Slides

Section : Move Semantics

Slide intentionally left empty

Move Semantics : Introduction

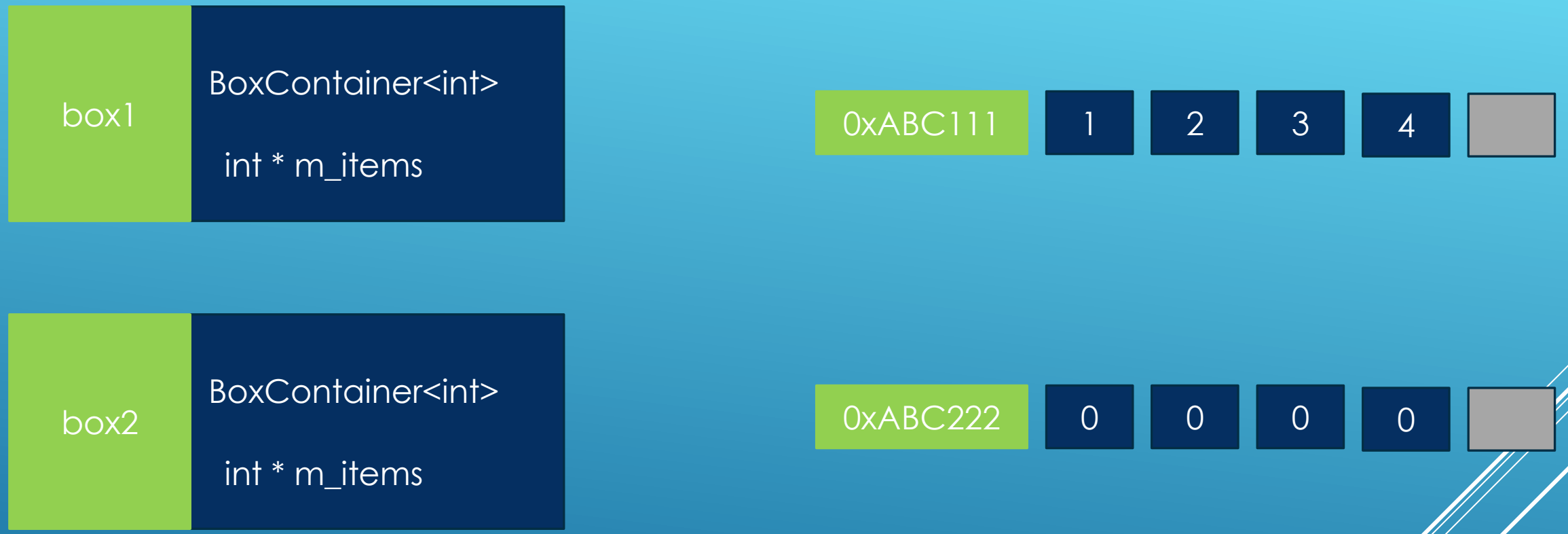
```
BoxContainer<int> box1 ;
box1.add(1);
box1.add(2);
box1.add(3);
box1.add(4);
```

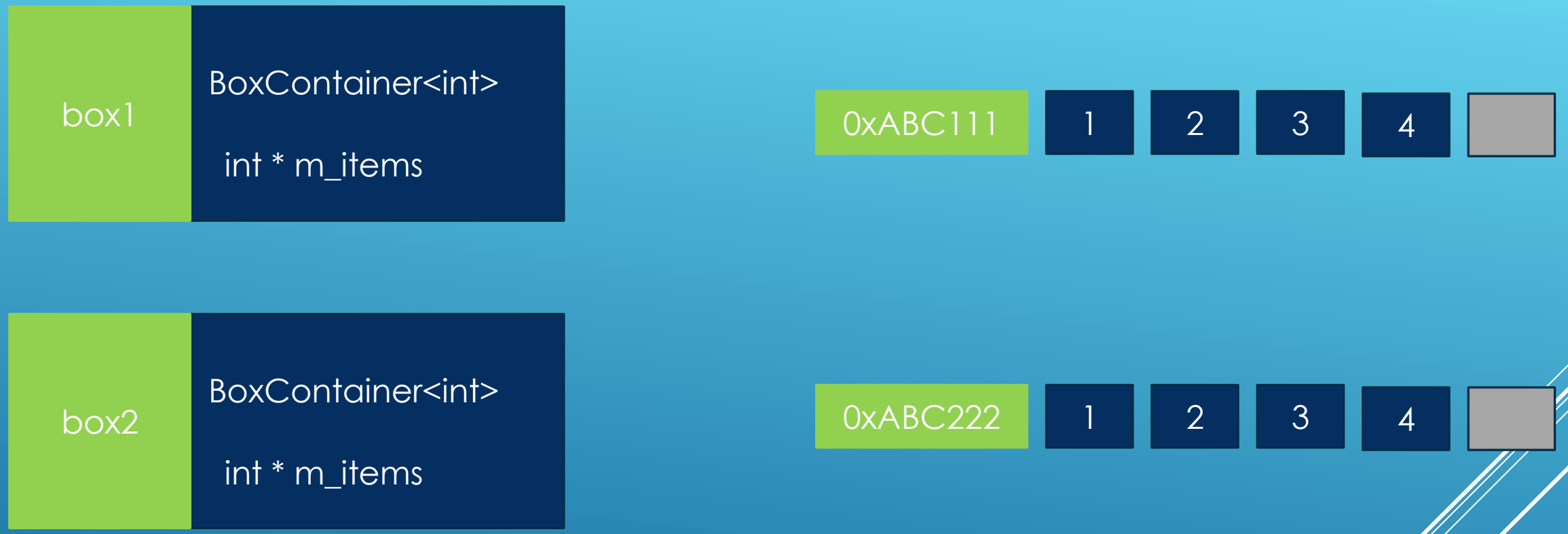

box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4

```
BoxContainer<int> box1 ;
box1.add(1);
box1.add(2);
box1.add(3);
box1.add(4);

BoxContainer<int> box2(box1);
```

Constructors

```
template <typename T>
BoxContainer<T>::BoxContainer(size_t capacity)
{
    m_items = new T[capacity];
    m_capacity = capacity;
    m_size =0;
}

template <typename T>
BoxContainer<T>::BoxContainer(const BoxContainer<T>& source)
{
    //Set up the new box
    m_items = new T[source.m_capacity];
    m_capacity = source.m_capacity;
    m_size = source.m_size;

    //Copy the items over from source
    for(size_t i{} ; i < source.size(); ++i){
        m_items[i] = source.m_items[i];
    }
}
```

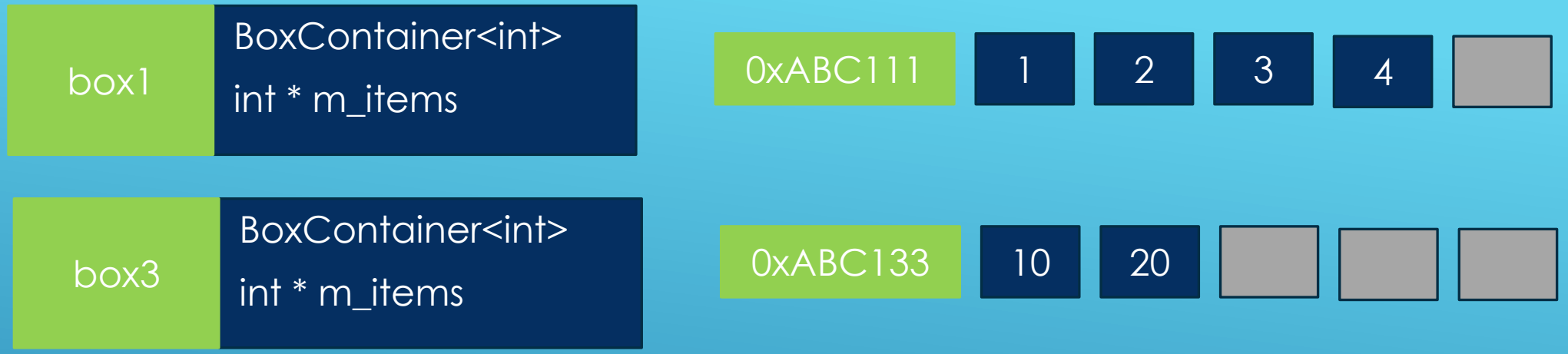

box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box2
BoxContainer<int>
int * m_items
0xABC222
0
0
0
0

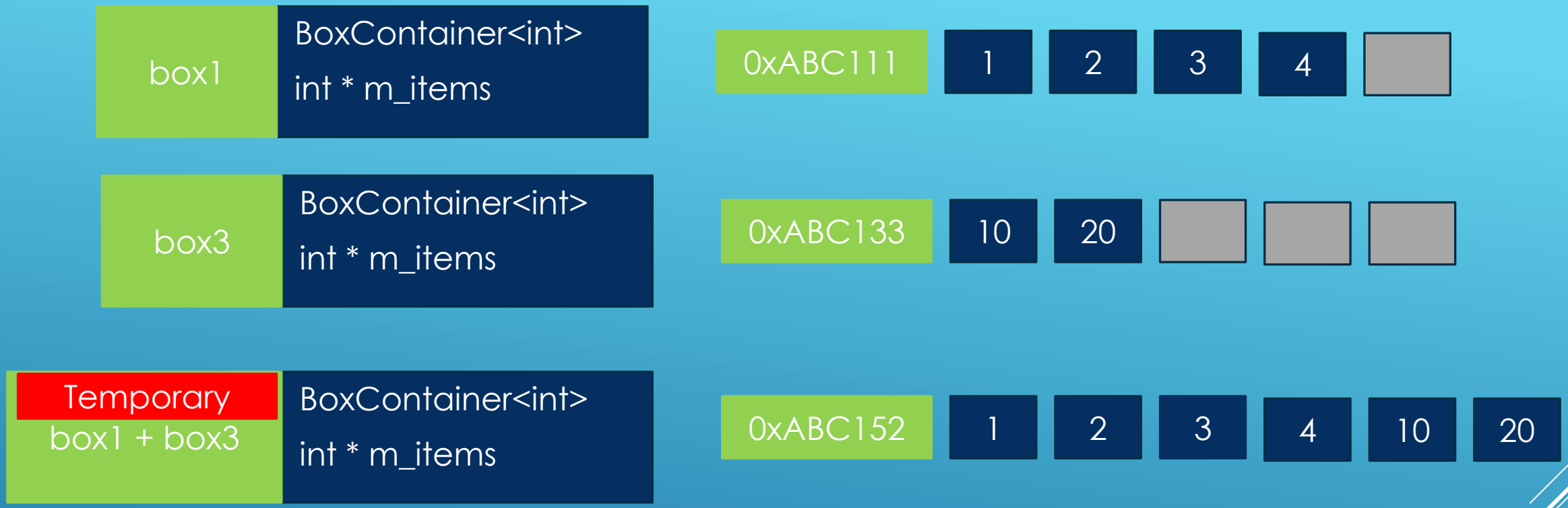

box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box2
BoxContainer<int>
int * m_items
0xABC222
1
2
3
4

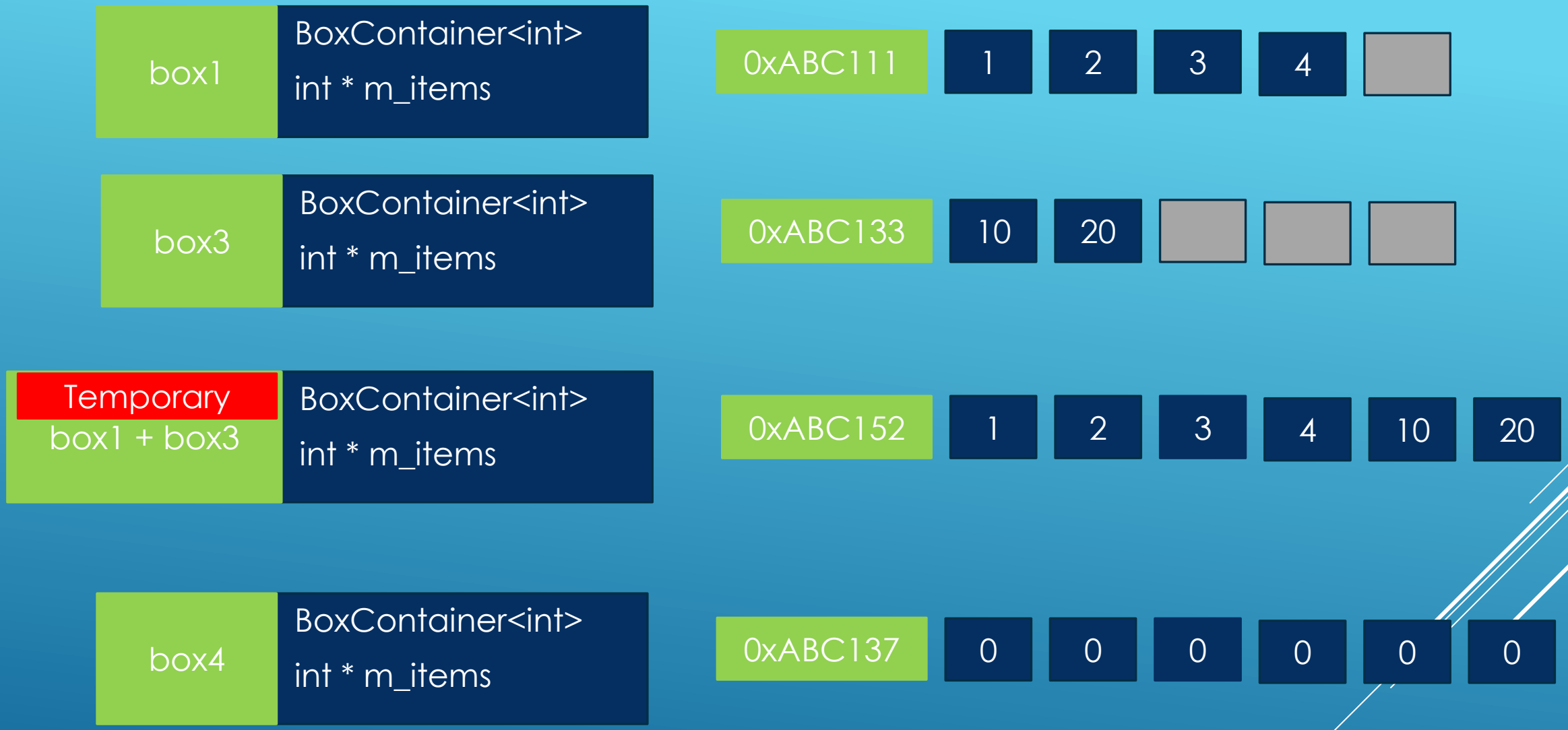
```
BoxContainer<int> box1 ;
box1.add(1);
box1.add(2);
box1.add(3);
box1.add(4);

BoxContainer<int> box2(box1);

BoxContainer<int> box3;
box3.add(10);
box3.add(20);

BoxContainer<int> box4(box1 + box3);
```

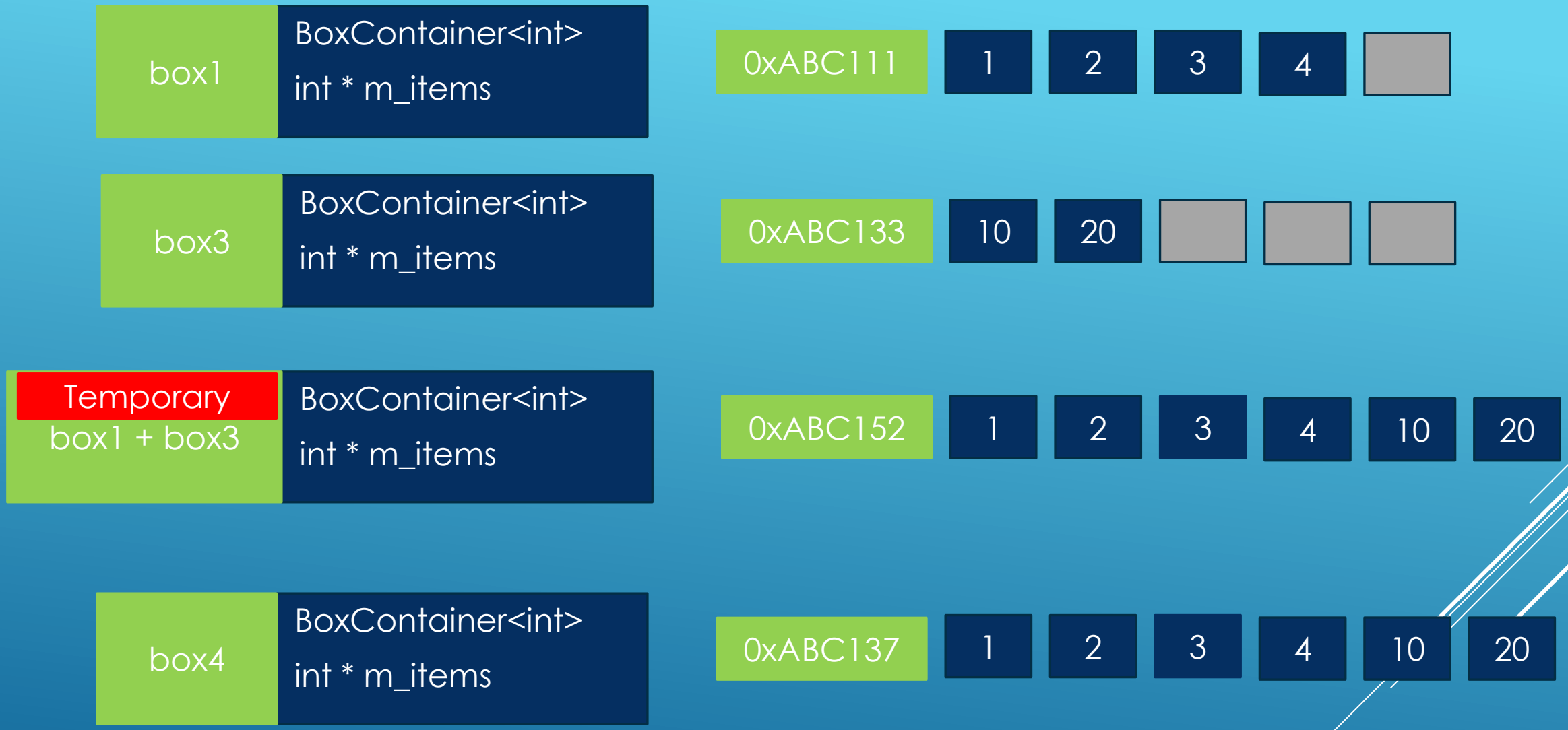
operator+= and operator+

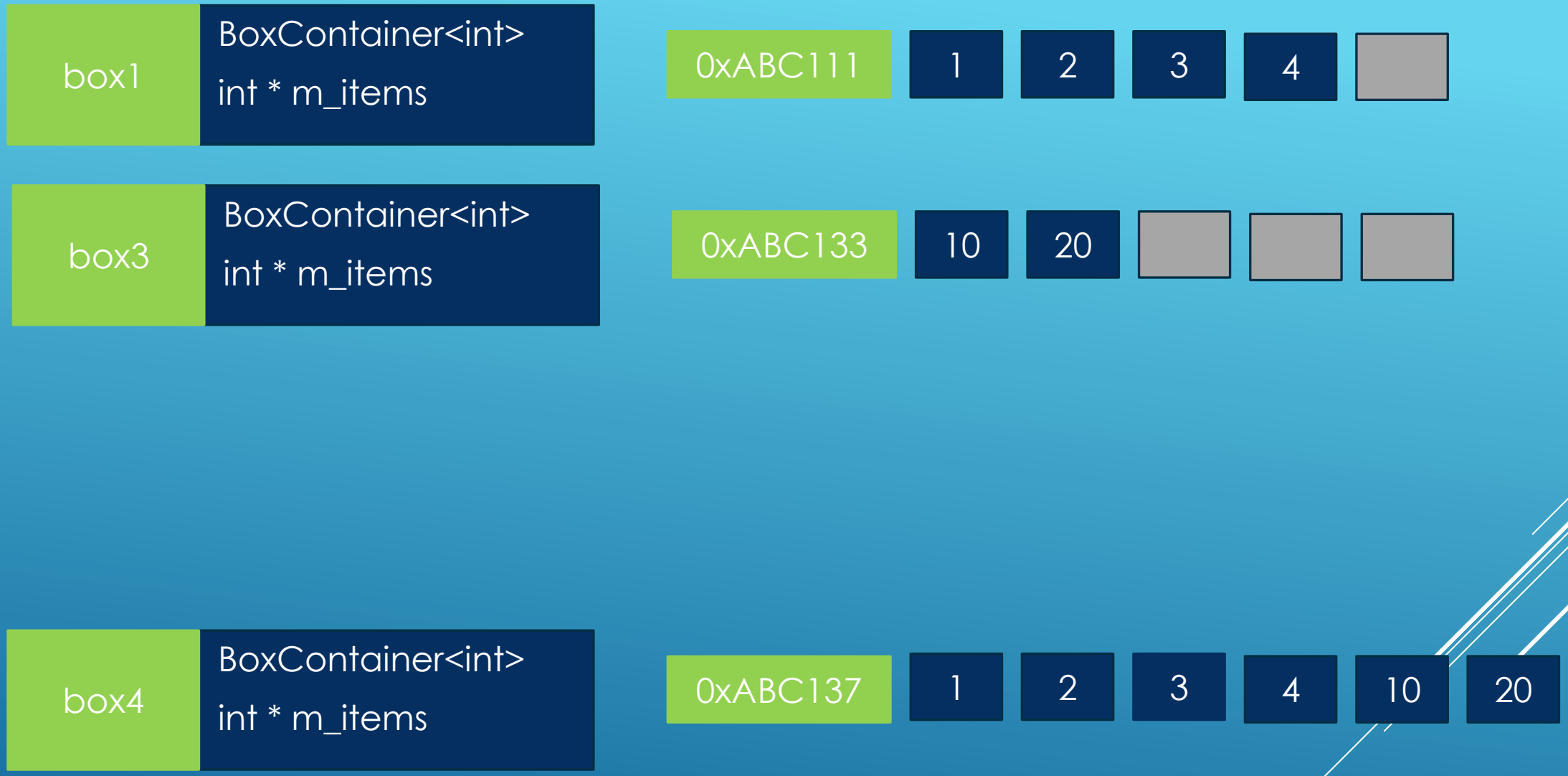
```
template <typename T>
void BoxContainer<T>::operator +=(const BoxContainer<T>& operand){

    //Make sure the current box can acommodate for the added new elements
    if( (m_size + operand.size()) > m_capacity)
        expand(m_size + operand.size());

    //Copy over the elements
    for(size_t i{} ; i < operand.m_size; ++i){
        m_items [m_size + i] = operand.m_items[i];
    }

    m_size += operand.m_size;
}

template <typename T>
BoxContainer<T> operator +(const BoxContainer<T>& left, const BoxContainer<T>& right){
    BoxContainer<T> result(left.size( ) + right.size( ));
    result += left;
    result += right;
    return result;
}
```


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
Temporary
box1 + box3
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20

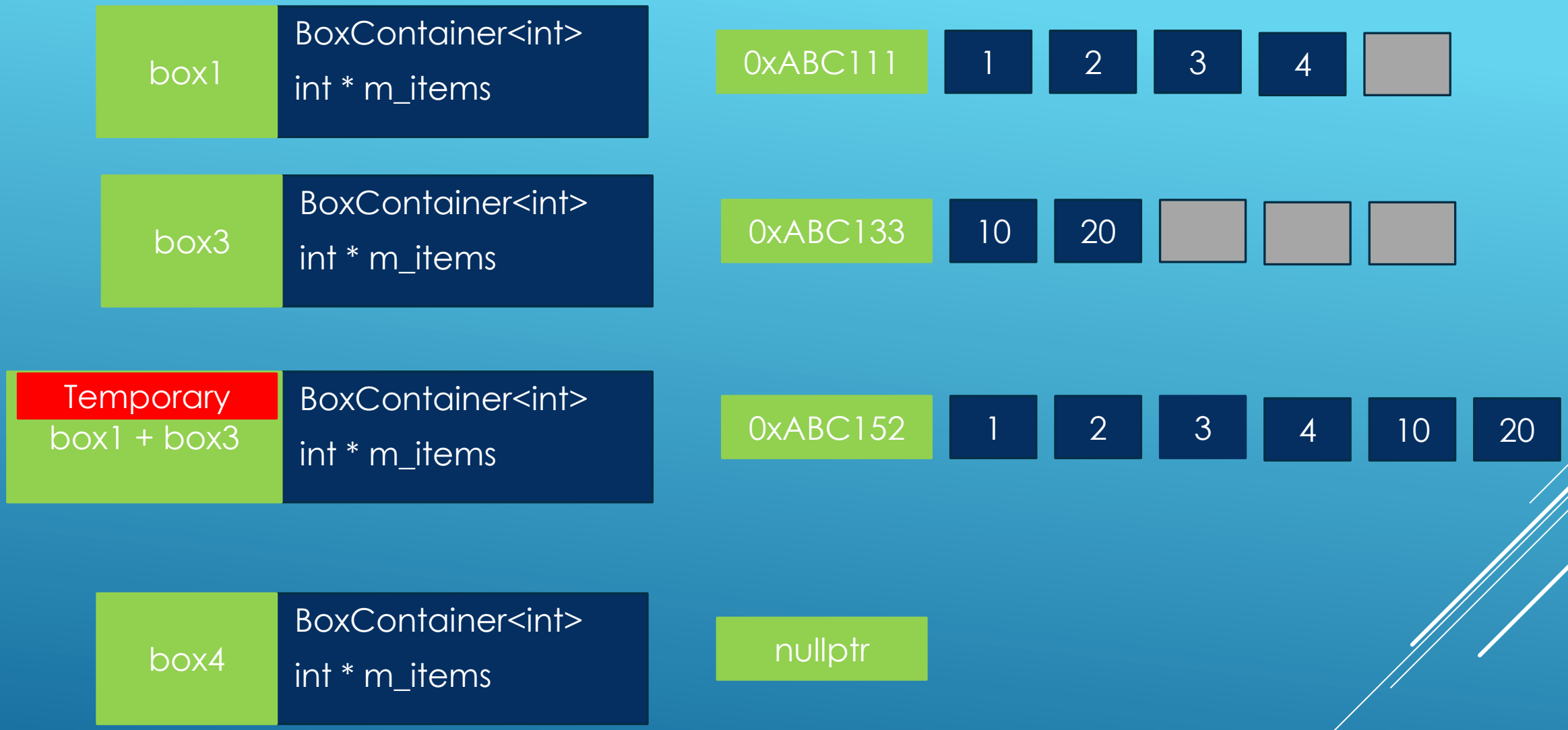

box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
Temporary
box1 + box3
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20
box4
BoxContainer<int>
int * m_items
0xABC137
0
0
0
0
0
0

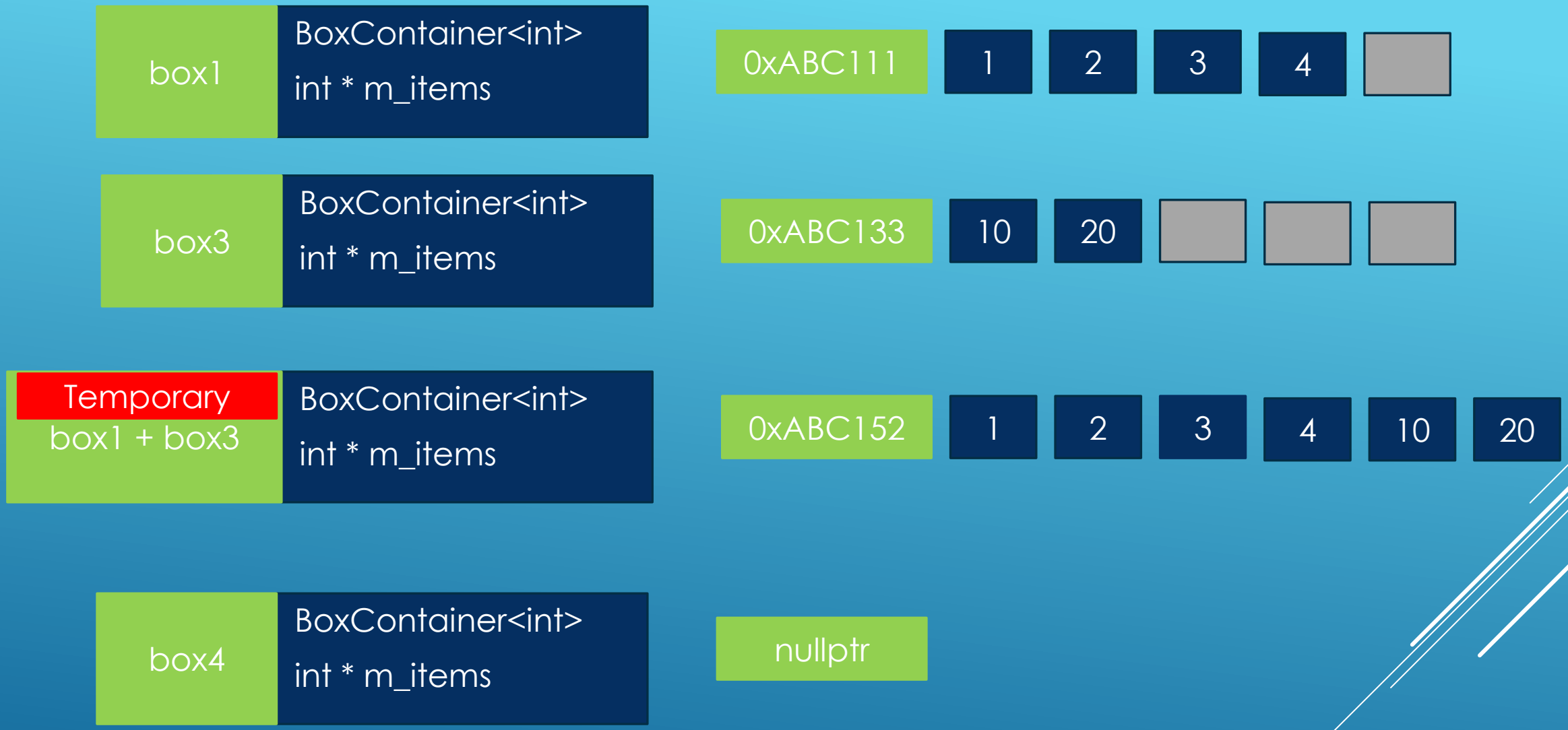

box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
Temporary
box1 + box3
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20
box4
BoxContainer<int>
int * m_items
0xABC137
1
2
3
4
10
20

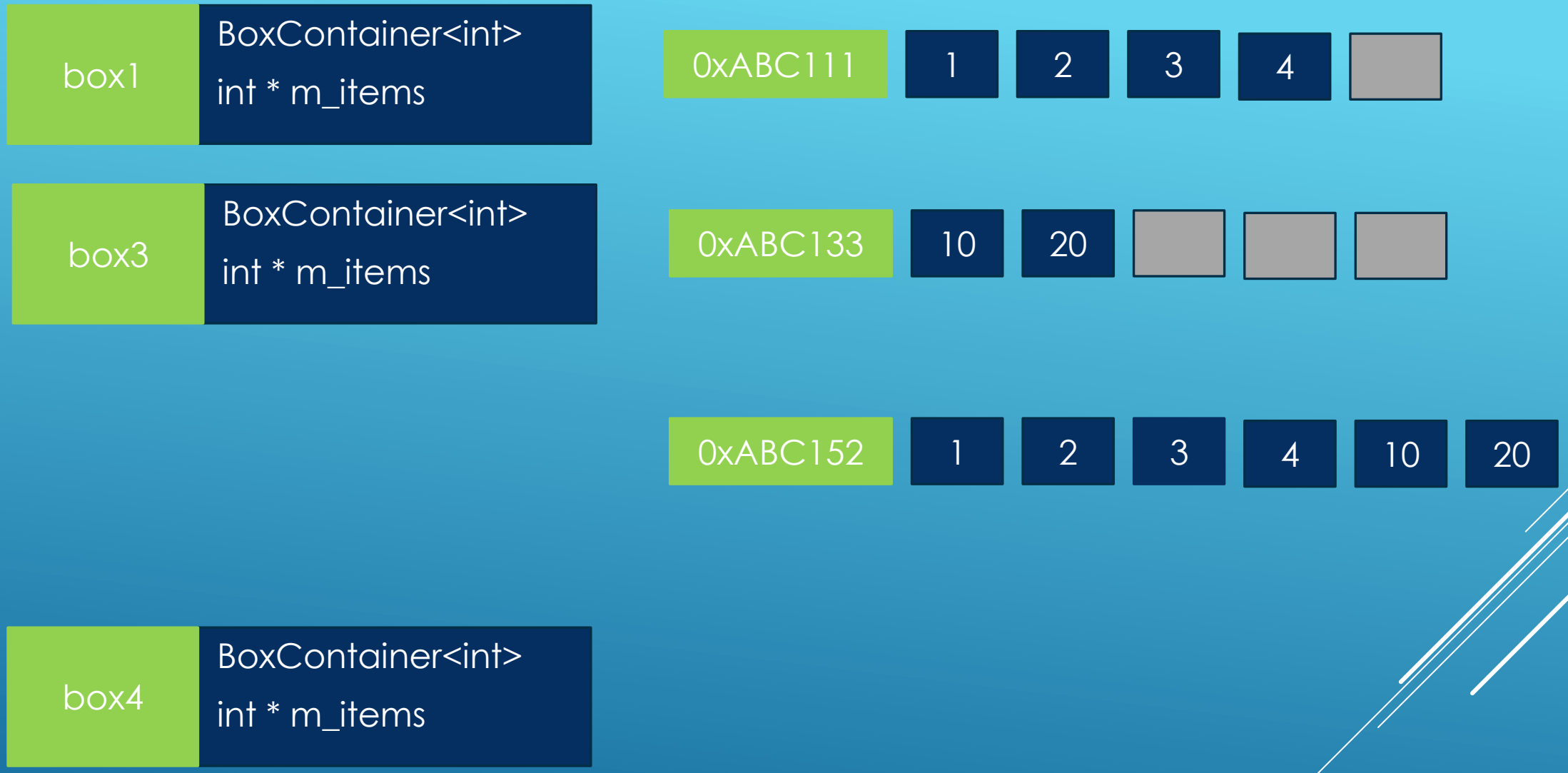

box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
box4
BoxContainer<int>
int * m_items
0xABC137
1
2
3
4
10
20

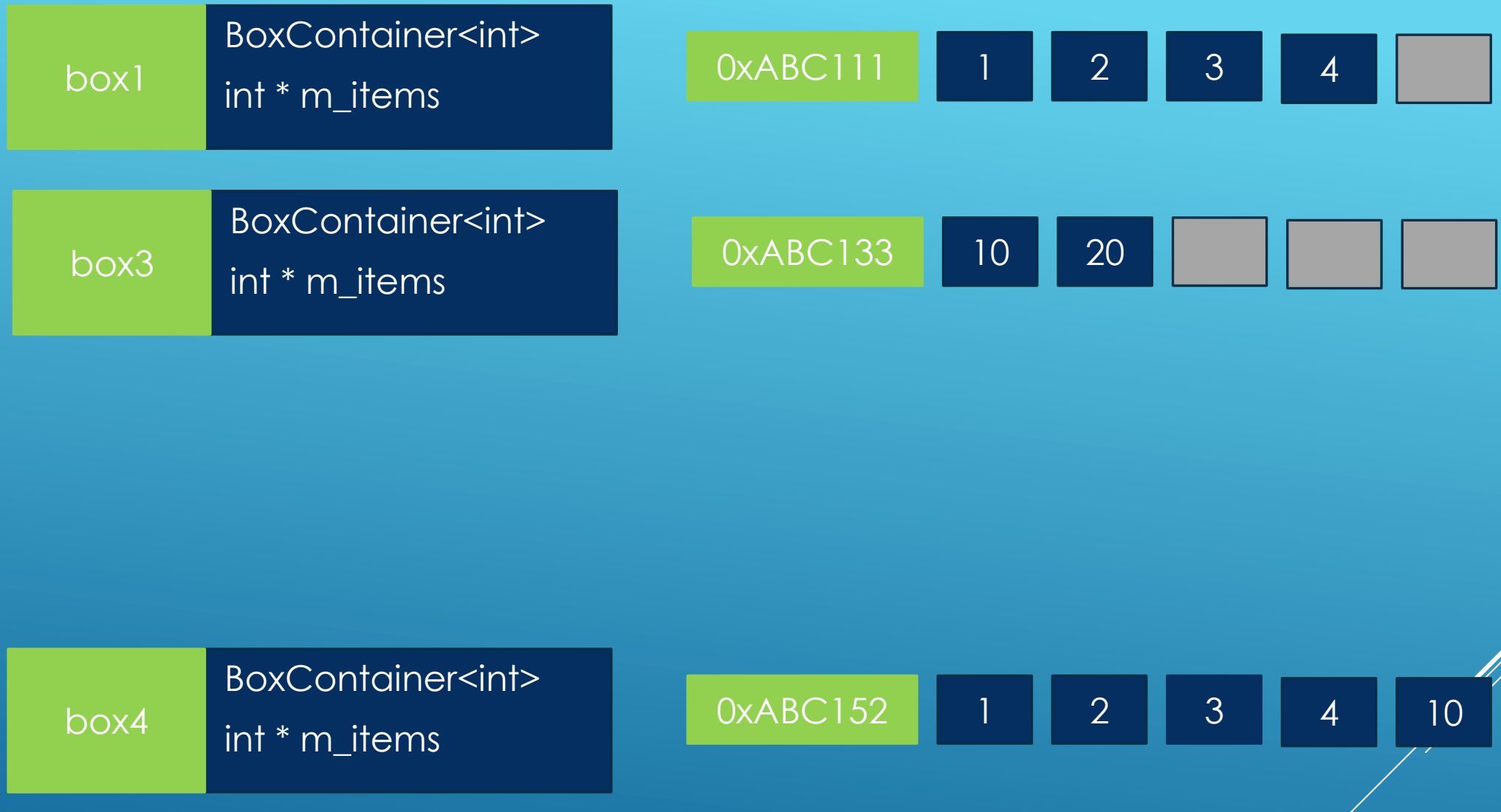
Slide intentionally left empty

Move Semantics


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
Temporary
box1 + box3
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20
box4
BoxContainer<int>
int * m_items
nullptr


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
Temporary
box1 + box3
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20
box4
BoxContainer<int>
int * m_items
nullptr


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
0xABC152
1
2
3
4
10
20
box4
BoxContainer<int>
int * m_items


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
box4
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20

Slide intentionally left empty

Lvalues and Rvalues

- Lvalues are things you can grab an address for and use at a later time
- Rvalues are transient or temporary in nature, they only exist for a short time, and are quickly destroyed by the system when no longer needed

Lvalues

```
int x{5};   // x,y and z are all lvalues, they have a memory address we
int y{10};  //  can retrieve and use later on ,
int z{20};  // as long as the variables are in scope.
```

Rvalues

```
int x{5};   // x,y and z are all lvalues, they have a memory address we
int y{10};  //  can retrieve and use later on ,
int z{20};  // as long as the variables are in scope.


z = x + y; // the result of (x+y) is stored in memory for a short time (transiently)
           //before it's assigned (copied) to z. After the assignment , the memory is discarded
           //(reclaimed by the system). (x+y) is (or evaluates to) an rvalue.
//std::cout << "address of (x+y) : " << &(x+y); // Can't grab the address of an rvalue.
std::cout << "z is : " << z << std::endl;
```

Rvalues

```
double result = add(10.1,20.2); // The result of add(10.1,20.2), is stored in some memory
            //location for a sh ort time, before it's assigned to result, and after it's copied
            //into result, the memory is reclaimed by the system.
            //add(10.1,20.2) is (evaluates to) an lvalue

//std::cout << "address of add(10.1,20.2) : " << &(add(10.1,20.2)) << std::endl;
std::cout << "result is : " << result << std::endl;
```

Rvalues

```
//Grab the addresses for later use

int * ptr1 = &(x + y); // Compiler error. The error clearly says what's wrong here
int * ptr2 = &add(10.1,20.2); // Compiler error. Can only take address of an lvalue,
                              // add(10.1 , 20.2) evaluates to an rvalue.
int* ptr3 = &45;
int* ptr4 = &x; // x is an lvalue, so we can grab its address
```

The benefits for rvalues and become apparent when we pass temporary objects as function parameters. These can be direct temporary objects created on the fly, or those returned from functions or expressions

Slide intentionally left empty

Rvalue references

When an rvalue reference is bound to an rvalue, the life of the rvalue is extended, and we can manipulate it through the rvalue reference

```cpp
double add(double a, double b){
    return a + b;
}

int main(int argc, char **argv)
{
    int x{5};
    int y{10};

    int&& outcome  = x + y; // Extends the lifetime of the temporary result

    double&& result = add(10.1,20.2);

    //Temporary values become usable way down through out the lifetime
    //of the program
    std::cout << "Program doing some other things..." << std::endl;

    std::cout << "outcome is  : " << outcome << std::endl;
    std::cout << "result is : " << result << std::endl;

    return 0;
}
```

Slide intentionally left empty

Moving temporaries around

Copy constructor, copy assignment operator

```
template <typename T>
class BoxContainer
{
    friend std::ostream& operator<< <T> (std::ostream&, const BoxContainer<T>&);
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer<T>& source);//Copy constructor
    ~BoxContainer<T>();
    /* ...
    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source); // Copy assignment operator
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Copy assignment operator called

```
BoxContainer<int> make_box(int modifier){
    BoxContainer<int> local_int_box(20);
    populate_box(local_int_box,modifier);
    return local_int_box;
}

int main(int argc, char **argv)
{
    BoxContainer<int> box_array[2];

    for(size_t i{0} ; i < 2 ; ++i){
        box_array[i] = make_box(i+1);//Copy assignment operator called at each iteration
                                     // We're copying data from the temporary and
                                     // throwing the temporary away (with data)
    }

    //Print out the box
    for(size_t i{};i < 2 ;++i){
        std::cout << "box_array[" << i << "]" << box_array[i] << std::endl;
    }
    return 0;
}
```

Copy constructor

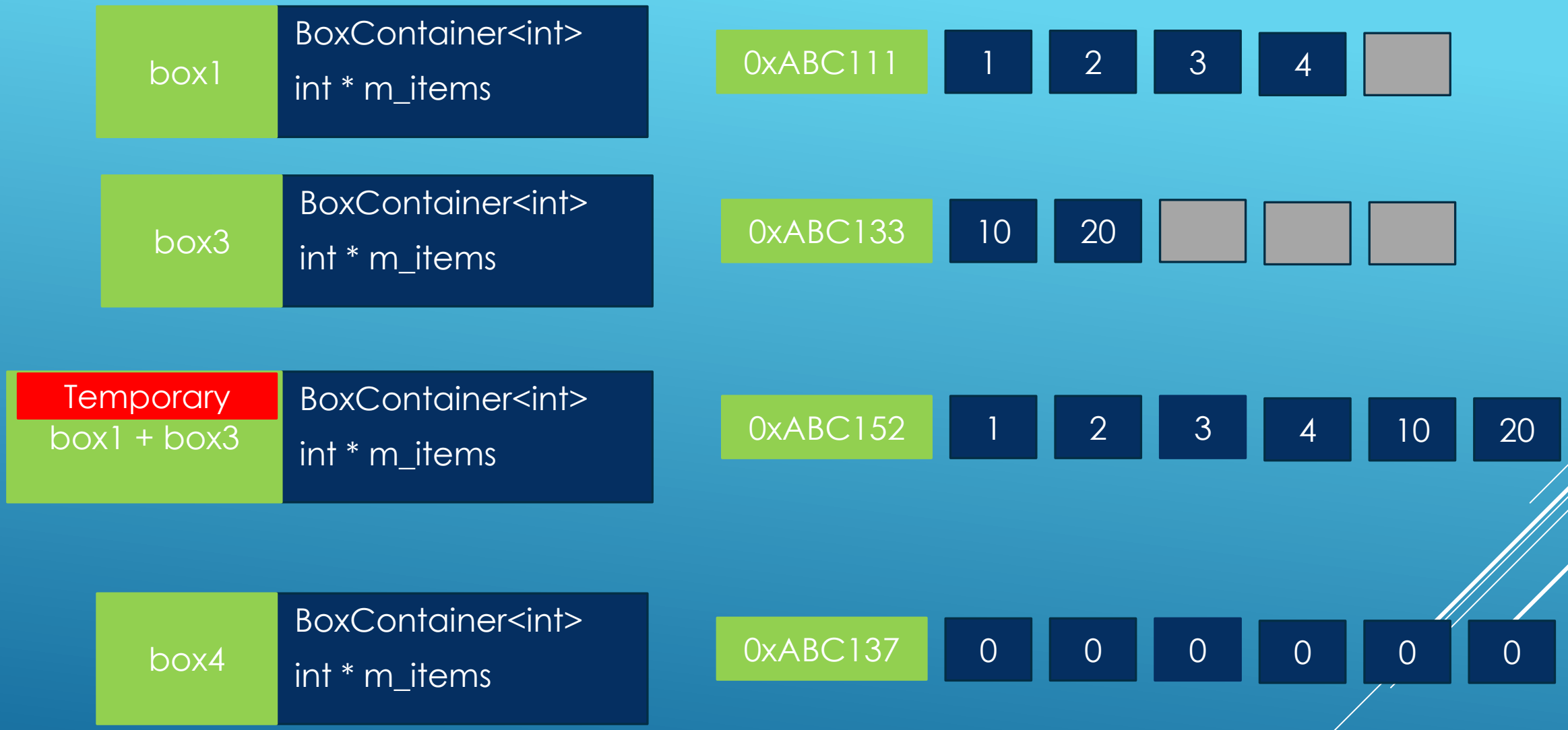
```
//Copy constructor
template <typename T>
BoxContainer<T>::BoxContainer(const BoxContainer<T>& source)
{
    std::cout << "BoxContainer copy constructor called. Copying "
            << source.m_size << " items..." << std::endl;
    //Set up a new box
    m_items = new T[source.m_capacity];
    m_capacity = source.m_capacity;
    m_size = source.m_size;

    //Copy the items over from source
    for(size_t i{} ; i < source.size(); ++i){
        m_items[i] = source.m_items[i];
    }
}
```

Copy assignment operator

```
template <typename T>
void BoxContainer<T>::operator =(const BoxContainer<T>& source){
        std::cout << "BoxContainer copy assignment operator called. Copying "
            << source.m_size << " items..." << std::endl;
    T *new_items;

    // Check for self-assignment:
    if (this == &source)
            return;
/* ...
    if (m_capacity != source.m_capacity)
    {
        new_items = new T[source.m_capacity];
        delete [ ] m_items;
        m_items = new_items;
        m_capacity = source.m_capacity;
    }

    //Copy the items over from source
    for(size_t i{} ; i < source.size(); ++i){
        m_items[i] = source.m_items[i];
    }

    m_size = source.m_size;
}
```

Slide intentionally left empty

Move constructor & move assignment operator


box1
BoxContainer<int>
int * m_items
0xABC111
1
2
3
4
box3
BoxContainer<int>
int * m_items
0xABC133
10
20
Temporary
box1 + box3
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20
box4
BoxContainer<int>
int * m_items
0xABC137
0
0
0
0
0
0

Move constructor, move assignment operator

```
template <typename T>
class BoxContainer
{
    friend std::ostream& operator<< <T> (std::ostream&, const BoxContainer<T>&);
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer<T>& source);
    BoxContainer(BoxContainer&& source); // Move constructor
    ~BoxContainer<T>();
    /* ...
    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source); // Copy assignment operator
    void operator=(BoxContainer<T>&& source); // Move assignment operator


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

Moving temporaries around : This time better.

```
BoxContainer<int> make_box(int modifier){
    BoxContainer<int> local_int_box(20);
    populate_box(local_int_box,modifier);
    return local_int_box;
}

int main(int argc, char **argv)
{

    BoxContainer<int> box_array[5];

    for(size_t i{0} ; i < 5 ; ++i){
        box_array[i] = make_box(i+1);//Move assignment operator called at each iteration
                                     // We're stealing data from the temporary and
                                     // throwing the (shell) temporary away (with no data)
    }

    std::cout << "box_array[0] : " << box_array[0] << std::endl;

    return 0;
}
```

Move constructor : parameter is rvalue reference

```
//Move constructor
template <typename T>
BoxContainer<T>::BoxContainer(BoxContainer&& source){

    // Check for construction from self:
    if (this == &source)
            return;

    m_items = source.m_items;
    m_size = source.m_size;
    m_capacity = source.m_capacity;

    //Remember to invalidate source
    source.invalidate();

}
```

Move assignment operator: parameter is rvalue reference

```
//Move assignment operator
template <typename T>
void BoxContainer<T>::operator=(BoxContainer&& source){

        std::cout << "BoxContainer move assignment operator called. Moving "
            << source.m_size << " items..." << std::endl;
    // Check for self assignment
    if (this == &source)
            return;

    m_items = source.m_items;
    m_size = source.m_size;
    m_capacity = source.m_capacity;

    //Remember to invalidate source
    source.invalidate();
}
```

Slide intentionally left empty

Moving Lvalues with std::move

Bad copies

```
template<class T>
void swap_data(T& a, T& b)
{
  T temp { a }; // invokes copy constructor
  a = b; // invokes copy assignment
  b = temp; // invokes copy assignment
}
int main(int argc, char **argv)
{
    BoxContainer<int> box1;
    populate_box(box1,2);
    BoxContainer<int> box2;
    populate_box(box2,15);

    std::cout << "box1 : " << box1 << std::endl;
    std::cout << "box2 : " << box2 << std::endl;

    swap_data(box1,box2);

    std::cout << "box1 : " << box1 << std::endl;
    std::cout << "box2 : " << box2 << std::endl;

    return 0;
}
```

Good moves

```
template<class T>
void swap_data(T& a, T& b)
{
  T temp { std::move(a) }; // invokes move constructor
  a = std::move(b); // invokes move assignment operator
  b = std::move(temp); // invokes move assignment operator
}
int main(int argc, char **argv)
{
    BoxContainer<int> box1;
    populate_box(box1,2);
    BoxContainer<int> box2;
    populate_box(box2,15);

    std::cout << "box1 : " << box1 << std::endl;
    std::cout << "box2 : " << box2 << std::endl;

    swap_data(box1,box2);

    std::cout << "box1 : " << box1 << std::endl;
    std::cout << "box2 : " << box2 << std::endl;

    return 0;
}
```

- std::move doesn't move data by itself, it just casts its parameter to an rvalue
- The moving of data is done when we construct an object from the resulting rvalue or if we assign it to another object of our class

Slide intentionally left empty

Invalidating pointers in stolen from objects

Move Constructor

```
//Move constructor
template <typename T>
BoxContainer<T>::BoxContainer(BoxContainer&& source){

    // Check for construction from self:
    if (this == &source)
            return;

    m_items = source.m_items;
    m_size = source.m_size;
    m_capacity = source.m_capacity;

    //Remember to invalidate source
    source.invalidate();
}
```

Move assignment operator

```
//Move assignment operator
template <typename T>
void BoxContainer<T>::operator=(BoxContainer&& source){

    std::cout << "BoxContainer move assignment operator called. Moving "
              << source.m_size << " items..." << std::endl;
    // Check for self assignment
    if (this == &source)
        return;

    m_items = source.m_items;
    m_size = source.m_size;
    m_capacity = source.m_capacity;

    //Remember to invalidate source
    source.invalidate();
}
```

```
template <typename T>
class BoxContainer
{
    /* ...
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
        /* ...
    void invalidate(){
        m_items = nullptr;
        m_size =0;
        m_capacity =0;
    }
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

```
BoxContainer<int> box1;
populate_box(box1,2);

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

BoxContainer<int> box2(std::move(box1));

std::cout << "box2 : " << box2 << std::endl;
std::cout << "box1 : " << box1 << std::endl;
```

Slide intentionally left empty

Move only types

Move only type

A type whose copy constructor and copy assignment operator have been deleted. Its objects can't be copied. The move constructor and move assignment operator are however left in. Its objects can only be moved.

Move only types

```
template <typename T>
class BoxContainer
{
    /* ...
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer<T>& source) = delete;
    BoxContainer(BoxContainer&& source); // Move constructor
    ~BoxContainer<T>();
    /* ...
    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source) = delete; // Copy assignment operator
    void operator=(BoxContainer<T>&& source); // Move assignment operator
/* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Slide intentionally left empty

Passing by rvalue reference

Rvalue references with a name

If by any chance, an rvalue happens to be assigned a name, it's treated by the compiler as if it's an lvalue. In other words, if it's assigned or copy assigned from, the copy constructor or the copy assignment operator will be called.

Item

```
class Item{
    friend std::ostream& operator<<( std::ostream& out, const Item& operand);
public :
    Item() : m_data{new int} { ...
    Item(int value) : m_data{new int(value)}{ ...
    //Copy Members
    Item( const Item& source) : m_data{new int}{ ...
    Item& operator=(const Item& right_operand){ ...

    //Move Members
    Item( Item&& source){ ...
    Item& operator=(Item&& right_operand){ ...
private :
    int * m_data{nullptr};
};
```

Simple assignment

```
Item&& rvalue_ref {get_value()};
//Direct assignments
Item item1;
item1 = rvalue_ref; // Copy assignment operator called, rvalue_ref
                    // temporary has a name, compiler treats it as
                    // an lvalue
item1 = std::move(rvalue_ref); // Calls the move assignment operator
```

Passing to a function

```
void do_something( Item&& item){
    std::cout << "Do something move version called..." << std::endl;
   Item internal = item;
   // Item internal = std::move(item);
   std::cout << "internal : " << internal << std::endl;
}
int main(int argc, char **argv)
{
    Item&& rvalue_ref {get_value()};
    do_something(rvalue_ref);  // Compiler error, can't pass an lvalue.
                               // Again, our temporary has a name, it's treated
                               //like an lvalue, and can't be passed where an
                               //rvalue is expected. Hence the error.
    do_something(std::move(rvalue_ref));
    return 0;
}
```

BoxContainer::add()

```
template <typename T>
void BoxContainer<T>::add( T&& item){
    std::cout << "Move version of add called..." << std::endl;
    if (m_size == m_capacity)
        expand(m_size + EXPAND_STEPS);
    //m_items[m_size] = item;
    m_items[m_size] = std::move(item);
    ++m_size;
}
```

Move Semantics


Temporary
BoxContainer<int>
int * m_items
0xABC152
1
2
3
4
10
20

- Lvalues are things you can grab an address for and use at a later time
- Rvalues are transient or temporary in nature, they only exist for a short time, and are quickly destroyed by the system when no longer needed

Rvalue references

```
double add(double a, double b){
    return a + b;
}

int main(int argc, char **argv)
{
    int x{5};
    int y{10};

    int&& outcome  = x + y; // Extends the lifetime of the temporary result

    double&& result = add(10.1,20.2);

    //Temporary values become usable way down through out the lifetime
    //of the program
    std::cout << "Program doing some other things..." << std::endl;

    std::cout << "outcome is  : " << outcome << std::endl;
    std::cout << "result is : " << result << std::endl;

    return 0;
}
```

Slide intentionally left empty

Moving temporaries around

Copy constructor, copy assignment operator

```
template <typename T>
class BoxContainer
{
    friend std::ostream& operator<< <T> (std::ostream&, const BoxContainer<T>&);
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer<T>& source);//Copy constructor
    ~BoxContainer<T>();
    /* ...
    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source); // Copy assignment operator
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Slide intentionally left empty

Move constructor & move assignment operator

Move constructor, move assignment operator

```
template <typename T>
class BoxContainer
{
    friend std::ostream& operator<< <T> (std::ostream&, const BoxContainer<T>&);
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer<T>& source);
    BoxContainer(BoxContainer&& source); // Move constructor
    ~BoxContainer<T>();
    /* ...
    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source); // Copy assignment operator
    void operator=(BoxContainer<T>&& source); // Move assignment operator


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

Stealing from lvalues

```
template<class T>
void swap_data(T& a, T& b)
{
  T temp { std::move(a) }; // invokes move constructor
  a = std::move(b); // invokes move assignment operator
  b = std::move(temp); // invokes move assignment operator
}
int main(int argc, char **argv)
{
    BoxContainer<int> box1;
    populate_box(box1,2);
    BoxContainer<int> box2;
    populate_box(box2,15);

    std::cout << "box1 : " << box1 << std::endl;
    std::cout << "box2 : " << box2 << std::endl;

    swap_data(box1,box2);

    std::cout << "box1 : " << box1 << std::endl;
    std::cout << "box2 : " << box2 << std::endl;

    return 0;
}
```

Leave shell objects in a good state

```
BoxContainer<int> box1;
populate_box(box1,2);

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

BoxContainer<int> box2(std::move(box1));

std::cout << "box2 : " << box2 << std::endl;
std::cout << "box1 : " << box1 << std::endl;
```

Move only types

```
template <typename T>
class BoxContainer
{
    /* ...
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer<T>& source) = delete;
    BoxContainer(BoxContainer&& source); // Move constructor
    ~BoxContainer<T>();
    /* ...
    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source) = delete; // Copy assignment operator
    void operator=(BoxContainer<T>&& source); // Move assignment operator
/* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Rvalue reference parameters

```
Item&& rvalue_ref {get_value()};
//Direct assignments
Item item1;
item1 = rvalue_ref; // Copy assignment operator called, rvalue_ref
                    // temporary has a name, compiler treats it as
                    // an lvalue
item1 = std::move(rvalue_ref); // Calls the move assignment operator
```

Rvalue reference parameters

```
template <typename T>
void BoxContainer<T>::add( T&& item){
    std::cout << "Move version of add called..." << std::endl;
    if (m_size == m_capacity)
        expand(m_size + EXPAND_STEPS);
    //m_items[m_size] = item;
    m_items[m_size] = std::move(item);
    ++m_size;
}
```

Slide intentionally left empty