Overview of C++ Containers

including initialization samples

June 2021

Kevin Perez

C++ containers, or objects that hold other objects, are implemented in a framework known as the Standard Template Library. These containers are useful for writing generic code that can work with almost any data type as long as the type supports functions performed by the template class. The first section below has short descriptions about how the container classes work, what types of constructor prototypes (borrowed from `www.cplusplus.com`) are available for each container, and code using some of each of the containers' constructors. Afterwards follows a section dedicated to a handful of functions from the STL algorithms library with short descriptions and sample programs.

1 Containers

1.1 vector

A vector has a lot of the same properties of a C-style array: it stores elements sequentially and contiguously in memory, it allows random access to elements using subscripts, and all the elements it holds has to be of the same data type. Unlike a C-style array, the size of a vector does not need to be specified at compile time. Vectors can be expanded at runtime if more memory is required to store elements, but this procedure is completed in amortized time because it requires reallocation of memory which can double the amount of memory being used during the copy procedure and cause the system to crash.

Listing 1: vector constructor prototypes from C++14 standards

```
/* default constructor */
vector();

/* empty constructor */
explicit vector (const allocator_type& alloc);
explicit vector (size_type n, const allocator_type& alloc = allocator_type());
vector (size_type n, const value_type& val,
        const allocator_type& alloc = allocator_type());

/* range constructor */
template <class InputIterator>
vector (InputIterator first, InputIterator last,
        const allocator_type& alloc = allocator_type());

/* copy constructors */
vector (const vector& x);
vector (const vector& x, const allocator_type& alloc);

/* move constructors */
vector (vector&& x);
vector (vector&& x, const allocator_type& alloc);

/* initializer-list constructor */
vector (initializer_list<value_type> il,
        const allocator_type& alloc = allocator_type());
```

Listing 2: Example using vector constructors

```
#include <iostream>
#include <vector>

class TestClass
```

```
5  {
6  public:
7    static int id;
8    TestClass()
9    {
10     std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
11   }
12 };
13
14 int TestClass::id = 0;
15
16 void probe_vector(std::vector<TestClass>& v)
17 {
18   std::cout << "\nVector object capacity evaluation" << std::endl;
19   std::cout << "\tIs empty?: ";
20   (v.empty() ? std::cout << "True\n" : std::cout << "False\n");
21   std::cout << "\tSize: " << v.size() << " elements" << std::endl;
22   std::cout << "\tMax size: " << v.max_size() << " elements" << std::endl;
23   std::cout << "\tCapacity: " << v.capacity() << " elements\n" << std::endl;
24 }
25
26 int main(void)
27 {
28   std::cout << "Initializing vector object using the default constructor\n";
29   std::vector<TestClass> vec;
30   probe_vector(vec);
31
32   std::cout << "Initializing vector object using explicit fill constructor\n\n";
33   std::vector<TestClass> vec2(5);
34   probe_vector(vec2);
35
36   std::cout << "Initializing vector object using a copy constructor\n\n";
37   std::vector<TestClass> vec3 = vec2;
38   query_vector(vec3);
39 }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos$ ./vector.x
Initializing vector object using the default constructor.

Vector object capacity evaluation:
        Is empty?: True
        Size: 0 elements
        Max size: 18446744073709551615 elements
        Capacity: 0 elements

Initializing vector object using explicit fill constructor.

        In TestClass default constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass default constructor. (id: 2)
        In TestClass default constructor. (id: 3)
        In TestClass default constructor. (id: 4)

Vector object capacity evaluation:
        Is empty?: False
        Size: 5 elements
        Max size: 18446744073709551615 elements
        Capacity: 5 elements

Initializing vector object using a copy constructor.


Vector object capacity evaluation:
        Is empty?: False
        Size: 5 elements
        Max size: 18446744073709551615 elements
        Capacity: 5 elements
```

1.2 list

A list container is implemented as a linked list data structure where each element is a node allocated in memory containing an object and pointers to the next and previous nodes, if any, which means its elements can consume more memory than a vector's element. The memory location of other elements in a list are not affected when other elements in the list are added or deleted which is especially true when inserting a new element in-between other elements in the sequential linked-list sequence. A benefit of using a list is that insertion and deletion of an element requires only changing a few pointers connecting the sequence of elements, at least when a pointer to the insertion/deletion position is already known. And although it may not seem so, traversing a list is slower because of pointer indirection, that is, the need to dereference pointers consumes CPU time.

Listing 3: list constructor prototypes based on C++14 standards

```
1  /* Default constructors */
2  list();
3
4  /* empty constructor */
5  explicit list (const allocator_type& alloc);
6
7  /* fill constructors */
8  explicit list (size_type n, const allocator_type& alloc = allocator_type());
9  list (size_type n, const value_type& val,
10       const allocator_type& alloc = allocator_type());
```

```
11
12 /* range constructors */
13 template <class InputIterator>
14 list (InputIterator first, InputIterator last,
15       const allocator_type& alloc = allocator_type());
16
17 /* copy constructors */
18 list (const list& x);
19 list (const list& x, const allocator_type& alloc);
20
21 /* move constructors */
22 list (list&& x);
23 list (list&& x, const allocator_type& alloc);
24
25 /* initializer_list constructor */
26 list (initializer_list<value_type> il,
27       const allocator_type& alloc = allocator_type());
```

Listing 4: example using list constructors

```
1  #include <iostream>
2  #include <list>
3  #include <memory>
4
5  class TestClass
6  {
7    int id_;
8  public:
9    static int id;
10   TestClass() : id_(id)
11   {
12     std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
13   }
14   TestClass(const TestClass& t) : id_(t.id_)
15   {
16     std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
17   }
18 };
19
20 int TestClass::id = 0;
21
22 void probe_list(std::list<TestClass>& l)
23 {
24   std::cout << "\nList object evaluation" << std::endl;
25   std::cout << "\tObject memory location: " << &l << std::endl;
26   if(!l.empty())
27     std::cout << "\tFirst element memory location: " << &l.front() << std::endl;
28   std::cout << "\tIs empty?: ";
29   (l.empty() ? std::cout << "True\n" : std::cout << "False\n");
30   std::cout << "\tSize: " << l.size() << " elements" << std::endl;
31   std::cout << "\tMax size: " << l.max_size() << " elements\n" << std::endl;
32 }
33
34 int main(void)
35 {
36   std::cout << "Initializing list object (list1) using default constructor\n";
37   std::list<TestClass> list1;
38   probe_list(list1);
39
40   std::cout << "Initializing list object (list2) using fill constructor\n\n";
41   std::list<TestClass> list2(3, *(std::make_unique<TestClass>()));
```

```
42    probe_list(list2);
43
44    std::cout << "Initializing list object (list3) using range constructor\n\n";
45    std::list<TestClass> list3(list2.begin(), list2.end());
46    probe_list(list3);
47
48    std::cout << "Initializing list object (list4) using move constructor\n\n";
49    std::list<TestClass> list4(std::move(list2));
50    probe_list(list4);
51  }
```

```
.debian:~/Documents/EEL3370/homeworks/container_demos/list$ ./list.x
Initializing list object (list1) using default constructor

List object evaluation
        Object memory location: 0x7ffeb4ca2840
        Is empty?: True
        Size: 0 elements
        Max size: 768614336404564650 elements

Initializing list object (list2) using fill constructor

        In TestClass default constructor. (id: 0)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 0)

List object evaluation
        Object memory location: 0x7ffeb4ca2820
        First element memory location: 0x55df5c6512b0
        Is empty?: False
        Size: 3 elements
        Max size: 768614336404564650 elements

Initializing list object (list3) using range constructor

        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 0)

List object evaluation
        Object memory location: 0x7ffeb4ca2800
        First element memory location: 0x55df5c651290
        Is empty?: False
        Size: 3 elements
        Max size: 768614336404564650 elements

Initializing list object (list4) using move constructor


List object evaluation
        Object memory location: 0x7ffeb4ca27e0
        First element memory location: 0x55df5c6512b0
        Is empty?: False
        Size: 3 elements
        Max size: 768614336404564650 elements
```

1.3 queue

The queue container is known as a container adapter The queue container provides methods for inserting into the back and removing from the front of a sequential container; hence it is effectively ascribed the acronym *first in, first out*. It is a container adapter, rather than simply a container, because it provides an interface to an underlying container that represents the actual data structure. The container that manages the data can be assigned at the time a queue object is declared.

Listing 5: queue constructor prototypes from C++11 standards

```
/* default constructor */
queue();

/* copy constructors */
template <class Alloc>
queue (const queue& x, const Alloc& alloc);
explicit queue (const container_type& ctnr);
template <class Alloc>
queue (const container_type& ctnr, const Alloc& alloc);

/* move constructors */
explicit queue (container_type&& ctnr = container_type());
template <class Alloc>
queue (queue&& x, const Alloc& alloc);
template <class Alloc>
queue (container_type&& ctnr, const Alloc& alloc);

/* empty constructor */
template <class Alloc>
explicit queue (const Alloc& alloc);
```

Listing 6: Example using queue constructors

```
#include <iostream>
#include <list>
#include <memory>
#include <queue>

class TestClass;
typedef std::list<TestClass> TestClassList;

class TestClass
{
  int id_;
public:
  static int id;
  TestClass() : id_(id)
  {
    std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
  }
  TestClass(const TestClass& t) : id_(t.id_)
  {
    std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
  }
};

int TestClass::id = 0;

template<typename T>
```

```
27 void probe_queue(T& q)
28 {
29   std::cout << "\nQueue object evaluation" << std::endl;
30   std::cout << "\tObject memory location: " << &q << std::endl;
31   if(!q.empty())
32     std::cout << "\tFirst element memory location: " << &q.front() << std::endl;
33   std::cout << "\tIs empty?: ";
34   (q.empty() ? std::cout << "True\n" : std::cout << "False\n");
35   std::cout << "\tSize: " << q.size() << " elements\n" << std::endl;
36 }
37
38 int main(void)
39 {
40   std::cout << "Initializing queue object (queue1) using default constructor\n";
41   std::queue<TestClass, TestClassList> queue1;
42   probe_queue(queue1);
43
44   std::cout << "Creating a list of TestClass objects\n";
45   auto list = std::unique_ptr<TestClassList>(new TestClassList(3));
46   std::cout << std::endl;
47
48   std::cout << "Initializing queue object (queue2) explicitly using a list
       containing three objects\n";
49   std::queue<TestClass, TestClassList> queue2(*list);
50   probe_queue(queue2);
51
52   std::cout << "Initializing queue object (queue3) implicitly with a non-empty
       queue object\n";
53   std::queue<TestClass, TestClassList> queue3 = queue2;
54   probe_queue(queue3);
55
56   std::cout << "Initializing queue object (queue4) using a list containing three
        objects\n";
57   std::queue<TestClass, TestClassList> queue4 = std::move(queue2);
58   probe_queue(queue4);
59 }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos/queue$ ./queue.x
Initializing queue object (queue1) using default constructor

Queue object evaluation
        Object memory location: 0x7ffdc04a1520
        Is empty?: True
        Size: 0 elements

Creating a list of TestClass objects
        In TestClass default constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass default constructor. (id: 2)

Initializing queue object (queue2) explicitly using a list of three objects
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)

Queue object evaluation
        Object memory location: 0x7ffdc04a1500
        First element memory location: 0x563b60bf5310
        Is empty?: False
        Size: 3 elements

Initializing queue object (queue3) implicitly with a non-empty queue object
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)

Queue object evaluation
        Object memory location: 0x7ffdc04a14e0
        First element memory location: 0x563b60bf5370
        Is empty?: False
        Size: 3 elements

Initializing queue object (queue4) using a list containing three objects

Queue object evaluation
        Object memory location: 0x7ffdc04a14c0
        First element memory location: 0x563b60bf5310
        Is empty?: False
        Size: 3 elements
```

1.4 stack

A stack is another container adapter with an interface limited to providing operations for adding and removing elements from the back of a container; thus, its storage structure is often described by the acronym *last in, first out*. Stacks can grow dynamically to have as many elements as the computer allows. A stack should be verified as not empty before "popping" an element, otherwise the stack may "underflow" because unallocated memory may be accessed.

Listing 7: stack constructor prototypes from C++14 standards

```
/* default constructor */
stack();

/* empty constructor */
template <class Alloc>
explicit stack (const Alloc& alloc);

/* initializer constructors */
explicit stack (const container_type& ctnr);
template <class Alloc> stack (const container_type& ctnr, const Alloc& alloc);

/* move constructors */
template <class Alloc>
stack (stack&& x, const Alloc& alloc);
template <class Alloc>
stack (container_type&& ctnr, const Alloc& alloc);
explicit stack (container_type&& ctnr = container_type());

/* copy constructors */
template <class Alloc> stack (const stack& x, const Alloc& alloc);
```

Listing 8: Example using stack constructors

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>

class TestClass;
typedef std::vector<TestClass> TestClassVector;

class TestClass
{
  int id_;
public:
  static int id;
  TestClass() : id_(id)
  {
    std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
  }
  TestClass(const TestClass& t) : id_(t.id_)
  {
    std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
  }
};

int TestClass::id = 0;

template<typename T>
void probe_stack(T& s)
{
  std::cout << "\nStack object evaluation" << std::endl;
  std::cout << "\tObject memory location: " << &s << std::endl;
  if(!s.empty())
    std::cout << "\tFirst element memory location: " << &s.top() << std::endl;
  std::cout << "\tIs empty?: ";
  (s.empty() ? std::cout << "True\n" : std::cout << "False\n");
  std::cout << "\tSize: " << s.size() << " elements\n" << std::endl;
```

```
36 }
37
38 int main(void)
39 {
40   std::cout << "Initializing stack object (stack1) using default constructor\n";
41   std::stack<TestClass> stack1;
42   probe_stack(stack1);
43
44   std::cout << "Declaring a vector that initializes three TestClass elements\n";
45   TestClassVector tvec(3);
46   std::cout << std::endl;
47
48   std::cout << "Initializing stack object (stack2) using a copy constructor\n";
49   std::stack<TestClass, TestClassVector> stack2(tvec);
50   probe_stack(stack2);
51
52   std::cout << "Initializing stack object (stack3) using a move constructor\n";
53   std::stack<TestClass, TestClassVector> stack3(std::move(stack2));
54   probe_stack(stack3);
55 }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos/stack$ ./stack.x
Initializing stack object (stack1) using default constructor

Stack object evaluation
        Object memory location: 0x7ffc6a6ff750
        Is empty?: True
        Size: 0 elements

Declaring a vector that initializes three TestClass elements
        In TestClass default constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass default constructor. (id: 2)

Initializing stack object (stack2) using a copy constructor
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)

Stack object evaluation
        Object memory location: 0x7ffc6a6ff710
        First element memory location: 0x55c627d0e508
        Is empty?: False
        Size: 3 elements

Initializing stack object (stack3) using a move constructor

Stack object evaluation
        Object memory location: 0x7ffc6a6ff6f0
        First element memory location: 0x55c627d0e508
        Is empty?: False
        Size: 3 elements
```

1.5 set

A set is an ordered associative container where each element represents a unique constant value that depends on itself rather than a key for being sorted. A set data structure is usually implemented as a binary search tree, specifically a red-black tree, which provides a fast search

operation with logarithmic asymptotic run times which assists with fast insertion and deletion of elements. The default comparison function used by a set object, which is the less-than operator, can be changed when a set is declared.

Listing 9: set constructor prototypes from C++14 standards

```
/* default */
set();

/* empty constructors */
explicit set (const key_compare& comp,
              const allocator_type& alloc = allocator_type());
explicit set (const allocator_type& alloc);

/* range constructors */
template <class InputIterator>
set (InputIterator first, InputIterator last,
     const key_compare& comp = key_compare(),
     const allocator_type& = allocator_type());
template <class InputIterator>
set (InputIterator first, InputIterator last,
     const allocator_type& = allocator_type());

/* copy constructors */
set (const set& x);
set (const set& x, const allocator_type& alloc);

/* move constructors */
set (set&& x);
set (set&& x, const allocator_type& alloc);

/* initializer-list constructors */
set (initializer_list<value_type> il,
     const key_compare& comp = key_compare(),
     const allocator_type& alloc = allocator_type());
set (initializer_list<value_type> il,
     const allocator_type& alloc = allocator_type());
```

Listing 10: Example using set constructors

```
#include <iostream>
#include <list>
#include <memory>
#include <set>

class TestClass;

class TestClass
{
  int id_;
public:
  static int id;
  TestClass() : id_(id)
  {
    std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
  }
  TestClass(const TestClass& t) : id_(t.id_)
  {
    std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
  }
```

```
21    bool operator<(const TestClass& a) const
22    {
23      return id_ < a.id_;
24    }
25  };
26
27  int TestClass::id = 0;
28
29  template<typename T>
30  void probe_set(T& s)
31  {
32    std::cout << "\nSet object evaluation" << std::endl;
33    std::cout << "\tObject memory location: " << &s << std::endl;
34    if(!s.empty())
35      std::cout << "\tFirst element memory location: " <<
36        &(*s.begin()) << std::endl;
37    std::cout << "\tIs empty?: ";
38    (s.empty() ? std::cout << "True\n" : std::cout << "False\n");
39    std::cout << "\tSize: " << s.size() << " elements\n" << std::endl;
40  }
41
42  int main(void)
43  {
44    std::cout << "Initializing set object (set1) using default constructor\n";
45    std::set<TestClass> set1;
46    probe_set(set1);
47
48    std::cout << "Initializing set object (set2) using an initializer list
        constructor\n";
49    std::set<TestClass> set2({*std::make_unique<TestClass>(),
50                              *std::make_unique<TestClass>()});
51    probe_set(set2);
52
53    std::cout << "Initializing set object (set3) using a copy constructor\n";
54    std::set<TestClass> set3(set2);
55    probe_set(set3);
56
57    std::cout << "Initializing set object (set4) using a range constructor\n";
58    std::set<TestClass> set4(set2.begin(), set2.end());
59    probe_set(set4);
60
61    std::cout << "Initializing set object (set5) using a move constructor\n";
62    std::set<TestClass> set5(std::move(set4));
63    probe_set(set5);
64  }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos/set$ ./set.x
Initializing set object (set1) using default constructor

Set object evaluation
        Object memory location: 0x7ffdfabd3720
        Is empty?: True
        Size: 0 elements

Initializing set object (set2) using an initializer list constructor
        In TestClass default constructor. (id: 0)
        In TestClass copy constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)

Set object evaluation
        Object memory location: 0x7ffdfabd36f0
        First element memory location:    0x558f9b4fd2e0
        Is empty?: False
        Size: 2 elements

Initializing set object (set3) using a copy constructor
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)

Set object evaluation
        Object memory location: 0x7ffdfabd36c0
        First element memory location:    0x558f9b4fd340
        Is empty?: False
        Size: 2 elements

Initializing set object (set4) using a range constructor
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)

Set object evaluation
        Object memory location: 0x7ffdfabd3690
        First element memory location:    0x558f9b4fd3a0
        Is empty?: False
        Size: 2 elements

Initializing set object (set5) using a move constructor

Set object evaluation
        Object memory location: 0x7ffdfabd3660
        First element memory location:    0x558f9b4fd3a0
        Is empty?: False
        Size: 2 elements
```

1.6 map

A map is an ordered associative container similar to the set container except that it stores elements that are key/value pairs. The map container also provides an overload of the bracket operator, which is the same operator used on array and vector objects, except it does not provide linear asymptotic search run times; but it is a convenient syntax when the need arises to access or modify the value in one of the key/value pairs. A useful feature of the map is that the

key/value pair can remain in the same sequential order if the value is changed because the sorting mechanism depends only on the keys.

Listing 11: map constructor prototypes from C++14 standards

```
/* default */
map();

/* empty constructors */
explicit map (const key_compare& comp,
              const allocator_type& alloc = allocator_type());
explicit map (const allocator_type& alloc);

/* range constructors */
template <class InputIterator>
map (InputIterator first, InputIterator last,
     const key_compare& comp = key_compare(),
     const allocator_type& = allocator_type());
template <class InputIterator>
map (InputIterator first, InputIterator last,
     const allocator_type& = allocator_type());

/* copy constructors */
map (const map& x);
map (const map& x, const allocator_type& alloc);

/* move constructors */
map (map&& x);
map (map&& x, const allocator_type& alloc);

/* initializer-list constructor */
map (initializer_list<value_type> il,
     const key_compare& comp = key_compare(),
     const allocator_type& alloc = allocator_type());
map (initializer_list<value_type> il,
     const allocator_type& alloc = allocator_type());
```

Listing 12: Example using map constructors

```
#include <iostream>
#include <list>
#include <memory>
#include <map>
#include <initializer_list>

class TestClass;
typedef std::allocator<std::tuple<TestClass,int>> allocTuple;

class TestClass
{
  int id_;
public:
  static int id;
  TestClass() : id_(id)
  {
    std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
  }
  TestClass(const TestClass& t) : id_(t.id_)
  {
    std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
  }
```

```
23    bool operator<(const TestClass& a) const
24    {
25      return id_ < a.id_;
26    }
27  };
28
29  int TestClass::id = 0;
30
31  template<typename T>
32  void probe_map(T& m)
33  {
34    std::cout << "\nMap object evaluation" << std::endl;
35    std::cout << "\tObject memory location: " << &m << std::endl;
36    if(!m.empty())
37      std::cout << "\tFirst element memory location: " <<
38        &(*m.begin()) << std::endl;
39    std::cout << "\tIs empty?: ";
40    (m.empty() ? std::cout << "True\n" : std::cout << "False\n");
41    std::cout << "\tSize: " << m.size() << " elements\n" << std::endl;
42  }
43
44  int main(void)
45  {
46    std::cout << "Initializing map object (map1) using default constructor\n";
47    std::map<TestClass, int> map1;
48    probe_map(map1);
49
50    std::cout << "Creating an initializer_list of TestClass and integers\n";
51
52    std::initializer_list<std::pair<const TestClass, int>> initList = {
53      std::make_pair(*std::make_unique<TestClass>(), 0),
54      std::make_pair(*std::make_unique<TestClass>(), 1)
55    };
56    std::cout << std::endl;
57
58    std::cout << "Initializing map object (map2) using an initializer-list
        constructor\n";
59    std::map<TestClass, int> map2{initList};
60    probe_map(map2);
61
62    std::cout << "Initializing map object (map3) using copy+alloc constructor\n";
63    std::map<TestClass, int> map3(std::move(map2),
64                                  *std::make_unique<allocTuple>());
65    probe_map(map3);
66  }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos/map$ ./map.x
Initializing map object (map1) using default constructor

Map object evaluation
        Object memory location: 0x7ffdbc581660
        Is empty?: True
        Size: 0 elements

Creating an initializer_list of TestClass and integers
        In TestClass default constructor. (id: 0)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 1)

Initializing map object (map2) using an initializer-list constructor
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)

Map object evaluation
        Object memory location: 0x7ffdbc581610
        First element memory location: 0x5631bfbe62c0
        Is empty?: False
        Size: 2 elements

Initializing map object (map3) using copy+allocator constructor

Map object evaluation
        Object memory location: 0x7ffdbc5815e0
        First element memory location: 0x5631bfbe62c0
        Is empty?: False
        Size: 2 elements
```

1.7 multiset

A multiset is an ordered associative container that, unlike a set, allows the storage of elements with equivalent values. Otherwise, the multiset and set containers are very similar. The only major difference between the multiset and the set containers is that the multiset likely stores duplicate values in a list container linked to a balanced search tree's node.

Listing 13: multiset constructor prototypes from C++14 standards

```
1  /* default constructor */
2  mutiset();
3
4  /* empty constructors */
5  explicit multiset (const key_compare& comp,
6                     const allocator_type& alloc = allocator_type());
7  explicit multiset (const allocator_type& alloc);
8
9  /* range constructors */
10 template <class InputIterator>
11 multiset (InputIterator first, InputIterator last,
12           const key_compare& comp = key_compare(),
13           const allocator_type& = allocator_type());
14 template <class InputIterator>
15 multiset (InputIterator first, InputIterator last,
16           const allocator_type& = allocator_type());
```

```
17
18 /* copy constructors */
19 multiset (const multiset& x);
20 multiset (const multiset& x, const allocator_type& alloc);
21
22 /* move constructors */
23 multiset (multiset&& x);
24 multiset (multiset&& x, const allocator_type& alloc);
25
26 /* initializer-list constructors */
27 multiset (initializer_list<value_type> il,
28           const key_compare& comp = key_compare(),
29           const allocator_type& alloc = allocator_type());
30 multiset (initializer_list<value_type> il,
31           const allocator_type& alloc = allocator_type());
```

Listing 14: Example using multiset constructors

```
1  #include <iostream>
2  #include <memory>
3  #include <set>
4  #include <initializer_list>
5
6  class TestClass;
7
8  class TestClass
9  {
10   int id_;
11 public:
12   static int id;
13   TestClass() : id_(id)
14   {
15     std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
16   }
17   TestClass(const TestClass& t) : id_(t.id_)
18   {
19     std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
20   }
21   bool operator<(const TestClass& a) const
22   {
23     return id_ < a.id_;
24   }
25 };
26
27 int TestClass::id = 0;
28
29 template<typename T>
30 void probe_multiset(T& m)
31 {
32   std::cout << "\nMultiset object evaluation" << std::endl;
33   std::cout << "\tObject memory location: " << &m << std::endl;
34   if(!m.empty())
35     std::cout << "\tFirst element memory location: " <<
36       &(*m.begin()) << std::endl;
37   std::cout << "\tIs empty?: ";
38   (m.empty() ? std::cout << "True\n" : std::cout << "False\n");
39   std::cout << "\tSize: " << m.size() << " elements\n" << std::endl;
40 }
41
42 int main(void)
43 {
```

```
44    std::cout <<"Initializing multiset object(mset1) using default constructor\n";
45    std::multiset<TestClass> mset1;
46    probe_multiset(mset1);
47
48    std::cout << "Declaring 3 TestClass objects\n";
49    TestClass tc1, tc2, tc3;
50    std::cout << std::endl;
51
52    std::cout << "Initializing multiset object (mset2) using list+comparator\n";
53    auto t = [](const TestClass& a, const TestClass& b){return a<b;}; //redundant
54    std::multiset<TestClass, decltype(t)> mset2({tc1, tc2, tc3}, t);
55    probe_multiset(mset2);
56
57    std::cout << "Initializing multiset object (mset3) using copy constructor\n";
58    std::multiset<TestClass, decltype(t)> mset3(mset2);
59    probe_multiset(mset3);
60 }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos/multiset$ ./multiset.x
Initializing multiset object (mset1) using default constructor

Multiset object evaluation
        Object memory location: 0x7ffebed152b0
        Is empty?: True
        Size: 0 elements

Declaring 3 TestClass objects
        In TestClass default constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass default constructor. (id: 2)

Initializing multiset object (mset2) using init list+comparator
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)

Multiset object evaluation
        Object memory location: 0x7ffebed15270
        First element memory location: 0x558b69a6e2a0
        Is empty?: False
        Size: 3 elements

Initializing multiset object (mset3) using copy constructor
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)
        In TestClass copy constructor. (id: 0)

Multiset object evaluation
        Object memory location: 0x7ffebed15240
        First element memory location: 0x558b69a6e390
        Is empty?: False
        Size: 3 elements
```

1.8 multimap

The multimap is an ordered associative container very similar to the map container. The multimap container can store key/value pairs with duplicate keys. Key/value pairs with duplicate keys are likely stored in a list container linked to one of the nodes in the balanced search tree because the multimap member function *find* returns an iterator for some container (probably a list) containing all key/value pairs with the same key.

Listing 15: multimap constructor prototypes from C++14 standards

```
/* default constructor */
multimap();

/* empty constructors */
explicit multimap (const key_compare& comp,
                   const allocator_type& alloc = allocator_type());
explicit multimap (const allocator_type& alloc);

/* range constructors */
template <class InputIterator>
multimap (InputIterator first, InputIterator last,
          const key_compare& comp = key_compare(),
          const allocator_type& = allocator_type());
template <class InputIterator>
multimap (InputIterator first, InputIterator last,
          const allocator_type& = allocator_type());

/* copy constructors */
multimap (const multimap& x);
multimap (const multimap& x, const allocator_type& alloc);

/* move constructors */
multimap (multimap&& x);
multimap (multimap&& x, const allocator_type& alloc);

/* initializer-list constructors */
multimap (initializer_list<value_type> il,
          const key_compare& comp = key_compare(),
          const allocator_type& alloc = allocator_type());
multimap (initializer_list<value_type> il,
          const allocator_type& alloc = allocator_type());
```

```
#include <iostream>
#include <memory>
#include <map>
#include <initializer_list>

class TestClass;

class TestClass
{
  int id_;
public:
  static int id;
  TestClass() : id_(id)
  {
    std::cout << "\tIn TestClass default constructor. (id: " << id++ << ")\n";
  }
  TestClass(const TestClass& t) : id_(t.id_)
  {
```

```
19      std::cout << "\tIn TestClass copy constructor. (id: " << id_ << ")\n";
20    }
21    bool operator<(const TestClass& a) const
22    {
23      return id_ < a.id_;
24    }
25  };
26
27  int TestClass::id = 0;
28
29  template<typename T>
30  void probe_multimap(T& m)
31  {
32    std::cout << "\nMultimap object evaluation" << std::endl;
33    std::cout << "\tObject memory location: " << &m << std::endl;
34    if(!m.empty())
35      std::cout << "\tFirst element memory location: " <<
36        &(*m.begin()) << std::endl;
37    std::cout << "\tIs empty?: ";
38    (m.empty() ? std::cout << "True\n" : std::cout << "False\n");
39    std::cout << "\tSize: " << m.size() << " elements\n" << std::endl;
40  }
41
42  int main(void)
43  {
44    std::cout<<"Initializing multimap object (mmap1) using default constructor\n";
45    std::multimap<TestClass, int> mmap1;
46    probe_multimap(mmap1);
47
48    std::cout << "Declaring 3 TestClass objects\n";
49    TestClass tc1, tc2, tc3;
50    std::cout << std::endl;
51
52    std::cout<<"Initializing multimap object(mmap2) using init list+comparator\n";
53    auto t = [](const TestClass& a, const TestClass& b){return a < b;}; //redundant
54    std::multimap<TestClass, int, decltype(t)> mmap2({{tc1,1},
55                                                      {tc2,2},
56                                                      {tc3,3}}, t);
57    probe_multimap(mmap2);
58
59    std::cout << "Initializing multimap object (mmap3) using copy constructor\n";
60    std::multimap<TestClass, int, decltype(t)> mmap3(mmap2);
61    probe_multimap(mmap3);
62  }
```

```
debian:~/Documents/EEL3370/homeworks/container_demos/multimap$ ./multimap.x
Initializing multimap object (mmap1) using default constructor

Multimap object evaluation
        Object memory location: 0x7ffefcbef950
        Is empty?: True
        Size: 0 elements

Declaring 3 TestClass objects
        In TestClass default constructor. (id: 0)
        In TestClass default constructor. (id: 1)
        In TestClass default constructor. (id: 2)

Initializing multimap object (mmap2) using init list+comparator
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)
        In TestClass copy constructor. (id: 0)
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)

Multimap object evaluation
        Object memory location: 0x7ffefcbef910
        First element memory location: 0x559c3c3662a0
        Is empty?: False
        Size: 3 elements

Initializing multimap object (mmap3) using copy constructor
        In TestClass copy constructor. (id: 1)
        In TestClass copy constructor. (id: 2)
        In TestClass copy constructor. (id: 0)

Multimap object evaluation
        Object memory location: 0x7ffefcbef8e0
        First element memory location: 0x559c3c366390
        Is empty?: False
        Size: 3 elements
```

2 Algorithms

The STL has an algorithms class that contains a set of algorithms that can be performed on almost all STL containers plus C-style arrays. This section

2.1 find

The find function's algorithm performs a linear search in a container and stops when it encounters the desired value in a container element. It is implemented as a function template, and it may work with any container that can provide iterators and stores objects that have an overloaded equivalence operator. The find function returns an iterator pointing to either the container's element that contains the desired value or, if the value is not found, an empty element that is beyond the last value-containing element of the container.

Listing 16: demonstration of the find algorithm

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4
```

```
5  class TestClass
6  {
7    int id_;
8  public:
9    TestClass(int id) : id_(id) {}
10   bool operator==(const TestClass& t) const
11   {
12     std::cout << "In bool TestClass::operator==" << std::endl;
13     return id_ == t.id_;
14   }
15 };
16
17 int main(void)
18 {
19   TestClass t1(1), t2(2), t3(3);
20   std::vector<TestClass> v{t1, t2, t3};
21   std::cout << "Searching for 2" << std::endl;
22   if(std::find(v.begin(), v.end(), 2) != v.end())
23   {
24     std::cout << "Found 2" << std::endl;
25   }
26 }
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/find$ ./find.x
Searching for 2
In bool TestClass::operator==
In bool TestClass::operator==
Found 2
```

2.2 search

The search function algorithm is implemented using iterators, and, unlike the find function, the goal of the algorithm is to find a specific subsequence of objects inside a container. If the subsequence cannot be found in the container, the function will return an iterator pointing to the last empty element of the container, otherwise the function returns an iterator pointing to the first element in the container that matches the first element of the given subsequence. The search function uses the equivalence operator overloaded by the container(s) or a custom predicate function used to determine equivalence.

Listing 17: demonstration of the search algorithm

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4
5  template<class T, class G=const std::string>
6  void confirmation(T a, G& b, G& c)
7  {
8    if(a != b.end())
9    {
10     std::cout << c << " found in " << b << std::endl;
11   }
12   else
13   {
14     std::cout << c << " not found in " << b << std::endl;
15   }
16 }
17
18 int main(void)
```

```
{
  std::string s1 = "cde";
  std::string s2 = "acf";
  std::string s3 = "abcdef";
  auto its = std::search(s3.begin(), s3.end(), s1.begin(), s1.end());
  confirmation(its, s3, s1);
  its = std::search(s3.begin(), s3.end(), s2.begin(), s2.end());
  confirmation(its, s3, s2);
}
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/search$ ./search.x
cde found in abcdef
acf not found in abcdef
```

2.3 copy

The copy function is used to copy a range of elements in the same order as they appear from one container to another, regardless if the containers are of the same type or not; but some target containers, such as the vector, need to have sufficient memory allocated before receiving the elements because the copy function uses a container's overloaded increment operator plus the assignment operator which together do not (implicitly) cause all containers to expand in size. After the copy function's algorithm has successfully finished copying elements, the function returns an iterator that points to the element in the target container representing the last value that was copied from the source container. The copy function may not succeed because of a segmentation faults or because the iterator pointing to the first element of the target container is also pointing to the last element.

Listing 18: demonstration of the copy algorithm

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <set>
#include <list>
#include <exception>

template<typename T>
void copy_results(T c)
{
  std::cout << "Target container size: " << c.size() << std::endl;
  std::cout << "Iterating [begin,end] container values: ";
  for(auto i = c.begin(); i != c.end(); ++i)
  {
    std::cout << *i;
  }
  std::cout << std::endl << std::endl;
}

int main(void)
{
  std::string s = "abcde";

  std::vector<char> v(10);
  std::cout << "Copying \"" << s << "\" to vector size " << v.size() << "\n";
  std::copy(s.begin(), s.end(), v.begin());
  copy_results(v);

  std::set<char> set;
```

```
30    std::cout << "Copying \"" << s << "\" to set size " << set.size() << "\n";
31    std::copy(s.begin(), s.end(), inserter(set, set.begin()));
32    copy_results(set);
33
34    std::list<char> l(100);
35    std::cout << "Copying \"" << s << "\" to list size " << l.size() << "\n";
36    std::copy(s.begin(), s.end(), inserter(l,l.begin()));
37    copy_results(l);
38
39    std::list<char> l2(100);
40    std::cout << "Copying \"" << s << "\" to list size " << l2.size() << "\n";
41    std::copy(s.begin(), s.end(), l2.begin());
42    copy_results(l2);
43
44    /* scenario leading to segmentation faults */
45    //std::vector<char> v; // problem: vector has no elements
46    //std::copy(s.begin(), s.end(), v.begin());
47
48    /* scenario where nothing is copied because iterator end() == begin() */
49    //std::list<char> l; // or l(s.size());
50    //std::copy(s.begin(), s.end(), l.begin());
51
52    /* scenario that generates compiler error */
53    //std::set<char> set;
54    //std::copy(s.begin(), s.end(), s.begin());
55 }
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/copy$ ./copy.x
Copying "abcde" to vector size 10
Target container size: 10
Iterating [begin,end] container values: abcde

Copying "abcde" to set size 0
Target container size: 5
Iterating [begin,end] container values: abcde

Copying "abcde" to list size 100
Target container size: 105
Iterating [begin,end] container values: abcde

Copying "abcde" to list size 100
Target container size: 100
Iterating [begin,end] container values: abcde
```

2.4 move

The move function from the STL algorithm header has a similar implementation to the copy function except that it uses the move function defined in the utility header to assign values (as rvalues) from a range of elements in one container to the elements of another container. The use of this function reduces the amount of memory used in a program because the data in the source container is copied to the target container and then deleted from the source. The utility move function leaves built-in types and user-defined types in different *moved-from* states: the value of built-in types is unchanged whereas the element values of container are empty yet valid in the sense that the copy assignment and destructors on the objects still work correctly. (Stroustrup, The C++ Programming Language, p. 519)

Listing 19: demonstration of the move algorithm

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
#include <set>
#include <iterator>
#include <memory>

template<typename T>
void container_probe(T& c)
{
  std::cout << "\tContainer size: " << c.size() << std::endl;
  std::cout << "\tIterating [begin,end] container values: \n";
  for(auto i = c.begin(); i != c.end(); ++i)
  {
    std::cout << "\t\t" << *i << " (" << (void*)&(*i) << ")\n";
  }
  std::cout << std::endl << std::endl;
}

int main(void)
{
  /* Initializing containers that will be moved */

  std::string s(5, 'a');
  std::cout << "Printing addresses of each character in string s\n";
  for(int i = 0; i < s.size(); ++i)
  {
    std::cout << "\t" << s[i] << " : " << (void*)&s[i] << std::endl;
  }
  std::cout << std::endl;

  std::cout << "Printing address of first character in each string in vector v1\
\n";
  std::vector<std::string> v1{"hello", "there", " sir "};
  for(int i = 0; i < v1.size(); ++i)
  {
    std::cout << "\t" << v1[i] << " : " << (void*)&v1[i] << std::endl;
  }
  std::cout << std::endl;

  /* Performing the range moves */

  std::cout << "Moving string from s above to a vector v2 of chars\n";
  std::vector<std::string> v2(s.size());
  std::move(s.begin(), s.end(), v2.begin());
  std::cout << "Inspecting target container\n";
  container_probe(v2);
  std::cout << "Inspecting source container after a range move\n";
  container_probe(s);

  std::cout << "Moving vector v1 from above to a vector v3 of strings\n";
  std::vector<std::string> v3(v1.size());
  std::move(v1.begin(), v1.end(), v3.begin());
  std::cout << "Inspecting target container\n";
  container_probe(v3);
  std::cout << "Inspecting source container after a range move\n";
  container_probe(v1);
}
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/move$ ./move.x
Printing addresses of each character in string s
        a : 0x7ffe7233aae0
        a : 0x7ffe7233aae1
        a : 0x7ffe7233aae2
        a : 0x7ffe7233aae3
        a : 0x7ffe7233aae4

Printing address of first character in each string in vector v1
        hello : 0x5628f726d280
        there : 0x5628f726d2a0
         sir  : 0x5628f726d2c0

Moving string from s above to a vector v2 of chars
Inspecting target container
        Container size: 5
        Iterating [begin,end] container values:
                a (0x5628f726d2f0)
                a (0x5628f726d310)
                a (0x5628f726d330)
                a (0x5628f726d350)
                a (0x5628f726d370)


Inspecting source container after a range move
        Container size: 5
        Iterating [begin,end] container values:
                a (0x7ffe7233aae0)
                a (0x7ffe7233aae1)
                a (0x7ffe7233aae2)
                a (0x7ffe7233aae3)
                a (0x7ffe7233aae4)


Moving vector v1 from above to a vector v3 of strings
Inspecting target container
        Container size: 3
        Iterating [begin,end] container values:
                hello (0x5628f726d3a0)
                there (0x5628f726d3c0)
                 sir  (0x5628f726d3e0)


Inspecting source container after a range move
        Container size: 3
        Iterating [begin,end] container values:
                 (0x5628f726d280)
                 (0x5628f726d2a0)
                 (0x5628f726d2c0)
```

Notice the string still contains the characters.

Notice the string values are no longer in the vector. Vector elements are in a "moved-from" state

2.5 swap

The swap function, defined in the utility header instead of the algorithm header since C++11, is used to exchange the values of two variables using move semantics. The swap function is more efficient than swapping values by typical copy constructors/ assignment operators because the move semantics reduce the use of memory.

Listing 20: demonstration of the swap algorithm

```
#include <iostream>
#include <memory>
#include <utility>
class TestClass
{
  int id_;
public:
  TestClass(int id) : id_(id) {}
  TestClass(TestClass&& t) noexcept : id_(0)
  {
    std::swap(id_, t.id_);
  }

  TestClass& operator=(TestClass&& t) noexcept
  {
    std::cout << "In \"move-assignment\" function" << std::endl;
    std::swap(id_, t.id_);
    return *this;
  }

  int getId() { return id_; }
};

int main(void)
{
  std::cout << "Initializing TestClass object t1\n";
  TestClass t1(1);
  std::cout << "\tt1 id: " << t1.getId() << "\n\n";

  std::cout << "Assigning t1 to TestClass object t2 using move-constructor\n";
  TestClass t2(std::move(t1));
  std::cout << "\tt1 id: " << t1.getId() << "\n";
  std::cout << "\tt2 id: " << t2.getId() << "\n\n";

  std::cout << "Assigning t2 to TestClass object t3 using move-constructor\n";
  TestClass t3 = std::move(t2);
  std::cout << "\tt2 id: " << t2.getId() << "\n";
  std::cout << "\tt3 id: " << t3.getId() << "\n\n";

  std::cout << "Swapping t3 into t1\n";
  std::swap(t1, t3);
  std::cout << "\tt1 id: " << t1.getId() << "\n";
  std::cout << "\tt3 id: " << t3.getId() << std::endl;
}
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/swap$ ./swap.x
Initializing TestClass object t1
        t1 id: 1

Assigning t1 to TestClass object t2 using move-constructor
        t1 id: 0
        t2 id: 1

Assigning t2 to TestClass object t3 using move-constructor
        t2 id: 0
        t3 id: 1

Swapping t3 into t1
In "move-assignment" function
In "move-assignment" function
        t1 id: 1
        t3 id: 0
```

2.6 sort

The sort function is used to sort in ascending order a range of elements determined by two iterators of a container object. For a successful sort, the first iterator passed as the first argument to the sort function should point to an element in its container that is sequentially positioned before the element pointed by the second iterator which is used as the sort function's second argument; otherwise a segmentation fault may occur. The algorithm is very efficient when it comes to memory consumption because it is implemented using the swap function which uses move semantics; and in terms of time complexity, it is

Listing 21: demonstration of the sort algorithm

```
#include <iostream>
#include <algorithm>
#include <vector>

class TestClass
{
  int id_;
public:
  TestClass(int id) : id_(id) {}
  TestClass(const TestClass& t) { id_ = t.id_; }
  TestClass(TestClass&& t) noexcept : id_(0)
  {
    std::swap(id_, t.id_);
  }

  TestClass& operator=(TestClass&& t) noexcept
  {
    //std::cout << "In \"move-assignment\" function" << std::endl;
    std::swap(id_, t.id_);
    return *this;
  }

  bool operator<(const TestClass& t) const
  {
    return this->id_ < t.id_;
  }

  friend std::ostream& operator << (std::ostream& o, const TestClass& t)
  {
```

```
30      o << t.id_;
31      return o;
32    }
33
34    int getId() { return id_; }
35  };
36
37  template <typename C>
38  void print_container_seq(C& c)
39  {
40    for(auto& itr : c)
41    {
42      std::cout << itr << "\n";
43    }
44  }
45
46  int main(void)
47  {
48    TestClass t1(1), t2(2), t3(3), t4(4), t5(5);
49    std::vector<TestClass> v{t3,t1,t5,t2,t4};
50    std::cout << "Printing vector elements in sequential order (pre-sort)\n";
51    print_container_seq(v);
52    std::cout << std::endl;
53
54    std::sort(v.begin(), v.end());
55    std::cout << "Printing vector elements in sequential order (post-sort)\n";
56    print_container_seq(v);
57  }
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/sort$ ./sort.x
Printing vector elements in sequential order (pre-sort)
3
1
5
2
4

Printing vector elements in sequential order (post-sort)
1
2
3
4
5
```

2.7 max

The max function returns the largest of two values passed as the first two arguments of the function or the largest in an std::initializer_list passed as the first argument. A function used to compare the objects can be passed to the max function as a function pointer or function object so the max function can use it to compare the objects and determine the largest one.

Listing 22: demonstration of the max algorithm

```
1  #include <iostream>
2  #include <algorithm>
3
4  class TestClass
5  {
6    int id_;
```

```
public:
  TestClass(int id) : id_(id) {}
  TestClass(const TestClass& t) { id_ = t.id_; }
  TestClass(TestClass&& t) noexcept : id_(0)
  {
    std::swap(id_, t.id_);
  }

  bool operator<(const TestClass& t) const
  {
    return this->id_ < t.id_;
  }

  friend std::ostream& operator << (std::ostream& o, const TestClass& t)
  {
    o << t.id_;
    return o;
  }
};

int main(void)
{
  TestClass t1(1), t2(2), t3(3), t4(4), t5(5);
  auto testClassCompare = [](auto a, auto b){return a < b;};

  TestClass max = std::max(t4, t5);
  std::cout << "Comparing two TestClass objects\n";
  std::cout << "\tObject with largest id: " << max << std::endl;
  std::cout << std::endl;

  TestClass max2 = std::max({t3,t1,t2,t4,t4}, testClassCompare);
  std::cout << "Comparing two TestClass objects\n";
  std::cout << "\tObject with largest id: " << max2 << std::endl;
}
```

```
debian:~/Documents/EEL3370/homeworks/stlalgorithms/max$ ./max.x
Comparing two TestClass objects
        Object with largest id: 5

Comparing two TestClass objects
        Object with largest id: 4
```