【C++】Day56(2)

Class	C++
Date	@February 15, 2022
Material	
Series Number	
Summary	Copy Assignment Operator and Destructor

【Ch13】Copy Control

13.1.2 The Copy-Assignment Operator

Just as a class controls how objects of that class are initialized, it also controls how objects of its class are assigned.

```
Sales_data trans, accum;
trans = accum; // uses the Sales_data copy-assignment operator
```

As with the copy constructor, the compiler synthesizes a copy-assignment operator if the class does not defines its own.

Introducing Overloaded Assignments

Overloaded operators are functions that have the name `operator` followed by the symbol for the operator being defined. Hence, the assignment operator is a function named `operator=`. Like any other function, an operator function has a return type and a parameter list.

When an operator is a member function, the left-hand operand is bound to the implicit `this` parameter. The right-hand operand in a binary operator, such as assignment, is passed as an explicit parameter.

The copy-assignment operator takes an argument of the same type as the class:

```
class Foo {
public:
  Foo& operator=(const Foo&); //assignment operator
};
```

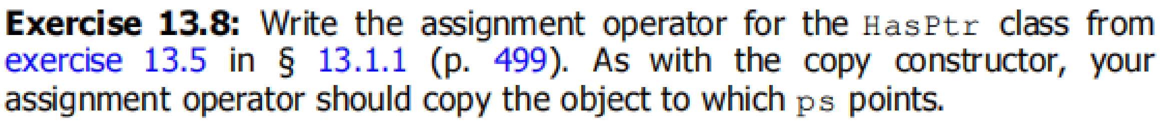
Best Practices: Assignment operators ordinarily should return a reference to their left-hand operand.

The Synthesized Copy-Assignment Operator

The following code is equivalent to the synthesized `Sales_data` copy-assignment operator:

```
// equivalent to the synthesized copy-assignment operator
Slaes_data& Sales_data::operator=(const Sales_data &rhs) {
  bookNo = rhs.bookNo; //calls the string::operator=
  units_sold = rhs.units_sold; //uses the built-in int assignment
  revenue = rhs.revenue; //uses the built-in double assignment
  return *this; //return a reference to this object.
}
```

Exercise



Exercise 13.8: Write the assignment operator for the `HasPtr` class from exercise 13.5 in § 13.1.1 (p. 499). As with the copy constructor, your assignment operator should copy the object to which `ps` points.

See 13_8.cpp

13.1.3 The Destructor

The destructor operates inversely to the constructors: Constructors initialize the `nonstatic` data members of an object and may do other work; destructors do whatever work is needed to free the resources used by an object and destroy the `nonstatic` data members of the object.

The destructor is a member function with the name of the class prefixed by a tilde(~). It has no return value and takes no parameters:

```
class Foo {
public:
  ~Foo(); //destructor
};
```

What a Destructor Does

Just as a constructor has an initialization part and a function body, a destructor has a function body and a destruction part.

In a constructor, members are initialized before the function body is executed, and members are initialized in the same order as they appear in the class.

In a destructor, the function body is executed first and then the members are destroyed. Members are destroyed in reverse order from the order in which they were initialized.

What happens when a member is destroyed depends on the type of the member. Members of class type are destroyed by running the member's own destructor. The built-in types do not have destructors, so nothing is done to destroy members of built-in type.

Note: The implicit destruction of a member of built-in pointer type does not delete the object to which that pointer points.

When a Destructor is Called.

The destructor is used automatically whenever an object of its type is destroyed:

- Variables are destroyed when they go out of scope
- Members of an object are destroyed when the object of which they are a part is destroyed
- Elements in a container-whether a library container or an array-are destroyed when the container is destroyed.

- Dynamically allocated objects are destroyed when the `delete` operator is applied to a pointer to the object.

Note: The destructor is not run when a reference or a pointer to an object goes out of scope.