12.2 Function Declarations with Throw

A function declaration may optionally specify a set of exceptions that it directly or indirectly will throw by providing a *throw list*. For example, the following statement declares an external function *funct*, which may throw exceptions with data tag of *const char** or *int*.

```
extern void funct (char * ar)  throw(const char*, int);
```

A throw list may be empty, which means that a function will not throw any exceptions. The following statement declares *funct2* not raise any exception, either directly or indirectly:

```
extern void funct2(char * ar)  throw();
```

However, if the above *funct2* does invoke a throw statement, the built-in *unexpected* function is called. The *unexpected* function, by default, calls the *abort* function to terminate the program.

A throw list is not part of a function type. Thus, it cannot be used to overload functions. For example, the following two function declarations are treated the same by the C++ compiler:

```
extern void funct3 (char * ar)  throw(char*);
extern void funct3 (char * ar=0);
```

2.12.3 The Terminate and Unexpected Functions

The *terminate* function is called when a *throw* statement is executed but no matching catch-block is found. By default, the *terminate* function invokes the *abort* function to abort the program. However, users may install their functions in place of *abort* in *terminate* via the *set_terminate* function, as follows:

```
extern void user_terminate( void );
void (*old_handler)(void);
old_handler = set_terminate ( user_terminate );
```

In the above sample statement, the *user_terminate* is a user-defined function to be called when *terminate* is invoked. The *user_terminate* function is installed via the *set_terminate* function, and the *old_handler* variable holds the old function installed in the *terminate* function.

The *unexpected* function is called when a *throw* statement is executed in a function that is declared with an empty throw list. By default, the *unexpected* function invokes the *terminate* function to abort the program. However, users may install their functions in place of *terminate* in *unexpected* via the *set_unexpected* function as follows:

```
extern void user_unexpected ( void );
void (*old_handler)(void);
old_handler = set_unexpected ( user_unexpected );
```

In the above sample statement, the *user_unexpected* is a user-defined function to be called when *unexpected* is invoked. The *user_unexpected* function is installed via the *set_unexpected* function, and the *old_handler* variable holds the old function installed in the *unexpected* function.

Since the *terminate* and *unexpected* functions are assumed to never return to their caller, any user-installed functions to be invoked by either *terminate* or *unexpected* should terminate the program at their completion.

Actually, the *terminate* and *unexpected* functions should rarely be called in a well-designed program, as they really signal that users programs have not accounted for all possible exceptions raised in their programs, and this is not a good programming practice. The only exception to this is the use of third-party C++ libraries, which raises undocumented exceptions. In this case, users should report the problems to their vendors and install the handler for *terminate* and/or *unexpected* only as a short-term work-around.

2.13 Summary

This chapter covers the object-oriented program design and the draft ANSI/ISO C++ language features. As discussed in the chapter, C++ is derived from C and has added constructs to support an object-oriented programming style. These new constructs include classes declaration, classes inheritance, polymorphism via virtual functions, and template functions and classes. Finally, the C++ exception handling method is also discussed. Extensive examples are depicted to illustrate the uses of these concepts.

These C++ features are discussed here to refresh users' memories of the C++ programming techniques so that they can understand the rest of the book. Furthermore, some readers may not be familiar with the new features of ANSI/ISO C++. This chapter describes those new features in more detail.

The next chapter discusses the C++ I/O stream libraries. These libraries are used extensively by all C++ applications, but their full features are not always understood by readers. The next chapter should help users refresh their memories and learn some new features.

2.14 References

[1]. Andrew Koenig, *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++ (Committees: WG21/N0414, X3J16/94-0025)*, January 1994.

[2]. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, 1990.

[2]. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

CHAPTER 3

C++ I/O Stream Classes

This chapter reviews the C++ I/O stream classes. These classes are defined in the draft ANSI/ISO C++ standard. They enable users to perform I/O operations with the standard input and output streams, disk files, and character buffers. They essentially eliminate the needs of users to use the C stream I/O functions, string functions, and the *printf* class functions. The advantage of using the C++ I/O stream classes is that they allow the compiler do more type-checking on the actual arguments supplied to these classes. They can also be extended to support user-defined classes.

There are three major headers for the I/O stream classes. The <iostream.h> header declares the *istream, ostream,* and *iostream* classes for the standard input and output stream I/O operations. It also declares the *cout, cin, cerr,* and *clog* objects that are used in most C++ programs. The <fstream.h> header declares the *ifstream, ofstream,* and *fstream* classes for disk file I/O operations. Finally, the <strstream.h> header declares the *istrstream, ostrstream,* and *strstream* classes for in-core data formatting with character buffers.

Although C++ I/O stream classes, especially the *cout, cin,* and *cerr* objects, are widely used in most C++ applications, not all users know the detailed features of these classes, nor are they being described fully by many textbooks on C++ programming. The rest of this chapter gives a comprehensive description of these I/O stream classes.

3.1 The I/O Stream Classes

The <iostream.h> header declares three classes for the standard input and output streams I/O: *istream, ostream,* and *iostream*. The *istream* class is for data input from an input stream, the *ostream* class is for data output to an output stream, and the *iostream* class is for data input and output within a stream. Besides these classes, the <iostream.h> header also declares four objects:

Stream object	**Function**
cin	An *istream* class object tied to the standard input
cout	An *ostream* class object tied to the standard output
cerr	An *ostream* class object tied to the standard error, providing unbuffered output
clog	An *ostream* class object tied to the standard erroi, providing buffered output

Note that the *cin, cout, clog,* and *cerr* objects do not use the same file descriptors as do the C *stdin, stdout,* and *stderr* stream pointers. However, users may force *cin* and *stdin* to use the same file descriptor; *cout* and *stdout* to use another file descriptor; and *cerr, clog,* and *stderr* to use a third file descriptor; via the static function:

```
ios::sync_with_stdio();
```

The above function should be called in a user program before any stream I/O operation is performed.

3.1.1 The istream Class

The *istream* class is used to extract data from an input stream. The *cin* object is of this data type. The user-visible operations that are defined for the *istream* class are:

Operation	**Function**
>>	Extracts white space delimited data (of any standard data type) from an input stream
istream& get(char c) , int get()	Extracts a character from an input stream. White spaces are treated as legal characters
istream& read(char* buf, int size)	Extracts *size* bytes of data from an input stream and puts it into *buf*

Operation	**Function**
istream& getline(char* buf, int limit, char delimiter='\n')	Extracts from an input stream at most *limit*-1 byte of data, or when the delimiter character or *EOF* (end-of-file) is encountered. The extracted data are put into *buf*. The delimiter character, if found, is not included in *buf*
int gcount()	Returns the number of bytes extracted by the last call of *read* or *getline*
istream& putback(char c)	Puts the specified character *c* back into an input stream
int peek()	Returns the next character in an input stream but does not extract it
istream& ignore(int limit=1, int delimiter=EOF)	Discards up to *limit* characters in an input stream, or if the delimiter character or *EOF* are encountered
streampos tellg()	Returns the current stream marker byte offset from the beginning of the stream
istream& seekg(streampos offset, seek_dir d=ios::beg)	Repositions the stream marker to *offset* bytes from the beginning of the file (if *d*=ios::beg), current stream marker position (if *d*=ios::cur), or *EOF* (if *d*=ios::end)

The following is a UNIX *wc*-like program that illustrates the use of *istream* class operations. Specifically, the program counts the number of lines, words, and characters found in the standard input stream:

```
/* source module: wc.C */
#include <iostream.h>
#include <ctype.h>

int main()
{
    int ch, lineno=0, charno = 0, wordno = 0;
    for (int last=0; cin && (ch = cin.get()) != EOF; last=ch)
      switch (ch)  {
        case '\n':  lineno++, wordno++;
                    break;
```

```
                case '/':  if (cin.peek()=='/')    {     // don't count comments
                              cin.ignore(10000,'\n');
                              lineno++;
                           }
                           else charno++;
                           break;
                default:   charno++;
                           if (isspace(ch) && last!=ch) wordno++;
            }
            cout << charno << " " << wordno << " " << lineno << "\n" << flush;
    }
```

The compilation and sample run of this program are:

```
%    CC wc.C
%    a.out < /etc/passwd
557  23   14
```

The ">>" operator may be overloaded as a friend function for each user-defined class. This enables users to extract class object data in the same manner as they do for the standard C++ data type objects. The overloaded ">>" function should be defined in the following manner:

```
class X                          // user-defined class
{    ...
   public:
     friend istream& operator >> (istream& is, X& xObj)
     {
          is >> <class X data members>;
          return is;
     };
     ...
};
```

3.1.2 The ostream Class

The *ostream* class is used to insert data to an output stream. The *cout*, *cerr*, and *clog* objects are of this data type. The user-visible operations that are defined for the *ostream* class are:

Operation	**Function**
<<	Inserts the value of any standard data type to an output stream
ostream& put(char ch)	Inserts a character *ch* to an output stream
ostream& write(const char*buf, int size)	Inserts *size* byte of data contained in *buf* to an output stream
typedef streampos long; streampos tellp()	Returns the current stream marker byte offset from the beginning of the stream
ostream& seekp(streampos offset, seek_dir d=ios::beg)	Repositions the stream marker to *offset* bytes from the beginning of the file (if *d*=ios::beg), current stream marker position (if *d*=ios::cur), or *EOF* (if *d*=ios::end)
ostream& flush()	Forces flushing of data to an output stream

The following statements illustrate the use of *ostream* class operations:

```
cout << "x=" << x << ",y=" << y << "\n";
cout.put('\n').write("Hello world",11).put('\n');
```

The "<<" operator may be overloaded as a friend function for each user-defined class. This enables users to print class objects' data in the same manner as they do for the standard C++ data type objects. The overloaded "<<" function should be defined in the following manner:

```
class X                          // user-defined class
{
    ...
  public:
    friend ostream& operator << (ostream& os, X& xObj)
    {
        os <<<class X data members>;
        return os;
    };
    ...
};
```

3.1.3 The iostream Class

The *iostream* class is derived from the *istream* and *ostream* classes. It contains all the properties of its two base classes. This class is used primarily as a base class for the *fstream* class, and the latter is commonly used for defining objects to read/write disk files.

3.1.4 The ios Class

The *istream, ostream,* and *iostream* classes contain a virtual base class *ios*. The *ios* class records an error state and a format state for each I/O stream class object. Specifically, the *ios* class declares the following operations:

Operation	**Function**
int eof()	Returns 1 if *EOF* has been encountered in a stream
int bad()	Returns 1 if an invalid operation (e.g., seeking pass *EOF*) has been detected
int fail()	Returns 1 if an I/O operation is unsuccessful or bad() is true
int good()	Returns 1 if all previous I/O operations have been successful
int rdstate()	Returns the error state of a stream
void clear(bits=0)	Sets the error state's bit vector to the value given in *bits*. If *bits* is 0, resets the error state to 0
int width(int len)	Sets the field width to *len* for the next data to be inserted, or sets the buffer limit to *len-1* for character string extraction. This routine returns the previous field width
char fill(char ch)	Sets the fill (padding) character to *ch*. Returns the previous fill character
int precision(int)	Sets the number of significant digits to be displayed for real number insertion. Returns the previous precision value
long setf(long bitFlag)	Adds the format bit(s) as specified in *bitFlag* to the existing format state for insertion. Returns the old format state value. Possible values for *bitFlag* may be:

ios::showbase	Displays numeric base
ios::showpoint	Displays trailing decimal point and zero
ios::showpos	Displays sign character for numeric values
ios::uppercase	Uses "X" for hexadecimal number display (when *ios::showbase* is set), and "E"to print floating point numbers in scientific notation

long setf(long bitFlag long bitField)

Sets/resets (as according to *bitFlag*) the format bits as specified in *bitField* for insertion. Returns the old format state value. Possible values for *bitField/ bitFlag* may be:

ios::basefield	ios::hex	Sets the numeric base to hexadecimal
	ios::oct	Sets the numeric base to octal
	ios::dec	Sets the numeric base to decimal (the default)
ios::floatfield	ios::fixed	Displays real numbers in decimal notation
	ios::scientific	Displays real numbers in scientific notation
ios::adjustfield	ios::left	Left-justifies the next argument by inserting fill characters after the value
	ios::right	Right justifies the next argument by inserting fill characters before the value
	ios::internal	Fill characters are added after any leading sign or base indication, but before the value
ios::skipws	0	Extraction will not skip white spaces
	ios::skipws	Extraction will skip white spaces (default)

All I/O operations of a stream object are aborted if its error state is not zero. Users can check the error state of a stream object via the *ios::bad* or *ios::fail* function, or by using the overloaded "!" operator. The following sample statements illustrate how this is done:

```
if (!cin || !cout) cerr << "I/O error detected";
if (!(cout << x) || x<0) cout.clear(ios::badbit | cout.rdstate());
if (cout.fail()) clog << "cout fails\n";
```

A stream object's error state can be reset via the *ios::clear* function:

```
if (!cin) cin.clear();
```

Besides dealing with error states, the *ios* class functions are also used to set data formatting options of stream objects. The data formatting features provided by the *ios* class are as powerful as those provided by the C *printf* class functions.

The following example program illustrates the use of these *ios* functions:

```
// source module: ios.C
#include <iostream.h>
int main()
{
     int x = 1024;
     double y= 200.0;
     static char str[80] = "Hello";
     cout.setf( ios::showbase | ios::showpos | ios::uppercase );
     cout.setf( ios::scientific, ios::floatfield );
     cout.precision(8);
     cout << "'";
     cout.width(10);
     cout.fill('*'):
     cout  << x << "', y='" << y << "'\n";
     cout << "'";
     cout.width(7);
     cout.setf(ios::left,ios::adjustfield);
     cout.setf(ios::fixed, ios::floatfield ),
     cout << x << "', y='" << y << "'\n";
     cout << "'";
     cout.width(8);
     cout.setf(ios::right,ios::adjustfield);
     cout << str << "'\n":
}
```

In the above example, the *ios::setf(ios::scientific, ios::floatfield)* and the *ios::precision(8)* set the display format for floating point data in scientific notation with a precision of eight, respectively. The *ios::setf(ios::fixed, ios::floatfield)* statement, on the other hand, sets the display format for floating point value to be fixed point notation. Note that once a display format is set for an object, the setting is unchanged until it is overridden by the next *ios::setf* call.

The rest of the example should be quite self-explanatory. The compilation and sample output of the program is:

```
%    CC ios.C -o ios; ios
'*****+1024', y='+2.00000000E+02'
'+1024**', y='+200.00000000'
'***Hello'
```

3.2 The Manipulators

A manipulator is a function that can be included in an I/O stream class operation to cause some special effects. For example, the flush manipulator is commonly used with *ostream* class objects to force flushing of buffered data held in these objects:

```
cout << "A big day" << flush;
```

A simple manipulator is a function that takes an *istream&* or *ostream&* argument, operates on it in some way, and returns a reference of the object. The following example illustrates the definitions of two manipulators, *tab* and *fld*. The *tab* manipulator inserts a TAB character to an output stream, and the *fld* manipulator sets the display format of an output stream to display integer data in octal format and with the *O* prefix. Furthermore, the minimum field width for displaying a value is 10:

```
ostream& tab(ostream& os)
{
            return os << '\t';
}

ostream& fld(ostream& os)
{
            os.setf(ios::showbase,ios::showbase);
            os.setf( ios::oct, ios:basefiled);
            os.width(10);
            return os;
}
```

The following statements show how to use the manipulators:

```
int x = 50, y = 234;
cout << fld << x << tab << y << '\n';
```

The <iomanip.h> header file declares a set of system-supplied manipulators that are commonly used with stream class objects. Some of these manipulators are:

Manipulator	**Function**
flush	Forces flushing of data to an output stream
setw(int width)	Sets the minimum field width for the next argument to be inserted and the maximum buffer limit (*width-1*) for the next character string extraction

resetiosflags(long bitFlag)
setiosflags(long bitFlag)
	Resets or sets the specified format bits in the stream's format state

setprecision(int p)	Sets the precision to *p* for the next real number to be inserted

The following statements show the sample uses of some of these system-supplied manipulators:

```
cout << x << setw(5) << y << flush;      // force flushing of cout
cin  >> resetiosflags(ios::skipws)       // No white spaces skipping
     >> c
     >> setiosflags(ios::skipws);        // Skip white spaces
cout <<  setprecision(8) << Dval;        // set precision
```

3.3 The File I/O classes

The <fstream.h> header declares the *ifstream, ofstream,* and *fstream* classes for file manipulation. These classes provide functionality that is equivalent to the *fopen, fread, fwrite,* and *fclose*, etc. C stream files functions.

Specifically, the *ifstream* class is derived from the *istream* class, and it enables users to access files and read data from them. The *ofstream* class is derived from the *ostream* class, and it enable users to access files and write data to them. Finally, the *fstream* class is derived from both the *ifstream* and *ofstream* classes. It enable users to access files for both data input and output.

The constructors of the *ifstream* and *ofstream* classes are defined in the <fstream.h> header as:

```
ifstream::ifstream();
ifstream::ifstream( const char* name, int open_mode=ios::in,
                         int prot = filebuf::openprot /* 0644 */);
ofstream::ofstream();
ofstream::ofstream( const char* name, int open_mode=ios::out,
                         int prot = filebuf::openprot);
```

The possible values of the *open_mode* argument and their meanings are:

Open mode	**Meaning**
ios::in	Opens a file for read
ios::out	Opens a file for write
ios::app	Appends new data to end of file. This implies *ios::out*
ios::ate	A seek to the end of file is performed after the file is opened. This does not imply *ios::out*
ios::nocreate	Returns an error if the file does not exist
ios::noreplace	Returns an error if the file already exists
ios::trunc	If a file exists, truncates its previous content. This mode is implied with *ios::out* if it is not specified with *ios::app* or *ios::ate*

The *prot* argument specifies the default access permission to be assigned to a file if it is created by the constructor function. The default value *filebuf::openprot* is 0644, which means read-write for a file owner, and read-only for anyone else. This argument value is not used when a file to be opened by the constructor already exists.

The following sample statements illustrate the use of the *ifstream* and *ofstream* classes. In the example, a file called *from* is opened for read, and another file called *to* is opened for write. If both files are opened successfully, the content of the *from* file is copied to that of the *to* file. If either file cannot be opened successfully, an error is flagged:

```
ifstream source ("from");
ofstream target("to");
if (!source || !target)
     cerr << "Error: File 'from' or 'to' open failed\n";
else for (char c=0; target && source.get(c); )
     target.put(c);
```

Besides the member functions inherited from the *iostream* classes, the *ifstream*, *ofstream*, and *fstream* classes also define their own specific functions:

Function	Meaning
void open(const char* fname, int mode, int prot=openprot)	Attaches the stream object to the named file
void close()	Closes the file to which the stream object is attached
void attach(int fd)	Attaches the stream object to the stream referenced by the file descriptor *fd*
filebuf* rdbuf()	Returns a *filebuf* array associated with the stream object

The following is a simple program to illustrate the use of the *open, close,* and *attach* functions that are unique to the *fstream* classes:

```
#include <iostream.h>
#include <fstream.h>
int main(int argc, char *argv[])
{
     ifstream source;
     if (argc ==1 || *argv[1]=='-')
          source.attach(0);              // attach to stdin
     else source.open(argv[1],ios::in);
     ...
     if (source.rdbuf()->is_open)
          source.close();                // Close the file if it is opened
}
```

Finally, random file I/O may be performed using the *seekg* and *tellg* functions that the *fstream* classes inherited from the *iostream* classes. The following example statements illustrate how these functions are used:

```
fstream tmp("foo",ios::in|ios::out);
streampos pos = tmp.tellg();           // remember file location
...
tmp.seekg(pos);                        // return to previous location
...
tmp.seekg(-10, ios::end);              // Goto 10 bytes from EOF
tmp.seekg(0, ios::beg);                // Rewind to beginning of file
tmp.seekg (20, ios::cur);              // Move 20 byte ahead
```

3.4 The strstream Classes

The <strstream.h> header defines the *istrstream, ostrstream*, and *strstream* classes for incore formatting. These classes provide functions that are equivalent to the *sprintf* and *sscanf* C library functions. The advantages of using these classes over the *sprintf* and *sscanf* functions are that these classes may be overloaded to work with user-defined classes and they allow the C++ compiler to do type-checking on programs at compile time.

The *istrstream* class is derived from the *istre.ı* class, and it enables users to extract formatted data from a character buffer. The *ostrstream* class is derived from the *ostream* class, and it enables users to insert formatted data into a character buffer. Finally, the *str-stream* class is derived from both the *istrstream* and *ostrstream* classes. It enables users to extract and insert format data with a character buffer.

The *istrstream, ostrstream*, and *strstream* classes do not declare any member functions that are unique to their classes. The following is an example of incore formatting using these *strstream* classes:

```
// source module: strstream.C
#include <iostream.h>
#include <strstream.h>
main()
{
      double dval;
      int ival;
      char wd[20];
      static char buf[32] = "45.67 99 Hello";

      // dval= 45.67, ival=99, wd="Hello"
      istrstream(buf) >> dval >> ival >> wd;
      ostrstream(buf,sizeof(buf)) << ival << " <- " << dval
               << ',' << wd << '\0';
      cout << buf << endl;                    // "99 <- 45.67,Hi"
}
```

In the above example, the *47.67, 99*, and *Hello* data are extracted from the *buf* variable and assigned to the *dval, ival*, and *wd* variables, respectively, via the *istrstream* class. This is similar to using the C *sscanf* function. In the statement, a temporary *istrstream* class object is created, and it uses *buf* as its internal buffer to perform data extraction.

The *dval, ival*, and *wd* variables values are assigned to *buf* again in a different format, via the *ostrstream* class. This is similar to using the C *sprintf* function. Note that a temporary *ostrstream* class object is created by the statement, and it uses *buf* as its internal buffer for data insertion. Finally, the terminating "\0" character in the statement is needed so that the format string stored in *buf* is NULL-terminated.

The compilation and sample output of the program is:

```
%    CC strstream.C
%    a.out
99 <- 45.67,Hi
```

If no buffer is supplied to an *ostrstream* object's constructor, the object keeps an internal dynamic array to store the input data. Users can access this array by invoking the *ostrstream::str* function. However, once the *ostrstream::str* function is called, the dynamic array is "frozen," which means that no more data can be inserted into the array via the *strstream* object, and the users are responsible for deallocating the array when done.

The following example illustrate the uses of the *ostrstream::str* function:

```
// source module: strstream.C
#include <iostream.h>
#include <fstream.h>
#include <strstream.h>

int main(int argc, char *argv[])
{
     fstream source;
     if (argc ==1 || *argv[1]=='-')
          source.attach(0);                 // attach to stdin
     else source.open(argv[1],ios::in);

     // Read the input stream and store into internal array
     ostrstream str;
     for (char c=0; str && source.get(c); )
          str.put(c);


     // Get the internal array
     char *ptr = str.str();
```

```
        // Doing something to the data in the array
        ...

        // Deallocate the array
        delete ptr;

        // Close the input stream
        source.close();                    // Close the file
}
```

3.5 Summary

This chapter reviews the C++ I/O stream classes. These classes and their system-defined objects (*cin, cout, cerr,* and *clog*) are used extensively in most C++ programs because they offer more type-checking and are extensible in supporting user-defined classes. Thus, it is advantageous to know the basic, as well as advanced, features of these classes.

The next chapter is a review of some advanced standard C library functions. These functions are not covered by the C++ standard classes or by the UNIX and POSIX application program interface functions, but they are very useful to know and should help users in development of advanced system applications.

Standard C Library Functions

C defines a set of library functions that have no direct correspondence in C++ standard classes or in UNIX and POSIX APIs. These functions provide the following services:

- Data manipulation, conversion, and encryption
- Enabling definition of variable argument functions by users
- Dynamic memory management
- Date and time processing
- Obtaining system informatioɲ

The major advantages of using the standard C library functions are portability and low maintenance of users' applications. This is because most systems (UNIX or others) that support C provide the same set of standard C library functions. These functions should have the same function prototypes and behave the same on differen systems. Furthermore, these library functions do not change constantly; thus, programs that use them are easy to maintain. Finally, ANSI C has standardized some of these library functions, further ensuring the availability of these functions on all ANSI C-compliant sysiems. Thus, C library functions should be used where applicable to reduce application development time and costs.

This chapter depicts the major ANSI C-defined library functions and a few library functions that are non-ANSI C standard but are widely available on all UNIX systems. The objective of describing these functions is to make users aware of them so that they can make use of these functions to reduce their applications development time and improve the portability and maintenance of their programs.

If portability and maintenance are major concerns of your applications, it is recommended that readers use the C++ standard classes and standard C library functions as much as possible, and use the system APIs only when necessary. However, if your applications are time-critical or require extensive kernel interfacing, use the system APIs more often than the C++ standard classes and standard C library functions.

The standard C library functions are declared in a set of header files that are commonly placed in the */usr/include* directory on UNIX systems. The archive and shared libraries that contain the object code of these library functions are the *libc.a* and *libc.so*, respectively. These libraries are commonly placed in the */usr/lib* directory on UNIX systems.

The next few sections describe the ANSI C library functions as defined in the following header files:

- <stdio.h>
- <stdlib.h>
- <string.h>
- <memory.h>
- <malloc.h>
- <time.h>
- <assert.h>
- <stdarg.h>
- <getopt.h>
- <setjmp.h>

Besides the above, the following headers are not defined in ANSI C but are available on most UNIX systems:

- <pwd.h>
- <grp.h>
- <crypt.h>

These header files declare functions which aid users in accessing UNIX systems' user and group account information, and they are defined in the *libc.a* library on UNIX systems. These headers are also described in this chapter, in case users find them useful in application development.

4.1 <stdio.h>

The <stdio.h> header declares the FILE data type that is used to reference stream files in C programs. There are also a set of macros and functions to support the manipulation of stream files. Examples of these macros and functions, which should already be familiar to readers are:

Stream function/macro	**Uses**
fopen	Opens a stream file for read and/or write
fclose	Closes a stream file
fread	Reads a block of data from a stream file
fgets	Reads a line of text from a stream file
fscanf	Reads formatted data from a stream file
fwrite	Writes a block of data to a stream file
fputs	Writes a line of text to a stream file
fprintf	Writes formatted data to a stream file
fseek	Re-positions the next read or write location in a stream file
ftell	Returns the current location in a stream file where the next read or write will occur. The return value is the number of bytes offset from the beginning of the file
freopen	Re-uses a stream pointer to reference a new file
fdopen	Converts a file descriptor to a stream pointer
feof	A macro which returns a non-zero value if end-of-file is found in a given stream file, or a zero value otherwise
ferror	A macro which returns a non-zero value if an error or end-of-file has been encountered in a given stream file, or a zero value otherwise
clearerr	A macro which clears the error and end-of-file flags of a given stream file
fileno	A macro which returns the file descriptor associated with a given stream file.

The *freopen* function is often used to redirect the standard input or standard output of an executed program. The function prototype is:

```
FILE* freopen ( const char* file_name, const char* mode, FILE* old_stream);
```

The *file_name* argument is a path name of a new stream to be opened. The *mode* argument specifies the new stream is to be opened for read and/or write. This is the same argument as that used in *fopen*, and the new stream must be opened for access consistent with that of the stream, as referenced by the *old_stream* argument. For example, if an old stream is opened for read-only, so must the new stream be opened. The same is true if the old stream is write-only or read-write. The function attempts to open the new stream of the specified access mode. If the new stream is opened successfully, the old stream is closed, and the stream pointer *old_stream* is set to reference the new stream. If the new stream cannot be opened, the *old_stream* is closed regardless. The function returns the *old_stream* value if it succeeds, or a NULL value if it fails.

The following example program emulates the UNIX *cp* (copy file) command. The program takes two file path names as arguments, and it copies the content of the file specified by the first argument (*argv[1]*) to the file specified in the second argument (*argv[2]*). Note that instead of using two stream pointers to reference the two files, the *stdin* and *stdout* stream pointers are set to reference the source and destination files, respectively, via the *freopen* function. Then, data in the source file is read via the *gets* library function and is written to the destination file via the *puts* library function:

```
#include <stdio.h>
int main( int argc, char* argv[] )
{
    if ( argc !=3 )   {
       cerr << "usage: " << argv[0] << " <src> <dest>\n";
       return 1;
    }
    (void)freopen( argv[1], "r", stdin );      // stdin references source file
    (void)frepen( argv[2],"w",stdout );        // stdout references dest. file
    for ( char buf[256]; gets( buf ); )
       puts( buf );
    return 0;
}
```

The *fdopen* function converts a file descriptor to a stream pointer. File descriptors are used in UNIX APIs to access files. Unlike stream pointers, they do not provide data buffering services. If users wish to do I/O data buffering, they may use this function to convert a file descriptor to a stream pointer. The *fdopen* function prototype is:

FILE* ***fdopen*** (**const int** file_desc, **const char*** mode);

The *file_desc* argument is a file descriptor to be converted. The *mode* argument specifies the access mode of the new stream pointer to be created. The possible *mode* values are the same as those for the *fopen* call and must be consistent with the way the *file_desc* descriptor was opened. Specifically, if a given file descriptor is opened for read-only, the *mode* value should be "*r.*" Similarly, if a given file descriptor is for write-only, the *mode* value should be "*w.*" The function returns a new stream pointer if it succeeds or a NULL pointer if it fails. One possible cause of failure is the *mode* argument value is inconsistent with the a *file_desc* descriptor access mode.

The following sample function illustrates one possible implementation of the *fopen* function by using *fdopen*:

```
FILE* fopen ( const char* file_name, const char* mode )
{
        int fd, access_mode;
        /* convert mode to integer valued access_mode */
        if (( fd = open( file_name, access_mode, 0666 )) < 0 )
            return NULL;
        return fdopen ( fd, mode );
}
```

In the above example, the character string mode argument is converted to an integer-valued access mode flag. The *open* API is then called to open a file that is named by the *file_name* argument, and the returned file descriptor is stored in *fd*. The function converts *fd* to a stream pointer via the *fdopen* call and returns that stream pointer as the return value.

The *fdopen* call is also used in other situations, such as the implementation of the *popen* function. This will be depicted in Chapter 8.

Finally, the <stdio.h> header also declares the *popen* and *pclose* functions. These functions are used to execute a shell command within a user program. This is very useful in enabling user programs to perform system functions conveniently, and some of these functions cannot be done via any standard library function or system API.

The function prototypes of the *popen* and *pclose* functions are:

```
FILE* popen ( const char shell_cmd, const char* mode );
int pclose ( FILE* stream_ptr );
```

The *shell_cmd* argument of the *popen* function is a user-defined shell command. It can be any command that can be executed on a command line by a shell. Users may specify input redirection, output redirection, or command pipes in the command. In UNIX, the function invokes a Bourne shell to execute the command. Furthermore, the *mode* argument value may be "*r*" or "*w*", which specifying the function to return a stream pointer for users to read the standard input data or to write data to the standard output, respectively, of the to-be-executed command. The function returns NULL if the command cannot be executed, or a stream pointer if it succeeds. Note that the *popen* function creates an *unnamed pipe* for passing data between a process calling *popen* and the executed command. Unnamed pipes are discussed in Chapter 7.

The *pclose* function is called to close a stream pointer that is obtained from *popen*. It also makes sure the executed command is terminated properly. The implementation of the *popen* and *pclose* function is explained in Chapter 8, when the UNIX process APIs are discussed.

The following example program, *ps.C*, displays all executing processes on a UNIX system that are owned by the user *root*:

```
#include <stdio.h>
int main ()
{
      /* execute the command */
      FILE * cmdp = popen( "ps -ef | grep root","r" );
      if ( !cmdp )   {
         perror ( "popen" );
         return 1;
      }
      char result [256] ;
      /* now read the "grep" command outputs */
      while ( fgets( result, sizeof(result), cmdp ) )
         fputs( result, stdout );          // echo each line read

      pclose( cmdp );                      // close the stream
      return 0;
}
```

4.2 <stdlib.h>

The <stdlib.h> header declares a set of functions for data conversion, random number generation, get and set shell environment variables, program execution control, and execution of shell commands. These functions were traditionally declared in the <stdio.h> header, but because they do not involve stream manipulation, they are grouped into a separate header by the ANSI C standard.

The *system* function declared in the <stdlib.h> header performs a function similar to the *popen* function, except that users can access the standard output or standard input of the executed command. The function prototype of the *system* function is:

```
int system ( const char* shell_cmd );
```

The *shell_cmd* argument is a character string that contains a user-defined shell command. The command may be anything that is legally entered on a shell command line of a given system. Furthermore, input redirection, output redirection, and command pipes may be specified in a *shell_cmd*. In UNIX, the function invokes a Bourne-shell to execute the command. The function returns a zero value if it succeeds and a nonzero value if the execution of a given command fails. For example, the following statement executes the shell commands: *cd /bin ; ls -l | sort -b | wc > /tmp/wc.out*:

```
if (system ( "cd /bin; ls -l | sort -b | wc > /tmp/wc.out" ) == -1)
      perror( "system ");
```

This executes the commands in the same manner as if they were entered in a UNIX console. Note that because the *system* function invokes Bourne shell as a subshell to execute a *shell_cmd*, any definition of shell variables or change of work directory in a *shell_cmd* is not effective when the *system* function call returns.

The following *mini-shell.C* program emulates a UNIX shell. It takes one or more lines of commands from a user. For each input command line, it calls *system* to execute the command. The program terminates when end-of-file is encountered in the standard input:

```
#include <iostream.h>
#include <stdlib.h>
int main()
{
      char cmd[256];
      for ( ; ; )      {
            /* show a mini-shell prompt */
            cout << "*> " << flush;
            /* Get a user's input. Quit if EOF */
            if ( !cin.getline( cmd, 256 ) ) break;

            /* Execute the user command */
            if ( system( cmd ) == -1 ) perror( cmd );
      }
```

```
        return 0;
    }
```

The following functions are defined in the <stdlib.h> header and convert data from character strings to other C data values, such as double, long, int, etc.:

```
int          atoi (const char* str_val );
double       atof ( const char str_val );
long         atol ( const char* str_val );
double       strtod( const char* str_val, char ** endptr );
long         strtol ( const char* str_val, char** endptr, int radix );
unsigned long strtoul (const char* str_val, char**endptr, int radix );
```

Each of the above functions converts a numerical string specified in *str_val* into its actual data value (float, double, long, or unsigned long) and returns that value. If the *endptr* argument is present and its value is an address of a character pointer, that pointer is set to point to a location in *str_val* where the conversion ends. If the conversion fails, the pointer is set to *str_val,* and the function returns a zero value. The *radix* argument specifies the base of the numerical string stored in *str_val.*

C and C++ provide the *sscanf* function and the *istrstream* class, respectively, to perform operations similar to the above conversion functions. For example. the *atol* function can be written as any one of the following:

```
/* C method */
#include <stdio.h>
long atol ( const char* str_val )
{
    long x;
    if (sscanf( str_val, "%ld", &x ) ==1 )
        return x;
    else return 0;
}

/* C++ method */
#include <strstream.h>
long atol ( const char* str_val )
{
    long x;
    istrstream( str_val,strlen(str_val)+1 ) >> x;
```

```
        return x;
    }
```

The *rand* and *srand* functions, as declared in the <stdlib.h> header, perform random number generation. Their function prototypes are:

```
int rand ( void );
void srand ( unsigned int seed );
```

The *srand* function obtains a *seed* number from the user and sets a starting point for a new sequence of pseudorandom numbers to be returned on each subsequent call of *rand*. The sequence of pseudorandom numbers returned by *rand* may be repeated if *srand* is called again with the same *seed* value. If *rand* is called before *srand*, the default *seed* value is 1. The integer numbers returned by *rand* are in the range of 0 to $2^{15} - 1$. If a user wishes to restrict the pseudo-random numbers returned to be in the range of 1 to *N* (where *N* is any arbitrary positive integer value), the *rand* call may be modified as:

```
int    random_num = rand() % N + 1;
```

The following example function returns a random number that is unique on each call:

```
#include <time.h>
int get_rand()
{
    srand( (unsigned)time( 0 ) );
    return rand();
}
```

In the above example, the *time* function is declared in the <time.h> header. It returns an integer that is the number of seconds elapsed since January 1, 1970 up to the current moment. (This function is described in more detail in a later section when the <time.h> header is discussed). Because the *time* function's return value is unique per call (assuming at least a one-second interval between any two consecutive calls), the *seed* to the *srand* function is also unique and is, thus, the returned random number from *rand*. Random numbers are used extensively in programs that do statistical sampling and analysis.

In addition to the above functions, the <stdlib.h> also declares the following functions that are used in the termination of executed programs:

```
void exit ( int status_code );
int atexit ( void (*cleanup_fnptr)(void) );
void abort ( void );
```

The *exit* function should be familiar to readers, as it is used to terminate a user's program (a process) and returns an integer exit status code to a calling shell. By UNIX convention, a *status_code* value of 0 means that the program's execution was successful. Otherwise, the *status_code* value is nonzero.

The *atexit* function may be called to register a user-defined function. This function takes no argument and does not return any value. The function is called by the *exit* function and is supposed to do clean-up work before the calling process is terminated. Multiple functions may be registered in a process via multiple *atexit* function calls, and these functions are invoked, in an order reversed from that registered, when the containing process calls exit.

The *abort* function is called when a process is in a panic state. The function terminates the process, and in UNIX it causes a *core* file to be generated. A *core* file is useful in aiding users to debug an aborted process.

Finally, the *getenv* function is declared in the <stdlib.h> header. This function allows a process to query a shell environment variable value. There is also a *putenv* function that allows a process to define a shell environment variable. However, the *putenv* function is not defined in ANSI C, even though it is available on most UNIX systems. The function prototypes of the *getenv* and *putenv* functions are:

```
char* getenv ( const char* env_name );
int putenv ( const char*env_def );
```

The *env_name* argument value to a *getenv* call is a character string of a shell environment variable name. The function returns a NULL value if a given environment variable is undefined.

The *env_def* argument value to a *putenv* call is a character string that contains an environment variable name, an equal character, and the value to be assigned to the variable. The function returns a zero value if it succeeds, a nonzero value otherwise.

The following statements show the value of the PATH shell environment variable, then sets an environment variable *CC* to have the value of *c++*:

```
char* env = getenv( "PATH" );
cout << "\"PATH\" value is: " << env << '\n';
if ( putenv( "CC=c++" ) )  cerr << "putenv of CC failed\n";
```

4.3 <string.h>

The <string.h> header declares a set of functions for character string manipulations. These functions are well known to C and C++ programmers and are used in almost every C program that deals with character strings. The commonly used string functions are:

```
int     strlen  (const char* str );
int     strcmp  ( const char* str1, const char* str2);
int     strncmp ( const char* str1, const char* str2, const int n);
char*   strcat  (char* dest, const char* src);
char*   strncat ( char* dest, const char* src, const int n);
char*   strcpy  ( char* dest, const char* src);
char*   strncpy ( char* dest, const char* src, const int n);
char*   strchr  ( const char* str, const char ch);
char*   strrchr ( const char* str, const char ch);
char*   strstr  ( const char* str, const char* key);
char*   strpbrk ( const char* str1, const char* delimit);
```

The uses of these string functions are:

Function	**Use**
strlen	Returns the number of characters of the NULL-terminated *str* argument. The NULL character is not counted in the return value
strcmp	Compares the equality of the *str1* and *str2* arguments. This function returns zero if the two strings are the same, nonzero otherwise
strncmp	Compares up to *n* characters of the *str1* and *str2* argument strings for equality. The function returns zero if the result is a match, nonzero otherwise
strcat	Concatenates the *src* argument string to the *dest* argument string. The resultant *dest* string is appended a NULL character. The function returns the address of the *dest* argument string
strncat	Concatenates up to *n* characters of the *src* argument string to the *dest* argument string. The result-

	ant *dest* string is NULL-terminated. The function returns the address of the *dest* argument string
strcpy	Overrides the content of the *dest* argument string by the *src* argument string, including the terminating NULL character. The function returns the address of the *dest* argument string
strncpy	Overrides the first *n* characters of the *dest* argument string by the *src* argument string. If the *src* argument string's size is equal to or larger than *n*, the NULL character is not copied over. The function returns the address of the *dest* argument string
strchr	Searches the *str* argument for the first occurrence of the *ch* character. The function returns the address of the *ch* character in the *str* string, or NULL if *ch* is not found
strrchr	Searches the *str* argument for the last occurrence of the *ch* character. The function returns the address of the *ch* character in the *str* string or NULL if *ch* is not found
strstr	Searches the *str* argument for the first occurrence of the *key* character string. The function returns the address of the *key* string in *str* sting or NULL if the *key* string is not found
strpbrk	Searches the *str* argument for the occurrence of any character as specified in the *delimit* argument. The function returns the address of the matched character in the *str* string or NULL if there is no match

Besides the above functions, the following sections describe a few useful functions that are not commonly known by C and C++ programmers:

4.3.1 strspn, strcspn

The *strspn* and *strcspn* function prototypes are:

```
const char* strspn ( char*str, const char* delimit);
const char* strcspn ( char*str, const char* delimit);
```

The *strspn* function returns the number of leading characters in *str* that are specified in the *delimit* argument. This function is useful in skipping leading delimiting characters in a string. The following example returns the address of the next nonwhite space character in the input argument *buf*:

```
#include <string.h>
char* skip_spaces ( char* buf )
{
    return buf + strspn( buf, "   \t\n" );
}
```

The *strcspn* function returns the number of leading characters in *str* that are not specified in the *delimit* argument. This function is useful in finding the next delimiting character in a character string. The following example returns the next white-space delimited token in the input argument *buf*:

```
#include <string.h>
char* get_token ( char* buf )
{
    char* ptr = buf + strspn( buf," \n\t" );        // find beginning of a token
    char *endptr = ptr + strcspn( ptr," \n\t" );    // find delimiter after token
    if ( endptr > ptr ) *endptr = '\0';
    if ( *ptr )
        return ptr;                                 // return token
    else return NULL;                               // end of string. No token
}
```

4.3.2 strtok

The *strtok* function prototype is:

```
const char* strtok ( char*str, const char* delimit);
```

This function breaks the *str* argument into one or more tokens. Each token is delimited by characters as specified in the *delimit* argument. If the *str* argument is an address of a character string, the *strtok* function returns the first token in the string. If, however, the *str* argument is a NULL value, the function returns the next token in a previously given string.

This function returns NULL if there are no more tokens to be returned from a string.

The following example breaks a string into tokens that are delimited by white-space characters. Each token obtained is printed to the standard output:

```
#include <iostream.h>
#include <string.h>
int main ( int argc, char* argv[] )
{
    while ( --argc > 0 )
      for ( char* tok; tok = strtok( argv[argc], "  \n\t" ); argv[argc]= 0 )
          cout << "tok: " << tok << endl;
}
```

Note that the *strtok* function modifies the input *str* argument by replacing delimiting characters after tokens in *str* with the NULL character. Users who wish to reuse character strings to be parsed by *strtok* should make a copy of these strings so that they can use the copied strings later.

The following example illustrates a possible implementation of the *strtok* function. Note that the function has a static pointer, *lptr,* to remember where to parse the next token in either a new or an old string. Furthermore, if the function finds a delimiting character after a token, it replaces the delimiter by a NULL character. Thus, the function modifies the input character string as it extracts tokens from it.

```
#include <string.h>
char* my_strtok ( char* str, const char* delimit )
{
    static char* lptr;
    if ( tr )  {                               // start parsing a new string
      str += strspn( str,delimit );            // skip leading delimiters
      if (!*str ) return NULL;                 // done if it is a NULL string
      lptr = str;
    }
    else ( !lptr )                             // continue to parse old string
      return NULL;                             // return if no more token

    char* tokn = lptr + strspn( lptr, "  \t\n" );  // skip leading delimiter
    lptr = tokn + strcspn( tokn,"    \t\n" );      // find next delimiter
    if ( *tokn && lptr > tokn )
      *lptr++ = '\0';                          // NULL-terminate token
```

```
        else lptr = NULL;                    // find last token in a string
        return *tokn ? tokn : NULL;          // return token, if any
    }
```

4.3.3 strerror

The *strerror* function prototype is:

```
const char* strerror ( int errno );
```

This function can be used to get a system diagnostic message. The *errno* argument value may be any error code as defined in the <sys/errno.h> header file, or may be the global *errno* variable. The global *errno* variable is set whenever a system API is called, and its value is zero if the API execution is successful, nonzero otherwise.

The return character string is read-only and should not be deallocated by users.

The *perror* function may be called to print a system diagnostic message if any system API call fails. The *strerror* function allows users to define their own version of the *perror* function.

The following example depicts one possible implementation of the *perror* function using *strerror:*

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

void my_perror ( const char* msg_header )
{
    if (msg_header && strlen(msg_header)) // print if it is defined by a user
       cerr << msg_header << ": " << strerror( errno ) << endl;
    else cerr << strerror(errno ) << endl;
}

/* test program for my_perror function */
int main( int argc, char* argv[] )
{
    FILE *fp;
    while ( --argc > 0 )                     // for each cmd line argument
```

```
            if ( (fp = fopen( *++argv, "r" )) )        // fp=0 if open fails
               my_perror( *argv );                     // print a diagnostic
            else fclose ( fp );                        // close file if is opened OK
        return 0;
    }
```

4.4 <memory.h>

The <memory.h> header declares a set of functions for byte stream manipulations. These functions are more similar to the string functions, except that they have a more general purpose and can be used for noncharacter string object manipulation. For example, one can use these functions to initialize, compare, and copy struct-typed objects.

The functions declared in the <memory.h> header are:

```
void*    memset   (const void* memp, int ch, size_t len );
int      memcmp   ( const void* mem1, const void* mem2, size_t len );
void*    memcpy   (void* dest, const void* src, size_t len );
void*    memccpy  ( void* dest, const void* src, const int ch, size_t len);
void*    memchr   ( const void* memp, const int ch, size_t len );
```

The *memset* function initializes the first *len* bytes of a memory region pointed to by *memp*. The memory is initialized with the *ch* byte throughout. The function returns the address of *memp*.

The following statements illustrate the initialization of a *struct stat*-typed variable to contain all NULL data:

```
struct stat *statp = new stat;
if (statp)
    (void)memset( (void*)statp, NULL, sizeof( struct stat ) );
```

The *bzero* function in BSD UNIX initializes a memory region to all zero bytes. This function may be implemented via the *memset* function as:

```
void bzero ( char *memp, int len )
{
     (void) memset( memp, NULL,(size_t)len );
}
```

The *memcmp* function compares the equality of the first *len* bytes of two memory regions pointed to by *mem1* and *mem2*. The function returns zero if the two memory regions are identical in the first *len* bytes, a positive value if the *mem1* region contains data that are lexicographically greater than those of *mem2*, or a negative value if the *mem1* region contains data that are lexicographically less than those of *mem2*.

The following statements compare the equality of two *struct stat*-typed variables:

```
int cmpstat ( struct stat* statp1, struct stat* statp2 )
{
        return memcmp( (void*)statp1, (void*)statp2, sizeof( struct stat ) );
}
```

The *strcmp* function may be implemented via the *memcmp* function as follows:

```
int my_strcmp (const char* str1, const char* str2 )
{
        int len1 = strlen( str1 ), len2 = strlen( str2 );
        if ( len1 > len2 ) len1 = len2;
        return memcmp( (void*)str1, (void*)str2, len1 );
}
```

Furthermore, the BSD UNIX *bcmp* function may also be implemented via the *memcmp* function:

```
#define bcmp ( s1, s2, n )    memcmp( (void*)s1, (void*)s2, (size_t)n )
```

The *memcpy* function copies the first *len* bytes of data from the memory region pointed to by *src* to the memory region pointed to by *dest*. The function returns the address of the *dest* memory region.

The BSD UNIX *bcopy* function may also be implemented via the *memcpy* function:

```
#define bcopy ( src, dest, n ) memcpy( (void*)dest, (void*)src, (size_t)n )
```

Note that the *bcopy, bcmp,* and *bzero* functions are not defined in the ANSI C standard, but they are widely used in UNIX system programs.

The *memcpy* function may be used to implement the *strcpy* function:

```
#define strcpy( dest, src )    \
        memccpy( (void*)dest, (void*)src, '\0', (size_t)strlen( src ) )
```

The *memccpy* function copies data from a memory region pointed to by *src* to a memory region pointed to by *dest*. The function either copies the first *len* bytes of data from *src* to *dest* or wait until the *ch* byte that is found within the first *len* byte of *src* is copied to *dest*.

The *memccpy* function may be used to implement the *strncpy* function:

```
#define strncpy( dest, src, n ) \
            memccpy ( (void*)dest, (void*)src, '\0', (size_t)n )
```

Finally, the *memchr* function searches the first *len* byte of a memory region pointed to by *memp* and returns the address of the first occurrence of *ch* in that region, or NULL if there is no match. The *memchr* function may be used to implement the *strchr* function:

```
#define strchr(str,ch)      memchr( (void*)str, ch, (size_t)strlen( str ) )
```

4.5 <malloc.h>

The <malloc.h> header declares a set of functions for dynamic memory allocation and disposal. These functions are not used extensively by C++ programmers, as they use the *new* and *delete* operators to perform the same functions. However, the *realloc* function declared in the <malloc.h> header can be used to adjust the size of any dynamic memory, and this feature is not provided by the C++ *new* operator. This section explains in detail the use of *realloc*.

The functions declared in the <malloc.h> header are:

```
void*   malloc (const size_t size );
void*   calloc ( const size_t num_record, const size_t size_per_record );
void    free    ( void* memp );
void*   realloc ( void* old_memp, const size_t new_size );
```

The uses of *malloc, calloc,* and *free* should be familiar to users already. Specifically, the following statements both allocate a dynamic memory of size 1048 bytes:

```
char* mem1 = (char*)malloc( 1048 ) );        // C style
```