【C++】Day nine(2)

Class	C++
Date	@November 27, 2021
Material	
Series Number	
Summary	

【Ch3】Library vector Type

3.3 Library vector Type

A vector is a collection of objects, all of which have the same type. Every object in the collection has an associated index, which gives access to that object. A vector is often referred to as a container because it "contains" other objects.

To use a vector, we must include the appropriate header.

```
#incluede <vector>
using std::vector;
```

A vector is a class template.

Templates are not themselves functions or classes. Instead, they can be thought of as instructions to the compiler for generating classes or functions. The process that the compiler uses to create classes or functions from templates is called instantiation. When we use a template, we specify what kind of class or function we want the compiler to instantiate.

For a class template, we specify which class to instantiate by supplying addtional information, the nature of which depends on the template. We supply the information insde a pair of angle brackets `<>` following the template's name.

```
vector<int> ivec; //ivec holds objects of type int
vector<Sales_item> Sales_vec; //holds Sales_items
vector<vector<string>> file; //vector whose elements are vectors of string
```

In this example, the compiler generates three distinct types from the vector template: `vector<int>`, `vector<Sales_item>`, and `vector<vector<string>>`.

Note: vector is a template, not a type. Types generated from vector must include the element type, for example, `vector<int>`.

We can define vectors to hold objects of most any type. Because references are not objects, we cannot have a vector of references. However, we can have vectors of most other built-in types and most class types. In particular, we can have vectors whose elements are themselves vectors.

3.3.1 Defining and Initializing vectors

The vector template controls how we define and initialize vectors.

Table 3.4. Ways to Initialize a vector

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

We can default initialize a vector, which creates an empty vector of the specified type:

```
vector<int> intVec; //default initialization: intVec has no elements
```

We can also copy elements from another vector:

```
vector<int> ivec = {1, 2, 3};
vector<int> ivec2(ivec); //copy elements of ivec into ivec2
vector<int> ivec3 = ivec; //copy elements of ivec into ivec3
vector<string> svec(ivec2); //error: svec holds strings, not ints
```

List Initializing a vector

Under the new standard, we can list initialize a vector from a list of zero or more initial element values enclose in curly braces:

```
vector<string> articles = {"a", "an", "the"};
```

Creating a Specified Number of Elements

We can also initialize a vector from a count and an element value. The count determines how many elements the vector will have; the value provides the initial value for each of those elements

```
vector<int> ivec(10, -1); //ten int elements, each initialized to -1
vector<string> svec(10, "hi"); //ten strings; each element is "hi"
```

Value Initialization

We can usually omit the value and supply only a size. In this case, the library creates a value-initialized element initializer for us. This library-genereated value is used to initialize each element in the container.

If the vector holds elements of a built-in type, such as int, then the element initializer has a value of 0. If the elements are of a class type, such as string, then the element initializer is iteself default initialized.

```
vector<int> ivec(10); //ten elements, each initialized to 0
vector<string> svec(10); //ten elements, each an empty string
```

There are two restrictions on this form of initialization:

1. The first restriction is that some classes require that we always supply an explicit initializer. If our vector holds objects of a type that we cannot default initialize, then we must supply an initial element value; it is not possible to create vectors of such types by supplying only a size.
2. The second restriction is that when we supply an element count without also supplying an initial value, we must use the direct form of initialization:

```
vector<int> vi = 10; //error: msut use direct initialization to supply a size
```

When there is no way to use the initializers to list initialize the object, those values will be used to construct the object.

```
vector<string> v{10}; //v has ten default-initialized elements
```

3.3.2 Adding Elements to a vector

It is better to create an empty vector and use a vector mbmer names `push_back` to add elements at run time. The `push_back` operation takes a value and "pushes" that value as a new last element onto the "back" of the vector.

```
vector<int> v2; //empty vector
for(int i = 0; i != 100; i++)
  v2.push_back(i); //append sequential integers to v2
```

Key Concept: vectors Grow Efficiently

Becuase vectors grow efficiently, it is often unnecessary—and can result in poorer performance— to define a vector of a specific size. The execption to this rule is if all the elements actually need the same vaule.

Note: The body of a range for must not change the size of the sequence over which it is iterating.