类

- C++ Primer第五版
- 第7章

类的基本思想是数据抽象和封装

定义抽象数据类型

设计Sales_data类

- 一个isbn成员函数，用于返回对象的ISBN编号
- 一个combine成员函数，用户一个Sales_data对象加到另一个对象上
- 一个名为add的函数，指向两个Sales_data对象的加法
- 一个read函数，将数据从istream读入到Sales_data对象中
- 一个print函数，将Sales_data对象的值输出到ostream

```
Sales_data total; //保持当前求和结果的变量
if(read(cin,total)){ //读入第一笔交易
      Sales_data trans; //保持下一条交易数据的变量
      while( read(cin,trans)){
            if(total.isbn() == trans.isbn()) //检查isbn
                  total.combine(trans); //更新变量total当前的值
            else{
                  print(cout,total)<<endl;//输出结果
                  total = trans; //处理下一种书
            }
      }
      print(cout,total)<<endl; //输出最后一条交易
}else{//没有输入任何信息
     cerr<<"No data?!"<<endl;
}
```

类的用户，即类的使用者，也是开发人员，包括类的开发人员本人。但设计类的接口时，需要假设类的用户对类的细节并不知情。

成员函数的声明必须在类的内部，定义可以在类内部也可以在类外部。

```
struct Sales_data{
      //关于Sales_data对象的操作
      std::string isbn() const {return bookNo;}
      Sales_data& combine(const Sales_data&);
      double avg_price() const;
      std::string bookNo;
      unsgined units_sold = 0;
      double revenue = 0.0;
};
//非成员接口函数
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&,const Sales_data&);
std::istream &read(std::istream&,Sales_data&);
```

常量成员函数：类的成员函数后面加 const，表明这个函数不会修改这个类对象的数据成员

等价于 ：return this->bookNo;
this是一个常量指针，不允许改变this中保存的对象
this的类型是（Sales_data * const ）

指针常量: const int *PtrConst;
常量指针: int *const ConstPtr=&a; //必须初始化

```
//伪代码，说明隐式的this指针是如何使用的
//下面的代码是非法的：因为我们不能显式地定义自己的this指针
std::string Sales_data::isbn(const Sales_data *const this)
{return this->isbn;}
```

类作用域和成员函数

成员体可以随意使用类中的其他成员而无需在意这些成员出现的次序。

```
//在类的外部定义成员函数
double Sales_data::avg_price() const{ //使用了域作用运算符
    if(units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

编译器分两步处理类：首先编译成员的声明，然后才轮到成员函数体。

定义一个返回this对象的函数

```
//模拟符合运算符+=，为了和+=一致，返回为左值
Sales_data& Sales_data::combine(const Sales_data &rhs)//right hand side
{
    units_sold += rhs.units_sold; //把rhs的成员加到this对象成员上
    revenue += rhs.revenue;
    return *this; //返回调用该函数的对象
}
```

定义类相关的非成员函数

```
//输入的交易信息包括ISBN、售出总数和售出价格
istream &read(istream &is, Sales_data &item)
{
    double pricce = 0;
    is >> item.bookNo >> item.units_sold >> price;
    item.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;
}

Sales_data add(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; //把lhs的数据成员拷贝给sum
    sum.combine(rhs);//把rhs的数据成语加到sum当中
    return sum;
}
```

如果非成员函数是类接口的组成部分，则应该与类在同一个头文件中声明

构造函数

- 构造函数与类名同名，它的任务是初始化类对象的数据成员
- 类可以包括多个构造函数，和其他重载函数差不多
- 构造函数不能被声明为const的
  - 直到构造函数完成初始化，对象才能真正得到“常量”属性。因此构造函数在const对象的构造过程中可以向其写值。

合成的默认构造函数：编译器创建的构造

如果存在类内的初始值，用它来初始化成员
否则，默认初始化该成员

```
Sales_data total; //保持当前求和结果的变量
Sales_data trans; //保持下一条交易数据的变量
```

类内初始值：初始化数据成员

只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数。

定义Sales_data的构造函数

这里default有效，因为我们为内置类型的数据提供了初始值

```
struct Sales_data{
    //新增的构造函数
    Sales_data() = default; //C++ 11要求编译器生成构造函数 相当于默认的构造函 数
    //冒号和花括号之间是构造函数初始化列表
    Sales_data(const std::string &s):bookNo(s){ } ;
    Sales_data( csont std::string &s, unsgined n, double p):
        bookNo(s),units_sold(n), revenue(p*n) {}
    Sales_data(std::istream &);
    //之前已有的其他成员函数
    std::string isbn() const {return bookNo;}
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    std::string bookNo;
    unsgined units_sold = 0;
    double revenue = 0.0;
}

Sales_data::Sales_data(std::istream &is)
{
    read(is,*this);//从is中读取一条交易信息然后存入this对象中
}
```

拷贝、赋值和析构

```
total = trans; //它的行为与下面的代码相同

//Sales_data的默认赋值操作等价于
total.bookNo = trans.bookNo;
total.units_sold = trans.units_sold;
total.revenue = trans.revenue;
```

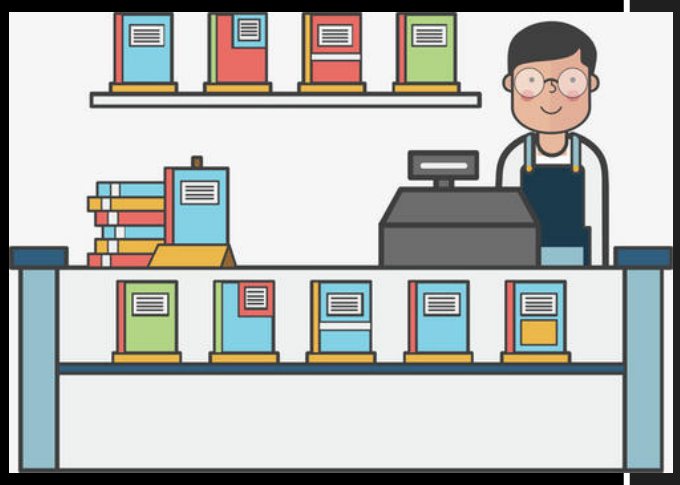
管理动态内存的类通常不能依赖于编译器合成的版本。使用vector或string除外

访问控制与封装

使用访问说明符加强类的封装性

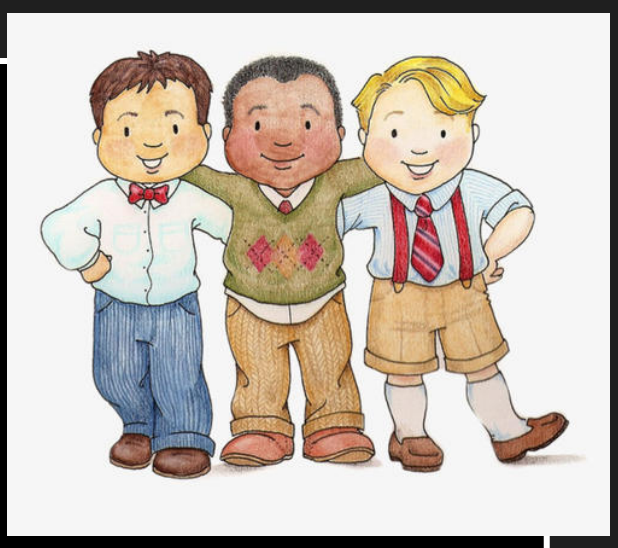
- **public**：类的接口，在整个程序内可以被访问。
- **private**：封装（即隐藏）类的实现细节。

使用class和struct定义类唯一的区别就是默认的访问权限。



```
class Sales_data{
public://添加了访问说明符
      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 &conmbine(const Sales_data&);
private://添加了访问说明符
      double avg_price() const { return units_sold?revene/units_sold:0;}
      std::string bookNo;
      unsigned units_sold = 0;
      double revenue =0.0;
}
```

友元：允许其他类或函数访问自己的非公有成员



```
class Sales_data{
//为Sales_data的非成员函数所做的友元声明
friend Sales_data add(const Sales_data&, const Sales_data&);
friend std::ostream &print(std::ostream&,const Sales_data&);
friend std::istream &read(std::istream&,Sales_data&);
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 &conmbine(const Sales_data&);
private:
      double avg_price() const  { return units_sold?revene/units_sold:0;}
      std::string bookNo;
      unsigned units_sold = 0;
      double revenue =0.0;
}
//非成员接口函数
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&,const Sales_data&);
std::istream &read(std::istream&,Sales_data&);
```

类的其他特性

@阿西拜-南昌

定义一个类型成员

```
//Screen表示显示器中的一个窗口
class Screen{
public: //类定义的类型名称和其他成员一样存在访问限制
      typedef std::string::size_type pos;
      //using pos = std::string::size_type;
private:
      pos cursor = 0;
      pos height = 0, width = 0;
      std::string contents;
};
```

Screen的用户不需要知道Screen使用了一个string对象来存放它的数据，pos隐藏了细节

Screen类的成员函数

```
//Screen表示显示器中的一个窗口
class Screen{
public:
      typedef std::string::size_type pos;

      Screen()=default; //因为Screen有另一个构造函数，所以不能少
      //cursor被其类内初始值初始化为0
      Screen(pos ht,pos wd,char c): height(ht),width(wd),contents(ht*wd,c) {}

      //读取光标处的字符
      char get() const { return contents[cursor];}          //隐式内联
      inline char get(pos ht,pos wd) const;                 //显示内联
      Screen &move(pos r, pos c);                           //能在之后被设为内联
private:
      pos cursor = 0;
      pos height = 0, width = 0;
      std::string contents;
};

inline Screen &Screen::move(pos r,pos c)
{
      pos row = r*width;      //计算行的位置
      cursor = row + c;       //在行内将光标移动到自定的列
      return *this;           //以左值的形式返回对象
}

char Screen::get(pos r, pos c) const //在类的内部声明成inline
{
      pos row = r*width;                //计算行的位置
      return contents[row + c];         //返回改定列的字符
}
```

定义在里面才是默认inline

这个里面只有声明，没定义
所以在外部定义 如果需要inline
的话 需要加上 inline 关键字才行
否则就是普通的函数

重载成员函数

成员函数也可以被重载

```
Screen myscreen;
char ch = myscreen.get(); //调用Screen::get()
ch = myscreen.get(0,0); //调用Screen::get(pos, pos)
```

可变数据成员

```
class Screen{
public:
    void some_member() const;
private:
    mutable size_t access_ctr; //即使在一个const对象内也能被修改
    //其他成员与之前的版本一致
};

void Screen::some_member() const
{
    ++access_ctr;  //保持一个计数值，用于记录成员函数被调用的次数
    //该成员需要完成的其他工作
}
```

类数据成员的初始化

```
class Window_mgr{
private:
    //这个窗口管理类，管理一组screen
    //默认情况下，一个Window_mgr包含一个标准尺寸的空白Screen
    std::vector<Screen> screens {Screen(24,80,' ')};
}
```

当提供一个类初始值时，必须以符号 = 或者花括号表示。

返回*this的成员函数

```
class Screen{
public:
	Screen &set (char);
	Screen &set (pos, pos, char);
	//其他成员和之前的版本一致
};

inline Screen &Screen::set(char c)
{
	contents[cursor] = c; //设置当前光标所在位置的新值
	return *this; //将this对象作为左值返回
}
inline Screen &Screen::set(pos r, pos col, char ch)
{
	contents[r*width + col] = ch; //设置给定位置的新值
	return *this; //将this对象作为左值返回
}

//把光标移动到一个指定的位置，然后设置该位置的字符值
myScreen.move(4,0).set('#');
//myScreen.move(4,0);
//myScreen.set('#');
//如果move返回Screen而非Screen&
Screen temp = myScreen.move(4,0); //对返回值进行拷贝
temp.set('#'); //不会改变myScreen的contents
```

从const成员函数返回*this

```
//添加一个diplay操作，打印Screen的内容
//我们希望函数能和move以及set出现在同一序列中
//diplay不需要改变对象的内容，所以我们定义为const成员函数。
Screen myScreen;
//如果display返回常量引用，则调用set将引发错误
myScreen.display(cout).set('*');

class Screen{
public:
      //根据对象是否是const重载了display函数
      Screen &display(std::ostream &os)
      {
            do_display(os);
            return *this;
      }
      const Screen &display(std::ostream &os) const
      {
            do_display(os);
            return *this;
      }
private:
      //该函数负责显示Screen的内容
      void do_display(std::ostream &os) const{os<<contents;}
      //其他成员与之前的版本一致
}

Screen myScreen(5,3);
const Screen blank(5,3);
myScreen.set('#').display(cout); //调用非常量版本(set返回的是非常量引用)
blank.display(cout); //调用常量版本
```

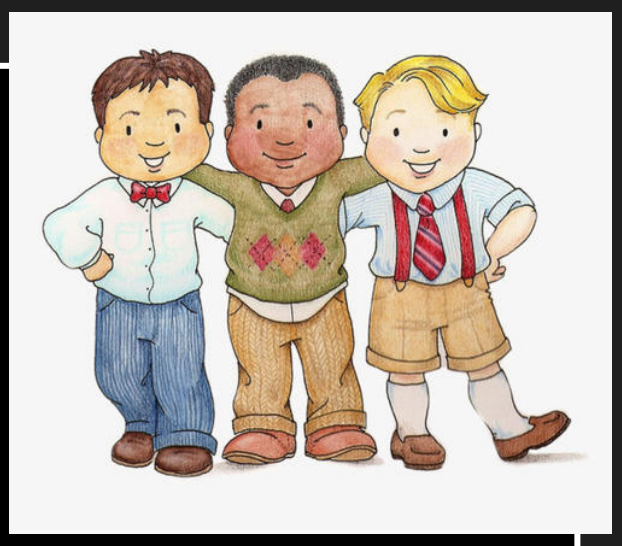
类类型

```
struct First{
        int memi;
        int getMem();
};

struct Second{
        int memi;
        int getMem();
};

First obj1;
Second obj2 = obj1; //错误：obj1和obj2的类型不同


//类的声明可以和定义分离
class Screen; //前向声明，在定义之前是一个不完整类型

class Link_screen{
        Screen window; //错误：不完整类型
        Link_screen *next; //正确：一旦类名字出现后，它就被认为是声明过了
        Link_screen *prev;
}
```

每个类定义了唯一的类型。

不完整类型可以定义指向该类型的指针或引用，也可以作为函数声明中的参数或返回类型

类之间的友元关系



每个类负责控制自己的友元类或友元函数

```
class Screen{
    //Window_mgr的成员可以访问Screen类的私有部分
    friend class Window_mgr;
    //Screen类的剩余部分
};

class Window_mgr{
public:
    //窗口中每个屏幕的编号
    using ScreenIndex = std::vector<Screen>::size_type;
    //按照编号将制定的Screen重置为空白
    void clear(ScreenIndex);
private:
    std::vector<Screen> screens{Screen(24,80,' ')};
};

void Window_mgr::clear(ScreenIndex i)
{
    //s是一个Screen的引用，指向我们想清空的那个屏幕
    Screen &s = screens[i];
    //将那个选定的Screen重置为空白
    s.contents = string(s.height*s.width, ' ');
}
```

令成员函数作为友元

```
class Screen{
    //Window_mgr::clear必须在Screen类之前被声明
    friend void Window_mgr::clear(ScreenIndex);
    //Screen类的剩余部分
};
//1、定义Window_mgr类，声明clear函数，但不能定义它
//2、定义Screen,包括对于clear的友元声明
//3、定义clear，此时才能使用Screen的成员
```

作用域名字查找

首先，在名字所在的块中寻找其声明语句，只考虑在名字的使用之前出现的声明。

· 如果没找到，继续查找外层作用域。

· 如果最终没有找到匹配的声明，则程序报错。

对于定义在类内部的成员函数来说，解析其中名字的方式与上述的查找规则有所区别，不过在当前的这个例子中体现得不太明显。

编译器处理完类中的全部声明后才会处理成员函数的定义

类的作用域

```
Screen::pos ht = 24, wd = 80; //使用Screen定义的pos类型
Screen scr(ht, wd, ' ');
Screen *p = &scr;
char c = scr.get(); //访问scr对象的get成员
c =  p->get();  //访问所指对象的get成员

void Window_mgr::clear/*一旦遇到类名*/(ScreenIndex i)
{
      Screen &s = screens[i];
      s.contents = string(s.height*s.width, ' ');
      //直到定义的结束，都是类的作用域之内
}

class Window_mgr{
public:
      //项窗口添加一个Screen,返回它的编号
      ScreenIndex addScreen(const Screen&);
      //其他成语于之前的版本一致
};
//首先处理返回类型，之后我们才进入Window_mgr的作用域
Window_mrg::ScreenIndex Window_mgr::addScreen(const Screen &s)
{
      screens.push_back(s);
      return screens.size()-1;
}

typedef double Money;
class Account{
public:
      Money balance() {return bal;} //使用外层作用域的Money
private:
      typedef double Money; //错误：重复定义，编译器并不负责
      Money bal;
      //...
};
```

隐式使用了this指

一个类就是一个作用域

返回类型必须指名它是哪个类的

类型名的定义应该放在类的开始处

小心使用　编译器可能不会报错

成员定义中的普通块作用域的名字查找

```
//注意：这段代码仅为了说明而用，不是一段很好的代码
//通常情况下不建议为参数和成员使用同样的名字
int height; //定义了一个名字，稍后将在Screen中使用
class Screen{
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height) {
        cursor = width * height; //哪个height？是那个参数
    }
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};

//不建议的写法，成员函数中的名字不应该隐藏同名的成员
void Screen::dummy_fcn(pos height){
    cursor = width * this->height; //成员height
    //另外一种表示该成员的方法
    cursor = width * Screen::height; //成员height
    cursor = width * ::height; //全局height
}
```

全局

尽管外层的对象被隐藏掉了，但我们仍然可以用作用域运算符访问它

```
int height; //定义了一个名字，稍后将在Screen中使用
class Screen{
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0; //隐藏了外层作用域中的height
};

Screen::pos verify(Screen::pos);
void Screen::setHeight(pos var){
    //var：参数
    //height:类的成员
    //verify:全局函数
    height = verify(var);//可以：verify声明在前
}
```

构造函数再探

构造函数初始值列表


```
Sales_data::Sales_data(const string &s, unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
```

这个是赋值
不是初始化
写初始化列表好

```
string foo = "Hello World!"; //定义并初始化
string bar; //默认初始化成空string对象
bar = "Hello World!"; //为bar赋一个新值
```

```
//有时候初始化列表必不可少
class ConstRef{
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
ConstRef::ConstRef(int ii) //错误：ci和ri必须被初始化
{//赋值：
    i = ii; //正确
    ci = ii; //错误
    ri = i; //错误：ri没被初始化
}
//正确：显示地初始化引用和const成员
ConstRef::ConstRef(int ii):i(ii),ci(ii),ri(i){ }
//成员初始化的顺序
class X{
    int i;
    int j;
public:
    //未定义的：i在j之前被初始化
    X(int val):j(val),i(j) { }
    //尽量使用参数作为初始化值
    //X(int val):j(val),i(val) { }
    //这样就与i和j初始化的顺序无关了
};

//默认实参和构造函数
class Sales_data{
public:
    //定义默认构造函数，令其与只接受一个string实参的构造函数功能相同
    Sales_data(std::string s = ""):bookNo(s) { }
    //...
};
```

如果成员是const、引用、或者属于某种未提供默认构造函数的类类型，必须通过构造函数初始列表提供初始值

改用 初始化列表

构造函数初始值列表中的顺序，不会影响实际的初始化顺序

构造函数初始化顺序与成员声明的顺序相同

一个构造函数为所有参数都提供了默认实参，也就定义了默认构造函数

委托构造函数

```
class Sales_data{
public:
    //非委托构造函数使用对应的实参初始化成员
    Sales_data(std::string s, unsigned cnt, double price):
        bookNo(s),units_sold(cnt),revenue(cnt*price) { }
    //其余构造函数全部委托给另一个构造函数
    Sales_data():Sales_data("",0,0) { }
    Sales_data(std::string s):Sales_data(s,0,0){ }
    Sales_data(std::istream &is):Sales_data() {read(is,*this)}
    //...
};
```

把自己的一些（或全部）职责给了其他构造函数

受委托的构造函数初始化列表和函数体都会被执行，然后如果委托函数的函数体不为空，则再执行委托函数的函数体.

默认构造函数的作用

```
class NoDefault {
public:
    NoDefault(const std::string&);
    //还有其他成员，但是没有其他构造函数了
};
struct A { //默认情况下my_mem是public的
    NoDefault my_mem;
};
A a; //错误：不能为A合成构造函数
struct B {
    B(){} //错误：b_member没有初始值
    NoDefault b_member;
};
```

如果定义了其他构造函数，最好也提供一个默认构造函数

定义了其他构造函数，
但是没有默认构造函数，
倒置A 错误

```
Sales_data obj(); //正确：定义了一个函数而非对象使用默认构造函数  不能写( )
if (obj.isbn == Primer_5th_ed.isbn()) //错误：obj是一个函数          不然就成函数了
```

隐式的类类型转换

```
string null_book = "9-999-99999-9";
//构造一个零时的Sales_data对象
//该对象的units_sold和revenue等于0，bookNo等于null_book
item.combine(null_book);

//错误：需要用于定义的两种转换：
//（1）把“9-999-99999-9”转换成string
//（2）再把这个（临时的）string转换成Sales_data
item.combine("9-999-99999-9");

//正确：显式地转换成string，隐式的转换成Sales_data
item.combine(string("9-999-99999-9"));
//正确：隐式的转换成string，显式地转换成Sales_data
item.combine(Sales_data("9-999-99999-9"));

//使用istream构造函数创建一个函数传递给combine
item.combine(cin);
```

能通过一个实参调用的构造函数定义了一条从构造函数的参数类型向类类型隐式转换的规则

只允许一步类类型转换

抑制构造函数定义的隐式转换：explicit（清楚、明白的）

```
class Sales_data{
public:
    Sales_data()=default;
    Sales_data(const std::string &s, unisgned n, double p):
        bookNo(s),units_sold(n),revenue(p*n){ }
    explicit Sales_data(const std::string &s):bookNo(s) { }
    explicit Sales_data(std::istream&);
    //...
};

item.combine(null_book); //错误：必须是explicit的
item.combine(cin); //错误：必须是explicit的

//错误：explicit关键字只允许出现在类内的构造函数声明处
explicit Sales_data::Sales_data(istream& is){ read(is, *this); }

Sales_data item1(null_book); //正确：直接初始化
//错误：不能将explicit构造函数用于拷贝形式的初始化过程
Sales_data item2 = null_book;

//正确：实参是一个显式构造的Sales_data对象
item.combine(Sales_data(null_book));
//正确：static_cast可以使用explicit的构造函数
item.combine(static_cast<Sales_data>(cin));
```

explicit构造函数只能用于直接初始化

除了底层const 不行
其他都能强制类型转换

聚合类：

```
struct Data{
        int ival;          初始化列表的顺序应该与它
        string s;          声明时的顺序相同
};

//vali.ival=0;val1.s=string("Anna")
Data ival1 = { 0, "Anna" }; //可以使用初始值列表

//错误：不能使用 “Anna”初始化ival，也不能使用1024初始化s
Data val2 = {"Anna",1024};
//副作用：添加或删除一个成员之后，所有的初始化语句都需要更新
```

- 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

字面值常量类：

- 数据成员必须都是字面值类型
- 类必须至少有一个constexpr构造函数
- 数据成员的类内初始值
  - 内置类型：必须是一条常量表达式
  - 类类型：使用成员自己的constexpr构造函数
- 类必须使用析构函数的默认定义

```
class Debug {
public:
        constexpr Debug(bool b = true):hw(b),io(b),other(b) { }
        constexpr Debug(bool h, bool i, bool o):hw(h),io(i),other(o) { }
        constexpr bool any() {return hw || io || other;}
        void set_io(bool b) { io = b; }
        void set_hw(bool b) { hw = b; }
        void set_other(bool b) { hw = b; }
private:
        bool  hw;  //硬件错误，而非IO错误
        bool io; //IO错误
        bool other; //其他错误
};

constexpr Debug io_sub(false, true, false); //调试IO
if(io_sub.any()) //等价于if(true)
        cerr<<"print appropriate error messages"<<endl;
constexpr Debug prod(false); //无调试
if (prod.any()) //等价于if(false)
        cerr<<"print an error message"<<endl;
```

- 算数类型、引用、指针是字面值类型

```
constexpr int a = 0； //算数类型int是字面值类型；
```

- constexpre构造函数体一般来说应该是空的
  - constexpr函数只能有返回语句
  - 构造函数不能有返回语句

类的静态成员

与类本身关联，而不需要与每个对象关联（例如：利率）


```
class Account {
public:
      void calculate() { amount += amount * interestRate; }
      static double rate() { return interestRate; }
      static void rate(double);
private:
      std::string owner;
      double amount;
      static double interestRate;
      static double initRate();
};

double r;
r = Account::rate(); //使用作用域运算符访问静态成员
//虽然不属于某一个对象，但可以通过对象访问
Account ac1;
Account *ac2 = &ac1;
//调用静态成员函数rate的等价形式
r = ac1.rate(); //通过Account的对象或引用
r = ac2->rate(); //通过指向Account对象的指针

//不能重复static关键字
void Account::rate(double newRate)
{   外部定义 不需要static 关键字
      interestRate = newRate;
}

//类似于全局变量，静态成员定义在任何函数之外
//定义并初始化一个静态成员
double Account::interestRate = initRate();//initRate使用时在类的作用域之内
```

static函数不包含this指针，所以不能定义为const函数

静态成员存在于任何对象之外，所有对象共享

可以使用类的对象、引用or指针来访问静态成员

static关键字值出现在类内部的声明语句中

静态成员

静态成员的类内初始化


```
class Account {
public:
        static double rate() { return interestRate; }
        static void rate(double);
private:
        static constexpr int period = 30; //period是常量表达式
        double daily_tbl[period];
};

//一个不带初始值的静态成员的定义
constexpr int Account::period; //初始值在类的定义内提供
//即使一个常量静态数据成员在类内部被初始化了，
//通常情况下也应该在类的外部定义一下该成员
```

通常，类的静态成员不该在类的内部初始化

静态成员能用于某些场景，而普通成员不能

```
class Bar {                不完全类 型：只是声明了类 但是
public:                    这个类并没有定义
        //...
private:
        static Bar meml; //正确：静态成员可以是不完全类型
        Bar *mem2;   正确 ：指针成员也可以是不完全类型
        Bar mem3;  //错误：数据成员必须是完全类型
};

class Screen {
public:
        //bkground表示一个在类中稍后定义的静态成员
        Screen& clear(char = bkground);
private:
        static const char bkground;
};
```

静态数据类型可以是不完全类型

可以使用静态成员作为默认实参，
因为它本身并不是对象的一部分

而非静态成员就不能作为默认实参