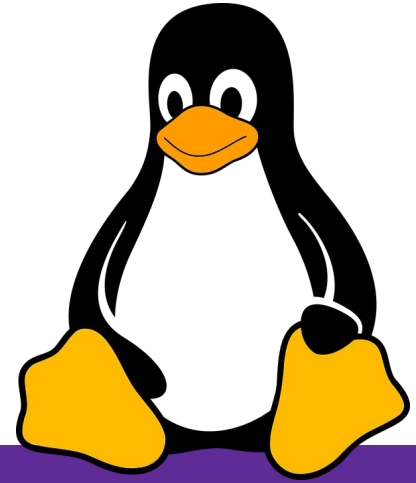# CSE 333 Section 4

POSIX I/O, C++ Intro

Logistics

- Homework 2
  - Due next **Wednesday, 4/24 @ 10:00pm**
  - Indexing files to allow for searching
- Exercise 8.5
  - Due **Tomorrow @ 11:00am**
- Exercise 9
  - Write a Vector class in C++
  - Due **Monday @ 11:00am**

POSIX

POSIX (<u>P</u>ortable <u>O</u>perating <u>S</u>ystem <u>I</u>nterface)

A family of IEEE standards that maintains compatibility across variants of Unix-like operating systems for basic I/O (*file*, terminal, and *network*) and for *threading*.

1. Why might a POSIX standard be beneficial (*e.g.*, from an application perspective or vs. 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

What's Tricky about (POSIX) File I/O?

- Communication with input and output devices doesn't always work as expected
    - Some details might be unknown (*e.g.*, size of a file)
    - May not process all data or fail, necessitating read/write *loops*

- Different system calls have a variety of different failure modes and error codes
    - Look up in the documentation and use pre-defined constants!
    - Lots of error-checking code needed
        - Need to handle resource cleanup on *every* termination pathway

Messy Roommate

I/O Analogy – Messy Roommate

- The Linux kernel (Tux) 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 they are unreliable
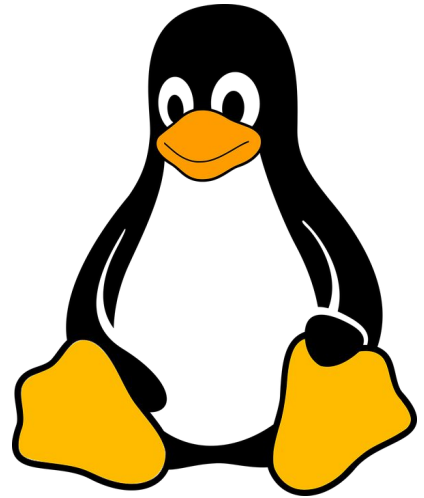
I/O Analogy – Messy Roommate

```
num_trash = Pickup(room_num, trash_bin, amount)
```

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

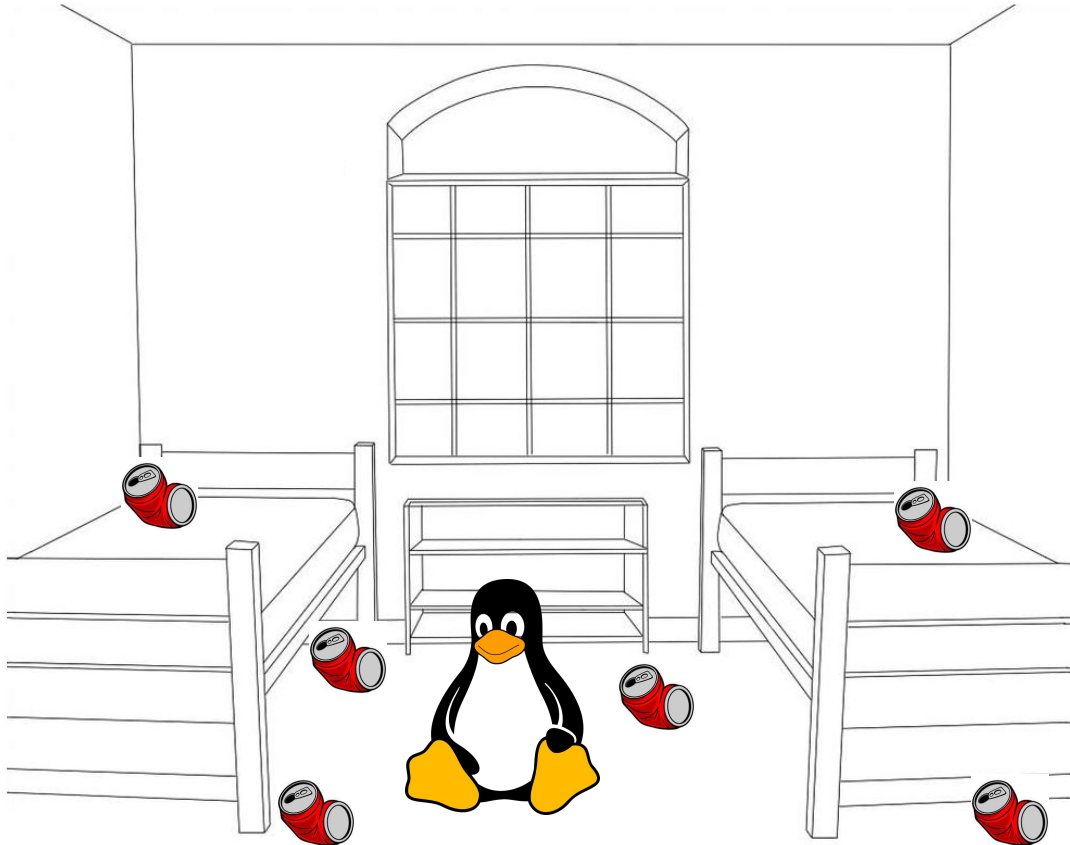num_trash = Pickup(room_num, trash_bin, amount)

num_trash == -1, errno == excuse
num_trash == 0
num_trash < Amount
num_trash == Amount

How do we get room 333 clean?

What do we do in the following scenarios?

bin[N-1]

bin[0]

num_trash pickup(roomNum, trashBin, Amount)

num_trash == -1, errno == excuse
num_trash == 0
num_trash < Amount
num_trash == Amount

How do we get room 333 clean?

I have to study for cse333! I'll do it later.

bin[N-1]

bin[0]

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

num_trash pickup(roomNum, trashBin, Amount)

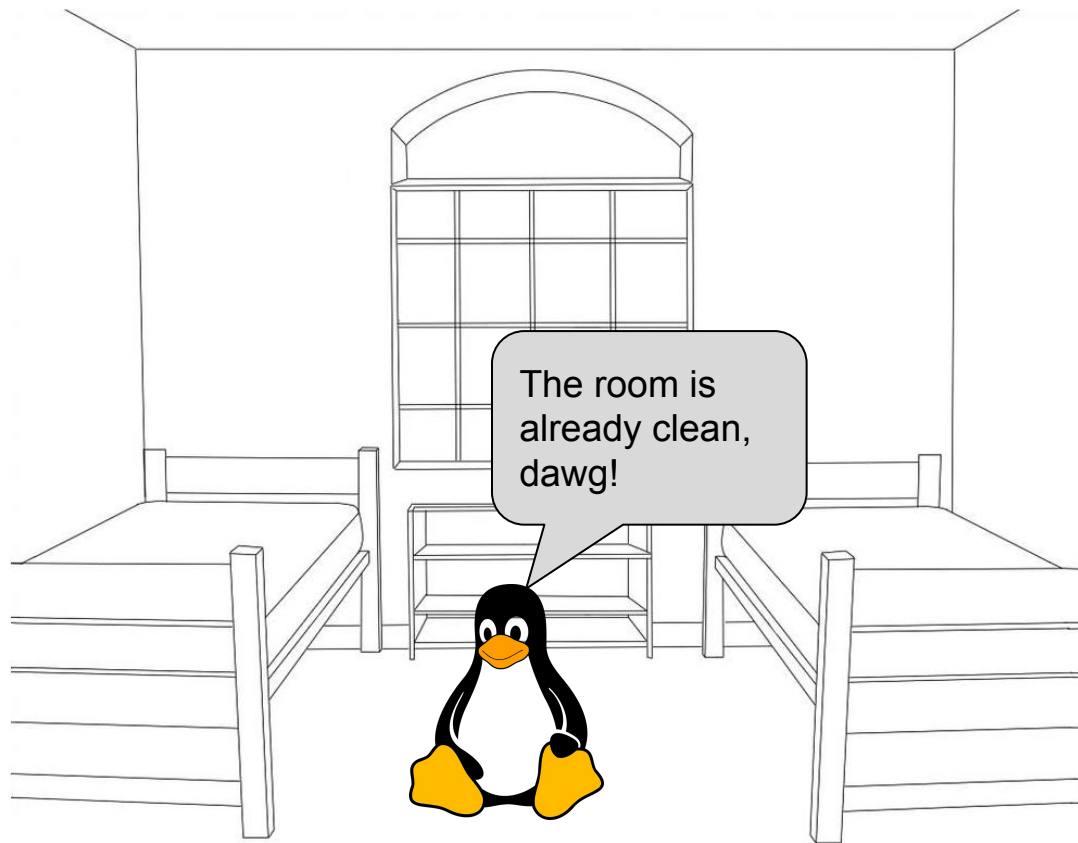| |
|---|
| num_trash == -1, errno == excuse |
| num_trash == 0 |
| num_trash < Amount |
| num_trash == Amount |

How do we get room 333 clean?

The room is already clean, dawg!

bin[N-1]

bin[0]

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

How do we get room 333 clean?

num_trash pickup(roomNum, trashBin, Amount)

num_trash == -1, errno == excuse
num_trash == 0
num_trash < Amount
num_trash == Amount

I picked up 3 whole pieces of trash! What more do you want from me?
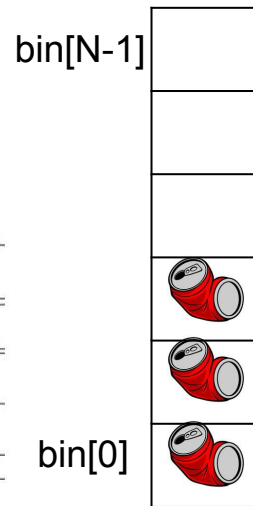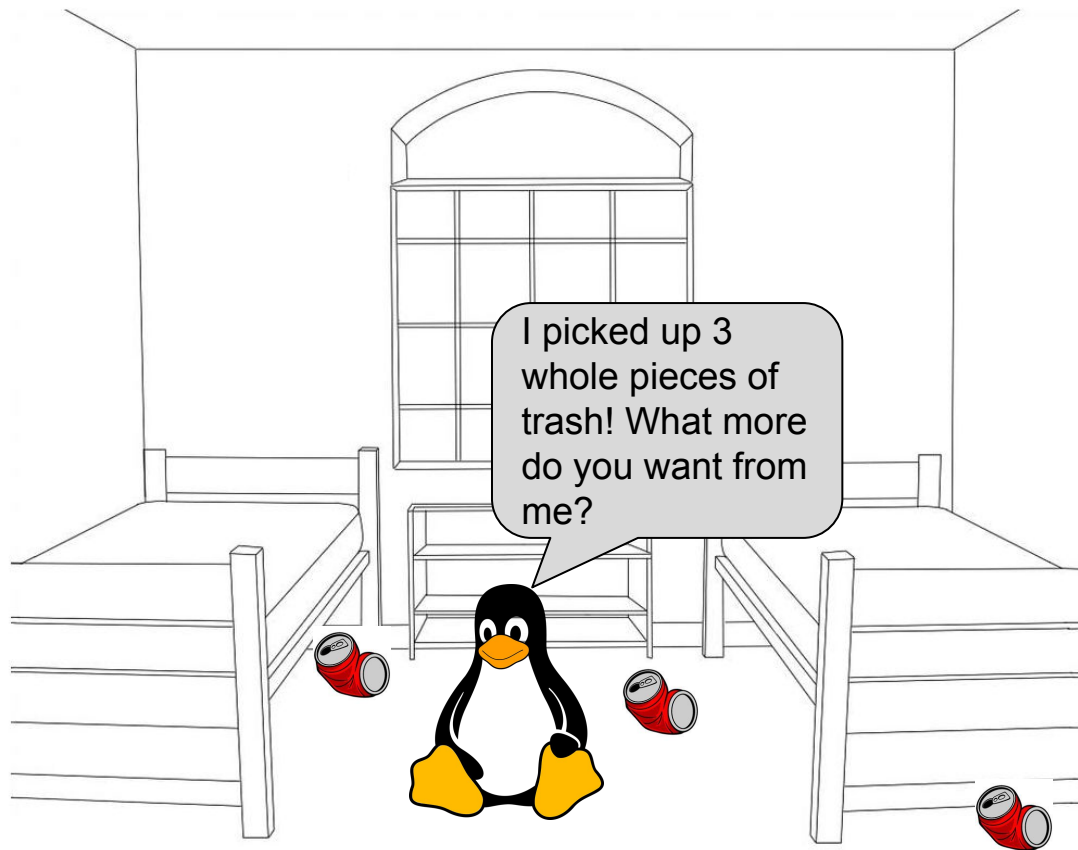
bin[N-1]

bin[0]

Ask them again to pick up the rest of it.

12

num_trash pickup(roomNum, trashBin, Amount)

num_trash == -1, errno == excuse
num_trash == 0
num_trash < Amount
num_trash == Amount

How do we get room 333 clean?

I did it! The whole room is finally clean.
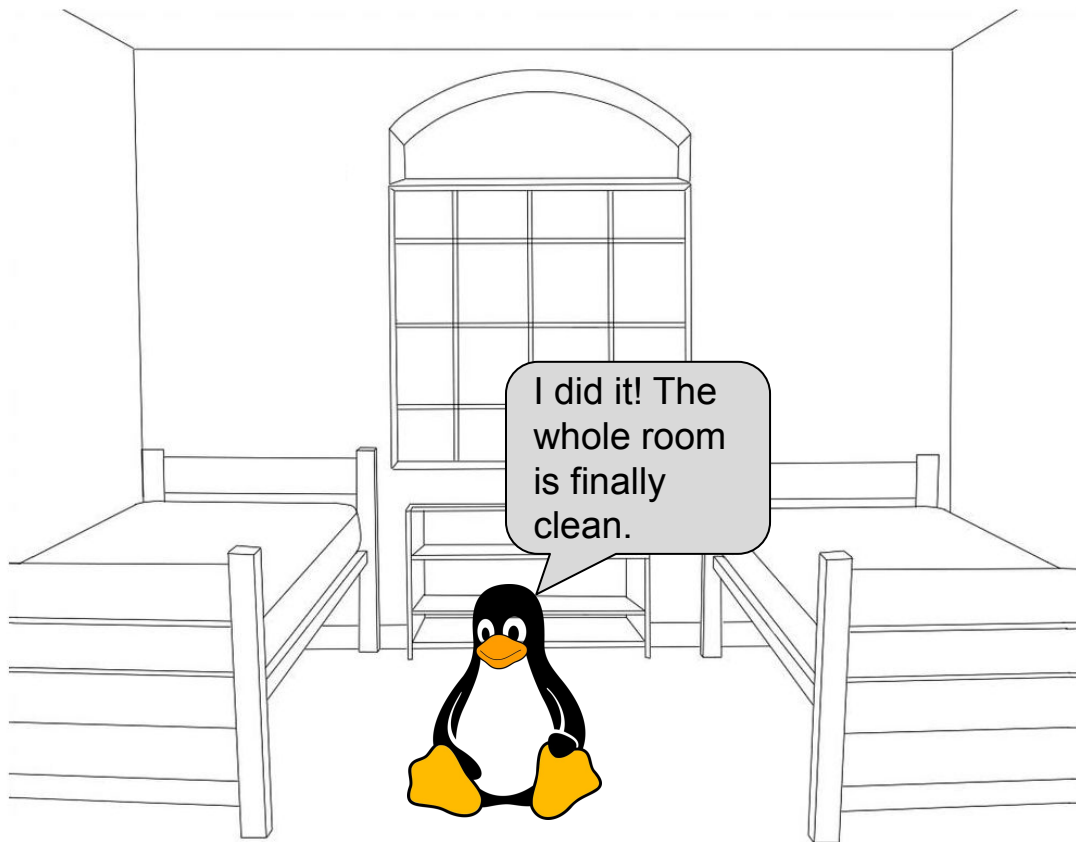
bin[N-1]

bin[0]

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

Review from Lecture – POSIX Read

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

An error occurred	`result == -1` `errno = error`
Nothing left to read (already at EOF)	`result == 0`
Partial Read	`result < count`
Success!	`result == count`

https://man7.org/linux/man-pages/man2/read.2.html

Not fully
comprehensive, please
refer to the man pages

```
                              ┌──────────┐
           ┌─────────────────►│   Read   │◄─────────────────┐
           │                  └──────────┘                  │
           │                        │                       │
           │                        ▼                       │
           │                   ╭──────────╮                 │
           │          ┌────────┤  Return  ├────────┐        │
           │          │        │  Value   │        │        │
           │          │        ╰──────────╯        │        │
           │          ▼             │              ▼        │
           │     ╭─────────╮        ▼         ╭─────────╮   │
           │     │   -1    │   ╭─────────╮    │   > 0   │   │
           │   ┌─┤         ├─┐ │    0    │  ┌─┤         ├─┐ │
           │   │ ╰─────────╯ │ ╰─────────╯  │ ╰─────────╯ │ │
           │   │             │              │             │ │
           │   ▼             ▼   FINISHED   ▼             ▼ │
           │ ┌──────┐   ┌────────┐ READING  ┌──────┐  ┌──────┐
           └─┤ errno│   │ Other  │ THE FILE │  ==  │  │  <   ├┘
             │  ==  │   │ errno  │          │Count │  │Count │
             │EINTR │   └────────┘          └──────┘  └──────┘
             └──────┘
```
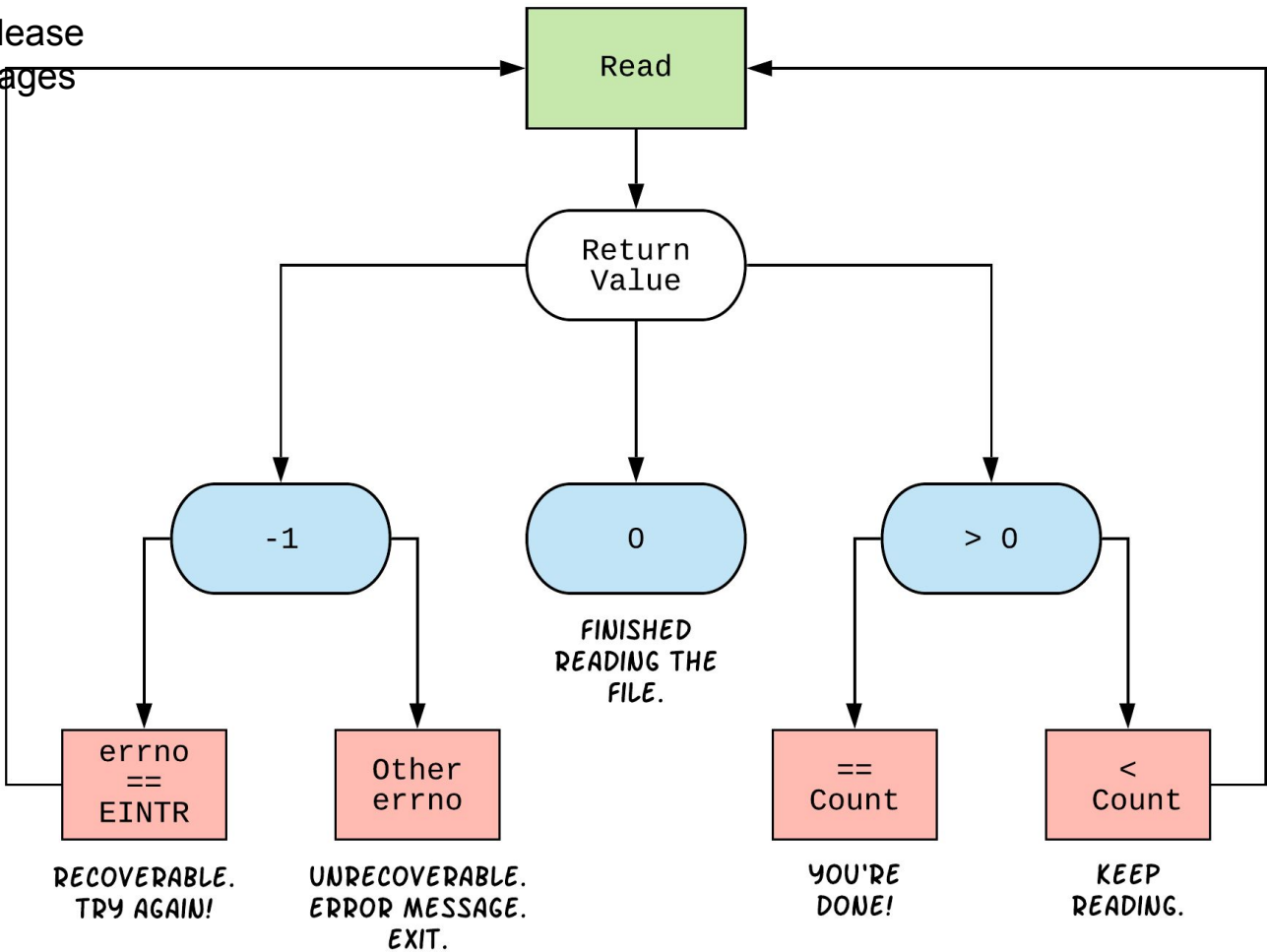
errno == EINTR — RECOVERABLE. TRY AGAIN!

Other errno — UNRECOVERABLE. ERROR MESSAGE. EXIT.

0 — FINISHED READING THE FILE.

== Count — YOU'RE DONE!

< Count — KEEP READING.

Exercises 1-4

```
int open(char *name, int flags);
```
➔ `name` *is a string representing the name of the file. Can be relative or absolute.*
➔ `flags` *is an integer code describing the access. Some common flags are listed below:*
- ◆ `O_RDONLY` – *Open the file in read-only mode.*
- ◆ `O_WRONLY` – *Open the file in write-only mode.*
- ◆ `O_RDWR` – *Open the file in read-write mode.*
- ◆ `O_APPEND` – *Append new information to the end of the file.*

★ Returns an integer which is the file descriptor. Returns `-1` if there is a failure.

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```
➔ `fd` *is the file descriptor (as returned by open()).*
➔ `buf` *is the address of a memory area into which the data is read or written.*
➔ `count` *is the* maximum *amount of data to read from or write to the stream.*

★ Returns the *actual* amount of data read from or written to the file.

```
int close(int fd);
```

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

(ℹ) This is just ONE possible way to solve this exercise!

POSIX Analysis 🐧

3. Why is it important to store the return value from `write`? Why don't we check for a return value of `0` like `read`?

    `write` may not actually write all the bytes specified in count.

    The `0` case for reading was EOF, but writing adds length to your file and we know exactly how much we are trying to write.

4. Why is it important to remember to call `close` once you have finished working on a file?

    In order to free resources (*i.e.*, locks on those files, file descriptor table entries).

There is No One True Loop!!!

You will need to tailor your POSIX loops to the specifics of what you need.

Some design considerations:
- Read data in fixed-sized chunks or all at once?
  - Trade-off in disk accesses versus memory usage.
- What if we don't know N (how many bytes to read) ahead of time?
  - Keep calling **read** until we get 0 back (EOF).
  - Can determine N dynamically by tracking the number of bytes read and using **malloc**/**realloc** to allocate more space as we go.
  - This case comes up when reading/writing to the network (later in 333)!

Directories

Directories

- A directory is a special file that stores the names and locations of the related files/directories
  - This includes itself (`.`), its parent directory (`..`), and all of its children (*i.e.*, the directory's contents)
  - Take CSE 451 to learn more about the directory structure

- Accessible via POSIX (`dirent.h` in C/C++)

- Why might we want to work with directories in a program?
  List files, find files, search files, recursively traverse directories, etc.

POSIX Directory Basics

- POSIX defines operations for directory *traversal*
  - `DIR *` is not a file descriptor, but used similarly
  - `struct dirent` describes a <u>dir</u>ectory <u>ent</u>ry
  - **`readdir`**`()` returns the 'next' directory entry, or `NULL` at end

- Error values (they also set `errno`):
  - `DIR *`**`opendir`**`(const char *name);`   `// NULL`
  - `struct dirent *`**`readdir`**`(DIR *dirp);`   `// NULL`
  - `int `**`closedir`**`(DIR *dirp);`   `// -1`

struct dirent

- Returned value from **readdir**
  - Does *not* need to be "freed" or "closed" 🎉
- Fields are "unspecified" (depends on your file system)
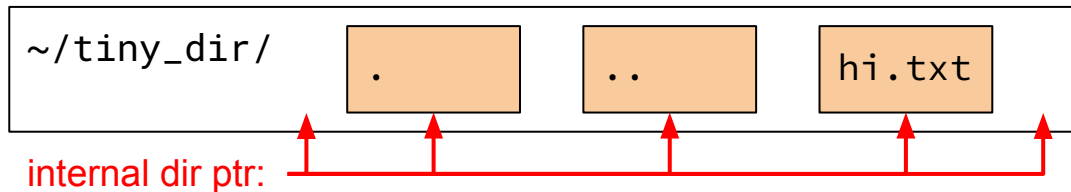 - glibc specifies:

```
struct dirent {
    ino_t            d_ino;
    off_t            d_off;
    unsigned short   d_reclen;
    unsigned char    d_type;
    char             d_name[256];
};
```

directory entry metadata stored in integer types

Null-terminated directory entry name (what we care about in 333)

readdir Example

```
~/tiny_dir/        [ . ]        [ .. ]        [ hi.txt ]
```
internal dir ptr:

```
DIR *dirp = opendir("~/tiny_dir");   // opens directory
struct dirent *file = readdir(dirp);  // gets ptr to "."
file = readdir(dirp);    // gets ptr to ".."
file = readdir(dirp);    // gets ptr to "hi.txt"
file = readdir(dirp);    // gets NULL
closedir(dirp);          // clean up
```

24

Exercise 5

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!

```c
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;
   }

🐧..
```

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!

```c
/* 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;
}
```

Pointers, References, & Const

Example

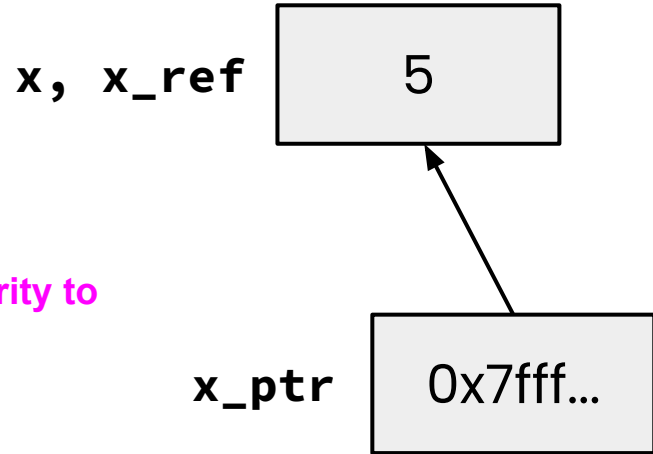Consider the following code:

```
int x = 5;
int& x_ref = x;
int* x_ptr = &x;
```

Note syntactic similarity to pointer declaration

Still the address-of operator!

x, x_ref [5]

x_ptr [0x7fff…]

What are some tradeoffs to using pointers vs references?

Pointers vs. References

Pointers

- Can move to different data via reassignment/pointer arithmetic

- Can be initialized to **NULL**

- Useful for output parameters: `MyClass* output`

References

- References the same data for its entire lifetime - _can't reassign_

- No sensible "default reference," must be an alias

- Useful for input parameters: `const MyClass &input`

Pointers, References, Parameters

- `void func(int& arg)` vs. `void func(int* arg)`

- Use references when you don't want to deal with pointer semantics

  - Allows real pass-by-reference

  - Can make intentions clearer in some cases

- **STYLE TIP:** use references for input parameters and pointers for output parameters, with the output parameters declared last

  - Note: A reference can't be `NULL`

Const

0x7fff...	42	0x7fff...
ro_x_ptr	**x**	**rw_x_ptr**
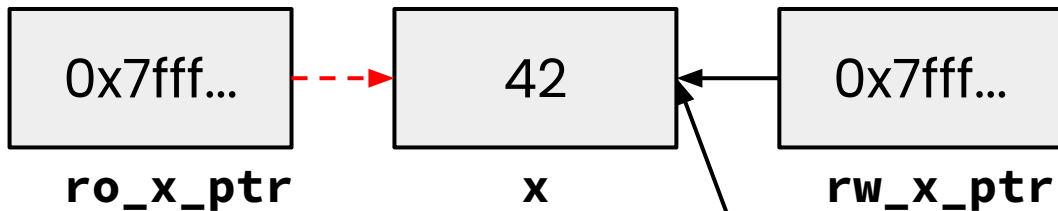
- Mark a variable with `const` to make a compile time check that a variable is never reassigned

- <u>Does not change the underlying write-permissions</u> for this variable

```
int x = 42;

// Read only
const int* ro_x_ptr = &x;

// Can still modify x with
rw_x_ptr!
int* rw_x_ptr = &x;

// Only ever points to x
int* const x_ptr = &x;
```

0x7fff...
x_ptr

Legend
Red = can't change box it's next to
Black = read and write

Exercise 6

Exercise 6

x, x_ref
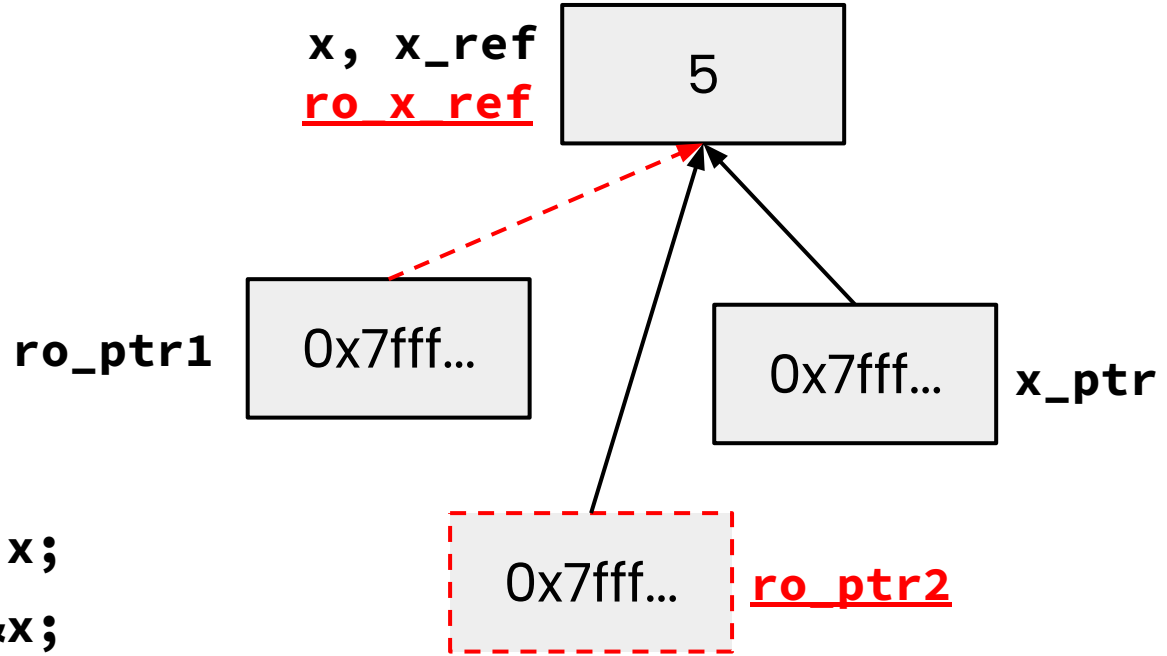**ro_x_ref**

5

```
int x = 5;
int& x_ref = x;
int* x_ptr = &x;
const int& ro_x_ref = x;
const int* ro_ptr1 = &x;
int* const ro_ptr2 = &x;
```

ro_ptr1   0x7fff…

0x7fff… **x_ptr**

0x7fff… **ro_ptr2**

"Pointer to a const int"

"Const pointer to an int"

Tip: Read the declaration "right-to-left"

Legend
Red = can't change box it's next to
Black = read and write

Exercise 6

When would you prefer void Func(int &arg); to void Func(int *arg);? Expand on this distinction for other types besides int.

- When you don't want to deal with pointer semantics, use references
- When you don't want to copy stuff over (doesn't create a copy, especially for parameters and/or return values), use references
- Style wise, we want to use **references for input parameters** and **pointers for output parameters**, with the output parameters declared last

Exercise 6

```
void foo(const int& arg);
void bar(int& arg);
```

```
int x = 5;
int& x_ref = x;
int* x_ptr = &x;
const int& ro_x_ref = x;
const int* ro_ptr1 = &x;
int* const ro_ptr2 = &x;
```


x, x_ref
ro_x_ref
5

ro_ptr1   0x7fff…

ro_ptr2   0x7fff…

0x7fff…
x_ptr
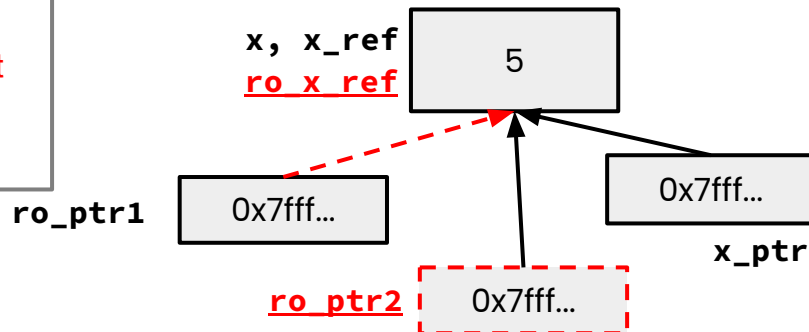
Which lines result in a compiler error?

✔️ OK   ❌ ERROR

✔️ `bar(x_ref);`
❌ `bar(ro_x_ref);` **ro_x_ref is const**
✔️ `foo(x_ref);`
✔️ `ro_ptr1 = (int*) 0xDEADBEEF;`
❌ `x_ptr = &ro_x_ref;` **ro_x_ref is const**
❌ `ro_ptr2 = ro_ptr2 + 2;` **ro_ptr2 is const**
❌ `*ro_ptr1 = *ro_ptr1 + 1;` **(*ro_ptr1) is const**

Objects and `const` Methods

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const int y);
  int  get_x() const { return x_; }
  int  get_y() const { return y_; }
  double  Distance(const Point& p) const;
  void  SetLocation(const int& x, const int& y);

 private:
  int  x_;
  int  y_;
};  // class Point

#endif  // POINT_H_
```

Cannot mutate the object it's called on.

Trying to change x_ or y_ inside will produce a compiler error!

A **const** class object can only call member functions that have been declared as **const**

Exercise 7

Exercise 7

Which *lines* of the snippets of code below would cause compiler errors?

✔️ OK   ❌ ERROR

```
class MultChoice {
  public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { }  // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const;  // do these MultChoice's match?

  private:
    int  q_;      // question number
    char resp_;   // response: 'A','B','C','D', or 'E'
}; // class MultChoice
```

✔️   `const MultChoice m1(1,'A');`
✔️   `MultChoice m2(2,'B');`
❌   `cout << m1.get_resp();`
✔️   `cout << m2.get_q();`

✔️   `const MultChoice m1(1,'A');`
✔️   `MultChoice m2(2,'B');`
✔️   `m1.Compare(m2);`
❌   `m2.Compare(m1);`

What would you change about the class declaration to make it better?

```cpp
class MultChoice {
  public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { }  // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const;  // do these MultChoice's match?

  private:
    int  q_;      // question number
    char resp_;  // response: 'A','B','C','D', or 'E'
};  // class MultChoice
```

```
class MultChoice {
  public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { }  // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() const { return resp_; }
    bool Compare(const MultChoice &mc) const;  // do these match?

  private:
    int  q_;      // question number
    char resp_;  // response: 'A','B','C','D', or 'E'
};  // class MultChoice
```

- Make get_resp() const
- Make the parameter to Compare() const
- Stylistically:
  - Add a setter method and default constructor
  - Disable copy constructor and assignment operator