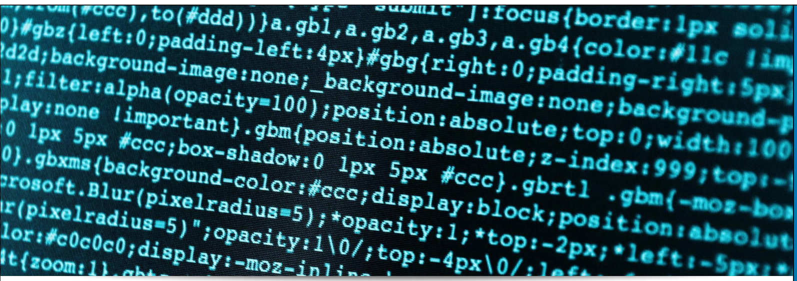


01

September 3-9

Makefiles, Getopt Long, Input Redirection, ADTs

Announcements

- Lab 1 Handwritten Problem due by **Monday, September 9 IN LAB**
- Lab 1 Autograder portion due **Monday, September 16 (by 11:59pm)**
- Lab 1 Quiz Assignment due **Monday, September 16 (by 11:59pm)**
    - Make sure you get in the habit of starting early!
    - If you always do your lab assignments near the deadline, you might get into trouble with higher workload labs that appear later in the semester.
    - You don't want to be losing points on lab assignments, especially since projects and exams will be a lot tougher!
- Project 1 has been released! **Due on 9/17!!!**

Course Breakdown

Assignment	Percentage of Grade
Project 1	10%
Project 2	10%
Project 3	10%
Project 4	10%
Midterm Exam	20%
Final Exam	20%
Lab Assignments	20%

Compatibility with CAEN

- Your code MUST compile and run with g++ as it is on the CAEN machines.
- You still might want to develop on your own computer.
- If you do, you must find a way to test on CAEN g++ as well.

- Open up Unix/Linux/Mac terminal or GitBash.
- For the following commands, **DO NOT copy/paste from the PDF**; there are invisible characters that could make it fail and/or cause future problems
- Run the following command, where you is your uniqname:

```
ssh you@login.engin.umich.edu "echo 'module load gcc/11.3.0' >> ~/.bash_profile"
```

- Sign in to your UMich account (again changing to your uniqname):

```
ssh you@login.engin.umich.edu
```

- Ensure you are running the right version of g++ by running this command, checking that it says `11.3.0`:

```
g++ --version
```

Workflow Advice

- You can sign in using your UofM info and create a git repo at https://gitlab.eecs.umich.edu or https://github.com
    - **Make it private to avoid honor code violations!**
- If you don't use git, either:
    - SFTP/SCP/RSYNC Transfer from your local machine (Mac is built in, PuTTY/PSFTP can be used for PC users).
    - Use a 3rd party program such as FileZilla or WinSCP.
- If you choose to develop on your machine and copy to CAEN only when you finish, leave plenty of time to debug because small errors can become system-crashers on a different system!

Makefiles

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

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

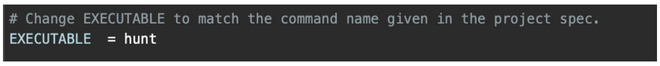
```
# Change IDENTIFIER to match the project identifier given in the project spec.
IDENTIFIER  = EEC50281EEC50281EEC50281EEC50281EEC50281
```

Makefiles To-Dos

- 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:

②

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



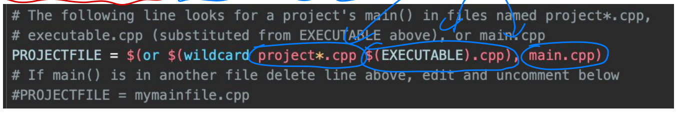
```
# Change EXECUTABLE to match the command name given in the project spec.
EXECUTABLE  = hunt
```

Makefiles To-Dos

- 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:

3

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
```

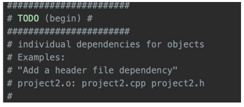
use this line if your file name is NOT in the form of project*.cpp
use this line if your file name is in the form of project*.cpp, where * can be anything (e.g. project1.cpp)

Makefiles To-Dos

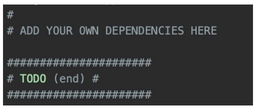
- We give you a Makefile to use for this course. You're free to modify it however you like, but you really only need to do four things:

4

4. Set up any custom file dependencies.



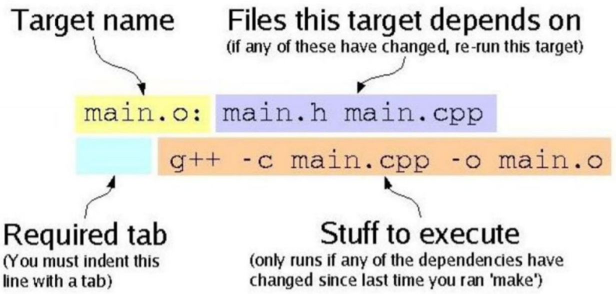
```
######################
# TODO (begin) #
######################
# individual dependencies for objects
# Examples:
# "Add a header file dependency"
# project2.o: project2.cpp project2.h
#
```



```
#
# 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


Target name
Files this target depends on
(if any of these have changed, re-run this target)
main.o:
main.h main.cpp
g++ -c main.cpp -o main.o
Required tab
(You must indent this line with a tab)
Stuff to execute
(only runs if any of the dependencies have changed since last time you ran 'make')

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`
  - compile your code for general debugging, includes the address sanitizer
- `make valgrind`
  - 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
  - other 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
    - you won't need to manage much memory manually in 281
    - 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
      - fine in real life, but we need some way to grade for 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 valgrind
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 10% 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.

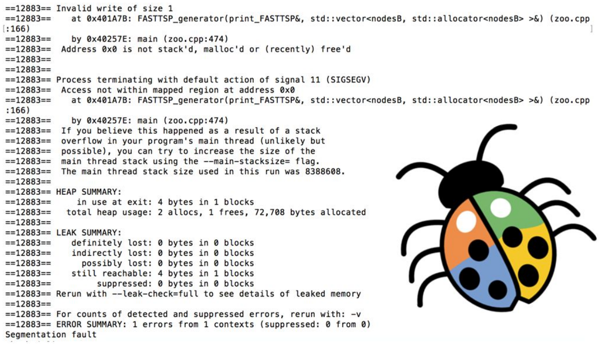
gdb Debugger

- Text-based debugging tool
- Useful for solving segfaults (i.e. program received signal SIGSEGV) and memory issues (e.g. index out of bounds)
- **Note:** the debuggers built into IDEs are a lot easier to use, so they are recommended!
- To use, type gdb `<executable_name>`
  - gdb is now waiting
- To start the program, type run `<command_line>`

gdb Helpful Commands

- **(r)**un: start the executable
- **(b)**reak: sets points where gdb will halt
- **where**: prints function line where segfault occurred
- **(b)**ack**(t)**race: prints full chain of function calls
- **(s)**tep: executes current line of the program, enters function calls
- **(n)**ext: like step but does not enter functions
- **(c)**ontinue: continue to the next breakpoint
- **(p)**rint <var>: prints the value of <var>
- **watch** <var>: watch a certain variable
- **(l)**ist <lineNum>: list source code near <lineNum>
- **kill**: terminate the executable
- **(q)**uit: quit gdb

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

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

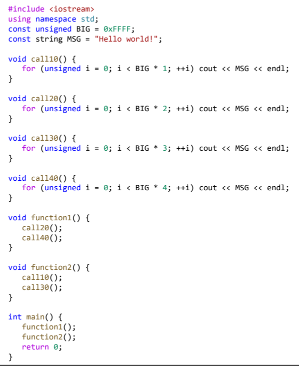
perf

perf

- Is your program too slow? Do you want to know where your bottleneck is? If so, use perf!
  - perf tells you 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!
- To run perf properly, make sure you run with the -g3 or -Og flags while compiling. This can be done with `make debug`
- Run the following code:

```
make debug
./program_debug # make sure that it runs, if not check the module load instructions
perf record -F 1000 --call-graph dwarf -e cycles:u ./program_debug < [input]
perf report
```

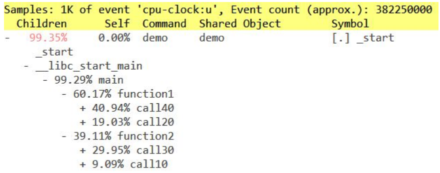
perf



```
#include <iostream>
using namespace std;
const unsigned BIG = 0xFFFF;
const string MSG = "Hello world!";

void call10() {
   for (unsigned i = 0; i < BIG * 1; ++i) cout << MSG << endl;
}

void call20() {
   for (unsigned i = 0; i < BIG * 2; ++i) cout << MSG << endl;
}

void call30() {
   for (unsigned i = 0; i < BIG * 3; ++i) cout << MSG << endl;
}

void call40() {
   for (unsigned i = 0; i < BIG * 4; ++i) cout << MSG << endl;
}

void function1() {
   call20();
   call40();
}

void function2() {
   call10();
   call30();
}

int main() {
   function1();
   function2();
   return 0;
}
```

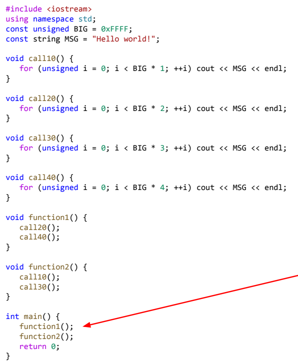
Let's run perf on this code
Can be found in canvas under LAB01/Lab Slides and Resources/Perf Demo

What does this output mean?



```
Samples: 1K of event 'cpu-clock:u', Event count (approx.): 382250000
  Children      Self  Command  Shared Object        Symbol
-   99.35%     0.00%  demo     demo                 [.] _start
     _start
   - __libc_start_main
      - 99.29% main
         - 60.17% function1
            + 40.94% call40
            + 19.03% call20
         - 39.11% function2
            + 29.95% call30
            + 9.09% call10
```

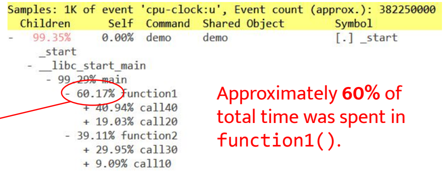
perf



```
#include <iostream>
using namespace std;
const unsigned BIG = 0xFFFF;
const string MSG = "Hello world!";

void call10() {
   for (unsigned i = 0; i < BIG * 1; ++i) cout << MSG << endl;
}

void call20() {
   for (unsigned i = 0; i < BIG * 2; ++i) cout << MSG << endl;
}

void call30() {
   for (unsigned i = 0; i < BIG * 3; ++i) cout << MSG << endl;
}

void call40() {
   for (unsigned i = 0; i < BIG * 4; ++i) cout << MSG << endl;
}

void function1() {
   call20();
   call40();
}

void function2() {
   call10();
   call30();
}

int main() {
   function1();
   function2();
   return 0;
}
```

Let's run perf on this code
Can be found in canvas under LAB01/Lab Slides and Resources/Perf Demo

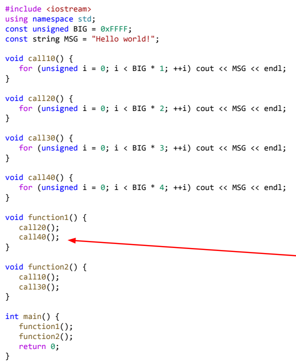
What does this output mean?



```
Samples: 1K of event 'cpu-clock:u', Event count (approx.): 382250000
  Children      Self  Command  Shared Object        Symbol
-   99.35%     0.00%  demo     demo                 [.] _start
     _start
   - __libc_start_main
      - 99.29% main
         - 60.17% function1
            + 40.94% call40
            + 19.03% call20
         - 39.11% function2
            + 29.95% call30
            + 9.09% call10
```

Approximately **60%** of total time was spent in `function1()`.

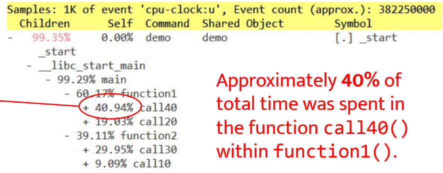
perf



```
#include <iostream>
using namespace std;
const unsigned BIG = 0xFFFF;
const string MSG = "Hello world!";

void call10() {
   for (unsigned i = 0; i < BIG * 1; ++i) cout << MSG << endl;
}

void call20() {
   for (unsigned i = 0; i < BIG * 2; ++i) cout << MSG << endl;
}

void call30() {
   for (unsigned i = 0; i < BIG * 3; ++i) cout << MSG << endl;
}

void call40() {
   for (unsigned i = 0; i < BIG * 4; ++i) cout << MSG << endl;
}

void function1() {
   call20();
   call40();
}

void function2() {
   call10();
   call30();
}

int main() {
   function1();
   function2();
   return 0;
}
```

Let's run perf on this code:
Can be found in canvas under LAB01/Lab Slides and Resources/Perf Demo

What does this output mean?



```
Samples: 1K of event 'cpu-clock:u', Event count (approx.): 382250000
  Children      Self  Command  Shared Object        Symbol
-   99.35%     0.00%  demo     demo                 [.] _start
     _start
   - __libc_start_main
      - 99.29% main
         - 60.17% function1
            + 40.94% call40
            + 19.03% call20
         - 39.11% function2
            + 29.95% call30
            + 9.09% call10
```

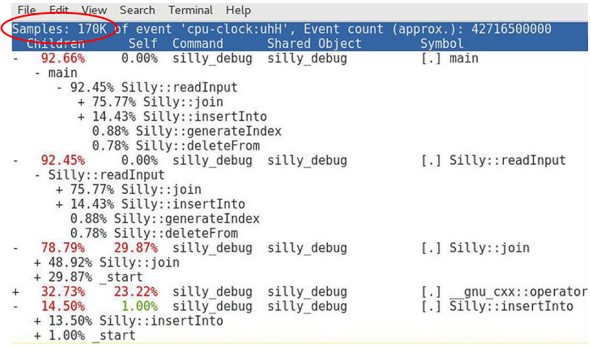
Approximately **40%** of total time was spent in the function `call40()` within `function1()`.

perf

What's this number, and why is it important?

When you run perf, samples are taken when your program runs. perf determines the percentage of samples that are taken in each of the operations/functions listed in the report.

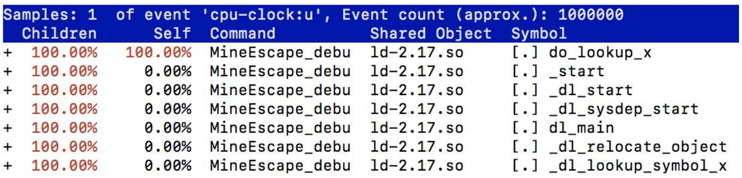
Thus, the more samples you have, the more accurate your perf report will be!



```
Samples: 170K of event 'cpu-clock:uhH', Event count (approx.): 42716500000
  Children      Self  Command      Shared Object        Symbol
-   92.66%     0.00%  silly_debug  silly_debug          [.] main
   - main
      - 92.45% Silly::readInput
         + 75.77% Silly::join
         + 14.43% Silly::insertInto
           0.88% Silly::generateIndex
           0.78% Silly::deleteFrom
-   92.45%     0.00%  silly_debug  silly_debug          [.] Silly::readInput
   - Silly::readInput
      + 75.77% Silly::join
      + 14.43% Silly::insertInto
        0.88% Silly::generateIndex
        0.78% Silly::deleteFrom
-   78.79%    29.87%  silly_debug  silly_debug          [.] Silly::join
   + 48.92% Silly::join
   + 29.87% _start
+   32.73%    23.22%  silly_debug  silly_debug          [.] __gnu_cxx::operator
-   14.50%     1.00%  silly_debug  silly_debug          [.] Silly::insertInto
   + 13.50% Silly::insertInto
   + 1.00% _start
```

perf

- Thus, if you get a result like this… run perf on a larger file!
  - perf only took 1 sample here, and that sample was taken during do_lookup_x
  - 100% of your runtime is taking place during this operation?!
  - Run on a larger input file for more valuable information (also, check for typos)
- See the perf Reference Guide in the Resources folder for more information on how to use perf.



```
Samples: 1  of event 'cpu-clock:u', Event count (approx.): 1000000
  Children      Self  Command          Shared Object  Symbol
+  100.00%   100.00%  MineEscape_debu  ld-2.17.so     [.] do_lookup_x
+  100.00%     0.00%  MineEscape_debu  ld-2.17.so     [.] _start
+  100.00%     0.00%  MineEscape_debu  ld-2.17.so     [.] _dl_start
+  100.00%     0.00%  MineEscape_debu  ld-2.17.so     [.] _dl_sysdep_start
+  100.00%     0.00%  MineEscape_debu  ld-2.17.so     [.] dl_main
+  100.00%     0.00%  MineEscape_debu  ld-2.17.so     [.] _dl_relocate_object
+  100.00%     0.00%  MineEscape_debu  ld-2.17.so     [.] _dl_lookup_symbol_x
```

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, do not use these in place of redirection

Redirecting Input from Files

- In this class, instead of opening an input file using ifstream or ostream, you will often redirect 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[]`
  - You can now use `getline(cin, var)` or `cin >> var` to read in the input file
- 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 `""`

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 `""`

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 `"there"`

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 `"there"`

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 "are"

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 "are"

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 "1253"

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 "1253"

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 "words"

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
}
```

cin >> word is now false, so while loop terminates

File to be read:
"•" represents a space
"¶" represents a new line

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

word "words"

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 "••• there ••• are"

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

- 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 `"••• there ••• are"`

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 `"••• 1253 ••• words"`

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
}
```

`getline(cin, line)` is now false, so while loop terminates

File to be read:
"•" represents a space
"¶" represents a new line

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

line `"••• 1253 ••• words"`

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¶
```

I/O Tips

- When I/O becomes a bottleneck, avoid reading/writing character-by-character or word-by-word.
- While C++ streams are slower than stdlibc I/O, this can be changed by setting the following on the first line of main:

```
std::ios_base::sync_with_stdio(false);
```

- What does this line of code do? It cuts down on runtime by specifying the C++ and C I/O do not need to be synced. Optimization is just a side effect of this operation, so do not blindly include it in everything just to make things run faster. **However, for the purposes of this class, it is safe (and imperative) that you always include this line in your code!**

I/O Tips

- When printing to cout, use `"\n"` instead of `endl`.
  - Why? While both add in a new line, `endl` also flushes the output buffer (output writes to hard drive) every time it is called, while `"\n"` does not.

```
for (char index = 'A'; index <= 'Z'; ++index) {
  cout << index << endl;
}
```

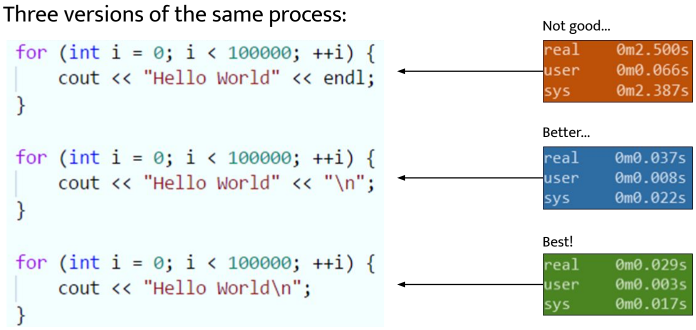
output buffer flushes 26 times

```
for (char index = 'A'; index <= 'Z'; ++index) {
  cout << index << "\n";
}
```

output buffer flushes only once at the very end

I/O Tips

- Three versions of the same process:



```
for (int i = 0; i < 100000; ++i) {
    cout << "Hello World" << endl;
}
```

Not good...

```
real    0m2.500s
user    0m0.066s
sys     0m2.387s
```

```
for (int i = 0; i < 100000; ++i) {
    cout << "Hello World" << "\n";
}
```

Better...

```
real    0m0.037s
user    0m0.008s
sys     0m0.022s
```

```
for (int i = 0; i < 100000; ++i) {
    cout << "Hello World\n";
}
```

Best!

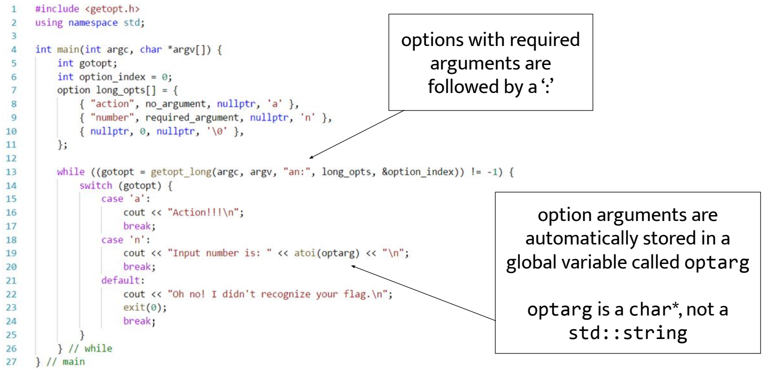
```
real    0m0.029s
user    0m0.003s
sys     0m0.017s
```

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



```
#include <getopt.h>
using namespace std;

int main(int argc, char *argv[]) {
    int gotopt;
    int option_index = 0;
    option long_opts[] = {
        { "action", no_argument, nullptr, 'a' },
        { "number", required_argument, nullptr, 'n' },
        { nullptr, 0, nullptr, '\0' },
    };

    while ((gotopt = getopt_long(argc, argv, "an:", long_opts, &option_index)) != -1) {
        switch (gotopt) {
            case 'a':
                cout << "Action!!!\n";
                break;
            case 'n':
                cout << "Input number is: " << atoi(optarg) << "\n";
                break;
            default:
                cout << "Oh no! I didn't recognize your flag.\n";
                exit(0);
                break;
        }
    } // while
} // 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

- Define a collection of valid operations and their behaviors on stored data.
- This *interface* to the data (the operations) is called an abstract data type.
- The *implementation* of the interface is called a data structure.
- When using an interface, we (mostly) don't have to worry about the implementation - if it were changed, our code would not have to change, except possibly to improve time/space usage.
- We want you to understand:
  - when to use a specific ADT
  - how your choice of ADT affects time and space usage
  - how and when to use a more time- and space-efficient ADT, 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

- Suppose we have a program that can run using either a stack or a queue to manage some data. In order to reuse code more effectively, it would be better to have one ADT manage both, and use if statements to decide which behaviors to apply.

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

- By only using `push_back` and `pop_back` to simulate a stack, and `push_back` and `pop_front` to simulate a queue, we have all the functionality we need!
- 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.
- Vectors are a good candidate for the implementation of a stack (the top of the stack is equivalent to the back of the vector) as they cover all necessary operations and are very fast.
- Unless a fast `push_front(value)` or `pop_front()` is required, vectors should be used for data that requires random access to its elements (if they are, use a deque). 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

- Vectors have a data pointer, a size, and a capacity.
- The **capacity** is how much *room* they have.
- The **size** is how many elements they actually contain.
- `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

- 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 *size of the bucket and add water to fill it.*

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

- 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.
- Because we are treating vectors as an ADT for now, you don't have to worry about why vectors have a separate capacity and size - it's just part of the specification for the (incredibly useful) ADT.
- Capacity increases (triggered when size needs to exceed capacity, or upon reserve/resize use) are ***expensive*** and should be minimized.
- Because having a consistently larger capacity than size wastes memory, but repeatedly increasing the size beyond the capacity wastes time, you need to be aware of this when coding. 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

However, as explained before, since we know the size of both the outer and inner vector in advance, **we should either use `resize` and `reserve`, or initialize the size in the constructor call** - so let's initialize the temp vector to be of size 10 containing 0, and reserve `my_vec` to 10 so that size never needs to exceed capacity, triggering an expensive increase in capacity.

```
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 on 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

- In lab, we want to have you attempt some practice problems, so that it's not just us talking at you. There'll be more of these in future labs.
- Here are some situations - 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.

Practice

- In lab, we want to have you attempt some practice problems, so that it's not just us talking at you. There'll be more of these in future labs.
- Here are some situations - 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

Project 1 Tips

Project 1 Tips

- Write modular code (i.e. separate your code into functions)
- Think hard about data structures
  - This project is strict on memory!
- Look at Ed posts
- Submit ASAP
  - Students often take 10+ submits to get their desired score
  - Submitting early has its benefits - for example, the Autograder will tell you if your test cases catch a bug in your own program (and what the correct output should be so that you can debug)
- Pass anything other than basic types (int, char, bool) by reference

Project 1 Tips

- Break up the work into parts, so that it's not so overwhelming
  - Makefile
  - Getopt
  - Storing Input
  - Routing
  - Backtracking
  - First Output Mode
  - Second Output Mode

Steps for Solving Coding Problems

1. **Read carefully and pay attention to problem specifics**
   - Typically every detail of the problem description is needed to come up with the optimal solution
2. **Come up with a good example and use it to inform your algorithm design**
   - Make sure this example is fairly large and generic
   - Walk through your algorithm and run it on this example BEFORE you start coding
3. **Code your algorithm**
   - Make sure to write neatly with clear indentations and appropriate variable names
   - Modularize your code where appropriate
4. **Test your solution with multiple small examples**
   - Start with a generic example and then test edge cases
   - Make sure to be thorough – go line by line and actually test your code, not just your algorithm!
5. **Test your solution with large examples and look at time and memory use**

Handwritten Problem

Handwritten Problem

- Completion of this written problem is worth 5 points

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

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

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

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

Maps and Sets (Very Optional)

Maps and Sets

- A **map** is an ADT that gives us **associative lookup** - we can recall some information (a value) related to a key, and freely insert and remove keys. For example, we might look up the temperature (value) of a place (the key - perhaps a latitude and a longitude) on Earth.
- It also has a form where values are ignored, called a **set** - but both maps and sets can be used for checking membership (with find).

Operation	Behavior	Operation	Behavior
insert(key, value)	insert key and value pair into map	operator[](key)	return the key's value from map
erase(key)	erase key and its value from map	find(key)	find the position of key in map
size()	return # of keys in map	empty()	return whether size() is 0

Vectors as Maps and Sets

- Suppose the keys of the map or set do not change.
- We could set up a sorted vector that holds the keys in sorted order, as well as their associated value if we have a map.
- We can then implement lookup with binary search!
- This would have much better performance than a tree based map.
- If the keys are sequential integers or can be easily mapped to sequential integers, we don't even have to sort the vector!
- The fully general map or set ADT only comes in useful when we need fast insert and erase at **any time** during the running of a program - as this is an important weakness of vectors.

Practice

- What ADT(s) will come in useful?
  - 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?
  - 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

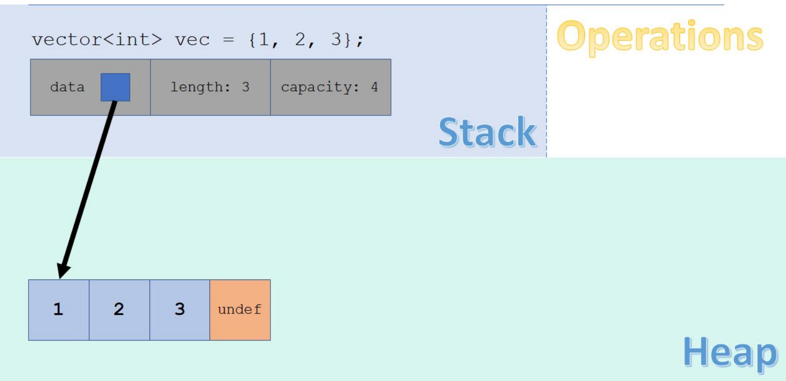
Again, vectors, including multidimensional vectors, can be used to perform associative lookup in certain cases, such as the above. This is particularly useful when there is very rarely or never a need to support insertion or deletion of elements (only lookup is needed after setting up the set/map).

Vectors

Resize and Reserve

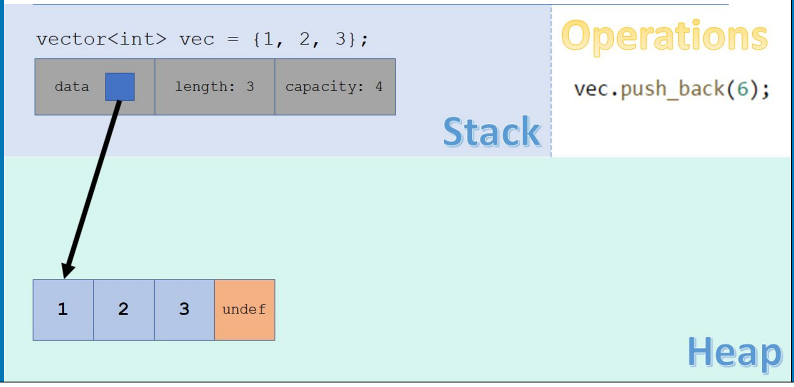
- To recap:
  - Resizing changes *the number of elements* in a vector.
  - Reserving changes *how many elements* a vector can hold.
- Now let's look at how vectors allocate memory under the hood.

Resize and Reserve



```
vector<int> vec = {1, 2, 3};
```

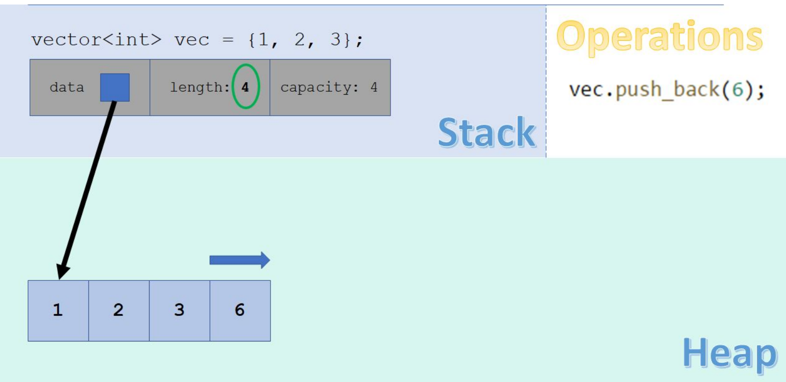
Resize and Reserve



```
vector<int> vec = {1, 2, 3};
```

```
vec.push_back(6);
```

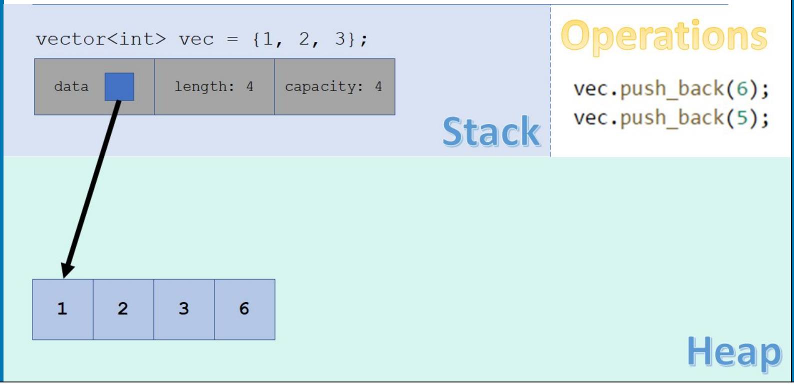
Resize and Reserve



```
vector<int> vec = {1, 2, 3};
```

```
vec.push_back(6);
```

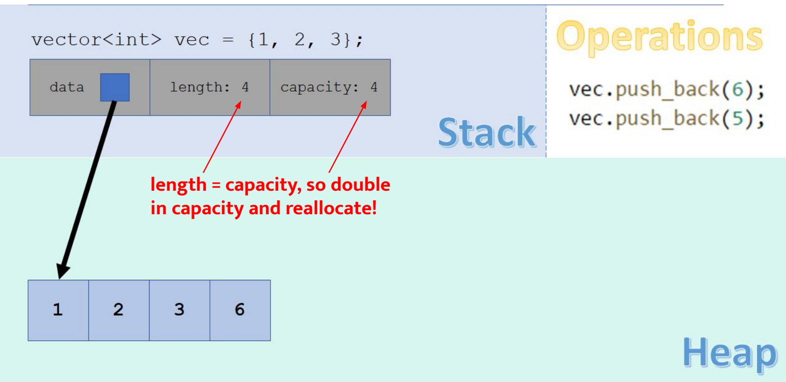
Resize and Reserve



```
vector<int> vec = {1, 2, 3};
```

```
vec.push_back(6);
vec.push_back(5);
```

Resize and Reserve



```
vector<int> vec = {1, 2, 3};
```

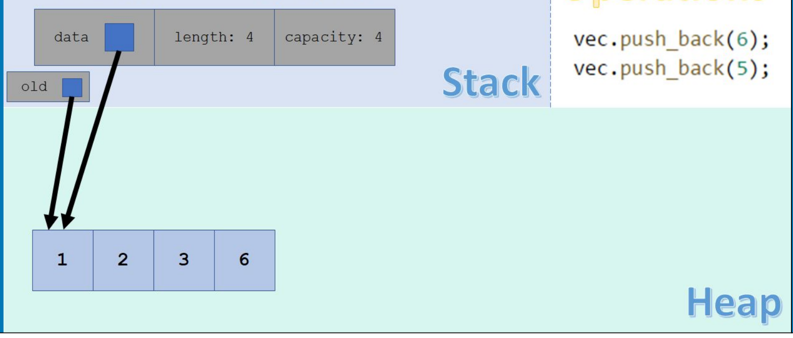
```
vec.push_back(6);
vec.push_back(5);
```

Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

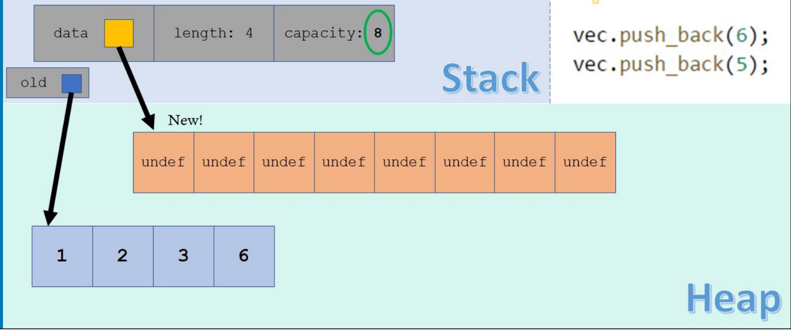


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

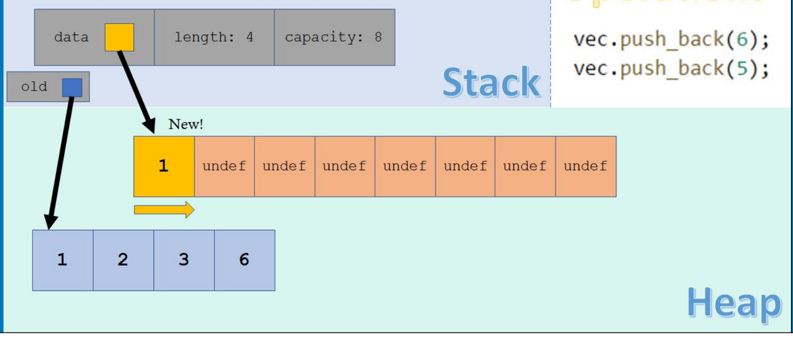


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

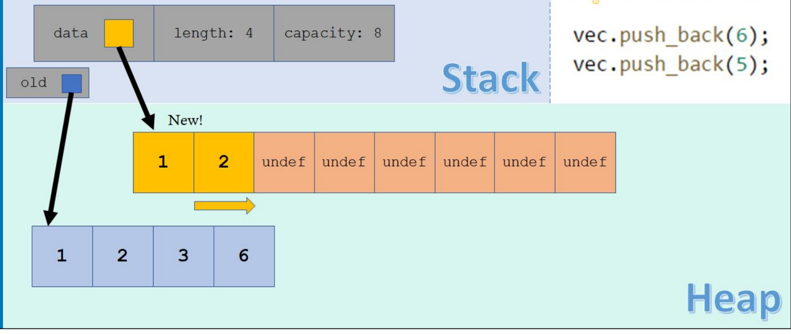


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

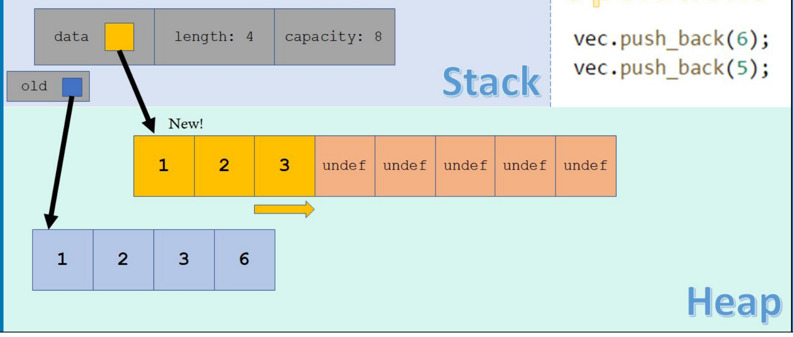


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

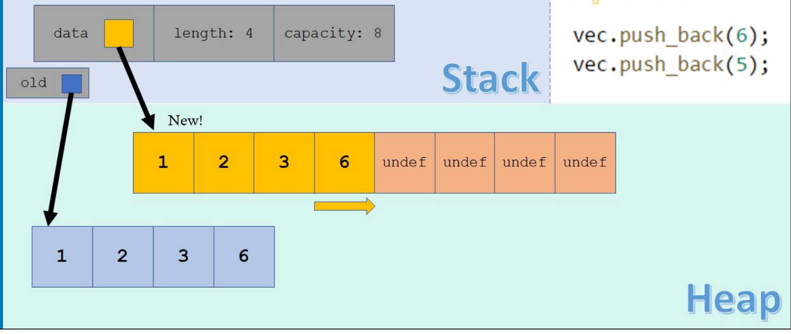


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

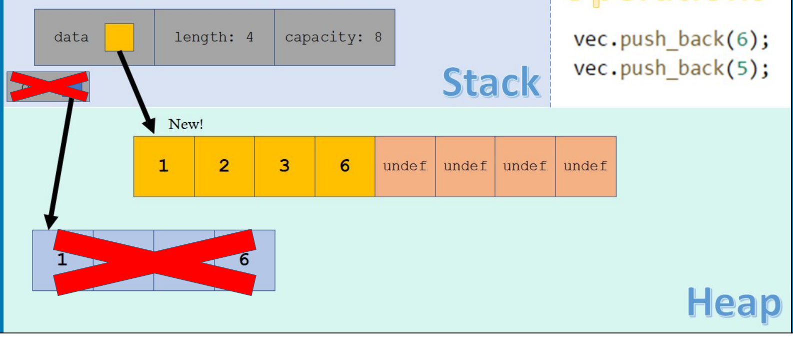


Resize and Reserve

```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```

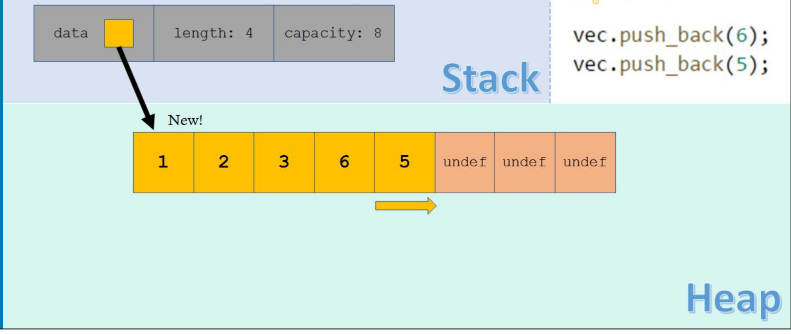


Resize and Reserve

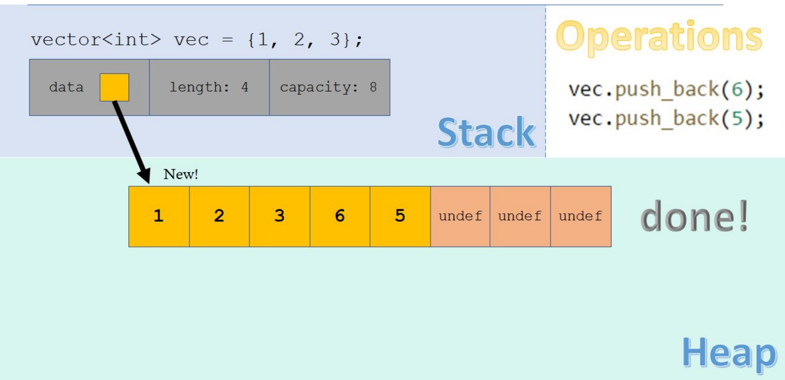
```
vector<int> vec = {1, 2, 3};
```

Operations

```
vec.push_back(6);
vec.push_back(5);
```



Resize and Reserve

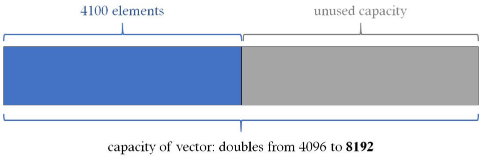


```
vector<int> vec = {1, 2, 3};
```

```
vec.push_back(6);
vec.push_back(5);
```

Resize and Reserve

- Why is this important to know? Suppose you wanted to store 4100 elements in a vector, and you only call `.push_back()` without resizing or reserving. What happens?
  - The vector doubles capacity from 1 to 2 to 4 to 8 to 16 to 32 to 64 to 128 to 256 to 512 to 1024 to 2048 to 4096…
  - When you insert the 4097th element, the vector's capacity doubles to **8192**!

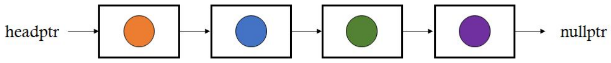


Linked Lists

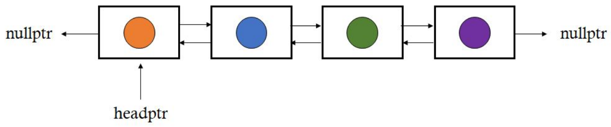
this material will be covered in depth in a later lab, but feel free to read over these additional slides during your free time

Linked Lists

- Singly-linked list: each node points to the next node



- Doubly-linked list: each node points to both previous and next node

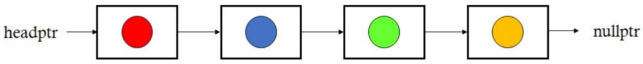


Linked Lists: The "Runner" Technique

- The **"runner" technique** is a technique that you can use if you are ever asked a linked list question during an interview.
- Iterate through the list with two (or more) pointers simultaneously, with one either a fixed distance from the other, or one that moves faster than the other (slow and fast).

Linked Lists: The "Runner" Technique

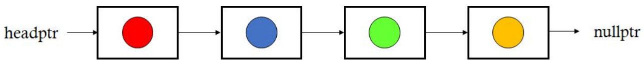
- **Example 1:** Given an integer *k*, find the k^{th} to last element in a singly-linked list. You do not know the length of the linked list.



- How can you solve the problem?

Linked Lists: The "Runner" Technique

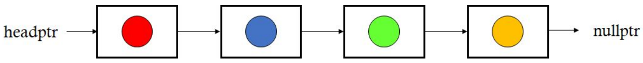
- **Example 1:** Given an integer *k*, find the k^{th} to last element in a singly-linked list. You do not know the length of the linked list.



- How can you solve the problem? *Use the "runner" technique!*
  - The k^{th} to last element is *k* from the end of the list.
  - We can take two pointers that are a distance of *k* nodes apart, `fast` and `slow`. We start from the beginning and increment `fast` until it reaches the end of the list. Since `slow` is *k* nodes behind `fast`, `slow` must point to the k^{th} to last element!
  - O(*n*) time and O(1) space

Linked Lists: The "Runner" Technique

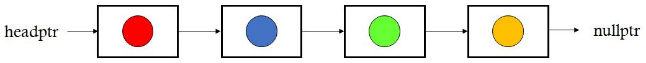
- **Example 2:** Given a singly-linked list, devise an algorithm that returns the value of the middle node. If there are two middle nodes, return the value of the second middle node.



- How can you solve the problem?

Linked Lists: The “Runner” Technique

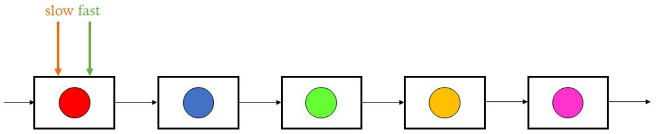
- **Example 2:** Given a singly-linked list, devise an algorithm that returns the value of the middle node. If there are two middle nodes, return the value of the second middle node.



- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!

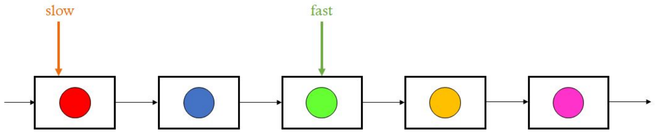
Linked Lists: The “Runner” Technique

- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



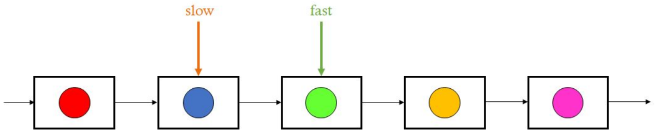
Linked Lists: The “Runner” Technique

- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



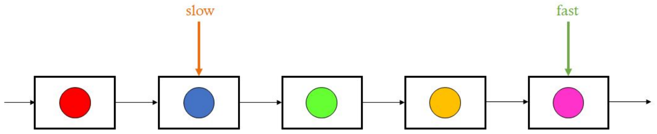
Linked Lists: The “Runner” Technique

- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



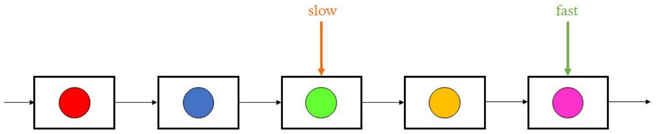
Linked Lists: The “Runner” Technique

- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



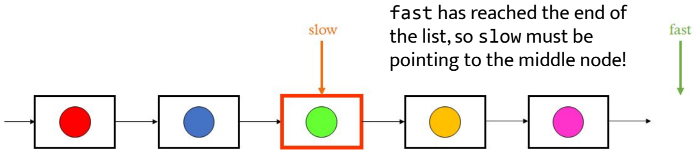
Linked Lists: The “Runner” Technique

- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



Linked Lists: The “Runner” Technique

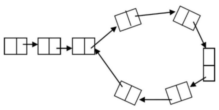
- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



`fast` has reached the end of the list, so `slow` must be pointing to the middle node!

Linked Lists: The “Runner” Technique

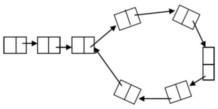
- **Example 3:** Given a singly-linked list, determine if the list contains a cycle (or loop) - where the a node’s next points to a previous node in the list.



- How can you solve the problem?

Linked Lists: The “Runner” Technique

- **Example 3:** Given a singly-linked list, determine if the list contains a cycle (or loop) - where the a node’s next points to a previous node in the list.



- How can you solve the problem? *Use the “runner” technique!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - If there is a cycle, the two pointers will momentarily equal each other; otherwise, `fast` will reach the end (an additional exercise: prove that this will always work!).