Array:
Most common container in C++

```
const int size = 9;
int data[size] = {1, 1, 2, 3, 5, 8, 13, 21, 34};
for (int* p = data; p != data[size]; ++p) {
        cout << *p << ' ';
}
cout << endl;
```

OUTPUT:
1 1 2 3 5 8 13 21 34

REMEMBER: elements start at 0; data[size] is one past the last element!
REMEMBER: data holds address of the beginning of the array

Downsides of the array:

1. no out of bounds check
2. passing an array to a function usually requires passing its size in too
3. need to know an array's size when writing a program
4. can't add or remove elements

Putting arrays on the heap:

1. can wait to decide how large they need to be
2. if we need more space we can make a new array and copy items while adding new ones

Linked Lists:

```
struct Node {
        Node(int data = 0, Node* prev = nullptr, Node* next = nullptr) : data(data), prev(prev),
                                                                         next(next) {}
        int data;
        Node* prev;
        Node* next;
};
```

For copy control:

- have to delete each node (iterate over each and delete) in destructor
- have to deep copy each node

Functions written for linked lists:

- listClear, listRemoveHead, listDisplay, listLength, listRemoveHead, listAddHead, listRemove, listDuplicate

Begin and end:
Every container has a begin method that points to first element and end method that points just AFTER the last element

```
Iterator begin() { return Iterator(data); }
Iterator end() { return Iterator(data + theSize); }

Iterator begin() const { return Iterator(data); }
Itereator end() const { return Iterator(data + theSize); }
```

Iterators:
Generalization of pointers

- each type of container has its own iterator
- important: vectors, lists, arrays, etc don't have one for every type; have to declare that they're holding too in the type name

```
vector<int>::iterator iter;
```

```
for (vector<int>::iterator iter = vec.begin(); iter != vec.end(); iter++) {
        cout << *iter << ' ';
}
cout << endl;
```

For iterators of const objects: const_iterator

Iterators have dereference, increment, and equal/not equal methods

- usually also a constructor and a private pointer attribute

```
class Iterator{
	friend bool operator!=(const Iterator& lhs, const Iterator& rhs) {
		return lhs.where != rhs.where;
	}
public:
	Iterator (int* p ) : where(p) {}
	Iterator& operator++() {
		++where; //pointer arithmetic
		return *this;
	}
	int operator*() const {
		return *where;
	}
	int& operator*() {
		return *where;
	}
private:
	int* where;
};
```

NOTE: if const_iterator, dereference and increment methods return const
NOTE: replace all iterator with const_iterator

```
const Const_Iterator& operator++() {
	++where;
	return *this;
}
const int& operator*() {
	return *where;
}
private:
	const int* where;
};
```

Lambda expressions:
Used in place of a function; for when we only need to run a function once and it'd be wasteful to write an entire function?

Syntax: [] () -> void {} ();

1. [] – can be empty, can put a variable inside to be captured by the lamba expression
2. () – for parameters
3. specify return type with ->; don't have to specify return type though

4. {} – where the actual code goes
5. (); – the function call part with or without parameters

```
[] {} (); //most basic
  [] () -> void {} ();
```

```
[] (int n) { return n%2 == 0; } (4);
myfind_if(li.begin(), li.end(), [] (int n) { return n%2 == 0} );
```

Functor:
A class with only one method inside; used like a function

- the INSTANCE of the class that is created is called a functor

```
class IsEven {
public:
        bool operator() (int n) { return n%2 == 0; }
};

list<int> li{1,2,3,4,5,6};
IsEven myFunctor;
myFunctor(3);
myfind_if(li.begin(), li.end(), myFunctor);

OR

myfind_if(li.begin(), li.end(), IsEven());
```

```
class IsMultiple {
public:
        IsMultiple(int factor) : factor(factor) {}
        bool operator() (int n) { return n%factor == 0;}
private:
        int factor;
};

myfind_if(li.begin(), li.end(), isMultiple(2));
```

Auto:
Type inferencing; lets the compiler figure out the type for us; c++ knows the type

```
auto whereList = myfind(lc.begin(), lc.end(), 'S');
auto var = 17;
auto p = new ClassName(17);
```

Recursion:
A function that calls itself

- needs a base case and recursive step(s)

Examples:
Towers of Hanoi:

```
void towers(int n, char start, char target, char spare) {
        if (n>0) {
                towers(n-1, start, spare, target);
                towers(n-1, spare, target, start);
        }
}
```

Function to calculate the power of a number to another number:

```
int powerRec(int a, int b) {
        if (b==0) { return 1; }
        else if (b==1) { return a; }
        else {
                res = powerRec(a, b-1);
                return a*res;
        }
}
```

Function to determine if a positive integer has an even number of ones (ex: 11 vs 14):

```
int ones(int n) {
        if (n<10 && n==1) { return 1; }
        if (n<10 && n!= 1) { return 0; }
        if (n%10 == 1) { return ones(n/10) + 1; }
        else { return ones(n/10); }
}
bool checkOnes(int n) {
        int res = ones(n);
        if (res%2 == 0) { return true; }
        else { return false; }
}
```

Templates:
Used for general types; when we want to substitute different types in for the same thing

template <class VariableName>
template <typename VariableName>

Templated variables only work for the things immediately after it!!

Standard Template Library (STL):
container class- designed to contain/hold something of a given type
ex: stacks, queues, lists, sets, vectors

STL provides a set of container classes and generic algorithms to work with different containers

- #include <algorithm>
- max, min, swap

Many generic algorithms compare between two items in a container/in separate containers

- MUST overload operator< and operator==
- copy constructor, default constructor, assignment operator – either use system provided or overload them

Containers:

- main containers: stack, queue, vector, deque, list, set, multiset, map, multimap

Common features:
*contType = container type (like string s, int v, vector l)

contType c;	default constructor
contType c1(c2);	copy constructor
contType c(begin, end);	constructor using begin and end iterators (from another container) to specify which elements to be included in c
c.size();	returns actual number of elements
c.empty();	returns true if c.size()==0, else false
c.begin();	iterator that points to beginning of c
c.end();	iterator that points AFTER last element of c
c1 == c2;	tests if equal (same size and elements equal)
c1 < c2;	tests if less (lexicographically - alphabetically)
c.clear();	removes all, container empty

Stack:

- can only access top of stack (add to top, pop from top, access the top item)
- Last In First Out

Vector:

- similar to array but can grow
- makes copies when adding to/removing items

Generic Algorithms:
Will work with any container that supports iterators

- sterling talks about 3 on his website: count, count_if, copy

count: counts all elements in a range that are equal to a specific value

- takes 3 parameters:
    1. iterator to beginning of range
    2. iterator to the end of the range (past last element)
    3. value you're trying to count

```
const int size = 9;
int data[size] = {1, 1, 2, 3, 5, 8, 13, 21, 34};
for (int* p = data; p != data[size]; ++p) {
        cout << *p << ' ';
}
cout << endl;

int countOfOnes = count(data, &data[size], 1);
cout << countOfOnes << endl;
```

count_if: counts all elements in a specific range that have a specific property

- takes 3 parameters:
    1. iterator to beginning of range
    2. iterator to end of the range (past the last element)
    3. boolean function that defines the property

```
bool isEven (int n) {
        return n%2 == 0;
}

int countOfEven = count_if(data, &data[size], isEven);
cout << countOfEven << endl;
```

NOTE: using the name of a function as an argument

copy: copies the elements in a specific range

- takes 3 parameters:
    1. iterator pointing to beginning of range
    2. iterator to end of the range (past the last element)
    3. iterator pointing to beginning of the target range

```
vector<int> v1{1, 2, 3, 4, 5};
vector<int> v2{6, 7, 8, 9, 10};
copy(v1.begin(), v1.end(), v2.begin());
```

NOTE: copy DOES NOT create space in the target; it already has to have the required space!

Exceptions and Assertions:
Allows us to handle special cases that could cause a program to crash

- allows for recovery!!
- doesn't immediately terminate the program
- most exceptions don't have a standard constructor
- usually requires string argument that will be returned by *.what()* method

```
#include <stdexcept>
#include <exception>
```

Assert:
Enforcing a condition; a "let them know something is wrong and terminate right now" approach

```
void bar(int n) {
        assert (n<200);
        if (n<200) { //code }
};
```

Can also add a display message:

```
assert(n<200 && "can't be larger than 200");
 assert(("can't be larger than 200", n<200));
```

Throw:
Can throw anything (number, floats, string, etc)

- most common to throw an instance of an exception class
- provided classes all inherent from base class: exception

Throwing an exception will immediately end function call!! (like a return with no return value)

Try and Catch:
Can have multiple catch "clauses"; ORDER matters

- should go from most specific to least specific

```
int main() {
        try {
                vector<int> v;
                //code to fill vector
                v.clear();
                //cout << v[0] << endl; //undefined behavior, will get random value at that
                                                                memory point
                cout << v.at(0); //out_of_range exception thrown
        }
        catch (out_of_range e) {
                cout << "caught an out_of_range exception" << endl;
        }
        // if exception handled properly, code will continue here
        cout << "Done" << endl;
}
```

"Catch-all" exception: for when we don't know what kind of exception might be thrown

- will catch anything that is thrown
- will not know what type of exception is thrown

```
catch (…) { //code here }
```

```
void foo(int n) {
        try { bar(n); }
        catch (out_of_range oor) {
                cerr << "some message\n";
                throw;
        }
}

int main() {
        try { foo(300); }
        catch (out_of_range oor) { cerr << "caught an out_of_range: " << oor.what << endl; }
        catch (exception& ex) { cerr << "caught an exception: " << ex.what() << endl;
        catch (…) { cerr << "caught something …." << endl; }
}
```

If a general exception is used and passed in by value, slicing occurs, all info is lost about it.

- will get a std::exception

If it is passed in by reference, polymorphism occurs, the info derived from exception is captured and accessed → we will get the error message they intended us to get

```
//generic
template <typename Fred> // only apply to thing immediately after it
class Vector {
public:
	Vector() : data(nullptr), theSize(0), theCapacity(0) {}

	explicit Vector (size_t howMany, int val = 0 ){
		theSize = howMany;
		theCapacity = howMany;
		data = new int[howMany];
		for (size_t i =0; i < theSize; ++i) {
			data[i] = val;
		}
	 }

	// Copy control
	~Vector () { delete[] data; }

	Vector(const Vector& rhs) {
		//code here
	}

	 // Square brackets
	int operator[] (size_t index) const {
		return data[index];
	}

	int& operator[] (size_t index) {
		return data[index];
	}

	Iterator begin () { return Iterator(data); }
	Iterator end () { return Iterator(data + theSize); }

	Iterator begin () const { return Iterator(data); }
	Iterator end () const { return Iterator(data + theSize); }

private:
	Fred* data;
	size_t theSize;
	size_t theCapacity;
};
```

PRACTICE:

Iterator for linked list:

```
class iterator {
	Node* thing;
public:
	iterator(Node* thing) : thing(thing) {}
	T operator*() { return thing->data; }
	iterator& operator++() {
		thing = thing->next;
		return *this;
	}
	bool operator!=(const iterator& rhs) {
		return thing->data != rhs.data;
	}
};
```

Iterator for vectors:

```
class iterator {
	T* thing;
public:
	iterator(T* thing) : thing(thing) {}
	T& operator*() { return *thing; }
	iterator& operator++() {
		thing = thing+1;
		return *this;
	}
	bool operator!=(const iterator& rhs) {
		return rhs.thing != thing;
	}
};
```

A function that multiplies everything in a vector by 2 using iterators:

```
void multiply(vector<int> vec) {
	for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
		*it *= 2;
	}
}
```

Given:

```
bool isCapital(char x);
class Letter {
public:
        Letter(char l) : theLetter(l) {}
private:
        char theLetter;
};
int main() {
        Letter myLetter('T');
        if (myLetter) {
                cout << "I have a capital latter";
        }
}
```

OUTPUT:

Will not compile unless a bool operator is written

```
explicit operator bool() {
        return isCapital(theLetter);
}
```

Given:

```
template <typename T, typename E>
class Vector {
private:
        T* array;
public:
};
```

Write the square bracket operator

```
T operator[] (size_t index) const {
        return *(array+index)
}
```

```
void foo(int x) {
        if (x>1) {
                cout << x << ':';
                foo(x/2);
                for (int i = x; i > 0; i--) {
                        cout << ':';
                }
                cout << ',';
                foo(x/2);
        }
}
```

OUTPUT:

4:2:::,::::,2:::,