Defining Our Own Data Structures

Overview

A data structure is a way to group together related data elements and a strategy for using those data. In C++ we define our own data types by defining a class. The library types `string`, `istream`, and `ostream` are all defined as classes. C++ support for classes is extensive.

Class Data Members

To create a custom class by using the keyword `struct` followed by the name of the class and a class body. The class body is surrounded by curly braces and forms a new scope. The names defined inside the class must be unique within the class but can reuse named defined outside the class.

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

The semicolon marks the end of the list of declarators. 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.

Warning

It is a common mistake among new programmers to forget the semicolon at the end of a class definition.

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. Each object has its own copy of the class data members. Modifying the data members of one object does not change the data in other `Sales_data` object.

Writing Our Own Header Files

Classes ordinarily are not defined inside functions. When we define a class outside of a function, there may be only one definition of the class in any given source file. 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. For example, the`string` library type is define in the `string` header.

Note

Whenever a header is updated, the source file that use the header must be recompiled to get new or change declarations.

A Brief Introduction to the Preprocessor

The preprocessor is a program that runs before the compiler and change the source text of the program. Our program 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 states: defined or not defined. The `#define` directive takes a name and defines that name as a preprocessor variable. There are two other directives 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, then everything following the `#ifdef` or `#ifndef` is processed up to the matching `#endif`.

We can use these facilities to guard against multiple inclusion as follows

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>

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

⚠ **Warning**

Preprocessor variable named do not respect C++ scoping rules.

Preprocessors must be unique throughout the program. To avoid clashes with other entities in our program, preprocessors variables usually are written in all uppercase.

Defined Terms

`#define`

- Preprocessor directive that defines a preprocessor variable

`#endif`

- Preprocessor directive that ends an `#ifdef` or `#ifndef` region.

`#ifdef`

- Preprocessor directive that determines whether a given variable is defined.

`#ifndef`

- Preprocessor directive that determines whether a given variable is not defined.

preprocessor

- Program that runs as part of compilation of a C++ program.

preprocessor variable

- Variable managed by the preprocessor. The preprocessor replaces each preprocessor variable by its value before our program is compiled.