MODULE 33
--THE STL--
ALGORITHM PART I

My Training Period: hours

Note: Compiled using Microsoft Visual C++ .Net, win32 empty console mode application. **g++** compilation examples given at the end of this Module.

Ability

- Able to understand the fundamental of the algorithm.

33.1 Algorithms

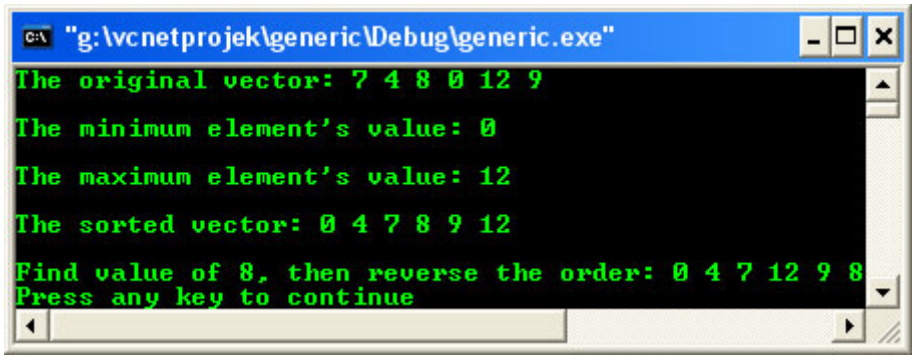
33.1.1 Some Introduction Story

- We have covered containers and iterators, now we are going to complete our discussion by introducing algorithms.
- We create data structure by using containers, and then we use iterators to traverse or iterate the data structure. During the iteration we will use algorithm, a set of rules that actually defined what to do to the data structure: sorting, searching, re ordering etc.
- The STL provides several standard algorithms for the processing of elements of collections.
- These algorithms offer general fundamental services, such as searching, sorting, copying, reordering, modifying, and numerical processing, so that no need for us to start developing program portions or routines from scratch. Furthermore they have been tested and your task is to learn how to manipulate these 'creatures'.
- Algorithms are global functions that operate with iterators; hence, all algorithms can be implemented once for any container type. The algorithm might even operate on elements of different container types. Furthermore you can also use the algorithms for user-defined container types.
- Let's start with a simple example, the use of STL algorithms. Consider the following program:

```
//simple algorithm example
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
//declare a vector and the iterator
vector<int> vec;
vector<int>::iterator pos;

//insert elements from 1 to 6 in arbitrary order
vec.push_back(7);
vec.push_back(4);
vec.push_back(8);
vec.push_back(0);
vec.push_back(12);
vec.push_back(9);

//print the vector...
cout<<"The original vector: ";
for(pos = vec.begin(); pos != vec.end(); pos++)
cout<<*pos<< " ";
cout<<endl;

//find and print minimum and maximum elements
pos = min_element(vec.begin(), vec.end());
cout<<"\nThe minimum element's value: "<<*pos<<endl;

pos = max_element(vec.begin(), vec.end());
cout<<"\nThe maximum element's value: "<<*pos<<endl<<endl;

//sort algorithm, sort all elements
sort(vec.begin(), vec.end());

//print the vector...
cout<<"The sorted vector: ";
for(pos = vec.begin(); pos != vec.end(); pos++)
cout<<*pos<< " ";
```

```
cout<<endl<<endl;

cout<<"Find value of 8, then reverse the order: ";
//find algorithm, find the first element with value 8
pos = find(vec.begin(), vec.end(), 8);

//reverse algorithm, reverse the order of the
//found element with value 8 and all the following elements
reverse(pos, vec.end());

//print all elements
for(pos=vec.begin(); pos!=vec.end(); ++pos)
cout<<*pos<<' ';
cout << endl;
}
```

Output:



- To be able to call the algorithms, you must include the header file `<algorithm>` as shown below.

```
#include <algorithm>
```

- The first two algorithms called are `min_element()` and `max_element()`.
- They are called with two parameters that define the range of the processed elements. To process all elements of a container you simply use `begin()` and `end()`.
- Both algorithms return an iterator for the minimum and maximum elements respectively. Thus, in the statement

```
pos = min_element(vec.begin(), vec.end());
```

- The `min_element()` algorithm returns the position of the minimum element (if there is more than one, the algorithm returns the first). The next statement prints that element:

```
cout<<"\nThe minimum element's value: "<<*pos<<endl;
```

- And for the `max_element()`

```
cout<<"\nThe maximum element's value: "<<*pos<<endl<<endl;
```

- The next algorithm called is `sort()`. As the name indicates, it sorts the elements of the range defined by the two arguments.
- You could also pass an optional sorting criterion. The default sorting criterion is `operator <` (less than). Thus, in this example all elements of the container are sorted in ascending order:

```
sort(vec.begin(), vec.end());
```

- So afterward, the vector contains the elements in this order:

```
0 4 7 8 9 12
```

- The `find()` algorithm searches for a value inside the given range. In this example, it searches the first element that is equal to the value 3 in the whole container:

```
pos = find(vec.begin(), vec.end(), 8);
```

- If the `find()` algorithm is successful, it returns the iterator position of the element found. If it fails, it returns the end of the range, the past-the-end iterator, which is passed as the second argument.
- In this example, the value 8 is found as the third element, so afterward pos refer to the third element of `vec`.
- The last algorithm called in the example is `reverse()`, which reverses the elements of the passed range. Here the third element that was found by the `find()` algorithms and the past-the end iterator are passed as arguments:

```
reverse(pos, vec.end());
```

- This call reverses the order of the third element up to the last one.

33.1.2 Ranges

- All algorithms process one or more ranges of elements. Such a range might, but is not required to, embrace all elements of a container.
- Therefore, to be able to handle subsets of container elements, you pass the **beginning** and the **end** of the range as **two separate arguments** rather than the whole collection as one argument.
- The caller must ensure that the first and second arguments define a valid range. This is the case if the end of the range is reachable from the beginning by iterating through the elements.
- This means, it is up to the programmer to ensure that both iterators belong to the same container and that the beginning is not behind the end. If this is not the case, the behavior is undefined and endless loops or forbidden memory access may result.

33.1.3 Multiple Ranges

- Several algorithms process more than one range. In this case you usually must define both the beginning and the end only for the first range.
- For all other ranges you need to pass **only their beginnings**. The ends of the other ranges follow from the number of elements of the first range.
- For example, the following call of `equal()` compares all elements of the collection `vec1` element-by-element with the elements of `vec2` beginning with its first element:

```
if(equal(vec1.begin(), vec2.end(), vec2.begin()))
{...}
```

- Thus, the number of elements of `vec2` that are compared with the elements of `vec1` is specified indirectly by the number of elements in `vec1`.
- Hence, when you call algorithms for multiple ranges, and make sure the second and additional ranges have at least as many elements as the first range.
- That means make sure that **destination** ranges are big enough for algorithms that write to collections.
- Consider the following program:

```
//algorithms, example
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
        list<int> lst1;
        vector<int> vec1;
        //insert elements from 1 to 9
        for(int i=1; i<=9; ++i)
                lst1.push_back(i);

        //RUNTIME ERROR:
        // - overwrites nonexistence elements in the destination
        copy (lst1.begin(), lst1.end(), //the source
        vec1.begin()); //the destination
        ......
}
```

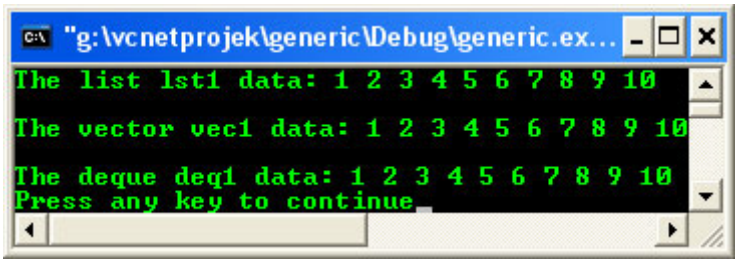
- Here, the `copy()` algorithm is called. It simply copies all elements of the first range into the destination range.
- Notice that, for the first range, the beginning and the end are defined, whereas for the second range, only the beginning is specified.
- However, the algorithm overwrites rather than inserts. So, the algorithm requires that the destination has enough elements to be overwritten. If there is not enough room, as in this case, the result is undefined behavior and normally compiler will generate errors.
- In practice, this often means that you overwrite whatever comes after the `vec.end()`. To avoid these errors, you can:

    1. Ensure that the destination has enough elements on entry, or
    2. Uses insert iterators.

- To make the destination big enough, you must either create it with the correct size or change its size explicitly. Both alternatives apply only to sequence containers (vectors, deques, and lists).
- The following program shows how to increase the size of containers:

```
//algorithm, example
#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
list<int> lst1;
list<int>::iterator pos;
vector<int> vec1;
vector<int>::iterator pos1;

//insert elements from 1 to 10
for(int i=1; i<=10; ++i)
lst1.push_back(i);

//display data
cout<<"The list lst1 data: ";
for(pos=lst1.begin(); pos!=lst1.end(); pos++)
cout<<*pos<<" ";
cout<<endl;

//resize destination to have enough
//room for the overwriting algorithm
vec1.resize(lst1.size());

//copy elements from first into second collection
//overwrites existing elements in destination
copy(lst1.begin(), lst1.end(), //source
vec1.begin()); //destination

cout<<"\nThe vector vec1 data: ";
for(pos1=vec1.begin(); pos1!=vec1.end(); pos1++)
cout<<*pos1<<" ";
cout<<endl;

//create third collection with enough allocation
//initial size is passed as parameter
deque<int> deq1(lst1.size());
deque<int>::iterator pos2;

//copy elements from first into third collection
copy(lst1.begin(), lst1.end(), //source
deq1.begin()); //destination

cout<<"\nThe deque deq1 data: ";
for(pos2=deq1.begin(); pos2!=deq1.end(); pos2++)
cout<<*pos2<<" ";
cout<<endl;
}
```

Output:



- Here, `resize()` is used to change the number of elements in the existing container `vec1`:

```
vec1.resize(vec1.size());
```

- `deq1` is initialized with a special initial size so that it has enough room for all elements of `lst1`:

```
deque<int> deq1(vec1.size());
```

- Note that both resizing and initializing the size create new elements. These elements are initialized by their default constructor because no arguments are passed to them.
- You can pass an additional argument both for the constructor and for `resize()` to initialize the new elements.

33.2 Algorithms versus Member Functions

- Even if you are able to use an algorithm, it might not be suitable in certain circumstances.
- You have learned from the program examples in the module (container), a container might have member functions that provide much better solution such as in term of performance.
- Calling `remove()` for elements of a list is a good example of this. If you call `remove()` for elements of a list, the algorithm doesn't know that it is operating on a list.
- Thus, it does what it does for any container: It reorders the elements by changing their values. If, for example, it removes the first element, all the following elements are assigned to their previous elements.
- This behavior contradicts the main advantage of lists, the ability to insert, move, and remove elements by modifying the links instead of the values. To overcome this, lists provide special member functions for all manipulating algorithms and it is better to use them instead of using algorithm version.

33.3 User Defined Generic Functions

- The STL is an extensible framework. This means you can write your own functions and algorithms to process elements of collections and these operations may also be generic.
- However, to declare a valid iterator in these operations, you must use the type of the container, which is different for each container type.
- To facilitate the writing of generic functions, each container type provides some internal type definitions.

33.4 Functions as Algorithm Arguments

- To increase their flexibility and power, several algorithms allow the passing of user-defined auxiliary functions. These functions are called internally by the algorithms.

33.5 Predicates

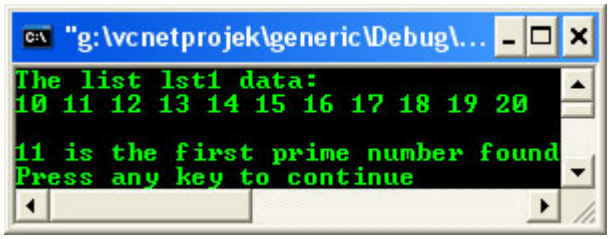
- A special kind of auxiliary function for algorithms is a **predicate**.
- Predicates are **functions** that return a `Boolean` value. They are often used to specify a sorting or a search criterion.
- Depending on their purpose, predicates are **unary** or **binary**. Note that not every unary or binary function that returns a Boolean value is a valid predicate.
- The STL requires that predicates always yield the same result for the same value.

33.5.1 Unary Predicates

- Unary predicates check a specific property of a single argument. A typical simple example is a function that is used as a search criterion to find the first prime number:

```
//Algorithm, simple example
#include <iostream>
#include <list>
#include <algorithm>
//for abs()
#include <cstdlib>
using namespace std;

//predicate, which returns whether an integer is a prime number
bool isPrimeNum(int number)
{
//ignore negative sign
number = abs(number);

//0 and 1 are prime numbers
if(number == 0 || number == 1)
{
        return true;
}

//find divisor that divides without a remainder
int divisor;
for(divisor = (number/2); (number%divisor) != 0; --divisor)
{;}

//if no divisor greater than 1 is found, it is a prime number
return divisor == 1;
}

int main()
{
list<int> lst1;

//insert elements from 24 to 30
for(int i=10; i<=20; ++i)
lst1.push_back(i);

//search for prime number
list<int>::iterator pos;

cout<<"The list lst1 data:\n";
for(pos=lst1.begin(); pos!=lst1.end(); pos++)
cout<<*pos<<" ";
cout<<endl<<endl;

pos = find_if(lst1.begin(), lst1.end(), //range
isPrimeNum); //predicate

if(pos != lst1.end())
{
//found
cout<<*pos<<" is the first prime number found"<<endl;
}
else {
//not found
cout<<"no prime number found"<<endl;
}
}
```

Output:



- In this example, the `find_if()` algorithm is used to search for the first element of the given range for which the passed unary predicate yields true.
- The predicate is the `isPrimeNum()` function. This function checks whether a number is a prime number. By using it, the algorithm returns the first prime number in the given range.

- If the algorithm does not find any element that matches the predicate, it returns the end of the range (its second argument).

33.5.2 Binary Predicates

- Binary predicates typically compare a specific property of two arguments.
- For example, to sort elements according to your own criterion you could provide it as a simple **predicate function**.
- This might be necessary because the elements do not provide `operator <` or because you wish to use a different criterion.
- The following example sorts elements of a set by the first name and last name of a person:

```
//algorithm, predicate
#include <iostream>
#include <string>
#include <deque>
#include <algorithm>
using namespace std;

class Person
{
public:
string firstname() const;
string lastname() const;
...
};

//binary function predicate:
//returns whether a person is less than another person
bool SortCriterion(const Person& p1, const Person& p2)
{
//a person is less than another person
//if the last name is less
//if the last name is equal and the first name is less

return p1.lastname()<p2.1astname() || (!(p2.1astname()<p1.lastname()) &&
p1.firstname()<p2.firstname());
}

int main()
{
deque<Person> deq1;
...
sort(deq1.begin(), deq1.end(), SortCriterion);
...
}
```

- Note that you can also implement a sorting criterion as a function object. This kind of implementation has the advantage that the criterion is a type, which you could use, for example, to declare sets that use this criterion for sorting its elements.

33.6 Complexity and the Big-O Notation

- A specialized notation is used to compare the relative complexity of an algorithm. Using this measure, we can categorize quickly the relative runtime of an algorithm as well as perform qualitative comparisons between algorithms. This measure is called *Big-O notation*.
- The Big-O notation expresses the **runtime** of an algorithm as a function of a given input of size **n**. For example, if the runtime grows linearly with the **number of elements** (doubling the input doubles the runtime) the complexity is **O(n)**. If the runtime is independent of the input, the complexity is **O(1)**.
- The following Table lists typical values of complexity and their Big-O notation.

Type	Notation	Description
Constant	`O(1)`	The runtime is independent of the number of elements.
Logarithmic	`O(log(n))`	The runtime grows logarithmically with respect to the number of elements.
Linear	`O(n)`	The runtime grows linearly (with the same factor) as the number of elements grows.
n-log-n	`O(n *log(n))`	The runtime grows as a product of linear and logarithmic complexity.
Quadratic	$O(n^2)$	The runtime grows quadratically with respect to the

		number of elements.

Table 33.1: The O-Notation

- It is only a rule of thumb; the algorithm with optimal complexity is not necessarily the best one.
- Some complexity definitions in the C++ reference manual are specified as **amortized**. This means that the operations **in the long term** behave as described.
- However, a single operation may take longer than specified. For example, if you append elements to a dynamic array, the runtime depends on whether the array has enough memory for one more element.
- If there is enough memory, the complexity is constant because inserting a new last element always takes the same time.
- But, if there is not enough memory, the complexity is linear. This is because, depending on the actual number of elements, you have to allocate new memory and copy all elements.
- Reallocations are rather rare, so any sufficiently long sequence of that operation behaves as if each operation has constant complexity. Thus, the complexity of the insertion is "amortized" constant time.
- Program example compiled using g++.

```
//********algo.cpp*********
//algorithm, example
#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
list<int> lst1;
list<int>::iterator pos;
vector<int> vec1;
vector<int>::iterator pos1;

//insert elements from 1 to 10
for(int i=1; i<=10; ++i)
lst1.push_back(i);

//display data
cout<<"The list lst1 data: ";
for(pos=lst1.begin(); pos!=lst1.end(); pos++)
cout<<*pos<<" ";
cout<<endl;

//resize destination to have enough
//room for the overwriting algorithm
vec1.resize(lst1.size());

//copy elements from first into second collection
//overwrites existing elements in destination
copy(lst1.begin(), lst1.end(), //source
vec1.begin()); //destination

cout<<"\nThe vector vec1 data: ";
for(pos1=vec1.begin(); pos1!=vec1.end(); pos1++)
cout<<*pos1<<" ";
cout<<endl;

//create third collection with enough allocation
//initial size is passed as parameter
deque<int> deq1(lst1.size());
deque<int>::iterator pos2;

//copy elements from first into third collection
copy(lst1.begin(), lst1.end(), //source
deq1.begin()); //destination

cout<<"\nThe deque deq1 data: ";
for(pos2=deq1.begin(); pos2!=deq1.end(); pos2++)
cout<<*pos2<<" ";
cout<<endl;
}
```

```
[bodo@bakawali ~]$ g++ algo.cpp -o algo
[bodo@bakawali ~]$ ./algo
```

```
The list lst1 data: 1 2 3 4 5 6 7 8 9 10

The vector vec1 data: 1 2 3 4 5 6 7 8 9 10

The deque deq1 data: 1 2 3 4 5 6 7 8 9 10
```

----------End of Algorithm Part I----------

---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.