Stacks and Queues

Amrita Kaur

July 3, 2023

Contributions made from previous CS106B Instructors

Announcements and Reminders

- Assignment 1 due Friday at 11:59pm
  - YEAH Hours recording on Canvas
- No class or LaIR tomorrow
- Midterm conflicts or OAE accommodations emailed to us by 7/10
- Anonymous weekly feedback survey for extra credit
  - Remember to fill out confirmation survey at the end
  - About 15 people who forgot
  - Due Wednesday by class time (1:30pm)

Assign 0 Takeaways

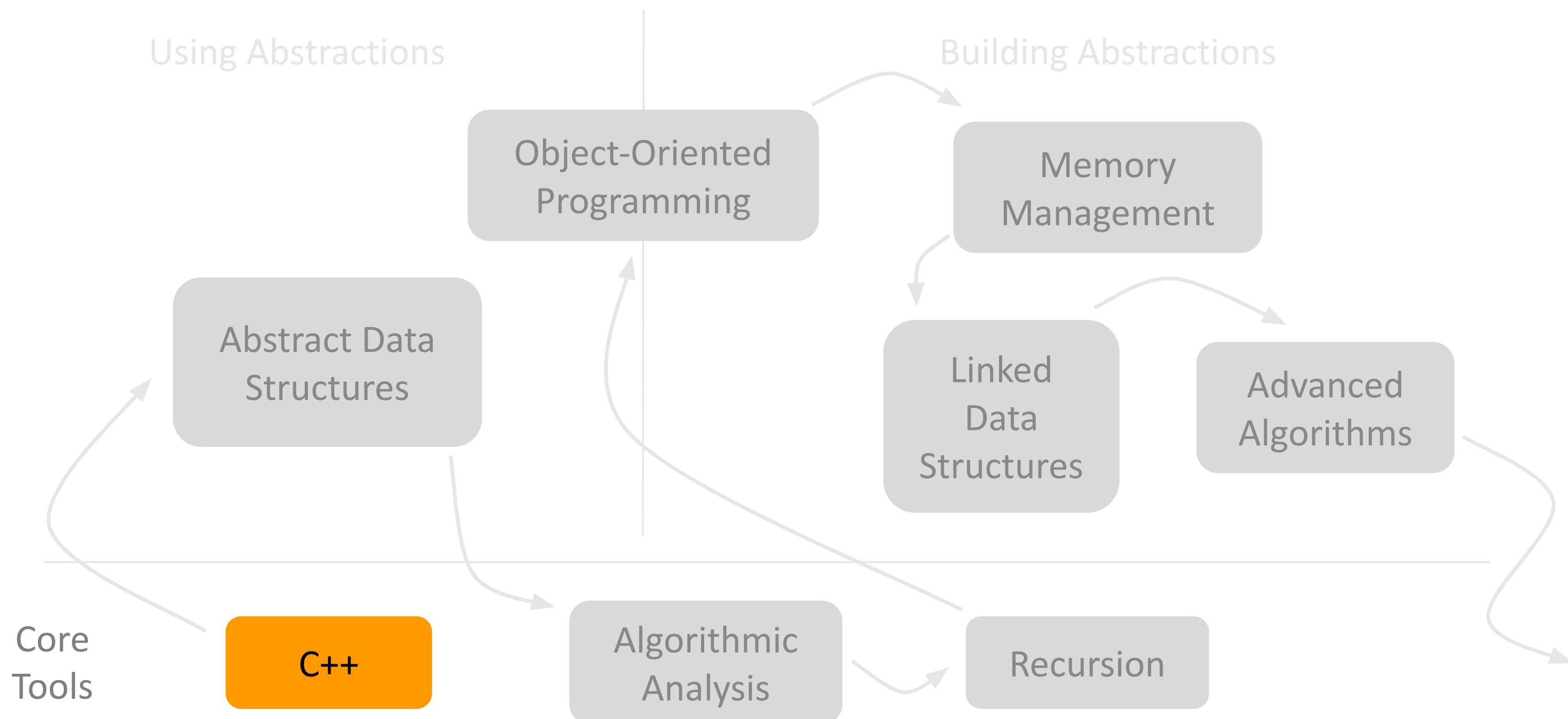
- What are you most looking forward to in the class?
  - “Learning some cool C++ skills”
  - “Problem solving is just so much fun!”
  - “Gaining the skills needed to build more complex programs for my personal projects”
  - “Learning about the various uses of computer science in real-life, especially with a focus on making life easier”
  - “Interacting with different people from all around the world”
  - “Feeling accomplished after a struggle.”

Assign 0 Takeaways

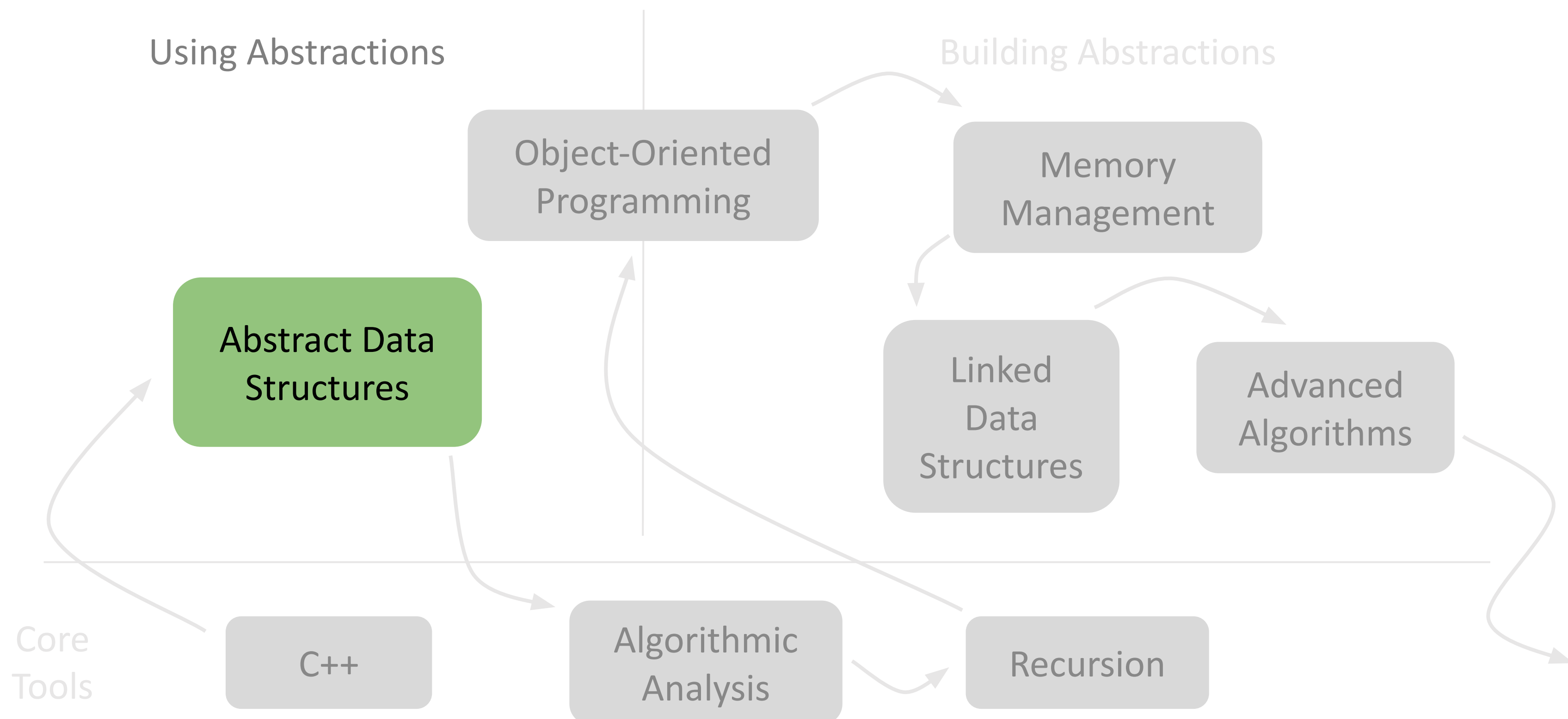
- Are you worried about anything in the class?
  - “Worried about my ability to write code on paper. I know we will have lots of practice but I'm someone who has to constantly google syntax even for languages I've been coding in for years.”
  - “Worried that my past experience is not adequate for this class and that C++ may be challenging language.”
  - “Worried about the class size making it more difficult to communicate with a section leader/lecturer.”
  - “Worried about the fast pace of the class.”
  - “Worried about not being good enough”

Review

Roadmap



Roadmap

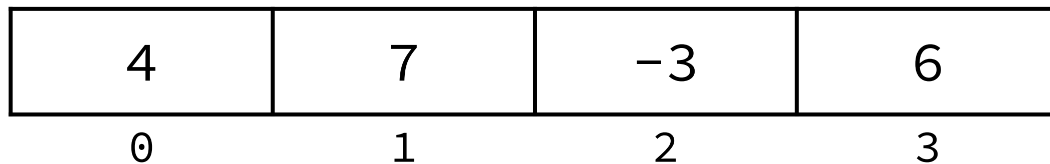


Abstract Data Type (ADTs)

- Also known as containers or data structures
- Allow programmers to store data in predictable, organized ways
- Can use without understanding the underlying implementation
- Transcends language boundaries/specific libraries

Vectors

- Ordered (indexed)
- 1-dimensional
- Can grow and shrink in size
- All elements must be of the same type



Grids

- Ordered (rows and cols are indexed)
- 2-dimensional
- Fixed dimensions
- All elements must be of the same type

	0	1	2
0	2	5	-1
1	10	11	3
2	19	-4	-2
3	4	6	2

Let’s Compare

Pass by value

- Callee gets a copy of a variable from the caller function
- Changes to that variable that occur in callee do not persist in caller

Pass by reference

- Callee gets a **reference** to a variable from the caller function
- Now, the callee can directly modify the original variable

Let's Trace Some Code (Pass by value)

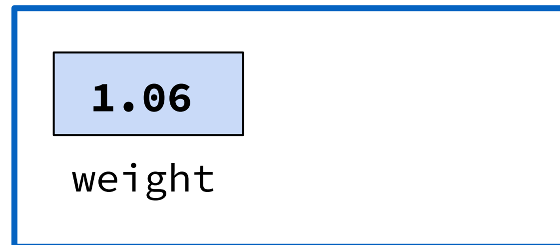
```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

Let’s Trace Some Code (Pass by value)

```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

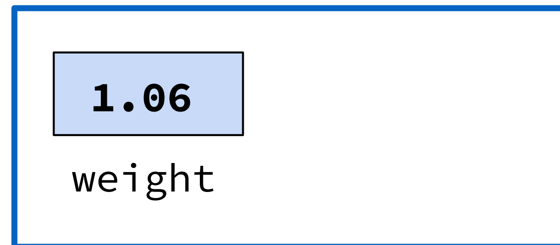
Let’s Trace Some Code (Pass by value)

```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



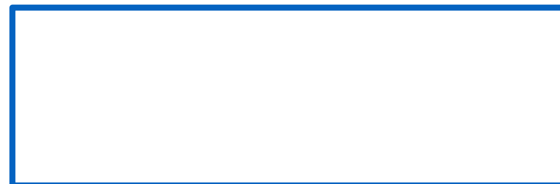
Let’s Trace Some Code (Pass by value)

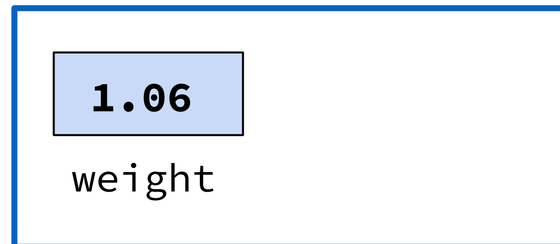
```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



Let's Trace Some Code (Pass by value)

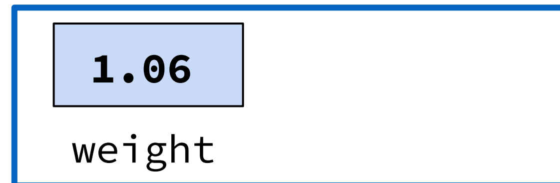
```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

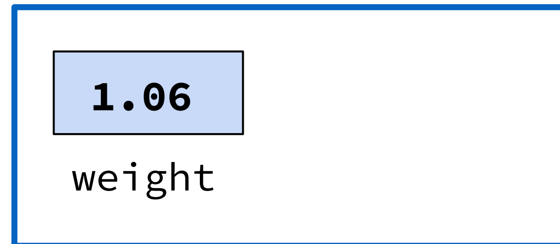




Let’s Trace Some Code (Pass by value)

```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```





Let's Trace Some Code (Pass by value)

```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

tripleWeight

3.18

weight

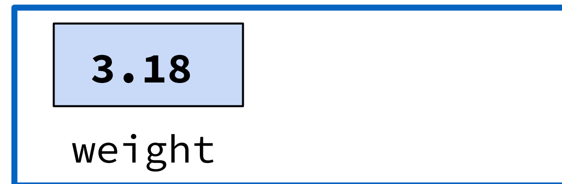
main

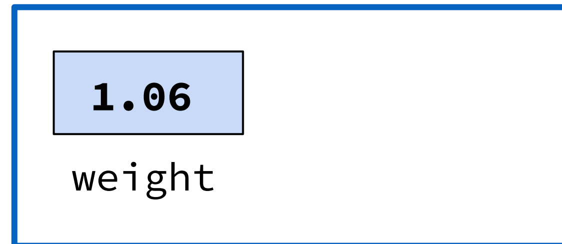
1.06

weight

Let’s Trace Some Code (Pass by value)

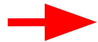
```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

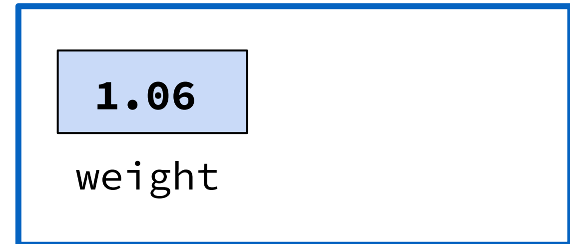




Let’s Trace Some Code (Pass by value)

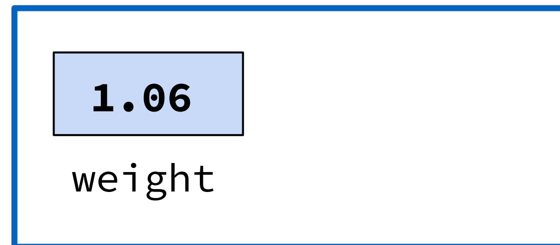
```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```





Let's Trace Some Code (Pass by value)

```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



Let’s Trace Some Code (Pass by reference)

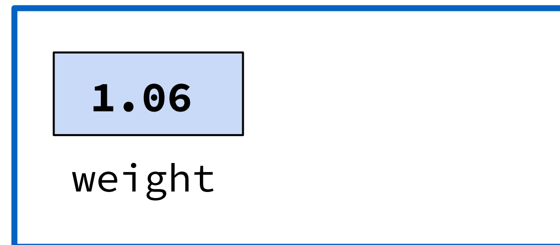
```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

Let's Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

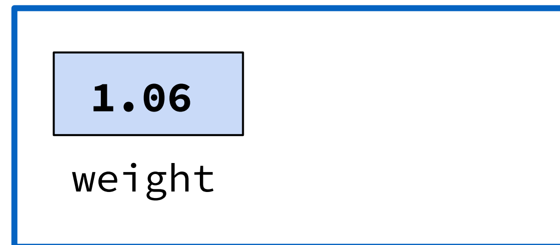
Let's Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



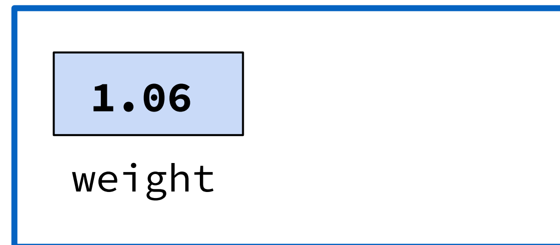
Let’s Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



Let's Trace Some Code (Pass by reference)

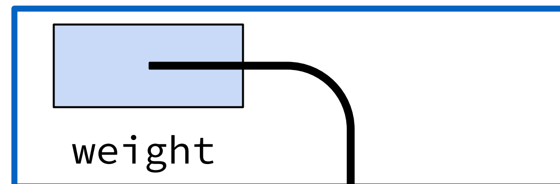
```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



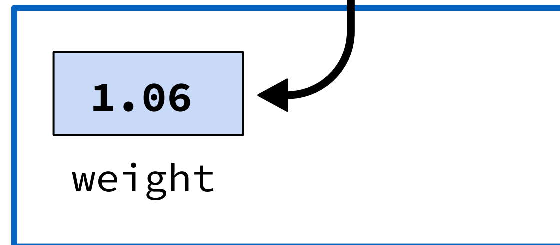
Let’s Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

tripleWeight



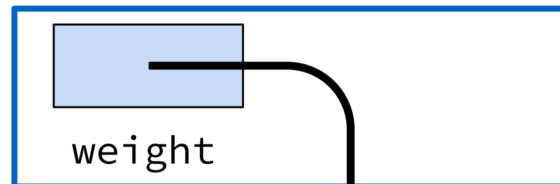
main



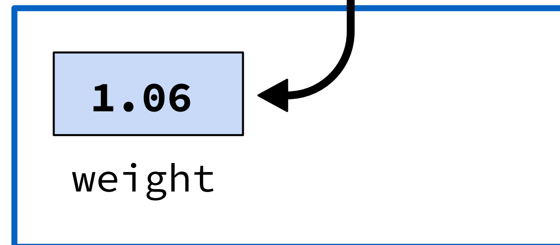
Let's Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

tripleWeight



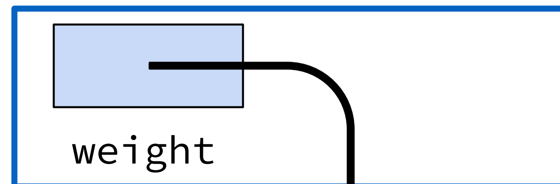
main



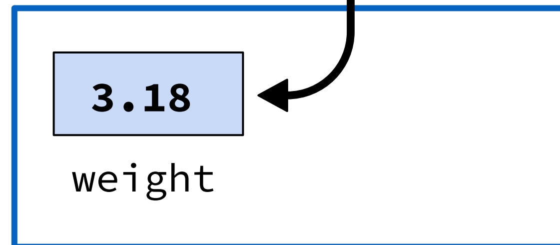
Let's Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

tripleWeight



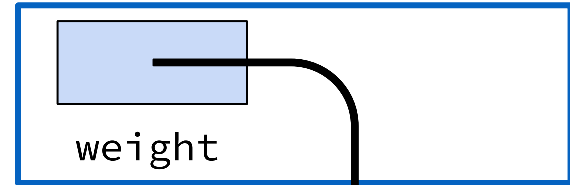
main



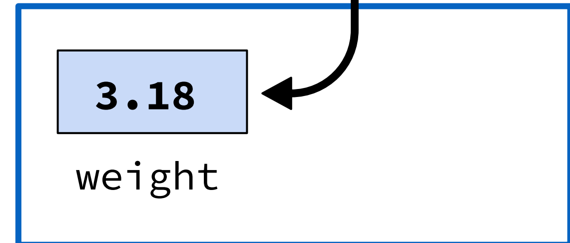
Let's Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```

tripleWeight

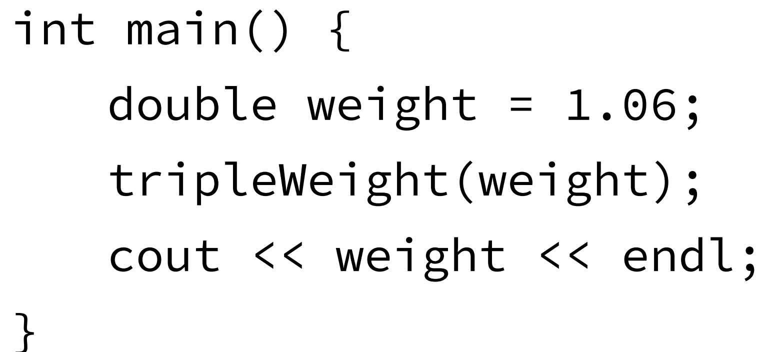


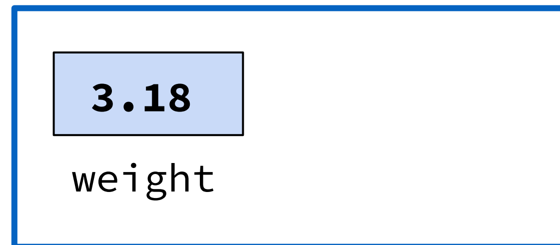
main



Let’s Trace Some Code (Pass by reference)

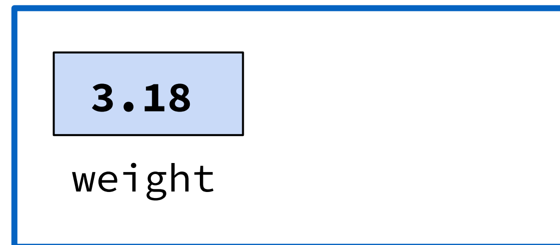
```
void tripleWeight(double& weight) {
    weight *= 3;
}
```





Let's Trace Some Code (Pass by reference)

```
void tripleWeight(double& weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



When Do We Pass by Reference?

Yes:

- When we want the callee function to edit our data
- To avoid making copies of large data structures
- When we need to return multiple values

No:

- Just because
  - Passing by reference is risky because another function can modify your data!
- When the data we're passing to the callee is small, and thus copying isn't expensive

What is the output of this code?

```
void mystery(int& b, int c, int& a) {
    a++;
    b--;
    c += a;
}

int main() {
    int a = 5;
    int b = 2;
    int c = 8;
    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

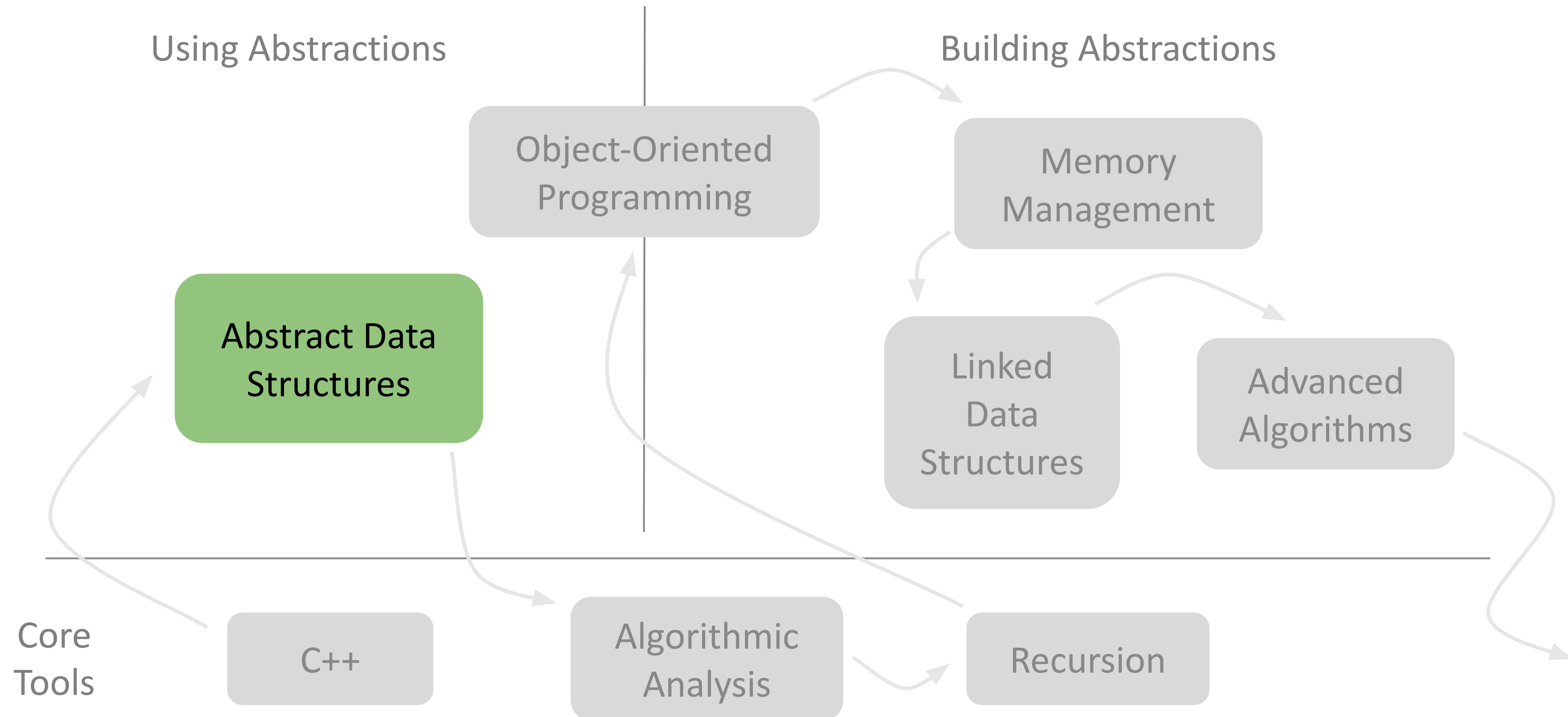
What is the output of this code?

```
void mystery(int& b, int c, int& a) {
    a++;
    b--;
    c += a;
}

int main() {
    int a = 5;
    int b = 2;
    int c = 8;
    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

Console:

```
5 3 7
```

Roadmap



Stacks

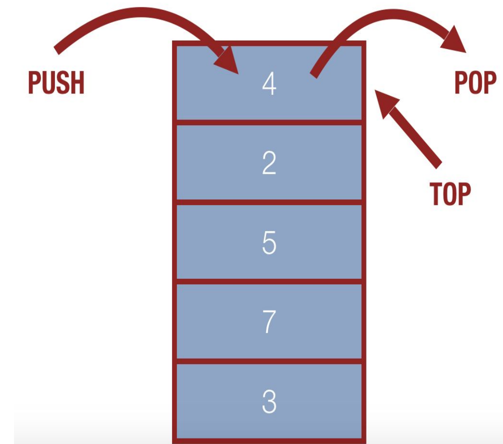
What is a Stack?

- An abstract data type (ADT)
  - Ordered collection of elements
- Stanford C++ library (here)
  - `#include “stack.h”`
- Modeled like an actual stack (of pancakes)
- Only the top element of the stack is accessible
- **L**ast **I**n, **F**irst **O**ut (LIFO)

The Stanford Stack Library

```
#include "stack.h"
```

- **`stack.push(value)`**: Add an element onto the top of the stack
- **`stack.pop()`**: Remove an element from the top of the stack and return it
- **`stack.peek()`**: Look at the element from the top of the stack, but don't remove it
- **`stack.isEmpty()`**: Returns a boolean value, true if the stack is empty, false if it has at least one element
  - Note: a runtime error occurs if a pop() or peek() operation is attempted on an empty stack
- **`stack.clear()`**: Removes all elements from the stack
- **`stack.size()`**: Returns the number of elements in the stack



For more information, check out the Stanford Stack class documentation!

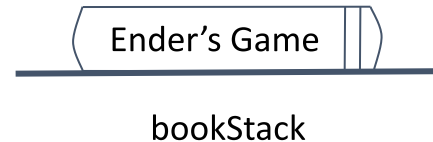
Stack Operations: Creation

```
Stack<string> bookStack;
```

bookStack

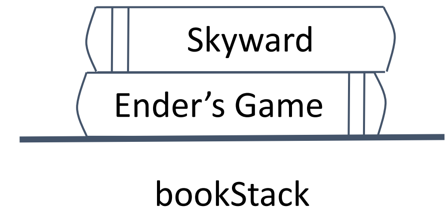
Stack Operations: Adding Elements

```
Stack<string> bookStack;
bookStack.push("Ender's Game");
```



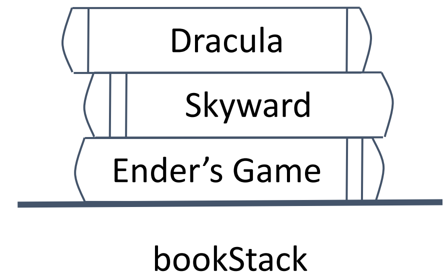
Stack Operations: Adding Elements

```
Stack<string> bookStack;
bookStack.push("Ender's Game");
bookStack.push("Skyward");
```



Stack Operations: Adding Elements

```
Stack<string> bookStack;
bookStack.push("Ender's Game");
bookStack.push("Skyward");
bookStack.push("Dracula");
```

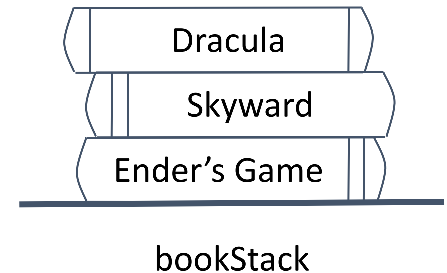


Stack Operations: Removing Elements

```
Stack<string> bookStack;
bookStack.push("Ender's Game");
bookStack.push("Skyward");
bookStack.push("Dracula");
cout << bookStack.pop() << endl;
```

Console:

```
Dracula
```

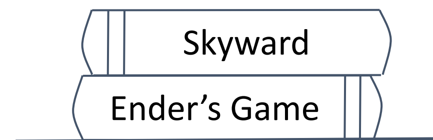


Stack Operations: Accessing Elements

```
Stack<string> bookStack;
bookStack.push("Ender's Game");
bookStack.push("Skyward");
bookStack.push("Dracula");
cout << bookStack.pop() << endl;
cout << bookStack.peek() << endl;
```

Console:

```
Dracula
Skyward
```



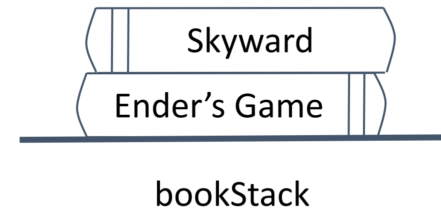
bookStack

Stack Operations: Removing Elements

```
Stack<string> bookStack;
bookStack.push("Ender's Game");
bookStack.push("Skyward");
bookStack.push("Dracula");
cout << bookStack.pop() << endl;
cout << bookStack.peek() << endl;
cout << bookStack.pop() << endl;
```

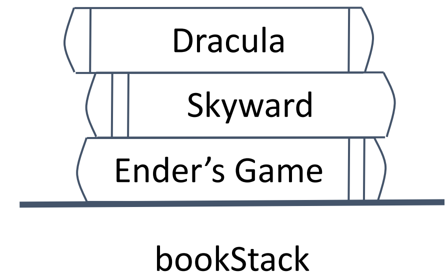
Console:

```
Dracula
Skyward
Skyward
```



Stack Operations: Creation with Elements

```
Stack<string> bookStack = {“Ender’s Game”,
                           “Skyward”, “Dracula”};
```

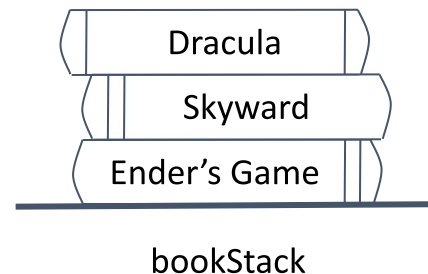


Stack Operations: Printing

```
Stack<string> bookStack = {“Ender’s Game”,
                                          “Skyward”, “Dracula”};
cout << bookStack << endl;
```

Console:

```
{“Ender’s Game”,
“Skyward”, “Dracula”}
```

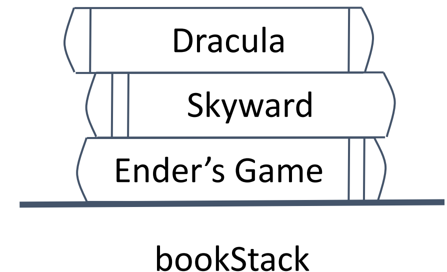


Stack Operations: Printing

```
Stack<string> bookStack = {"Ender's Game",
                           "Skyward", "Dracula"};
cout << bookStack << endl;
cout << bookStack << endl;
```

Console:

```
{"Ender's Game",
"Skyward", "Dracula"}
{"Ender's Game",
"Skyward", "Dracula"}
```



Queues

What is a Queue?

- An abstract data type (ADT)
  - Ordered collection of elements
- Stanford C++ library (here)
  - `#include “queue.h”`
- Modeled like a real queue/line
- **F**irst **I**n, **F**irst **O**ut (FIFO)

The Stanford Queue Library

```
#include "queue.h"
```

- **`queue.enqueue(value)`**: Add an element to the back of the queue
- **`queue.dequeue()`**: Remove an element from the front of the queue and return it
- **`queue.peek()`**: Look at the element from the front of the queue, but don't remove it
- **`queue.isEmpty()`**: Returns a boolean value, true if the queue is empty, false if it has at least one element
  - Note: a runtime error occurs if a dequeue() or peek() operation is attempted on an empty queue
- **`queue.clear()`**: Removes all elements from the queue
- **`queue.size()`**: Returns the number of elements in the queue

For more information, check out the Stanford Stack class documentation!

Queue Operations: Creation

```
Queue<string> bankQueue;
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue(“Matilda”);
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue(“Matilda”);
bankQueue.enqueue(“Emma”);
bankQueue.enqueue(“Coraline”);
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
```

Console:

```
Matilda
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
```

Console:

```
Matilda
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
cout << bankQueue.peek() << endl;
```

Console:

```
Matilda
Emma
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
cout << bankQueue.peek() << endl;
cout << bankQueue.dequeue() << endl;
```

Console:

```
Matilda
Emma
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
cout << bankQueue.peek() << endl;
cout << bankQueue.dequeue() << endl;
```

Console:

```
Matilda
Emma
Emma
```

Queue Operations: Creation

```
Queue<string> bankQueue;
bankQueue.enqueue("Matilda");
bankQueue.enqueue("Emma");
bankQueue.enqueue("Coraline");
cout << bankQueue.dequeue() << endl;
cout << bankQueue.peek() << endl;
cout << bankQueue.dequeue() << endl;
```

Console:

```
Matilda
Emma
Emma
```

Queue Operations: Creation with Elements

```
Queue<string> bankQueue = {“Matilda”, “Emma”,
                                                   “Coraline”};
```

Queue Operations: Printing

```
Queue<string> bankQueue = {“Matilda”, “Emma”,
                                        “Coraline”};

cout << bankQueue << endl;
```

Console:

```
{“Matilda”, “Emma”,
“Coraline”}
```

Queue Operations: Printing

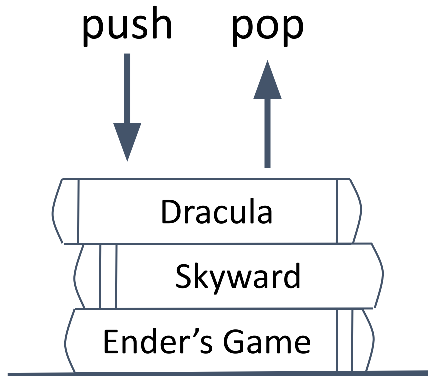
```
Queue<string> bankQueue = {“Matilda”, “Emma”,
                                                  “Coraline”};
cout << bankQueue << endl;
cout << bankQueue << endl;
```

Console:

```
{“Matilda”, “Emma”,
“Coraline”}
{“Matilda”, “Emma”,
“Coraline”}
```

Stack

Last **I**n, **F**irst **O**ut (LIFO)



Queue

First **I**n, **F**irst **O**ut (FIFO)

Tradeoffs with Stacks and Queues

What are some downsides?

- No random access of elements
- Difficult to traverse - requires removal of elements
- No easy way to search

What are some benefits?

- Useful for many real world problems
- Easy to build such that access is guaranteed to be fast

Stacks in Programming

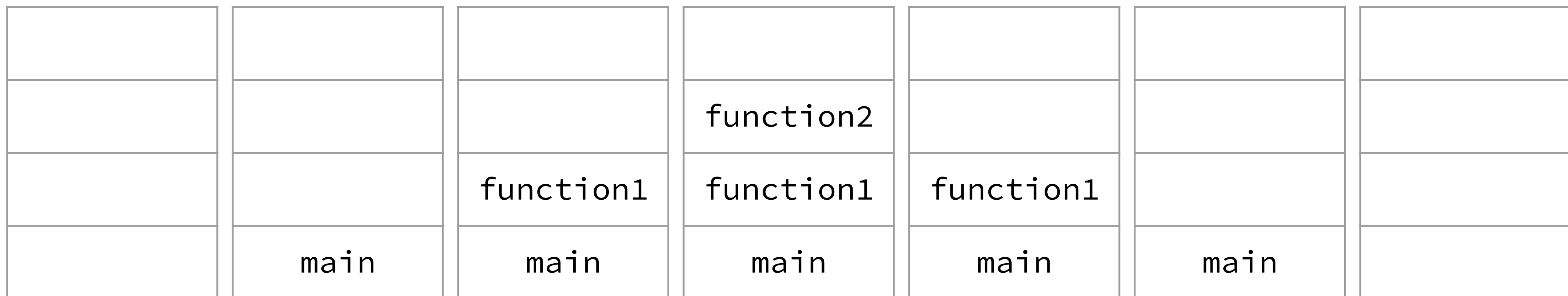
- Stacks are very frequently used in programming
  - Most computer architectures implement a stack
- There is a stack built into every program running on your computer
- Postfix notation (Reverse Polish Notation)

Stacks in Programming

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

```
void function1() {
    …
    function2();
    …
    return;
}
```

```
void function2() {
    …
    return;
}
```



Which ADT would be best for…

1. the undo button in a text editor?
2. jobs submitted to a printer that can also be cancelled?
3. LaIR sign-up?
4. your browsing history?
5. Google spreadsheets

What is the output?

```
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue << "  size " << queue.size() << endl;
```

What is the output?

```
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue << "  size " << queue.size() << endl;
```

Console:

```
1 2 3 {4,5,6} size 3
```

Idiom 1: Emptying a Stack/Queue

```
Queue<int> queueIdiom1;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queueIdiom1.enqueue(i);
}
while (!queueIdiom1.isEmpty()) {
    cout << queueIdiom1.dequeue() << " ";
}

cout << queueIdiom1 << "  size " << queueIdiom1.size() << endl;
```

Console:

```
1 2 3 4 5 6 {} size 0
```

Idiom 1: Emptying a Stack/Queue

```
Stack<int> stackIdiom1;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    stackIdiom1.push(i);
}
while (!stackIdiom1.isEmpty()) {
    cout << stackIdiom1.pop() << " ";
}

cout << stackIdiom1 << "  size " << stackIdiom1.size() << endl;
```

Console:

```
6 5 4 3 2 1 {} size 0
```

Idiom 2: Iterating over a Stack/Queue

```
Queue<int> queueIdiom2 = {1, 2, 3, 4, 5, 6};

int origQSize = queueIdiom2.size();
for (int i=0; i < origQSize; i++) {
    int value = queueIdiom2.dequeue();
    cout << value << " ";
    // re-enqueue even values
    if (value % 2 == 0) {
        queueIdiom2.enqueue(value);
    }
}
cout << queueIdiom2 << endl;
```

Console:

```
1 2 3 4 5 6 {2, 4, 6}
```

Idiom 2: Iterating over a Stack/Queue

```
Stack<int> stackIdiom2 = {1, 2, 3, 4, 5, 6};
Stack<int> result;

int origSSize = stackIdiom2.size();
for (int i=0; i < origSSize; i++) {
    int value = stackIdiom2.pop();
    cout << value << " ";
    // add back even values
    if (value % 2 == 0) {
        result.push(value);
    }
}
cout << result << endl;
```

Console:

```
6 5 4 3 2 1 {6, 4, 2}
```

Reversing Words in a Sentence

Let’s build a program from scratch that reverses the words in a sentence.

Example input: “the cat in the hat”

Example output: “hat the in cat the”

Let’s make a plan! Some things to think about:

- Which ADT should we use?
- What steps will we need to do?

Recap of ADTs So Far

ADTs with indices

Types

- Vectors (1D)
- Grids (2D)

Properties

- Easily able to search through all elements
- Can use the indices as a way of accessing specific elements

ADTs without indices

Types

- Stacks (LIFO)
- Queues (FIFO)

Properties

- Constrains the way you can insert and remove data
- More efficient for solving specific LIFO/FIFO problems