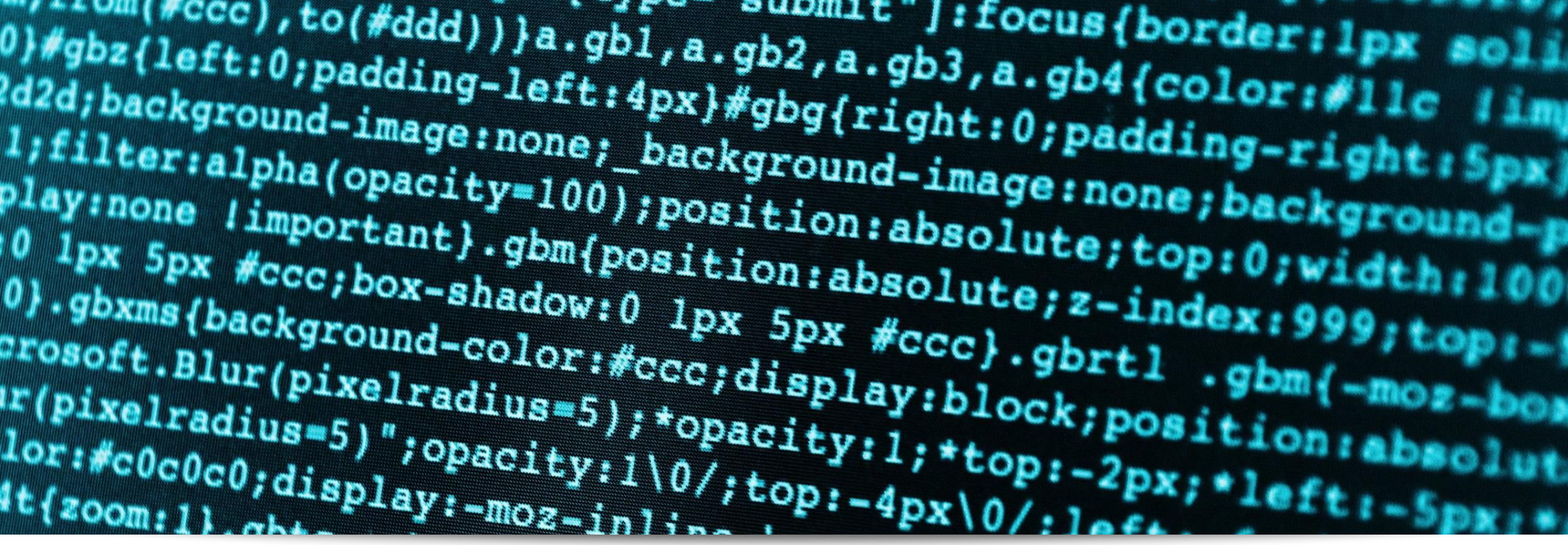


01

EECS 281 - Week of 01/13/2020

Makefiles, Getopt Long, Input Redirection, ADTs

Who am I?

- Ph.D Candidate in Computer Science
  - I study Human-Robot Interaction
- Second time GSI
- Been coding for 12 years now! and yet..
- From Mumbai, India
- Software Engineer at Microsoft for 3 years

Who am I?

- Ph.D Candidate in Computer Science
  - I study Human-Robot Interaction
- Second time GSI
- Been coding for 12 years now! and yet..
- From Mumbai, India
- Software Engineer at Microsoft for 3 years

Fun icebreaker!

Lab Attendance and Grading Policy

- Lab attendance is a part of your grade!
- **5 points** (25% of lab grade) for handwritten code in lab
  - must be completed during lab!
  - turning in work for someone who is not present is a **violation of the honor code!**
- You may attend any lab section as long as there is space
- Email us at eecs281admin@umich.edu if you have any issues

Lab logistics

- You can work on lab assignments with a partner
  - Partners may submit identical code and answers.

- ~30 minutes -> Work on the written problem
  - PLEASE PUT YOUR NAME AND UNIQNAME ON TOP!!
- ~60 minutes -> Lab material
- ~30 minutes -> Mini-office hours, work on the lab

Announcements

- Lab 1 Autograder and Quiz is due **Friday 1/31**
  - Useful for project 1
  - Helps with GetOpt
- Project 1 has been released **due 02/03**
- If you couldn’t make the How To session this past Sunday, check out the recording on Canvas

General 281 Tips

- **Start early!**
- Finish up lab assignments
- For projects
  - Start with test files
  - Get to office hours early (Profs go through concepts)
  - Write pseudocode, function design before writing code
  - Git is your friend (Use http://gitimmersion.com)
  - Code needs to run successfully on CAEN

General 281 Tips

- For debugging
  - Piazza posts
  - Use cerr and cout statements
  - Learn to use breakpoints on your IDE
  - Google error messages!

Compatibility with CAEN

Check Piazza post @76

- Open up Unix/Linux/Mac terminal or GitBash.
- Run the following command:

```
ssh <uniqname>@login.engin.umich.edu
```

  where <uniqname> is your uniqname.

- Sign in to your UMich account.
- Ensure that you are always running the right version of gcc:

```
cd ~
cat >> .bash_profile
module load gcc/6.2.0
<Ctrl+D>
```

Important Dates/Info

- Midterm: **2/26, 6:30pm – 8:30pm**
- Final: **4/27, 8am - 10am**
- Minimum Competency Rules:
  - 55% overall on projects
  - 50% overall on exams (curved)
  - 75% overall on labs
  - Guarantees a C

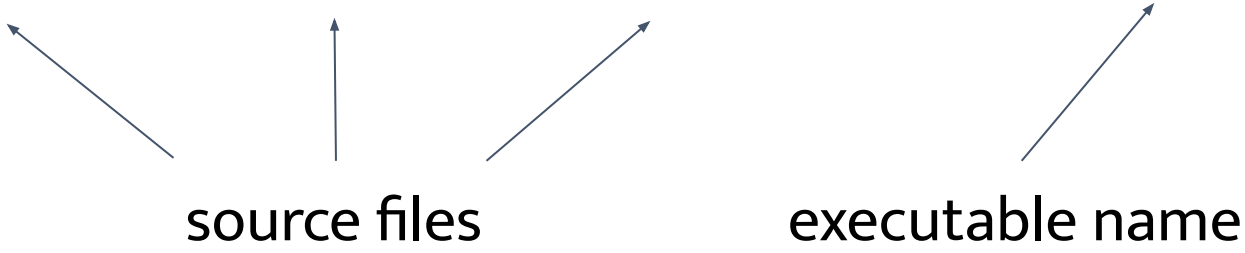
Agenda

- Makefile, Valgrind, perf
- C++ Input and Output
- Parsing Command Line Options Using Getopt
- Abstract Data Types - Stacks, Queues, Deques, Vectors
- Project 1 Tips
- Handwritten Problem

Makefiles

Compiling with g++

- Example: `g++ main.cpp file1.cpp file2.cpp -o main`



- Problems:
  - long to retype over and over
  - easy to mistype the command and produce wrong result
    - `g++ main -o main.cpp`
    - `g++ main.cpp -o main.cpp`

The Solution: Makefiles

- Make is a program that reads a description of a project from a Makefile (i.e. the file called “Makefile” in the current directory).

- Makefiles specify a set of compilation rules that make compiling easier.

- **Previous compilation command:**

```
g++ main.cpp file1.cpp file2.cpp -o main
```

- **Command using Make:** `make all`

Makefiles To-Dos

- We give you a Makefile to use for this course. You're free to modify it however you like, but you really only need to do four things:

1. Set the project identifier to the identifier given in the spec.

```
# Change IDENTIFIER to match the project identifier given in the project spec.
IDENTIFIER  = EEC50281EEC50281EEC50281EEC50281EEC50281
```

Makefiles To-Dos

2. Set the executable name to the name given in the project spec.

```
# Change EXECUTABLE to match the command name given in the project spec.
EXECUTABLE  = hunt
```

Makefiles To-Dos

3. Set the project file name to the name of the file in your program that has a main function (comment out the one you don't use).

```
# The following line looks for a project's main() in files named project*.cpp,
# executable.cpp (substituted from EXECUTABLE above), or main.cpp
PROJECTFILE = $(or $(wildcard project*.cpp $(EXECUTABLE).cpp), main.cpp)
# If main() is in another file delete line above, edit and uncomment below
#PROJECTFILE = mymainfile.cpp
```

use this line if your file name is NOT in the form of project*.cpp

use this line if your file name is in the form of project*.cpp, where * can be anything (e.g. project1.cpp)

Makefiles To-Dos

4. Set up any custom file dependencies.

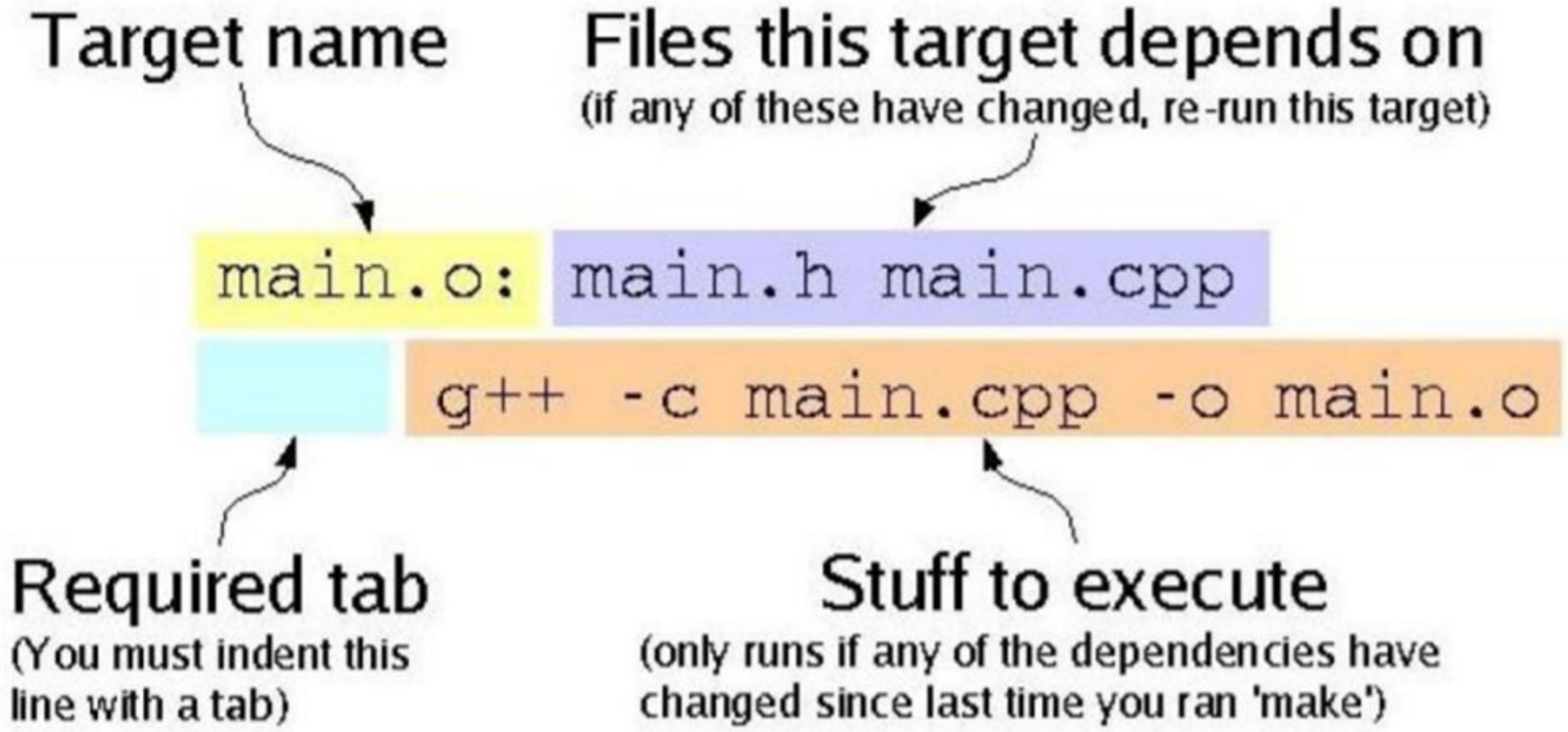
```
######################
# TODO (begin) #
######################
# individual dependencies for objects
# Examples:
# "Add a header file dependency"
# project2.o: project2.cpp project2.h
#
```

```
#
# ADD YOUR OWN DEPENDENCIES HERE

######################
# TODO (end) #
######################
```

TIP: Use Ctrl+F and search for the “TODO”s to identify the things you need to change!

Dependencies



Basic Makefile Commands

- `make` (or `make all`)
  - recompiles all files that have been changed and their dependencies, creates a new executable file
- `make clean`
  - removes all generated object files and the executable file
- `make debug`
  - use this when you run Valgrind: we will see this later!
- `make profile`
  - used when running profiling tools (perf)

Basic Makefile Commands

- `make partialsubmit`
  - creates a tarball in the current directory that does not include test cases
  - for compilation checks on the Autograder - **won't count as a submit** if your code does not compile
- `make fullsubmit`
  - creates a tarball in the current directory that does include test cases
  - **will count as a submit** regardless of whether it compiles or not
- tarball
  - a *.tar.gz file that contains a compressed copy of your program. The Autograder unwraps it and then compiles/runs your code.
  - submit the *.tar.gz file to the Autograder!

Basic Makefile Commands

- `make alltests`
  - compiles and generates executable files for all files of the form test*.cpp
  - cleans all tests generated before building
- `make test*`
  - builds executable for a specific test file

- Testing with Make:
  - write your test driver programs in test*.cpp files
  - files that you want to include in your final submission cannot match this pattern

Debugging with Print Statements

- Our Makefile allows you to utilize print statements that only print in debug mode (when you `make debug`). Simply use the `#ifdef` preprocessor directive, as shown below:

```
int main() {
   #ifdef DEBUG
   std::cout << "This only prints in debug mode!\n";
   #endif
   return 0;
}
```

Valgrind

Valgrind

- Valgrind is used to detect undefined behavior such as:
  - the use of uninitialized values - even inside an array or dynamic memory
  - out-of-bounds reads (“invalid read of size...”)
  - out-of-bounds writes (“invalid writes of size...”)
  - memory leaks
    - however, all of the STL containers use dynamic memory
    - the C function `exit(status)` will stop the program **without** calling any container destructors, which may leave your program with a bunch of memory leaks
  - memory profiling

Valgrind

Buggy Script:

```
 7: int main() {
 8:   vector<int> foo = {1, 2, 3};
 9:   for (int i = 0; i <= 3; i++) {
10:      cout << foo[i] << endl;
11:   }
12: }
```

Running Valgrind in CAEN:

make clean

`make debug`

`valgrind ./<executable_name_debug> [options/flags]`

Valgrind

Buggy Script:

```
 7: int main() {
 8:   vector<int> foo = {1, 2, 3};
 9:   for (int i = 0; i <= 3; i++) {
10:      cout << foo[i] << endl;
11:   }
12: }
```

Running Valgrind in CAEN:

```
make debug
valgrind ./<executable_name>
```

Valgrind Output:

location where the bad memory access occurred. **If you don't see this, you didn't compile with -g3.**

12 bytes = 3*(4 bytes) = 3 ints; the array contains only 3 things, but you asked for a 4th!

```
==30809== Invalid read of size 4
==30809==    at 0x400B3E: main (main.cpp:10)
==30809== Address 0x5aa4c8c is 0 bytes after a block of size 12 alloc'd
```

Valgrind

- Always Valgrind code before submitting to the Autograder!
  - If Valgrind detects errors, you will lose 5% for memory leaks, even if you didn't leak memory.
  - If you have undefined behavior, it may cause erroneous output.
- Once you know what lines are causing problems, you can examine further with gdb, an IDE debugger, etc.

Demo: Makefile and Valgrind

```
==12883== Invalid write of size 1
==12883==    at 0x401A7B: FASTTSP_generator(print_FASTTSP&, std::vector<nodesB, std::allocator<nodesB> >&) (zoo.cpp
:166)
==12883==    by 0x40257E: main (zoo.cpp:474)
==12883==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==12883==
==12883==
==12883== Process terminating with default action of signal 11 (SIGSEGV)
==12883==  Access not within mapped region at address 0x0
==12883==    at 0x401A7B: FASTTSP_generator(print_FASTTSP&, std::vector<nodesB, std::allocator<nodesB> >&) (zoo.cpp
:166)
==12883==    by 0x40257E: main (zoo.cpp:474)
==12883==  If you believe this happened as a result of a stack
==12883==  overflow in your program's main thread (unlikely but
==12883==  possible), you can try to increase the size of the
==12883==  main thread stack using the --main-stacksize= flag.
==12883==  The main thread stack size used in this run was 8388608.
==12883==
==12883== HEAP SUMMARY:
==12883==     in use at exit: 4 bytes in 1 blocks
==12883==   total heap usage: 2 allocs, 1 frees, 72,708 bytes allocated
==12883==
==12883== LEAK SUMMARY:
==12883==    definitely lost: 0 bytes in 0 blocks
==12883==    indirectly lost: 0 bytes in 0 blocks
==12883==      possibly lost: 0 bytes in 0 blocks
==12883==    still reachable: 4 bytes in 1 blocks
==12883==         suppressed: 0 bytes in 0 blocks
==12883== Rerun with --leak-check=full to see details of leaked memory
==12883==
==12883== For counts of detected and suppressed errors, rerun with: -v
==12883== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault
```

perf

perf

- Is your program too slow? Do you want to know where your bottleneck is? If so, use perf!
  - How much time is being spent in different portions of your code?
  - This is (part of) why it is important to separate your code into different functions - perf will be of little help if you have a 1000-line main!

perf

- To run perf properly, make sure you run with the -g3 or -O3 flags while compiling. This can be done with `make debug`.
- Run the following code:

```
make profile
perf record --call-graph=dwarf -e cycles:u time ./program < [input]
perf report
```

- **Demo time!**
- You will get practice with perf in this week's lab assignment.

C++ Input and Output

C++ Input/Output

- **cin**: read from stdin
- **cout**: write to stdout
- **cerr**: write to stderr
- **ifstream**: open files with read permission
- **ofstream**: open files with write permission
- **fstream**: open files with read and write permission

in this course, you probably won't use these

Redirecting Input from Files

- In this class, you will be redirecting the standard input stream (cin) to come from a file rather than a console.
- Example: `./path281 < input.txt`

indicates that the next entry on command line will be the file name that you want cin to be associated with

  - in this case, < and `input.txt` are **not** command line arguments, and they **don't appear** in `char* argv[]`
  - do NOT open files if you are using input/output redirection
- Directions for redirecting input on XCode and VS can be found on Canvas.

Redirecting Output to Files

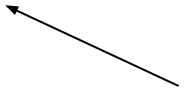
- On the command line, you can also specify a file that you want to associate with the standard output stream (cout).
  - Example: `./path281 > output.txt`
  - You can then use `cout << var` to write to output.txt

C++ stdin

- To read in input, do **NOT** use `cin.eof`, `cin.good`, `cin.bad`, `cin.fail`
- Conversion after extracting data from an input stream behaves like a boolean, so you can use it to control read loops in your programs.

```
while (cin >> new_value) {
   // only executes if new_value
   // is read in properly
}

while (getline(cin, new_line)) {
   // only executes if new_line
   // is read in properly
}
```



if you are done reading the file, `cin >> new_value` becomes false and the while loop terminates

Reading Char by Char

- The code below **does not** preserve whitespace.

```
int main(int argc, char* argv[]) {
   char c;
   while (cin >> c) {
      cout << c;
   }
}
```

- What gets printed? `EECS281isfun`

text.in

```
EECS 281
is fun
```

Operator >>

- The >> operator
  - **ignores** leading whitespace
  - consumes a "word" (characters until the next whitespace/end of line or file)

- Stream Extraction Example:

```
string word;
while (cin >> word) {
   // do something
}
```

File to be read:

"•" represents a space

"¶" represents a new line

```
••• there ••• are¶
••• 1253 ••• words
```

word ""

Reading Line by Line

- The code below **does** preserve whitespace.

```
int main(int argc, char* argv[]) {
   string s;
   while (getline(cin, s)) {
      cout << s << endl;
   }
}
```

- What gets printed?

```
EECS 281
is fun
```

text.in

```
EECS 281
is fun
```

getline

- getline
  - consumes *all* characters (even whitespace) until a given one (default newline)
  - removes and discards the given character

- Stream Extraction Example:

```
string line;
while (getline(cin, line)) {
   // do something
}
```

File to be read:

"•" represents a space

"¶" represents a new line

```
••• there ••• are¶
••• 1253 ••• words
```

line

```
""
```

getline: Common Mistakes

- Watch out: if you are using both >> and getline, >> does not read in spaces or newlines at the end of a line. Thus, make sure to get rid of all spaces before the next new line before using getline.

- Example:

```
int n;
cin >> n;                  // n is 5
string word;
getline(cin, word); // word is "•••"
                           // not "apple"!
```

text.in

```
••• 5 •••¶
apple¶
banana¶
cactus¶
dog¶
elephant¶
```

getline: Common Mistakes

- Watch out: if you are using both >> and getline, >> does not read in spaces or newlines at the end of a line. Thus, make sure to get rid of all spaces before the next new line before using getline.

- Example (fixed):

```
int n;
cin >> n;                  // n is 5
string junk;
getline(cin, junk); // junk is "•••"
string word;
getline(cin, word); // word is "apple"
```

text.in

```
••• 5 •••¶
apple¶
banana¶
cactus¶
dog¶
elephant¶
```

Getopt Long

getopt_long

- `getopt_long` is a function that helps to automate command line parsing.
- Command line examples:
  - `./project0 --first 5 -s`
  - `./project0 --summary -f5`
  - `./project0 -sf 5`
  - `./project0 --first 5 summary`
  - `./project0 -f 5 -s < input.txt`
- All of the above commands are equivalent, and your program should behave the same for all of them!
- `getopt_long` takes the work out of accounting for all these different possibilities.

getopt_long

```
1   #include <getopt.h>
2   using namespace std;
3
4   int main(int argc, char *argv[]) {
5       int gotopt;
6       int option_index = 0;
7       option long_opts[] = {
8           { "action", no_argument, nullptr, 'a' },
9           { "number", required_argument, nullptr, 'n' },
10          { nullptr, 0, nullptr, '\0' },
11      };
12
13      while ((gotopt = getopt_long(argc, argv, "an:", long_opts, &option_index)) != -1) {
14          switch (gotopt) {
15              case 'a':
16                  cout << "Action!!!\n";
17                  break;
18              case 'n':
19                  cout << "Input number is: " << atoi(optarg) << "\n";
20                  break;
21              default:
22                  cout << "Oh no! I didn't recognize your flag.\n";
23                  exit(0);
24                  break:
25          }
26      } // while
27  } // main
```

options with required arguments are followed by a ':'

option arguments are automatically stored in a global variable called `optarg`

`optarg` is a `char*`, not a `std::string`

Abstract Data Types

Abstract Data Types

- This *interface* to the data (the operations) is called an abstract data type.
- The *implementation* of the interface is called a data structure.

- We want you to understand:
  - **when** the use of each ADT is called for
  - **how** the use of each ADT affects time and space usage
  - how and when to use ADTs that are **more efficient with time and space** if the complete functionality of a called-for ADT is not required

Stacks and Queues

- For each ADT, we will define a set of operations and their behaviors.

Operation	Stack Behavior	Queue Behavior
push(value)	append value on top of stack	append value at back of queue
pop()	remove top value from stack	remove value at front of queue
top()/front()	return top value of stack	return value at front of queue
size()	return # of elements in stack	return # of elements in queue
empty()	return whether size() is 0	return whether size() is 0

- Random access of elements in the middle is not supported. If we want that, we'll need a different ADT. What's a use case for stacks? Queues?
- We'll cover these in more depth in later labs and in lecture.

Deques

- When we have a program that can run using either a stack or a queue to manage some data. Use deques to reuse code more effectively

Operation	Behavior	Operation	Behavior
push_front(value)	append value to front of deque	push_back(value)	append value to back of deque
pop_front()	remove value from front of deque	pop_back()	remove value from back of deque
front()	return value at front of deque	back()	return value at back of deque
size()	return # of elements in deque	empty()	return whether size() is 0

- `push_back` and `pop_back` to simulate a stack,
- `push_back` and `pop_front` to simulate a queue
- Deques also support `operator[position]`, giving them efficient random access to all elements. This means they can also be used to represent a **list** of items.

Vectors

- Vectors are similar to deques, but lose `push_front(value)` and `pop_front()` in exchange for better performance.
- We'll cover the implementation later, but for now, you should keep in mind these two things:
  - Use `resize(new_size)` or `reserve(new_capacity)`.
  - Use vectors to hold reference data that can be identified by indices.

Resize and Reserve

- Consider the following:

```
vec.resize(new_size);

vec.reserve(new_capacity);
```

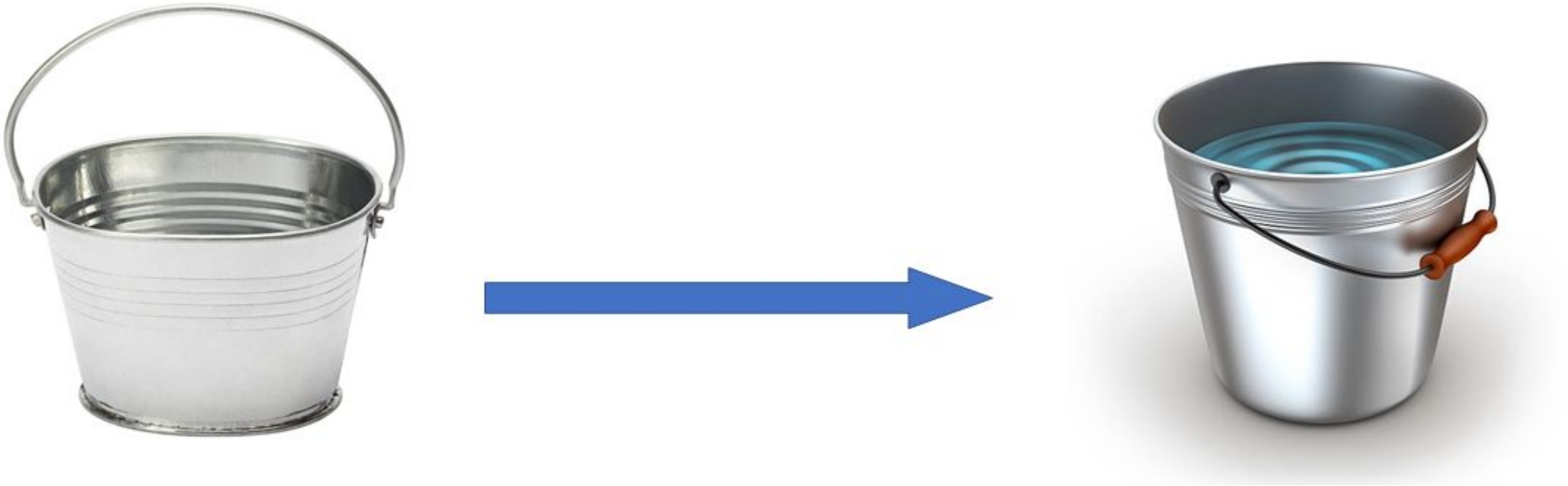
- What is the difference?
  - Resize changes **size**.
  - Reserve changes **capacity**.

Resize and Reserve

- Let's consider an analogy: suppose you had a bucket, and you wanted to put water in it. Let this bucket represent a vector.

Resize and Reserve

- When you **resize** the bucket (or vector), you change the *amount of water in the bucket.*



Resize and Reserve

- When you call **reserve**, you change the capacity of the bucket, or *how much water it can hold*.
- Note: calling **reserve** does <u>not</u> actually add water to the bucket!

Resize and Reserve

- `vec.resize(new_size)`
  - changes **size** (and increases capacity if needed)
  - calling `vec.push_back(x)` after resizing adds x AFTER newly created items
- `vec.reserve(new_capacity)`
  - does NOT change size, but increases **capacity** (if needed)
  - calling `vec.push_back(x)` adds x as the next element normally (the relative position it would have been added without the call to reserve)
  - does not do anything if `new_capacity` is smaller than the current capacity

Resize and Reserve

- Example:

```
vector<int> vec;
vec.reserve(10);

// What happens here?
vec[0] = 5;

// Is this true or false?
vec.empty();
```

Resize and Reserve

- Example:

```
vector<int> vec;
vec.reserve(10);

// What happens here?
vec[0] = 5;

// Is this true or false?
vec.empty();
```

undefined behavior - you changed the capacity of your bucket, but your bucket is still empty! Remember that reserve doesn't actually add elements to the vector. Thus, there isn't an element 0 yet for you to access.

this is true, since the vector doesn't contain anything yet

Resize and Reserve

- In summary:
  - Resizing changes *the number of elements* in a vector.
  - Reserving changes *how many elements* a vector can hold.
- Details will be covered later
- If size exceeds capacity, the capacity is automatically doubled (this is **expensive)**
- If you know what the size will be in advance, use one of these functions (or preset the size when calling the constructor).

Multi-Dimensional Vectors

What do we do if we want to have a two- or three- dimensional vector?

Say we wanted to make a 2D vector of size 10x10 containing 0. We might initialize it like this:

```
vector< vector<int> > my_vec;

while(my_vec.size() < 10) {
    vector<int> temp;

    while(temp.size() < 10){
        temp.push_back(0);
    }
    my_vec.push_back(temp);
}
```

Multi-Dimensional Vectors

But we know the size in advance!

we should either use `resize` and `reserve`, or initialize the size in the constructor call

```
vector< vector<int> > my_vec;
my_vec.reserve(10);

while(my_vec.size() < 10) {
    vector<int> temp(10, 0);

    my_vec.push_back(temp);
}
```

Multi-Dimensional Vectors

We can do all of this one line! This line initializes the entire 2D vector in one line, using an internal call to the constructor for the 1D vector.

first parameter: **how many elements** you want the vector to have

second parameter: what each new element is **initialized to** (here, it's a vector initialized to size 10 with 0's)

```
vector< vector<int> > my_vec(10, vector<int>(10, 0));
```

What about for 3D vectors (or higher dimensions)? Do more of the same.

```
vector< vector< vector<int> > > my_vec(10, vector< vector<int> >(10, vector<int>(10,0)));
```

You'll find these useful in Project 1.

Practice

- What ADT(s) will come in useful? Is there any way we can use a faster ADT instead?
  - We want to keep a list of the names of people in the order that they entered a classroom, and be able to find the name of the nth person who entered.

- We want to simulate travelling from one road intersection to another, and then backtrack in the exact opposite order once some condition is met.
- We want to serve requests received by a modem in the same order that they were received in.
- We want to retrieve a list of students who picked each number between 0 and 10 when asked for a random number between 0 and 10.

Practice

- What ADT(s) will come in useful? Is there any way we can use a faster ADT instead?
  - We want to keep a list of the names of people in the order that they entered a classroom, and be able to find the name of the nth person who entered. **Vector**
  - We want to simulate travelling from one road intersection to another, and then backtrack in the exact opposite order once some condition is met.
  - We want to serve requests received by a modem in the same order that they were received in.
  - We want to retrieve a list of students who picked each number between 0 and 10 when asked for a random number between 0 and 10.

Practice

- What ADT(s) will come in useful? Is there any way we can use a faster ADT instead?
  - We want to keep a list of the names of people in the order that they entered a classroom, and be able to find the name of the nth person who entered. **Vector**
  - We want to simulate travelling from one road intersection to another, and then backtrack in the exact opposite order once some condition is met. **Stack**
  - We want to serve requests received by a modem in the same order that they were received in.
  - We want to retrieve a list of students who picked each number between 0 and 10 when asked for a random number between 0 and 10.

Practice

- What ADT(s) will come in useful? Is there any way we can use a faster ADT instead?
  - We want to keep a list of the names of people in the order that they entered a classroom, and be able to find the name of the nth person who entered. **Vector**
  - We want to simulate travelling from one road intersection to another, and then backtrack in the exact opposite order once some condition is met. **Stack**
  - We want to serve requests received by a modem in the same order that they were received in. **Queue**
  - We want to retrieve a list of students who picked each number between 0 and 10 when asked for a random number between 0 and 10.

Practice

- What ADT(s) will come in useful? Is there any way we can use a faster ADT instead?
  - We want to keep a list of the names of people in the order that they entered a classroom, and be able to find the name of the nth person who entered. **Vector**
  - We want to simulate travelling from one road intersection to another, and then backtrack in the exact opposite order once some condition is met. **Stack**
  - We want to serve requests received by a modem in the same order that they were received in. **Queue**
  - We want to retrieve a list of students who picked each number between 0 and 10 when asked for a random number between 0 and 10. **Map or Vector**

Handwritten Problem

- Pull out a piece of paper and CLEARLY write your **name** and **uniqname** at the top. Completion of this written problem is worth 5 points.

```
struct Node {
  char value;
  Node* prev;
  Node* next;
};
```

Watch out! The nodes in a linked list are **NOT** contiguous in memory... make sure your comparisons aren't assuming that they are!

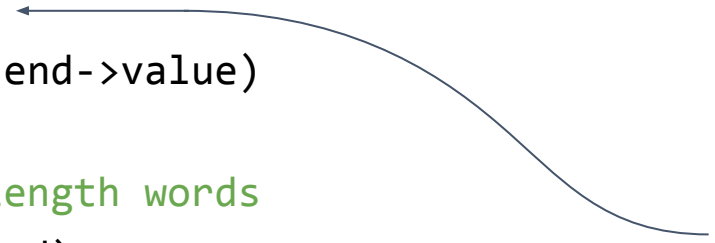
```
// check if a doubly-linked list is a palindrome
bool isPalindrome(Node* start, Node* end);
```

- Write the implementation for isPalindrome, O(1) space and O(*n*) time.

Lab 1 Written Problem: Linked List Palindrome

- Check if a doubly-linked list is a palindrome.
- One possible solution:

```
bool isPalindrome(Node* start, Node* end) {
    // Breaks when they meet in the middle
    // or when they both traverse the whole list.
    while (start != end) {
        if (start->value != end->value)
            return false;
        // Needed for even-length words
        if (start->next == end)
            return true;
        start = start->next;
        end = end->prev;
    } // while
    return true;
} // isPalindrome()
```



cannot use “less than” since linked lists are not contiguous in memory!