# 【C++】Day26

| | |
|---|---|
| Class | C++ |
| Date | @December 28, 2021 |
| Material | |
| Series Number | |
| Summary | Delegate Constructor and Default Constructor |

## 【Ch7】Classes

### 7.5.2 Delegating Constructors

A delegating constructor uses another constructor from its own class to perform its initialization. It is said to “delegate” some (or all) of its work to this other constructor.

Like any other constructor, a delegating constructor has a member initializer list and a function body. In a delegating constructor, the member initializer list has a single entry that is the name of the class itself. The name of the class is followed by a parenthesized list of arguments.

See the following `Sales_data` class for an example:

```
class Sales_data {
  //nondelegating constructor initializes members from corresponding arguments
  Sales_data(string s, unsigned cnt, double price) : bookNo(s), units_sold(cnt), revenue(cnt * price) {}

  //remaining constructors all delegate to another constructor
  Sales_data() : Sales_data("", 0, 0) {}
  Sales_data(string s) : Sales_data(s, 0, 0) {}
  Sales_data(istream &is) : Sales_data() { read(is, *this); }
};
```

### 7.5.2 The Role of the Default Constructor

The default constructor is used automatically whenever an object is default or value initialized.

Default initialization happens when:

- When we define nonstatic variables or arrays at block scope without initializers
- When a class that itself has members of class type uses the synthesized default constructor
- When members of class type are not explicitly initialized in a constructor initializer list

See the following for an example:

```
class NoDefault {
public:
  NoDefault(const string&);
};

struct A {
  NoDefarult my_mem;
};
A a; //error: cannot synthesize a constructor for A

struct B {
  B() {} //error: no initilaizer for b_member
  NoDefault b_member;
};
```

*Best Practices: It is almost always right to provide a default constructor if other constructors are being defined*

## *Using the Default Constructor*

The following declaration of obj compiles without complaint. However, when we try to use obj:

```
Sales_data obj(); //ok: but defines a function, not an object
if(obj.isbn() == Primer.isbn()) //error: obj is a function
```

The problem is that, although we intended to declare a default-initialized object, `obj` actually declares a function taking no parameters and returning an object of type `Sales_data`.

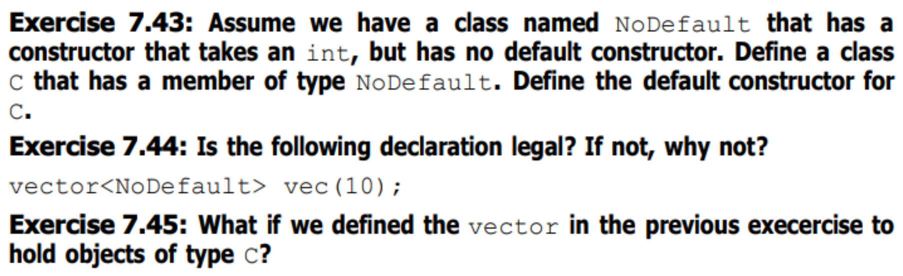
The correct way to define an object that uses the default constructor for initialization is to leave off the trailing, empty parentheses:

```
//ok: obj is a default-initialized object
Sales_data obj;
```

*Warning: It is a common mistake to try to declare an object initialized with the default constructor as follows:*

```
Sales_data obj1(); //error: Declares a function, not an object
Sales_data obj2; //ok: obj2 is an object, not a function
```

*Exercise*



**Exercise 7.43: Assume we have a class named `NoDefault` that has a constructor that takes an `int`, but has no default constructor. Define a class `C` that has a member of type `NoDefault`. Define the default constructor for `C`.**

**Exercise 7.44: Is the following declaration legal? If not, why not?**

```
vector<NoDefault> vec(10);
```

**Exercise 7.45: What if we defined the `vector` in the previous execercise to hold objects of type `C`?**

```
class NoDefault {
public:
  NoDefault(int n) {}
};

class C {
public:
  C() : mem(NoDefault(0)) {}
  NoDefault mem;
};

//7.44: No, because NoDefault doesn't provide a default constructor
//7.45: Yes, C has a default constructor
```