【C++】Day61

Class	C++
Date	@February 23, 2022
Material	
Series Number	
Summary	Rvalue references

【Ch13】Copy Control

13.6 Moving Objects

In some cases, an object is immediately destroyed after it is copied. In those cases, moving, rather than copying, the object can provide a significant performance boost.

Note: The library containers, `string`, and `shared_ptr` classes support move as well as copy. The IO and `unique_ptr` classes can be moved but not copied.

13.6.1 Rvalue References

The new standard introduced a new kind of reference, an rvalue reference. An rvalue reference is a reference that must be bound to an rvalue. An rvalue reference is obtained by using `&&` rather than `&`.

As we'll see, rvalue references have the important property that they may be bound only to an object that is about to be destroyed.

Generally speaking, an lvalue expression refers to an object's identity whereas an value expression refers to an object's value.

Like any reference, an rvalue reference is just another name for an object. We cannot bind regular references-which we'll refer to an lvalue references when we need to

distinguish them from rvluae references-to expressions that require a conversion , to literals, or to expressions that return an rvalue.

```
int i = 42;
int &r = i;
int &&rr = i; //error: cannot bind an rvalue reference to an lvalue
int &r2 = i * 42; //error: i * 42 is an rvalue
const int &r3 = i * 42; //ok: we can bind a reference to const to an rvalue
int &&rr2 = i * 42; //ok: bind rr2 to the reslult of the multiplication.
```

Lvalues Persist; Rvalues Are Ephemeral

Lvalues have persistent state, whereas rvalues are either literals or temporary objects created in the course of evaluating expressions.

Because rvalue references can only be bound to temporaries, we know that

- The referred-to object is about to be destroyed
- There can be no other users of that object.

These facts together mean that code that uses an rvalue reference is free to take over resources from the object to which the reference refers.

Note: Rvalue references refer to objects that are about to be destroyed. Hence, we can "steal" state from an object bound to an rvalue reference.

Variables Are Lvalues

A variable is an expression with one operand and no other operator. Like any other expression, a variable expression ahs the lvalue/rvalue property. Varaible expressions are lvalues.

We cannot bind an rvalue reference to a variable defined as an rvalue reference type:

```
int &&rr1 = 42; //ok: literals are rvalues
int &&rr2 = rr1; //error: the expression rr1 is an lvalue
```

Note: A variable is an lvalue; we cannot directly bind an rvalue reference to a variable even if that varaible was defined as an rvalue reference type.

The Library move Function

Although we cannot directly bind an rvalue reference to an lvalue, we can explicitly cast an lvalue to its corresponding rvalue reference type.

We can also obtain an rvalue reference bound to an lvalue by calling a new library function named `move`, which is defined in the `utility` header.

```
int &&rr3 = std::move(rr1);
```

Calling move tells the compiler that we have an lvalue that we want to treat as if it were an rvalue. It is essential to realize that the call to move promises that we do not intend to use `rr1` again except to assign to it or destroy it. After a call to move, we cannot make any assumptions about the value of the moved-from object.

Note: We can destroy a moved-from object and can assign a new value to it, but we cannot use the value of a moved object.

Warning: Code that uses move should use `std::move`, not `move`. Doing so avoids potential name collisions.

Exercise

Exercise 13.45: Distinguish between an rvalue reference and an lvalue reference.

An rvalue reference must be bound to an rvalue. It refers to an object's value and is usually ephemeral.

An lvalue reference must be bound to an lvaue or a const rvalue. Lvalues are persistent.

Exercise 13.46: Which kind of reference can be bound to the following initializers?

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

```
int &&r1 = f();
int &r2 = vi[0];
int &r3 = r1;
int &&r4 = vi[0] * f();
```