【C++】Day35

Class	C++
Date	@January 6, 2022
Material	
Series Number	
Summary	

【Ch8】Sequential Container

9.4 How a vector Grows

Given that elements are contiguous, and that the size of the container is flexible, consider what must happen when we add an element to a vector or a string: If there is no room for the new element, the container can't just add an element somewhere else in memory-the elements must be contiguous.

Instead, the container must allocate new memory to hold the existing elements plus the new one, move the elements from the old location into the new space, add the new element, and deallocate the old memory.

When they have to get new memory, `vector` and `string` implementations typlically allocate capacity beyond what is immediately needed. The container holds this storage in reserve and uses it to allocate new elements as they are added. Thus, there is no need to reallocate the container for each new element.

Members to Manage Capcity

The `vector` and `string` types provide memebers that let us interact with the memory-allocation part of the implementation.

Table 9.10. Container Size Management

shrink_to_fit valid only for vector, string, and deque. capacity and reserve valid only for vector and string.	
`c.shrink_to_fit()`	Request to reduce `capacity()` to equal `size()`.
`c.capacity()`	Number of elements `c` can have before reallocation is necessary.
`c.reserve(n)`	Allocate space for at least `n` elements.

The `capacity()` operation tells us how many elements the container can hold before it must allocate more space.

The `reserve()` option lets us tell the container how many elements it should be prepared to hold.

Note: `reserve()` does not change the number of elements in the container; it affects only how much memory the vector preallocates.

A call to `reserve()` changes the capacity of the vector only if the requested space exceeds the current capacity. If the requested size is greater than the current capacity, reserve allocates at least as mushc as(and may allocate more than) the requested amount.

If the requested size is less than or equal to the existing capacity, reserve does nothing. In particular, calling reserve with a size smaller than capacity does not cause the container to give back memory.

Thus, after calling reserve, the capacity will be greater than or equal to the argument passed to reserve.

Under the new library, we can call `shrint_to_fit` to ask a deque, vector, or string to return unneeded memory. This function indicates that we no longer need any excess capacity.

However, the implementation is free to ignore this request. There is no guarantee that a call to `shrink_to_fit` will return memory.

capacity and size

The difference between capacity and size:

- The size of a container is the number of elements it already holds
- Its capacity is how many elements it can hold before more space must be allocated.

The following code illustrates the interaction between size and capacity:

```
vector<int> vec;
//size should be zero; capacity is implementation defined
cout << "vec: size: " << vec.size() << " capacity: " << vec.capacity() << endl;
//give ivec 24 elements
for(vector<int>::size_type ix = 0; ix !=24; ++ix) {
  vec.push_back(ix);
}
//size should be 24; capacity will be >=24 and is implementation defined
cout << "vec: size: " << vec.size() << " capacity: " << vec.capacity() << endl;
```

When run on my machine, it produces the following output:

```
ivec: size: 0 capacity: 0
ivec: size: 24 capacity: 32
```

We can now reserve some additional space:

```
vec.reserve(50); //sets capacity to at least 50; may be more
```

Here, the output indicates that the call to reserve allocated exactly as much space as we wanted.

```
ivec: size: 24 capacity: 50
```

We can call `shrink_to_fit` to ask that memory beyond what is needed for the current size be returned to the system:

```
vec.shrink_to_fit(); //ask for the memory to be returned
//size should be unchanged; capacity is implementation defined
cout << "vec: size: " << vec.size() << " capacity: " << vec.capacity() << endl;
```

Calling `shrink_to_fit` is only a request, there is no guarantee that the library will return the memory.

Note: Each vector implementation can choose its own allocation strategy. However, it must not allocate new memory until it is forced to do so.

Exercise

Exercise 9.37: Why don't `list` or `array` have a `capacity` member?

`list` is stored seperately in the memory, there is no need to store the capacity of a list. If it needs to insert another element, it just allocates a new node and links the pecessor of the end note to the new note.

`array` has fixed size. Thus, it has no need to be reallocated and no need for capacity.

Exercise 9.38: Write a program to explore how `vector`s grow in the library you use.

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

int main() {
  std::vector<int> vi;
  std::cout << "Size: " << vi.size()
            << "\tCapacity : " << vi.capacity() << std::endl;

  vi.push_back(1);
  std::cout << "Size: " << vi.size()
            << "\tCapacity : " << vi.capacity() << std::endl;

  for (std::vector<int>::size_type ix = 0; ix != 100; ++ix)
    vi.push_back(ix);
```

```
  std::cout << "Size: " << vi.size()
            << "\tCapacity : " << vi.capacity() << std::endl;

  vi.shrink_to_fit();
  std::cout << "Size: " << vi.size()
            << "\tCapacity : " << vi.capacity() << std::endl;

  return 0;
}
```

Exercise 9.39: Explain what the following program fragment does:

Click here to view code image

```
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
        svec.push_back(word);
svec.resize(svec.size()+svec.size()/2);
```

If the capacity is greater than the argument passed in `resize()`, nothing happens.

If the capacity is less, the capacity will be expanded to at least as much as the given argument.