Practice: BoxContainer

```
/*
    . A BoxContainer class
    . Our inhouse built version of std::vector
    . You shouldn't build your own containers in practice,
      unless there's a good reason those from the standard won't work for you.
    . Topics:
       . Construction and destruction
       . Adding items
       . Removing items
       . Useful operators:
          . operator+
          . operator+=
       . Class templates and modernization

*/
```

G

Gakwaya Daniel

𝕏

BoxContainer: Construction, destruction

```cpp
export class BoxContainer : public StreamInsertable
{
    using value_type = int;
    static constexpr size_t DEFAULT_CAPACITY = 30;
    static constexpr size_t DUMMY_ITEM_COUNT = 10;

public:
    BoxContainer(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer(const BoxContainer& source); // Copy constructor
    BoxContainer& operator=(BoxContainer source);  // Copy assignment using copy-and-swap idiom
    BoxContainer(BoxContainer&& source) noexcept;  // Move constructor
    BoxContainer& operator=(BoxContainer&& source) noexcept;    // Move assignment
    ~BoxContainer();

    virtual void stream_insert(std::ostream& out) const override;

    size_t size() const { return m_size; }
    size_t capacity() const { return m_capacity; }

    void dummy_initialize();

    // Swap function
    void swap(BoxContainer& other) noexcept;

private:
    value_type* m_items;
    size_t m_capacity;
    size_t m_size;
};
```

BoxContainer: Construction, destruction

```cpp
// Constructor
BoxContainer::BoxContainer(size_t capacity)
    : m_items(new value_type[capacity]), m_capacity(capacity), m_size(0) {}

// Copy constructor
BoxContainer::BoxContainer(const BoxContainer& source)
    : m_items(new value_type[source.m_capacity]), m_capacity(source.m_capacity), m_size(source.m_size) {
    //Loop through the source items and copy them to the new object
    /*
    for(size_t i{}; i < source.m_size; ++i) {
        m_items[i] = source.m_items[i];
    }
    */
    std::copy(source.m_items, source.m_items + source.m_size, m_items);
}

// Move constructor
BoxContainer::BoxContainer(BoxContainer&& source) noexcept
    : m_items(source.m_items), m_capacity(source.m_capacity), m_size(source.m_size) {
    source.m_items = nullptr;
    source.m_size = 0;
    source.m_capacity = 0;
}
```

BoxContainer: Assignments, copy and swap

```cpp
// Swap function for copy-and-swap idiom
void BoxContainer::swap(BoxContainer& other) noexcept {
    std::swap(m_items, other.m_items);
    std::swap(m_capacity, other.m_capacity);
    std::swap(m_size, other.m_size);
}

// Copy assignment using copy-and-swap
BoxContainer& BoxContainer::operator=(BoxContainer source) {
    // Swap contents with the temporary (source) object
    swap(source);
    // Return *this after the swap
    return *this;
}
```

BoxContainer: Assignments, copy and swap

```
// Move assignment operator
BoxContainer& BoxContainer::operator=(BoxContainer&& source) noexcept {
    if (this ≠ &source) {

        //Release the current resources
        delete[] m_items;

        //Steal data from the source
        m_items = source.m_items;
        m_capacity = source.m_capacity;
        m_size = source.m_size;

        //Reset the source
        source.m_items = nullptr;
        source.m_size = 0;
        source.m_capacity = 0;
    }
    return *this;
}
```

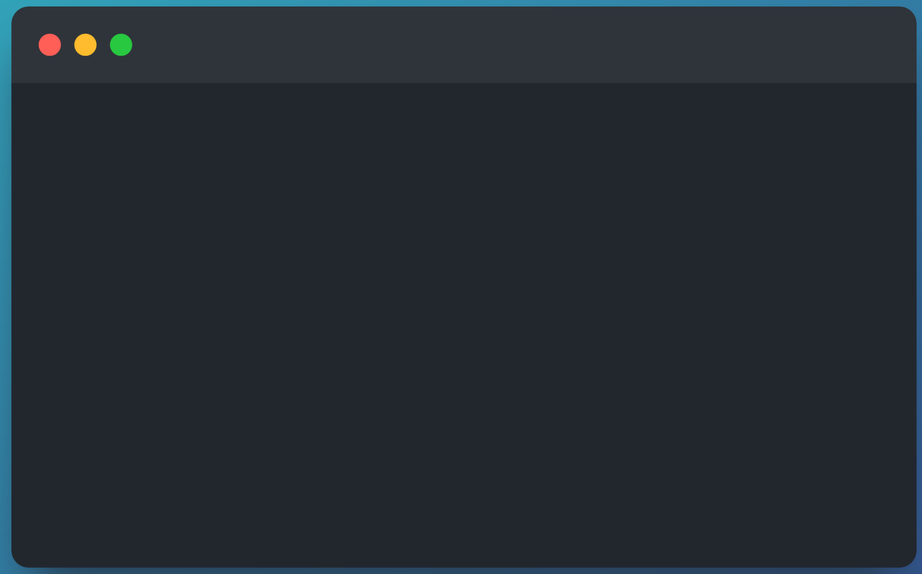
BoxContainer: Destruction

```cpp
// Destructor
BoxContainer::~BoxContainer() {
    delete[] m_items;
}
```

BoxContainer: Test run

```cpp
BoxContainer box1(15);
box1.dummy_initialize();
std::cout << "box1 : " << box1 << std::endl;

BoxContainer box2(box1);    // Copy constructor
std::cout << "box2 : " << box2 << std::endl;

BoxContainer box3;
box3 = box1;    // Copy assignment
std::cout << "box3 : " << box3 << std::endl;
```

BoxContainer: Demo time!



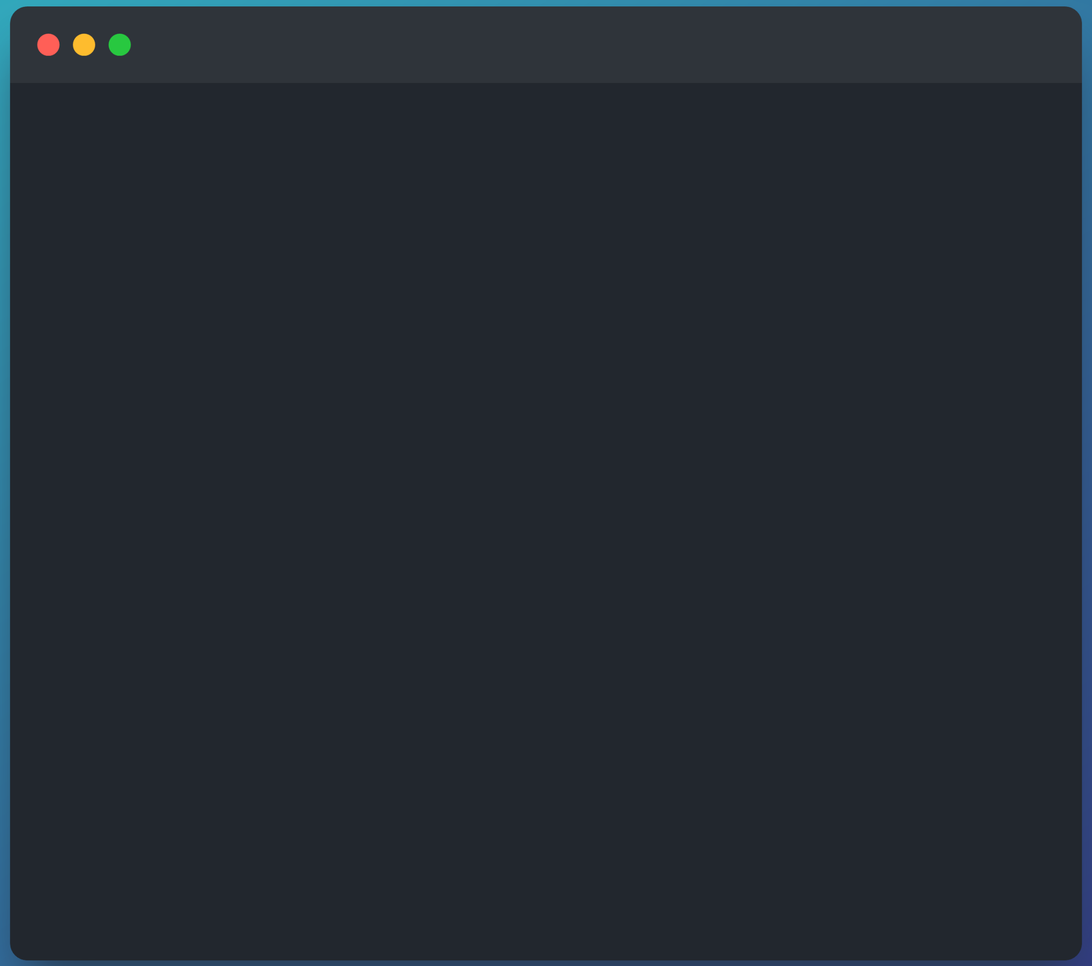
BoxContainer: Adding items

```cpp
// Add method: adds a new item to the container
void BoxContainer::add(value_type item) {
    if (m_size >= m_capacity) {
        // Expand if capacity is insufficient
        expand(m_capacity * 2);
    }
    m_items[m_size++] = item;
}

// Expand method: increases the capacity of the container
void BoxContainer::expand(size_t new_capacity) {
    value_type* new_items = new value_type[new_capacity];

    // Copy the old items into the new array
    std::copy(m_items, m_items + m_size, new_items);

    // Delete old items and reassign the new array
    delete[] m_items;
    m_items = new_items;
    m_capacity = new_capacity;
}
```

Adding items: Test run

```
BoxContainer box1(5);
std::cout << "box1 : " << box1 << std::endl;

box1.add(11);
box1.add(12);
box1.add(13);
std::cout << "box1 : " << box1 << std::endl;

box1.add(14);
box1.add(15);
std::cout << "box1 : " << box1 << std::endl;

box1.add(16);
std::cout << "box1 : " << box1 << std::endl;

for(size_t i{0}; i < 4 ; ++i){
    box1.add(17+i);
}
std::cout << "box1 : " << box1 << std::endl;

box1.add(21);
std::cout << "box1 : "  << box1 << std::endl;
```

Adding items: Demo time!



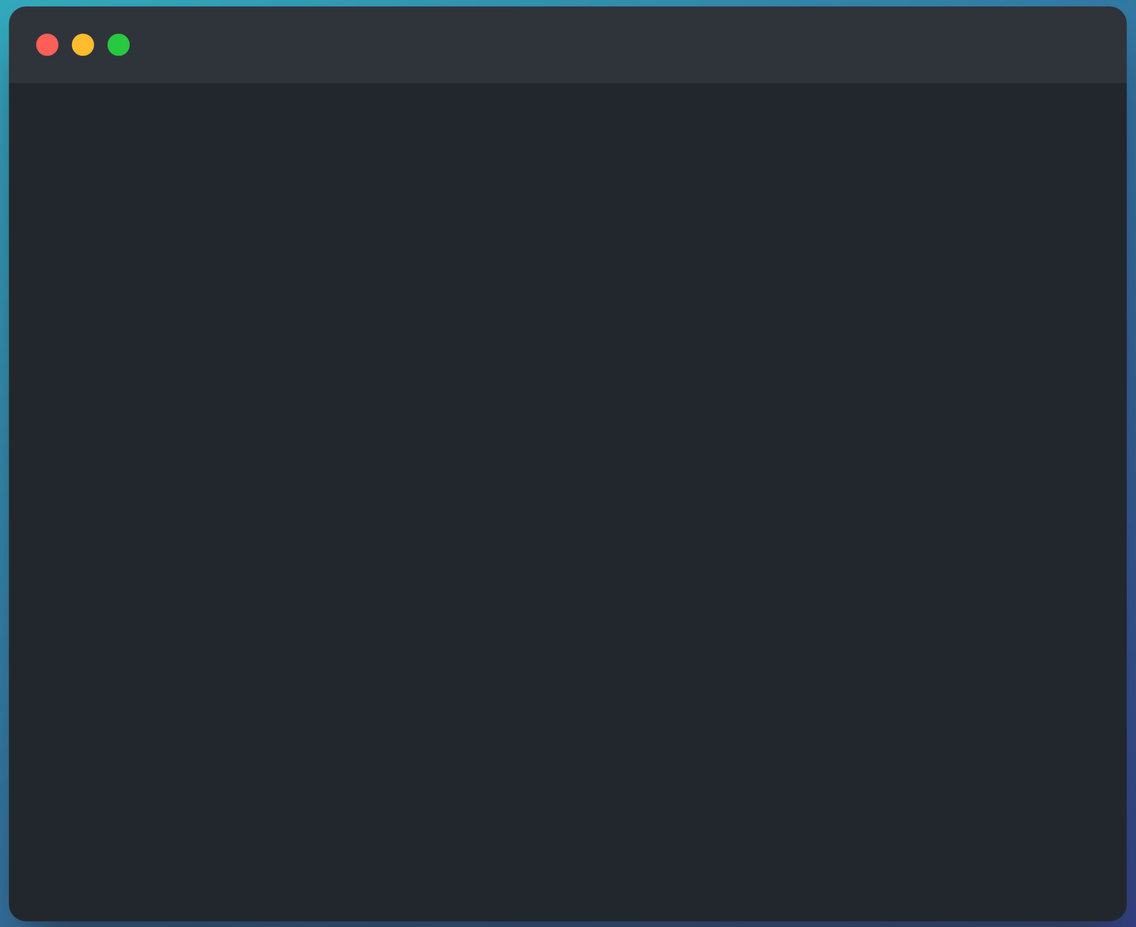
BoxContainer: Removing Items

```cpp
// Remove the first occurrence of the item
bool BoxContainer::remove_item(const value_type& item) {
    for (size_t i = 0; i < m_size; ++i) {
        if (m_items[i] == item) {
            // Shift all elements after the found item
            for (size_t j = i; j < m_size - 1; ++j) {
                m_items[j] = m_items[j + 1];
            }
            --m_size;
            return true;
        }
    }
    return false;
}

// Remove all occurrences of the item
size_t BoxContainer::remove_all(const value_type& item) {
    size_t count = 0;
    for (size_t i = 0; i < m_size; ) {
        if (m_items[i] == item) {
            remove_item(item);
            ++count;
        } else {
            ++i;
        }
    }
    return count;
}
```

Removing Items: Test Run.

```cpp
BoxContainer box1;
box1.add(11);
box1.add(12);
box1.add(13);

std::cout << "Removing items : " << std::endl;
box1.remove_item(15);

box1.add(11);
box1.add(33);
box1.add(52);
box1.add(11);

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

//Removing all instances  of 11
std::cout << "Removing all instances of 11 : " << std::endl;
box1.remove_all(11);
std::cout << "box1 : " << box1 << std::endl;
```

Removing Items: Demo time!



BoxContainer: Other operators

```cpp
// Operator+ : combines two BoxContainers into a new one
BoxContainer BoxContainer::operator+(const BoxContainer& other) const {
    BoxContainer result(m_size + other.m_size);
    std::copy(m_items, m_items + m_size, result.m_items);
    std::copy(other.m_items, other.m_items + other.m_size, result.m_items + m_size);
    result.m_size = m_size + other.m_size;
    return result;
}

// Operator+= : appends items from another BoxContainer to the current one
BoxContainer& BoxContainer::operator+=(const BoxContainer& other) {
    if (m_size + other.m_size > m_capacity) {
        expand(m_size + other.m_size);
    }
    std::copy(other.m_items, other.m_items + other.m_size, m_items + m_size);
    m_size += other.m_size;
    return *this;
}
```

Other operators: Test run.

```
//Operator+=
BoxContainer box1;
box1.add(1);
box1.add(2);
box1.add(3);

BoxContainer box2;
box2.add(10);
box2.add(20);
box2.add(30);

box2+= box1;

BoxContainer box3;
box3.add(81);
box3.add(82);


std::cout << "box1 + box3 : " << (box1 + box3) << std::endl;
```

Other operators: Demo time!

```
//Operator+=
BoxContainer box1;
box1.add(1);
box1.add(2);
box1.add(3);

BoxContainer box2;
box2.add(10);
box2.add(20);
box2.add(30);

box2+= box1;

BoxContainer box3;
box3.add(81);
box3.add(82);


std::cout << "box1 + box3 : " << (box1 + box3) << std::endl;
```

BoxContainer: Template class

```cpp
// Concept to constrain the types that can be used in BoxContainer
export template <typename T>
concept cout_printable = requires(T a, std::ostream& out) {
    { out << a } -> std::same_as<std::ostream&>;
};

export template <typename T>
concept BoxItem = std::copyable<T> && std::equality_comparable<T> && std::movable<T> && cout_printable<T>;

// Template class to make BoxContainer flexible for any type T
export template<BoxItem T>
class BoxContainer: public StreamInsertable {
    using value_type = T;
    static constexpr size_t DEFAULT_CAPACITY = 30;
    static constexpr size_t DUMMY_ITEM_COUNT = 10;

public:
    // Constructors and assignment operators
    BoxContainer(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer(const BoxContainer& source); // Copy constructor
    BoxContainer(BoxContainer&& source) noexcept;  // Move constructor
    BoxContainer& operator=(BoxContainer source);  // Copy assignment using copy-and-swap idiom
    BoxContainer& operator=(BoxContainer&& source) noexcept;    // Move assignment
    ~BoxContainer() = default;

    // Member functions
    void add(const value_type& item);     // Add item
    void expand(size_t new_capacity);     // Expand capacity
    bool remove_item(const value_type& item);    // Remove first occurrence
    size_t remove_all(const value_type& item);   // Remove all occurrences

    // Operator overloads
    BoxContainer operator+(const BoxContainer& other) const;  // operator+
    BoxContainer& operator+=(const BoxContainer& other);  // operator+=

    // Stream insert for pretty printing
    void stream_insert(std::ostream& out) const;

    // Getters
    size_t size() const noexcept { return m_size; }
    size_t capacity() const noexcept { return m_capacity; }

private:
    std::unique_ptr<value_type[]> m_items;  // Use unique_ptr for automatic memory management
    size_t m_capacity;
    size_t m_size;
};
```

BoxContainer: Template class

```cpp
// Constructor with default capacity
template<BoxItem T>
BoxContainer<T>::BoxContainer(size_t capacity)
    : m_items(std::make_unique<value_type[]>(capacity)), m_capacity(capacity), m_size(0) {}

// Copy constructor
template<BoxItem T>
BoxContainer<T>::BoxContainer(const BoxContainer& source)
    : m_items(std::make_unique<value_type[]>(source.m_capacity)), m_capacity(source.m_capacity), m_size(source.m_size) {
    std::copy(source.m_items.get(), source.m_items.get() + source.m_size, m_items.get());
}

// Move constructor
template<BoxItem T>
BoxContainer<T>::BoxContainer(BoxContainer&& source) noexcept
    : m_items(std::move(source.m_items)), m_capacity(source.m_capacity), m_size(source.m_size) {
    source.m_capacity = 0;
    source.m_size = 0;
}
```

BoxContainer: Template class

```cpp
// Move assignment operator
template<BoxItem T>
BoxContainer<T>& BoxContainer<T>::operator=(BoxContainer&& source) noexcept {
    if (this != &source) {
        m_items = std::move(source.m_items);
        m_capacity = source.m_capacity;
        m_size = source.m_size;
        source.m_capacity = 0;
        source.m_size = 0;
    }
    return *this;
}

// Copy assignment using copy-and-swap idiom
template<BoxItem T>
BoxContainer<T>& BoxContainer<T>::operator=(BoxContainer source) {
    swap(source);
    return *this;
}
```

BoxContainer: Template class

```cpp
// Operator+ : Combines two BoxContainers into a new one
template<BoxItem T>
BoxContainer<T> BoxContainer<T>::operator+(const BoxContainer& other) const {
    BoxContainer<T> result(m_size + other.m_size);
    std::copy(m_items.get(), m_items.get() + m_size, result.m_items.get());
    std::copy(other.m_items.get(), other.m_items.get() + other.m_size, result.m_items.get() + m_size);
    result.m_size = m_size + other.m_size;
    return result;
}

// Operator+= : Appends items from another BoxContainer to the current one
template<BoxItem T>
BoxContainer<T>& BoxContainer<T>::operator+=(const BoxContainer& other) {
    if (m_size + other.m_size > m_capacity) {
        expand(m_size + other.m_size);
    }
    std::copy(other.m_items.get(), other.m_items.get() + other.m_size, m_items.get() + m_size);
    m_size += other.m_size;
    return *this;
}
```

BoxContainer: Template class - Test run.

```cpp
// BoxContainer for int
BoxContainer<int> int_box;
int_box.add(10);
int_box.add(20);
int_box.add(30);
//std::cout << "int_box: " << int_box << "\n";
std::cout << "int_box : " << int_box << std::endl;

// BoxContainer for double
BoxContainer<double> double_box;
double_box.add(10.5);
double_box.add(20.3);
double_box.add(30.7);
std::cout << "double_box: " << double_box << "\n";

// BoxContainer for std::string
BoxContainer<std::string> string_box;
string_box.add("Hello");
string_box.add("World");
string_box.add("C++");
std::cout << "string_box: " << string_box << "\n";

// BoxContainer for Point
BoxContainer<Point> point_box;
point_box.add(Point{1, 2});
point_box.add(Point{3, 4});
point_box.add(Point{5, 6});
std::cout << "point_box: " << point_box << "\n";
```