

Systems Programming in C++
CS330

Nathan Warner



**Northern Illinois
University**

Computer Science
Northern Illinois University
March 29, 2024
United States

Contents

1	C Library Functions	3
1.1	cstdlib utils	3
1.2	getenv	3
1.3	exit	4
1.4	system	4
1.5	Perror	4
1.5.1	Signature	5
1.5.2	Behavior	5
1.5.3	Setting errno manually	5
2	Posix regex API <regex.h>	6
2.1	regcomp	6
2.1.1	Signature	6
2.2	Regexec	6
2.2.1	Signature	6
2.3	Regerror	7
2.3.1	Signature	7
2.4	Regfree	7
2.4.1	Signature	7
2.5	regmatch_t and pmatch	8
2.5.1	regmatch_t	8
2.5.2	pmatch array	9
2.6	Regex Example	9
3	Directory Input/Output <dirent.h>	12
3.1	Functions	12
3.2	The DIR type	12

3.3	dirent Structure	12
3.3.1	Example	13
4	Unix System Calls <sys/stat.h>	14
4.1	System Call Categories	14
4.1.1	File management	14
4.1.2	Process Control	14
4.1.3	Communication	14
4.1.4	Device management	14
4.2	System Call Invocation	14
5	File Management	15
5.1	Open	15
5.1.1	Additonal flags, used with O_WRONLY	15
5.2	Open with mode	16
5.2.1	Example	16
5.3	Read	16
5.4	Close	16
5.5	Write	17
5.6	Creat	17
6	I/O Management	18
6.1	Unlink	18
6.2	Dup	18
6.3	Stat	18
6.3.1	stat structure st_mode field	19
6.4	Chmod	19
6.5	Fchmod	20
6.5.1	Permission modes	20

C Library Functions

1.1 cstdlib utils

1.2 getenv

Concept 1: The **getenv** function is a standard library function that provides access to the environment variables of the process. Environment variables are dynamic-named values that affect the processes running on a computer. They can be used to configure system settings, pass configuration data to applications, and enable communication between different parts of an operating system or between different programs.

Signature

```
1 char* getenv(const char* name);
```

Where name is a C-string (const char*) representing the name of the environment variable whose value is being requested.

Return Value

If the environment variable is found, getenv returns a pointer to a C-string containing the value of the variable.

If the environment variable is not found, it returns a null pointer (NULL in C, nullptr in C++).

Example

```
1  #include <cstdlib>
2  #include <iostream>
3
4  int main() {
5      // Attempt to retrieve the PATH environment variable
6      const char* path = getenv("PATH");
7      if (path != nullptr) {
8          std::cout << "PATH: " << path << std::endl;
9      } else {
10         std::cout << "PATH environment variable not found." <<
↪         std::endl;
11     }
12     return 0;
13 }
```

1.3 exit

Signature

```
1 void exit(int status)
```

The exit function is quite simple, it terminates the calling process

Zero for a successful termination, anything else is unsuccessful termination.

1.4 system

Signature

```
1 int system(const char* command)
```

The system command allows us to run shell commands. It invokes the command processor to execute a command. The function returns the exit status of the command.

Note: If command is a nullptr, the function only checks if a command processor is available.

Example

```
1 int main(int argc, const char* argv[]) {  
2     int rs;  
3     if (!system(NULL)) {  
4         exit(EXIT_FAILURE);  
5     }  
6     cout << system("ls -la");  
7  
8     return EXIT_SUCCESS;  
9 }
```

1.5 Perror

Concept 2: The perror function in C is a standard library function that prints a descriptive error message to the standard error stream (stderr). The message corresponds to the current value of the global variable errno, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

1.5.1 Signature

```
1 void void perror(const char *s);
```

- **s:** A C string containing a custom message to be printed before the error message itself. If this argument is not NULL, the string pointed to by s is printed first, followed by a colon (:) and a space. If s is an empty string or NULL, only the error message is printed.

1.5.2 Behavior

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The actual error message printed by **perror** is system-dependent but generally reflects the error state represented by **errno**.

1.5.3 Setting **errno** manually

You can set **errno** simply by assigning it a value. First, you need to include the **errno.h** header to ensure that you have access to the **errno** variable and the standard error codes.

```
1 #include <errno.h>
2
3 errno = ENOENT; // No such file or directory
```

Posix regex API <regex.h>

2.1 regcomp

Concept 3: Compiles a regular expression into a format that the `regexexec()` function can use to perform pattern matching.

2.1.1 Signature

```
1 int regcomp(regex_t *preg, const char *regex, int cflags)
```

- **preg:** A pointer to a `regex_t` structure that will store the compiled regular expression.
- **regex:** The regular expression to compile.
- **cflags:** Compilation flags that modify the behavior of the compilation. Common flags include `REG_EXTENDED` (use extended regular expression syntax), `REG_ICASE` (ignore case in match), `REG_NOSUB` (don't report the match), and `REG_NEWLINE` (newline-sensitive matching).

2.2 Regexexec

Concept 4: After compiling a regular expression, we can use `regexexec` to match against strings.

2.2.1 Signature

```
1 int regexexec(const regex_t *preg, const char *string, size_t  
  ↪ nmatch, regmatch_t pmatch[], int eflags)
```

- **preg:** The compiled regular expression.
- **string:** The string to match against the regular expression.
- **nmatch:** The maximum number of matches and submatches to find.
- **pmatch:** An array of `regmatch_t` structures that will hold the offsets of matches and submatches.
- **eflags:** Execution flags that modify the behavior of the match. A common flag is `REG_NOTBOL` which indicates that the beginning of the specified string is not the beginning of a line.

2.3 Regerror

Concept 5: This function translates error codes from `regcomp()` and `regexexec()` into human-readable messages.

2.3.1 Signature

```
1  size_t regerror(int errcode, const regex_t *preg, char *errbuf,  
    ↪ size_t errbuf_size)
```

- **errcode:** The error code returned by `regcomp()` or `regexexec()`.
- **preg:** The compiled regular expression (if the error is related to `regexexec()`).
- **errbuf:** The buffer where the error message will be stored.
- **errbuf_size:** The size of the buffer.

2.4 Regfree

Concept 6: Frees the memory allocated to the compiled regular expression.

2.4.1 Signature

```
1  void regfree(regex_t *preg)
```

- **preg:** The compiled regular expression to free.

2.5 regmatch_t and pmatch

2.5.1 regmatch_t

regmatch_t is a structure used to describe a single match (or submatch) found by regexec(). It contains at least the following two fields:

- **rm_eo:** This is the end offset of the match, which is one more than the index of the last character of the match. In other words, rm_eo - rm_so gives the length of the match.
- **rm_so:** This is the start offset of the match, relative to the beginning of the string passed to regexec(). If the match is successful, rm_so will be the index of the first character of the match.

2.5.2 pmatch array

When you call `regexec()`, you can pass it an array of `regmatch_t` structures as the `pmatch` argument. This array is where `regexec()` will store information about the matches (and sub-matches) it finds. The size of this array (`nmatch`) determines how many matches `regexec()` will look for and fill in. The zeroth element of this array corresponds to the entire pattern's match, and the subsequent elements correspond to parenthesized subexpressions (sub-matches) within the regular expression, in the order they appear.

2.6 Regex Example

```

1     regex_t regex;
2     int reti;
3     char msgbuf[100];
4     regmatch_t pmatch[1]; // Array to store the match positions
5     const char* search = "abc";
6
7     // Compile regular expression
8     reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
9     if (reti) {
10         fprintf(stderr, "Could not compile regex\n");
11         exit(EXIT_FAILURE);
12     }
13
14     // Execute regular expression
15     // Note: Changed the third argument to 1 to indicate we want
    ↪ to capture up to 1 match
16     // and the fourth argument to pmatch to store the match
    ↪ position.
17     reti = regexec(&regex, search, 1, pmatch, 0);
18     if (!reti) {
19         printf("Match\n");
20         // If you want to use the match information, you can do
    ↪ so here.
21         // For example, to print the start and end positions of
    ↪ the match:
22         printf("Match at position %d to %d\n",
    ↪ (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
23     }
24     else if (reti == REG_NOMATCH) {
25         printf("No match\n");
26     }
27     else {
28         regerror(reti, &regex, msgbuf, sizeof(msgbuf));
29         fprintf(stderr, "Regex match failed: %s\n", msgbuf);
30         exit(EXIT_FAILURE);
31     }
32     regex_t regex;
33     int reti;
34     char msgbuf[100];
35     regmatch_t pmatch[1]; // Array to store the match positions
36
37     // Compile regular expression
38     reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
39     if (reti) {
40         fprintf(stderr, "Could not compile regex\n");
41         exit(EXIT_FAILURE);

```

```

1 // Execute regular expression
2 // Note: Changed the third argument to 1 to indicate we want to
   ↳ capture up to 1 match
3 // and the fourth argument to pmatch to store the match position.
4 reti = regexec(&regex, "abc", 1, pmatch, 0);
5 if (!reti) {
6     printf("Match\n");
7     // If you want to use the match information, you can do so
   ↳ here.
8     // For example, to print the start and end positions of the
   ↳ match:
9     printf("Match at position %d to %d\n", (int)pmatch[0].rm_so,
   ↳ (int)pmatch[0].rm_eo - 1);
10 }
11 else if (reti == REG_NOMATCH) {
12     printf("No match\n");
13 }
14 else {
15     regerror(reti, &regex, msgbuf, sizeof(msgbuf));
16     fprintf(stderr, "Regex match failed: %s\n", msgbuf);
17     exit(EXIT_FAILURE);
18 }
19
20 // Free the compiled regular expression
21 regfree(&regex);
22
23 for (int i=(int)pmatch[0].rm_so; i<=(int)pmatch[0].rm_eo;
   ↳ ++i) {
24     cout << search[i];
25 }
26 cout << endl;
27
28 regfree(&regex);

```

Directory Input/Output <dirent.h>

3.1 Functions

- **chdir(const char *path)** \mapsto **int**: Changes the current working directory of the calling process to the directory specified in **path**. Returns zero on success, and -1 on failure, setting **errno** to indicate the error.
- **getcwd(char *buf, size_t size)** \mapsto **char***: Copies an absolute pathname of the current working directory to the array pointed to by **buf**, which is of length **size**. If **size** is large enough, returns **buf**; if **size** is too small, NULL is returned, and **errno** is set to **ERANGE**; on other errors, NULL is returned, and **errno** is set appropriately.
- **opendir(const char *name)** \mapsto **DIR***: Opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory. Returns NULL if an error occurs, setting **errno** to indicate the error.
- **readdir(DIR *dirp)** \mapsto **struct dirent***: Reads the next directory entry from the directory stream pointed to by **dirp**. Returns a pointer to a **struct dirent** representing the next directory entry, or NULL when reaching the end of the directory stream or if an error occurs.

3.2 The DIR type

The DIR type is an opaque data type that represents a directory stream. A directory stream is an ordered sequence of all the directory entries in a particular directory. Directory entries include files, subdirectories, and other types of file system objects contained in the directory.

You cannot see the structure of DIR directly, as it is hidden (opaque) to provide abstraction and portability across different operating systems and filesystems. This means you interact with directory streams using pointer variables of type DIR *, and you manipulate these streams through the functions provided by <dirent.h>, such as opendir(), readdir(), and closedir().

3.3 dirent Structure

The struct dirent structure represents an individual directory entry, which could be a file, a subdirectory, or another type of file system object. The exact contents of this structure can vary between different operating systems, but it generally includes the following members:

- **d_ino**: The inode number of the directory entry. The inode number is a unique identifier within a filesystem.
- **d_name**: A character array (string) containing the name of the directory entry. This name is not the full path, but just the filename or directory name.

3.3.1 Example

```
1  DIR* d;
2  struct dirent* dir;
3  d = opendir("."); // Open the current directory
4
5  if (d) {
6      while ((dir = readdir(d)) != NULL) {
7          printf("%s\n", dir->d_name); // Print the name of each
↪      directory entry
8      }
9      closedir(d); // Close the directory stream
10 }
11
12 return 0;
13
```

Unix System Calls <sys/stat.h>

Concept 7: A system call is how a program requests services from the operating system. System calls execute code in the kernel and makes direct use of facilities provided by the kernel. Versus a library function which is linked to the executable, hence it becomes part of the executable.

4.1 System Call Categories

4.1.1 File management

- Create/delete file, open/close, read/write, get/set attributes

4.1.2 Process Control

- Create/terminate process, wait/signal event, allocate/free memory

4.1.3 Communication

- Create/delete connection, send/recieve messages, remote devices

4.1.4 Device management

- Attach/request/release/detach device, read/write/position

4.2 System Call Invocation

- Declare system call via appropriate C header file
- Prepare parametrs using basic C data types
- Prepare suitable return value variable

Then we call like any other function.

File Management

- **Open:** Open a file
- **Read:** Read data from a file
- **Write:** Write data to a file
- **Close:** Close a file
- **creat:** Make a new file

Note:-

All calls share file descriptor, ie number to identify file

5.1 Open

```
1 int open(const char* pathname, int flags)
```

- Opens file specified as **pathname** for access
- Flags determine access type
 - **O_RDONLY**: Read only
 - **O_WRONLY**: Write only
 - **O_RDWR**: Read and write

Note:-

Returns file descriptor, to be used in read/write/close, returns -1 on error

5.1.1 Additional flags, used with **O_WRONLY**

- **O_APPEND**: To append to an existing file
- **O_TRUNC**: existing file will be overwritten (Default)
- **O_CREAT**: Creates file, if file does not exist

Example:

```
1 O_WRONLY | O_TRUNC
2 O_WRONLY | O_APPEND
```


5.2 Open with mode

```
1  int open(const char* pathname, int flags, mode_t mode)
```

The parameter **mode** is used to specify permissions of type `mode_t`

- **S_IRWXU (00700)**: User has read, write, and execute permissions
- **S_IRUSR (00400)**: User has read permissions
- **S_IWUSR (00200)**: User has read, write, and execute permissions
- **S_IXUSR (00100)**: User has write permissions
- **S_IRWXG (00070)**: User has execute permissions
- **S_IRWXO (00007)**: Others have read, write, and execute permissions

5.2.1 Example

```
1  open("ex.txt", O_WRONLY | O_APPEND | O_CREAT, 00666)
```

5.3 Read

```
1  ssize_t read(int fd, void* buf, size_t count)
```

- Attempts to read **count** bytes from file descriptor **fd** into the buffer starting at **buf**
 - **Note:** `ssize_t` is like `size_t` can also be -1
- Returns the number of bytes read
 - May be smaller than count, zero indicates end of file
 - file position is advanced by this number

Note:-

Returns -1 on error

5.4 Close

```
1  int close(int fd)
```

- Closes file specified by **fd** file descriptor, makes file descriptor available

Note:-

Returns 0 on success

5.5 Write

```
1 ssize_t write(int fd, const void* buf, size_t count)
```

- Writes up to **count** bytes from buffer starting at **buf** to the file referred to by file descriptor **fd**
- Returns the number of bytes written

Note:-

Returns -1 on error

5.6 Creat

```
1 int creat(const char* pathname, mode_t mode)
```

- Creates new file specified as pathname and opens file for write access
- **mode** specifies permissions of type mode_t
- returns file descriptor

Note:-

Returns -1 on error

I/O Management

- **unlink**: Remove file
- **dup**: Duplicate file descriptor
- **stat**: Get file information
- **chmod**: Change permissions

6.1 Unlink

```
1 int unlink(const char* pathname)
```

- Removes a **pathname** from the file system
- If **pathname** was the last link to a file, then it is deleted
- If **pathname** refers to a symbolic link, then it is removed
- Returns zero on success

Note:-

Returns -1 on error

6.2 Dup

```
1 int dup(int oldfd)
```

- Creates a copy of file descriptor **oldfd**
- Uses lowest-numbered unused descriptor
- Returns a new file descriptor

Note:-

Returns -1 on error

6.3 Stat

Concept 8: Family of system calls to inquire about a file

```
1 int stat(const char* path, struct stat* buf) // Takes pathname
2 int fstat(int fd, struct stat* buf) // Takes file descriptor
3 int lstat(const char* path, struct stat* buf) // Reports on
   ↳ symbolic link as is
```

```

1  struct stat {
2      dev_t      st_dev;          /* ID of device containing file
   ↪  */
3      ino_t      st_ino;          /* inode number */
4      mode_t     st_mode;         /* file mode: contains
   ↪  permissions */
5      nlink_t    st_nlink;        /* number of hard links * /
6      uid_t      st_uid;          /* user ID of owner */
7      gid_t      st_gid;          /* group ID of owner * /
8      dev_t      st_rdev;         /* device ID (if special file) */
9      off_t      st_size;         /* total size, in bytes */
10     blksize_t   st_blksize;      /* blocksize for file system I/O
   ↪  */
11     blkcnt_t    st_blocks;        /* number of blocks allocated */
12     time_t      st_atime;         /* time of last access */
13     time_t      st_mtime;         /* time of last modification */
14     time_t      st_otime;         /* time of last status change */
15 };

```

6.3.1 stat structure st_mode field

This contains the file mode, including permissions

To check permissions:

- `st_mode & S_IRUSR`: User has read permissions
- `st_mode & S_IWUSR`: User has write permissions
- `st_mode & S_IXUSR`: User has execute permissions

To check file type:

- `S_ISREG(st_mode)`: It is a regular file
- `S_ISDIR(st_mode)`: It is a directory
- `S_ISLNK(st_mode)`: It is a symbolic link

6.4 Chmod

```

1  int chmod(const char* path, mode_t mode)

```

- Change permission settings for file given in **path** string
- new file permissions are specified in **mode**
- Returns zero on success, or -1 on error

Note:-

Must be called by owner of file, or superuser, returns -1 on error

6.5 Fchmod

```
1 int fchmod(int fd, mode_t mode)
```

- Change permission settings for file given in **fd**
- new file permissions are specified in **mode**
- Returns zero, or -1 on error

Note:-

Must be called by owner of file, or superuser, returns -1 on error

6.5.1 Permission modes

Permission mode

S_ISUID	(04000)	set-user-ID
S_ISGID	(02000)	set-group-ID
S_ISVTX	(01000)	sticky bit
S_IRUSR	(00400)	read by owner
S_IWUSR	(00200)	write by owner
S_IXUSR	(00100)	execute/search owner
S_IRGRP	(00040)	read by group
S_IWGRP	(00020)	write by group
S_IXGRP	(00010)	execute/search group
S_IROTH	(00004)	read by others
S_IWOTH	(00002)	write by others
S_IXOTH	(00001)	execute/search by others

mode bit mask is created by OR-ing together several of these constants:

S_IRUSR | S_IWUSR | S_IXUSR
S_IRUSR | S_IRGRP | S_IROTH

or:

00755

00644