第七章 类

类的基本思想是**数据抽象**和**封装**

- 数据抽象是一种依赖**接口**和**实现**分离的编程技术
  - 类的接口包括用户所能执行的操作
  - 类的实现则包括类的数据成员、负责接口的视线函数体以及定义类所需的各种私有函数
- 封装实现了类的接口和实现的分离；

  封装隐藏了类的实现细节
- 类要向实现数据抽象和封装，首先需要定义一个抽象数据类型。

  在抽象数据类型中，类的设计者负责考虑类的实现过程；

  使用类的程序员只需要抽象地思考类做了什么，而无须了解类型的工作细节

定义抽象数据类型

示例

```
struct Sales_data{
    // 成员：关于Sales_data对象的操作
    string isbin() const {return bookNo};
    Sales_data& combine(const Sales_data);
    double avg_price() const;

    // 数据成员
    string bookNo;
    unsigned units_sold=0;
    double revenue=0;
};
// Sales_data的非成员接口函数
Sales_data add(const Sales_data&, const Sales_data&);
ostream &print(ostream&,const Sales_data&);
istream &read(istream&,Sales_data&);
```

> 区分成员函数和接口(非成员函数)
>
> - 成员函数的声明必须在类的内部，但定义（实现）既可以在内部也可以在外部

> - 作为接口组成的非成员函数，他们的定义和声明都在外部
>
> **成员函数有一个隐式this指针作为参数，可以直接使用类内的对象，调用形式** `类对象.函数()`
>
> **非成员函数通常只能对传入的类进行操作，调用形式** `函数()` **,实际上就是会对类进行操作的普通函数**
>
> - 非成员函数的声明一般和类的声明在同一个头文件中

引入this

- 成员函数通过隐式参数 `this` 来访问调用它的那个对象。当我们调用一个成员函数时，用请求该函数的对象初始化this.

  例如 `total.isbn()` ，编译器负责把total的地址传递给this

- 在成员函数内部，我们可以直接调用该函数的对象的成员，而无须通过成员运算符，因为this所指的正是这个对象。

  任何对类成员的直接访问都被看作this的隐式引用 `this->bookNo`

引入const成员函数

```
1 string isbin() const {return bookNo};
```

const的作用是修改隐式 `this` 的类型

- 默认情况下，this的类型是 `T *` ，不能够修改this只是因为他是一个右值表达式

  > 这个时候，无法将常量的对象绑定的this上，即无法在一个常量对象上调用普通的成员函数

- 紧跟在参数列表后的 `const` 表示this是一个 `const T*` 的类型；

  像这样使用const的成员函数叫 **常量成员函数**

  - **常量成员函数不能改变调用它的对象的内容**
- **常量对象、以及常量对象的引用或指针都只能调用常量成员函数** （因为传入this参数的const限制）

  非常量对象则可以调用常量函数或非常量函数

在类的外部定义成员函数

- 类外定义成员函数，必须与声明匹配（返回类型、参数列表、函数名、const属性都一致）
- 类外部定义的函数名必须它所属的类别

```
double Sales_data::avg_price() const{

}
```

  - 这样实际上告诉编译器，该函数是属于类作用域的（类本身就是一个作用域），之后的代码都是位于类作用域内的。

定义一个返回this对象的函数

```
Sales_data& Sales_data::combine(const Sales_data & rhes){
    units_sold += rhes.units.sold;
    revenue += rhes.revenue;
    return *this;
}
```

- `return *this` 返回调用该函数的的对象，并且该对象内容已经发生类改变
- 返回类型 `Sale_data&` ，之所以要返回引用类型，是为了使得函数尽量模拟加法运算符的

定义read和input函数

```
istream &read(istream &is,Sales_data & item){
    double price = 0;
    is>>item.bookNo>>item.units_sold>>price;
    item.revenue=price*item.units_sold;
    return is;
}
ostream &print(ostream &os,Sales_data & item){
    os<<item.isbn()<<" "<<item.units_sold<<;
    return os;
}
```

- IO类属于不能拷贝的对象，所以只能通过引用来返回他们；

  所以要返回IO对象，返回类型也必须是引用类型
- print不负责换行。

一般来说，执行输出任务的函数应尽量减少对格式的控制，确保由用户自己决定是否换行

- `read(cin,total)>>t`是成立的，因为返回的对象仍然式一个标配输入

构造函数(后续还会介绍)

构造函数的任务是初始化类对象的数据成员，无论何时只要类的对象被创建，就会执行构造函数、

- 构造函数和类名相同，没有返回类型
- 类可以包含多个构造函数（类似函数重载）
- 构造函数不能被声明为`const`（即，不能声明为常量成员函数）；

  即使我们创建一个常量类对象，也是直到构造函数完成初始化过程后，才取得“常量”属性的，所以构造函数在const对象的构造过程中也能被调用
- 每一个类都有一个默认构造函数，无须任何参数
  - 如果类内成员有类内初始值，则用该值初始化
  - 否则进行默认初始化成员

> - 只有在没有显示定义一个构造函数时，这个默认构造函数才会构造；
>
>   也就是说，一旦存在构造函数，则将不会自动生成默认构造函数（这是就必须显示初始化声明的类对象）
> - 某些类不能依赖与编译器生成的默认构造函数
>   - 含有内置类型或复合类型对象的类（因为这些类型的默认初始化是未定义的）
>   - 类对象包含另一个类
> - 通常情况下，使用类内初始值可以确保对象能被赋予一个正确的值；
>
>   但是如果编译器不支持，那么所有构造函数都应该显示地初始化每个内置类型

构造函数写法

```
Sales_data()=default;        // 显式生成默认构造函数
Sales_data(const stirng& s, unsigned n):bookNo(s), units_sold(n){}
// 构造函数初始化列表
```

- 当某个数据成员被构造函数初始化列表忽略时，它将以与默认构造函数相同的方式隐式初始化
- 构造函数放在`public`下

拷贝、赋值和析构

如果不主动定义这些操作，则编译器将替我们合成它们。一般编译器生成的版本将对象的每个成员执行拷贝、赋值和销毁操作

> - 某些类不能依赖于编译器合成的版本
>   - 特别是，当类需要分配类对象之外的资源时，比如管理动态内存的类(较少出现)(**后续**)

访问控制和封装

> 什么叫"类还没有封装"?
>
> 用户可以直达类内对象并且控制它的具体实现细节。我们可以用 **访问说明符** 加强类的封装性
>
> `public`后的成员可以被整个程序访问（Struct 成员默认在该说明符下）
>
> `private`后的成员可以被类的成员函数访问，但是不能被使用该类的代码访问（class成员默认在该说明符下）

友元

类可以允许其他类或者函数访问其非公有成员（私有、保护），方法是令其他类或者函数成为它的友元

```
// 在类中的友元声明语句
Class Sales_data{
    friend 函数声明;     // 在该函数内，Sales_data对象可以通过.运算符访问private下成员
    friend class 类名;         // 在友元类的成员函数可以通过类对象访问该类所有成员
}
```

- 友元声明只能出现在类定义的内部，但具体位置不限；

  友元不是类的成员，不受所在区域访问控制级别的限制。

  > - 建议在类的开始或末尾处集中定义友元
  > - 建议除内部友元声明外，在类外部也提供一个该友元函数的独立函数声明，放在和类声明同一个头文件中

- 友元函数是不能被继承的;

  > 每个类控制自己的友元类或友元函数

- 友元函数能定义在类内部，这样的函数是隐式内联的

类的其他特性

类成员再探

- 在类中，仍然可以使用类型别名（存在访问限制）；

  用于定义类型的成员必须先定义后使用

```
class Screen{
    public:
        using pos=string::size_type;
    ...
}
```

- **定义**在类内部的成员函数是自动 `inline` 的(当然也可显示声明）；

  在外部定义的成员函数能在实现是再被设置为 `inline`；

> 建议只在类外部定义的地方说明 `inline`，这样可以使类容易理解

- `mutable关键字`：为了突破const的限制而设置的。被 `mutable` 修饰的变量，将永远处于可变的状态，即使它是一个 `const` 对象的成员

> - 一个可变数据成员永远不会是const
>
> ```
> void some_member() const{
>     ++num;      // 尽管是一个常量成员函数，但是被mutable修饰的成员变量仍然可变，而其他的则不可变
> }
> ```

- 对类成员的类内初始化，必须以符号 `=` 或花括号（列表初始化）表示、

返回对象的引用

```
Screen& Screen::set(){...}
```

- 这里，返回引用，可以将一系列对Screen对象的操作连接在一条表达式中，仍然可以对同一个对象操作
  但是，如果是普通类型，那么返回的是调用对象的副本，连续的调用不会对原对象造成影响

```
myScreen.set(.set()
```

从const成员函数返回*this

```
Stu t() const{      // 没问题，因为对于*this，const是顶层限制
    return *this;
}
Stu &t() const{     // 编译出错，引用类型要完全匹配 所以要用const Stu &t
    return *this
}
```

> 建议对于公共代码使用私有功能函数
>
> - 首先避免在多处使用同样的代码，方便调试和修改
> - 其次，对于类内定义的成员函数，会被隐式声明为内联函数，不会带来额外开销

类类型

类的声明

```
class Screen;
```

- 前向声明：只声明Screen是一种类类型，但清楚具体有那些成员

  在为定义之前，可以称Scree是一种不完全类型

- 使用场景有限：
  - 定义指向这种类型的指针或引用
  - 可以声明（但不可以定义）以不完全类型作为参数或者返回类型的函数
- 创建类的对象时，该类必须被定义。

友元再探

令成员函数作为友元

```
class Screen{
    friend void Window_mgr::clear(ScreenIndxe);
}
```

- 当把一个成员函数声明成友元时，必须明确指出该成员函数属于哪一个类

- 这这友元声明要**注意顺序**，即成员函数的声明应该在友元声明的出现，成员函数的定义(次时要访问类内对象了)要在友元声明的后面

> 但是如果令类作为友元，则成员函数是可以定义在类内部使用其他类的类内对象

友元声明和作用域

- 类和非成员函数的声明不是必须在它们的友元之前。当一个名字第一次出现在一个友元声明中，我们隐式的假定该名字在当前作用域是可见的。
- **友元本身不一定真的声明在当前作用域中**

  如果该友元函数是另一类的成员函数，则其作用域为另一类的类域，否则与一般函数相同。

> 与一般函数相同，是指必须在该函数声明后使用

> - 类和非成员函数的声明不是必须在它们的友元之前呢?
>
>   因为 `friend` 仅仅起到一个声明作用，有没有具体实现是无关紧要的;
>
>   但是这里理解 *`frined`* 声明，不是一个对应函数真正的声明，所以如果要使用被友元声明的函数，还得再其真正声明后使用才行
> - 为什们类成员函数的声明必须在友元前面呢？定义又要在友元声明后？（假设A,B,B内定义A中友元函数a)
>
>   因为类成员函数要指明所属类，这时类必须经过的定义才行，所有一定要在它前面出现过该类的定义才行;
>
>   而因为a一定在定义前被声明，如果在友元定义前就实现a，那么a里面是无法访问B的（因为B的声明和实现都没有在A之前出现

```
struct x{
    friend void f(){  /* 友元函数可以定义在内部 */}
 // friedn void f() 是同样的情况
    x(){f();}        // 错误，f还没正式声明
}
void x::g(){f();}    // 错误
void f();
void x::h(){f();}    // 正确
```

类的作用域

名字查找和类的作用域

- 编译器编译类的顺序：编译成员的声明（类型声明和函数声明），在编译成员函数体

> 成员函数体可以随意使用类中的其他成员而无须在意这些成员出现的次序

> **类中，不能对变量进行赋值操作**
>
> **类比较特殊，实例化前不分配空间**
>
> 原因在于，声明类的时候他是不存在于内存中，也就是说他不是实际存在的对象，只是一张设计图；而变量赋值的工作是，在该变量的内存位置中保存某某值
>
> 因此你可以在里面**声明一个变量 （int a)**（类内部该操作是一个声明操作不分配内存，在main中则是定义操作）。但是它其实不存在实体因此给该变量的内存位置保存数据是不可能的。
>
> 而之所以C++11支持初始化赋值操作，其实实现是由默认构造函数实现的

- 类型名要特殊处理 **??**

  > 我觉得没有错误

```
typedef double Money;
class Account{
    public:
        Money banance(){     // 使用外层作用域的Money
           Money a=1.2;      // 使用类内Money
           return bal;
        }
    private:
        typedef double Money;
        Money bal;
}
```

  - **typedef是声明不是定义**

```
1
2 class Account{
3     public:
4         Money banance(){     // 报错，提示找不到类型Money
5            return bal;
6         }
7     private:
8         typedef double Money;
9         Money bal;
10 }
```

- 因为类内编译的顺序是先编译声明，而对于声明的编译则是从上往下，所以编译到`Money banance()`是会提示错误

再探构造函数

构造函数初始化列表

如果没有在构造函数初始化列表中显式地初始化成员，那么成员将在构造函数体之前执行默认初始化

> **注意：** 赋值并不总是等于初始化操作
>
> ```
> 1 class test{
> 2  test(int t){
> 3      a=t;
> 4      b=t;        // 错误，不能给const赋值
> 5      c=t;        // 错误，b没有被初始化
> 6  }
> 7     private:
> 8     int a;
> 9     int &b;
> 10    const int c;
> 11 }
> 12 // 正确,显示初始化引用和const对象
> 13 test(int t):a(t),b(t),c(t){}
> ```
>
> - 随着构造函数的函数体一开始执行，初始化操作就结束了
>
> 所以，那些需要显示初始化的变量，必须利用初始值列表进行初始化

- 构造函数初始化列表中的初始值的前后关系不会影响实际的初始化顺序；

成员的初始化顺序与它们在类定义中的出现顺序一致

> 最好保证顺序一致性；
>
> 可能的话，避免使用某些成员初始化其他成员

委托构造函数

C++11

- 一个委托构造函数使用它所属类的其他构造函数执行自己的初始化过程；
- 一个委托构造函数也有一个成员初值的列表和一个函数体
  初始化列表只有一个唯一的入口，就是类名本身，后面紧跟着圆括号的参数列表（调用对应的构造函数）

  依次执行委托任务，再执行调用者的函数体

默认初始化函数

- 一旦定义类其他构造函数，编译器就不会构造隐式初始化函数

```
class Test{
    public:
        Test(){a=3;}        // 这个实际上是显式定义了默认构造函数
    int a;
}
```

- `Test t` 调用默认构造函数进行初始化对象（不需要加括号）

隐式的类类型转换

如果构造函数**只接受一个实参**，则它实际上定义了转换为此类类型的隐式转换机制

```
class test{
public:
    test(int t) {       // 实际上定义了从int类型向test类型隐式转换的规则
        a=3;
    }
    test(){
        a=3;
    }
    test combie(test t,int t2){
        a=t.a+t2;
```

```
        return *this;
    }
    int a=1;
};
int main(){

    test t;
    cout<<t.a;
    test t2(2);
    t.combie(2,2);      // 第一个2,实际上进行了类类型转换
    cout<<"s"<<t.a;
    return 0;

```

- 只允许一步隐式类型转换

```
t.combine("aaa",2);     // 报错,"aaa" -> string -> Sales_data
t.combie(string("aaa"));    // 正确，显式+隐式
```

- `explicit`：抑制构造函数定义的隐式初始化
  - **只对一个实参的构造函数有效**，需要多个实参的构造函数不能用于执行隐式类型转换
  - 只能在类内声明构造函数时使用，在类外部定义时不应重复
  - `explicit构造函数` 执行拷贝初始化时，只能使用直接初始化，编译器不会在自动转换过程中使用构造函数

```
int a=3;
test t=a;       // 拷贝形式的初始化

// 使用explicit后
test t=a;       // 报错
test t(a);      // 正确
```

  - 仍然可以使用该函数进行显示初始化

```
t.combine(test(3),2);
```

  - 标准库中，接受单参数的 `string` 构造函数不是 `explicit` 的

    接受一个容量参数的 `vector` 构造函数是 `explicit` 的 **哪一个?**

聚合类的初始化方式

类需要满足的条件：

- 所有成员都是 `public` 的；
- 没有定义任何构造函数；
- 没有类内初始值；
- 没有基类，也没有 `virtual` 函数(虚函数) **后续15章**

> 可以理解为C中的结构体，所以结构体也可这样初始化

- 特性：使得用户可以直接访问其成员，并且具有特殊的初始化语法形式
  - 可以提供一个花括号括起来的成员初始值列表，用其初始化聚合类中的数据成员
    - 只能少，不能多；少了用值初始化
    - 顺序必须一致

```
struct t2{
  .
};
// test是一个聚合类
test2 t2;
t2={1,"2"};
test2 t3={1,"3"};
```

字面值常量类 不懂

- 类需要满足的条件：
  - 数据成员都是字面值类型的聚合类

  or

  - 数据成员必须是字面值类型

    > 确保编译时求值

  - 类中至少有一个 `constexpr` 构造函数

    > 可以创建这个类的 `constexpr` 类型对象

  - 如果一个数据成员含有类内初始值，则初始值必须是一条常量表达式；

    如果成员是一种类类型，则初始值必须使用成员自己的 `constexpr` 构造函数

    > 保证即使有类内初始化，也能在编译期间完成

- 类内的析构函数必须是所有默认的

  > 保证析构函数没有不能预期的操作

- `constexpr` 构造函数

  除了声明为 `=default` 或 `=delete` 外，

  - 两要求：
    - 构造函数不能包含返回语句
    - `constexpr` 函数唯一能拥有的可执行语句就是返回语句 `??`

      > 所以一般 `constexpr` 构造函数体为空

    - 必须初始化所有数据成员，初始值或者是使用 `constexpr` 函数，或者是一条常量表达式
  - `constexpr` 构造函数用于生成 `constexpr` 对象及 `constexpr` 函数的参数或返回类型

类的静态成员

声明

- 在类成员声明(数据成员或函数）之前加上 `static` 使其与类关联在一起
- 类的静态数据成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据

  静态成员函数不包含 `this` 指针，故不能声明为 `const` 的，不能在其中使用 `this` 指针

定义

- 静态数据成员不能在类中定义，可以在类外定义；

  一个静态数据成员只能定义一次

```
static int b=1;          // 错误，不能在类的定义内初始化

int 类名::b=1;          // 正确
```

> 因为静态成员属于整个类，而不属于某个对象，如果在类内初始化，会导致每个对象都包含该静态成员，这是矛盾的。

> 因为实际上在类中是不能进行`=`操作的，我们可以写成那样是进行了隐式构造函数初始化。而静态对象根本不属于类，所以也就不是由构造函数初始化的。

- 不能通过类对象访问静态数据成员，来改变其值

  > 因为静态数据成员根本不存在于任何对象内，所以可以访问，但是不能修改

- 静态成员函数可在类内或类外定义；

  在类外定义时，不能重复`static`关键字

使用

- 通过作用域运算符直接访问
- 通过类对象访问

```
cout<<类名::b<<endl;
Debug a;
cout<a.b;
```

> 类内成员函数可以直接使用静态成员（访问或者修改均可）

静常量静态数据成员

静态常量数据成员可以在提供类内初始值

> 认为是一种定义了，所以可以通过`=`来进行赋值操作
>
> 一般来说，即使常量静态数据成员在类内部被初始化了，通常在外面也应该给出定义语句(但是不能赋值)

注意

- 静态数据成员可以是**不完全类型**
  - 不完全类型：已经声明但尚未定义的类型（编译器编译该句时，不知道其大小是多少）

    不能用于定义变量或者类的成员，但用其定义指针或引用是合法的

```
class Bar{
    public:
    private:
        static Bar mem1;    // 正确
        Bar *mem2;      // 正确
        Bar mem3;       // 错误
}
```

> 类的静态成员，不是类对象的成员，不是类对象的一部分，所以不存在不能定义的问题

- 可以用静态数据成员做默认实参

  > 非静态数据成员不能作默认实参，因为它的值本身属于对象的一部分。
  >
  > 在编译器期间，默认实参就应该被确认，但是非static成员在此时还没被初始化，所以不行