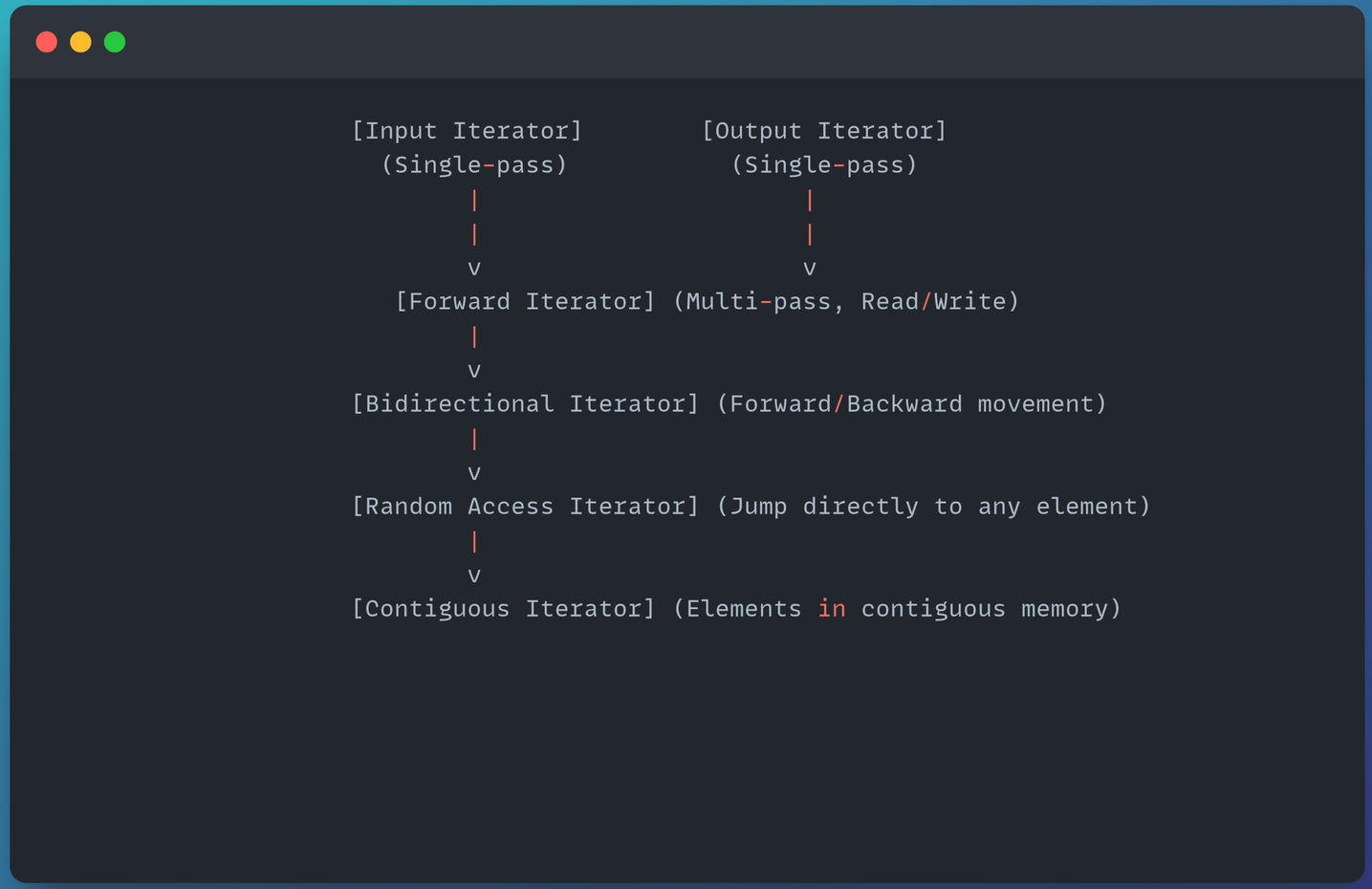
Custom Iterators for your containers

```
/*
. Custom container iterators:
    .#1: Input iterator
        . These are iterators that can be used to read from a container: read only
        . Input iterators are single pass.

    .#2: Output iterator
        . They have input iterator capabilities, but can also be used to modify the value referenced by the iterator.

    .#3: Forward iterator
        . forward iterators allow muliple passes over the same range.
        . you can reset the iterator to the beginning of the range, and expect to get the same values again.
        . they are not single pass

    .#4: Bidirectional iterators:
        . operators:
            . prefix and post fix operator--

    .#5: Random access iteators:
        . operators:
            . Lots of operators that allow them to behave just like regular raw pointers.

    .#6: Custom iterators with views:
        . Rationale: When custom containers are powered with iterators, they work seamlesly with standard view types.

    .#7: Constant and reverse iterators

    .#8: Raw pointers as iterators

    .#9: Wrapping around existing iterators
*/
```

Iterator relationship



Iterator relationship

Iterator Type	Access Type	Movement Type
Input Iterator	Read-only (Single-pass)	Forward
Output Iterator	Write-only (Single-pass)	Forward
Forward Iterator	Read/Write (Multi-pass)	Forward
Bidirectional Iterator	Read/Write	Forward/Backward
Random Access Iterator	Read/Write	Forward/Backward (Random access)

Iterator Compatibility

```
/*
  . Random Access Iterator: The most versatile iterator, can be used wherever other iterators are required.
  . Bidirectional Iterator: Can be used wherever a Forward Iterator is required.
  . Forward Iterator: Can be used in scenarios requiring forward traversal and can be used for multiple passes over the same data
*/
```

The procedure

```
/*
        . Steps to create a custom iterator

            . Define the Container Class:
                . Create a class to manage the data and operations of your container.
            . Nested Iterator Class:
                . Implement a nested Iterator class with:
                    . Type Definitions: Define value_type, pointer, reference, etc.
                    . Constructor: Initialize with a pointer/reference to the data.
                    . Operator Overloads:
                        . Dereference (*), Arrow (→), Increment (++), Decrement (--), and Comparison (═, ≠).
                        . Optional: Subscript operator ([]) for indexed access.
            . Begin and End Methods:
                . Implement begin() and end() methods:
                    . begin(): Returns an iterator to the start.
                    . end(): Returns an iterator to one past the last element.
            . Const Begin and End Methods:
                . Implement cbegin() and cend() for constant access.
            . Additional Considerations:
                . Support reverse iteration with rbegin(), rend(), etc.
                . Ensure const-correctness.
                . Handle exceptions for out-of-bounds access.
*/
```

Input iterator

```
/*
 . Input iterator scheleton:
        . Type Definitions:
            . iterator_category: Defines the iterator as an input iterator (std::input_iterator_tag).
            . difference_type: Defines the type used for differences between iterators (std::ptrdiff_t).
            . value_type: Represents the type of the value pointed to by the iterator (T).
            . pointer_type: Type of pointer to the value (T*).
            . reference_type: Type of reference to the value (T&).
        . Constructors:
            . Default constructor: Allows the creation of an Iterator object without initializing it.
            . Parameterized constructor: Accepts a pointer to the underlying data to initialize
                the iterator (explicit Iterator(pointer_type ptr)).
        . Copy Operations:
            . Copy constructor: Defaulted, allows copying of iterator instances.
            . Copy assignment operator: Defaulted, allows assignment of one iterator to another.
        . Dereference Operators:
            . operator*: Provides access to the value pointed to by the iterator, allowing dereferencing.
            . operator→: Provides access to the members of the value pointed to, using the arrow operator.
        . Increment Operators:
            . Pre-increment operator (operator++): Advances the pointer to the next element and
                returns the updated iterator.
            . Post-increment operator (operator++(int)): Advances the pointer but returns a copy
                of the iterator before the increment.
            . These increment operators are what is used for example in a find algorithm, visiting
                elements one by one in a container, until the desired element is found.
       Comparison Operators - eg: Used to compare with the end iterator:
            . Equality comparison operator (operator==): Compares two iterators for equality based on their pointers.
            . Inequality comparison operator (operator≠): Compares two iterators for inequality,
                using the equality operator for implementation.
        . Private Members:
            m_ptr: A pointer to the current element being pointed to by the iterator, initialized to nullptr by default.
*/
```

Input iterator

```cpp
// BoxContainer class template definition
    export template <typename T>
    requires std::default_initializable<T> // Ensuring type T is default-initializable
    class BoxContainer {
        // Iterator class for BoxContainer (Input Iterator)
        class Iterator {
        public:
            using iterator_category = std::input_iterator_tag;   // Input iterator tag
            using difference_type = std::ptrdiff_t;              // Difference type
            using value_type = T;                                // Value type
            using pointer_type = T*;                             // Pointer type
            using reference_type = T&;                           // Reference type
            Iterator() = default;                                // Default constructor
        private:
            pointer_type m_ptr{ nullptr }; // Pointer to the current element
        };

        // Begin iterator (points to the first element)
        Iterator begin() { return Iterator(m_items.get()); }
        // End iterator (points one past the last element)
        Iterator end() { return Iterator(m_items.get() + m_size); }
    private:
    private:
        std::unique_ptr<T[]> m_items;  // m_items is a smart pointer that will manage
                                       // a dynamically allocated array of type T.
        size_t m_capacity{ 0 };        // Current capacity of the container
        size_t m_size{ 0 };            // Current number of elements in the container
    };
```

Input iterator: Methods and operators

```cpp
class BoxContainer {
    // Iterator class for BoxContainer (Input Iterator)
    class Iterator {
    public:
        Iterator() = default;                                   // Default constructor
       // Constructor taking a pointer to the underlying data
        explicit Iterator(pointer_type ptr) : m_ptr(ptr) {}

        // Copy constructor and assignment operator can be defaulted
        Iterator(const Iterator&) = default;
        Iterator& operator=(const Iterator&) = default;

        // Dereference operator to access the value
        reference_type operator*() const { return *m_ptr; }

        // Arrow operator to access members directly
        pointer_type operator->() { return m_ptr; }

        // Pre-increment operator (advances the pointer)
        Iterator& operator++() { ++m_ptr; return *this; }

        // Post-increment operator (advances but returns old value)
        Iterator operator++(int) { Iterator tmp = *this; ++(*this); return tmp; }

        // Equality comparison operator for iterators
        friend bool operator==(const Iterator& a, const Iterator& b) { return a.m_ptr == b.m_ptr; }

        // Inequality comparison operator for iterators
        friend bool operator!=(const Iterator& a, const Iterator& b) { return !(a == b); }
    private:
        pointer_type m_ptr{ nullptr }; // Pointer to the current element
    };
};
```

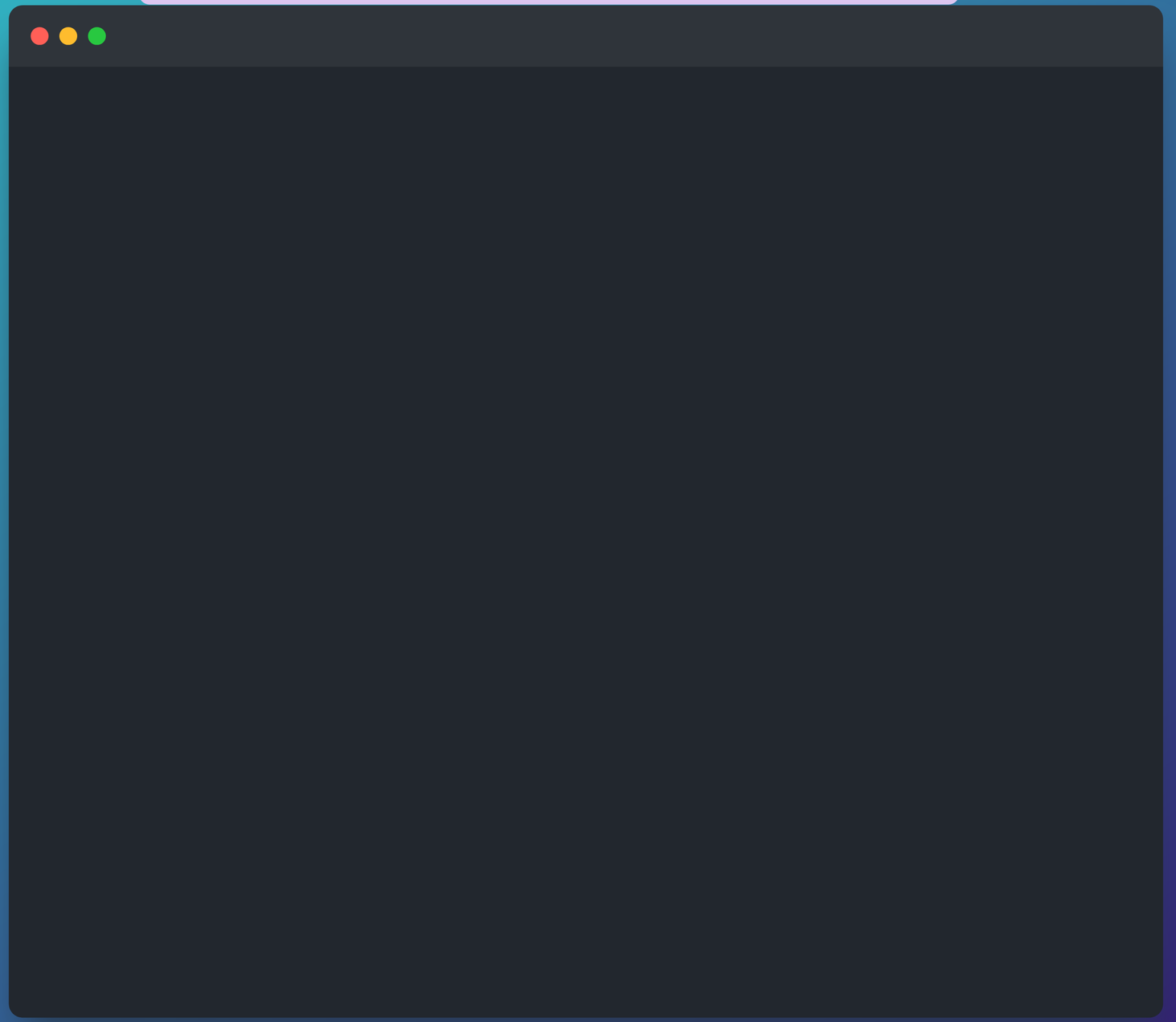
Spoiler alert!

```
/*
    . What we have so far will work as:
        . input iterator
        . output iterator
        . forward iterator
*/
```

Test drive

```
//Road test with where input iterator is required.
//std::vector<int> box1 {8,1,4,2,5,3,7,9};
custom_iterators_01::BoxContainer<int> box1;
//improvement::BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(8);
box1.add(5);

//Use iterator to access the elements
for(auto it = box1.begin(); it=/=box1.end(); ++it){
    fmt::print("{} ",*it);
}

//find algorithm: requires an input iterator
if (std::ranges::find(box1, 8) =/= box1.end()) {
    fmt::println("numbers contains: {}",8);
} else {
    fmt::println("numbers does not contain: {}",8);
}

//Won't work. Needs a bidirectional iterator
//std::ranges::reverse(box1.begin(),box1.end());


//Range based for loop
for(auto n : box1){
    fmt::print("{} ",n);
}
  fmt::println("\n");
```

Input iterator: Demo time!



Output iterator test drive.

```cpp
//Ouput iterators: write only - we go through operator* and operator-> to modify the value referenced by the iterator.
//The class in custom_iterators_01 already supports output iterators through operator* and operator->
custom_iterators_01::BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(8);
box1.add(5);
box1.add(3);
box1.add(7);
box1.add(9);
box1.add(6);

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

//Destination box
custom_iterators_01::BoxContainer<int> box2;
for(size_t i{}; i < box1.size();++i){
    box2.add(0);
}

std::cout << "box2-1 : " << box2 << std::endl;
//The copy algorithm needs an output iterator to paste into the destination.
std::ranges::copy(box1,box2.begin());
std::cout << "box2-2 : " << box2 << std::endl;
```

Forward iterator test drive

```cpp
//Forward iterators: The std::replace algorithm requires forward iterators.
// A vector of integers with consecutive duplicates
 custom_iterators_01::BoxContainer<int> box1;
 box1.add(5);
 box1.add(1);
 box1.add(4);
 box1.add(8);
 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;
```

Bidirectional iterator: Forward iterator as a base

```
class BoxContainer {
    // Iterator class for BoxContainer (Input Iterator)
    class Iterator {
    public:
        Iterator() = default;                                   // Default constructor
       // Constructor taking a pointer to the underlying data
        explicit Iterator(pointer_type ptr) : m_ptr(ptr) {}

        // Copy constructor and assignment operator can be defaulted
        Iterator(const Iterator&) = default;
        Iterator& operator=(const Iterator&) = default;

        // Dereference operator to access the value
        reference_type operator*() const { return *m_ptr; }

        // Arrow operator to access members directly
        pointer_type operator->() { return m_ptr; }

        // Pre-increment operator (advances the pointer)
        Iterator& operator++() { ++m_ptr; return *this; }

        // Post-increment operator (advances but returns old value)
        Iterator operator++(int) { Iterator tmp = *this; ++(*this); return tmp; }

        // Equality comparison operator for iterators
        friend bool operator==(const Iterator& a, const Iterator& b) { return a.m_ptr == b.m_ptr; }

        // Inequality comparison operator for iterators
        friend bool operator!=(const Iterator& a, const Iterator& b) { return !(a == b); }
    private:
        pointer_type m_ptr{ nullptr }; // Pointer to the current element
    };
};
```

Bidirectional iterator: The decrement operators

```
class BoxContainer {
    // Iterator class for BoxContainer (Input Iterator)
    class Iterator {
    public:

        //All the methods we had for Forward iterator plus the decrement methods below:

        // Pre-decrement operator (moves the pointer backward)
        Iterator& operator--() { --m_ptr; return *this; }

        // Post-decrement operator
        Iterator operator--(int) { Iterator tmp = *this; --(*this); return tmp; }

    private:
        pointer_type m_ptr{ nullptr }; // Pointer to the current element
    };
};
```

Bidirectional iterator: Test drive

```
custom_iterators_04::BoxContainer<int> box1;
 box1.add(5);
 box1.add(1);
 box1.add(4);
 box1.add(8);
 box1.add(5);
 box1.add(3);
 box1.add(7);
 box1.add(9);
 box1.add(6);

 std::cout << "box1 : " << box1 << std::endl;
 std::ranges::reverse(box1);
 std::cout << "box1 : " << box1 << std::endl;

 //Traverse in reverse
 std::cout << "box1 (printed reversed) : ";
 for(auto it = box1.end();it!=box1.begin(); ){
     std::cout << *(--it )<< " ";
 }
 std::cout << std::endl;
```

Random access iterator: Bidirectional as a foundation

```
/*
 Random access iterator additional methods:

        Iterator& operator+=(const difference_type offset) {}

        Iterator operator+(const difference_type offset) const  {}

        Iterator& operator-=(const difference_type offset) {}

        Iterator operator-(const difference_type offset) const  {}

        difference_type operator-(const Iterator& right) const {}

        reference_type operator[](const difference_type offset) const  {}

        bool operator<(const Iterator& right) const  {}

        bool operator>(const Iterator& right) const  {}

        bool operator⩽(const Iterator& right) const {}

        bool operator⩾(const Iterator& right) const  {}

        friend Iterator operator+(const difference_type offset, const Iterator& it){}
*/
```

Random access iterator: Test drive

```cpp
custom_iterators_05::BoxContainer<int> box1;
box1.add(5);
box1.add(1);
box1.add(4);
box1.add(8);
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::ranges::sort(box1);
//std::sort(box1.begin(),box1.end());
std::cout << "box1 : " << box1 << std::endl;
```

BoxContainer can work with C++ Views and view adaptors

```
custom_iterators_05::BoxContainer<int> vi;
    vi.add(5);
    vi.add(1);
    vi.add(4);
    vi.add(8);
    vi.add(5);
    vi.add(3);
    vi.add(7);
    vi.add(9);
    vi.add(6);

    //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 : " ;
    custom_iterators_06::print(vi);


    std::ranges::filter_view v_evens = std::ranges::filter_view(vi,evens); //No computation
    std::cout << "vi evens : ";
    custom_iterators_06::print(v_evens); //Computation happens in the print function
```

Const and reverse iterators (random access)

```
//Add respective iterator nested classes and begin and end methods
// Begin iterator (points to the first element)
Iterator begin() { return Iterator(m_items.get()); } // Begin iterator: points to the first element
Iterator end() { return Iterator(m_items.get() + m_size); } // End iterator: points one past the last element

ConstIterator begin() const { return ConstIterator(m_items.get()); }
ConstIterator end() const { return ConstIterator(m_items.get() + m_size); }

ReverseIterator rbegin() { return ReverseIterator(m_items.get() + m_size - 1); }
ReverseIterator rend() { return ReverseIterator(m_items.get() - 1); }

ConstReverseIterator rbegin() const { return ConstReverseIterator(m_items.get() + m_size - 1); }
ConstReverseIterator rend() const { return ConstReverseIterator(m_items.get() - 1); }
```

Raw pointers as iterators

```
//Add respective  begin and end methods: make them return variations on the raw pointer
T* begin() { return m_items.get(); }
T* end() { return m_items.get() + m_size; }

const T* begin() const { return m_items.get(); }
const T* end() const { return m_items.get() + m_size; }

T* rbegin() { return m_items.get() + m_size - 1; }
T* rend() { return m_items.get() - 1; }

const T* rbegin() const { return m_items.get() + m_size - 1; }
const T* rend() const { return m_items.get() - 1; }
```

Wrapping around existing iterators

```cpp
export 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(); }

        //Const overloads
        std::vector<T>::const_iterator begin() const{ return m_items.cbegin(); }
        std::vector<T>::const_iterator end() const  { return m_items.cend(); }


        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;
    };
```