73. STL vector class

Objectives

- Use the `std::vector` class

STL

STL = Standard Template Library .

STL is a collection of C++ class templates that come with most C++ compilers . STL provides many useful classes. We will only scratch the surface. More information on STL will appear in CISS350, CISS358, CISS362, etc.

Vector class template

Recall you have an `IntDynArr` class for modeling dynamic arrays. There is a similar class called `vector`. To use it, you must include the `vector` header (i.e. `#include <vector>`). The vector class is in the namespace `std`. So

```
std::vector< int > v;
```

will declare object `v` so that `x` models a dynamic array of integer values. In this case the size of `v` is 0 – it has no values. You can also do this:

```
std::vector< int > v(10);
```

In this case the size of `v` is 10. You can also do

```
std::vector< int > v(10, 42);
```

in which case `v` is size 10 and all the values in `v` are set to 42.

Exercise. Create a vector of 10 characters, setting all values to `' '`.

Member Functions/Methods

```
assign()    // Erases a vector and copies the
            // specified elements to the empty
            // vector.

at()        // Returns a reference to the element at
            // a specified location in the vector.

back()      // Returns a reference to the last
            // element of the vector.

capacity()  // Returns the number of elements
            // that the vector could contain
            // without allocating more storage.

clear()     // Erases all elements of the vector.

empty()     // Tests if the vector container is
            // empty.

erase()     // Removes an element or a range
            // of elements in a vector from
            // specified positions.

front()     // Returns a reference to the first
            // element in a vector.

insert()    // Inserts an element or a number of
            // elements into the vector at a
            // specified position.

max_size()  // Returns the maximum possible
            // length of the vector.

pop_back()  // Deletes the element at the end of the
            // vector.

push_back() // Add an element to the end of the
            // vector.

resize()    // Specifies a new size for a vector.

reserve()   // Reserves a minimum length of storage
            // for a vector object.

size()      // Returns the number of elements in the
            // vector.

swap()      // Exchanges the elements of two vectors.

vector()    // Constructor. Constructs a vector of a
            // specific size or with elements of
            // specific value.

operator[]  // Returns a reference to the vector
```

```
            // element at a specified position.
```

Operators ==, !=, = do the obvious things.

Now for a quick experiment on some of the methods. Run the following program and study the output carefully.

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

std::ostream & operator<<(std::ostream & cout,
                          const std::vector< int > & v)
{
    std::string sep = "";
    cout << '{';
    for (int i = 0; i < v.size() - 1; ++i)
    {
        cout << sep << v[i];
        if (i == 0) sep = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    int s = 3;
    std::vector< int > v(s);

    v[0] = 1; v[1] = 10; v[2] = 100;

    std::cout << "v: " << v << "\n";
    v[0] = 42;
    std::cout << "v: " << v << "\n";

    std::vector< int > u(v);
    std::cout << "u: " << u << "\n";

    if (u == v)
    {
        std::cout << "same\n";
    }

    v.resize(10);
    std::cout << "v: " << v << "\n";

    std::vector< int > w(4);
    std::cout << "w: " << w << "\n";

    w = u;
    std::cout << "w: " << w << "\n";

    return 0;
}
```

Study the output carefully. Try all the methods on your own – most of them are very similar to the methods in the `IntDynArr` class.

Iterators

Recall that we have written an `IntPointer` class so that objects of this class behaves like integer pointers. Your `std::vector` class also comes with a pointer class that can be used to point to values in a `std::vector` object. These are called **iterator** classes.

Besides accessing values in the vector using an index integer value, you can also use iterators. Iterators are object which pretend to be pointers. This is similar to the previous program but using iterators. Run it and study the code very carefully.

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

std::ostream & operator<<(std::ostream & cout,
                          const std::vector< int > & v)
{
    cout << '{';
    std::string sep("");

    for (std::vector< int >::const_iterator p =
                                                v.begin();
         p != v.end(); ++p)
    {
        cout << sep << (*p);
        if (sep == "") sep = ", ";
    }
    cout << '}';
    return cout;
}


int main()
{
    int s = 3;
    std::vector< int > v(s);

    v[0] = 1; v[1] = 10; v[2] = 100;

    std::cout << "v: " << v << "\n";

    std::vector< int >::iterator p = v.begin();
    *p = 42;
    ++p;
    *p = 43;
    p = v.end();
    --p;
    *p = 99;
    std::cout << "v: " << v << "\n";

    p = v.begin();
    p += 1;
    v.erase(p);
    std::cout << "v: " << v << "\n";
```

```
    return 0;
}
```

`std::vector< int >::const_iterator` objects are pointers that do not change the value they point to. `std::vector< int >::iterator` objects are pointers that can change the value they point to. When you do

```
std::vector< int >::iterator p = v.begin();
```

the iterator `p` will point to the first value of `v`. In this case `p` can change the value that it points to. So you can change the first value of v to 42 if you do this:

```
std::vector< int >::iterator p = v.begin();
*p = 42;
```

However if you do

```
std::vector< int >::const_iterator p = v.begin();
```

then `p` cannot change the value it points to. So for instance the compiler will yell at you if you write this:

```
std::vector< int >::const_iterator p = v.begin();
*p = 42;
```

By the way the reason for the second "`::`" in

```
std::vector< int >::iterator p;
```

is because the iterator class is defined **inside** the `std::vector` class:

```
template < typename T >
class vector
{
...
    class iterator
    {
    ...
    };
    class const_iterator
    {
    ...
    };
...
};
```

Yes, that's right … you can nest class definitions.

Let's go on. If you do

```
std::vector< int >::iterator p = v.end();
```

then p will point to the **very first value beyond** the last value of v. Therefore a for-loop that runs an iterator through the value of v would look like:

```
for (std::vector< int >::iterator p = v.begin();
     p != v.end(); ++p)
{
    ...
}
```

In this case the iterator p can change the values of v. If you do not want to change the value of v, you can use a constant iterator:

```
for (std::vector< int >::const_iterator p = v.begin();
     p != v.end(); ++p)
{
    ...
}
```

Like a regular pointer, you can do pointer arithmetic on your iterators.

```
    std::vector< int >::iterator p = v.begin();
    *p = 42; // Index 0 value of v is change to 42
    ++p;     // p now points to index 1 value of v
    p += 3   // p not points to index 4 value of v
```

You can also do `--p` and `p -= 5`.

While `v.begin()` gives you the address of the index `0` value of v, `v.end()` will give you the address of the last value of v, i.e., the value at index `v.size() - 1`.

If p is an iterator,

```
v.erase(p)
```

will erase the value that p points to – values in v after the value that is being erased will be move backward by one index step and the size of v is decremented by 1. You can also erase a whole section of your vector. For instance if v has 100 values, then

```
v.erase(v.begin() + 3, v.begin() + 10);
```

will erase the values from index 0 + 3 to 0 + 10; the size decrements by 8. Make sure you create a test case for this and verify it for yourself.

By the way, make sure you see

- the difference between `v.front()` and `v.begin()` and
- the difference between `v.back()` and `v.end()`.

If you don't see the difference, start from the beginning of this document and read it all over again.

Iterators are **extremely important** when working with containers classes of C++ because all container classes (like std::string

and std::vector) have iterators. So iterators provide a **common way** to access, modify, delete values in these containers.

Exercise. Create a vector v of 10 doubles. Using iterators, do the following:

- Set the first value to 3.1
- Set the second value to 3.14
- Set the third value to 3.141
- Using a for-loop, set the rest to -1.0
- Using a for-loop, print all the values of v.
- Erase all the values which are -1.0.
- Print the size of v – you should get 3.
- Using a for-loop, print all the values of v – you should see only 3 values.

Exercise. Implement the following function prototypes (which you learn from CISS240):

```
int count(const std::vector< int > & v, int target);
int linearsearch(const std::vector< int > & v, int target);
void bubblesort(std::vector< int > & v);
int binarysearch(const std::vector< int > & v, int target);
```

2D arrays

Since `std::vector` is a class template, I can create a vector where each value in this vector … **is also a vector!!!**

```
std::vector< std::vector< int > > v;
v.resize(3);
```

At this point, `v` has 3 values, i.e., `v` has 3 vectors. Now I'll make each of the 3 vectors in `v` into vectors of size 5:

```
for (int i = 0; i < 3; ++i)
{
    v[i].resize(5);
}
```

Viola … `v` is now a 2D array of size 3-by-5. Get it?

Exercise. Write a function to print the values of a `std::vector< std::vector < int > >` object.

Exercise. Write a function to print the `v` above. Here's an example of how the output should look like:

```
[[1 3 5 7]
 [2 4 6 8]
 [3 6 9 12]]
```

Your function should work for any sizes of `v`.

Exercise. Write a tic-tac-toe game where the board size is n-by-n where n is specified by the user.

Exercise. Create a 2D array of integer values using `std::vector` where even index rows have size 4 and odd index rows have size 5.

References

- C++ textbooks usually contain information on STL classes.
- There are lots of C++ reference on the web. Example: cppreference.com.
- Knowing the STL classes help the engineering of software. However knowledge of constructing your own classes is also important because (obviously) STL does not provide all the possibly useful classes! Sometimes you might need to write your own classes for performance reasons.
- At other times, because of the nature of your problem, you might be able to write your own class which will produce objects with faster methods.
- Therefore it's extremely important to know how some of the standard classes in your C++ compilers are implemented.