# 【C++】Day seven(3)

| | |
|---|---|
| Class | C++ |
| Date | @November 25, 2021 |
| Material | |
| Series Number | |
| Summary | |

## 【Ch2】Define Own Data Structure

In C++, we define our own data types by defining a class.

### 2.6.1 Defining the Sales_data Type

We will define our class as follows:

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

Our class begins with the keyword `struct`, followed by the name of the class and a (possibly empty) class body. The class body is surrounded by curly braces and forms a new scope.

The close curly that ends the class body must be followed by a semicolon. The semicolon is needed because we can define variables after the class body.

The two ways of defining Sales_data variables are equivalent:

```
struct Sales_data {} accum, trans, *salesptr;
```

```
//Equivalent, but better way to define these variables
struct Sales_data {};
Sales_data accum, trans, *salesptr;
```

The semicolon marks the end of the (usually empty) list of declarators. Ordinarily, it is a bad idea to define an object as part of a class definition. Doing so obscures the code by combining the definitions of two different entities-the class and a variable-in a single statement.

### *Class Data Members*

The class body defines the members of the class. Our class has only data members. The data members of a class define the contents of the objects of that class type. Modifying the data members of one object does not change the data in any other `Sales_data` object.

Under the new standard, we can supply an in-class initializer for a data member. When we create objects, the in-class initializers will be used to initialize the data members.

```
struct my_data {
  int a = 3;
  std::string b;
}
```

Members without an initializer are default initialized. In this case, b would be set to the empty string.

## 2.6.3 Writing Our Own Header Files

Although we can define a class inside a function, such classes have limited functionality. As a result, classes ordinarily are not defined inside functions. When we define a class outside of a function, there may be only one definition of that class in any given source file. In addition, if we use a class in several different files, the class' definition must be the same in each file.

In order to ensure that the class definition is the same in each file, classes are usually defined in header files. Typically, classes are stored in headers whose name derives from the name of the class.

### *A Brief Introduction to The Preprocessor*

The most common technique for making it safe to include a header multiple times relies on the preprocessor. The preprocessor is a program that runs before the compiler and changes the source text of our programs.

Our programs already rely on one preprocessor facility, `#include`. When the preprocessor sees a `#include`, it replaces the `#include` with the contents of the specified header.

C++ programs also use the preprocessor to define header guards. header guards rely on preprocessor variables. Preprocessor variables have one of two possible states: defined or not defined. The `#define` directive takes a name and defiens that name as a preprocessor variable. There are two other directive that test whether a given preprocessor variable has or has not been defined: `#ifdef` is true if the variable has been defined, and `#ifndef` is true if the variable has not been defined. If the test is true, the neverything following the `#ifdef` or `#ifndef` is processed up to the matching `#endif.`

See the following code for an example:

```
#ifndet SALES_DATA_H
#define SALES_DATA_h
#include <string>
struct Sales_data {/*...*/};
#endif
```

The first time `Sales_data.h` is included, the `#ifndef` test will succeed. The prerprocessor will process the lines following `#ifndef` up to the `#endif`. As a reulst, the preprocessor variable `SALES_DATA_H` will be defined and the contents of `Sales_data.h` will be copied into our program. If we include `Sales_data.h` later on in the same file, the #`ifndef` directive will be false. The lines between it and the `#endif` directive will be ignroed.

Preprocessor variables, including names of header guards, must be unique throughout the program. Typically we ensure uniqueness by basing the guard's name on the name of a class in the header. To avoid name clashes with other entities in our prograsm, preprocessor variables usually are written in all uppercase.

*Note: Heaers should have guards, even if they aren't(ye) included by another header. Header guards are trivial to write, and by habitually defining them you don't need to decide whether they are needed.*