【C++】Day50

Class	C++
Date	@February 2, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

12.1.1 The shared_ptr Class

shared_ptrs Automatically Destroy Their Objects

When the last `shared_ptr` pointing to an object is destroyed, the `shared_ptr` dass automatically destroy the object to which that `shared_ptr` points.

It does so through another special member function known as a destructor. Analogous to its constructors, each class has a destructor. Just as a constructor controls initialization, the destructor controls what happens when objects of that class type are destroyed.

The destructor for `shared_ptr` decrements the reference count of the object to which that `shared_ptr` points. If the count goes to zero, the `shared_ptr` destructor destroys the object to which the `shared_ptr` points and frees the memory used by that object.

shared_ptrs Automatically Free the Associated Memory

The fact that the `shared_ptr` class automatically frees dynamic objects when they are no longer needed makes it fairly easy to use dynamic memory.

For example, we might have a function that returns a `shared_ptr` to a dynamically allocated object of a type named Foo that can be initialized by an argument of type T:

```
//factory returns a shared_ptr pointing to a dynamically allocated object.
shared_ptr<Foo> factory(T args) {
  //shared_ptr will take care of deleting this memory
  return make_shared<Foo>(args)
}
```

Because factory returns a `shared_ptr`, we can be sure that the object allocated by factory will be freed when appropriate.

For example, the following function stores the `shared_ptr` returned by factory in a local variable.

```
void use_factory(T arg) {
  shared_ptr<Foo> p = factory(arg);
} //p goes out of scope; the memory to which p points is automatically freed
```

Because p is local to `use_factory`, it is destroyed when `use_factory` ends.

When p is destroyed, its reference count is decremented and checked. In this case, p is the only object referring to the memory returned by factory. Because p is about to go away, the object to which p points will be destroyed and the memory in which that object resides will be freed.

Note: If we put `shared_ptrs` in a container, and we subsequently need to use some, but not all, of the elements, remember to erase the elements we no longer need.

Classes with Resources That Have Dynamic Lifetime

Programs tend to use dynamic memory for one of three purposes:

1. They don't know how many objects they'll need
2. They don't know the precise type of the objects they need
3. They want to share data between several objects

Defining the StrBlob Class

Let's now define a version of out class that can manage strings. We'll name this version of our class `StrBlob`.

```
class StrBlob {
private:
  typedef vector<string>::size_type size_type;
  StrBlob();
  StrBlob(std::initializer_list<string> il);
  size_type size() const { return data->size_type; }
  bool empty() const { return data->empty(); }

  //add and remove elements
  void push_back(const string &t) { data->push_back(t); }
  void pop_back();

  //element access
  string &front();
  string &back();

public:
  shared_ptr<vector<string>> data;
  void check(size_type i, const std::string &msg) const;
};
```

StrBlob Constructors

Each constructor uses its constructor initializer list to initialize its data member to point to a dynamically allocated vector. The default constructor allocates an empty vector:

```
StrBlob() : data(make_shared<vector<string>>()) {}
StrBlobk(std::initializer_list<string> il) : data(make_shared<vector<string>>(il)) {}
```

Element Access Members

```
void StrBlobk:check(size_type i, const string &msg) const {
  if(i >= msg.size())
    throw out_of_range(msg);
}
```

The `pop_back` and element access members first call check. If check succeeds, these members forward their work to the underlying vector operation:

```
string &StrBlob::front() {
  // if the vector is empty, check will throw
 check(0, "front on empty StrBlob"); return data->front();
}
string& StrBlob::back() {
 check(0, "back on empty StrBlob"); return data->back();
}

void StrBlob::pop_back() {
 check(0, "pop_back on empty StrBlob"); data->pop_back();
}
```

Exercise

Exercise 12.2: Write your own version of the `StrBlob` class including the `const` versions of `front` and `back`.

See 12_2.cpp for code