

EECS 281: Lab 1

Week of Sep 10th, 2018

Announcements

- Lab 1 is due Friday 9/21
- If you couldn't make the How To session this past Sunday, check out the recording on Canvas
- Project 1 released on 9/11, due on 9/27

Agenda

- 281 Success Tips
 - Debugging and Testing Tips
- Makefile/Valgrind Demos
- C++ Input and Output
- Parsing Command Line Options Using Getopt
- Vectors
- Handwritten Problem

Lab Attendance and Grading Policy

- Lab attendance **IS** part of your grade
 - Each week, there is a handwritten problem worth 5 points - to get credit, you must do the problem on paper while physically in lab, and turn in your solution to the lab instructor
 - A fair effort (right or wrong solution) will earn you the full 5 points, but a lab instructor may choose to not award you full credit if it is clear that you did not try
 - First time, the lab instructor will still give you full credit, but will also give you a warning via email, informing you that you will lose points the next time you show insufficient effort
 - Second time and beyond, you may receive partial or no credit at the lab instructor's discretion
 - Honor code: do not turn in a solution on behalf of a student who is not physically present in lab, or have another student do so for you - BOTH of you are violating the honor code if this happens
- You should plan on attending a lab each week
 - We prefer that you attend the section you are officially registered for, but if space permits, you may attend a different lab section due to instructor preference or scheduling
 - If an extreme situation makes attendance impossible for all sections of the week, then you should email eeecs281admin@umich.edu to work something out

Syllabus Must-Knows

- Exam Dates: Wednesday, 10/24, 6:30pm to 8pm for the midterm; Monday, 12/17, 8am to 10am (it sucks, but we don't have a say) for the final
- Minimum Competency Rules: must have **ALL** of 55% overall on projects, 50% overall on exams (curved score), 60% overall in the course - if you meet all three requirements, you will be guaranteed at least a C
- Honor Code:
 - Projects: should not be any different from how it was in 280 and 183/101
 - Exams: basically the same as it is for any other course on campus
 - Labs: you may work with a partner, both partners may submit identical code and answers, but otherwise know your limits
 - If you choose to do them dishonestly, you will likely have a hard time with the exams

281 Success Tips

- **Start projects early** - often easier said than done, and too many people will say “I can wait until tomorrow” and end up putting off Project 1 until it is fatal
- Come to office hours often (and early) - if you wait until close to when the project is due, you will face very long wait times, and might not get the help you need
- Get used to Googling things (C++ standard library functions, compiler errors, Linux command line, git commands etc.)
- Think carefully about the design of your code BEFORE you begin implementation - you can go over your design with us in office hours before you start coding
- Lab 1 is meant to help you get comfortable with Make, redirecting input, the autograder, late days, etc.
- Read the “Optimization Tips” doc in the Resources folder, and “The STL and You” doc in the Project 1 folder

Debugging Tips

- Take a deep breath!
- Trace through your algorithm by hand - is it *really* doing what you think it's supposed to?
 - Actually trace through it with a test case - don't just read your code and say "Oh I'm sure this part works"
- Print statements are helpful! Just remember to take them out before submitting.
- Read the spec again - are you understanding it correctly?
 - Specifically look at assumptions your program should or should not be making.
- Write more test cases
- Use an IDE - it takes time to learn, but can really save you time in the long run
- Come to office hours once you've tried debugging thoroughly.

Testing Tips

Note: these are to help write test cases to find bugs in your own code

- Consider off-by-one errors and edge cases
- Test your branches and their conditions - write tests that hit each part of your code

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

Accessing your CAEN account with SSH

- Open up Unix/Linux/Mac terminal or GitBash
- Run the following command:
 - `ssh <username>@login.engin.umich.edu`
- Sign in with your umich account password
- Ensure that you are always running the right version of gcc (**by doing this once only!**):
`echo "module load gcc/6.2.0" >> ~/.bash_profile`

Workflow advice

- Don't use Mfile to transfer your files to the CAEN server!
- You can sign in using your U of M info and create a git repo at <https://gitlab.eecs.umich.edu>
 - Make sure to 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: <https://filezilla-project.org/> or WinSCP: <https://winscp.net/eng/index.php>
- 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

Compiling with g++

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



Source files



Executable name

Problems:

- Long to re-type 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 a lot easier
- Previous compilation command: `g++ main.cpp file1.cpp file2.cpp -o main`
- Command using Make: `make all`

Makefile to-dos

1. Set the executable name to the name given in the project spec
2. Set the project file name to the name of the file in your program that has a main function
3. Set up any custom file dependencies

Makefile basic commands

- `make` (or `make all`)
 - Recompile 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)

Submitting to the Autograder

- `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 it doesn't compile.
- `make fullsubmit`
 - Creates a tarball in the current directory that does include test cases.
 - Will count as a submit if 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.

Makefile testing 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 specific test file

Testing with Make:

- Write your test driver programs in `test*.cpp` files
- Files that you want to include in your final project submission cannot match this pattern

Valgrind

Valgrind

- 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 write of size..”)
 - Memory leaks
 - (Although you shouldn’t need to manage much memory manually in 281)
 - Be aware that all of the STL containers use dynamic memory
 - the C function `exit(status)` will stop the program without calling any container destructors - leaving your program with a bunch of leaked memory.

Example

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 #(compiles code with -g3)  
valgrind ./<executable_name>
```

Example

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: }
```

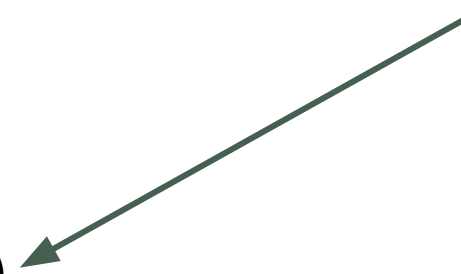
Valgrind output:

```
==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
```


Running Valgrind in CAEN:

```
make debug #(compiles code with -g3)  
valgrind ./<executable_name>
```

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 we asked for the 4th!



Use Valgrind!

- Always `valgrind` code before submitting to the autograder
 - If `valgrind` produces errors, you will lose ALL of the points awarded for no memory leaks
 - If you have undefined behavior, it may cause erroneous output
- Once you know what lines are causing problems, you can examine further with `gdb`, IDE debugger, etc.

gdb debugger

Overview

- Text-based debugging tool
- Useful for solving segmentation faults (i.e. program received signal SIGSEGV) and memory issues (e.g. index is out of bounds)
- Note: the debuggers built into IDEs are a lot easier to use so they are recommended!

Usage

- 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 seg. fault occurred
- (b)ack(t)race: prints the 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 <line_num> : list source code near <line_num>
- kill: terminate the executable
- (q)uit: quit gdb

Demo: Makefile + Valgrind

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 & write permission

Redirecting input from files

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

Indicates that next entry on command line will be the file name that you want cin to be associated
 - `<` and `input.txt` ^{with} are **not** command line args; don't appear in `char * argv[]`
 - You can now use `getline(cin, var)` or `cin >> var` to read in the input file
 - DO NOT open files if you are using input/output redirection.
- Directions for redirecting input in Xcode and Visual studio can be found in the Resources folder on canvas (in the corresponding IDE guides)

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 now 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) {
 // execute only if new_value read properly
}
```
- ```
while (getline(cin, new_line)) {  
    // execute only if new_line read properly  
}
```

Read char by char

```
int main(int argc, char* argv[])
{
    char c;
    while (cin >> c) {
        cout << c; // note: does not preserve whitespace
    }
}
```

Read line by line

```
int main(int argc, char* argv[])
{
    string s;
    while (getline(cin, s)) {
        cout << s << endl; // note: does preserve whitespace
    }
}
```


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 `std::ios_base::sync_with_stdio(false);`
 - This saves time by specifying that input and output from c++ and c streams do not need to be synced

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`
 - Note that these are all equivalent: your program should behave the same for them all!!
- `getopt_long` takes the work out of accounting for all these different possibilities

Using getopt_long

```
#include <getopt.h>
```

```
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
```

Options with required arguments
are followed by a ':'



Option arguments are
automatically stored in a global
variable called 'optarg'



note: optarg is a char*, not a
std::string.

Vectors - how do they work?

```
vec.resize(new_size)  
vec.reserve(new_capacity)
```

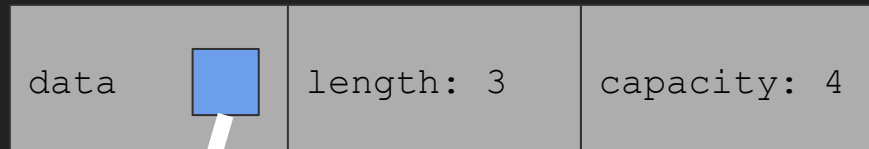
What's the difference?

Resize changes **size**.

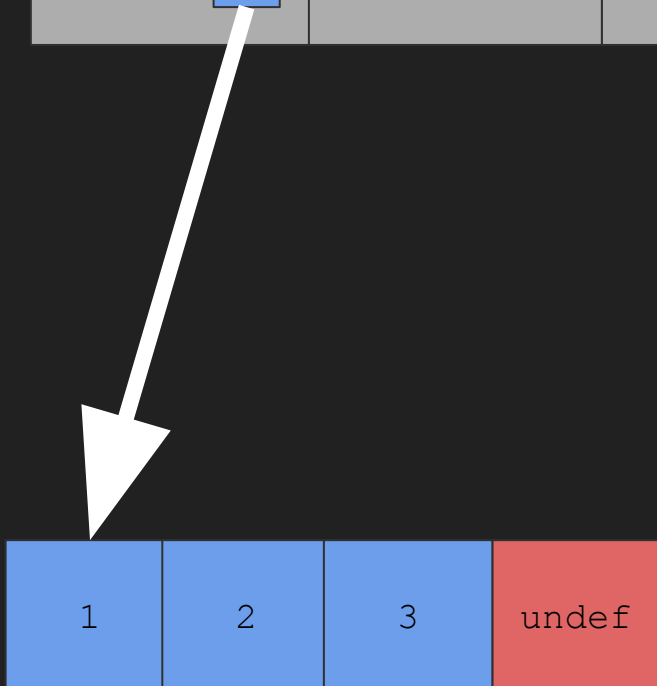
Reserve changes **capacity**.

Vectors - how do they work?

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



Stack



Heap

Vectors - how do they work?

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



Stack

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



Heap

Vectors - how do they work?

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



Stack

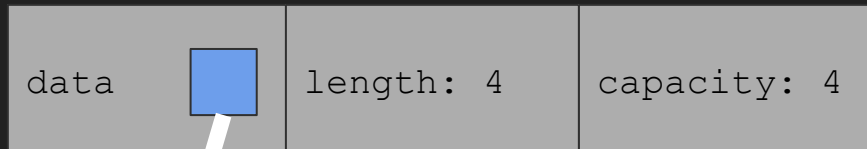
```
vec.push_back(6) // done
```



Heap

Vectors - how do they work?

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



Stack

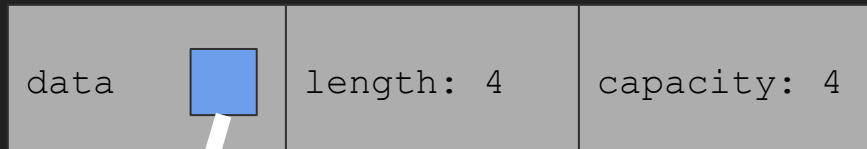
```
vec.push_back(6) // done  
vec.push_back(5)
```



Heap

Vectors - how do they work?

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



Stack

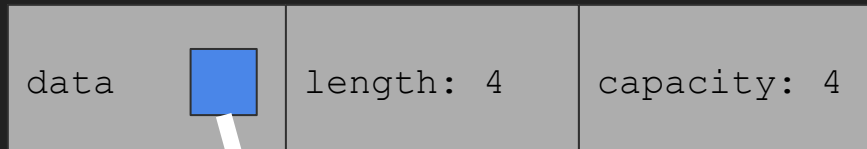
```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```



Heap

Vectors - how do they work?

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



Stack



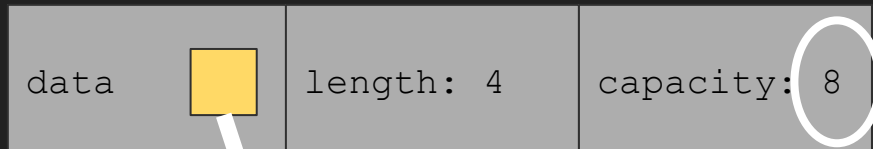
```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```



Heap

Vectors - how do they work?

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

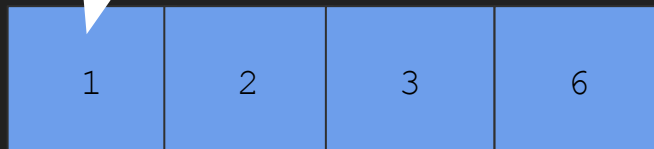
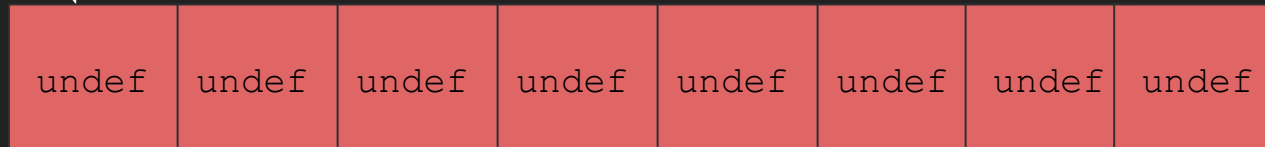


Stack



```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```

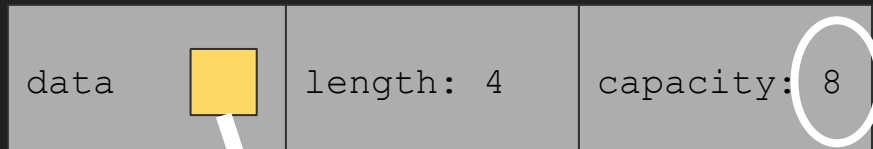
New



Heap

Vectors - how do they work?

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

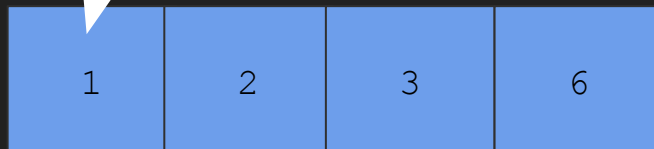
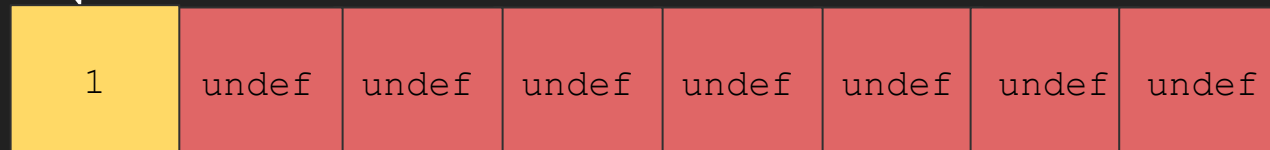


Stack



```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```

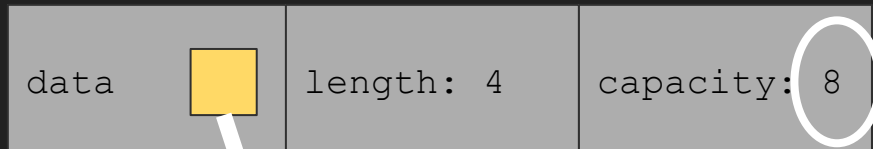
New



Heap

Vectors - how do they work?

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

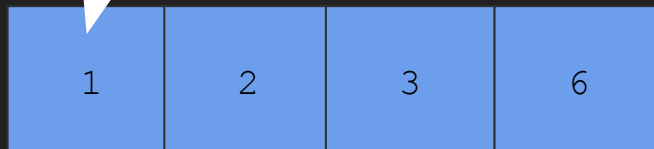
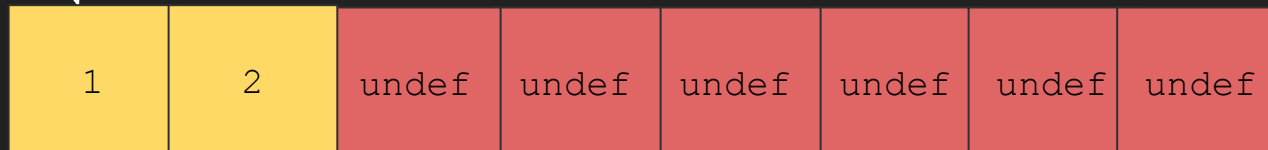


Stack



```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```

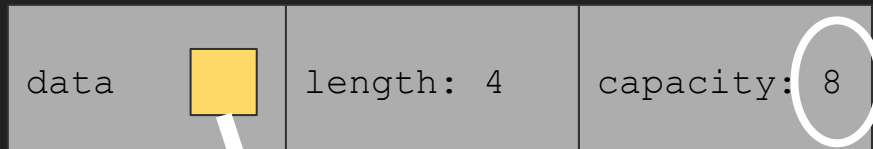
New



Heap

Vectors - how do they work?

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

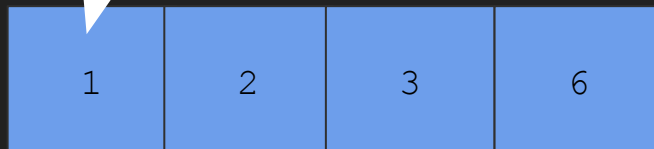
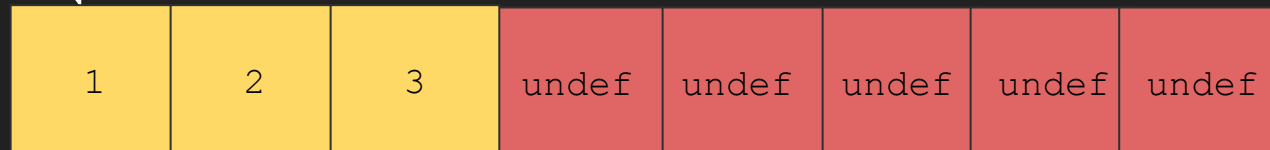


Stack



```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```

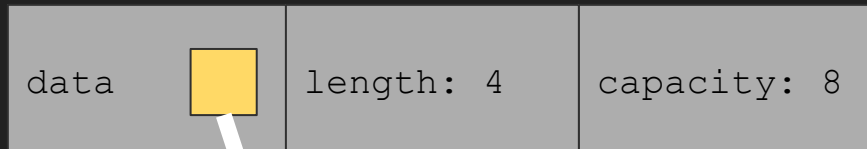
New



Heap

Vectors - how do they work?

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



Stack



```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```

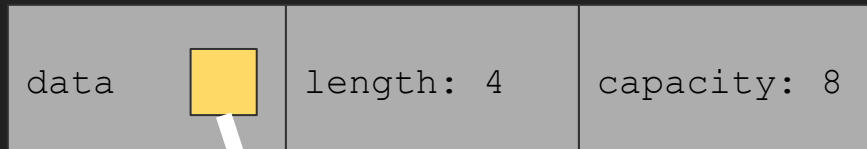
New



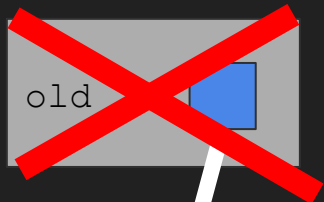
Heap

Vectors - how do they work?

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

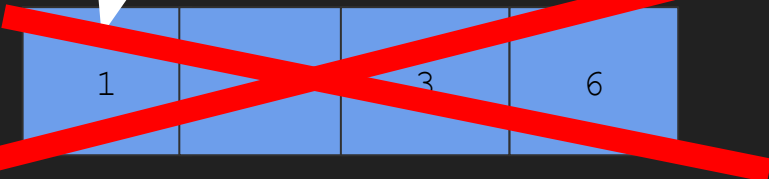


Stack



```
vec.push_back(6) // done  
vec.push_back(5) // ... working
```

New



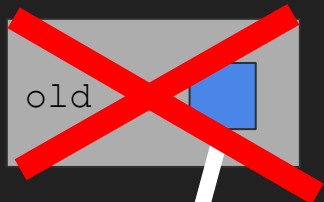
Heap

Vectors - how do they work?

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

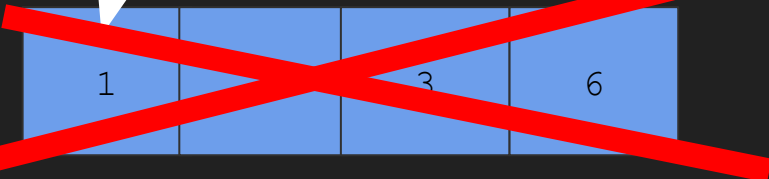
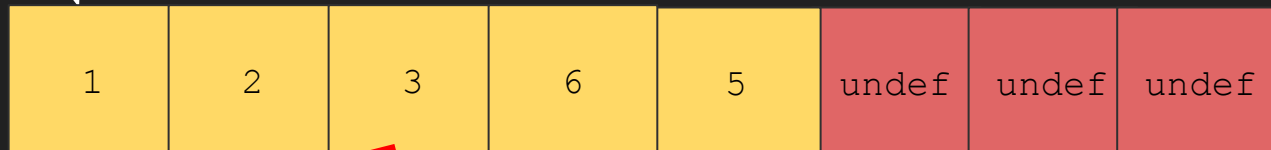


Stack



```
vec.push_back(6) // done  
vec.push_back(5) // done!
```

New



Heap

Vectors - how do they work?

Vectors have a data pointer, a “size” and a “capacity”.

Capacity is how much *room* they have

Size is how many elements they actually contain.

```
vec.resize(new_size)
```

changes **size** (and increases capacity if needed)

```
vec.push_back(x) // adds x AFTER newly created items
```

```
vec.reserve(new_capacity)
```

does NOT change size, but increases **capacity** if needed

```
vec.push_back(x) // adds x at same place as before
```

Vectors - how do they work?

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

```
my_vec[0] = 5; // what happens?
```

Vectors - how do they work?

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

```
my_vec[0] = 5; // what happens? undefined behavior  
               it probably won't crash.  
               this is bad because the vector is now in a probably invalid state.
```

```
my_vec.empty(); // true, it doesn't contain anything
```

Handwritten Problem

Pull out of piece of paper and CLEARLY write your username at the top

```
struct Node{
    int value;
    Node* prev; // nullptr if no previous
    Node* next; // nullptr if no next
};

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

Write the implementation for isPalindrome. $O(1)$ space, $O(N)$ time.