STL in C++

Name: Dazhi Li
NetID: dl939

1. Container

In my understanding, container is a kind of data carrier, which means they store different kind of data. But containers are divided into many kinds, they use template to fit into different type of data. Using container will help programmer finish some complete task with useful tools. For example, if we want to develop a double ended queue in C++, we can import deque library to create a deque in one step. Deque container will store those data we want the program to keep. Also, container will support some member function for users to manipulate their data or container structure. STL support seven basic kind of containers in two groups. One group is sequence containers, which include vector, deque and list. The other group is associative containers, which include set, multiset, map and multimap. Different type of containers will support various service for C++ programmer according to their own advantages and disadvantages.

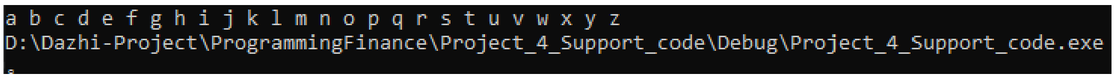
Here, let us use list as an example to show how container works in C++.

Code example:

```
    #include <iostream>
    #include <list>//Import STL container list
    //List is actually doubly linked list

    using namespace std;

    int main(void)
    {
        list<char> my_List;//Instantiate a list like using class templete
        //my_List should contains char type data

        for (char c = 'a'; c <= 'z'; ++c)
            my_List.push_back(c);//insert variable c at the end of my_List
        //By using a loop to create a list contains a to z alphabet

        while (!my_List.empty())//.empty() function will return back bool value
        //empty() will return true if my_List is empty
        {
            cout << my_List.front() << " ";//Print our the first value in my_List
            my_List.pop_front();//pop out the first value from my_List
        }
        //At the end of the loop, all the values will be pop out and left
        // an empty list

        return 0;
}
```

This code example shows us how to create a list container and store the alphabet in it. Without the iterator, C++ programmer is still able to visit the value inside it and print it out. As I have mentioned above that, there are multi containers for user to choose. If user want to learn how to use deque or other containers, it is easy for user to find reference on Cplusplus.com.

Running result:



```
a b c d e f g h i j k l m n o p q r s t u v w x y z
D:\Dazhi-Project\ProgrammingFinance\Project_4_Support_code\Debug\Project_4_Support_code.exe
```

Fig 1 List Code Example Result

2. Iterator

As we have containers in STL now, but how we can visit each element in the container becomes a question. Smart C++ programmers utilize iterator to conveniently visit each element in the container. Usually there are four kinds of container. The first is normal iterator, the second one is const iterator, the third one is reverse iterator and the last one is the const reverse iterator. Iterator will help C++ programmer manipulate their container or read the data from container. But different kind of container's iterator has different functions. And some container does not have iterator like stack or queue.

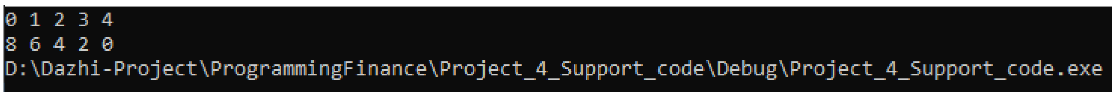
Here is just a code example on how to use iterator to visit or manipulate elements in vector container:

```
#include <iostream>
#include <vector>//Import STL container vector and its iterator
using namespace std;
int main()
{
     vector<int> my_vector;
     //my vector will store int type data and does not have a fixed length

     for (int n = 0; n < 5; ++n)
          my_vector.push_back(n);
     //push_back()member function will push
     //numbers 0 to 5 into the vector container

     vector<int>::iterator my_iterator;
     //Create a vector int type iterator my_iterator

     for (my_iterator = my_vector.begin();
          my_iterator != my_vector.end();
          ++my_iterator)
     {  //Using iterator to visit each element in my_vector

          cout << *my_iterator << " ";  //iterator is actually a pointer
          //*my_vector is going to visit the element my_vector is pointing at
          *my_iterator *= 2;  //Double each element
     }
```

```
        cout << endl;

        vector<int>::reverse_iterator my_reverse_iterator;
        //Creating a reverse iterator
        for (my_reverse_iterator = my_vector.rbegin();
            my_reverse_iterator != my_vector.rend();
            ++my_reverse_iterator)
            cout << *my_reverse_iterator << " ";
        //Print out the iterator pointing element from the end to the beginning
        return 0;
    }
```

In this code example, I show how to use iterator to visit element in a container and how can we manipulate the data in the container. Moreover, I try to use reverse iterator to express that, there are multiple kind of iterator for C++ programmer to utilize based on their own design or functional need. Iterator is a tool for programmer better understand and use the STL library.

Running result:



```
0 1 2 3 4
8 6 4 2 0
D:\Dazhi-Project\ProgrammingFinance\Project_4_Support_code\Debug\Project_4_Support_code.exe
```

Fig 2 Iterator code example result

3. Algorithm

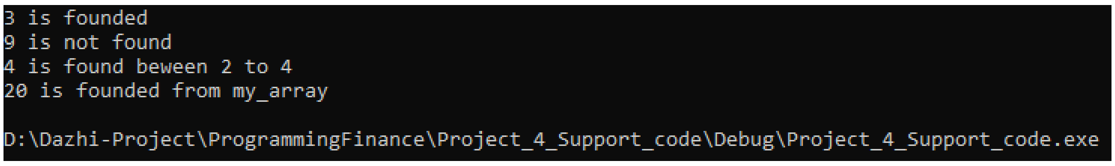
STL algorithm is a very important part in STL library. Since there are many common data manipulation or unique data manipulation in those containers, C++ programmer come up with STL algorithms to solve those functional problems. For example, inserting, deleting, finding, and sorting are those functional problems we need to use algorithm to achieve. Algorithm usually use iterator to manipulate those elements in the containers. Also, some algorithm will change the data members in the container, and some will not. For example, sorting function will change the order in the container, remove function will erase one element from the container but count function will just count the number of elements in one container without any data manipulation.

Here, I will just show you the find algorithm as a code example:

```
    #include <vector>
     #include <algorithm>
     #include <iostream>
     using namespace std;
     int main() {
        vector<int> my_vector;
        for (int n = 1; n < 5; ++n)
            my_vector.push_back(n);//my_vector includes 1,2,3,4
        vector<int>::iterator my_iterator; //Create an iterator for my_vector

        my_iterator = find(my_vector.begin(), my_vector.end(), 3);
        //Find element 3 in my_vector
        //If exist, return the iterator where 3 is

        if (my_iterator != my_vector.end())
            //If find algorithm cannot find target, it will return my_vector.end()
            //Notice that my_vector.end() does not point to the end of numbers
            //my_vector.end()-1 points to 4
            //my_vector.end() points to the end of container which store no element.
            cout << *my_iterator << " is founded" << endl;
            //Find number 3 in my_vector

        my_iterator = find(my_vector.begin(), my_vector.end(), 9);
        if (my_iterator == my_vector.end())
            cout << "9 is not found " << endl; //Did not find 9 in my_vector

        my_iterator = find(my_vector.begin() + 1, my_vector.end() - 1, 4);
        //Find 4 from the second element of my_vector(2) to the last element(4)
        //Notice my_vector.begin() really points to the begin of the container
        //my_vector.begin() points at number 1

        cout << *my_iterator << " is found beween 2 to 4" << endl;

        int my_array[10] = { 10,20,30,40 };
        int* my_pointer = find(my_array, my_array + 4, 20);
        //Even without container and iterator, algorithm could still be used
        //Be careful about the data type and pointer data type
        if (my_array == my_array + 4)
            //my_array+4 is the same way as my_vector.end()
            cout << "not found" << endl;
        else
            cout << *my_pointer << " is founded from my_array" << endl;
    }
```

In my code example, it is not hard to find that algorithm usually works with iterator and container together. In this code example, I simply show you how find algorithm works in C++. However, there are more than 70 kinds of algorithms support different kind of functions for C++ programmers better utilize STL library.

Running result:



```
3 is founded
9 is not found
4 is found beween 2 to 4
20 is founded from my_array

D:\Dazhi-Project\ProgrammingFinance\Project_4_Support_code\Debug\Project_4_Support_code.exe
```

Fig 3 Find algorithm code example result

4. Lambda expression

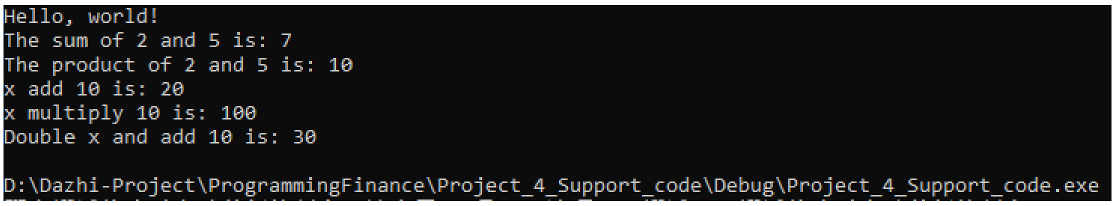
Lambda expression is new technique created since C++ 11 standard. Although C++ programmer could achieve the same result in more complex as what lambda expression

could do, lambda expression still brings a lot of influence to C++ programming. It makes code more readable and convenient. Usually lambda expression stands for [capture](parameters){body} structure. Capture part allows lambda expression to capture and use some existing variable. Parameters part allows lambda expression to set important parameters needed before being used. Body part is just like the same as function's body, which is coded by programmer to achieve functions. However, lambda expression is still able to return a type. Usually it will automatically adjust the return type, but C++ programmers could specify a return type to make the code more readable.
Here is just some lambda expression code example for you to better understand:

```
#include<iostream>
using namespace std;
int main()
{
    auto basicLambda = [] { cout << "Hello, world!" << endl; };
    basicLambda();
    /* In this case we create a lambda expression without any
    captures or parameters, simply show how lambda expression
    achieve functional part
    */

    auto add = [](int a, int b) -> int { return a + b; };
    // auto will automatically adjust the return type
    // In this case, the return type is int
    auto multiply = [](int a, int b) { return a * b; };
    int sum = add(2, 5);
    int product = multiply(2, 5);
    cout << "The sum of 2 and 5 is: " << sum << endl;
    cout << "The product of 2 and 5 is: " << product << endl;
    /* In this code part we are showing you how does parameters and
    return type work in lambda expression
    */

    int x = 10;
    auto add_x = [x](int a) { return a + x; };
    //Capture part will capture variable x, then x could be used in body part
    auto multiply_x = [&x](int a) { return a * x; };
    // In this case lambda expression will capture the reference of x
    cout << "x add 10 is: " << add_x(10) << endl;
    cout << "x multiply 10 is: " << multiply_x(10) << endl;
    /* In this part we are showing how does capture part works.
    Lambda expression could capture many types like reference, data copy etc.
    Programmers could choose what to capture by their design need.
    */
    auto change_x_add = [x](int a) mutable { x *= 2; return a + x; };
    //copy capture x
    cout << "Double x and add 10 is: " << change_x_add(10) << endl;
    return 0;
    /* This code part is going to show us that, by using copy capture
    the lambda expression is const type. If we want to change
    the copy of x, we need to add mutable identifier
    If we directly use copy capture x multiply 2, compiler error occurs.
    */
}
```

In this code example I simply show how lambda expression works in C++. As there are still a lot of usage about lambda expression I have not mentioned, but lambda expression really helps C++ programmer easily coding.

Running result:



```
Hello, world!
The sum of 2 and 5 is: 7
The product of 2 and 5 is: 10
x add 10 is: 20
x multiply 10 is: 100
Double x and add 10 is: 30

D:\Dazhi-Project\ProgrammingFinance\Project_4_Support_code\Debug\Project_4_Support_code.exe
```

Fig 4 Lambda expression code example result