

Chapter 7: Classes (A First Look)

(ppt)

Overview

In C++ we use **classes** to define **our own data types**. By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier to write, debug, and modify.

This chapter continues the coverage of classes begun in Chapter 2 (`struct`). Here we will focus on the importance of **data abstraction**, which lets us **separate** the implementation of an object from the operations that that object can perform. In Chapter 13 we'll learn how to control what happens when objects are copied, moved, assigned, or destroyed. In Chapter 14 we'll learn how to define our own operators.

Fundamental Ideas Behind Classes and Abstract Data Types (ADT)

The fundamental ideas behind classes are data abstraction and encapsulation. **Data abstraction** means representation of information in terms of its interfaces (member or non-member functions) with the user (e.g., `push_back` in `vector`). **Encapsulation** means hiding details of implementation (e.g., internal memory management in `vector`).

A class that uses data abstraction and encapsulation is an **abstract data type**. In an abstract data type, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. **They can instead think abstractly about what the type does.**

Q: Does the `Student` data type listed below an **abstract** data type (ADT)? Recall that an ADT must have well-defined interfaces (member or non-member functions) so that programmers who use the class need not know how the type works. **They can instead think abstractly about what the type does.**

```
struct Student
{
    std::string name;
    int id;
    int age;
};
```

A:

(see ppt on Different Kinds of Programming Role in C++)

7.1 Defining Abstract Data Types

7.1.1 Designing the Sales_data Class

Recall what we have on Sales_data class (2.6.1, pp. 72):

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

We will add operations for data abstraction. The interface to Sales_data consists of the following operations:

- An isbn **member function** to return the object's ISBN
- A combine **member function** to add one Sales_data object into another
- An add function to add two Sales_data objects
- A read function to read data from an istream into a Sales_data object
- A print function to print the value of a Sales_data object on an ostream.

And the users might use these interfaces like the following:

```
Sales_data total; // variable to hold the running sum
if (read(cin, total)) { // read the first transaction
    Sales_data trans; // variable to hold data for the next transaction
    while (read(cin, trans)) { // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check the isbn
            total.combine(trans); // update the running total
        else {
            print(cout, total) << endl; // print the results
            total = trans; // process the next book
        }
    }
    print(cout, total) << endl; // print the last transaction
} else { // there was no input
    cerr << "No data?!" << endl; // notify the user
}
```

Remark: once we learn how to define our own operators (Ch. 14), we can use `Sales_item` as we have done in Ch. 1 and shown below.

```

Sales_item total; // variable to hold the running sum
if (cin >> total) { // read the first transaction
    Sales_item trans; // variable to hold data for the next transaction
    while (cin >> trans) { // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check the isbn
            total += trans; // update the running total
        else {
            cout << total << endl; // print the results
            total = trans; // process the next book
        }
    }
    cout << total << endl; // print the last transaction
} else {
    cerr << "No data?!" << endl; // notify the user
}

```

2 Defining the Revised `Sales_data` Class and Member Functions

As we have seen from the **use** of the abstract data type, the revised `Sales_data` class will have two **member functions**, `combine` and `isbn`. In addition, we'll give `Sales_data` another member function to return the average price at which the books were sold. This function, which we'll name `avg_price`, isn't intended for general use. It will be part of the implementation, not part of the interface.

We define and declare **member functions** similarly to ordinary functions. Member functions **must be declared** inside the class. Member functions may be **defined** inside the class itself or outside the class body. **Nonmember functions** that are part of the interface, such as `add`, `read`, and `print`, are **declared and defined** outside the class.

With the above knowledge, the revised `Sales_data` class is now ready:

Sales_data.h

```

struct Sales_data {
    // new members: operations on Sales_data objects
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    // data members are unchanged from § 2.6.1 (p. 72)
    std::string bookNo;
    unsigned units_sold = 0;
}

```

```

    double revenue = 0.0;
};

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream& print(std::ostream&, const Sales_data&);
std::istream& read(std::istream&, Sales_data&);

```

Defining Member Functions

Member functions may be defined inside the class itself or outside the class body. In general, if the function body consists of more than three lines, we should do it outside the class.

(1) Declare and define member function inside the class definition

Sales_data.h

```

struct Sales_data {
    std::string isbn() const { return bookNo; }
    ...
}

```

(2) Declare member function inside the class definition and define it outside

Sales_data.h

```

struct Sales_data {
    ...
    double avg_price() const;
}

```

Sales_data.cpp

```

double Sales_data::avg_price() const
{ if (units_sold)
    return revenue/units_sold;
else
    return 0;
}

```

(Important) Notice that the member functions defined outside the class definition look like ordinary functions except that they contain the class name and a double colon (::) before the function name (but after the return type). The :: symbol is called the **scope resolution operator**. It is needed to indicate that these are class member functions and to tell the compiler which class they belong to.

Ex 7.1: In-class Coding Exercise

Write a class named `Person` that represents the name and age of a person. Use a `string` and an `unsigned` to hold each of these elements. Provide member functions `setName` and

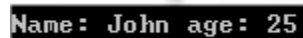
setAge to set proper values of name and age. Provide member functions getName and getAge in your Person class to return the name and age. You should put the class definition in a header file Person.h with a proper header guard and const correctness. A sample client code and output look like:

PersonClient.cpp

```
#include <iostream>
#include "Person.h"

using namespace std;

int main()
{
    Person p;
    p.setName("John");
    p.setAge(25);
    cout << "Name: " << p.getName() << " age: " << p.getAge() << endl;
    return 0;
}
```

A screenshot of a terminal window with a black background and white text. The text displayed is "Name: John age: 25".

A:

Person.h

Knowing Your Objects: The `this` Pointer

We shall introduce an important concept: the `this` pointer, before proceed further. In C++, a special pointer called `this` is implicitly defined to point to the object used to invoke a member function. (Basically, `this` is passed as a hidden argument to the method.)

The `this` pointer is very useful when we will need to **return the object on which the member function was called**.

Let us now look at the `combine` member function to illustrate the concept. The `combine` member function is intended to act like the compound assignment operator, `+=`. The object on which this member function is called represents the left-hand operand of the assignment. The right-hand operand is passed as an explicit argument:

```
| total += trans; // update the running total

| total.combine(trans); // update the running total

| Sales_data& Sales_data::combine(const Sales_data &rhs)
| {
|     units_sold += rhs.units_sold; // add the members of rhs into
|     revenue += rhs.revenue; // the members of ''this'' object
|     return ???; // return the object on which the function was called
| }
```

We will need to **return the object on which the function was called**. The C++ solution to this problem is to use a special pointer called `this`. The `this` pointer points to the object used to invoke a member function. (Basically, `this` is passed as a hidden argument to the method.)

Thus, the function call `total.combine(trans)` sets `this` to the address of the `total` object and makes that pointer available to the `combine` method. In general, all class methods have a `this` pointer set to the address of the object that invokes the method. Indeed, `units_sold` in the `combine` member function is just shorthand notation for `this->units_sold`.

Q: Now complete ??? in the definition of `combine` member function.

A:

What you want to return is not `this` because `this` is the address of the object. You want to return the object itself, and that is symbolized by `*this`. (Recall that applying the dereferencing operator `*` to a pointer yields the value to which the pointer points.)

Example: Quick Check on Concept

Consider the previous exercise on the class `Person`. If we would like to concatenate a sequence of `set` actions into a single expression so we can do:

```
Person p;
p.setName("John").setAge(25);
p.setAge(25).setName("John");
```

This is called chained operations. We can modify `setAge` and `setName` implementation in the `Person` class to accomplish these.

```
void setName(const std::string& s){name = s;}
void setAge(unsigned num){age = num;}

Person& setName(const std::string& s){name = s; return *this;}
Person& setAge(unsigned num){age = num; return *this;}
```

const Member Functions

```
struct Sales_data {
    // new members: operations on Sales_data objects
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    ...
}
```

The `const` that follows the parameter listed in the declarations of the `Sales_data` member functions is called a const member function. `const` relates to `this`. `this` is a pointer to `const` so a `const` member function **cannot change the object on which it is called**. It implies that that `avg_price` and `isbn` may read but not modify/write to the data members of the objects on which they are called.

Q: Can the following code pass the compilation?

```
double Sales_data::avg_price() const {
    this->units_sold = 10.0;
```

```

    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}

```

A:

7B Defining Non-member Class-Related Functions

Class authors often define **auxiliary functions**, such as the non-member `add`, `read`, and `print` functions. Although such functions define operations that are conceptually part of the interface of the class, they are not part of the class itself.

We define **nonmember functions** as we would do for any other function. As with any other function, we normally separate the declaration of the function from its definition (§6.1.2, p. 206). Functions that are conceptually part of a class, but not defined inside the class, are typically declared (but not defined) in **the same header** as the class itself. That way users need to include only one file to use any part of the interface.

Sales_data.h

```

struct Sales_data {
    ...
};

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);

```

Sales_data.cpp

```

ostream &print(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}

```

Client code

```

| print(cout, total) << endl; // print the results

```

Notice that the IO classes are types that **cannot be copied**, so we may only pass them by reference. Moreover, reading or writing to a stream changes that stream, so both functions take ordinary references, not references to `const`.

74 Constructors

(see ppt)

A constructor is a special member function that is called whenever an object of the class is created. In C++, constructors are member functions with **the same name as the class**; they have **no return type**.

Synthesized Default Constructor

Classes control default initialization by defining a special constructor, known as the **default constructor**. The default constructor is one that takes no arguments.

The default constructor is special and 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**. 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.

Quick Concept Check: How does the synthesized default constructor initialize the data members `a` and `b` in `struct A`?

```

| struct A {
|     int a;
|     std::string b;
| };

```

A:

For our `Sales_data` class we'll define **three** constructors with the following parameters:

A `const string&` representing an ISBN, an `unsigned` representing the count of how many books were sold, and a `double` representing the price at which the books sold.

A `const string&` representing an ISBN. This constructor will use default values for the other members.

- An empty parameter list (i.e., the default constructor) which as we've just mentioned;

we must define the default constructor by ourselves because we have defined other constructors.

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

What = default Means

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

Under the new standard (C++11), if we simply want the default behavior, we can ask the compiler to generate the constructor for us by writing `= default` after the parameter list.

Constructor Initializer List

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

The colon up to the open curly is the constructor initializer list. A constructor initializer list specifies initial values 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). Multiple member initializations are separated by commas.

Quick Concept Check: consider a header file `democlass.h` and a class definition

DemoClass

democlass.h

```
#ifndef DEMOCLASS_H
#define DEMOCLASS_H
struct DemoClass
{
    // add a constructor
    ...
    int itemA, itemB;
};
#endif
```

Add a default constructor using an **initializer list**. We shall (1) initialize `itemA` with the

value of 0 and `itemB` with the value of 1 and (2) in the initializer list, initialize `itemA` with the first parameter and `itemB` with the second parameter.

A:

Ex 7.3: In-class Coding Exercise

CircleClient.cpp

Write a simple `Circle` class (`Circle.h` and `Circle.cpp`) so one can set its radius through a constructor or by a member function. In addition, the `Circle` object can report its radius and area when we print the object. Below are a sample client code and output:

```
#include <iostream>
#include "Circle.h"
using namespace std;

int main()
{
    Circle c1;
    print (cout, c1);
    c1.setRadius(1);    // This sets c1 radius to 1
    print (cout, c1);
    Circle c2(2.5);    // This sets c2 radius to 2.5
    print (cout, c2);
    return 0;
}
```

```
Circle radius: 0 area: 0
Circle radius: 1 area: 3.14159
Circle radius: 2.5 area: 19.6349
```

Answer:

7.2 Access Control and Encapsulation

(see ppt)

In C++ we use **access labels** to enforce encapsulation. A class may contain zero or more access labels:

Members defined after a **public** label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

Members defined after a **private** label are accessible by the member functions of the class, but are not accessible to codes that use the class. The private sections encapsulate (e.g., hide) the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

A class may define members before any access label is seen. The access level of members defined after the open curly of the class and before the first access label depends on how the class is defined. If the class is defined with the `struct` keyword, then members defined before the first access label are public; if the class is defined using the `class` keyword, then

the members are private.

Quick Concept Check: which members are private members in the following?

```
class Fraction
{ private:
    int numer;
    int denom;
public:
    Fraction(int);
    void print();
private:
    Fraction();
};
```

```
struct Fraction
{ int numer;
  int denom;
};
```

```
class Fraction
{ int numer;
  int denom;
};
```

We can now redefine Sales_data again with proper access control for encapsulation.

```
class Sales_data {
public: // access specifier added
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private: // access specifier added
    double avg_price() const
        { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Quick Guideline: In designing a class, we typically put **data** members into the private section and put **member functions** into the public section. A typical class definition has the following form:

```
class className
```

```

{
private:
    data member declarations
public:
    member function prototypes
};

```

7.2.1 Friends

Now that the data members of `Sales_data` are private, our `read`, `print`, and `add` non-member functions will no longer compile. Why?

A: The problem is that although these functions are part of the `Sales_data` interface, they are **not members of the class**.

A class can allow another class or function to access its nonpublic members by making that class or function a **friend**. A class makes a function its friend by including a declaration for that function preceded by the keyword **friend**:

```

class Sales_data {
    // friend declarations for nonmember Sales_data operations added
    friend Sales_data add(const Sales_data&, const Sales_data&);
    friend std::istream &read(std::istream&, Sales_data&);
    friend std::ostream &print(std::ostream&, const Sales_data&);
    // other members and access specifiers as before
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// declarations for nonmember parts of the Sales_data interface
Sales_data add(const Sales_data&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);

```

Quick Exercise: let us modify Ex7.3 `Circle` class to incorporate private and public accessible concepts:

```

#include <iostream>
#include "Circle.h"

```

```

using namespace std;

int main()
{
    Circle c1;
    print (cout, c1);
    c1.setRadius(1);    // This sets c1 radius to 1
    print (cout, c1);
    Circle c2(2.5);    // This sets c2 radius to 2.5
    print (cout, c2);
    return 0;
}

```

```

Circle radius: 0 area: 0
Circle radius: 1 area: 3.14159
Circle radius: 2.5 area: 19.6349

```

Circle.h

```

#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle
{
    public:
        Circle() = default;
        Circle(double r){radius = r;}
        void setRadius(double r){radius = r;}
        double getArea() const {return PI*radius*radius;}
    private:
        double radius = 0.0;
        const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H

```

Circle.cpp

```

#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c)
{
    os << "Circle radius: " << c.radius << " area: "
        << c.getArea() << endl;
    return os;
}

```

Q: Will the code be compilable?

A:

You can fix the problem by (1) adding a simple public utility function `getRadius` (2) or to declare `print` a friend of class `Circle`.

<Solution 1>

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle
{
    public:
        Circle() = default;
        Circle(double r){radius = r;}
        void setRadius(double r){radius = r;}
        double getArea() const {return PI*radius*radius;}
        double getRadius() const {return radius;}
    private:
        double radius = 0.0;
        const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c)
{
    os << "Circle radius: " << c.getRadius() << " area: "
        << c.getArea() << endl;
    return os;
}
```

<Solution 2>

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle
{
```



```
public:
    friend std::ostream &print(std::ostream &os, const Circle &c);
    Circle() = default;
    Circle(double r){radius = r;}
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
private:
    double radius = 0.0;
    const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c)
{
    os << "Circle radius: " << c.radius << " area: "
        << c.getArea() << endl;
    return os;
}
```

Chapter 8: The IO Library

8.3 string Stream

The `sstream` header defines three types to support in-memory IO; these types read from or write to a string **as if the string were an IO stream**. This feature is particularly useful when we deal with input parsing so we will put more emphasis on `istringstream`.

The `istringstream` type reads a string, `ostringstream` writes a string. Table below lists specific operations (in addition to standard IO stream operations) for them.

<code>sstream strm;</code>	<code>strm</code> is an unbound <code>stringstream</code> . <code>sstream</code> is one of the types defined in the <code>sstream</code> header.
<code>sstream strm(s);</code>	<code>strm</code> is an <code>sstream</code> that holds a copy of the string <code>s</code> . This constructor is explicit (§ 7.5.4, p. 296).
<code>strm.str()</code>	Returns a copy of the string that <code>strm</code> holds.
<code>strm.str(s)</code>	Copies the string <code>s</code> into <code>strm</code> . Returns void.

8.3.1. Using an `istringstream`

Familiar with features provided by the `istringstream` is essential to deal with input parsing.

An `istringstream` is often used when we **NEED** to do some work with **individual words within a string**.

The `istringstream` type reads a string. The characters in the string sequence can be extracted from the `istringstream` object using any operation allowed on input streams (e.g., `>>`).

We should 善用 `>>` 的兩大優勢：1. 自動跳過 white space 2. 自動轉型別。

題型 A: string 的內容與格式單純，我們可以簡單用 `istringstream` 解析出需要的資訊

Example: A phone dataset typically consists of the name of the person and his/her phone numbers. A person can have many phone numbers associated with him. For example, phone number from work, from home and mobile. Typical records in an input file (`input.txt`) might look like:

```
| John 33664275 0937495295
```

| Lutz 33664175 0912120212 0987346123

Each record in this file starts with a name, which is followed by one or more phone numbers. We'll start by defining a simple class to represent our input data:

```
| struct PersonInfo
|     { string name;
|       vector<string> phones;
|     };
```

Our program will read the data file and build up a vector of `PersonInfo`. We'll extract the name and phone numbers for each line (work to do with individual words within a line):

Phone.cpp

```
| #include <iostream>
| #include <sstream>
| #include <fstream>
| #include <vector>
| #include <string>
| #include <cstdlib>
| using namespace std;
|
| // members are public by default
| struct PersonInfo
|     { string name;
|       vector<string> phones;
|     };
|
| ifstream& open_file(ifstream& in, const string& file)
| {
|     in.close();    // close in case it was already open
|     in.clear();    // clear any existing errors
|     // if the open fails, the stream will be in an invalid state
|     in.open(file.c_str()); // open the file we were given
|     return in; // condition state is good if open succeeded
| }
|
| void printRecords(ostream &os, const vector<PersonInfo>& people)
| {
|     for (const auto &entry : people) { // for each entry in people
|         os << entry.name << " has " << entry.phones.size()
|           << " phones and the numbers are: ";
|         for (const auto &nums : entry.phones) { // for each number
|             os << nums << " ";
|         }
|         os << endl;
|     }
| }
|
| int main()
| {
|     ifstream fin;
```

```

string file_name;
cout << "Enter the file name: ";
cin >> file_name;
if (!open_file(fin, file_name)) {
    cerr << "Complain: I cannot find the file" << endl << endl;
    cerr << system("dir") << endl; //only for windows OS
    return -1;
}

string line, word;
vector<PersonInfo> people;

// read the input a line at a time until end-of-file (or other error)
while (getline(fin, line)) {
    PersonInfo info; // object to hold this record's data
    istringstream record(line); // bind record to line we just read
    record >> info.name; // read the name
    while (record >> word) // read the phone numbers
        info.phones.push_back(word); // and store them
    people.push_back(info); // append this record to people
}

printRecords(cout, people);
return 0;
}

```

```

Enter the file name: input.txt
John has 2 phones and the numbers are: 33664275 0937495295
Lutz has 3 phones and the numbers are: 33664175 0912120212 0987346123

```

Example: Another common usage of `istringstream` is to **retrieve the numeric value** from the string. Reading an `istringstream` automatically converts from the character representation of a numeric value to its corresponding arithmetic value:

CovertNum.cpp

```

#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main()
{
    string s = "Some numerical values are 23 5 7";
    cout << s << endl;
    istringstream input_istring(s);
    string dump;
    int val1, val2, val3;
    input_istring >> dump >> dump >> dump
    >> val1 >> val2 >> val3;
    cout << "The sum is: " << val1 + val2 + val3 << endl;
    return 0;
}

```

```
Some numerical values are 23 5 7
The sum is: 35
```

In-class Exercise 8.1: Input two stocks with each stock containing stock name, number of share owned and the share price and report the most valuable one. For example:

```
Enter the first share (name, num_holding, share_price): AAPL 1 100.73
Enter the second share (name, num_holding, share_price): MSFT 2 44.03
AAPL with total value of 100.73 is more valuable.
```

The client code looks like:

StockClient.cpp

```
#include <iostream>
#include "Stock.h"

using namespace std;

int main()
{
    cout << "Enter the first share (name, num_holding, share_price): ";
    string line;
    getline(cin, line);
    Stock s0(line);

    cout << "Enter the second share (name, num_holding, share_price): ";
    getline(cin, line);
    Stock s1(line);
    Stock top = s0.topval(s1);
    cout << top.getName() << " with total value of "
        << top.getTotalVal() << " is more valuable." << endl;
    return 0;
}
```

And Stock.h looks like:

Stock.h

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

class Stock
{
public:
    Stock(std::string record);
    const Stock& topval(const Stock& s) const;
    std::string getName() const {return name;}
    double getTotalVal() const {return total_val;}
private:
    std::string name;
    unsigned share_num;
```

```
double share_val;  
double total_val;  
};  
  
#endif // STOCK_H
```

Implement the `Stock.cpp`.

A:

題型 B: string 的內容與格式不單純，我們需先整理 string，才可以簡單用 `istringstream` 解析出需要的資訊

Example: write a `Matrix` class that reads its contents with `double` prescribed from input.

We will use MATLAB-like syntax to parse the input. For example, if you have a matrix

$A = \begin{bmatrix} 1.1 & 3.0 & 6.5 \\ 7.8 & 4.5 & 2.2 \end{bmatrix}$, the MATLAB-like input syntax discards white space and uses

semi-colon to separate different rows and comma to separate the elements in a row. The syntax goes like:

```
| A=[1.1, 3.0, 6.5; 7.8, 4.5, 2.2]
```

We would like to put these `double` elements into a matrix `vector<vector<double>>`.

解題技巧：我們需先處理此 string before using `istringstream`. 我們第一步先把 elements in a row separated by semicolon 解析出，並將資訊放在 `vector<string>`。

```
| A=[1.1, 3.0, 6.5; 7.8, 4.5, 2.2; 3.4, 5.6, 8.9]

| string A;
| getline(cin, A); // A=[1.1, 3.0, 6.5; 7.8, 4.5, 2.2; 3.4, 5.6, 8.9]
| vector<string> vs;
| string s;
| for (auto c : A)
|     { if (c == '[')
|       {
|           s.clear();
|       }
|       else if (c == ',')
|           s.push_back(' ');
|       else if (c == ';' || c == ']') {
|           vs.push_back(s);
|           s.clear();
|       }
|       else
|           s.push_back(c);
|     }
```

Q: If we print the contents of `vector<string> vs` after parsing:

```
| for (auto e : vs)
|     cout << e << endl;
```

What are the outputs if we input a string?

```
| A=[1.1, 3.0, 6.5; 7.8, 4.5, 2.2; 3.4, 5.6, 8.9]
```

A:

第二步再解析出 **each element in a row**. 因為 `vector<string>` vs 的每一 `string` 的內容與格式單純，我們可以簡單用 `istringstream` 解析出需要的資訊。

8.3.2. Using an `ostringstream`

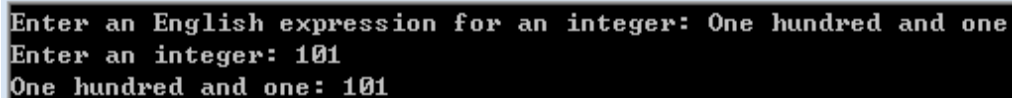
The `ostringstream` type writes a string and characters can be inserted into the `ostringstream` object with any operation allowed on output streams (e.g., `<<`). With `ostringstream`, we can easily build up our output a little at a time and finally output the combining string.

CombineNumeric.cpp

Example

```
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

int main() {
    cout << "Enter an English expression for an integer: ";
    string s;
    getline(cin, s);
    cout << "Enter an integer: ";
    int num;
    cin >> num;
    ostringstream oss;
    oss << s << ": " << num;
    cout << oss.str() << endl;
    return 0;
}
```

A screenshot of a terminal window showing the output of the program. The text is as follows:

```
Enter an English expression for an integer: One hundred and one
Enter an integer: 101
One hundred and one: 101
```