Systems Programming in C++

CS330

Nathan Warner

Computer Science
Northern Illinois University
March 29, 2024
United States

Contents

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

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

Where name is a 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

```
#include <cstdlib>
#include <iostream>

int main() {
    // Attempt to retrieve the PATH environment variable
    const char* path = getenv("PATH");
    if (path != nullptr) {
        std::cout << "PATH: " << path << std::endl;
    } else {
        std::cout << "PATH environment variable not found." <<
    ↪ std::endl;
    }
    return 0;
}
```

1.3 exit

Signature

```
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

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

The system command allows us to run shell commands. It invokes the command processor to execute a comamnd. 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

```
int main(int argc, const char* argv[]) {
    int rs;
    if (!system(NULL)) {
        exit(EXIT_FAILURE);
    }
    cout << system("ls -la");

    return EXIT_SUCCSESS;
}
```

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

```c
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.

```c
#include <errno.h>

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 **regexec()** function can use to perform pattern matching.

2.1.1 Signature

```
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 Regexec

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

2.2.1 Signature

```
int regexec(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 regexec() into human-readable messages.

2.3.1 Signature

```c
size_t regerror(int errcode, const regex\_t *preg, char *errbuf,
    size\_t errbuf\_size)
```

- **errcode:** The error code returned by regcomp() or regexec().
- **preg:** The compiled regular expression (if the error is related to regexec()).
- **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

```c
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 submatches) 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 (submatches) within the regular expression, in the order they appear.

2.6 Regex Example

```
    regex_t regex;
    int reti;
    char msgbuf[100];
    regmatch_t pmatch[1]; // Array to store the match positions
    const char* search = "abc";

    // Compile regular expression
    reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
    if (reti) {
        fprintf(stderr, "Could not compile regex\n");
        exit(EXIT_FAILURE);
    }

    // Execute regular expression
    // Note: Changed the third argument to 1 to indicate we want to capture up to 1 match
    // and the fourth argument to pmatch to store the match position.
    reti = regexec(&regex, search, 1, pmatch, 0);
    if (!reti) {
        printf("Match\n");
        // If you want to use the match information, you can do so here.
        // For example, to print the start and end positions of the match:
        printf("Match at position %d to %d\n", (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
    }
    else if (reti == REG_NOMATCH) {
        printf("No match\n");
    }
    else {
        regerror(reti, &regex, msgbuf, sizeof(msgbuf));
        fprintf(stderr, "Regex match failed: %s\n", msgbuf);
        exit(EXIT_FAILURE);
    }regex_t regex;
int reti;
char msgbuf[100];
regmatch_t pmatch[1]; // Array to store the match positions

// Compile regular expression
reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
if (reti) {
    fprintf(stderr, "Could not compile regex\n");
    exit(EXIT_FAILURE);
}
```

```
// Execute regular expression
// Note: Changed the third argument to 1 to indicate we want to capture up to 1 match
// and the fourth argument to pmatch to store the match position.
reti = regexec(&regex, "abc", 1, pmatch, 0);
if (!reti) {
    printf("Match\n");
    // If you want to use the match information, you can do so here.
    // For example, to print the start and end positions of the match:
    printf("Match at position %d to %d\n", (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
}
else if (reti == REG_NOMATCH) {
    printf("No match\n");
}
else {
    regerror(reti, &regex, msgbuf, sizeof(msgbuf));
    fprintf(stderr, "Regex match failed: %s\n", msgbuf);
    exit(EXIT_FAILURE);
}

// Free the compiled regular expression
regfree(&regex);

    for (int i=(int)pmatch[0].rm_so; i<=(int)pmatch[0].rm_eo; ++i) {
        cout << search[i];
    }
    cout << endl;

    regfree(&regex);
```

Directory Input/Output <dirent.h>

3.1 Functions

- **chdir(const char *path)** ↦ `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)** ↦ `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)** ↦ `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)** ↦ `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

```c
DIR* d;
struct dirent* dir;
d = opendir("."); // Open the current directory

if (d) {
    while ((dir = readdir(d)) != NULL) {
        printf("%s\n", dir->d_name); // Print the name of each directory entry
    }
    closedir(d); // Close the directory stream
}

return 0;
```

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. Verus 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

```c
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 Additonal 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:

```c
O_WRONLY | O_TRUNC
O_WRONLY | O_APPEND
```

5.2 Open with mode

```c
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

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

5.3 Read

```c
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

```c
int close(int fd)
```

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

Note:-

Returns 0 on success

5.5 Write

```
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

```
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

```c
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

```c
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

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

```c
struct stat  {
    dev_t       st_dev;         /* ID of device containing file */
    ino_t       st_ino;         /* inode number */
    mode_t      st_mode;        /* file mode: contains permissions */
    nlink_t     st_nlink;       /* number of hard links  * /
    uid_t       st_uid;         /* user ID of owner */
    gid_t       st_gid;         /* group ID of owner  * /
    dev_t       st_rdev;        /* device ID (if special file) */
    off_t       st_size;        /* total size, in bytes */
    blksize_t   st_blksize;     /* blocksize for file system I/O */
    blkcnt_t    st_blocks;      /* number of blocks allocated */
    time_t      st_atime;       /* time of last access */
    time_t      st_mtime;       /* time of last modification */
    time_t      st_otime;       /* time of last status change */
};
```

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_IFLNK(st_mode)**: It is a symbolic link

6.4 Chmod

```c
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

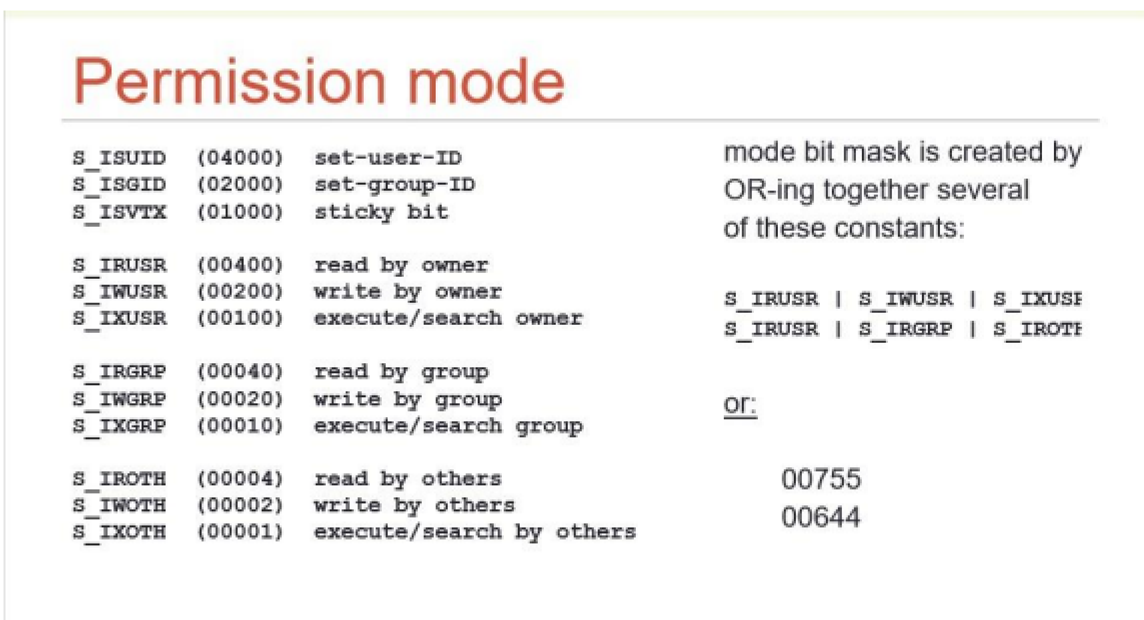
```
int fchmod(int fd, mode_t mode)
```

- 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



Processes and Pipe

7.1 System call: fork

Concept 9: Fork creates a new process that is a duplicate of a current process. The new process is almost the same as current process.

New process is **child** of current process. Old process is **parent** of new process

After the call to fork, both processes run concurrently

7.1.1 Signature

```
pid_t fork(void);
```

7.1.2 Return values

- **Parent:** fork returns process id of child process
- **Child:** fork returns zero
- fork returns -1 on failure

7.1.3 Example

```
cout << "Before fork\n";

pid_t pid = fork();

if (pid == -1) {
    perror("Error: ");
    exit(EXIT_FAILURE);
}
cout << "After fork\n";
cout << "Hello world" << endl;

/* Output:
    Before fork
    After fork
    Hello world
    After fork
    Hello world
*/
```

7.1.4 Example with typical branching logic

```c
pid_t pid = fork();
if (pid == 0) {
    // Child code here
} else {
    // Parent code here
}
```

7.2 System call: wait

Concept 10: Wait lets parent process wait until a child process terminates, parent is resumed onec child process terminates

7.2.1 Signature

```c
pid_t wait(int* status);
```

Where *status* holds exit status of child

7.2.2 WEXITSTATUS(status)

This function allows us to examine *status* (the parameter in the fork call).

7.2.3 Returns

Wait returns process id of terminated child, or -1 if there is no child to wait for.

7.2.4 Example

```
pid_t pid, status;

cout << "Before fork\n";

fork();

pid = wait(&status)

if (pid == -1) {
    cout << "Nothing to wait for \n";
} else {
    cout << "Done waiting for: " << pid << endl;
}

cout << "After fork\n";

/* Output:
     Before fork
     Nothing to wait for
     After fork
     Done waiting for: 66983
     After fork
*/
```

7.2.5 Understanding the output of the example

1. **Before the fork() call:** The program begins execution and prints ”Before fork”. At this point, there is only one process running—the parent process.

2. **The fork() system call:** This call creates a new process, referred to as the child process. After fork() is executed, there are now two processes in execution: the parent and the child. Both processes will execute the code following the fork() call, but in their separate memory spaces.

3. **The wait(status) system call:**

   - This is used by a process to wait for one of its child processes to exit or to be terminated. The call also retrieves the exit status of the child.
   - **In the parent process:** The wait() call will block the parent process until the child process finishes its execution. The pid variable will receive the process ID of the terminated child. Therefore, the parent process will print ”Done waiting for: ”, followed by the PID of the child process.
   - **In the child process:** There is no call to fork(), so wait() immediately returns with a value of -1, indicating that there are no child processes to wait for (since the child process itself hasn’t created any child processes). As a result, the child process prints ”Nothing to wait for ”.

4. **After the wait() call:** Both processes continue execution. The child process, having printed "Nothing to wait for ", will print "After fork" and then terminate. The parent process, after waiting for the child to terminate and printing the message indicating it is done waiting, will also print "After fork" before finishing execution.

7.3 System call: exec

Concept 11: Family of functions that replace current process image with a new process image.

The **actual** system call is *execve*

7.3.1 Function

- **execl**: Specify arguments and environment as list
- **execlp**: Specify arguments and environment as list, look for new executable via PATH
- **execle**: Specify arguments and environment as list
- **execv**: Specify arguments and environment as array of string values
- **execvp**: Specify arguments and environment as array of string values, look for new executable via PATH

7.3.2 Signatures

```
int execl(const char *path, const char *arg, ..., NULL);
int execle(const char *path, const char *arg, ..., NULL, char *
 ↪  const envp[]);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ..., NULL);
int execvp(const char *file, char *const argv[]);
```

7.3.3 Returns

These functions return -1 on error and does not return on success

Note:-

Does not return on success because the calling process's image is entirely replaced

7.3.4 Example: execl

```
int rs;
cout << "Proram started in process: " << getpid() << endl;

rs = execl("/bin/ps", "ps", (char*) NULL);

if (rs == -1) {
    perror("excel");
    exit(rs);
}
cout << "Maybe we see this?\n";
```

7.4 getpid() and getppid() functions

getpid() returns the process ID of the current process. It never throws any error therefore is always successful.

getppid() returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for init process.

7.4.1 Signatures

```
#include <unistd.h>
pid_t getpid(void)
pid_t getppid(void)
```

7.4.2 Example

```
int pid = fork();
if (pid == 0) {
    cout << "\nCurrent process id of Process : "
         << getpid() << endl;
}
```

7.5 Together: fork and exec

Unix does not have a single system call to spawn a new additional process with a new executable, instead

1. fork to duplicate current process
2. exec to morph child process into new executable

7.5.1 Example

```
int rs, pid, status;

pid = fork();
if (pid == -1) {
    perror("fork");
    exit(pid);
}
if (pid == 0) {  // Child process
    rs = execvp("echo", argv);
    if (rs == -1) {
        perror("execvp");
        exit(rs);
    }
} else { // Parent process
    cout << "Done waiting for: " << wait(&status) << endl;
}
```

7.6 Unix Pipe: Pipe system call

The **software pipeline** is a set of processes chained by their standard IO. The output of one process becomes the input of second process

Implemented via **pipe** system call.

7.6.1 signature

```
int pipe(int pipefd[2])
```

This system call has two directions: one side to write, one side to read

- **read side**: pipefd[0]
- **write side**: pipefd[1]

7.6.2 Example

```
cout << "Before pipe\n";

int pipefd[2], rs;

rs=pipe(pipefd);
if (rs == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}

write(pipefd[1],"Hello", 6);

char buffer[256];
read(pipefd[0], buffer, sizeof(buffer));

cout << "pipe contained: " << buffer << endl;
```

7.7 Process communication: pipe and fork

Idea: read and write end of pipe in different processes

Fork creates two processes

- **Parent process**:
  - Close read end of pipe
  - Write to write end of pipe
- **Child process**:
  - close write end of pipe
  - read from read end of pipe

7.7.1 Example

```
int pipefd[2], rs;
char buffer[256];

// create pipe
rs = pipe(pipefd);
if (rs == -1) { perror("pipe"); exit(EXIT_FAILURE); }
cout << "pipe created\n";

// fork into 2 processes
rs = fork();
if (rs == -1) { perror("fork"); exit(EXIT_FAILURE); }

if (rs == 0) { // child process
    // close write end of pipe
    close(pipefd[1]);
    // read from read end of pipe
    read(pipefd[0], buffer, sizeof(buffer));
    cout << "Child: pipe contained: " << buffer << endl;
} else {  // parent process
    // close read end of pipe
    close(pipefd[0]);
    // write to write end of pipe
    write(pipefd[1],"Hello", 6);
    wait(NULL);
    cout << "parent resumes after wait for child\n";
}
```

7.8 Implementing unix pipe, goal: ls | wc

```

    // create pipe
        int pipefd[2];
        int rs = pipe(pipefd);
        if (rs == -1) { perror("pipe"); exit(EXIT_FAILURE); }

        // fork into 2 processes
        rs = fork();
        if (rs == -1) { perror("fork"); exit(EXIT_FAILURE); }

        if (rs == 0) { // child process        1

                // close read end of pipe, keep write end open
                close(pipefd[0]);
                // close standard output
                close(1);
                // duplicate write end of pipe into standard output
                dup(pipefd[1]);
                // close write end of pipe
                close(pipefd[1]);
                // run first command
                rs = execlp("ls", "ls", (char*)NULL);
                if (rs == -1) { perror("execlp"); exit(EXIT_FAILURE); }

        }
```

```
else {  // parent process
    // close write end of pipe, keep read end open
    close(pipefd[1]);

    // fork into 2 processes
    rs = fork();
    if (rs == -1) { perror("fork"); exit(EXIT_FAILURE); }

    if (rs == 0) { // child process        2
        // close standard input
        close(0);
        // duplicate read end of pipe into standard input
        dup(pipefd[0]);
        // close read end of pipe
        close(pipefd[0]);

        // run second command
        rs = execlp("wc", "wc", (char*)NULL);
        if (rs == -1) { perror("execlp"); exit(EXIT_FAILURE); }

    } else { // parent process

        // close read end of pipe
        close(pipefd[0]);

        wait(NULL); // for first child
        wait(NULL); // for second child
        cout << "Parent done\n";

    }
}
```

C-Style user input

Concept 12: Getting user input in C can be done in several ways, depending on the type of input you want to capture. The standard library provides functions like scanf, getchar, and fgets for this purpose. Below are examples of how to use each to get user input.

8.1 Scanf

scanf is used to read formatted input from stdin. It's useful for reading specific types of data directly into variables, but it requires careful handling to avoid security issues and ensure good user experience.

8.1.1 Signature

```c
int scanf(const char *format, ...);
```

```c
int number;
printf("Enter a number: ");
scanf("%d", &number);
printf("You entered: %d\n", number);
return 0;
```

Note:-

scanf can lead to issues if the input doesn't match the expected format. It's often better to read a whole line of input and then parse it.

8.2 getchar

getchar reads a single character from stdin. It's useful for capturing single keystrokes.

8.2.1 Signature

```c
int getchar(void);
```

```c
char ch;
printf("Press a key: ");
ch = getchar();
printf("You pressed: %c\n", ch);
return 0;
```

8.3 fgets

fgets reads a string from stdin. It's safer than scanf for string input as it allows you to specify a maximum length, reducing the risk of buffer overflow.

8.3.1 Signature

```
char* fgets(char *str, int num, FILE *stream);
```

```
char str[100];
printf("Enter a string: ");
fgets(str, 100, stdin);
printf("You entered: %s", str);
return 0;
```

Note:-

fgets includes the newline character (\n) in the buffer if there's space, so you might want to remove it after capturing input.

8.4 Continously capture input

To continuously get user input in a loop until a special string, such as "exit", is entered, you can use fgets to read the input into a buffer and then compare the input with the special string. Here's an example using strcmp to compare strings. Remember, fgets captures the newline character when the user presses enter, so you might want to remove that from the input before comparing.

```c
char input[100]; // Buffer to hold the input

// Prompt the user
printf("Enter text (type 'exit' to quit): ");

// fgets reads until a newline or EOF. It keeps the newline, so
↪  we'll need to remove it.
while (fgets(input, sizeof(input), stdin)) {
    // Remove newline character, if present
    input[strcspn(input, "\n")] = 0;

    // Check if the input is "exit"
    if (strcmp(input, "exit") == 0) {
        printf("Exiting...\n");
        break;
    }

    // Echo the input back to the user
    printf("You entered: %s\n", input);
    printf("Enter text (type 'exit' to quit): ");
}
```