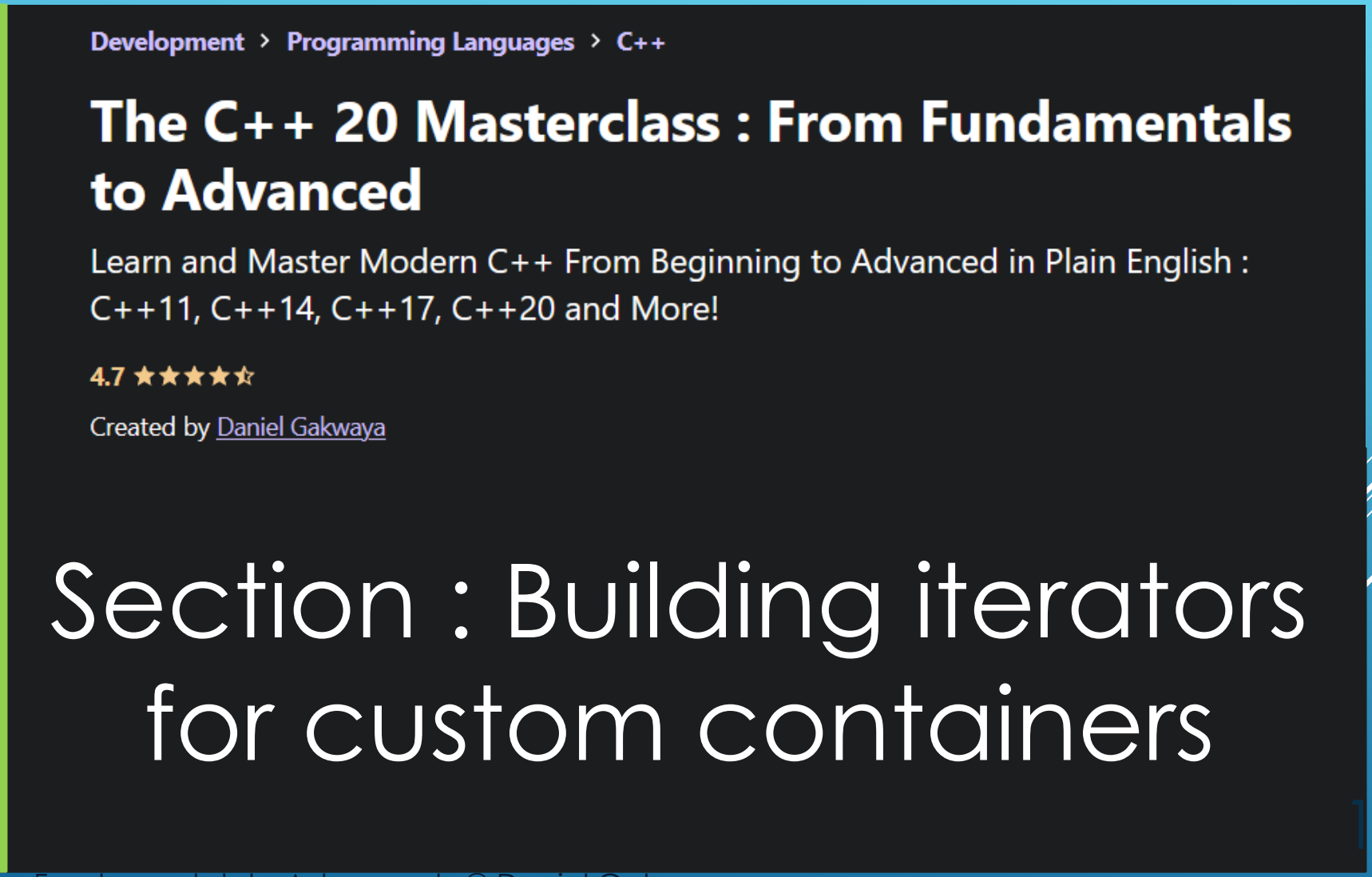
Slides



Section : Building iterators for custom containers

Slide intentionally left empty

Building iterators for custom containers

Iterators for BoxContainer

- We have played with stl algorithms for quite a while now, and seen the benefits of buil in stl containers
- Iterators are the glue that ties containers and algorithms together
- Sometimes, you want your own containers to be able to work with built in stl algorithms, like std::sort, std::find, std::fill,...
- That just opens a new world of flexibility for you container
- We want BoxContainer to support iterators, so that it can use range based for loops for example, and be usable with stl algorithms

Iterators for BoxContainer

```
BoxContainer<int> box1;
box1.add(8);
box1.add(1);
box1.add(4);
box1.add(2);

std::cout << "box1 : " << box1 << std::endl;

//Regular loop
std::cout << "Regular loop : " ;
for(size_t i{} ; i < box1.size() ; ++i){
    std::cout << box1.get_item(i) << " " ;
}
std::cout << std::endl;

//Range based for loop
std::cout << "Range based for  loop : " ;
for(auto i : box1){
    std::cout << i << " " ;
}
std::cout << std::endl;
```

Iterators for BoxContainer

```
std::fill(box1.begin(),box1.end(),5);
std::cout << "box1 : " << box1 << std::endl;

std::sort(box1.begin(),box1.end());
std::cout << "box1 : " <<box1 << std::endl;

int n{9};
auto result1 = std::find(std::begin(box1), std::end(box1), n);

if (result1 != std::end(box1)) {
    std::cout << "box1 contains: " << n << std::endl;
} else {
    std::cout << "box1 does not contain: " << n << std::endl;
}

std::ranges::fill(box1,6);

for(auto i : std::views::take(box1,3)){
    std::cout << i  << " ";
}
std::cout << std::endl;
```

Iterators for BoxContainer

- Input iterators
- Output iterator
- forward iterator
- bidirectional iterator
- random access iterator
- contiguous iterator (C++ 20)

Iterators for BoxContainer

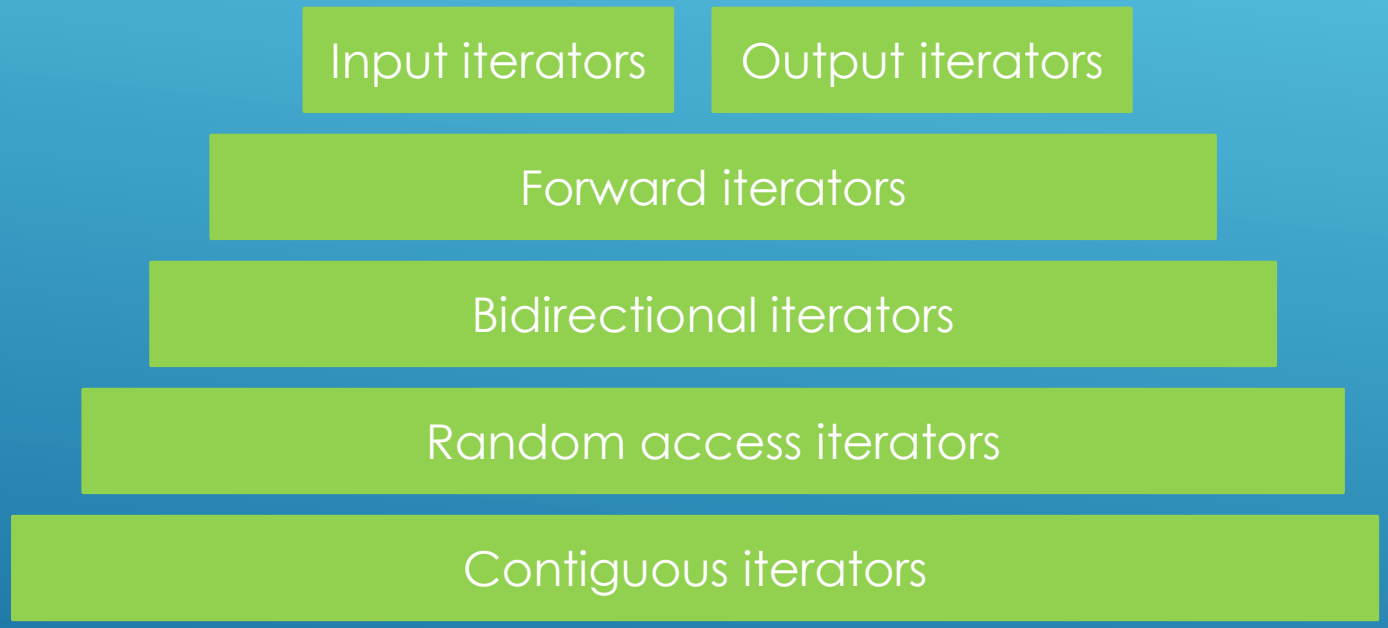
- Iterator types are hierarchical : a forward iterators is also an input iterator, a bidirectional iterator is also a forward iterator, a random access iterator is also a bidirectional iterator.
- An algorithm that works for forward iterators, should also work with bidirectional iterators. Similarly, an algorithm that works with bidirectional iterators, should also work with random access iterators .

Iterators for BoxContainer

Algorithm	Iterator type
std::find()	Input iterator
std::fill()	Forward Iterator
std::reverse()	Bidirectional Iterator
Std::sort()	Random Access Iterator

Slide intentionally left empty

Iterator Powers


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Iterators for BoxContainer

- We have played with stl algorithms for quite a while now, and seen the benefits of buil in stl containers
- Iterators are the glue that ties containers and algorithms together
- Sometimes, you want your own containers to be able to work with built in stl algorithms, like std::sort, std::find, std::fill,...
- That just opens a new world of flexibility for you container
- We want BoxContainer to support iterators, so that it can use range based for loops for example, and be usable with stl algorithms

std::ranges::find algorithm [Input Iterators]

```cpp
//Input iterator
std::cout << "---------------(find)-----------" << std::endl;
std::vector<int> numbers {1,9,3,7,2,5,4,6,8};
std::cout << "numbers : " <<  numbers << std::endl;

//Iterators returned by begin() are input iterators. The requirement is that we are
//able to read through them. That's al std::ranges::find needs.
//Show possible implementations at cppreference.
if (std::ranges::find(numbers.cbegin(),numbers.cend(), 8) != numbers.cend()) {
    std::cout << "numbers contains: " << 8 << '\n';
} else {
    std::cout << "numbers does not contain: " << 8 << '\n';
}
```

Operator<< for std::vector<T>

```
//operator<< for std::vector<T>
template <typename T>
std::ostream& operator<<( std::ostream& out,const  std::vector<T>& vec){
    out << " [ ";
    for(auto i : vec){
        out << i  << " ";
    }
    out << "]";
    return out;
}
```

std::ranges::copy algorithm [Output Iterators]

```
//Output iterator : std::ranges::copy
//Iterator through which we can write
std::cout << "---------------(copy)-----------" << std::endl;
std::vector<int> dest(numbers.size());
//std::vector<int> dest; //  BAD! Probably a crash
std::cout << "numbers : " << numbers << std::endl;
std::cout << "dest : " << dest << std::endl;

//dest.begin() has to be an output iterator, have to be able to write though it
std::ranges::copy(numbers.begin(),numbers.end(),dest.cbegin()); // Compiler Error dest.cbegin()
                                                                // is not an output iterator
std::cout << "numbers : " << numbers << std::endl;
std::cout << "dest : " << dest << std::endl;
```

std::ranges::replace algorithm [Forward Iterators]

```
//Forward iteator : std::ranges::replace, std::ranges::fill
std::cout << "---------------(replace)-------------" << std::endl;
std::cout << "numbers : " << numbers << std::endl;

//replacing every instance of 7 with 345. The iterator needs an
//operator++ to move forward. See possible implementation
std::ranges::replace(numbers.begin(),numbers.end(),7,345);
std::cout << "numbers : " << numbers << std::endl;
```

std::ranges::reverse algorithm [Bidirectional Iterators]

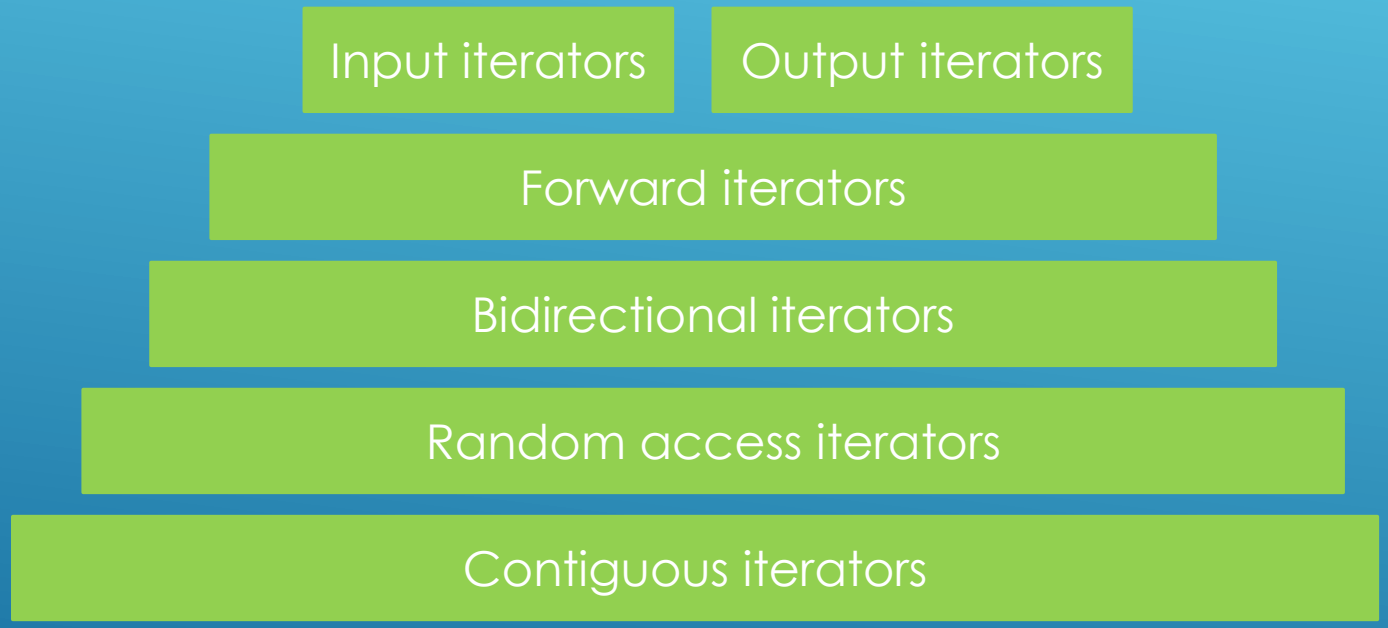
```
//Bidirectional iterator : // Print the list's contents out in reverse order.
// std::ranges::reverse
std::cout << "---------------(bi-directional)-------------" << std::endl;

std::cout << "numbers : " << numbers << std::endl;
auto it_first = numbers.begin();
auto it_last  = numbers.end();
while (it_last-- != it_first) {
    std::cout << *it_last << " ";
}

std::cout << std::endl;
 std::ranges::reverse(numbers);
std::cout << "numbers : " << numbers << std::endl;
```

std::ranges::sort algorithm [RandomAccess Iterators]

```
//Random access iteator : std::ranges::sort
std::cout << "---------------(sort)-------------" << std::endl;
std::cout << "numbers : " << numbers << std::endl;

//Sorting the collection
std::ranges::sort(numbers);
std::cout << "numbers : " << numbers << std::endl;
```

Slide intentionally left empty

Building Custom Iterators


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Iterators for BoxContainer

- We have played with stl algorithms for quite a while now, and seen the benefits of buil in stl containers
- Iterators are the glue that ties containers and algorithms together
- Sometimes, you want your own containers to be able to work with built in stl algorithms, like std::sort, std::find, std::fill,...
- That just opens a new world of flexibility for you container
- We want BoxContainer to support iterators, so that it can use range based for loops for example, and be usable with stl algorithms

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type aliases help algorithms work better

BoxContainer Initial changes

```
template <typename T> requires  std::is_default_constructible_v<T>
class BoxContainer
{
    class Iterator
    {
    public :
        using iterator_category = std::input_iterator_tag;
        using difference_type   = std::ptrdiff_t;
        using value_type        = T;
        using pointer_type          = T*;
        using reference_type        = T&;

    private:
        pointer_type m_ptr;
    };
    Iterator begin() { return Iterator(&m_items[0]); }
    Iterator end()   { return Iterator(&m_items[m_size]); }
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Input iterator operator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

std::ranges::find algorithm [Input Iterators]

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

//find algorithm
if (std::ranges::find(box1, 8) != box1.end()) {
    std::cout << "numbers contains: " << 8 << std::endl;
} else {
    std::cout << "numbers does not contain: " << 8 << std::endl;
}
```

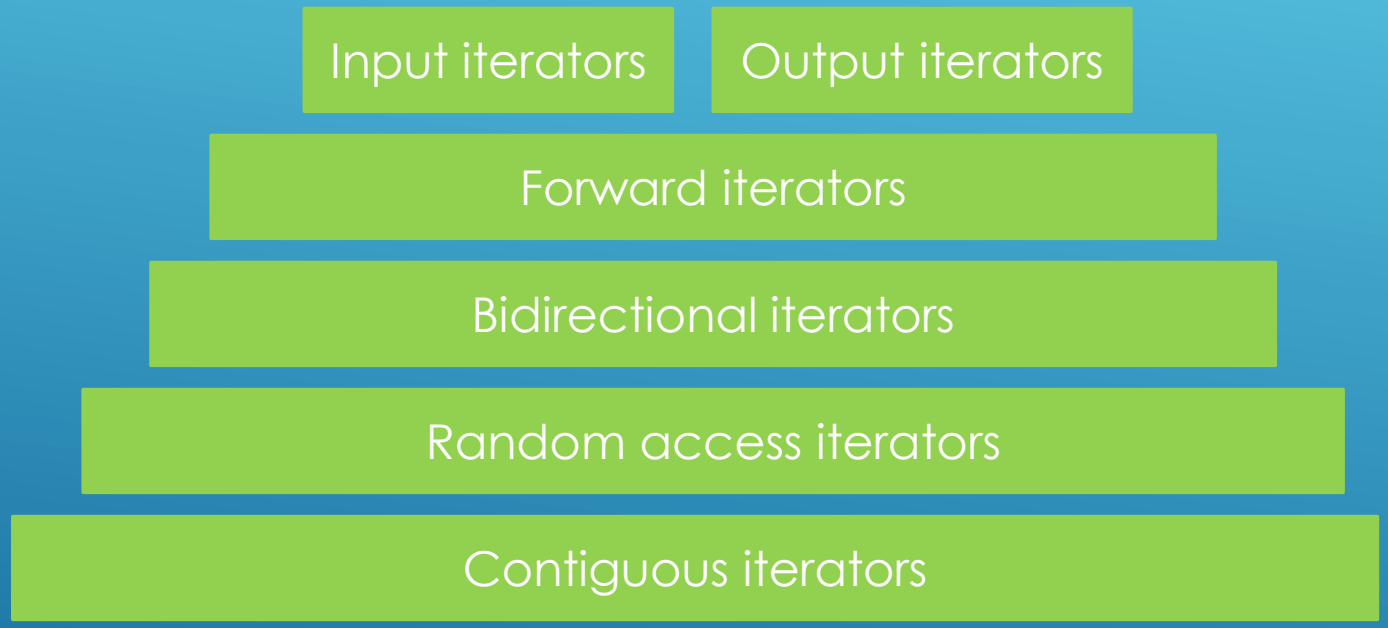
Range based for loop

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

//Range based for loop
for(auto n : box1){
    std::cout << n << " ";
}
std::cout << std::endl;
```

BoxContainer should also work with std::ranges algorithms as much as possible

Slide intentionally left empty

Custom Input Iterator


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type traits help algorithms work better

BoxContainer Initial changes

```
template <typename T> requires  std::is_default_constructible_v<T>
class BoxContainer
{
    class Iterator
    {
    public :
        using iterator_category = std::input_iterator_tag;
        using difference_type   = std::ptrdiff_t;
        using value_type        = T;
        using pointer_type          = T*;
        using reference_type        = T&;

    private:
        pointer_type m_ptr;
    };
    Iterator begin() { return Iterator(&m_items[0]); }
    Iterator end()   { return Iterator(&m_items[m_size]); }
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Input iterator operator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

Input iterator methods

```
Iterator() = default;

Iterator(pointer_type ptr) : m_ptr(ptr) {}

reference_type operator*() const {
    return *m_ptr;
}

pointer_type operator->() {
    return m_ptr;
}
```

Input iterator methods (contd)

```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

std::ranges::find algorithm [Input Iterators]

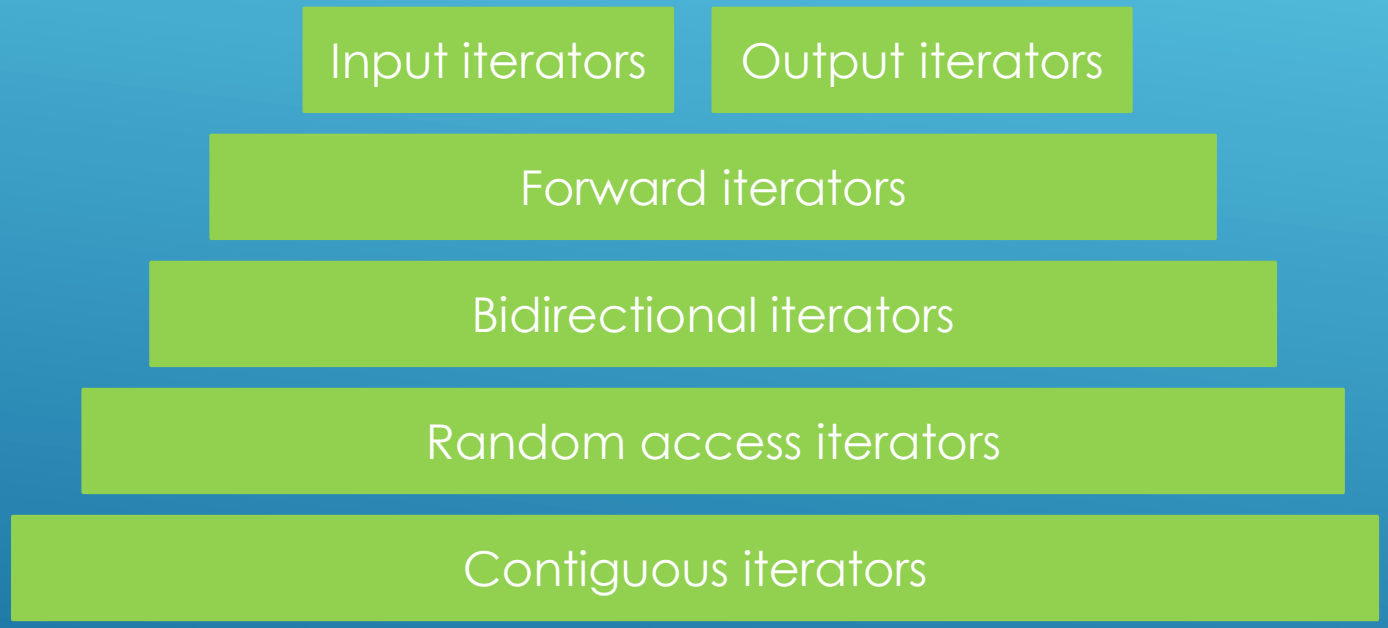
```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

//find algorithm
if (std::ranges::find(box1, 8) != box1.end()) {
    std::cout << "numbers contains: " << 8 << std::endl;
} else {
    std::cout << "numbers does not contain: " << 8 << std::endl;
}
```

Range based for loop

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

//Range based for loop
for(auto n : box1){
    std::cout << n << " ";
}
std::cout << std::endl;
```

Slide intentionally left empty

Custom Output Iterator


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type traits help algorithms work better

Type aliases

```
using iterator_category = std::forward_iterator_tag;
using difference_type   = std::ptrdiff_t;
using value_type        = T;
using pointer_type            = T*;
using reference_type          = T&;
```

Input iterator operator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

Input/output iterator methods

```
Iterator() = default;

Iterator(pointer_type ptr) : m_ptr(ptr) {}

reference_type operator*() const {
    return *m_ptr;
}

pointer_type operator->() {
    return m_ptr;
}
```

Input iterator methods (contd)

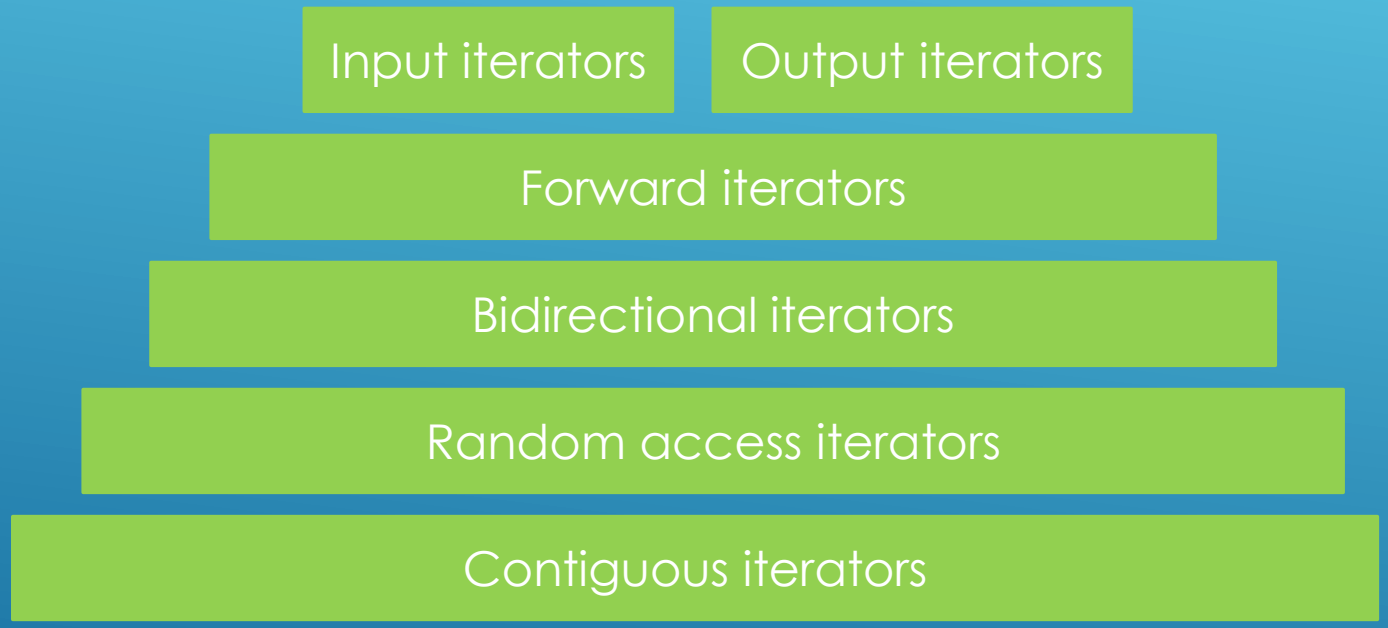
```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

std::ranges::copy algorithm [Output Iterators]

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

std::cout << "box : " << box1 << std::endl;

BoxContainer<int> box2; // Needs to populate to have the correct size
for(size_t i{}; i < box1.size();++i){
    box2.add(0);
}

std::cout << "box2-1 : " << box2 << std::endl;
std::ranges::copy(box1.begin(),box1.end(),box2.begin());
std::cout << "box2-2 : " << box2 << std::endl;
```

Slide intentionally left empty

Custom Forward Iterator


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type traits help algorithms work better

Type aliases

```
using iterator_category = std::forward_iterator_tag;
using difference_type   = std::ptrdiff_t;
using value_type        = T;
using pointer_type            = T*;
using reference_type          = T&;
```

Input iterator operator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

Input/output iterator methods

```
Iterator() = default;

Iterator(pointer_type ptr) : m_ptr(ptr) {}

reference_type operator*() const {
    return *m_ptr;
}

pointer_type operator->() {
    return m_ptr;
}
```

Input iterator methods (contd)

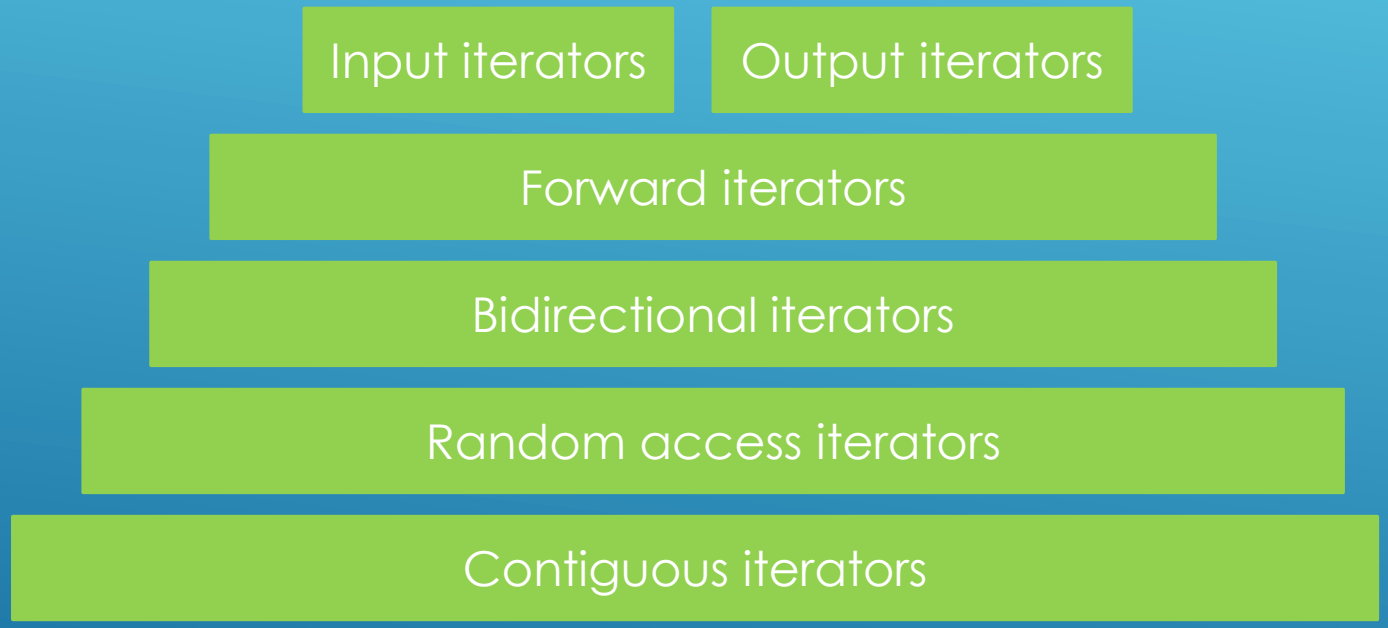
```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

std::ranges::forward algorithm [Output Iterators]

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(7);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

std::cout << "box1 : " << box1 << std::endl;
std::ranges::replace(box1.begin(),box1.end(),7,777);
std::cout << "box1 : " << box1 << std::endl;
```

Slide intentionally left empty

Custom Bidirectional Iterator


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type traits help algorithms work better

Type aliases

```
using iterator_category = std::bidirectional_iterator_tag;
using difference_type   = std::ptrdiff_t;
using value_type        = T;
using pointer_type            = T*;
using reference_type          = T&;
```

Forward iterator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

Forward iterator methods

```
Iterator() = default;

Iterator(pointer_type ptr) : m_ptr(ptr) {}

reference_type operator*() const {
    return *m_ptr;
}

pointer_type operator->() {
    return m_ptr;
}
```

Forward iterator methods (contd)

```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

Input iterator methods (contd)

```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

Bidirectional iterator methods

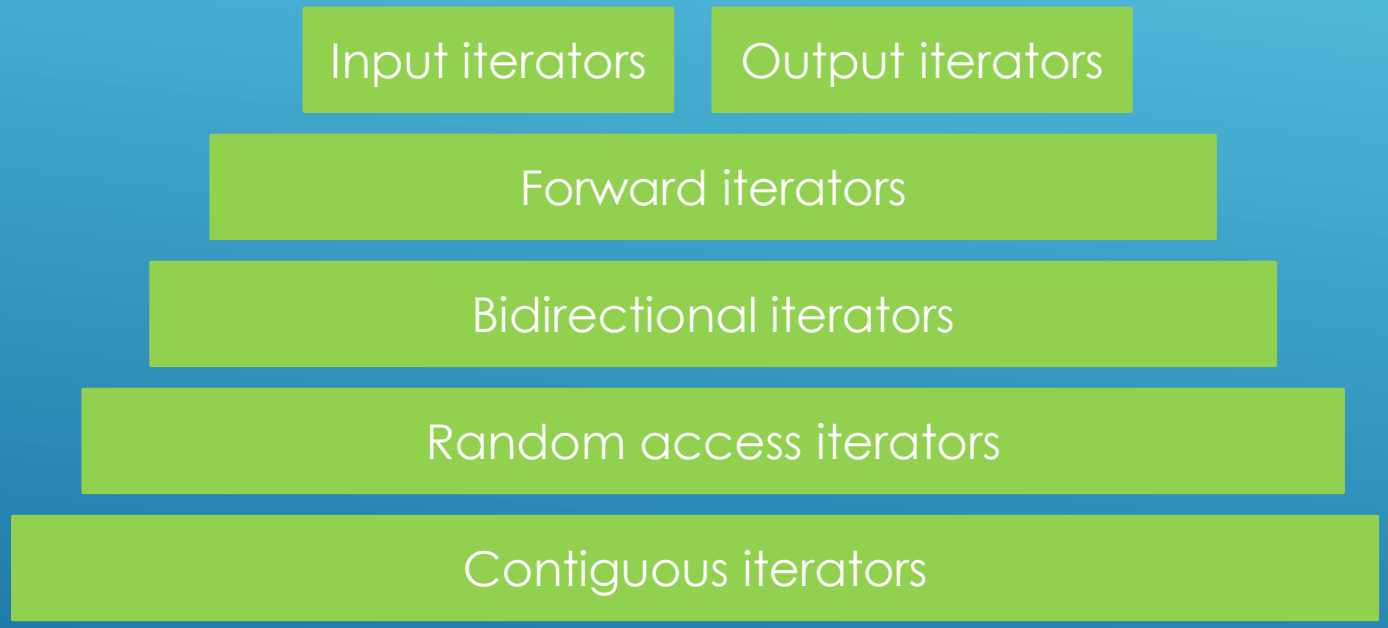
```
//Bidirectional
Iterator& operator--() {
    m_ptr--; return *this;
}
Iterator operator--(int) {
    Iterator tmp = *this;
    --(*this);
    return tmp;
}
```

std::ranges::reverse algorithm [Bidirectional Iterators]

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(7);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

std::cout << "box1 : " << box1 << std::endl;
std::ranges::replace(box1.begin(),box1.end(),7,777);
std::cout << "box1 : " << box1 << std::endl;
```

Slide intentionally left empty

Custom Random Access Iterator


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type traits help algorithms work better

Type aliases

```
using iterator_category = std::random_access_iterator_tag;
using difference_type   = std::ptrdiff_t;
using value_type        = T;
using pointer_type            = T*;
using reference_type          = T&;
```

Forward iterator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

Forward iterator methods

```
Iterator() = default;

Iterator(pointer_type ptr) : m_ptr(ptr) {}

reference_type operator*() const {
    return *m_ptr;
}

pointer_type operator->() {
    return m_ptr;
}
```

Forward iterator methods (contd)

```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

Input iterator methods (contd)

```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

Bidirectional iterator methods

```
//Bidirectional
Iterator& operator--() {
    m_ptr--; return *this;
}
Iterator operator--(int) {
    Iterator tmp = *this;
    --(*this);
    return tmp;
}
```

Random Access Methods

```
//Random access
Iterator& operator+=(const difference_type offset) {
    m_ptr += offset;
    return *this;
}

Iterator operator+(const difference_type offset) const  {
    Iterator tmp = *this;
    return tmp += offset;
}

Iterator& operator-=(const difference_type offset) {
    return *this += -offset;
}

Iterator operator-(const difference_type offset) const  {
    Iterator tmp = *this;
    return tmp -= offset;
}

difference_type operator-(const Iterator& right) const {
    return m_ptr - right.m_ptr;
}
```

Random Access Methods(contd)

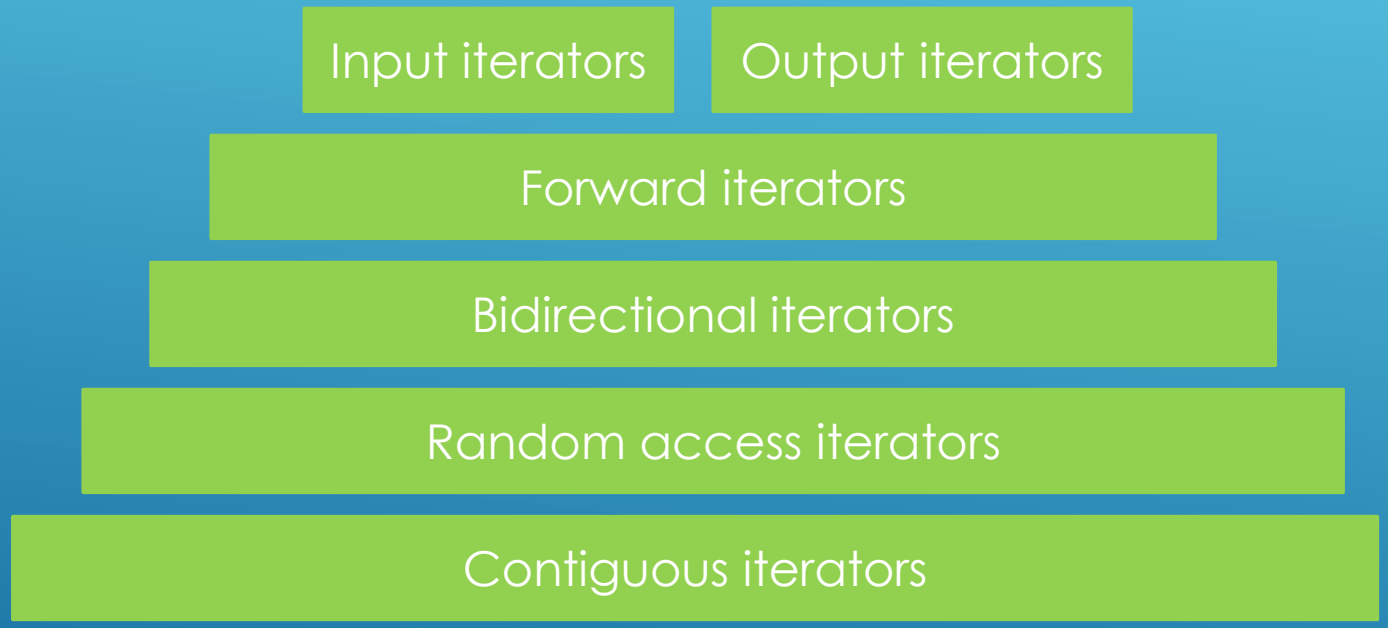
```
reference_type operator[](const difference_type offset) const  {
    return *(*this + offset);
}

bool operator<(const Iterator& right) const  {
    return m_ptr < right.m_ptr;
}

bool operator>(const Iterator& right) const  {
    return right < *this;
}

bool operator<=(const Iterator& right) const {
    return !(right < *this);
}

bool operator>=(const Iterator& right) const  {
    return !(*this < right);
}

friend Iterator operator+(const difference_type offset, const Iterator& it){
    Iterator tmp = it;
    return tmp += offset;
}
```

std::ranges::sort algorithm [Random Access Iterators]

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(7);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

std::cout << "box1 : " << box1 << std::endl;
std::ranges::sort(box1.begin(),box1.end());
std::cout << "box1 : " << box1 << std::endl;
```

Slide intentionally left empty

Custom Iterators With Views


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Std::vector with views and range adaptors

```
std::vector<int> vi {8,1,7,2,5,3,7,9};
/* ...
//std::ranges::filter_view
std::cout <<std::endl;
std::cout << "std::ranges::filter_view : " << std::endl;
auto evens = [](int i){
    return (i %2) == 0;
};
std::cout << "vi : " ;
print(vi);

std::ranges::filter_view v_evens = std::ranges::filter_view(vi,evens); //No computation
std::cout << "vi evens : ";
print(v_evens); //Computation happens in the print function

Print evens on the fly
std::cout << "vi evens : " ;
print(std::ranges::filter_view(vi,evens));
```

Std::vector with views and range adaptors

BoxContainer<int> vi {8,1,7,2,5,3,7,9};

```
//std::ranges::filter_view
std::cout <<std::endl;
std::cout << "std::ranges::filter_view : " << std::endl;
auto evens = [](int i){
    return (i %2) == 0;
};
std::cout << "vi : " ;
print(vi);

std::ranges::filter_view v_evens = std::ranges::filter_view(vi,evens); //No computation
std::cout << "vi evens : ";
print(v_evens); //Computation happens in the print function

Print evens on the fly
std::cout << "vi evens : " ;
print(std::ranges::filter_view(vi,evens));
```

The collection [Data owner]

```
//std::vector<int> vi {8,1,7,2,5,3,7,9};
BoxContainer<int> vi;
vi.add(5);
vi.add(1);
vi.add(7);
vi.add(2);
vi.add(5);
vi.add(3);
vi.add(7);
vi.add(9);
vi.add(6);
```

Filter views with a Custom Container

```
//std::ranges::filter_view
std::cout <<std::endl;
std::cout << "std::ranges::filter_view : " << std::endl;
auto evens = [](int i){
    return (i %2) == 0;
};
std::cout << "vi : " ;
print(vi);

std::ranges::filter_view v_evens = std::ranges::filter_view(vi,evens); //No computation
std::cout << "vi evens : ";
print(v_evens); //Computation happens in the print function

Print evens on the fly
std::cout << "vi evens : " ;
print(std::ranges::filter_view(vi,evens));
```

Transform view with a Custom Container

```
//std::ranges::transform_view
std::cout <<std::endl;
std::cout << "std::ranges::transform_view : " << std::endl;
std::ranges::transform_view v_transformed = std::ranges::transform_view(vi,[](int i){
    return i * 10;
});
std::cout << "vi : " ;
print(vi);
std::cout << "vi transformed : " ;
print(v_transformed);
std::cout << "vi : ";
print(vi);
```

```
//std::ranges::take_view
std::cout <<std::endl;
std::cout << "std::ranges::take_view : " << std::endl;
std::ranges::take_view v_taken = std::ranges::take_view(vi,5);
std::cout << "vi : " ;
print(vi);
std::cout << "vi taken : ";
print(v_taken);

//std::ranges::take_while_view
std::cout <<std::endl;
std::cout << "std::views::take_while : " << std::endl;
std::ranges::take_while_view v_taken_while = std::ranges::take_while_view(vi,[](int i){
    return (i%2)!=0;
});
std::cout << "vi : ";
print(vi);
std::cout << "vi taken_while : ";
print(v_taken_while);
```

```
//std::ranges::drop_view : drop n first elements
std::cout <<std::endl;
std::cout << "std::ranges::drop_view : " << std::endl;
std::ranges::drop_view v_drop = std::ranges::drop_view(vi,5);
std::cout << "vi : ";
print(vi);
std::cout << "vi_drop : ";
print(v_drop);

//std::views::drop_while_view : drops elements as long as the predicate is met
std::cout <<std::endl;
std::cout << "std::ranges::drop_while_view : " << std::endl;
std::ranges::drop_while_view v_drop_while = std::ranges::drop_while_view(vi,[](int i){
    return (i%2)!=0;
});
std::cout << "vi : ";
print(vi);
std::cout << "v_drop_while : ";
print(v_drop_while);
```

Using range adaptors

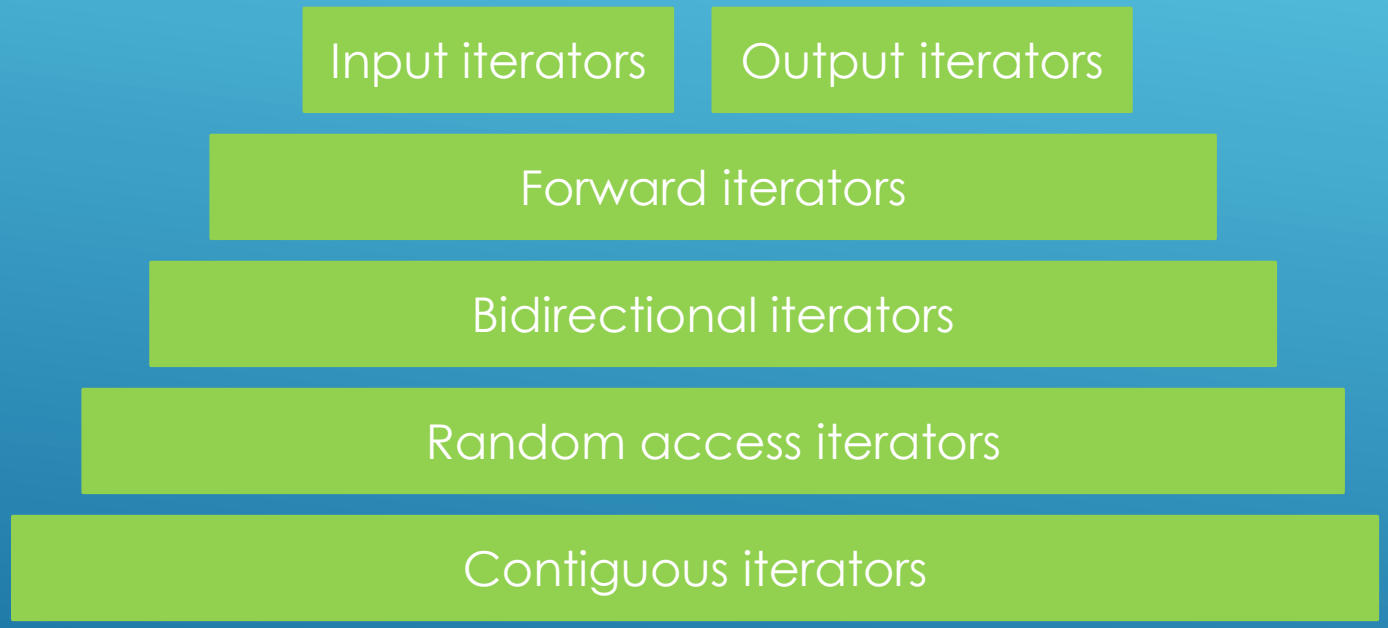
```
//std::views::filter()
std::cout <<std::endl;
std::cout << "std::views::filter : " << std::endl;
auto evens1 = [](int i){
    return (i %2) == 0;
};
std::cout << "vi : " ;
print(vi);
std::ranges::filter_view v_evens1 = std::views::filter(vi,evens1); //No computation
std::cout << "vi evens : ";
print(v_evens1); //Computation happens in the print function
//Print evens on the fly
std::cout << "vi evens : " ;
print(std::views::filter(vi,evens1));
//Print odds on the fly
std::cout << "vi odds : " ;
print(std::views::filter(vi,[](int i){
    return (i%2)!=0;
}));
```

```
//Students example
BoxContainer<Student> class_room; //  {{"Mike",12},{"John",17},{"Drake",14},{"Mary",16}};
class_room.add(Student("Mike",12));
class_room.add(Student("John",17));
class_room.add(Student("Drake",14));
class_room.add(Student("Mary",16));

std::cout << std::endl;
std::cout << "classroom : " << std::endl;
for( auto& s : class_room){
    std::cout << "    " <<  s << std::endl;
}
```

```
std::ranges::sort(class_room,std::less<>{},&Student::m_age);

std::cout << std::endl;
std::cout << "classroom (after sort) : " << std::endl;
for( auto& s : class_room){
    std::cout << "    " <<  s << std::endl;
}

std::cout << "students under 15 : " ;
//print(std::views::take_while(class_room,[](const Student& s){return (s.m_age <15);}));
auto less_than_15_v = class_room | std::views::take_while([](const Student& s){return (s.m_age <15);});
print(less_than_15_v);
std::cout << "End!" << std::endl;
```

Slide intentionally left empty

Constant Iterators


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

```
BoxContainer<int> vi;
vi.add(5);
vi.add(1);
vi.add(7);
vi.add(2);
vi.add(5);
vi.add(3);
vi.add(7);
vi.add(9);
vi.add(6);

const BoxContainer<int> copy(vi);

std::cout << "data : ";
for (auto it = copy.begin(); it!=copy.end(); ++it){
    std::cout << (*it) << " ";
}
std::cout << std::endl;
```

```
template <typename T>
void print(const BoxContainer<T>&  c){
    for(auto i : c){
        std::cout << i << " ";
    }
    std::cout << std::endl;
}
```

We need constant Iterators for BoxContainer

```cpp
template <typename T> requires  std::is_default_constructible_v<T>
class BoxContainer
{
    class ConstIterator
    {
    public :
        /* ...
        using pointer_type          = const T*;
        using reference_type        = const T&;
        //...
        private:
        pointer_type m_ptr;
    };
    Iterator begin() { return Iterator(&m_items[0]); }
    Iterator end()   { return Iterator(&m_items[m_size]); }

    //Picked up for const containers
    ConstIterator begin() const { return ConstIterator(&m_items[0]); }
    ConstIterator end() const   { return ConstIterator(&m_items[m_size]); }

    ConstIterator cbegin() { return ConstIterator(&m_items[0]); }
    ConstIterator cend()   { return ConstIterator(&m_items[m_size]); }
};
```

Slide intentionally left empty

Raw pointers as iterators

```
template <typename T>
class BoxContainer
{
public :
    T* begin() { return m_items; }
    T* end()   { return m_items + m_size; }

    const T* cbegin() { return m_items; }
    const T* cend()   { return m_items + m_size; }

public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    /* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

```
BoxContainer<int> box1;
box1.add(8);
box1.add(1);
box1.add(4);
/* ...
std::cout << "box1 : " << box1 << std::endl;

auto it = box1.begin();
std::cout << *it << std::endl;
*it = 6; // Compiler error

for (auto i : box1){
    std::cout << i << " ";
}
std::cout << std::endl;

std::sort(box1.begin(),box1.end());
std::cout << "box1 : " << box1 << std::endl;

std::cout << "view taking only 3 : " ;
for(auto i : std::views::take(box1,3)){
    std::cout << i << " " ;
}
std::cout << std::endl;
```

```
BoxContainer<int> vi;
vi.add(5);
vi.add(1);
vi.add(7);
vi.add(2);
vi.add(5);
vi.add(3);
vi.add(7);
vi.add(9);
vi.add(6);

const BoxContainer<int> copy(vi);

std::cout << "data : ";
for (auto it = copy.begin(); it!=copy.end(); ++it){
    std::cout << (*it) << " ";
}
std::cout << std::endl;
```

```
template <typename T>
void print(const BoxContainer<T>&  c){
    for(auto i : c){
        std::cout << i << " ";
    }
    std::cout << std::endl;
}
```

Slide intentionally left empty

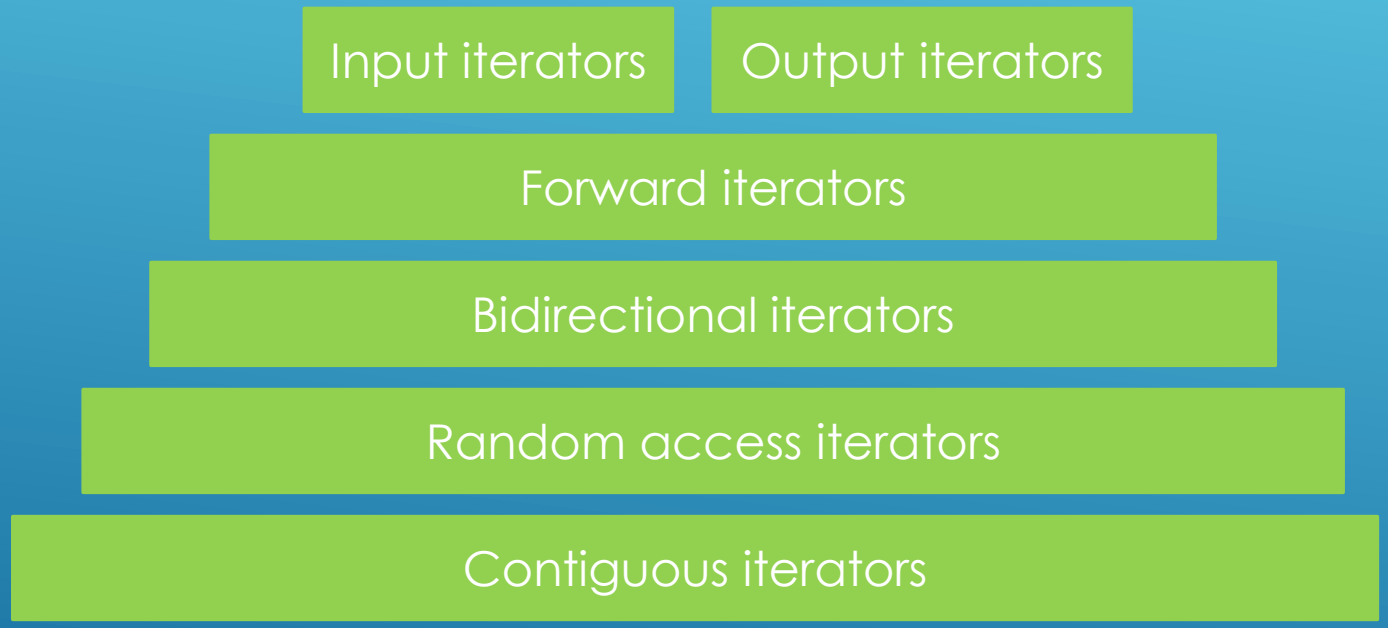
Wrapping iterators from other containers

```cpp
template <typename T>
class VectorWrapper{
public:
    //Iterator methods
    std::vector<T>::iterator begin() { return m_items.begin(); }
    std::vector<T>::iterator end()   { return m_items.end(); }
    std::vector<T>::const_iterator cbegin() { return m_items.cbegin(); }
    std::vector<T>::const_iterator cend()   { return m_items.cend(); }

    friend std::ostream& operator<< (std::ostream& out, const VectorWrapper<T>& vec){
        out << "Items : " ;
        for (auto i : vec.m_items){
            out << i << " ";
        }
        return out;
    }

    void add( T item){
        m_items.push_back(item);
    }
private :
    std::vector<T> m_items;
};
```

```
VectorWrapper<std::string> greeting;
greeting.add("Hello");
greeting.add("World!");
greeting.add("How");
greeting.add("are");
greeting.add("you");
greeting.add("all");
greeting.add("doing?");

std::cout << "greeting : " << greeting << std::endl;

std::cout << "Range based for loop : " << std::endl;
for(auto i : greeting){
    std::cout << i << " ";
}
std::cout << std::endl;

std::cout << "taking only 2 : " << std::endl;
for(auto i : greeting | std::views::take(2)){
    std::cout << i << " ";
}
std::cout << std::endl;
```

Slide intentionally left empty

Custom Iterators : Summary


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Custom Iterator Requirements

- Your container class needs to be a class template
- The container class has to model iterator types
- Your container needs begin() and end() methods that return those iterators
- The iterators have to model the operators needed by your algorithms

Iterator Type

- Needs to provide the type aliases expected by the standard template library
- These type traits help algorithms work better

BoxContainer Initial changes

```
template <typename T> requires  std::is_default_constructible_v<T>
class BoxContainer
{
    class Iterator
    {
    public :
        using iterator_category = std::input_iterator_tag;
        using difference_type   = std::ptrdiff_t;
        using value_type        = T;
        using pointer_type          = T*;
        using reference_type        = T&;

    private:
        pointer_type m_ptr;
    };
    Iterator begin() { return Iterator(&m_items[0]); }
    Iterator end()   { return Iterator(&m_items[m_size]); }
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Input iterator operator methods

```
class Iterator
{
public :
    //Type aliases here
    Iterator() = default;
    Iterator(pointer_type ptr) : m_ptr(ptr) {}
    reference_type operator*() const {return *m_ptr;}
    pointer_type operator->() { return m_ptr;}
    Iterator& operator++() {
        m_ptr++;
        return *this;
    }
    Iterator operator++(int) {
        Iterator tmp = *this;
        ++(*this);
        return tmp;
    }
    friend bool operator== (const Iterator& a, const Iterator& b) { ...
    friend bool operator!= (const Iterator& a, const Iterator& b) { ...
private:
    pointer_type m_ptr;
};
```

Input iterator methods

```
Iterator() = default;

Iterator(pointer_type ptr) : m_ptr(ptr) {}

reference_type operator*() const {
    return *m_ptr;
}

pointer_type operator->() {
    return m_ptr;
}
```

Input iterator methods (contd)

```
Iterator& operator++() {
    m_ptr++;
    return *this;
}
Iterator operator++(int) {
    Iterator tmp = *this;
    ++(*this);
    return tmp;
}
friend bool operator== (const Iterator& a, const Iterator& b) {
    return a.m_ptr == b.m_ptr;
}
friend bool operator!= (const Iterator& a, const Iterator& b) {
    //return a.m_ptr != b.m_ptr;
    return !(a == b);
}
```

std::ranges::find algorithm [Input Iterators]

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

//find algorithm
if (std::ranges::find(box1, 8) != box1.end()) {
    std::cout << "numbers contains: " << 8 << std::endl;
} else {
    std::cout << "numbers does not contain: " << 8 << std::endl;
}
```

Range based for loop

```
BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(2);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

//Range based for loop
for(auto n : box1){
    std::cout << n << " ";
}
std::cout << std::endl;
```

- Raw pointers as iterators
- Wrapping iterators from other containers
- Custom containers with views and range adaptors

Slide intentionally left empty