STL Library

Before you read about the STL library, it is important to understand the Templates, so ensure that you have covered that section before you start here.

Various Containers

There are different types of containers available in the STL library:

1. Sequence containers

array

An STL array is a fixed-size array, much like `int array[5]`. The difference is that the STL array is more of a "first class citizen" in terms of providing APIs on this array which are iterators, and other accessors such as `size()`, `front()` and `back()`.

```
#include <iostream>
#include <array>
int main() {
    // Create and initialize an std::array with 5 integers
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};
    // Accessing elements using operator[]
    std::cout << "The first element is: " << numbers[0] << std::endl;
    // Accessing elements using at() with bounds checking
    try {
        std::cout << "Element at index 4 is: " << numbers.at(4) << std::endl;
        // This will throw an exception if the index is out of bounds
        std::cout << "Element at index 5 is: " << numbers.at(5) << std::endl;
    } catch (std::out_of_range& e) {
```

```
        std::cerr << "Error: " << e.what() << std::endl;
    }
    // Iterating over elements using a range-based for loop
    std::cout << "All elements: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;}
```

Further examples

```
#include <iostream>
#include <array>
#include <algorithm>
void c_array() {
  // Standard C int array doesn't offer any APIs
  int int_array[] = {1, 2, 3, 4, 5};
  int_array[5] = 123; /* Uncaught memory exception */
}
void cpp_array__access_api() {
  // Better thing to do this:
  std::array<int, 5> cpp_int_array{1, 2, 3, 4, 5};
  // std::array offers some useful APIs
  cpp_int_array.at(0);
  cpp_int_array[1] = 22;
  //cpp_int_array[5] = 123; /* [] operator will not catch memory exception */
  //cpp_int_array.at(5);    /* C++ library catches memory exception */
  cpp_int_array.front() = 11; // Same as index[0] or .at(0)
  //cpp_int_array.end() = 67890;   // End is actually the first element out of bound
}
void cpp_array__iterate_api() {
  std::array<int, 5> cpp_int_array{1, 2, 3, 4, 5};
  // Let us iterate through each element of the array:
  for (int element_iterator : cpp_int_array) {
    std::cout << "Element value: " << element_iterator << std::endl;
```

```
  }
  // The loop above is the same as the following:
  for (auto it = cpp_int_array.begin();
            it < cpp_int_array.end(); /* end() is the first element address out of bound */
            it++) {                   /* end() is thus an element address following the last element o
    std::cout << "Element value using iterator: " << *it << std::endl;
  }
}
void cpp_array__other_apis() {
  std::array<int, 5> cpp_int_array;
  cpp_int_array.fill(123);
  for (int element_iterator : cpp_int_array) {
    std::cout << "Element value: " << element_iterator << std::endl;
  }
}
void cpp_array__powerful_things_about_containers_with_iterators() {
  std::array<int, 5> arr {10, 50, 4, -5, -100};
  std::sort(arr.begin(), arr.end());
  for (int element_iterator : arr) {
    std::cout << "Element value: " << element_iterator << std::endl;
  }}
```

Inconvenience of std::array

The usage of std::array becomes inconvenient when we have to pass it as parameters to a function because not only your code has to account for the type in the template, but also the size. The problem is that if and when we change the size of the array, then we have to change it in a number of places.

```
#include <iostream>
#include <array>
// Function that modifies the array elements
void modify_array(std::array<int, 5>& arr) {
    for (int& num : arr) {
        num *= 2;  // Double each element
    }
}
// Function that prints the array elements (passed by const reference to prevent modification)
```

```
void print_array(const std::array<int, 5>& arr) {
    std::cout << "Array elements: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
int main() {
    // Create and initialize an std::array with 5 integers
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};
    // Print the original array
    print_array(numbers);
    // Modify the array elements
    modify_array(numbers);
    return 0;}
```

We can improve the code above utilizing the "using" keyword, but functions still get bound to an array of a fixed size, and functions are not reusable for other types of arrays.

```
#include <iostream>
#include <array>
// Array size is fixed, and we could use this statement to improve our code
using fixed_size_array = std::array<int, 3>;
// Function that modifies the array elements
void modify_array(fixed_size_array& arr) {
    for (int& num : arr) {
        num *= 2;  // Double each element
    }
}
// Function that prints the array elements (passed by const reference to prevent modification)
void print_array(const fixed_size_array& arr) {
    std::cout << "Array elements: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
```

```
}
void yet_another_problem_with_aliased_type() {
  std::array<int, 10> another_array = { 1, 2, 3, 4, 5, };
  /* This won't work because print_array() is bound to array of 3 integers only */
  print_array(another_array);
}
int main() {
    // Create and initialize an std::array with 5 integers
    fixed_size_array numbers = {1, 2, 3};
    // Print the original array
    print_array(numbers);
    // Modify the array elements
    modify_array(numbers);
    return 0;
}
```

vector

A vector is a more flexible container than an `std::array` because it can dynamically grow (or shrink). Let us practice a code snippet that uses various APIs that this class provides.

A vector is simply an array that can grow as needed. Try out the following code:

```
#include <iostream>
#include <vector>
int main() {
  std::vector<int> v;
  v.reserve(10);
  for (int i = 0; i < 100; i++) {
    std::vector<int>::iterator before = v.begin();
    v.push_back(i);
    auto after = v.begin();
    if (before != after) {
      printf("Memory got re-allocated when we tried to insert element #%d\n", (i+1));
    }
  }
  return 0;
```

```
}
```

Here are more examples of a vector with more APIs that it offers:

```
#include <iostream>
#include <vector>
int main() {
    // Create a vector of integers
    std::vector<int> vec;
    // Check if the vector is initially empty
    std::cout << "Initially, is vector empty? " << (vec.empty() ? "Yes" : "No") << std::endl;
    // Add elements to the vector
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    // Size and capacity after adding elements
    std::cout << "Size after adding 3 elements: " << vec.size() << std::endl;
    std::cout << "Capacity after adding 3 elements: " << vec.capacity() << std::endl;
    // Print current elements in the vector
    std::cout << "Current elements in vector: ";
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
    return 0;}
```

A vector is definitely more powerful than std::array, and unlike the std::array, you can pass the vector to functions more conveniently. Let us modify our earlier example to a vector and see the difference.

```
#include <iostream>
#include <vector>
// Function that modifies the array elements
void modify_array(std::vector<int>& arr) {
    for (int& num : arr) {
        num *= 2;  // Double each element
    }
```

```
}
// Function that prints the array elements (passed by const reference to prevent modification)
void print_array(const std::vector<int>& arr) {
    std::cout << "Array elements: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
int main() {
    // Create and initialize an std::array with 5 integers
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    // Print the original array
    print_array(numbers);
    // Modify the array elements
    modify_array(numbers);
    return 0;}
```

deque

Double **E**nded **Que**ue is pronounced as "deck" (ue is missing in deque), and it works similar to a vector, except that we can insert data at the beginning or at the end meaning that there is a `push_front()` as well as `push_back()`. One key difference is that a <vector> maintains contiguous memory access to all its elements. If we have a pointer to the first element, we can offset the pointer by N to get to the Nth element. In other words, we have O(1) access to any of the element of the vector. Dequeue also has O(1) access to any of its elements through a complex [] operator function, however, its data is in series of chunks or blocks of contiguous memory, **but not one large contiguous memory** like a vector.

```
int main() {
  std::vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  int *pointer_to_first_element = v.data(); // data returns memory reference/pointer to first memory o
  pointer_to_first_element[0] = 11;
  pointer_to_first_element[1] = 22;
  pointer_to_first_element[2] = 33;
```

```
for(size_t i = 0; i < v.size(); i++) {
    std::cout << "[" << i << "] = " << v[i] << std::endl;
}
return 0;}
```

The other key difference is insertion operation. When a vector runs out of memory, new memory is allocated and data is copied to the new memory. However, a dequeue can not only grow in both directions (front and back), but when the container runs out of memory, it can grow the memory without incurring the full cost of memory allocation or copying data from old memory to new memory.

The following APIs are in addition to what a vector would provide:

- emplace_front()
- push_front()
- pop_front()

However, the following APIs are lacking because they are not needed based on the implementation of the dequeue:

- capacity()
- reserve()

forward_list

Forward list is a singly linked list; it has "forward" links to its elements and only forward links and not backward links of a <list> container. It's operation is actually very similar to the <list>, however, because it is a singly linked-list, the "end()" iterator is lacking because one has to iterate through the beginning of the list to reach the end of the list. Hence, you will find these APIs missing from this container:

- back()
- emplace_back()
- push_back()
- pop_back()

list

List is a doubly linked list. API wise, we have all the functionality of a vector, however because it is a linked-list, the following APIs are lacking:

- operator[]
- at()

2. Associative containers

set and multiset

`set` is a container that is meant to hold sorted and unique elements. For example, an array of unique integers.

It's job is to be able to hold your objects in a sorted manner.

`multiset` is very similar to a set with the exception that you can have duplicate members, such as duplicate integers `{1, 1, 2, 2, 3, 4}`.

```
#include <iostream>
#include <set>
int main() {
    // Create a set of integers
    std::set<int> numbers;
    // Insert elements
    numbers.insert(10);
    numbers.insert(40);
    numbers.insert(30);
    numbers.insert(20);
    numbers.insert(10); // This will not be added again, as sets do not allow duplicates
    // Print the elements of the set
    std::cout << "Elements of the set: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";
    return 0;}
```

Practical Example

```
#include <iostream>
#include <set>
#include <string>
class Book {
public:
    std::string title;
    std::string author;
    Book(const std::string& title, const std::string& author)
        : title(title), author(author), year(year) {}
    // Define the operator '<' that will be used by the 'std::set' to
    // insert elements in a sorted order
```

```
    bool operator<(const Book& other) const {
        return title < other.title;
    }
};
int main() {
    std::set<Book> library;
    library.insert(Book("1984", "George Orwell"));
    library.insert(Book("The Great Gatsby", "F. Scott Fitzgerald"));

    // A duplicate entry will not be inserted:
    library.insert(Book("1984", "George Orwell"));
    std::cout << "Books in the library:\n";
    for (const auto& book : library) {
        std::cout << book.title << " by " << book.author << "\n";
    }
    return 0;}
```

map and multimap

map is a container that holds data mapped to a key. For example, a map of integers (key) mapped to a string (value). Hence, it is essentially a key/value map. The keys must be unique that map to their corresponding values.

```
#include <iostream>
#include <map>
#include <string>
class Book {
public:
    std::string title;
    std::string author;
    Book(const std::string& title, const std::string& author)
        : title(title), author(author) {}
    // Unlike the previous example with std::set, we do not need < operator
    // because the map is sorted by the "key" which in this case is the book title (std::string)
};
int main() {
    std::map<std::string, Book> library;
```

```
    // Insert books into the library; the key is the book title, and the value is the Book instance
    library.insert(std::make_pair("1984", Book("1984", "George Orwell")));
    library.insert(std::make_pair("The Great Gatsby", Book("The Great Gatsby", "F. Scott Fitzgerald"))

    // Here are other ways to insert a "pair" (although since it is a map, duplicate entries won't be i
    library.insert({"1984", Book("1984", "George Orwell")});
    library.insert(std::make_pair("1984", Book("1984", "George Orwell")));
    library.insert(std::pair<std::string, Book>("1984", Book("1984", "George Orwell")));
    std::cout << "Books in the library:\n";
    for (const auto& entry : library) {
        const auto& book = entry.second;
        std::cout << book.title << " by " << book.author << "\n";
    }
    return 0;
}
```

Exercise

Here is the code structure of a "Bookstore" class. Note that the inventory of books is a private member, and we are exposing public APIs to be able to manage the Bookstore class.

```
#include <iostream>
#include <map>
#include <string>
class Bookstore {
private:
    std::multimap<std::string, int> inventory;
public:
    // Add or update books in the inventory
    void add_or_update(const std::string& title, int quantity) {
        inventory[title] += quantity;  // Adds quantity to existing or initializes with quantity if no
    }
    // Remove a book from inventory
    bool remove_book(const std::string& title) {
        auto it = inventory.find(title);
        if (it != inventory.end()) {
            inventory.erase(it);
```

```
            return true;
        }
        return false;
    }
    // Check the stock of a specific book
    bool is_book_present(const std::string& title) const {
        auto it = inventory.find(title);
        return (it != inventory.end());
    }
    // Print the current inventory
    void print_inventory() const {
        std::cout << "Current Inventory:\n";
        for (const auto& pair : inventory) {
            std::cout << "Title: " << pair.first << ", Quantity: " << pair.second << "\n";
        }
    }
};
```

For this exercise, please build a menu system to complete the following code:

```
int main() {
    Bookstore store;
    int choice;

    do {
        std::cout << "\nBookstore Inventory Management:\n";
        std::cout << "1. Add/Update Book\n";
        std::cout << "2. Remove Book\n";
        std::cout << "3. Print Inventory\n";
        std::cout << "4. Check Stock\n";
        std::cout << "5. Exit\n";
        std::cout << "Enter choice: ";
        std::cin >> choice;
        switch (choice) {
            case 1:
                // Implement adding/updating a book
```

```
                break;
            case 2:
                // Implement removing a book
                break;
            case 3:
                store.print_inventory();
                break;
            case 4:
                // Implement checking stock
                break;
            case 5:
                std::cout << "Exiting...\n";
                break;
            default:
                std::cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != 5);
    return 0;}
```

3. Unordered associative containers

The unordered associative containers are very similar to ordered ones; here are the differences:

- Ordered containers store data in sorted order
- Unordered containers store data using hash memory

Pseudo implementation of hash

```
 #include <iostream>
#include <memory>
#include <string>
template <typename type>
class unordered_set {
  const size_t bucket_size = 100;
  std::unique_ptr<type> elements[100];
public:
  void insert(const type& key) {
```

```
    const auto hashed_value = std::hash<type>{}(key);
    const auto mapped_index = hashed_value % bucket_size;
    std::cout << "Hashed value: " << hashed_value << std::endl;
    std::cout << "Mapped index: " << mapped_index << std::endl;
    // This mapped index is empty, hence add the element here
    if (elements[mapped_index] == nullptr) {
      std::cout << "Adding a new element to the set: " << key << std::endl;
      elements[mapped_index] = std::make_unique<type>(key);
    }
    // There is an existing key mapped already
    else {
      std::cout << "Mapped index (" << key << ") already exists" << std::endl;
      // Is there an existing key already?
      if (*elements[mapped_index] == key) {
        std::cout << "Duplicate element, will not add it" << std::endl;
      }
      else {
        std::cout << *elements[mapped_index] << " and "
                  << key <<
                  " share the same mapped memory" << std::endl;

        // TODO: Handle this case
      }
    }
  }
};
void line() { std::cout << "------------" << std::endl; }
void test_string() {
  unordered_set<std::string> set;
  set.insert("hello"); line();
  set.insert("world"); line();
  set.insert("world"); line();
}
void test_int() {
  unordered_set<int> set;
  set.insert(123); line();
```

```
  set.insert(456); line();
  set.insert(456); line();
  set.insert(223); line();
}
int main(int argc, char **argv) {
  std::cout << "Hello world!" << std::endl;
  test_string();
  test_int();
  return 0;
}
```

The code above doesn't handle the case when we have "collisions". The collisions occur when there aren't enough buckets or containers or when multiple items by chance happen to map to the same memory. In this case, we have to change our unique pointer of elements to a vector of elements.

```
#include <iostream>
#include <vector>
#include <algorithm>
template <typename type>
class unordered_set {
  const size_t bucket_size = 100;
  // When "collisions" occur, we need to be able to store more than one element at the "mapped memory"
  // Therefore, let us change the single element storage to multiple element storage
  // std::unique_ptr<type> elements[100];
  std::vector<type> elements[100];
public:
  void insert(const type& key) {
    const auto hashed_value = std::hash<type>{}(key);
    const auto mapped_index = hashed_value % bucket_size;
    std::cout << "Hashed value: " << hashed_value << std::endl;
    std::cout << "Mapped index: " << mapped_index << std::endl;
    // This mapped index is empty, hence add the element here
    auto& mapped_container = elements[mapped_index];
    if (mapped_container.size() == 0) {
      std::cout << "Adding a new element to the set: " << key << std::endl;
      mapped_container.push_back(key);
    }
```

```
    // There is an existing key mapped already
    else {
      std::cout << "Mapped entry (" << key << ") already exists" << std::endl;
      // Is there an existing key already?
      const auto& found_entry = std::find(mapped_container.begin(), mapped_container.end(), key);
      if (found_entry != mapped_container.end() ) {
        std::cout << "Duplicate entry, will not add it" << std::endl;
      }
      else {
        std::cout << key <<  "has to share the same mapped memory" << std::endl;
        mapped_container.push_back(key);
      }
    }
  }
};
void line() { std::cout << "------------" << std::endl; }
void test_string() {
  unordered_set<std::string> set;
  set.insert("hello"); line();
  set.insert("world"); line();
  set.insert("world"); line();
}
void test_int() {
  unordered_set<int> set;
  set.insert(123); line();
  set.insert(456); line();
  set.insert(456); line();
  set.insert(223); line();
}
int main(int argc, char **argv) {
  test_string();
  test_int();
  return 0;
}
```

Now that you have a deeper understanding of hashing and memory storage (STL calls it "buckets"), please try the following program and make sense out of the output.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_set>
void test_int() {
  std::unordered_multiset<int> set;
  set.insert(123);
  set.insert(456);
  set.insert(456); // duplicate
  for(int i = 0; i < 30; i++) {
    set.insert(i);
  }
  std::cout << "bucket count      : " << set.bucket_count() << std::endl;
  std::cout << "max_bucket_count  : " << set.max_bucket_count() << std::endl;
  std::cout << "bucket_size of 123: " << set.bucket_size(123) << std::endl;
  std::cout << "bucket_size of 456: " << set.bucket_size(123) << std::endl;
  std::cout << "bucket of 123     : " << set.bucket(123) << std::endl;
  std::cout << "bucket of 456     : " << set.bucket(456) << std::endl;
  std::cout << "load_factor       : " << set.load_factor() << std::endl;
  std::cout << "max_load_factor   : " << set.max_load_factor() << std::endl;
}
int main(int argc, char **argv) {
  test_int();
  return 0;}
```

Exercise

Please extend the unordered_set class by adding a new method that would print all elements in the set:

- `void unordered_set::print_all_elements()`

Note that some containers may be empty vectors, and you would have to go through each container and print the data if the container is not empty.

unordered_set and unordered_multiset

Similar to `std::set` and `std::multiset`, these containers provide the ability to store unique elements for `std::unordered_set` and possibly duplicate elements also using the `std::unordered_multiset`. In fact, the same example from our previous section would work if we merely change the container type.

One key difference would be that when the `std::set` is iterated for each element, it would print the elements in a sorted order. That is because fundamentally the data is stored in a sorted data structure inside. The "unordered" as its name implies stores data in a bit arbitrary way and elements are not sorted. Therefore, the data is iterated in a random order.

```
#include <iostream>
//#include <set>
#include <unordered_set>

int main() {
    // Create a set of integers
    //std::set<int> numbers;
    std::unordered_set<int> numbers;
    // Insert elements
    numbers.insert(10);
    numbers.insert(40);
    numbers.insert(30);
    numbers.insert(20);
    numbers.insert(10); // This will not be added again, as sets do not allow duplicates
    // Print the elements of the set
    std::cout << "Elements of the set: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";
    return 0;}
```

unordered_map and unordered_multimap

The "map" version of the unordered container is obviously a map, and not just a collection of keys. For the `std::unordered_map`, there is a unique key and only one mapped value to the key. The `multimap` version offers the ability to store multiple values mapped to a single key.

Let us modify our code to enable it for key/value type.

```
```

```
#include <iostream>
#include <vector>
#include <algorithm>
template <typename KeyType, typename ValueType>
class unordered_map {
  const size_t bucket_size = 100;
  // Use a vector of vector of pairs to handle collisions and store key-value pairs.
  std::vector<std::pair<KeyType, ValueType>> elements[100];
public:
  void insert(const KeyType& key, const ValueType& value) {
    const auto hashed_value = std::hash<KeyType>{}(key);
    const auto mapped_index = hashed_value % bucket_size;
    std::cout << "Hashed value: " << hashed_value << std::endl;
    std::cout << "Mapped index: " << mapped_index << std::endl;
    auto& mapped_container = elements[mapped_index];
    bool key_exists = false;
    // The key maps to this container, but multiple keys could be stored here due to collisions
    // Hence, search the container to check if our key exists
    for (auto& elem : mapped_container) {
      if (elem.first == key) {
        // Key exists, update value
        std::cout << "Key (" << key << ") exists, updating value to " << value << std::endl;
        elem.second = value;
        key_exists = true;
        break;
      }
    }
    if (!key_exists) {
      // Key does not exist, add new key-value pair
      std::cout << "Adding new key-value pair to the map: " << key << ": " << value << std::endl;
      mapped_container.emplace_back(key, value);
    }
  }
};
void line() { std::cout << "------------" << std::endl; }
void test_string() {
  unordered_map<std::string, int> map;
```

```
  map.insert("hello", 1); line();
  map.insert("world", 2); line();
  map.insert("world", 3); line();
}
void test_int() {
  unordered_map<int, std::string> map;
  map.insert(123, "abc"); line();
  map.insert(456, "def"); line();
  map.insert(456, "ghi"); line();
  map.insert(223, "xyz"); line();
}
int main(int argc, char **argv) {
  test_string();
  test_int();
  return 0;
}
```

Revision #21
Created 2 years ago by Preet Kang
Updated 1 year ago by Preet Kang