# The Standard Template Library

...

Written by Rachel J. Morris, last updated 1/6/17

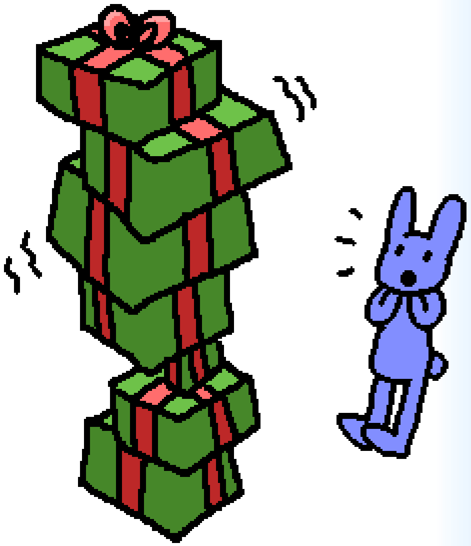
# Topics:

- The Standard Template Library
- Vectors
- Lists
- Stacks
- Queues
- Maps

# Introduction

C++ has some data structures already pre-built and available within the standard library that we can use to get familiar a bit with how these structures work.

Before we get into the details of *implementing* these data structures, it might be useful to see how they can actually be used first…



# Introduction

**Wait, if data structures already exist, why are we going to spend all semester learning how to write 'em?!**

# Introduction

**Wait, if data structures already exist, why are we going to spend all semester learning how to write 'em?!**


- Well… learning about data structures & algorithm analysis is a big part of computer science.
- You also need to know the inner-workings of these structures, so that when you're implementing solutions, you can choose the best structure for your particular problem – it isn't one size fits all!

# Introduction

**Wait, if data structures already exist, why are we going to spend all semester learning how to write 'em?!**

- You might need to create your own data structures down the line, or customize an existing one!

- You'll be asked about them during job interviews. Interviewers <u>love</u> asking about data structures.

# Introduction

**So let's see how some of these structures work by utilizing the structures available in the Standard Template Library.**

**In particular…**

- **STL Vector**
- **STL List**
- **STL Stack**
- **STL Queue**
- **STL Map**

# STL Vector

# STL Vector

**In some ways, vector objects are similar to arrays, which you may have used in previous classes.**

**You can access specific items of the vector with the subscript operator, [ ]**

```
cout << "Price: " << itemPrices[5] << endl;
```

# STL Vector

**A perk of the vector object is that it handles resizing on its own.**

**Recall that with a static array, we had to know what its size was at <u>compile time,</u> and it <u>couldn't be resized!</u>**

```
int sadArray[100];
for ( int i = 0; i < 100; i++ )
{
    sadArray[i] = i * 2;
}
cout << "sadArray is full and cannot store any more...";
```

# STL Vector

**With a STL vector, it handles resizing on its own (behind-the-scenes!), so we don't have to worry about it – we can just keep adding items on to it!**

```
vector<float> itemPrices;

itemPrices.push_back( 9.99 );
itemPrices.push_back( 7.99 );
itemPrices.push_back( 6.99 );
```

Now we're the "other programmer", and we don't really care how it's implemented, we just care that it works!

The vector's **push_back** function is how we add items into the "array".

# STL Vector

## A Vector can store any data-type.

```
vector<int> myNumbers;
myNumbers.push_back( 20 );

vector<string> studentNames;
studentNames.push_back( "Bob" );

vector<float> itemPrices;
itemPrices.push_back( 9.99 );
```

# STL Vector

## A Vector can store any data-type.

```
vector      myNumbers;
myNumbers.push_back( 20 );

vector        studentNames;
studentNames.push_back( "Bob" );

vector        itemPrices;
itemPrices.push_back( 9.99 );
```

The <int>, <string>, and <float> bits of code are because vector has been implemented as a **template**.

If you haven't covered templates before, or don't quite remember how they work, don't worry – we will go over them more later on.

# STL Vector

## A Vector can store any data-type.

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

```
vector<CoordPair> coordinatePairs;
```

If we write a **struct** or a **class**, a vector can even store those!

# STL Vector

## Some handy functions of a vector are...

- **push_back**
  Insert an item at the end of the vector
- **size**
  Returns the amount of items in the vector
- **empty**
  Returns whether the vector is empty or not (size == 0?)
- **operator[]**
  Access an item in the vector at any index
- **clear**
  Clears out all the elements of the vector.

# STL Vector

**Let’s try it out!**

# STL List

# STL List

Lists also store a linear series of data, but they're a little different from vectors.

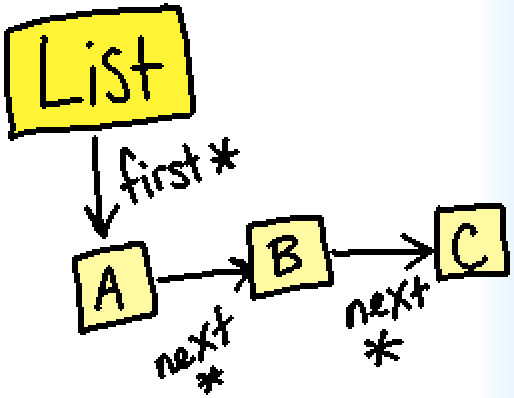
For one, you cannot *randomly access data* with the subscript operator [ ].

Generally, to step through a list, you have to start at the beginning and keep stepping through, one at a time.

The STL List does contain a sort() function and reverse() function, though!

# STL List

We cannot *randomly* access data in a List because it isn't implemented with an <u>array</u>, like <u>vector</u> is.



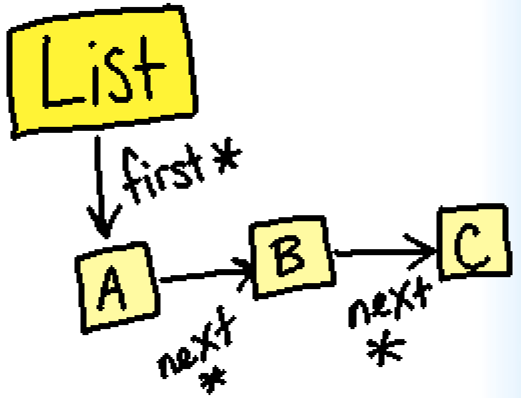

STL Lists use <u>pointers</u>. The list keeps track of what its <u>starting element</u> is, and each element points to the next element in the list.

**Therefore, unlike an array, the elements are not in contiguous memory slots.**

(This is why it's important to stay familiar with pointers for this class!)

# STL List

**We will discuss the pointer and memory aspect of lists later on, once we're implementing <u>linked lists</u>.**




**For now…**
**onto the STL List functionality!**

# STL List

- **push_back**
  Insert an item at the end of the list

- **size**
  Returns the amount of items in the list

- **empty**
  Returns whether the list is empty or not (size == 0?)

- **clear**
  Clears out all the elements of the list.

- **sort**
  Sorts the elements of the list

- **reverse**
  Reverses the order of elements in the list.

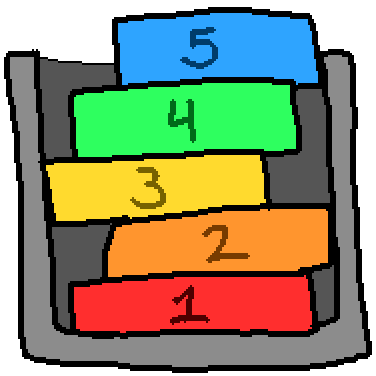
# STL List

**Let’s try it out!**

# STL Stack

# STL Stack

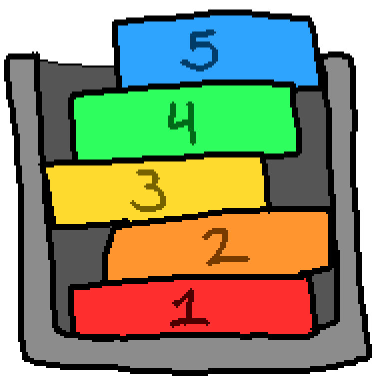
A **Stack** is a type of data structure that is linear, like a list or vector is, but it also **restricts access to the internal data**.




The main characteristic of a stack is that it is a **First In Last Out** (or) **Last In First Out** structure.

# STL Stack

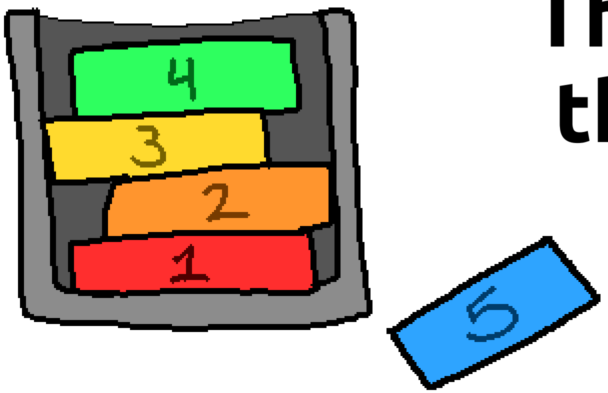
**At any time, you're only able to access <u>one item</u> from the stack – the top-most item.**




**And, as items are <u>pushed</u> onto the stack, the older items are on the bottom, and the newer items are on the top.**

# STL Stack

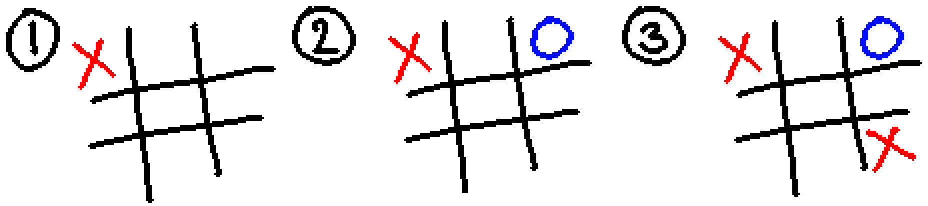
**As you remove items from the stack, you pull the newest item that was added to the stack.**



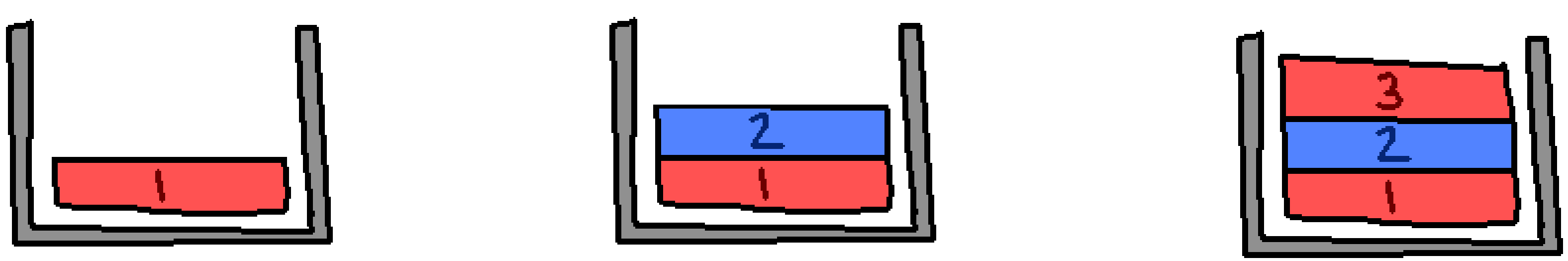

**The first item on the stack is the last one to be removed.**

# STL Stack

**One example of using a Stack is to keep track of moves in a game of tic-tac-toe**

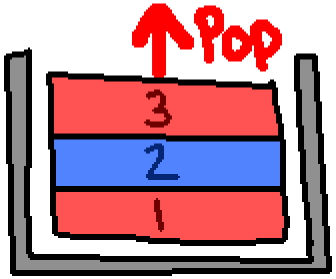



**Every time a move is made, we could push the game board's current state onto a stack...**

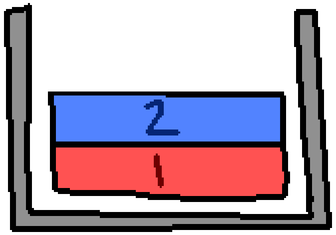


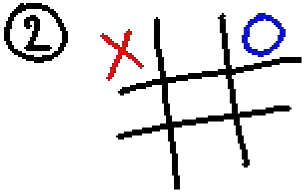
# STL Stack

**Then if we want to undo a turn, we can pop the most recent state off the stack**




**And the game board reverts to the state before.**



# STL Stack

## Some handy functions of a stack are...

- **push**
  Pushes an item to the top of the stack.

- **pop**
  Removes an item from the top of the stack.

- **top**
  Returns the item that is at the top of the stack.

- **size**
  Returns the amount of items in the stack.

- **empty**
  Returns whether the stack is empty or not.

# STL Stack

**Let's try it out!**

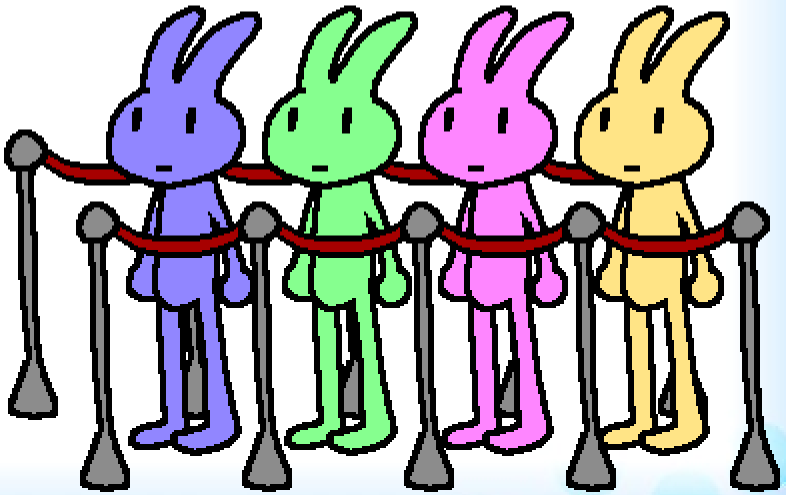
# STL Queue

# STL Queue

A queue is another kind of restricted-access data structure.

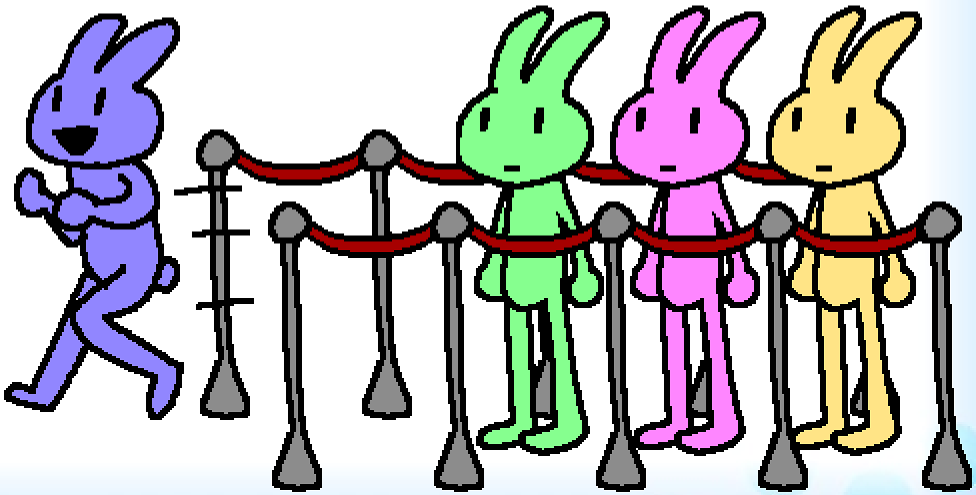
Like a stack, you can only access one item of the queue at a time.

However, a queue is **First In First Out** structure



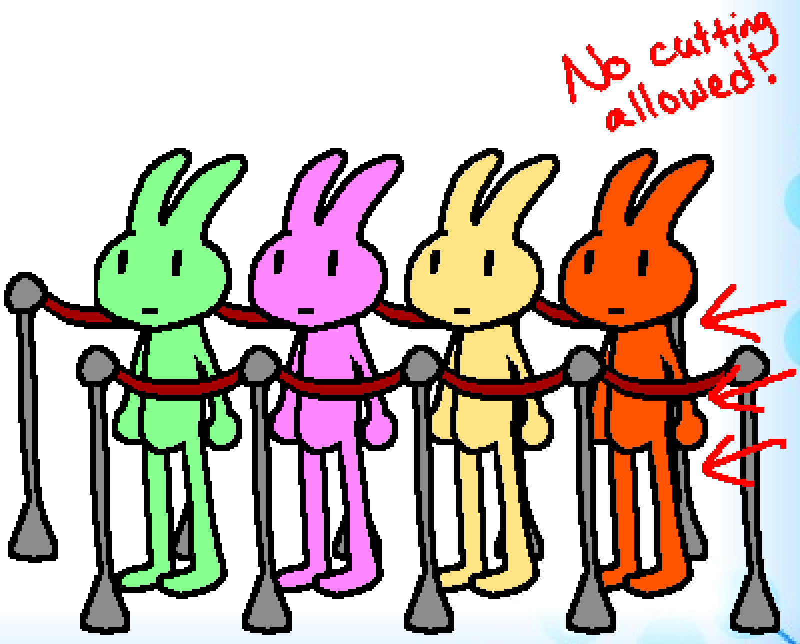
# STL Queue

**The first item that enters the queue, who sits at the <u>front</u> of the line, is the first one to get removed, just like in a grocery-store line.**



# STL Queue

**And when a new item is added to the queue, it enters at the end (or back) of the queue.**



# STL Queue

## Some handy functions of a queue are…

- **push**
  Pushes an item to the back of the queue.

- **pop**
  Removes an item from the front of the queue.

- **front**
  Returns the item that is at the front of the queue.

- **size**
  Returns the amount of items in the queue.

- **empty**
  Returns whether the queue is empty or not.

# STL Queue

**Let’s try it out!**

# STL Map

# STL Map

**When we're using a plain-old array, we have a series of elements in order, starting at 0, going until (size – 1).**


**7 items in the array**
**=**
**Index 0 through 6 is valid**

# STL Map

**A value of 0, 1, 2, 3, 4, 5, or 6, which specifies an element's position, is known as an <u>index</u>, but we can also think of it like a "key", which helps us locate the value we want**

# STL Map

**But with a data structure like a <u>map</u>, our keys don't have to just be integers, and they don't have to be array indices.**

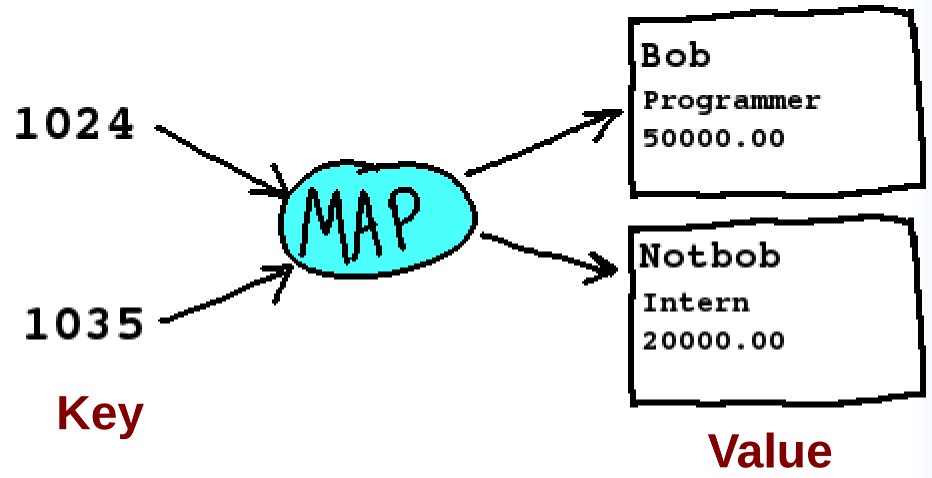
**The key can be <u>any data type</u>, and it can point to a value of <u>any data type as well</u>.**

# STL Map

**For example, think of an employee ID, that points to an employee object in a program...**

```
class Employee
{
    public:
    // ...

    private:
    string name;
    string jobTitle;
    float salary;
};
```

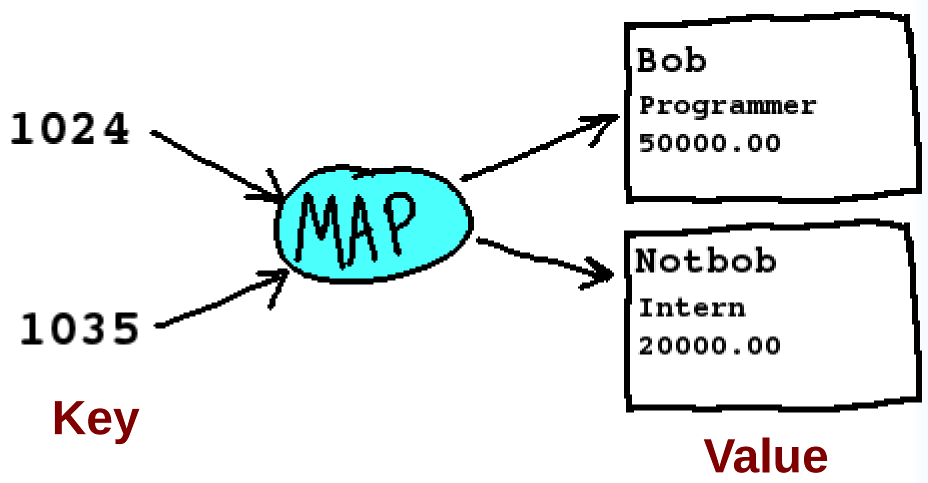


# STL Map

**Maps are built in a special way so that we can access elements by key, and it will access it quickly – search algorithm not required.**

```
class Employee
{
    public:
    // ...

    private:
    string name;
    string jobTitle;
    float salary;
};
```



# STL Map

## Some handy functions of a vector are...

- **operator[]**
  Access an element of the map, using a key
- **insert**
  Insert a new key-value pair into the map
- **empty**
  Returns whether the map is empty or not
- **size**
  Get the amount of elements in the map

# STL Map

**Let’s try it out!**

# CPlusPlus.com

A really handy page for C++ documentation is CplusPlus.com

If you look up stuff from the C++ standard library, you will find objects included in the library, functions that those objects have, and example code.

http://www.cplusplus.com/

# Practice

**Make sure to check out the course GitHub for example code and practice projects.**