【C++】Day56

Class	C++
Date	@February 15, 2022
Material	
Series Number	
Summary	Copy Constructor

【Ch13】Copy Control

In this chapter, we'll learn how classes can control what happens when objects of the class type are copied, assigned, moved, or destroyed. Classes control these actions through special member functions: the copy constructor, move constructor, copy-assignment operator, move-assignment operator, and destructor.

- The copy and move constructors define what happens when an object is initialized from another object of the same type.
- The copy- and move-assignment operators define what happens when we assign an object of a class type to another object of the same class type.
- The destructor defines what happens when an object of the type ceases to exist.

13.1 Copy, Assign, and Destroy

13.1.1 The Copy Constructor

A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values:

```
class Foo {
public:
  Foo();
  Foo(const Foo&); //copy constructor
};
```

The first parameter must be a reference type. That parameter is almost always a reference to `const`.

The Synthesized Copy Constructor

When we do not define a copy constructor for a class, the compiler synthesizes one for us. Unlike the synthesized default constructor, a copy constructor is synthesized even if we define other constructors.

As we will see later, the synthesized copy constructor for some classes prevents us from copying objects of that class type.

Otherwise, the synthesized copy constructor memberwise copies the members of its argument into the object being created.

As an example, the synthesized copy constructor for our `Sales_data` class is equivalent to:

```
class Sales_data {
public:
  // other members and constructors as before
  // declaration equivalent to the synthesized copy constructor
  Sales_data(const Sales_data&);
private:
  string bookNo;
  int units_sold = 0;
  double revenue = 0.0;
};

Sales_data::Sales_data(const Sales_data &org) :: bookNo(org.bookNo), units_sold(org.units_sold), revenue(org.revenue) {};
```

Copy Initialization

When we use direct initialization, we are asking the compiler to use ordinary function matching to select the constructor that best matches the arguments we provide.

When we use copy initialization, we are asking the compiler to copy the right-hand operand into the object being created, converting that operand if necessary.

Copy initialization ordinarily uses the copy constructor. However, as we'll see later, if a class has a move constructor, then copy initialization sometimes uses the move constructor instead of the copy constructor.

Copy initialization happens not only when we define variables using an `=`, but also when we

- Pass an object as an argument to a parameter of nonreference type.
- Return an object from a function that has a nonreference return type.
- Brace initialize the elements in an array or the members or an aggregate class.

Constraints on Copy Initialization

Whether we use copy or direct initialization matters if we use an initializer that requires conversion by an explicit constructor:

```
vector<int> v1(10); // ok: direct initialization
vector<int> v2 = 10; //error: constructor that takes a size is explicit
void f(vector<int>); //f's parameter is copy initializated
f(10); //error: can't use an explicit constructor to copy an argument
f(vector<int>(10); //ok: directly construct a temporary vector from an int
```

The Compiler Can Bypass the Copy Constructor

During copy initialization, the compiler is permitted to skip the copy/move constructor and create the object directly. That is, the compiler is permitted to rewrite

```
string null_book_author = "Arthur";
```

into

```
string null_book_author("Arthur");
```

Exercise

Exercise 13.5: Given the following sketch of a class, write a copy constructor that copies all the members. Your constructor should dynamically allocate a new `string` (§ 12.1.2, p. 458) and copy the object to which `ps` points, rather than copying `ps` itself.

Click here to view code image

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int    i;
};
```

See 13_5.cpp for code