

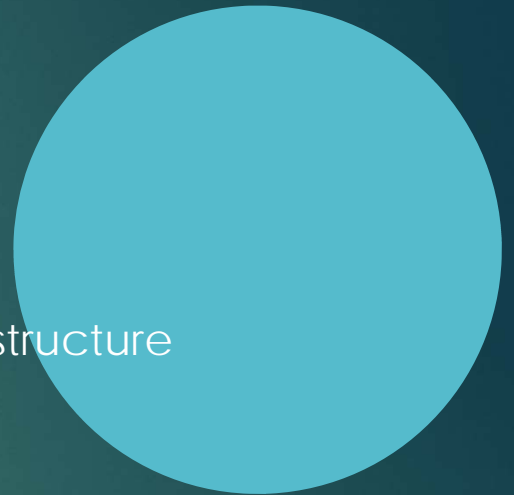
Algorithm & Container

IN MODERN C++



Containers

- ▶ General purpose template based data structure
- ▶ Organize data following different criteria
- ▶ Almost always equally or more efficient than user written structure
- ▶ Almost always safer than user written structure
- ▶ Safe By Design



Type of containers

- ▶ **vector** memory managed unordered array of data
- ▶ **map** key-value based container. Higher insert operation time but lower data finding time respect to **vector** or **list**.
- ▶ **list** collection of non-contiguous memory data.
- ▶ **set** holder of unique objects.
- ▶ **array** std version of the standard array.



Storing classes in containers

- ▶ *Class must be CopyAssignable and CopyConstructible:*
 - ▶ *If copy constructor or assignment operator is not provided C++ will provide one (uses member copying)*
- ▶ *If no constructors provided*
 - ▶ *Empty default constructor provided by C++*
- ▶ *Keys in associative(e.g. std::map) also need to be comparable*
 - ▶ *Provide an operator< overload*
 - ▶ *Or provide a comparison function as template parameter*

```
class Contact {  
public:  
    ~Contact(); // destructor  
    Contact(const Contact &other); // copy constructor  
    // assignment operator  
    Contact &operator=(const Contact &other);  
};
```

Changes since c++ 98

- ▶ *Initializer lists*

```
std::vector<std::string> vs = { "Hello", " ", " ", "World!", "\n" };
```

- ▶ *Rvalue push_back*

```
std::vector<std::pair<std::string, int>> vp;  
std::string s;  
int i;  
while (cin>>s>>i) {  
    // a std::pair is constructed and then moved inside the container  
    vp.push_back(s,i);  
}
```

- ▶ *emplace_back*

```
std::vector<std::pair<std::string, int>> vp;  
std::string s;  
int i;  
while (cin>>s>>i) {  
    // a std::pair is constructed directly inside the container  
    vp.emplace_back(s,i);  
}
```

- ▶ *Unordered associative container($O(1)$ lookup, insertion and removal)*

Std::array

- ▶ Standard container API, usable in algorithms
- ▶ Wrapper around C-style arrays
 - ▶ Constant size Data
 - ▶ Has automatic storage, no dynamic allocation

```
// Note: double-braces required in C++11
std::array<int, 10> test{ 1, 2, 4, 3, 7, 6, 9, 8, 4 };
std::sort(test.rbegin(), test.rend());
for (int value : test)
    std::cout << value << std::endl;
```

Modernization example

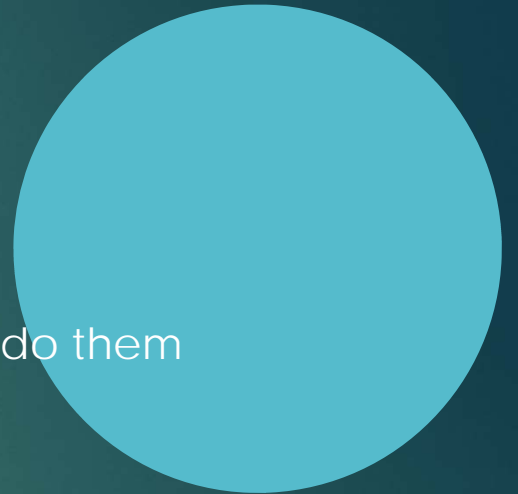
```
std::vector<std::string> v;  
v.push_back("one");  
v.push_back("two");  
v.push_back("three");  
std::string item1 = v[1]; // "two"  
for (int i = 0; i < v.size(); i++)  
{  
    std::string &item = v.at(i);  
    item += "_suffix";  
}  
item1 = v[1]; // "two_suffix"
```



```
std::vector<std::string> v = {"one", "two", "three"};  
auto item1 = v[1]; // "two"  
for (auto& item : v) {  
    item += "_suffix";  
}  
item1 = v[1]; // "two_suffix"
```

Algorithm

Don't do things your way if there is an algorithm that do them
do not reinvent the wheel



Advantage

- ▶ Algorithm are out of the box **safe by design** operation that can be applied on containers.
- ▶ Self commenting code
- ▶ Error free, can be excluded in debugging session



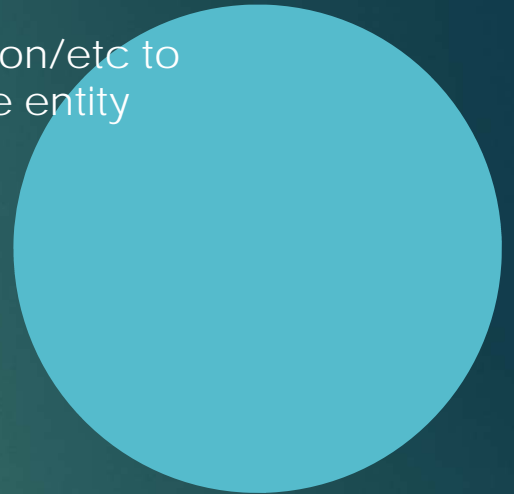
Basic structure

Autocommenting name of the algorithm

`Algorithm_name(from, to, predicate)`

Action/comparison/etc to perform. Callable entity

Range of application via iterators



Common algorithm

all_of/none_of
any_of
for_each
find/find_if/find_if_not/find_first_of
count/count_if
copy/copy_if
swap
remove/remove_if
replace/replace_if
generate
transform
unique
...
Check it out:
<https://en.cppreference.com/w/cpp/algorithm>



Example 1

```
std::vector<int> collection = {3, 6, 12, 6, 9, 12};

// Are all numbers divisible by 3?
bool divby3 = std::all_of(begin(collection), end(collection), [](int x) { return x % 3 == 0; });
// divby3 equals true, because all numbers are divisible by 3

// Is any number divisible by 2?
bool divby2 = std::any_of(begin(collection), end(collection), [](int x) { return x % 2 == 0; });
// divby2 equals true because 6, 12 divisible by 2

// Is no number divisible by 6?
bool divby6 = std::none_of(begin(collection), end(collection), [](int x) { return x % 6 == 0; });
// divby6 equals false because 6, 12 divisible by 6
```

Example 2

```
// C++98
std::vector<int> collection;
collection.push_back(2); collection.push_back(4); collection.push_back(4);
collection.push_back(1); collection.push_back(1); collection.push_back(3);
for(int i = 0; i<collection.size(); i++) {
    collection[i] += 26;
}
```

```
// C++11
std::vector<int> collection = {2,4,4,1,1,3};
for(auto& el:collection) {
    el +=26;
}
```

```
// C++11, using algorithm
std::vector<int> collection = {2,4,4,1,1,3,9};
std::for_each(begin(collection), end(collection), [] (int &x){x += 26;});
```

Example 3

Find max element

C++98

```
85 // Find highest earner and print his name
86 // TODO: Use an algorithm to simplify this,
87 std::vector<Employee>::iterator highestEarner = employees.end();
88 std::vector<Employee>::iterator it2 = employees.begin();
89 while (it2 != employees.end()) {
90     const Employee e = *(++it2);
91     if (highestEarner == employees.end() ||
92         e.salary > highestEarner->salary)
93         highestEarner = it2;
94 }
95
```

C++11

```
highestEarner = std::max_element(employees.begin(), employees.end(), [](const auto& a, const auto& b){ return a.salary > b.salary});
```

Transform

```
std::string s("hello");
std::transform(s.begin(), s.end(), s.begin(),
    [](unsigned char c) -> unsigned char { return std::toupper(c); });
```

Conclusion

Do you think you need a for loop to do stuff in a container?

Think again.
Use an algorithm

