【C++】Day23

Class	C++
Date	@December 13, 2021
Material	
Series Number	
Summary	

【Ch7】Classes

7.1.4 Constructors

Classes control object initialization by defining one or more special member functions known as constructors.

The job of a constructor is to initialize the data members of a class object. A constructor is run whenever an object of a class type is created.

Constructors have a (possibly empty) parameter list and a (possibly empty) function body. A class can have multiple constructors.

Like any other overloaded function, the constructors must differ from each other in the number or types or their parameters.

Unlike other member functions, constructors may not be declared as `const`. When we create a `const` object of a class type, the object does not assume its constness until after the constructor completes the object's initialization. Thus, constructors can write to const objects during their construction.

For our Sales_data class, we did not define any constructors. *How are the following variable initialized?*

```
Sales_data total; //variable to hold the running sum.
```

We did not supply an initializer for these objects, so we know that they are default initialized. Classes control default initialization by defining a special constructor, known as the default constructor. The default constructor is one that takes no arguments.

If our class does not explicitly define any constructors, the compiler will implicitly define the default constructor for us.

The compiler-generated constructor is known as the synthesized default constructor. For most classes, this synthesized constructor initializes each data member of the class as follows:

- If there is an in-class initializer, use it to initialize the member.
- Otherwise, default-initialize the member.

Some Classes Cannot Rely on the Synthesized Default Constructor

Only fairly simple classes can rely on the synthesized default constructor.

The most common reason that a class must define its own default constructor is that the compiler generates the default for us only if we do not define any other constructors for the class.

Note: If we define any constructor, the class will not have a default constructor.

A second reason to define the default constructor is that for some classes, the synthesized default constructor does the wrong thing. Remember that objects of built-in or compound type(such as arrays and pointers) that are defined inside a block have undefined value when they are default initialized.

Therefore, classes that have members of built-in or compound type should ordinarily either initialize those members inside the class or define their own version of the default constructor.

Warning: Classes that have members of built-in or compound type usually should rely on the synthesized default constructor only if all such members have in-class initializers.

A third reason that some class must define their own default constructor is that sometimes the compiler is unable to synthesize one.

For example, if a class has a member that has a class type, and that class doesn't have a default constructor, then the compiler can't initialize that member. For such classes, we must define our own version of the default constructor.

Defining the Sales_data Constructor

```
struct Sales_data{
  //constructors
  Sales_data() = default;
  Sales_data(const string &s) : bookNo(s) {}
  Sales_data(const string &s, unsigned n, double p) : bookNo(s), units_sold(n), revenue(n * p) {}
  Sales_data(istream &);

  //members as before
  string isbn() const { return bookNo; }
  Sales_data& combine(const Sales_data&);
  double avg_price() const;
  string bookNo;
  unsigned units_sold = 0;
  double revenue = 0.0;
};
```

Start by the default constructor:

```
Sales_data() = default;
```

We are defining this constructor only because we want to provide other constructors as well as the default constructor. We want this constructor to do exactly the same work as the synthesized version we had been using.

If we want the default behaviour, we can ask the compiler to generate the constructor for us by writing `= default`

after the parameter list.

Warning: The default constructor works for Sales_data only because we provide initializers for the data members with built-in type.

Constructor Initializer List

Now lets look at two other constructors:

```
Sales_data(const string &s) : bookNo(s) {}
Sales_data(const string &s, unsigned n, double p) : bookNo(s), units_sold(n), revenue(n * p) {}
```

The new parts in these definitions are the colon and the code between it and the curly braces that define the (empty) function bodies.

This new part is a constructor initializer list, which specifies initial value for one or more data members of the object being created.

The constructor initializer is a list of member names, each of which is followed by that member's initial value in parentheses (or inside curly braces).

It is usually best for a constructor to use an in-class initializer if one exists and gives the member the correct value.

Best Practices: Constructors should not override in-class initializers except to use a different initial value. If you can't use in-class initializers, each constructor should explicitly initialize every member of built-in type.

It is nothing that both constructors have empty function bodies. The only work these constructors need to do is given the data members their values. If there is no further work, then the function body is empty.

Defining a Constructor outside the Class Body

Unlike other constructors, the constructor that takes an `istream` does have work to do. Inside its function body, this constructor calls read to give the data members new values:

```
Sales_data::Sales_data(std::istream &is) {
  read(is, *this); //read will read a transcaction from is into this object
}
```

As with other member function, when we define a constructor outside of the class body, we must specify the class of which the constructor is a member. Thus, `Sales_data::Sales_data` says that we're defining the `Sales_data` member named `Sales_data`.

In this constructor, the constructor initializer list is empty. Even though the constructor initializer list is empty, the members of this object are still initialized before the constructor body is executed.

Members that do not appear in the constructor initializer list are initialized by the corresponding in-class initializer or are default initialized.

`bookNo` will be the empty string, and `units_sold` and `revenue` will both be 0.

Exercise

Exercise 7.11: Add constructors to your `Sales_data` class and write a program to use each of the constructors.

Exercise 7.14: Write a version of the default constructor that explicitly initializes the members to the values we have provided as in-class initializers.

```
struct Sales_data {
  Sales_data() {
    bookNo = "";
    units_sold = 0;
    revenue = 0.0;
  }
  Sales_data(const string &s, unsigned sNum, double price) : bookNo(s), units_sold(sNum), revenue(sNum * price) {}
  Sales_data(const string &s) : bookNo(s) {}
  Sales_data(istream &is);

  string isbn() const { return bookNo; }
  Sales_data& combine(const Sales_data&);
  double avg_price() const;
  string bookNo;
  unsigned units_sold = 0;
  double revenue = 0.0;
};

void read(istream &is, Sales_data *data) {
  is >> data->bookNo >> data->units_sold >> data->revenue;
}

Sales_data::Sales_data(std::istream &is) {
  read(is, this);
}
```