【C++】Day62

Class	C++
Date	@February 24, 2022
Material	
Series Number	
Summary	

【Ch13】Copy Control

13.6.2 Move Constructor and Move Assignment

To enable move operations for our own types, we define a move constructor and a move-assignment operator. Those members are similar to the corresponding copy operations, but they “steal” resources from their given object rather than copy them.

Like the copy constructor, the move constructor has an initial parameter that is a reference to the class type. Differently from the copy constructor, the reference parameter in the move constructor is an rvalue reference.

In addition to moving resources, the move constructor must ensure that the moved-from object is left in a state such that destroying that object will be harmless.

As an example, we’ll define the `StrVec` move constructor to move rather than copy the elements from one `StrVec` to another:

```
StrVec::StrVec(StrVec &&s) noexcept // move won't throw any exceptions
  : elements(s.elements), first_free(s.first_free), cap(s.cap) {
  // leave s in a state in which it is safe to run the destructor
  s.elements = s.first_free = s.cap = nullptr;
}
```

Move Operations, Library Containers, and Exceptions

Because a move operation executes by “stealing” resources, it ordinarily does not itself allocate any resources.

As a result, move operations ordinarily will not throw any exceptions. When we write a move operation that cannot throw, we should inform the library of that fact.

One way to inform the library is to specify `noexcept` on our constructor. We specify `noexcept` on a function after its parameter list. In a constructor, `noexcept` appears between the parameter list and the `:` that begins the constructor list:

```
class StrVec {
public:
  StrVec(StrVec&&) noexcept; //move constructor

};
StrVec::StrVec(StrVec &&s) noexcept : {}
```

Note: Move constructors and move assignment operators that cannot throw exceptions should be marked as noexcept.

Move-Assignment Operator

As with the move constructor, if our move-assignment operator won’t throw any exceptions, we should make it `noexcept`. Like a copy-assignment operator, a move-assignment operator must guard against self-assignment:

```
StrVec &StrVec::opeator=(StrVec &&rhs) noexcept {
  // direct test for self-assignment
  if(this !- &rhs) {
    free();
    elements = rhs.elements;
    first_free = rhs.first_free;
    cap = rhs.cap;
    rhs.elements = rhs.first_free = rhs.cap = nullptr;
  }
  return *this;
}
```

In this case we check directly whether the this pointer and the address of `rhs` are the same.

If they are, the right- and left-hand operands refer to the same object and there is no work to do.

Otherwise, we free the memory that the left-hand operand had used, and then take over the memory from the given object.

A Moved-from Object Must be Destructible

Moving from an object does not destroy that object: Sometime after the move operation completes, the moved-from object will be destroyed.

Therefore, when we write a move operation, we must ensure that the moved-from object is in a state in which the destructor can be run.

Warning: After a move operation, the "moved-from" object must remain a valid and destructible object but users may make no assumptions about its value.

Synthesized move Operations

The compiler will synthesize a move constructor or a move-assignment operator only if the class doesn't define any of its own copy-control members and if every nonstatic data member of the class can be moved.

The compiler can move members of built-in type. It can also move members of a class type if the member's class has the corresponding move operation:

```
// the compiler will synthesize the move operations for X and hasX
struct X {
  int i;
  string s; //string defines its own move operations
};

struct hasX {
  X mem; // X has synthesized move operations.
};
```