## Topics for Today

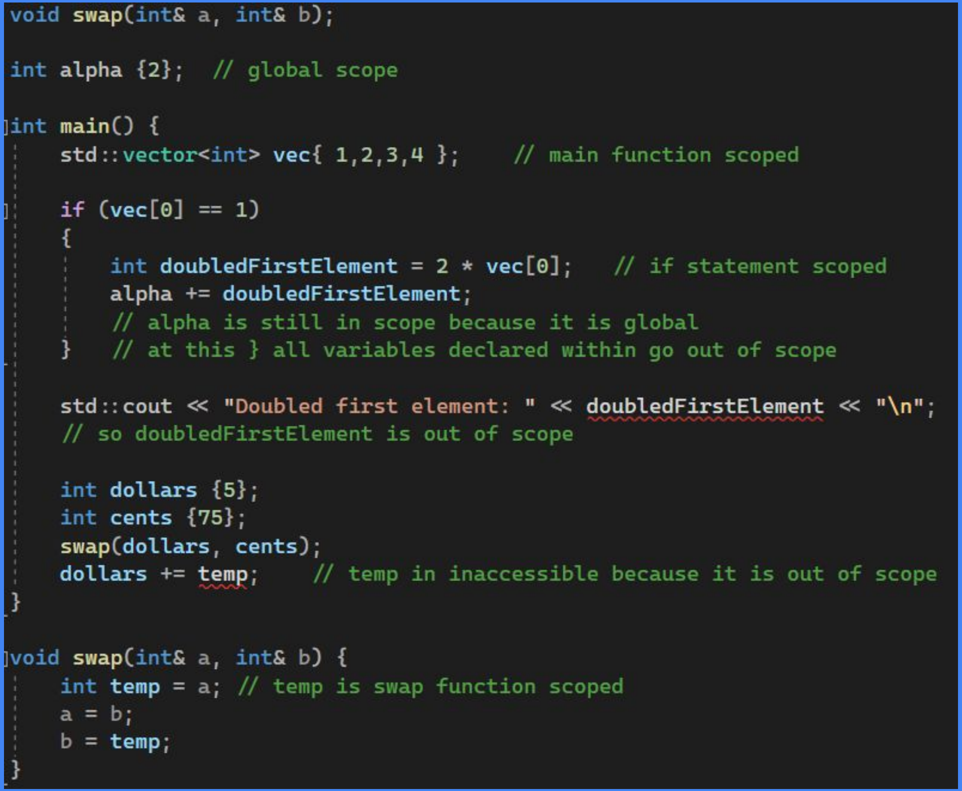
- Revision
- C++ theory
    - Scopes
    - Ternary operator
    - Iterators
    - Templates
- This week’s lab
    - Templated functions
    - Linked list
    - Templated MyVectors
    - More MyVector functions

## Upcoming Dates

March 18 - Assignment 1 Released
March 20 - Census date
April 12 - Assignment 1 Due

# Scopes

Variables exist inside scopes, which are indicated with { }



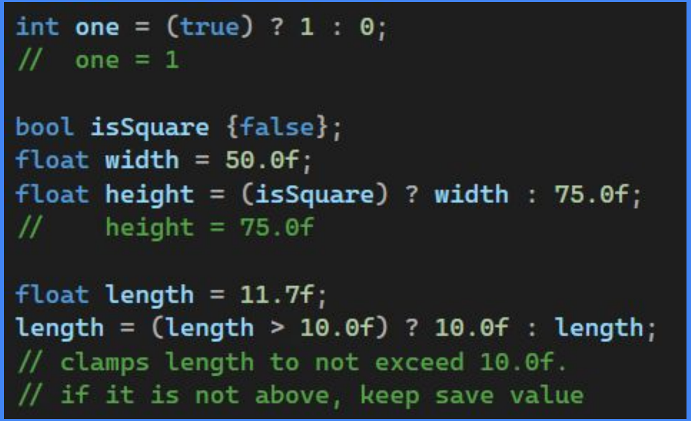
```
void swap(int& a, int& b);

int alpha {2};   // global scope

int main() {
    std::vector<int> vec{ 1,2,3,4 };     // main function scoped

    if (vec[0] == 1)
    {
        int doubledFirstElement = 2 * vec[0];    // if statement scoped
        alpha += doubledFirstElement;
        // alpha is still in scope because it is global
    }   // at this } all variables declared within go out of scope

    std::cout << "Doubled first element: " << doubledFirstElement << "\n";
    // so doubledFirstElement is out of scope

    int dollars {5};
    int cents {75};
    swap(dollars, cents);
    dollars += temp;     // temp in inaccessible because it is out of scope
}

void swap(int& a, int& b) {
    int temp = a; // temp is swap function scoped
    a = b;
    b = temp;
}
```

## Ternary Operator

The ternary operator is a one line expression that can be used to replace an in-else clause.

The ternary operator uses a boolean expression, and returns one of two values according to the result of the bool.

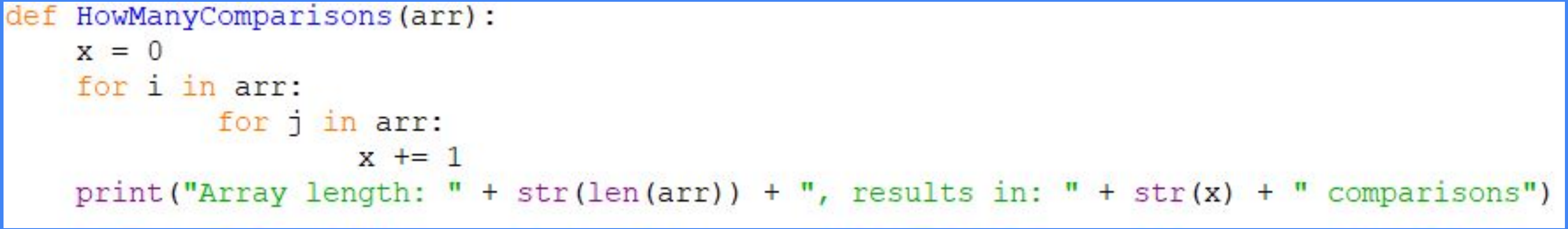
Form:
(boolean) ? trueVal : falseVal



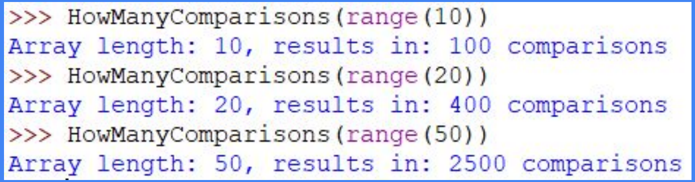
```
int one = (true) ? 1 : 0;
//  one = 1

bool isSquare {false};
float width = 50.0f;
float height = (isSquare) ? width : 75.0f;
//    height = 75.0f

float length = 11.7f;
length = (length > 10.0f) ? 10.0f : length;
// clamps length to not exceed 10.0f.
// if it is not above, keep save value
```

# What does O(n^2) mean?

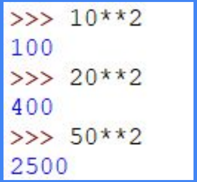
Looping n times within a loop of n iterations results in n^2 total iterations.



```python
def HowManyComparisons(arr):
    x = 0
    for i in arr:
            for j in arr:
                    x += 1
    print("Array length: " + str(len(arr)) + ", results in: " + str(x) + " comparisons")
```

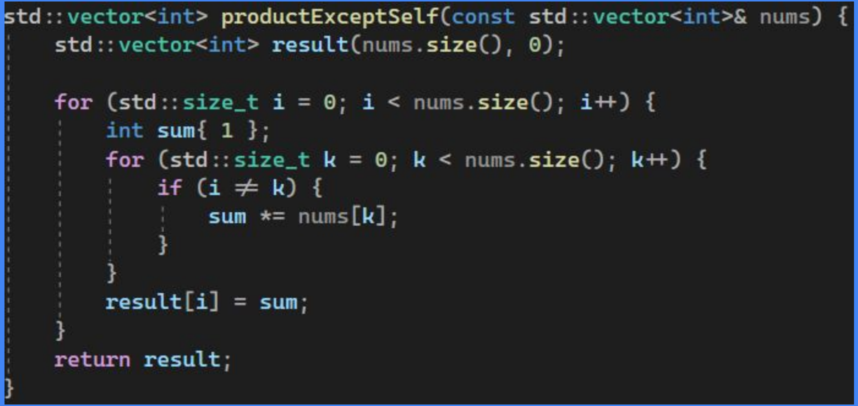


```
>>> HowManyComparisons(range(10))
Array length: 10, results in: 100 comparisons
>>> HowManyComparisons(range(20))
Array length: 20, results in: 400 comparisons
>>> HowManyComparisons(range(50))
Array length: 50, results in: 2500 comparisons
```



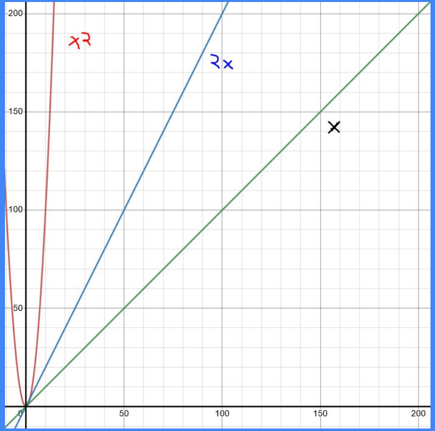
```
>>> 10**2
100
>>> 20**2
400
>>> 50**2
2500
```

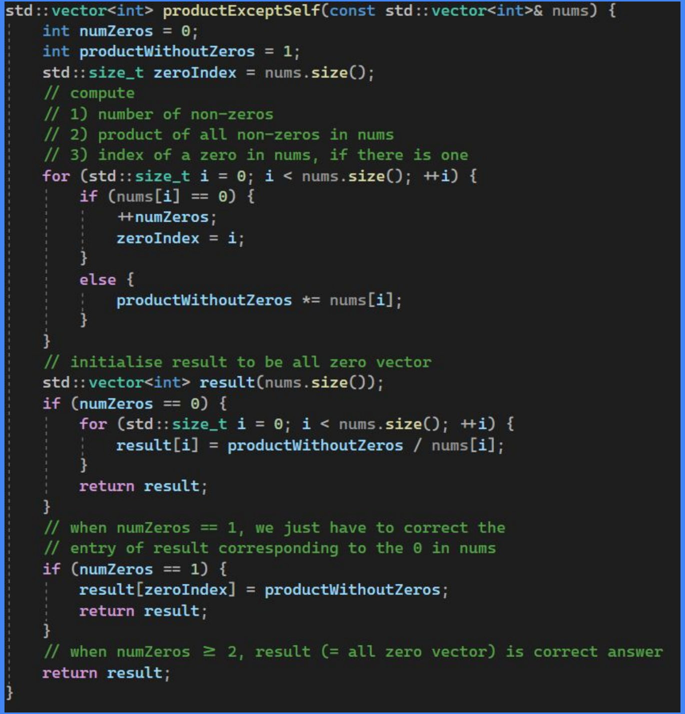
# O(n^2) and O(n) solutions to AXS



```cpp
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    std::vector<int> result(nums.size(), 0);

    for (std::size_t i = 0; i < nums.size(); i++) {
        int sum{ 1 };
        for (std::size_t k = 0; k < nums.size(); k++) {
            if (i ≠ k) {
                sum *= nums[k];
            }
        }
        result[i] = sum;
    }
    return result;
}
```

Worst case:   O(n^2)
Best case:     O(n^2)
Middle case: O(n^2)





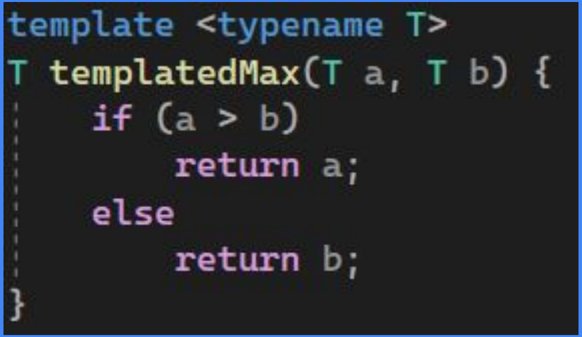
```cpp
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    int numZeros = 0;
    int productWithoutZeros = 1;
    std::size_t zeroIndex = nums.size();
    // compute
    // 1) number of non-zeros
    // 2) product of all non-zeros in nums
    // 3) index of a zero in nums, if there is one
    for (std::size_t i = 0; i < nums.size(); ++i) {
        if (nums[i] == 0) {
            ++numZeros;
            zeroIndex = i;
        }
        else {
            productWithoutZeros *= nums[i];
        }
    }
    // initialise result to be all zero vector
    std::vector<int> result(nums.size());
    if (numZeros == 0) {
        for (std::size_t i = 0; i < nums.size(); ++i) {
            result[i] = productWithoutZeros / nums[i];
        }
        return result;
    }
    // when numZeros == 1, we just have to correct the
    // entry of result corresponding to the 0 in nums
    if (numZeros == 1) {
        result[zeroIndex] = productWithoutZeros;
        return result;
    }
    // when numZeros ≥ 2, result (= all zero vector) is correct answer
    return result;
}
```

Worst case:  O(2*n)
Best case:     O(n)
Middle case: Either

# Templates

Templates allow us to quickly and easily write code that supports many datatypes without rewriting any of our code for each new datatype.

It basically allows us to pass the datatype as parameter, instead of rewriting the function.



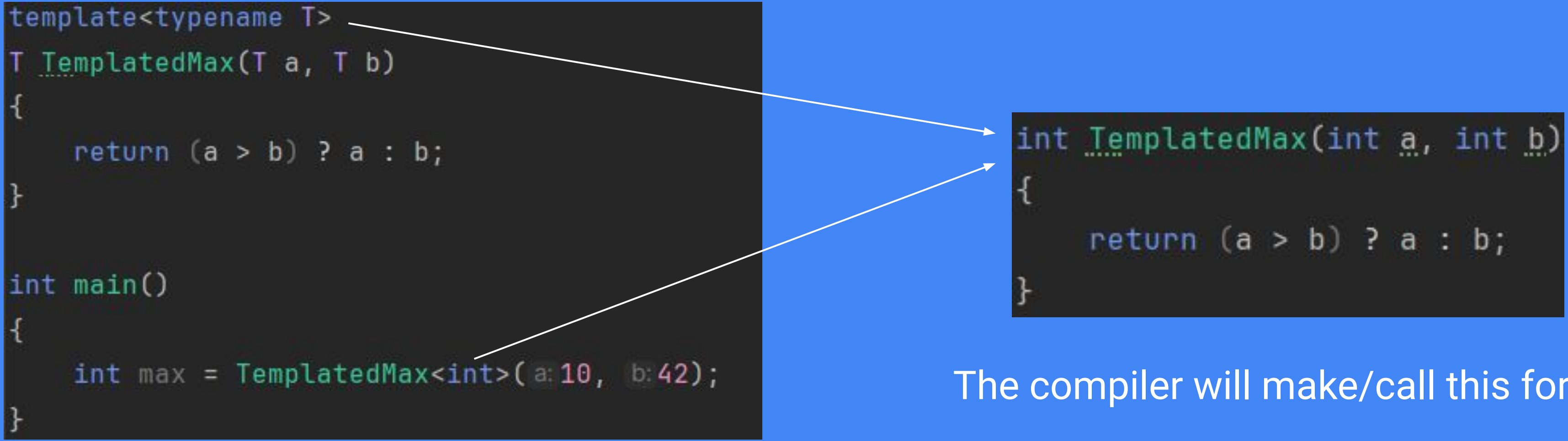
```cpp
template <typename T>
T templatedMax(T a, T b) {
    if (a > b)
        return a;
    else
        return b;
}
```

First line defines our standin type `T`, which represents whatever datatype we want to use, ie `int`. We can even give it our own types, if we define its necessary operators i.e. `>`,`*` It doesn't have to be called T, but it is convention to do so, especially when dealing with only one template.

How does it work? At compile time the compiler makes a new version of the function for each type that it gets used with in the code.

# Templates

For five minutes try
week 04 *Templated Functions*



```
template<typename T>
T TemplatedMax(T a, T b)
{
    return (a > b) ? a : b;
}

int main()
{
    int max = TemplatedMax<int>(a: 10, b: 42);
}
```

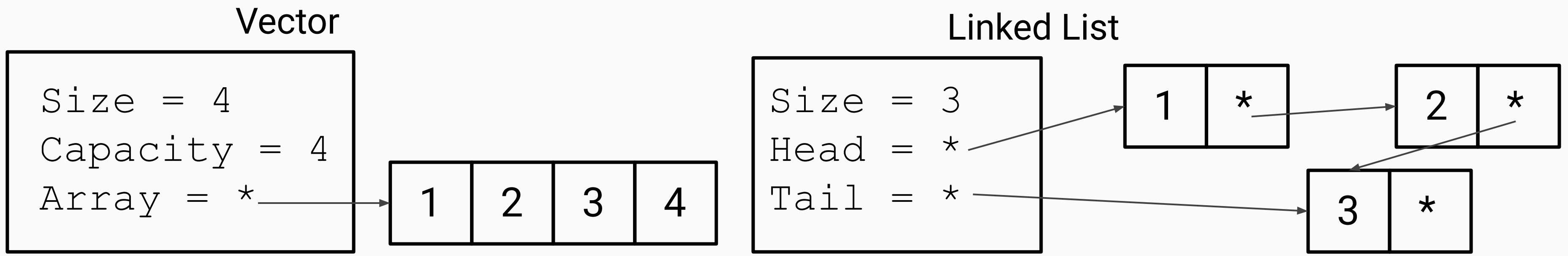
```
int TemplatedMax(int a, int b)
{
    return (a > b) ? a : b;
}
```

The compiler will make/call this for us

# Singly Linked List

Similarly to arrays and vectors, a linked list is a *sequence container* data structure. Where arrays and vectors store all their elements in continuous memory, this is not the case for linked lists.

Linked lists instead use a series of *node* objects to store their data, and a pointer to the following node in the sequence.



## Singly Linked List

Because it is a different data structure, it has different affordances and best use cases compared to an array/vector.

An array/vector gives O(1) random access (get any element)
A linked list's random access speed is O(n), proportional to where in the list the element is

However, adding/removing an element to a linked list anywhere in the sequence is faster than for a vector.
For a vector it is amortised constant to add to the end. But when we have to allocate more space this is slow. Removing an element from random access is always slow.

# Singly Linked List

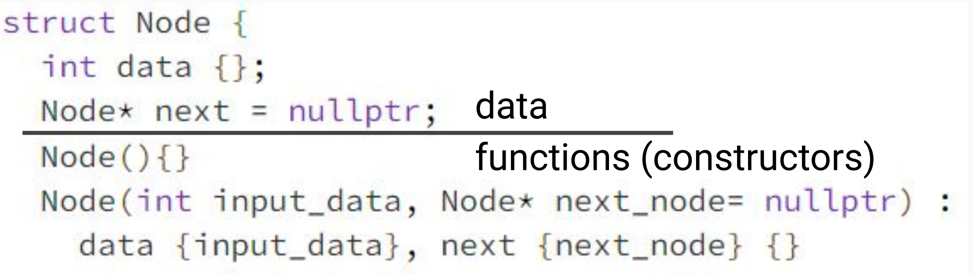
Linked lists can be divided into two categories:

Singly linked- each node contains a pointer to the following node only

Doubly linked- each node contains a pointer to the following and previous node

In this activity it is your job to write some functions for a singly linked list.

In the header file you can find the definition of our node structure.



```
struct Node {
  int data {};
  Node* next = nullptr;
  Node(){}
  Node(int input_data, Node* next_node= nullptr) :
    data {input_data}, next {next_node} {}
```

data

functions (constructors)

Functions to write:

`push_front()`

`display()`

`front()`

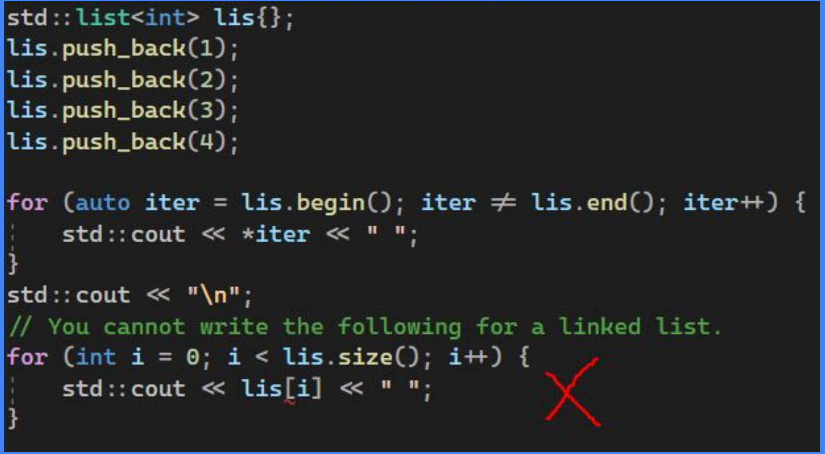
`empty()`

`pop_front()`

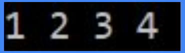
initializer list constructor

## Iterators

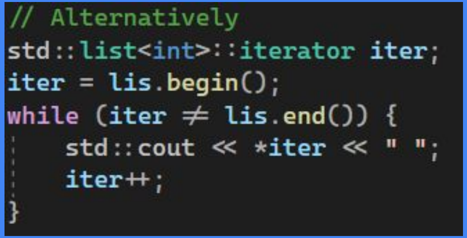
Iterators are objects that allow us another way of iterating through an array/list like data structure.
An iterator is a pointer to a place in memory within our list. We can increment and decrement them (traverse the list),
and dereference them to see what is inside.



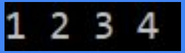
```cpp
std::list<int> lis{};
lis.push_back(1);
lis.push_back(2);
lis.push_back(3);
lis.push_back(4);

for (auto iter = lis.begin(); iter != lis.end(); iter++) {
    std::cout << *iter << " ";
}
std::cout << "\n";
// You cannot write the following for a linked list.
for (int i = 0; i < lis.size(); i++) {
    std::cout << lis[i] << " ";
}
```



```
1 2 3 4
```



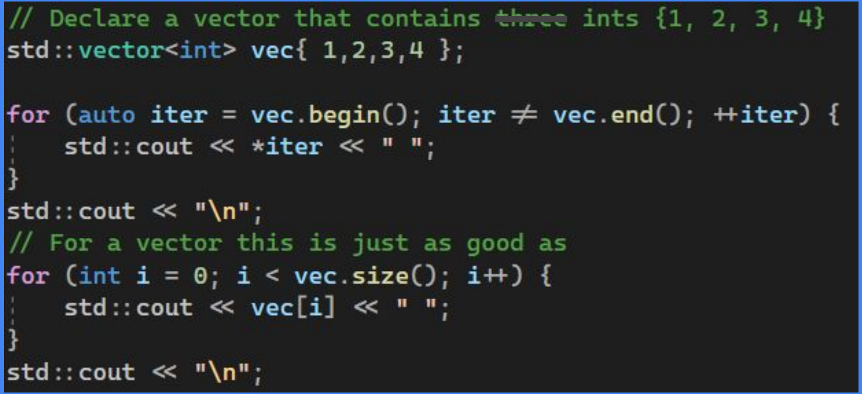
```cpp
// Alternatively
std::list<int>::iterator iter;
iter = lis.begin();
while (iter != lis.end()) {
    std::cout << *iter << " ";
    iter++;
}
```



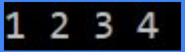
```
1 2 3 4
```

# Iterators

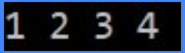
Iterators are objects that allow us another way of iterating through an array/list like data structure.
An iterator is a pointer to a place in memory within our list. We can increment and decrement them (traverse the list),
and dereference them to see what is inside.



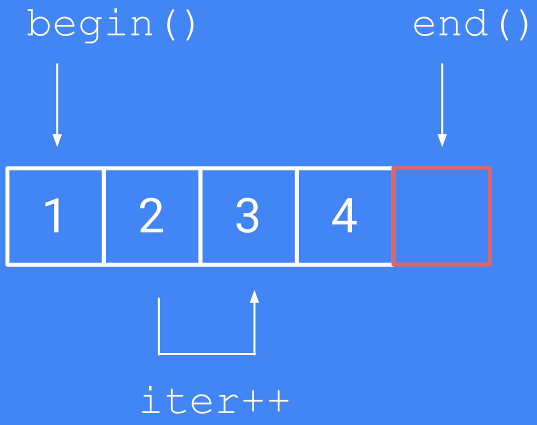
```cpp
// Declare a vector that contains ~~three~~ ints {1, 2, 3, 4}
std::vector<int> vec{ 1,2,3,4 };

for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
    std::cout << *iter << " ";
}
std::cout << "\n";
// For a vector this is just as good as
for (int i = 0; i < vec.size(); i++) {
    std::cout << vec[i] << " ";
}
std::cout << "\n";
```



```
1 2 3 4
```

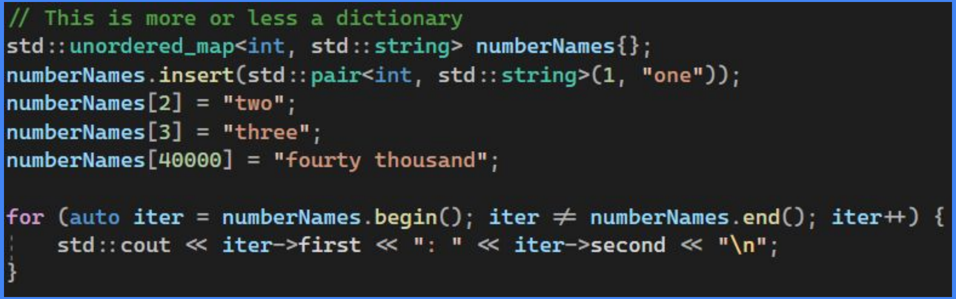


```
1 2 3 4
```

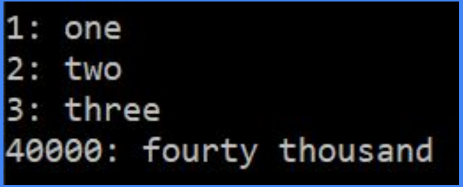


# Iterators

Iterators are objects that allow us another way of iterating through an array/list like data structure.
An iterator is a pointer to a place in memory within our list. We can increment and decrement them (traverse the list),
and dereference them to see what is inside.



```cpp
// This is more or less a dictionary
std::unordered_map<int, std::string> numberNames{};
numberNames.insert(std::pair<int, std::string>(1, "one"));
numberNames[2] = "two";
numberNames[3] = "three";
numberNames[40000] = "fourty thousand";

for (auto iter = numberNames.begin(); iter ≠ numberNames.end(); iter++) {
    std::cout << iter->first << ": " << iter->second << "\n";
}
```



```
1: one
2: two
3: three
40000: fourty thousand
```

It even works with unordered types, giving us a useful way to check through their elements

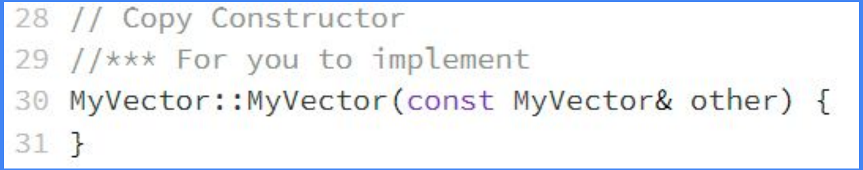
# Return to MyVector

It's alright we didn't finish this last week, it's the same problem but expanded this week. Vectors use an internal `int[]` that they handle and rebuild to change lengths for us, and we are just playing around with writing our own to understand that.

The first exercise is to implement a way to deep copy an instance of our vector. To do this we will use the *copy-and-swap idiom*. We need two things to achieve this:

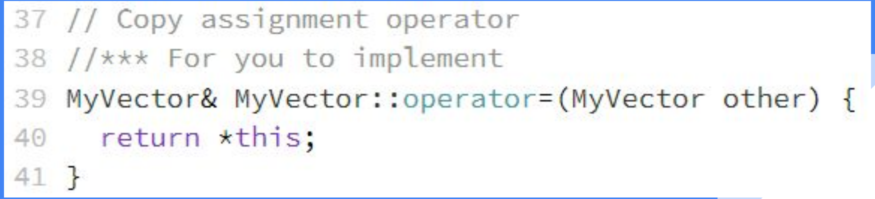
1. A *copy constructor* for our vector class.
2. An `operator=` overload.

For 1. go to line 30 and define a constructor using a list initialiser, like we mentioned above. You can have functionality in the final `{}`



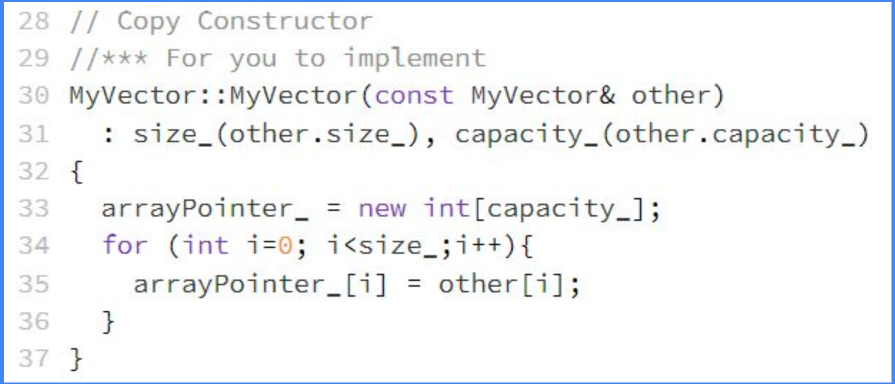
```
28 // Copy Constructor
29 //*** For you to implement
30 MyVector::MyVector(const MyVector& other) {
31 }
```

For 2. go to line 39 and use `std::swap()` to swap member variables with the cloned class instance *other*.

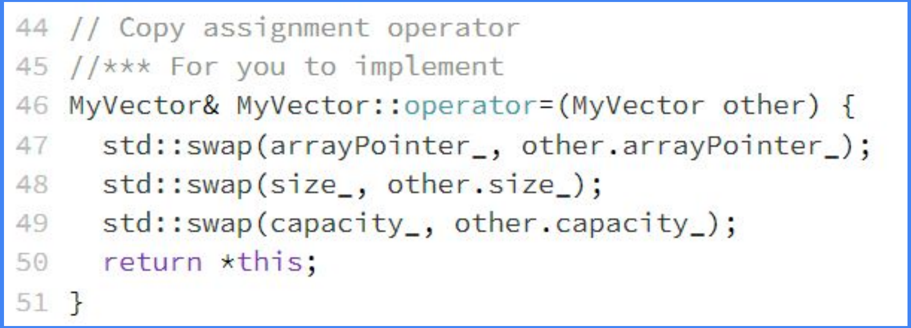


```
37 // Copy assignment operator
38 //*** For you to implement
39 MyVector& MyVector::operator=(MyVector other) {
40   return *this;
41 }
```

# Return to MyVector



```
28 // Copy Constructor
29 //*** For you to implement
30 MyVector::MyVector(const MyVector& other)
31   : size_(other.size_), capacity_(other.capacity_)
32 {
33   arrayPointer_ = new int[capacity_];
34   for (int i=0; i<size_;i++){
35     arrayPointer_[i] = other[i];
36   }
37 }
```



```
44 // Copy assignment operator
45 //*** For you to implement
46 MyVector& MyVector::operator=(MyVector other) {
47   std::swap(arrayPointer_, other.arrayPointer_);
48   std::swap(size_, other.size_);
49   std::swap(capacity_, other.capacity_);
50   return *this;
51 }
```

For 1. go to line 30 and define a constructor using a list initialiser, like we mentioned above. You can have functionality in the final `{ }`

For 2. go to line 39 and use `std::swap()` to swap member variables with the cloned class instance *other.*

# Return to MyVector

The final exercise for this week is to make a custom vector class using templating.
You need to define the following member functions:

- `MyVector<T>::MyVector(int n)` *constructor given initial size*
- `MyVector<T>::MyVector(const MyVector<T>& other)` *from prev*
- `MyVector<T>::~MyVector()` *destructor*
- `MyVector<T>& MyVector<T>::operator=(MyVector<T> other)` f.prev
- `void MyVector<T>::push_back(T val)` *adds* `val` *to the vector*
- `void MyVector<T>::pop_back()` *removes last element*

You have member variables:

- `int size_,` `int capacity_,` `T* arrayPointer_`

Remember! You are handling a C-array, of fixed length, and it is your job to write your functions to seamlessly handle the resizing & rebuilding of this array.