Ship's API

All classes and types would be inside the namespace: *shipping*
Entire implementation shall be inside *Ship.h* -- there is NO *Ship.cpp* as this is a template

Special Global Types:

X, Y, Height

each one of the above three types is *constructed explicitly by* ***int*** and has a ***casting to int***

Ship's template parameter:
Ship would be a templated class, with a template parameter: ***typename Container***

type:

```
template<typename Container>
using Grouping = std::unordered_map<string, std::function<string(const Container&)>>;
```

above type defines a groping map, with:

- name of the grouping: std::string, as the *key*
- a grouping function that gets a const Container& and returns its group name as a string, for this grouping function, as the *value*

Groupings can be according to: destination port, container's owner etc.

Ship's Constructor (1):

```
Ship(X x, Y y, Height max_height,
        std::vector<std::tuple<X, Y, Height>> restrictions,
        Grouping<Container> groupingFunctions) noexcept(false);
```

Ship's Constructor (2):

```
Ship(X x, Y y, Height max_height,
        std::vector<std::tuple<X, Y, Height>> restrictions) noexcept(false);
```

This constructor is useful if there are no groupings.

both methods above may throw BadShipOperationException
(see details of this Exception class below).

Ship's Constructor (3):

```
Ship(X x, Y y, Height max_height) noexcept;
```

This constructor is useful if there are no restrictions and no groupings.

So to create a ship, one can do for example:

```
Ship<int> myShip{X{5}, Y{12}, Height{8}};
```

Above creates a ship with containers of type int.

loading a container:

```
void load(X x, Y y, Container c) noexcept(false);
```

the method may throw BadShipOperationException.

unloading a container:

```
Container unload(X x, Y y) noexcept(false);
```

the method may throw BadShipOperationException.

moving a container from one location to another on the ship:

```
void move(X from_x, Y from_y, X to_x, Y to_y) noexcept(false);
```

the method may throw BadShipOperationException.

iterators begin and end:
The ship would only have a const version *begin* and *end* iterators for iterating over all containers on the ship. There is no defined order. Iteration shall not create a copy of the containers but rather run on the original containers on ship.

getContainersView:
The ship would have the following methods to obtain a "view" of the containers.
The return value of those functions is explained below.

- `getContainersViewByPosition(X x, Y y) const;`
- `getContainersViewByGroup(const string& groupingName, const string& groupName) const;`

functions would not throw an exception, but may return an empty view.

- The view functions would return something of your choice which has iterators *begin* and *end* to allow traversal on the view.
- The view would never be a copy of the containers. If the user calls one of these functions and holds the result, then loads, unloads or moves a container, then runs on the view - the run on the view would be on the new data. On the other hand, the view **doesn't have to support** traversing on the view, stopping, then loading, unloading or moving a container, then continuing the traversal - such operation is not defined, i.e. load/unload/move operations may invalidate the iterators of a view.
- After a full cycle over the view you cannot traverse over it again, but you can retrieve the same view again with the proper *getContainersView* function.
- The order for running on the view:
    - *getContainersViewByPosition* - from the **highest** container and **downwards**
    - *getContainersViewByGroup* - order is not important
- **iterator* provided by the view would be:
    - *getContainersViewByPosition* - const Container&
    - *getContainersViewByGroup* - std::pair<tuple {X, Y, Height}, const Container&>

BadShipOperationException
has the following ctor: `BadShipOperationException(string msg);`

- the message is yours, we will not check it, use it as you find suitable

Usage examples follow...

Usage Example 1

```
#include "Ship.h"

using namespace shipping;

   int main() {
                          // create restrictions for specific locations on the
   ship
                          std::vector<std::tuple<X, Y, Height>> restrictions = {
                                std::tuple(X{2}, Y{6}, Height{0}),
                                std::tuple(X{2}, Y{7}, Height{1}),
                                std::tuple(X{2}, Y{5}, Height{6}),
                          };

                          // create bad ship 1
                          try {
                                restrictions.push_back( std::tuple(X{2}, Y{5},
   Height{6}) );
                                Ship<std::string> myShip{ X{4}, Y{12},
   Height{16}, restrictions };
                          } catch(BadShipOperationException& e) {
                                // exception: duplicate restrictions (whether or
   not it has same limit):
                                // restriction with X{2}, Y{5} appears more than
   once (added in the try)
                                restrictions.pop_back(); // remove the duplicate
   restriction
                          }
                          // create bad ship 2
                          try {
                                Ship<std::string> myShip{ X{4}, Y{7}, Height{8},
   restrictions };
                          } catch(BadShipOperationException& e) {
                                // exception due to bad restrictions:
                                // restriction with Y=7, when the size of Y is 7
                          }
                          // create bad ship 3
                          try {
                                Ship<std::string> myShip{ X{4}, Y{12}, Height{6},
   restrictions };
                          } catch(BadShipOperationException& e) {
                                // exception due to bad restrictions:
                                // restriction with height=6, when original
   height is equal or smaller
                          }

                          // create good ship
                          Ship<std::string> myShip{ X{4}, Y{8}, Height{8},
   restrictions };

                          // bad load - no room
                          try {
                                myShip.load(X{2}, Y{6}, "Hello");
                          } catch(BadShipOperationException& e) { /* no room at
   this location */ }

                          // good load
                          myShip.load(X{2}, Y{7}, "Hello");

                          // bad load - no room
                          try {
                                myShip.load(X{2}, Y{7}, "Hello");
                          } catch(BadShipOperationException& e) { /* no room at
   this location */ }

                          // bad unload - no container at location
                          try {
                                 std::string container = myShip.unload(X{1},
   Y{1});
                          } catch(BadShipOperationException& e) { /* no container
   at this location */ }

                          // bad load - wrong index
                          try {
                                 myShip.load(X{1}, Y{8}, "Hi");
                          } catch(BadShipOperationException& e) { /* bad index Y
   {8} */ }
   }
```

Usage Example 2

```
#include "Ship.h"

using namespace shipping;
using std::string;

int main() {
   // create grouping pairs
   Grouping<std::string> groupingFunctions = {
                          { "first_letter",
                                 [](const string& s){ return string(1, s[0]); }
                          },
                          { "first_letter_toupper",
                                 [](const string& s){ return string(1,
   char(std::toupper(s[0]))); }
                          }
   };
   // create restrictions
   std::vector<std::tuple<X, Y, Height>> restrictions = {
                          std::tuple(X{2}, Y{6}, Height{4}),
                                 std::tuple(X{2}, Y{7}, Height{6}),
                          std::tuple(X{0}, Y{0}, Height{2})
   };
   // create ship
   Ship<std::string> myShip{ X{5}, Y{12}, Height{8},

                          restrictions,

                          groupingFunctions };
                          // load “containers”
   myShip.load(X{0}, Y{0}, "Hello");
   myShip.load(X{1}, Y{1}, "hey");
                          myShip.load(X{1}, Y{1}, "bye");

                          auto view00 = myShip.getContainersViewByPosition(X{0},
   Y{0});
                          auto view_h =
   myShip.getContainersViewByGroup("first_letter", "h");
                          auto view_Hh =
   myShip.getContainersViewByGroup("first_letter_toupper", "H");

                          myShip.load(X{0}, Y{0}, "hi");

                          // loop on all “containers”: Hello, hi, hey, bye - in
   some undefined order
                          for(const auto& container : myShip) { /*...*/}
```

```
                           // loop on view00: hi, Hello - in this exact order
    for(const auto& container : view00) { /*...*/}

                           // loop on view_h:  pair { tuple{X{0}, Y{0}, Height{1}},
    hi },
                           //                                      pair {
    tuple{X{1}, Y{1}, Height{0}}, hey }
                           // - in some undefined order
                           for(const auto& container_tuple : view_h) { /*...*/}

                           // loop on view_Hh: pair { tuple{X{0}, Y{0}, Height{0}},
    Hello },
                           //                                      pair {
    tuple{X{0}, Y{0}, Height{1}}, hi },
                           //                                      pair {
    tuple{X{1}, Y{1}, Height{0}}, hey }
                           // - in some undefined order
                           for(const auto& container_tuple : view_Hh) { /*...*/}
}
```

Good Luck!