【C++】Day20(2)

Class	C++
Date	@December 10, 2021
Material	
Series Number	
Summary	

【Ch7】Classes

The fundamental ideas behind classes are data abstraction and encapsulation.

Data abstraction is a programming (and design) technique that relies on the separation of interface(the usually `public` operations supported by a type) and implementation(the usually `private` members of a class that define the data and any operations that are not intended for use by code that uses the type).

Encapsulation enforces the separation of a class' interface and implementation.

A class that is encapsulated hides its implementation-users of the class can use the interface but have no access to the implementation.

7.1 Defining Abstract Data Types

7.1.2 Designing the Sales_data Class

The interface to `Sales_data` consists of the following oeprations:

- An `isbn` member function to return the object's ISBN
- A `combine` member function to add one Sales_data object into another
- A function named `add` to add two Sales_data objects
- A `read` function to read data from an istream into a Sales_data object

- A `print` function to print the value of A Sales_data object on an ostream

```
struct Sales_data {
  //new members: operations on Sales_data objects
  std::string isbn() const { return bookNo; }
  Sales_data& combine(const Sales_data&);
  double avg_price() const;

  std::string bookNo;
  unsigned units_sold = 0;
  double revenue = 0.0;
};
```

Note: Functions defined in the class are implicitly inlnie.

Defining Member Functions

Although every member must be declared inside its class, we can define a member functions' body either inside or outside of the class body.

In `Sales_data`, `isbn()` is defined inside the class; `combine` and `avg_price` will be defined elsewhere.

Introducint this

Let's look again at a call to the `isbn` member function:

```
total.isbn()
```

Here we use the dot operator to fetch the `isbn` member of the object named `total,` which we then call.

When we call a member function we do so on behalf of an object. When `isbn` refers to members of `Sales_data`, it is referring implicitly to the members of the object on which the function was called. In this call, when `isbn` returns `bookNo`, it is implicitly returning `total.bookNo`.

Member functions access the object on which they were called through an extra, implicit parameter names `this`. When we call a member function, this is initialized with the address of the object on which the function was invoked.

For example, when we call

```
total.isbn();
```

the compiler passes the address of `total` to the implicit `this` parameter in `isbn`. It is as if the compiler rewrites this call as

```
Sales_data::isbn(&total)
```

Inside a member function, we can refer directly to the members of the object on which the function was called. We do not have to use a member access operator to use the members of the object to which this points.

Any direct use of a member of the class is assumed to be an implicit reference through this. That is, when `isbn` uses `bookNo`, it is implicitly using the member to which this points. It is as if we had written `this->bookNo`.

The `this` parameter is defined for us implicitly. Indeed, it its illegal for us to define a parameter or variable named this.

Inside the body of a member function, we can use this.

It would be legal, although unnecessary, to define `isbn` as:

```
std::string isbn() { return this->bookNo; }
```

Because `this` is intended to always refer to "this" object, this is a const pointer. We cannot change the address that `this` holds.

Introducing const Member Functions

The purpose of the `const` that follows the parameter list is to modify the type of the implicit `this` pointer.

By default, the type of this is a const pointer to the nonconst version of the class type. For example, by default, the type of `this` in a `Sales_data` member function is `Sales_data *const`. Although this is implicit, it follows the normal initialization rules, which means that (by default) we cannot bind this to a const object. This fact, in turn, means that we cannot call an ordinary member function on a const object.

Our function would be more flexible if this were a pointer to const.

However, this is implicit and does not appear in the parameter list. There is no place to indicate that this should be a pointer to const. The language resolves this problem by letting us put const after the parameter list of a member function.

A const following the parameter list indicates that `this` is a pointer to const. Member functions that use const in this way are const member functions.

We can think of the body of isbn as if were written as:

```
std::string Sales_data::isbn(const Sales_data *const this) { return this-> isbn; }
```

Note: Objects that are const, and references or pointers to const object, may call only const member functions.