C++ Refresher, Move & File Descriptors

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk

Angie Cao

Aniket Ghorpade

Ashwin Alaparthi

Austin Lin

Hassan Rizwan

Lobi Zhao

Pearl Liu

Perrie Quek

Poll Everywhere

pollev.com/tqm

- How was spring break? Any questions now that we are back?

Administrivia

- “Check-in” posted
  - Due Wednesday
- HW06 – Hash Table
  - Posted☺
  - Due Friday 3/21 at midnight, leaving open till Sunday night tho
  - AG posted soon, but all tests are posted and public
- Mid-semester Survey Posted!
  - Due Sunday 3/23 & Anonymous
  - Please give feedback, it is useful for me to make the course better!
    And a lot has changed this semester!

Lecture Outline

- **C++ Programming Refresher**
- Move Semantics
- File Descriptors & Buffering

C++ Programming Refresher

- Implement the function rect() which takes in a vector of vector of integers. The function modifies the vector of vectors so that all rows are extended to be the same length (by adding 0’s to the rows).

```
void rect(vector<vector<int>>& m);
```

- For example, the following input

```
vector<vector<int>> m {
  {3, 4, 5},
  {2, 1},
  {},
  {0, 1, 2, 0, 0},
};

rect(m);
```

```
// what it should look
// like after calling rect
vector<vector<int>> m {
  {3, 4, 5, 0, 0},
  {2, 1, 0, 0, 0},
  {0, 0, 0, 0, 0},
  {0, 1, 2, 0, 0},
};
```

Lecture Outline

- C++ Programming Refresher
- **Move Semantics**
- File Descriptors & Buffering

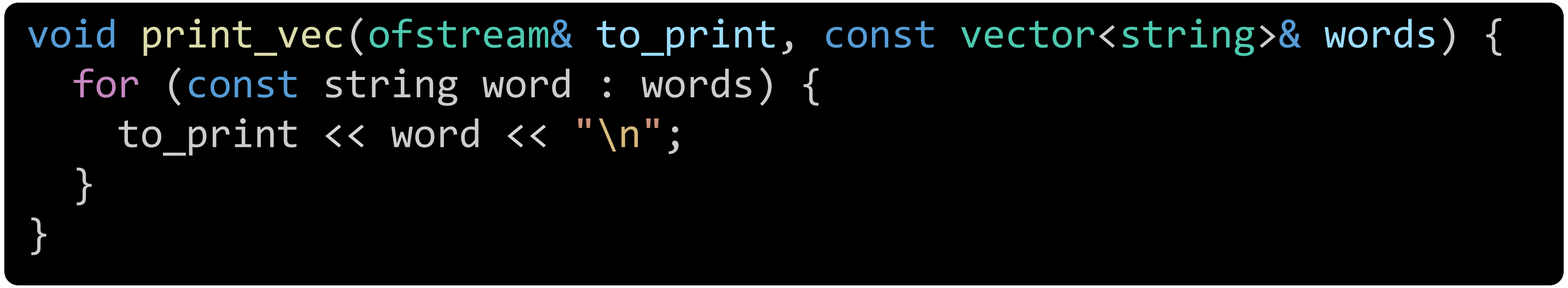
Memory Allocation in C++

- We rarely call new or delete directly in C++ code, but it is called implicity all the time if we are not careful
  - Whenever a data structure needs more space
  - Whenever we copy construct an object that needs allocation
  - Etc.

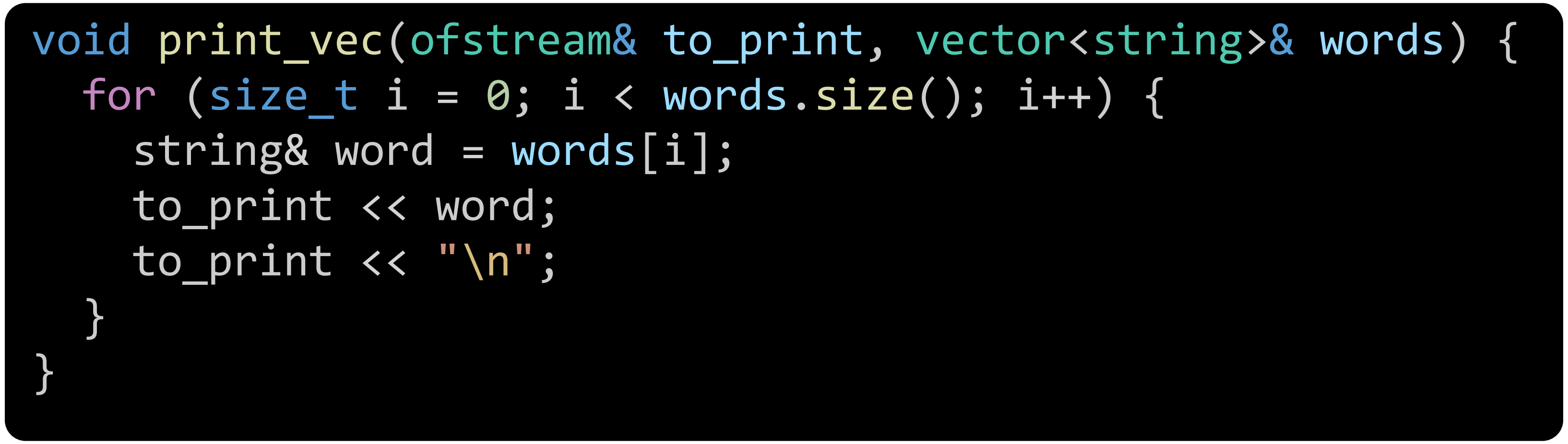
Poll Everywhere

pollev.com/tqm

- Which function is faster?



```
void print_vec(ofstream& to_print, const vector<string>& words) {
  for (const string word : words) {
    to_print << word << "\n";
  }
}
```



```
void print_vec(ofstream& to_print, vector<string>& words) {
  for (size_t i = 0; i < words.size(); i++) {
    string& word = words[i];
    to_print << word;
    to_print << "\n";
  }
}
```

Poll Everywhere

pollev.com/tqm

- How many memory allocations occur in each piece of code?
  - Assume vector resizes will double capacity
  - std::list is a linked list in C++

```
int main() {
  vector nums {4, 8};  // size and capacity == 2
  nums.push_back(5);
  nums.push_back(9);
  nums.push_back(5);
  nums.push_back(0);
}
```

```
int main() {
  list nums {4, 8};
  nums.push_back(5);
  nums.push_back(9);
  nums.push_back(5);
  nums.push_back(0);
}
```

Minimizing Allocations

- As we saw previously, memory allocations require time, sometimes a lot of time to compute.

- If performance is our goal, we should minimize the number of allocations we make.

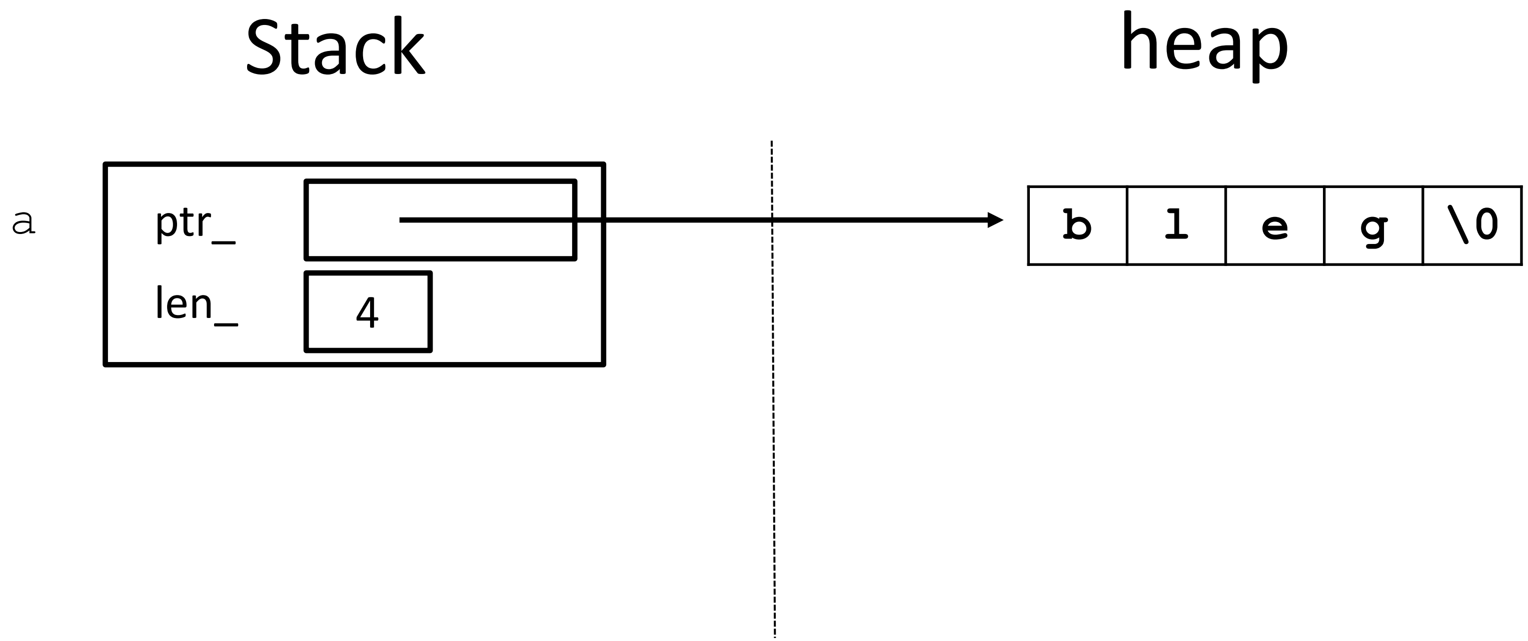
- This can include
  - Making references instead of copies
  - Using functions like `vector::reserve(size_t new_capacity)`
    - Java arraylist lets you specify capacity in the constructor.
    - std::string also has a reserve function
  - Using move semantics

Copy Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a{"bleg"};


}
```



Copy Semantics: close up look

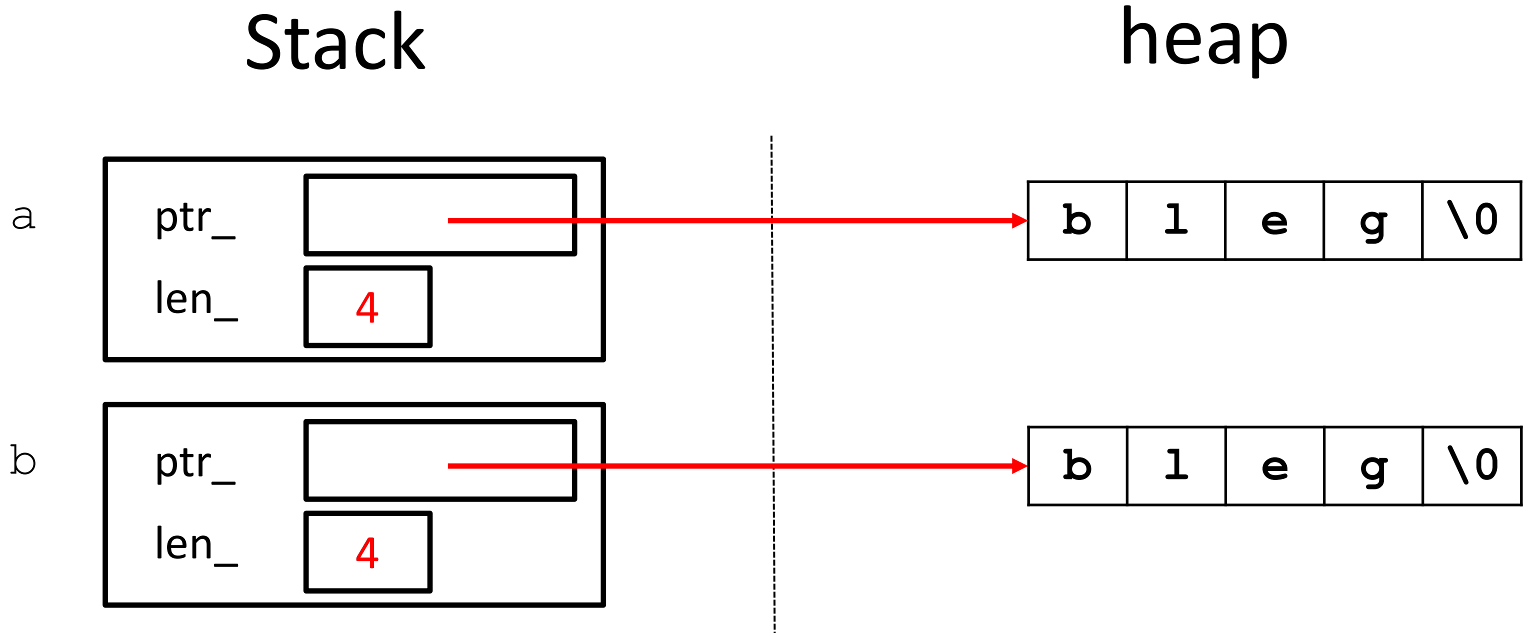
- When we copy construct string **b**

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  std::string b{a};
}
```

we could get something like:

This is another memory allocation, and we need to copy over the characters of the string



Move Semantics (C++11)

- "Move semantics" move values from one object to another without copying ("stealing")
  - A complex topic that uses things called "*rvalue references*"
    - Mostly beyond the scope of this class

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  // moves a to b
  std::string b{std::move(a)};
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;


  return EXIT_SUCCESS;
}
```

a: ""
b: "bleg"

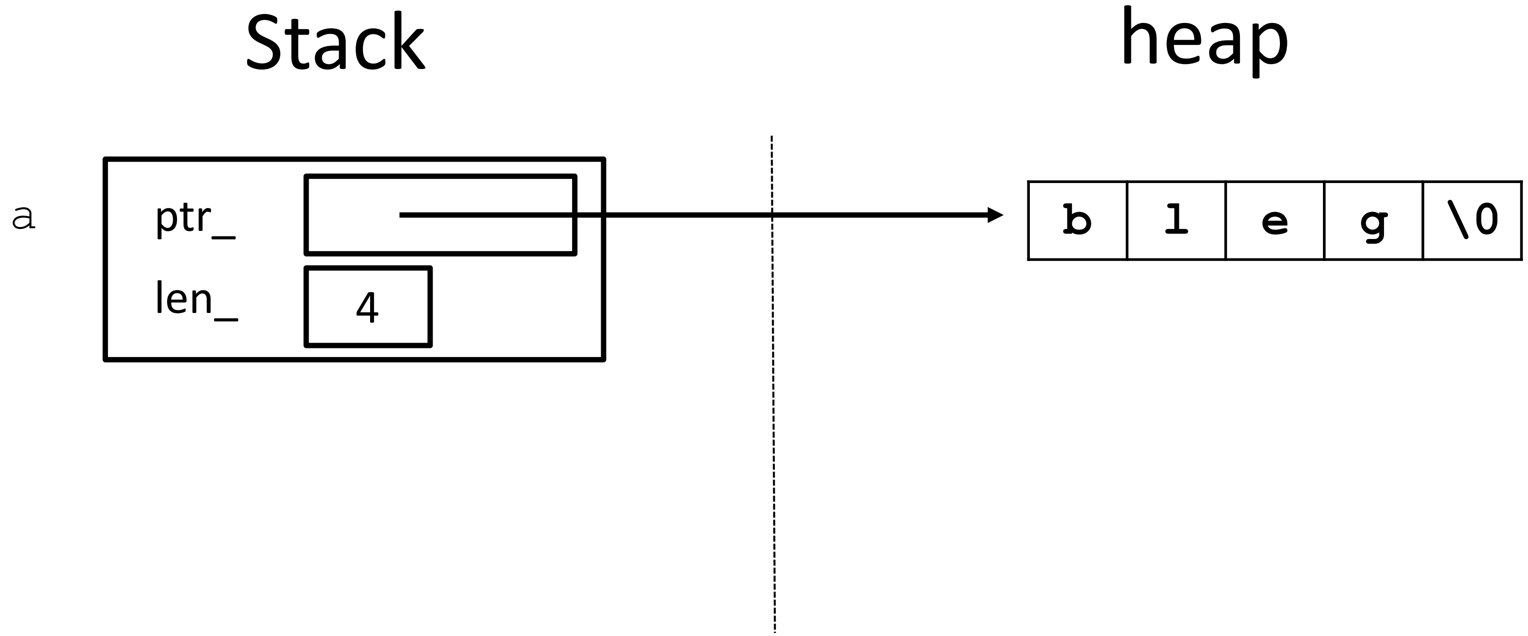
Note: we should NOT access 'a' after we move it. It is undefined to do so, it just so happens it is set to the empty string

Move Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a{"bleg"};


}
```



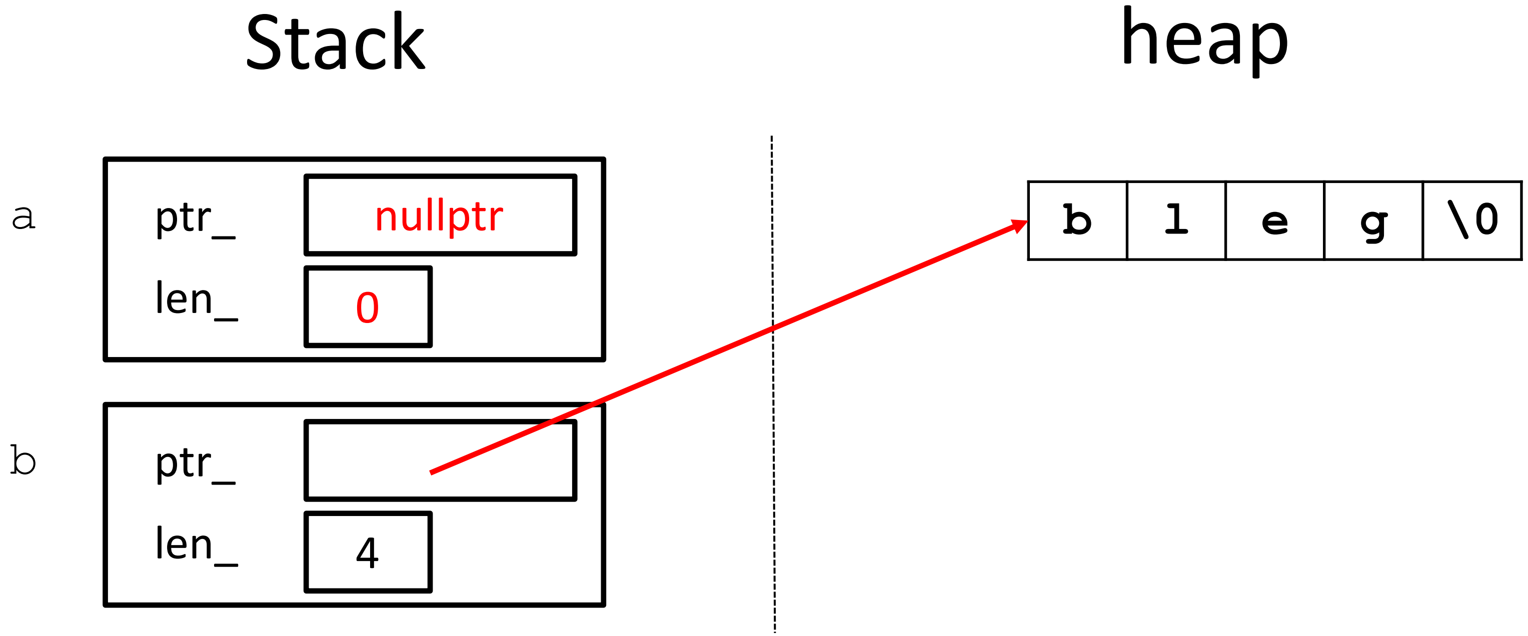
Move Semantics: close up look

- When we use move to construct string **b**

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  std::string b{std::move(a)};
}
```

we could get something like:



Move Semantics: Use Cases

- Useful for optimizing away temporary copies
- Preferred in cases where copying may be expensive
  - Consider we had a vector of strings… we could transfer ownership of memory to avoid copying the vector and each string inside of it.
- Can be used to help enforce uniqueness

- Rust is a systems programming language that is gaining popularity and by default it will move variables instead of copy them.

Move Semantics: Details

- Implement a "Move Constructor" with something like:

```
Point::Point(Point&& other) {
  // ...
}
```

- Implement a "Move assignment" with something like:

```
Point& Point::operator=(Point&& rhs) {
  // ...
}
```

Move Semantics: Details

- "Move Constructor" example for a fake **`String`** class:

```
String::String(String&& other) {
  this->len_ = other.len_;
  this->ptr_ = other.ptr_;

  other.len_ = 0;
  other.ptr_ = nullptr;
}
```

std::move

- Use `std::move` to indicate that you want to move something and not copy it

```
Point p {3, 2};            // constructor
Point a {p};               // copy constructor

Point b {std::move(p)}; // move constructor
```

Demo: Verbose Integer

- What happens when we resize?
- Making move operations noexcept
- What if this were strings and not ints?

Poll Everywhere

pollev.com/tqm

- Given a linked list object:
  - What do you think the copy constructor does?
  - What do you think the move constructor does?
  - (I don’t need code, high level idea is fine)

```
struct node {
  node* next;
  string value;
};
```

```
class LinkedList {
 public:
  LinkedList() {
    head_ = nullptr;
    tail_ = nullptr;
    len_ = 0;
  }

  LinkedList(const LinkedList& other) {
    // TODO: copy constructor
  }

  LinkedList(LinkedList&& other) {
    // TODO: move constructor
  }

 private:
  node* head_;
  node* tail_;
  size_t len_;
};
```

Lecture Outline

- C++ Programming Refresher
- Move Semantics
- **File Descriptors & Buffering**

From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface
  - **`open`**`()`, **`read`**`()`, **`write`**`()`, **`close`**`()`, **`lseek`**`()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient
  - C and C++ stdlib doesn't provide everything POSIX does
    - You will have to use these to read file system directories and for network I/O, so we might as well learn them now

open()/close()

- To open a file:
  - Pass in the filename and access mode
  - Get back a "file descriptor"
    - Similar to `FILE*` from **`fopen()`**, but is just an `int`
      - Returns `-1` to indicate error
    - Must manually close file when done ☹

Used to identify
a file w/ the OS

```
#include <fcntl.h>      // for open()
#include <unistd.h>     // for close()
  ...
  int fd = open("foo.txt", O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...
  close(fd);
```

Reading from a File

Stores read result in buf

Number of bytes

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

signed

  - Function is written in C: follows C design
    - Takes in a file descriptor
    - Takes in an array and length (In bytes) of where to store the results of the read
    - Returns number of bytes read
  - EVERY TIME we read from a file,
    this function is getting called somewhere
    - Even in Java or Python
    - There are wrappers around this, but they are all implemented on top of these system calls
    - The OS doesn't speak java or python, it "speaks" assembly and C so all languages must have a way to invoke these C functions.

Reading from a File

Stores read result in buf

Number of bytes

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

signed

- Function is written in C: follows C design
  - Takes in a file descriptor
  - Takes in an array and length of where to store the results of the read
- Returns the number of bytes read
  - Might be fewer bytes than you requested (**!!!**)
  - Returns **0** if you're already at the end-of-file
  - Returns **-1** on error (and sets `errno`)
  - Advances forward in the file by number of bytes read

Example Read Code

```
int fd = open(filename, O_RDONLY);
array<char, 1024> buf {};  // buffer of appropriate size
ssize_t result;

result = read(fd, buf.data(), 1024 * sizeof(char));
if (result == -1) {
  // an error happened, so exit the program
  // print out some error message to cerr
  exit(EXIT_FAILURE);
}

// If we want to construct a string from the bytes read
// we need to say how many bytes to take from the array.
string data_read(buf.data(), result);

// Whenever we are done with the file, we must close it
close(fd);
```

Poll Everywhere

pollev.com/tqm

- This code has some bugs, what are they? How do we fix this code?

```
char* read_stdin() {
  array<char, 1024> buf {};

  read(STDOUT_FILENO, buf.data(), 1024 * sizeof(char));

  return buf.data();
}

int main() {
  string input(read_stdin());

  cout << "You typed: " << input << endl;
}
```

Demo: read_stdin.cpp

Everything is a File (Descriptor)

- In Unix/Linux design, there is a uniform interface to interact with many aspects of the computer
  - Files
  - Network Sockets
  - Pipes
  - Special Device files
    - /dev/random
    - /usr/proc/<proc_id>/fds

Everything is Bytes

- In our computers, everything is stored as bits and bytes. We can read/write things other than characters. We just need to tell how many bytes to read

- Read an integer:

```
int fd = open(...);
int x;
read(fd, &x, sizeof(x));
```

- Write a struct:

```
struct Point {
  float x, y;
};

Point p{3.0F, 2.0F};
write(fd, &p, sizeof(p));
```

- Read a string? Why doesn’t this work

```
string x;
read(fd, &x, sizeof(x));
```

That’s it for now

- More next time!
  - Buffering refresher
  - Some misc C++ stuff we haven’t covered
    - Initializer list
    - Assignment operator
    - Casts
  - Maybeee virtual memory (briefly)