CSE 333 Section

I/O, POSIX, and System Calls!

Logistics

Due TODAY:

Homework 1 @ 11:59 pm

Due Monday:

Exercise 7 @ 10:30 am

POSIX

Posix is a family of standards specified by the IEEE. These standards maintains compatibility across variants of Unix-like operating systems by defining APIs and standards for basic I/O (file, terminal, and network) and for threading.

1. What does POSIX stand for?

   Portable Operating System Interface

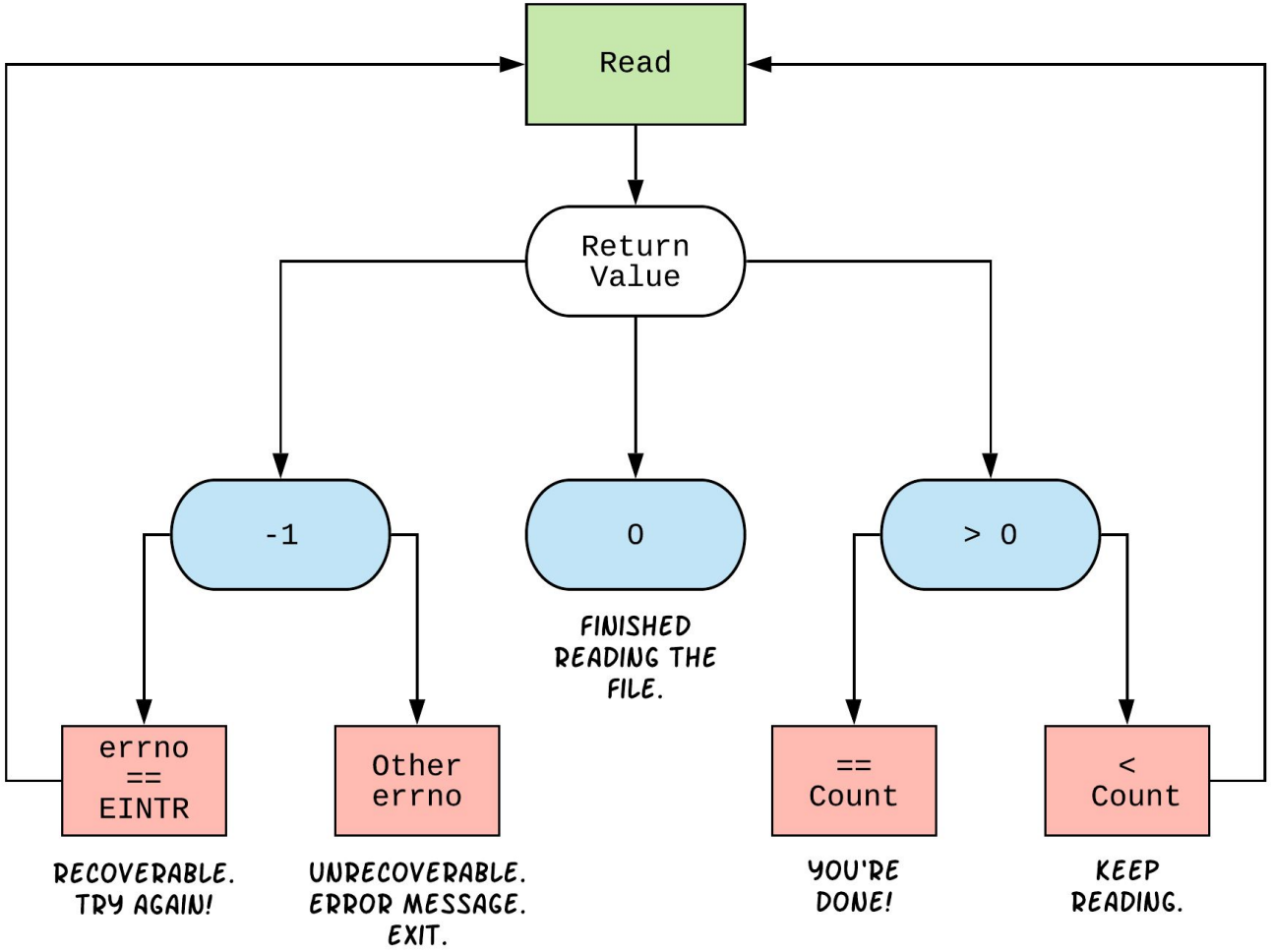
2.` Why might a POSIX standard be beneficial? From an application perspective? Versus using the C stdio library?

- **More explicit control since read and write functions are system calls and you can directly access system resources.**
- **POSIX calls are unbuffered so you can implement your own buffer strategy on top of read()/write().**
- **There is no standard higher level API for network and other I/O devices**

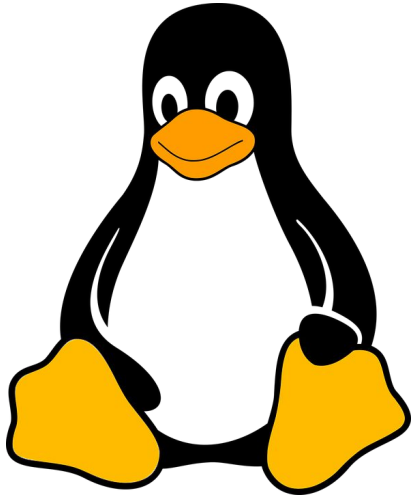
Review from Lecture

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

An error occurred	`result = -1` `errno = error`
Already at EOF	`result = 0`
Partial Read	`result < count`
Success!	`result == count`


Read
Return Value
-1
0
> 0
FINISHED READING THE FILE.
errno == EINTR
Other errno
== Count
< Count
RECOVERABLE. TRY AGAIN!
UNRECOVERABLE. ERROR MESSAGE. EXIT.
YOU'RE DONE!
KEEP READING.

New Scenario - Messy Roommate



- The Linux kernel now lives with you in room #333
- There are N pieces of trash in the room
- There is a single trash can, `char bin[N]`
  - (For some reason, the trash goes in a particular order)
- You can tell your roommate to pick it up, but he/she is unreliable

New Scenario - Messy Roommate

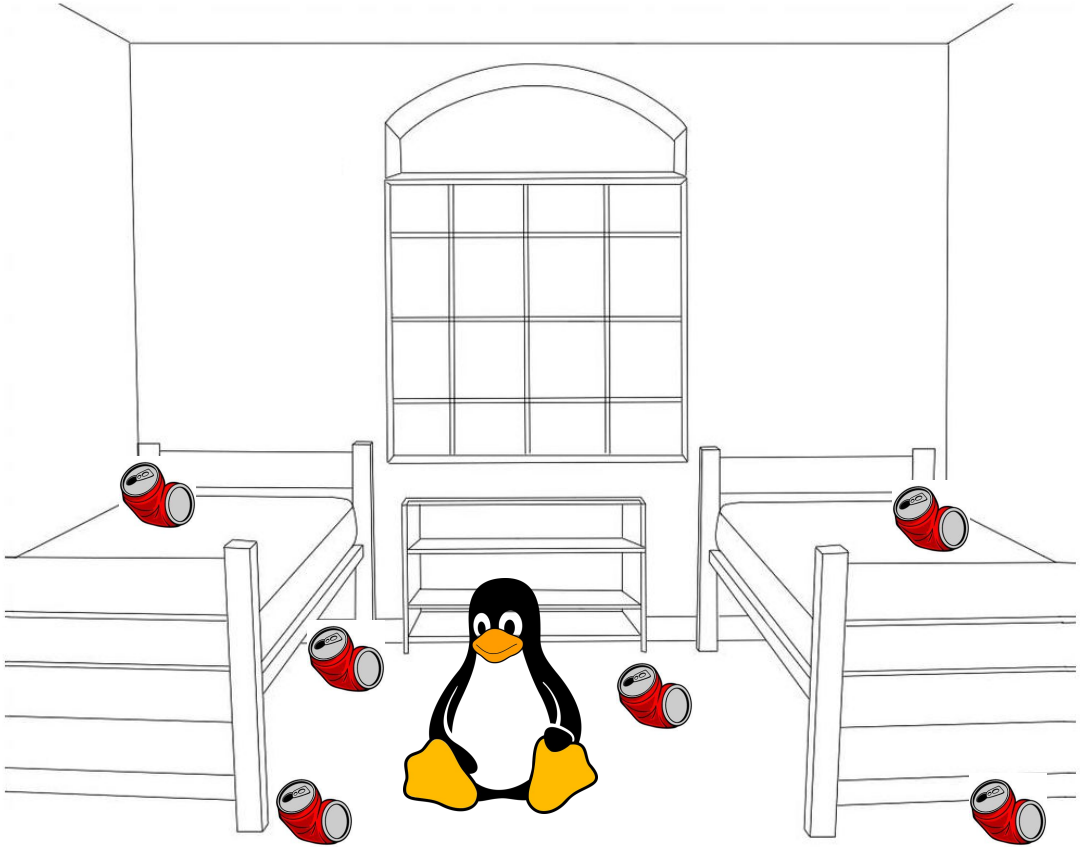
`NumTrash pickup(roomNum, trashBin, Amount)`

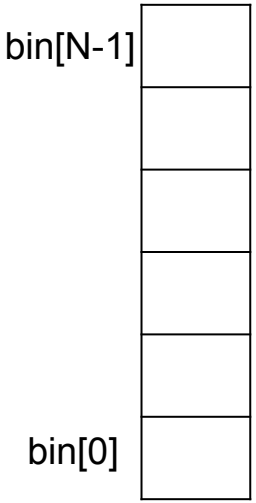
"*I tried to start cleaning, but something came up*" (got hungry, had a midterm, room was locked, etc.)	`NumTrash == -1` `errno == excuse`
"*You told me to pick up trash, but the room was already clean*"	`NumTrash == 0`
"*I picked up some of it, but then I got distracted by my favorite show on Netflix*"	`NumTrash < Amount`
"*I did it! I picked up all the trash!*"	`NumTrash == Amount`

How do we get room 333 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`



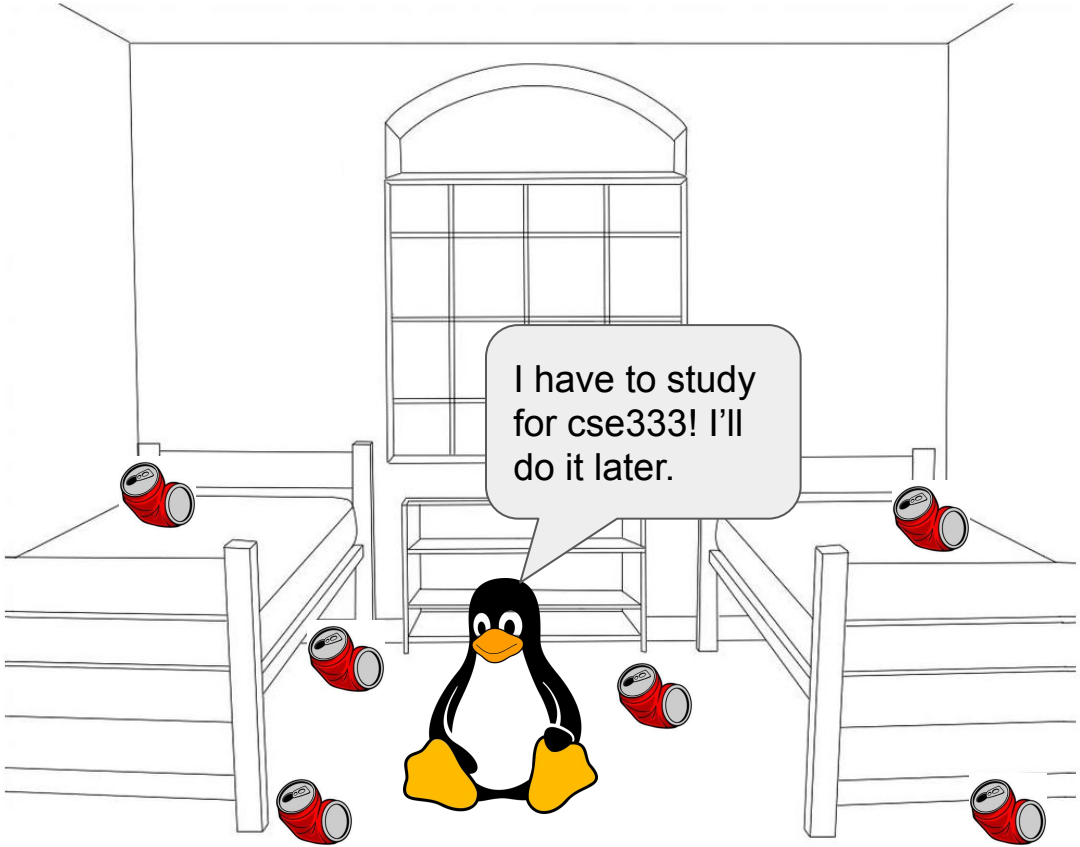


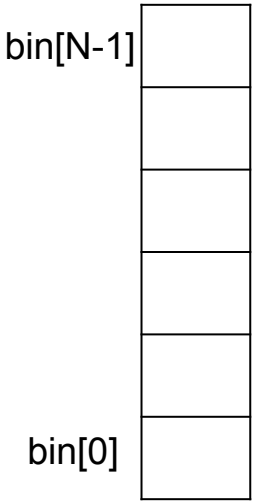
What do we do in the following scenarios?

How do we get room 333 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`



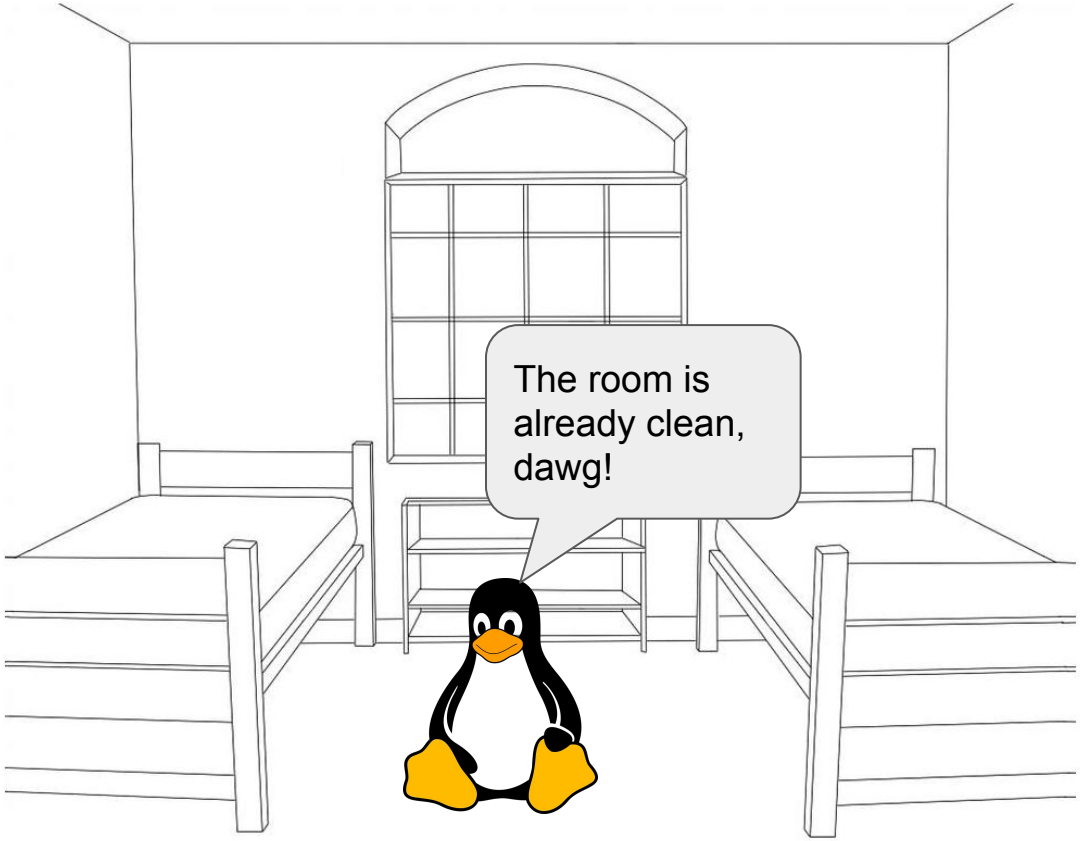


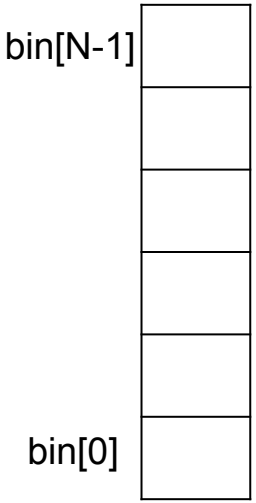
Decide if the excuse is reasonable, and either let it be or ask again.

How do we get room 333 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`



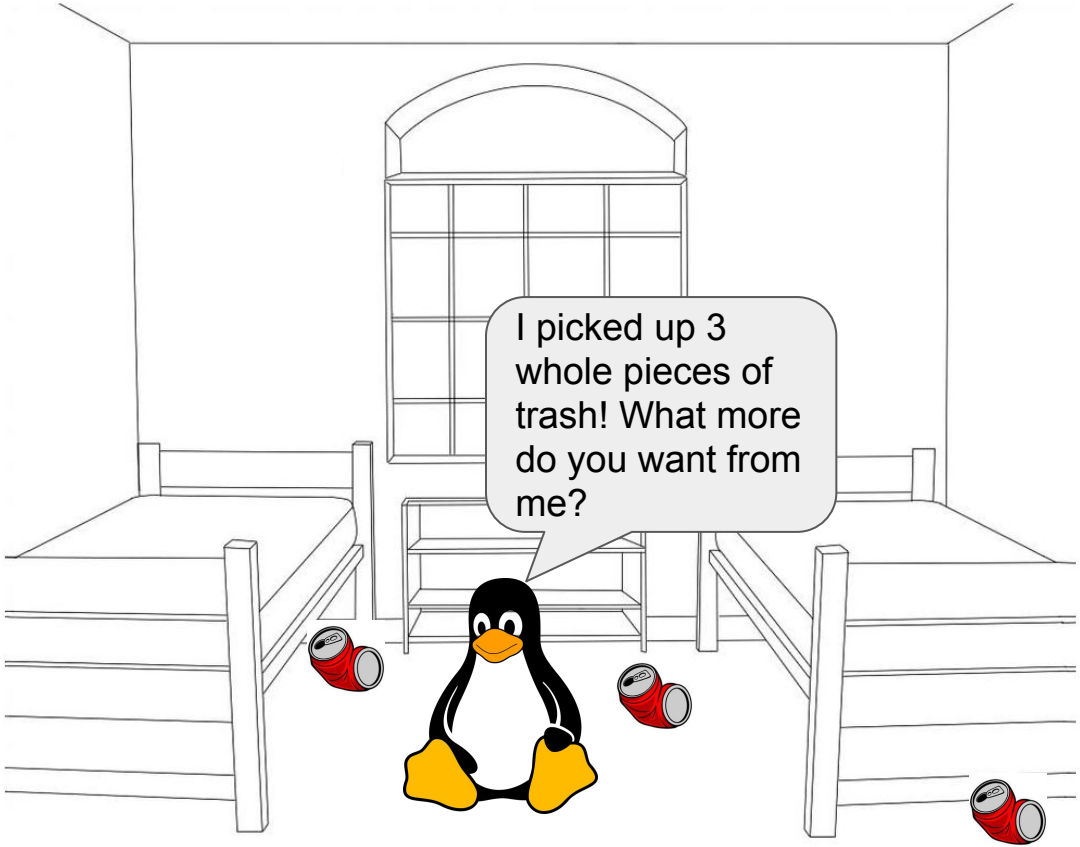


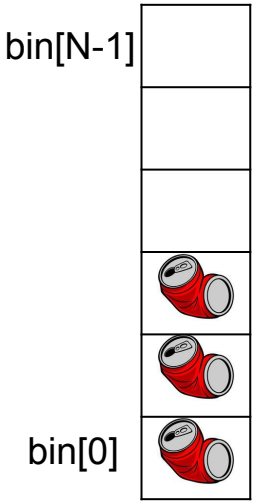
Stop asking them to clean the room! There's nothing to do.

How do we get room 333 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`



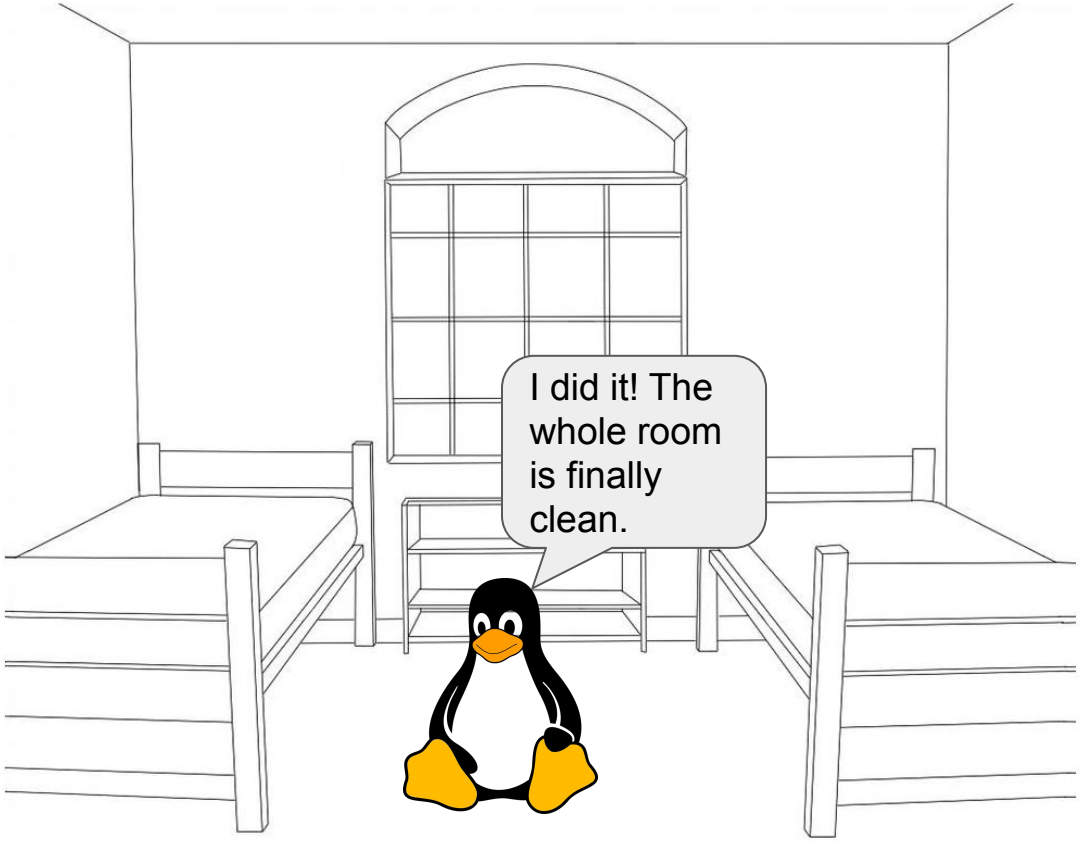


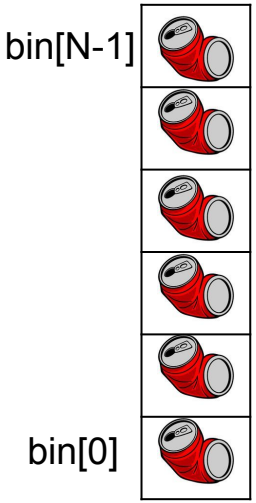
Ask them again to pick up the rest of it.

How do we get room 333 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`





They did what you asked, so stop asking them to pick up trash.

How do we get room 333 clean?

```
int pickedUp = 0;
while ( ____________ ) {



}
```

`NumTrash pickup(roomNum, trashBin, Amount)`

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`

How do we get room 333 clean?

`NumTrash pickup(roomNum, trashBin, Amount)`

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`

```
int pickedUp = 0;
while ( pickedUp < N ) {
    NumTrash = pickup( 333, bin + pickedUp, N - pickedUp )
    if ( NumTrash == -1 ) {
        if ( excuse not reasonable )
            ask again
        stop asking and handle the excuse
    }
    if ( NumTrash == 0 )  // we over-estimated the trash
         stop asking since the room is clean
    add NumTrash to pickedUp
}
```

How do we get room 333 clean?

`NumTrash pickup(roomNum, trashBin, Amount)`

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`

```
int pickedUp = 0;
while ( pickedUp < N ) {
    result = pickup( 333, bin + pickedUp, N - pickedUp )
    if ( result == -1 ) {
        if ( errno == E_BUSY_NETFLIX )
            continue;
        break;
    }
    if ( result == 0 )
         break;
    pickedUp += result;
}
```

Some Final Notes...

We assumed that there were exactly N pieces of trash (N bytes of data that we wanted to read from a file). How can we modify our solution if we don't know N?

(Answer): Keep trying to `read(...)` until we get 0 back (EOF / clean room)

We determine N dynamically by tracking the number of bytes read until this point, and use `malloc` to allocate more space as we read.

(This case comes up when reading/writing to the network!)

There is no one true loop (or true analogy).

Tailor your POSIX loops to the specifics of what you need!

Back to the worksheet (Q3)

```
int fd = ____________________________________________;  // open 333.txt
int n = ....;
char *buf = ....... ; //  Assume buf initialized with size n
int result;

_____________________________;   // initialize variable for loop

...   // code that populates buf happens here

while (_____________________) {

    result = write( _______,_______________,_____________________);

    if (result == -1) {
        if (errno != EINTR) {
            // a real error happened, return an error result
            ___________________;  //  cleanup
            perror("Write failed");
            return -1;
        }
        continue;   // EINTR happened, so loop around and try again
    }
   ___________________________________;   // update loop variable
}
_________________; //  cleanup
```

```
int fd = open("333.txt", O_WRONLY)                    ;  // open 333.txt
int n = ....;
char *buf = ....... ; //  Assume buf initialized with size n
int result;

char *ptr = buf                    ;    // initialize variable for loop

...    // code that populates buf happens here

while (      ptr < buf + n                 ) {

    result = write(  fd      ,    ptr              ,      buf + n - ptr          );

    if (result == -1) {
        if (errno != EINTR) {
            // a real error happened, return an error result
             close(fd)                ;  //  cleanup
            perror("Write failed");
            return -1;
        }
        continue;   // EINTR happened, so loop around and try again
    }
    ptr += result                                ;    // update loop variable
}
close(fd)
                ; //  cleanup
```

**This is one way to solve this exercise. There exist other correct solutions

More Posix!

4) Why is it important to store the return value from the `write()` function? Why do we not check for a return value of 0 like we do for `read()`?

5) Why is it important to remember to call the `close()` function once you have finished working on a file?

More Posix!

4) Why is it important to store the return value from the `write()` function? Why do we not check for a return value of 0 like we do for `read()`?

write() may not actually write all the bytes specified in count.
Writing adds length to your file, so you don't need to check for end of file.

5) Why is it important to remember to call the `close()` function once you have finished working on a file?

In order to free resources i.e. other processes can acquire locks on those files.

DIRECTORIES

DIR* in POSIX?

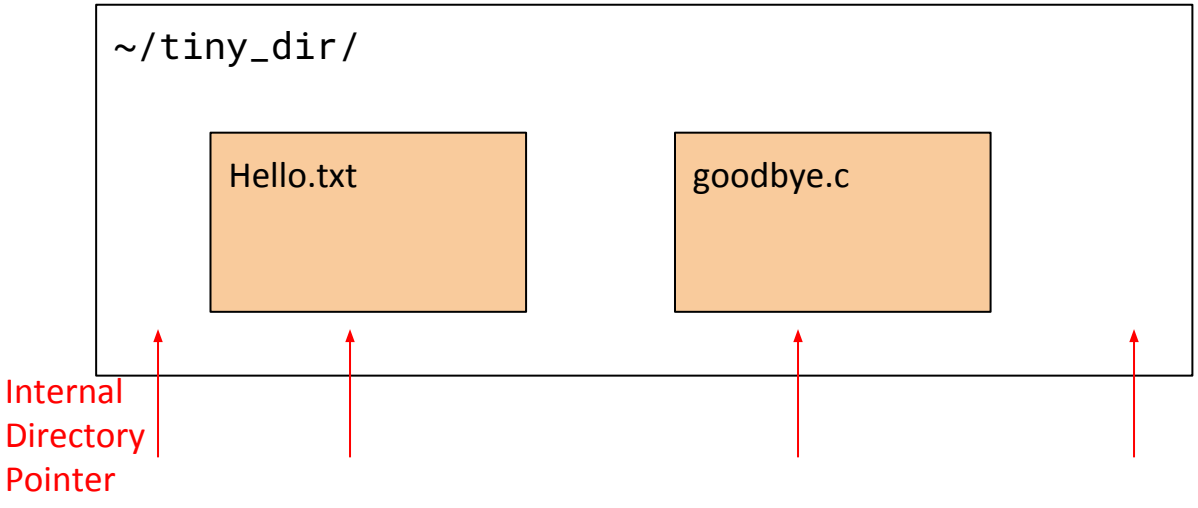
```
DIR *opendir(const char* name);

int closedir(DIR *dirp);

struct dirent *readdir(DIR *dirp);
```

Gives you the 'next' directory entry, returns null when end of directory reached.

Looks Like C-STDIO
But, it's actually POSIX!

DIR* is not quite a file descriptor, but we will use it very similarly

readdir() example




```
DIR* dirp = opendir("~/tiny_dir");  // opens directory
struct dirent *file = readdir(dirp); // gets pointer to "Hello.txt" dirent
file = readdir(dirp);   // gets pointer to "goodbye.c" dirent
file = readdir(dirp);   // gets NULL
closedir(dirp);  // cleanup
```

Struct dirent

Stands for: Directory Entry

Result from:

```
struct dirent *readdir(DIR *dirp);
```

data stored in integer types about the directory entry

```
struct dirent {
    ino_t          d_ino;    /* inode number for the dir entry */
    off_t          d_off;    /* not necessarily an offset */
    unsigned short d_reclen; /* length of this record */
    unsigned char  d_type;   /* type of file (not what you think);
                                not supported by all file system
                                types */

    // Probably the thing
    // we care about most
    char           d_name[NAME_MAX+1];   /* directory entry name*/
};
```

** You do not need to "free" or "close" dirent structs from readdir() **

Exercise:

7) Given the name of a directory, write a C program that is analogous to **`ls`**, *i.e.* prints the names of the entries of the directory to `stdout`. Be sure to handle any errors!
Example usage: "`./dirdump <path>`" where `<path>` can be absolute or relative.

```
int main(int argc, char** argv) {
  /* 1. Check to make sure we have a valid command line arguments
*/if (argc != 2) {
    fprintf(stderr, "Usage: ./dirdump <path>\n");
    return EXIT_FAILURE;
  }

  /* 2. Open the directory, look at opendir() */
DIR* dirp = opendir(argv[1]);
  if (dirp == NULL) {
    fprintf(stderr, "Could not open directory\n");
    return EXIT_FAILURE;
  }

  /* 3. Read through/parse the directory and print out file names
     Look at readdir() and struct dirent */
  struct dirent *entry;
  entry = readdir(dirp);
  while (entry != NULL) {
    printf("%s\n", entry->d_name);
    entry = readdir(dirp);
  }


  /* 4. Clean up */
  closedir(dirp);
  return EXIT_SUCCESS;


}
```