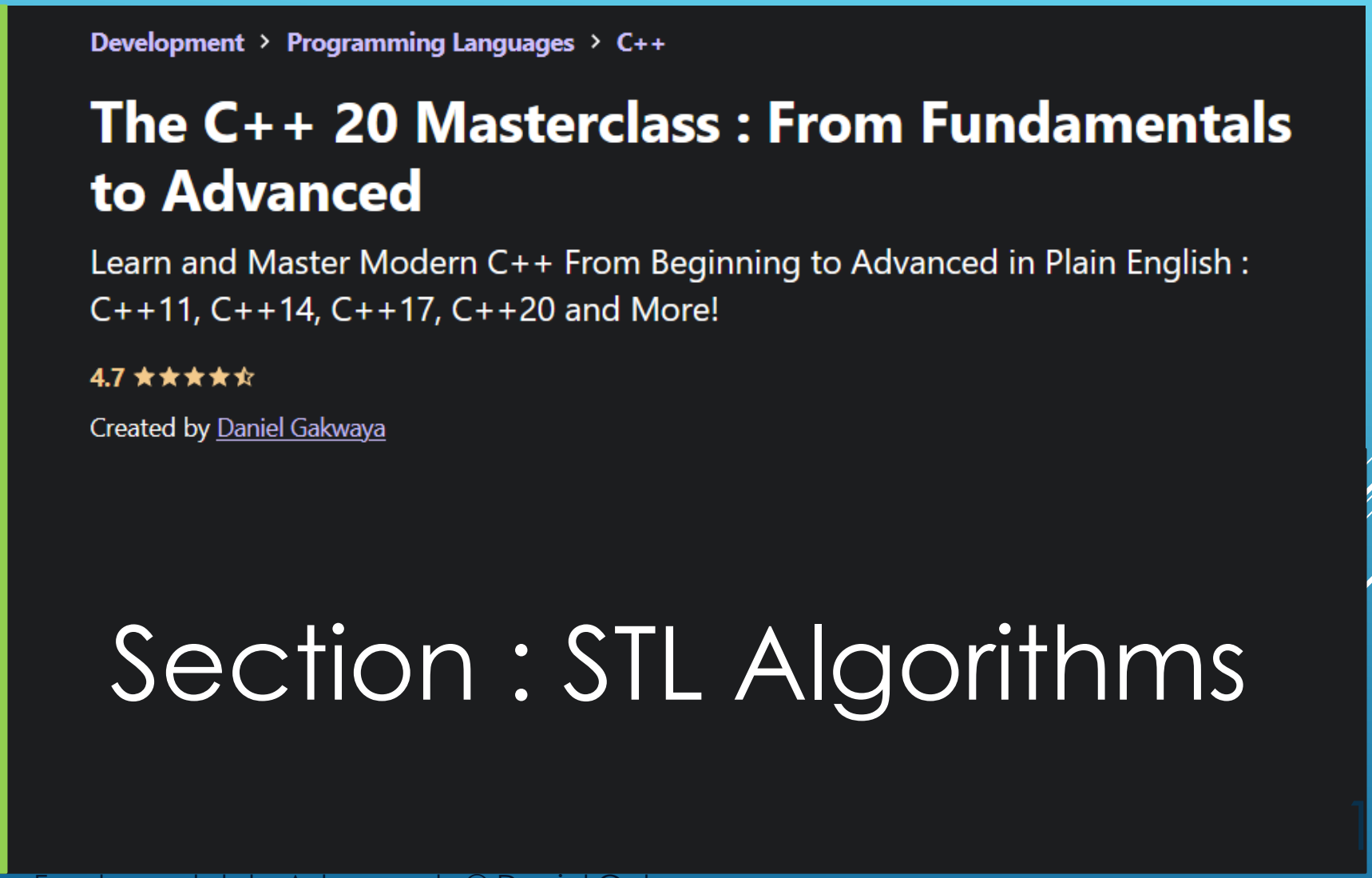
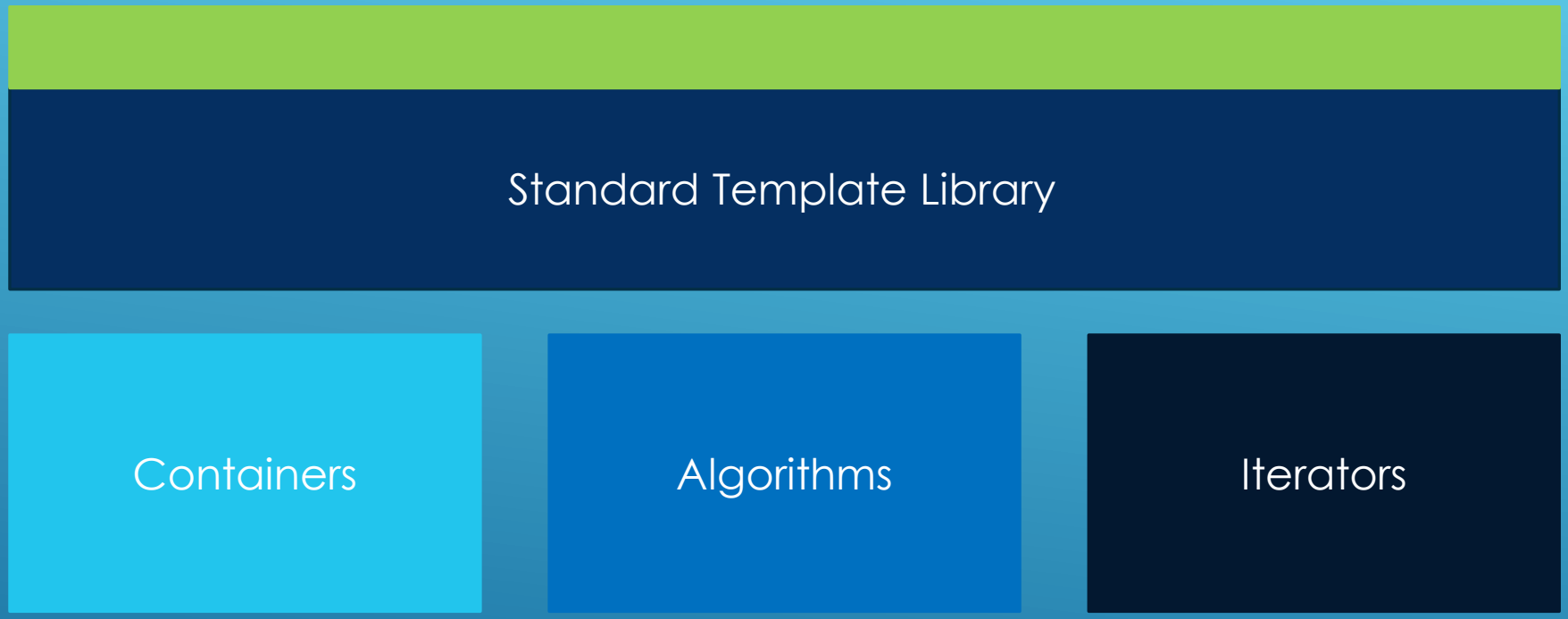
Slides



Section : STL Algorithms

Slide intentionally left empty

STL Algorithms : Introduction


Standard Template Library
Containers
Algorithms
Iterators


Legacy algorithms
Work on iterator pairs
Range algorithms
Work on containers directly

Slide intentionally left empty

std::all_of , std::any_of & std::none_of

std::all_of

```
//std::vector<int> collection{2,6,8,40,64,70};
//std::set<int> collection{2,6,8,40,64,70};
int collection[] {2,6,8,40,64,70};

//std::all_of , lambda function predicate
if (std::all_of(std::begin(collection), std::end(collection), [](int i){ return i % 2 == 0; })) {
    std::cout << "(std::all_of) :  All numbers in collection are even" << std::endl;
}else{
    std::cout << "(std::all_of) : Not all numbers in collection are even" << std::endl;
}
```

std::any_of

```
//std::any_of ,functor as predicate
class DivisibleBy
{
    private :
        const int d;
    public :
        DivisibleBy(int n) : d(n) {}
        bool operator()(int n) const { return n % d == 0; }
};

if (std::any_of(std::begin(collection),std::end(collection), DivisibleBy(7))) {
    std::cout << "(std::any_of) : At least one number is divisible by 7" << std::endl;
}else{
    std::cout << "(std::any_of) : None of the numbers is divisible by 7" << std::endl;
}
```

std::none_of

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

int main(int argc, char **argv)
{
    //std::vector<int> collection{2,6,8,40,64,70};
    //std::set<int> collection{2,6,8,40,64,70};
    int collection[] {2,6,8,40,64,70};


    //std::none_of , function pointer as predicate
    if (std::none_of(std::begin(collection), std::end(collection), is_odd)) {
        std::cout << "(std::none_of) :  None of the numbers is odd" << std::endl;
    }else{
        std::cout << "(std::none_of) : At least one number is odd" << std::endl;
    }
    return 0;
}
```

Slide intentionally left empty

std::for_each()

std::for_each

```
//std::vector<int> nums{3, 4, 2, 8, 15, 267};
//int nums[]{3, 4, 2, 8, 15, 267};
std::set<int> nums{3, 4, 2, 8, 15, 267};


auto print = [](const int& n) {
    std::cout << " " << n;
};

//Print each elt in the collection : lambda function predicate
std::for_each(std::begin(nums), std::end(nums), print);
std::cout << std::endl;

std::cout << "--------------------------------" << std::endl;

//Predicate that modifies elements in place
std::for_each(std::begin(nums), std::end(nums), [](int& n){ n++; });

//Print
std::for_each(std::begin(nums), std::end(nums), print);
std::cout << std::endl;
```

std::for_each

```
struct Sum
{
    void operator()(int n) { sum += n; }
    int sum{0};
};
int main(int argc, char **argv)
{
    //std::vector<int> nums{3, 4, 2, 8, 15, 267};
    //int nums[]{3, 4, 2, 8, 15, 267};
    std::set<int> nums{3, 4, 2, 8, 15, 267};

    //Capturing result through stateful functor that's returned
    // calls Sum::operator() for each number
    Sum s;
    s = std::for_each(std::begin(nums), std::end(nums), s);
    std::cout << "result : " << s.sum << std::endl;

    //Using a lambda that captures a local variables by ref and modifies it.
    int our_result{0};
    std::for_each(std::begin(nums), std::end(nums),[&our_result](int n) { our_result += n; });
    std::cout << "result : " << our_result << std::endl;
    return 0;
}
```

Slide intentionally left empty

std::max_element & std::min_element

std::max_element() & std::min_element()

```
//std::vector<int> v {3,400,51,6,7,23,56,71};
int v[] {3,400,51,6,7,23,56,71};
//std::list<int> v {3,400,51,6,7,23,56,71};

//max_elt and min_elt return an iterator to the found elt
auto result = std::max_element(std::begin(v), std::end(v));
std::cout << "max element is : " << *result << std::endl;

result = std::min_element(std::begin(v),std::end(v));
std::cout << "min element is : " << *result << std::endl;
```

std::max_element() & std::min_element()

```
//std::vector<int> v {3,400,51,6,7,23,56,71};
int v[] {3,400,51,6,7,23,56,71};
//std::list<int> v {3,400,51,6,7,23,56,71};

//Distances : closest and furthest
int number_to_find {67};

auto distance = [number_to_find](int x, int y){
    return (std::abs(x-number_to_find) < std::abs(y-number_to_find));
};

//Finding the closest
result = std::min_element(std::begin(v),std::end(v),distance);
std::cout << *result << " is closest to " << number_to_find << std::endl;

//Finding the furthest
result = std::max_element(std::begin(v),std::end(v),distance);
std::cout << *result << " is furthest from  " << number_to_find << std::endl;

//Capturing min and max in a pair object with the auto syntax
const auto[near,far] = std::minmax_element(std::begin(v),std::end(v),distance);
std::cout << *near << " is closest to " << number_to_find << std::endl;
std::cout << *far << " is furthest from  " << number_to_find << std::endl;
```

Slide intentionally left empty

std::find() & std::find_if()

std::find()& std::find_if()

```
int n = 23;
//int n = 24;

std::vector<int> collection{14, 24,7, 8, 98, 11};

auto result = std::find(std::begin(collection), std::end(collection), n);

if (result != std::end(collection)) {
    std::cout << "collection contains: " << n << std::endl;
} else {
    std::cout << "collection does not contain: " << n << std::endl;
}
```

std::find()& std::find_if()

```
std::vector<int> collection{14, 24,7, 8, 98, 11};

auto odd = [](int x){
    if( (x%2) != 0)
        return true;
    return false;
};

auto odd_n_position = std::find_if(std::begin(collection),std::end(collection),odd);

if (odd_n_position != std::end(collection)) {
    std::cout << "collection contains at least one odd number : " << *odd_n_position  << std::endl;
} else {
    std::cout << "collection does not contain any odd number" << std::endl;
}
```

Slide intentionally left empty

std::copy

std::copy()

```
//std::vector<int> source {1,2,3,4,5,6,7,8,9};
int source[] {1,2,3,4,5,6,7,8,9};

std::vector<int> dest {15,21,12,53,30,40};

std::cout << "source : ";
print_collection(source);

std::cout << "dest : ";
print_collection(dest);

//Copy from source to dest
//Copy elements from source in the range [std::begin(source),std::begin(source) + 4 )
//to dest, starging from iterator std::begin(dest)
//Make sure you are copying from valid ranges either in dest or source.
std::copy(std::begin(source),std::begin(source) + 4,std::begin(dest));

std::cout << "source(after copy) : ";
print_collection(source);

std::cout << "dest(after copy) : ";
print_collection(dest);
```

std::copy()

```
//std::vector<int> source {1,2,3,4,5,6,7,8,9};
int source[] {1,2,3,4,5,6,7,8,9};

std::vector<int> dest1{100,200,300,400,500,600};

std::cout << "source : ";
print_collection(source);

std::cout << "dest1 : " ;
print_collection(dest1);

auto odd = [](int n){
    return ((n%2)!=0);
};
//If there are more elements in source than the space available in dest,
//surplus elements will just be ignored.
std::copy_if(std::begin(source),std::end(source),std::begin(dest1),odd);

std::cout << "source(after copy) : ";
print_collection(source);

std::cout << "dest1(after copy) : " ;
print_collection(dest1);
```

Slide intentionally left empty

std::sort()

std::sort()

```
//Directly without predicate
std::vector<int> collection = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

std::cout << "collection(unsorted) : ";
print_collection(collection);

std::sort(std::begin(collection),std::end(collection));

std::cout << "collection(sorted) : ";
print_collection(collection);
```

std::sort()

```
//With custom compare function
collection = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

std::cout << "collection(unsorted) : ";
print_collection(collection);

std::sort(std::begin(collection),std::end(collection),std::less<int>());
//std::sort(std::begin(collection),std::end(collection),std::greater<int>());
//std::sort(std::begin(collection),std::end(collection),[](int x, int y){return x < y;});

std::cout << "collection(sorted) : ";
print_collection(collection);
```

std::sort()

```
//Sorting collections of custom items
std::vector<Book> books {Book(1734,"Cooking Food"),
                         Book(2000,"Building Computers"),Book(1995,"Farming for Beginners")};


std::cout << "books(before sort) : " << std::endl;
print_collection(books);

//std::sort(std::begin(books),std::end(books));
//std::sort(std::begin(books),std::end(books),std::less<Book>());
//std::sort(std::begin(books),std::end(books),std::greater<Book>());
                                                    // Will look for > operator.
                                                    // Put it in and make the compiler
                                                    //happy
auto cmp = [](const Book& book1, const Book& book2){
    return (book1.m_year < book2.m_year);
};
std::sort(std::begin(books),std::end(books),cmp);

std::cout << "books(after sort) : " << std::endl;
print_collection(books);
```

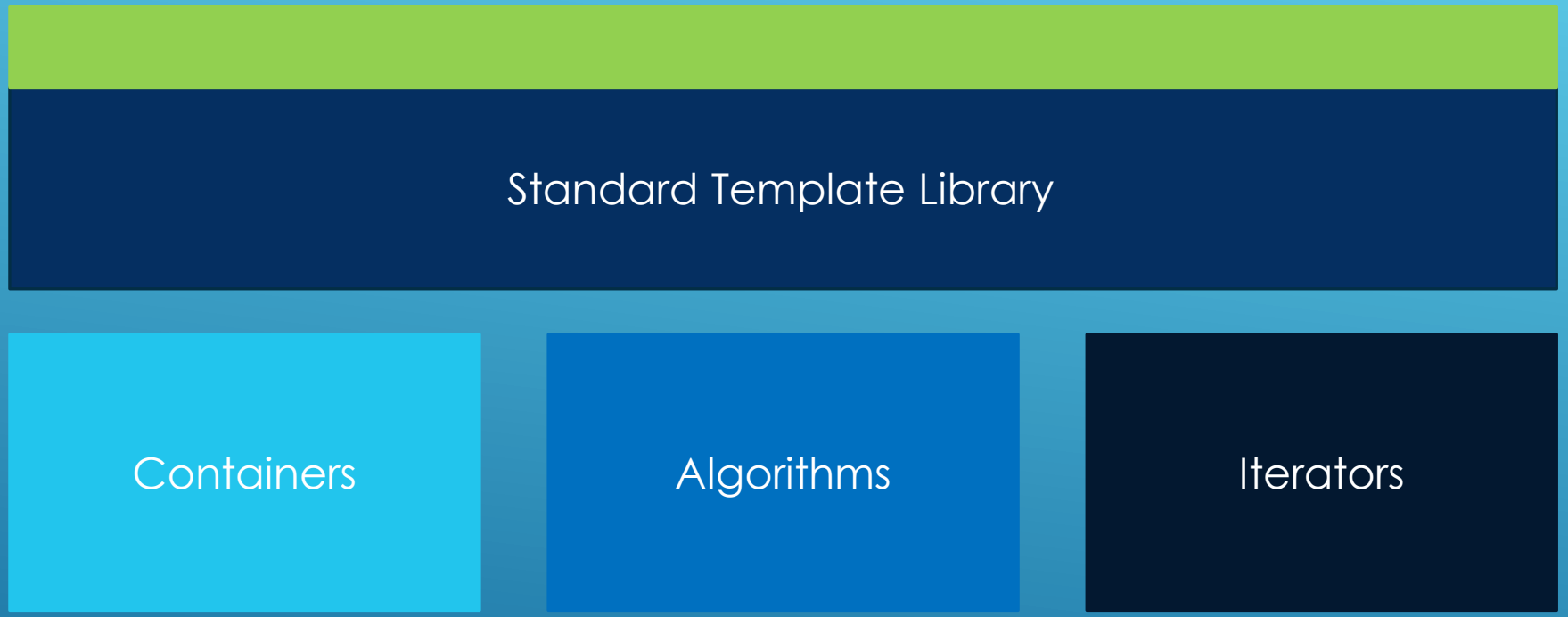
Slide intentionally left empty

std::transform()

```cpp
//Original collection
std::vector<int> input = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3,11,45,6,23};
std::vector<int> output{11,22,33};
std::cout << "output size : " << output.size() << std::endl;
std::cout << "output capacity : " << output.capacity() << std::endl;

print_collection(input);
print_collection(output);

//Uses whatever space is there, doesn't extend the capacity
std::transform(input.begin(),input.end(),output.begin(),[](int n){return n*2;});

//std::back_inserter_iterator : appends to the back, extends capacity if necessary.
//std::transform(input.begin(),input.end(),std::back_inserter(output),[](int n){return n*2;});

print_collection(output);
std::cout << "output size : " << output.size() << std::endl;
std::cout << "output capacity : " << output.capacity() << std::endl;
```

Slide intentionally left empty

STL Algorithms : Summary


Standard Template Library
Containers
Algorithms
Iterators


Legacy algorithms
Work on iterator pairs
Range algorithms
Work on containers directly

Slide intentionally left empty