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、负责接口的实现的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
- 类要想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&);
```

- 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

> 直接定义在类内的任何函数是隐式inline函数。

- 类的非成员接口函数的声明一般和类的声明在同一个头文件中

引入this

- 成员函数有一个隐式this指针作为参数。

  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.

```
total.isbn();       // this指针指向total
```

- 在成员函数内部，我们可以直接调用this所指对象的成员，而无须通过成员运算符。

  任何对类成员的直接访问都被看作this的隐式引用

```
... isbn() {
    bookN0 = 1;     // 相当于 this->bookNo = 1;
}
```

引入const成员函数

```
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 *`

  > 此时，无法将常量对象绑定的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**

  > 因为这个const是一个底层const，不能被转化

  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read和input函数

```
istream &read(istream &is,Sales_data & item){
    ...;
    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是一个标准输入

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，此时不会默认生成该函数。
>
>   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

访问说明符

- `public`：成员都可以被公开访问，即无论类内还是类外都能访问。
- `protected`：成员可以被【该类成员】，【友元】和【派生类成员】访问，但类外不能访问。
- `private`：成员只能被【该类成员】或【友元成员】访问，不能从类外或派生类访问

每个说明符的作用域在类内从名字向下，直到遇到新的说明符为止

> 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。

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;     // 普通函数友元
    friend class 类名;         // 友元类
    friend void [函数所属类]::clear(ScreenIndxe);         // 成员函数友元
}
```

- 成员函数友元：
  - 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
  - 这类友元声明要**注意顺序**：

    类定义和对应成员函数的声明应该在友元声明的出现，成员函数的定义要在友元声明的后面

    > （假设类A，B，A中函数a是B的友元）
    >
    > - 成员函数a要指明所属类A，这时A必须有定义才行，所以一定要在B前面出现过A定义才行；
    > - 而a需要要到类B的成员，所以a的定义在B定义的后面

- 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，使得友元对用户可将

- 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。
- **友元本身不一定真的声明在当前作用域中**

> `friend`声明的作用在于为该类声明一个友元，并不能决定该对象是否存在。
>
> 所以如果要使用被`friend`声明的函数，必须在该函数真正声明之后才行。一旦声明出现，则友元处的定义也有效

```
struct x{
friend void f(){          // 即使之前没有出现f()声明
  /* 友元函数可以定义在内部 */
 }
    x(){f();}         // 错误，f还没正式声明
}
void x::g(){f();}    // 错误
void f();    // 出现声明了，友元内定义也生效了
void x::h(){f();}    // 正确
```

类的其他特性

类的声明

```
class Screen;
```

- **前向声明**：只声明Screen是一种类类型，但不清楚具体有那些成员

  在声明之后 定义之前这个阶段，可以称Screen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

> 可以减少编译依赖——《Effective C++》

- 创建类的对象时，该类必须被定义

类成员再探

- 在类中，仍然可以使用类型别名（存在访问限制）

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

- `mutable关键字`：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

```
mutable size_t num;      // 给num加以限制
void some_member() const{
    ++num;       // 尽管const * ，但是被mutable修饰的成员变量仍然可变，而其他的则不可变
}
```

> 一般是类中记录性质的成员，这样类对象仍然可以以const对象调用成员函数，其他数据成员就不允许被改动

- 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

从const成员函数返回*this

```
Stu t() const{       // 没问题
    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
}
```

> 建议对于公共代码部分定义一个private的公共函数
>
> - 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

类的作用域

> 一个类就是一个作用域，所以在类外定义成员函数必须同时提供类名和函数名

名字查找和类的作用域

- 类中的名字查找的顺序：**先编译类中全部声明（类型声明和函数声明），直到所有都可见后再编译成员函数体**

> - 对于声明中使用的名字，必须确保使用前可见
> - 成员函数体**内**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

难点探究

- 定义一个类时，不会给该类分配空间

> 类是一个抽象的概念，并不是一个实体，比如说class Person表示的是人，它有属性name，由于Person是个抽象的概念，所以没法初始化，只有具体到某个人的时候，这个时候他的属性是确定，所以可以根据属性来初始化。

- 类中，不能对变量进行赋值操作

> 定义类中变量并没有分配内存给他
>
> 之所以C++11支持=操作，也是由默认构造函数实现的

构造函数

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

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
```

> 如果自定义了构造函数，则如果想要支持默认构造函数，则必须使用`=default`形式

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，之后在函数体中指执行赋值操作

```
class test{
test(int t){     // 实中间调用默认构造函数
  a=t;
     b=t;        // 错误，不能给const赋值
     c=t;        // 错误，b没有被初始化
 }
private:
    int a;
    int &b;
    const int c;
}
// 正确,显示初始化引用和const对象
test(int t):a(t),b(t),c(t){}
```

- 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；

  实际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；

```
Sales_data() : Sales_data("",0,0){}
```

- 成员初始化列表只能包含一个委托构造函数，不能再包含其它成员变量的初始化。

默认构造函数

编译器会构造默认构造函数，**该函数无须任何参数**

- 如果类内成员有类内初始值，则用该值初始化

  > 使用类内初始值可以确保对象能被赋予一个正确的值

- 否则进行默认初始化成员

> - 一旦存在用户自定义的构造函数，则编译器不会自动为类生成默认构造函数
> - 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，且被包含的类中没有默认构造函数，此时包含类的默认构造函数无法初始化被包含类，就不会生成默认构造函数

```
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;
}
```

- **注意：** 调用默认构造函数不需要添加括号

```
Test t;     // t是一个对象
Test t();   // t是一个函数
```

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
        ...;
    }
    test combie(test t,int t2){
        ...;
    }
    int a=1;
};

```

- 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

    > 建议对单参数构造函数设置explicit

  - 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该构造函数

```
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
```

> - 执行拷贝形式的初始化将创建临时的test从a(隐式转换），然后t调用拷贝构造函数
> - 编译器可以在执行过程中优化，直接调用一般构造函数

```
struct A
{
   int i;
        A(int i) : i(i) { std::cout << " A(int i)" << std::endl; }
   private:
        A(const A &a)  {  std::cout << " A(const A &)" << std::endl; }
};

int main() {
         A a = 10;       // 编译报错；但是，如果删除private,又只会输出A(int i)---执行优化结果
}
```

> 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

聚合类的初始化方式

聚合类需要满足的条件：

- 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

> struct是一个聚合类

特性：

> 可以使用花括号初始化

- 花括号内的值只能少，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 t3={19,"syy"};
```

字面值常量类

- 字面值常量类需要满足的条件：
  - 数据成员必须是字面值类型[1]

    > 确保编译时求值

  - 类必须至少有一个 `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

  - 类必须使用析构函数的默认定义

    > 保证析构函数没有不能预期的操作

  > 特例：数据成员都是字面值类型的聚合类是字面值常量类

- `constexpr` 构造函数：

  除了声明为 `=default` 或 `=delete` 外，相当于【构造函数】+【constexpr函数】

  > 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

- **静态成员函数**可以在类内或类外定义；
  - 在类外定义时，不能重复static关键字
- **静态数据成员**不能在类中定义，可以在类外定义；

> 静态常量数据成员可以提供类内初始值，初始值必须是常量表达式。
>
> 可以理解：编译时就可以得到值+const限制使得编译器可以直接赋值，而不会调用构造函数赋值

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

使用

- 通过作用域运算符直接访问

```
类名::静态成员名
```

- 通过类对象访问

```
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）

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，数据成员必须是完全类型
}
```

> 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

术语

构造函数初始值列表：在构造函数体执行之前首先用初始值列表中的值初始化数据成员

转换构造函数：可以用一个实参调用的non-explicit构造函数

封装：分离类的实现和接口，从而隐藏了类的实现细节。

- C++中，通过把实现部分设为private来完成封装任务

接口：类型提供的public操作

- 通常情况下，接口不包含数据成员

前向声明：对尚未定义的名字声明

---

1. 算术类型、引用和指针都属于字面值类型，还有字面值类 ↩