> 由 KiOii (*_EM_Cpper*)整理。 (KiOii (*_EM_Cpper*) makes this note.)

Classes

Contents

- **Section 7.1 Defining Abstract Data Types**
- **Section 7.2 Access Control and Encapsulation**
- **Section 7.3 Additional Class Features**
- **Section 7.4 Class Scope**
- **Section 7.5 Contructors Revisited**
- **Section 7.6 static Class Members**

- *fundamental ideas behind classes :* **data abstraction** and **encapsulation**
  - *data abstraction:* the sepatation of **interface** and **implementation**
    - *interface:* **operations**
    - *implementation:* **data members**、**functional body**、**private function**
  - *encapsulation:* **enforces** the **sepatation** of a class's interface and implementation

7.1 Defining Abstract Data Types

7.1.1 Designing the *Sales_data* Class

- *operations:* **isbn、combine、add、read、print、+,+=,=,<<,>> operators**

Using the Revised *Sales_data* Class

```
Sales_data total;
if(read(cin,total))
{
    Sale_data trans;
    while(read(cin,trans))
    {
        if(total.isbn() == trans.isbn())
            total.combine(trans);
        else
        {
            print(cout,total) << endl;
            total = trans;
        }
        print(cout,total) << endl;
    }
}else
{
    cerr << "No data?!" << endl;
}
```

7.1.2 Defining the Revised *Sales_data* Class

```
struct Sales_data
{
    // member :operations
    std::string isbn()const {return bookNo;}
    Sales_data &combine(const Sales_data&);
    // member :data
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// interface
Sales_data add(const Sales_data&,const Sales_data&);
std::ostream &print(std:ostream&,const Sales_data&);
std::istream &read(std::istream&,Sales_data&);
```

this

- *define:* implicitly to the members of the object
- ***total.isbn()***

    - **total.isbn(*this-type* this)**
        - parameter: **this**
        - argument: **the address of the object**
        - *such as:* ***Sale_data::isbn(&total);***
- ***return bookNo***
    - *such as:* ***return this->bookNo;***

const Member Functions

- *purpose:* **modify the implicit this pointer** (*const pointer*)

```
// psedo-code
std::string Sale_data::isbn(const Sale_data *const this){return this->bookNo;}
```

Class Scope and Member Functions

- *compiler does two step:* the member declarations are compiled first,after which the member function bodies.

Defining a Member Function outside the Class

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

Defining a Function to Return "This Object"

```
Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += ths.units_sold;
    revenue += ths.revence;
    return *this;
}
```

7.1.3 Defining Nomember Class-Related Functions

> Nomember functions that are part of the interface of a class should be declared in ther same heard as the class itself.

Defining the read and print Functions

```
istream& read(istream &is,Sales_data &item)
{
    double price = 0;
    is >> item.bookNo >> item.units_sold >> price;
    iterm.revenue = price * item.units_sold;
    return is;
}

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

7.1.4 Constructors

- *define:* control **object initialization**
  - we can define one or more
- *wraning:* run whenever **an object of a class type is created**
- *wraning:* **the same name as the class、no return type、not const function**
  - Constructors **can write** to **const objects** during their construction

The synthesized Default Constructor

- *default constructor:*
  - *wraning:* no parameters list
  - *calling way:* `Class-type object;`
    - If there is an in-class initailizer,use it to initializing member.
    - Otherwise, default-initialize the member.
- *synthesized default constructor:*
  - *define:* The compiler-generated constructor

Some Class Cannot Rely on the Synthesized Default Constructor

- we must define its own default constructor

Defining the *Sales_data* Constructors

```
struct Sales_data
{
    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),revence(p*n){}
    Sales_data(const std::string &s);
    // ...
};
```

- *= default:*
    - *standard:* **C++11**
    - the same work as the synthesized version we had been using.

Constructor Initializer List

```
Sales_data(const std::string &s)
    :bookNo(s) // : initializer list
    {}
```

- *define:* a list of member names
    - *way:* ***member(value)*** or ***member{value}***
    - *wraning:* the order of initialization depend on the order that define in the class

7.1.5 Copy, Assignment, and Destruction

> In addition to defining how objects of the class type are initialized, classes also control what happens when we copy, assign, or destroy objects of the class type.

Some Classes Cannot Rely on the Synthesized Versions

7.2 Access Control and Encapsulation

> Our class is not yet encapsulated—users can reach inside a class object and meddle with its implementation.

- *the way to Encapsulation:* use **access specifiers**
    - *public:* The public members **define the interface** to the class
    - *private:* **encapsulate(hide) the implementation**

```
class Sales_data
{
public:
    Sales_data() = default;
    Sales_data(const std::string &s,unsigned n,double p)
        :bookNo(s),units_sold(n),revence(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:
    double avg_price()const{return units_sold ? revence / units_sold : 0;}
    //...
};
```

- *wraning:* A class may contain **zero or more access specifiers**

Using the class or struct Keyword

- *struct:* **default public**
- *class:* **default private**

7.2.1 Friends

> Now that the data members of Sales_data are private, our read, print, and add functions will no longer compile.

- *define:* allow another class or function to **access its nonpublic members**
  - *way:* **making that class or function a friend**
  - *code look:*

```
class Sales_data
{
    // making interface friend
    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&);
public:
    //...
private:
    //...
};
// interface declaration
Sales_data add(const Sales_data&,const Sales_data&);
```

- *wraning:* Ordinarily it is a good idea to group **friend declarations together** at the beginning or end of the class definition.

7.3 Additional Class Features

7.3.1 Class Members Revisited

> To explore several of these additional features, we'll define a pair of cooperating classes named Screen and Window_mgr.

Defining a Type Member

```
class Scree
{
public:
    using pos = std::string::size_type;
private:
    pos cursor = 0;
    pos heigh = 0,with = 0;
    std::string contents;
};
```

mutable Data Members

- *define:* is never const,**even when it is a member of a const object.**

```
class Screen
{
public:
    void some_member() const;
private:
    mutable size_t access_ctr;  // mutable data member
};
void Screen::some_member()const
{
    ++ access_ctr;             // even is const object
}
```

Initailizers for Data Members of Class Type

```
class Windows_mgr
{
private:
    std::vector<Screen> Screen{Screen(24,80,' ')};
};
```

7.3.2 Function That Return *this

```
class Screen{
public:
    Screen &set(char);
    Screen &set(pos,pos,char);
};
inline Screen &Screen::set(char c)
{
    contents[cursor] = c;
    return *this;
}
```

```
inline Screen &Screen::set(pos r,pos col, char ch)
{
    contents[r * width + col] = ch;
    return *this;
}
```

Return *this from a const Member Function

- *return type:* **const class-type &**

Overloading Based on const

- *wraning:* we can only **call const member functions** on a **const object**

```
class Screen{
public:
    Screen &display(std::ostream &os)
    {
        do_display(os);
        return *this;
    }
    const Screen &display(std::ostream &os)const
    {
        do_display(os);
        return *this;
    }
private:
    void do_display(std::ostream &os)const
    {
        os << contents;
    }
}
```

7.3.3 Class Types

Class Declarations

- *way:* `class Screen;`
    - **forward declaration**
    - **incomplete type**
        - *wraning:* can only define pointer or reference to such types

```
class Link_screen{
    Screen window;
    Link_screen *next;
    Link_screen *prev;
};
```

7.3.4 Friendship Revisited

Friendship between Classes

```
class Screen{
    friend class Window_mgr;
    //...
};

class Window_mgr{
public:
    using ScreenIndex = std::vector<Screen>::size_type;
    void clear(ScreenIndex);
private:
    std::vector<Screen> screens{Screen(24,80,' ')};
};
```

Makeing A Member Function a Friend

```
class Screen{
    //...
    friend void Window_mgr::clear(ScreenIndex);
    //...
}
```

Friend Declarations and Scope

```
struct X{
    firend void f(){ }
    X() { f();}            // error:no declaration for f
    void g();
    void h();
};
void X::g(){reunrn f();}   // error:no declarations for f
void f();                  // declaration for f
void X::h(){return f();}   // ok
```

7.4 Class Scope

Scope and Members Defined outside the Class

- *wraning* **a class is a scope**

7.4.1 Name Lookup and Class Scope

- *wraning:* Member **function** definitions are processed **after** the compiler procresses all of **the declarations** in the class.
    - ***function can use any name defined inside the class.***

Name Lookup for Class Member Declarations

```
typedef double Money;
string bal;
class Account{
public:
    Money balance(){return bal;}
private:
    Moeny bal;
};
```

Type Names Are Special

```
typedef double Moeny;
class Account{
public:
    Moeny balance(){return bal;} // use the name Moeny
private:
    typedef double Moeny;        // error:cannot redefine Money;
    Moeny bal;
}
```

- *wraning:* if a member uses a name from an outer scope and that **name is a type**,then the class **may not** susequently **redefine that name**.

Normal Block-Scope Name Lookup inside Member Definitions

- *A name used int the body of a member function is resolved as follows:*
    1. the body of this function,precede the use of the name
    2. class
    3. look for a declaration that is in scope before the member function definition

After Class Scope,Look int the Surrounding Scope

- *use global name:* `::`

7.5 Constructors Revisited

- *wraning:* significant between **assignment and initialization** is depends on the **type** of the data member.

Constructor Initializers Are Sometimes Required

```
class ConstRef{
```

```
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
// 1  error
ConstRef::ConstRef(int ii)
{
    i = ii;
    ci = ii; // error
    ri = i;  // error
}
// 2 ok
ConstRef::ConstRef(int ii)
    :i(ii),ci(ii),ri(i)
{}
```

Order of Member Initialization

- *wraning:* Members are initailized in the order in which they appear in the class definition

```
class X{
    int i;
    int j;
public:
    // error : undefine error
    X(int val):j(val),i(j){}
};
```

Default Arguments and Constructors

```
class Sales_data{
public:
    Sales_data(std::string s = ""):bookNo(s){}  // define default constructor
    //...
}
```

7.5.2 Delegating Constructors

- *define standard:* **C++11**

```
class Sales_data{
public:
    Sales_data(std::string s,unsigned cnt,double price):
        bookNo(s),units_sold(cnt),revence(cnt*price){}
    Sales_data():Sales_data("",0,0){}
    // call Sales_data(std::string s,unsigned cnt,double price),then continue
}
```

7.5.3 The Role of the Default Constructor

- *default initialized:*
    - define **nonstatic variables** or **arrays** at block scope **without initializers**
- *wraning:* it is almost always right to **provide a default constructor** if other constructors are being defined

Using the Default Constructor

```
Sales_data obj();      // a function,not an object
Sales_data obj;        // a default-initialized object
```

7.5.4 Implicit Class-Type Conversions

- *converting constructor:* **a single argument constructor**
    - *define:* implict conversion from the **parameter type to the class type**

Only One Class-Type Conversion Is Allowed

Suppressing Implicit Conversions Defined by Constructors

- **use explicit :** `explicit Sales_data(const std::string&):bookNo(s){}`
- *wraning:* explicit allowed only on a constructor decaration in a class header

```
// error
explict Sales_data::Sales_data(istream& is)
{
    //...
}
```

explicit Constructors Can Be Used Only for Direct Initialization

```
/*
*   null_book is a string
*/
Sales_data item1(null_book); // ok
Sales_data item2 = null_book;// error
```

Explicitly Using Constructors for Conversions

- *using static_cast*
  - `item.combine(static_cast<Sales_data>(cin));`

Library Classes with explicit Constructors

- The vector constructor that takes a size is explicit.

7.5.5 Aggregate Classes

- *aggregate class:*
  - **All of it data member** are **public**
  - It does **not define** any **constructors**
  - It has no in-class initializers

    ```
    class C{
        int i = 10; // in-class initializer
    };
    ```

  - It has **no base classes** or **virtual** **functions**,which are class-related features
- *code-look:*

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

- *wraning:* We **can initialize** the data members of an aggregate class by providing a braced **list of member initializers**.

  ```
  Data vall = {0,"asfa"};
  ```

7.5.6 Literal Classes

- *two forms:*

- *one:* An **aggregate class** whose **data members** are all of **literal type**
- *second:*
  - **data members** all must have **literal type**
  - must have at least one **constexpr constructor**
  - if has an in-class initializer,the initializer for a member of buil-in type must be a constant expression or the member has class type and use its constepxr constructor.
  - use default definition for its destrutor

constexpr Constructors

- *wraning:* A **constexpr constructor can** be decalared as **= defualt** , or body is empty

```
class Debug{
public:
    constexpr Debug(bool b = true)
        :hw(b),io(b),other(b)
        {}
    constexpr Debug(bool h,bool i,bool c)
        :hw(b),io(i),other(c)
        {}
    constexpr bool any()
    {
        return hw||io||other;
    }
    //...
    void set_io(bool b){io = b;}
    //...
private:
    bool hw;
    bool io;
    bool other;
}
```

  - *wraning:* A **constexpr constructor** is used to generate objects that are **constexpr and for parameters** or **return types in constexpr functions**:

```
constexpr Debug io_sub(false,true,false);  // consexpr object
```

7.6 static Class Member

- *define:* associated with the class

Declaring static Member

```
class Account{
public:
    void calculate(){
        amount += amount * interestRate;
    }
```

```
    static double rate(){
        return interestRate;
    }
    static void rate(double);
private:
    std::string owner;
    static double interestRate;
    static double initReate();
}
```

- *wraning:* **static member functions** do not have a this pointer,so they **cannot define const** function

Using a Class static Member

- *way:*
  - use ::

    ```
    double r;
    r = Account::rate;
    ```

  - use object、reference、pointer

Defining static Members

- *wraning:* the **keyword appears only with the declaration** inside the class body

  ```
  void Account::rate(double newRate){
      interestRate = newTate;
  }
  ```

- *way:* define **outside the class body**

  ```
  double Account::interestRate = initRate();
  /*
  *   Account:: means this definition in the scope of the class
  */
  ```

In-Class Initialization of static Data Members

- *initalizer :* **constant integer**
- *constexpr data member:* `static constexr int period = 30;`

```
class Account{
 public:
    static double  rate(){return interestRate}
    static void rate(double);
private:
    static constexpr int period = 30;     // no define,just constexpr
    double daulytbl[period];
};
constexpr int Account::period;            // define,but we cannot give it initializer
```

- *wraning:* **we should define static constexpr member**
  - example: pass Account::period to a function that takes a const int&,the period must be defined.

static Member Can Be Used in Ways Ordinary Members Can't

- *As itself data member:*

```
class Bar{
public:
    //...
private:
    static Bar mem1;   // ok
    Bar       *mem2;   // ok
    Bar        mem3;   // error,must have complete type
};
```

- *default argument:*

```
class Screen{
public:
    Screen &clear(char = bkground);
private:
    static const char bkground;     // we must define it.
}
```