

# 【C++】 Day26(2)

▼ Class	C++
📅 Date	@December 28, 2021
🔗 Material	
# Series Number	
☰ Summary	Implicit Class-Type Conversion

## 【Ch7】 Classes

### 7.5.4 Implicit Class-Type Conversions

Every constructor that can be called **with a single argument** defines **an implicit conversion to a class type**. Such constructors are sometimes referred to as **converting constructors**.

*Note: A constructor that can be called with a single argument defines an implicit conversion from the constructor's parameter type to the class type.*

The `Sales_data` constructors that take a string and that take an `istream` **both define implicit conversions** from those types to `Sales_data`. That is, we can use a string or an `istream` where an object of type `Sales_Data` is expected.

```
string null_book = "9-999";
//constructs a temporary Sales_data object
//with units_sold and revenue equal to 0 and bookNo equal to null_book
item.combine(null_book);
```

Here we call the `Sales_data combine` member function with a string argument. This call is perfectly legal; the compiler **automatically creates a `Sales_data` object** from the given string. That newly generated(temporary) `Sales_data` is passed to `combine`.

#### *Only One Class-Type Conversion Is Allowed*

The compiler will automatically apply only one class-type conversion.

**For example, the following code is in error because it implicitly uses two conversions:**

```
//error: requires two user-defined conversions
// (1)convert "9-999" to string
// (2)convert that(temporary) string to Sales_data
item.combine("9-999");
```

If we wanted to make this call, we can do so by explicitly converting the character string to either a string or a `Sales_data` object:

```
//ok: explicit conversion to string, implicit conversion to Sales_data
item.combine(string("9-999"));
//ok: implicit conversion to string, explicit conversion to Sales_data
item.combine(Sales_data("9-999"));
```

### *Suppressing Implicit Conversions Defined by Constructors*

We can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor as `explicit`:

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

Now, neither constructor can be used to implicitly create a `Sales_data` object. Neither of our previous uses will compile:

```
item.combine(null_book); //error: string constructor is explicit
item.combine(cin); //error: istream constructor is explicit
```

The `explicit` keyword is meaningful only on constructor that can be called with a single argument. Constructors that require more arguments are not used to perform an implicit conversion, so there is no need to designate such constructors as explicit.

The `explicit` keyword is used only on the constructor declaration inside the class. It is not repeated on a definition made outside the class body:

```
//error
explicit Sales_data::Sales_data(istream &is) {
    read(is, *this);
}
```

### *explicit Constructors Can Be Used Only for Direct Initialization*

When we use the copy form of initialization(with an =), implicit conversions happen. We cannot use an explicit constructor with this form of initialization; we must use direct initialization:

```
Sales_data item1(null_book); //ok: direct initialization
Sales_data item2 = null_book; //error: cannot use the copy form of initialization with an explicit constructor
```

*Note: When a constructor is declared explicit, it can be used only with the direct form of initialization. Moreover, the compiler will not use this constructor in an automatic conversion.*

### Explicitly Using Constructors for Conversion

We can use explicit constructors explicitly to force a conversion

```
//ok: the argument is an explicitly constructed Sales_data object
item.combine(Sales_data(null_book));
//ok:static_cast use an explicit constructor
item.combine(static_cast<Sales_data>(cin))
```

## 7.5.5 Aggregate Classes

An **aggregate class** gives users **direct access to its members** and has **special initialization syntax**. A class is an aggregate if

- All of its data members are public
- It does not define any constructors
- It has no in-class initializers
- It has no base classes or virtual functions

For example, the following class is an aggregate:

```
struct Data {
    int ival;
    string s;
};
```

We can initialize the data members of an aggregate class by providing a braces list of member initializers:

```
//val.ival = 0; val.s = string("Anna");
Data val = { 0, "Anna" };
```

If the list of initializers has fewer elements than the class has members, **the trailing members are value initialized**. The list of initializers must not contain more elements than the class has members.

There are three significant drawbacks to explicitly initializing the members of an object of class type:

- It requires that all the data members of the class be public
- It puts the burden on the user of the class to initialize all of the members correctly
- If a member is added or removed, all initializations have to be updated.