Topics for Today

- Revision
    - Pointers
    - Function signatures in classes
- CS Theory
    - The stack and the heap
    - Hash maps/sets
- C++ Theory
    - std::pair
- This week's lab
    - Two Sum and the hashset
    - Implement a stack
    - Postfix stack calculator
    - Final assignment questions

Pointers and References

As initialisers

int* x “Pointer”
Initialises a variable containing a memory address.
Contains:
0x00000000

int& x “Reference”
Initialises an *alias* for an existing variable. They both refer to the same memory address to read/write their data.
Contains: data

As operators

&x “Address of”
Get the address of this variable.
Returns:
0x00000000

*x “Dereference”
Gets the data stored in that memory address.
Used on pointers.
Returns: data

Complex Function Signatures

We saw some complex function signatures last week, here's some help on reading them
They follow a precise grammar we can follow. First a basic example:

return type function name(parameter 1 type parameter1 name, ...) {
    return statement return value ;
}

```
int add(int a, int b) {
        return a + b;
}
```

Then we add in classes to the mix:

return type class name::function name(parameter 1 type parameter1 name, ...) {
    return statement return value ;
}

```
float vec3::dot(vec3 a, vec3 b)
{
        return a + b;
}
```

You can read the '`::`' separated names as "is owned by" reading right to left ←

```
typename MyVector<T>::Iterator& MyVector<T>::Iterator::operator++()
```

"++ operator is owned by *Iterator*, which is owned by *MyVector<T>*" and returns an Iterator reference.

std::pair

The pair is a simple data structure of two values, like an array of length two, but not necessarily of the same type.

For example, you can make a pair:

```
std::pair<std::string, int> example {"Hello", 5};
```

or return a pair of ints:

```
return {1,2};
```

std::size_t

A certain amount of memory is allocated for each type in C++, for an int this limits the range of numbers it can support.

The `std::size_t` is simply the largest size of `int` that your computer can handle.

You may have noticed your compiler sometimes complaining about iterating through a vector with an int for the index. This is because the size is returned as a size_t, and we are doing an implicit conversion to int which is not completely safe.

Hash maps/sets

You can think of this data structure as like an array, but instead of holding its elements in order, we trade this away for having O(1) search, add, remove.
It's equivalent to a dictionary in Python or C#.

So an array is very useful for storing things in order, but so far in this class if we've wanted to store a collection of things without order it's been the best tool.
Now we have hash maps as a much better tool for these scenarios.

C++'s `std::unordered_map` has the following functions:

- `[]` - Both accessor and add method.
    - map[key] will return the value stored there
    - map[key] = x; will place a value at that key
- contains - Check if the map/set contains this value already

Two Sum

This is a really fun example of how a hashset can come in handy.
In this problem, you have are given an array of ints, and a target sum.
Your goal is to return the indices of the two values in the array, that added together sum to the target.

In C++ it's called an unordered map/set, in our case we are using:

```
std::unordered_map<int, int> map {};
```

<key, value>

Find a way to determine if the necessary index to create the target has already been reached.
You are not allowed to use more than one for-loop.
You know where the necessary compliment would be, so search for it in O(1).
Keep in mind, your set is *unordered*.

Implement a Stack

A stack is a data structure you can imagine like a stack of books, you can only add to the top of the stack, and you can only take from the top of the stack.

You are implementing one using a `std::vector` inside your class
You are implementing the following functions:

- `void push(T val)` – Add `val` to the top of the stack.
- `void pop()` – Remove the top value on the stack.
- `T& top()` – Return a reference to the top value on the stack.
- `int size()` – Return the number of elements in the stack.
- `bool empty()` – Return whether or not the stack is empty.

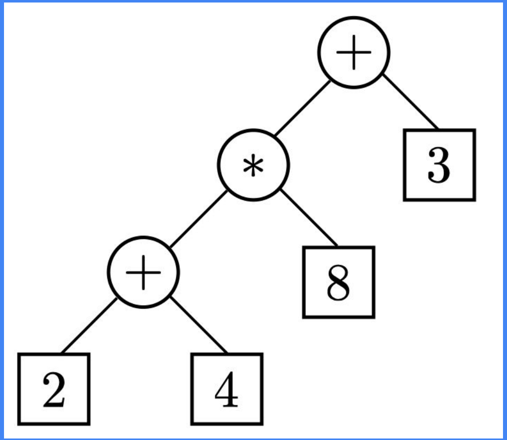
Stack Calculator & Postfix

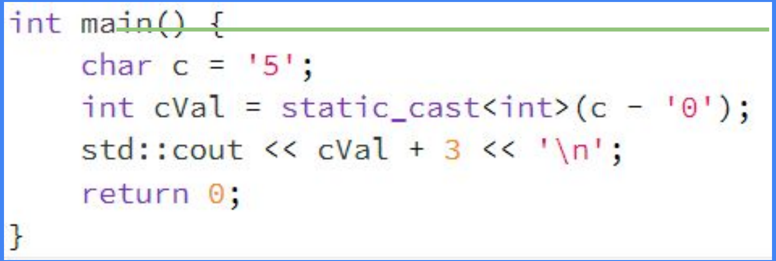
Postfix notation is a way of writing mathematical expressions that does not require brackets to disambiguate.
It is not the most human readable form, but it is a very easily computable form.
We are going to use this property to write a simple calculator, taking a vector of chars as input, and returning the evaluation of the expression.

In this format, we load our inputs onto a stack, and use this stack format to evaluate the expression. When we see a number, we put it on the stack, and when we see an operator, we take the two elements off the top of the stack, operate on them, and push the result back to the stack.

Stack Calculator & Postfix

This graph represents the input array
2 4 + 8 * 3 +
So we put 2 and 4 onto the stack,
Then operate + to them, producing 6,
Then we add 8 to the stack,
See a * and pop 6 and 8 off the stack, multiply, producing 48
Finally we see a 3, add it to the stack,
Then a + operator, take 3 and 48 off the stack and add them,
Leaving us with a final result of 51.





```
int main() {
    char c = '5';
    int cVal = static_cast<int>(c - '0');
    std::cout << cVal + 3 << '\n';
    return 0;
}
```

Note: char constants are written with single quotes `'a'` do not use double quotes `"a"`
Troy provides us with an example of how to convert from char to the int it represents: