第7章 类

1. 类的基本思想是：*数据抽象 data abstraction* 和 *封装 encapsulation*。
2. *数据抽象* 是一种依赖于 *接口 interface* 和 *实现 implementation* 分离的编程（以及设计）技术。
3. 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、负责接口实现的函数体以及定义类所需的各种私有函数。
4. *封装* 实现了类的接口和实现的分离。封装后的类隐藏了它的实现细节，类的用户只能使用接口而无法访问实现部分。

7.1 定义抽象数据类型

7.1.2 定义改进的 `Sales_data` 类

1. 成员函数通过一个名为 `this` 的额外隐式参数来访问调用它的那个对象。当调用一个成员函数时，用请求该函数的对象地址初始化 `this` 。

```
struct Sales_data {
    std::string isbn() const {return bookNo;}   // 隐式inline
    std::string bookNo;
}

Sales_data total;
total.isbn();

// 编译器负责把 total 的地址传递给 isbn() 的隐式形参 this
// 可以等价地认为编译器将该调用重写成了：
Sales_data::isbn(&total);   // 伪代码
```

2. 任何自定义名为 `this` 的参数或变量的行为都是非法的。
3. `this` 是个常量指针，初始化该指针之后，不再允许改变 `this` 中保存的地址。例如在 `Sales_data` 成员函数中， `this` 的类型是 `Sales_data *const` （顶层 `const` ）。
4. `this` 是隐式的而且不会出现在参数列表中，所以在哪里将 `this` 声明成指向常量的指针就是我们需要面对的问题。C++语言的做法是允许把 `const` 关键字放在成员函数的参数列表之后，此时 `this` 是一个指向常量对象的常量指针（ `const Sales_data *const this` ），向这样使用 `const` 的成员函数被称作 *常量成员函数*。

```
// 伪代码:
std::string Sales_data::isbn(const Sales_data *const this) {
    return this->bookNo;
}
```

7.1.3 定义类相关的非成员函数

1. IO类属于不能被拷贝的类型，因此我们只能通过引用来传递它们。

7.2 访问控制与封装

1. 使用 `class` 和 `struct` 定义类的唯一的区别就是默认的访问权限。

7.2.1 友元

1. 类可以允许其他类或者函数访问它的非公有成员，方式是令其他类或者函数成为它的 *友元 friend*
2. 封装有两个重要的优点：
   · 确保用户代码不会无意间破坏封装对象的状态。
   · 被封装的类的具体实现细节可以随时改变，而无须调整用户级别的代码

7.3 类的其他特性

7.3.1 类成员再探

1. 一个 *可变数据成员 mutable data member* （通过在变量的声明中加入 `mutable` 关键字）永远不会是 `const`， 即使它是 `const` 对象的成员。因此，一个 `const` 成员函数可以改变一个可变成员的值。
2. 当提供一个类内初始值时，必须以符号 `=` 或者 `{}` 表示。

7.3.3 类类型

1. 因为只有当类全部完成后类才算被定义，所以一个类的成员类型不能是该类自己。然而，一旦一个类的名字出现后，它就被认为是声明过了（但尚未定义），因此类允许包含指向它自身类型的引用或指针。

7.3.4 友元再探

1. 友元关系不具有传递性。
2. 如果一个类想把一组重载函数声明成它的友元，它需要对这组函数中的每一个函数分别声明。

7.4 类的作用域

1. 当我们在类的外部定义成员函数时，必须同时提供类名和函数名。一旦遇到了类名，定义的剩余部分就在类的作用域之内了，这里的剩余部分包括参数列表和函数体，此时我们可以直接使用类的其他成员而无需再次授权。但是函数的返回类型通常出现在函数名的前面，此时返回类型中使用的名字都位于类的作用域之外，所以必须声明它是哪个类的成员。

```
class Screen{}

class Window_mgr {
public:
    using ScreenIndex = std::vector<Screen>::size_type;
    ScreenIndex addScreen(const Screen&);
}

Window_mgr::ScreenIndex Window_mgr::addScreen(const Screen& s){}
```

7.4.1 名字查找与类的作用域

1. 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。
2. 一般来说，内层作用域可以重新定义外层作用域的名字，即使该名字已经在内层作用域中使用过。然而在类中，如果成员使用了外层作用域中的某个名字，而该名字代表一种类，则类不能在之后重新定义该名字：

```
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：不能重新定义 Money
    Money bal;
}
```

3. 即使 `Account` 中定义的 `Money` 类型与外层作用域一致，上述代码仍然是错误的。但是编译器并不为此负责，一些编译器仍将顺利通过这样的代码，而忽略代码错误的事实。

4. 类型名的定义通常出现在类的开始处，这样就能确保所有使用该类型的成员都出现在类名的定义之后。

7.5 构造函数再探

7.5.1 构造函数初始值列表

1. 成员的初始化顺序与它们在类定义中出现顺序一致。尽量避免使用某些成员初始化其他成员。

7.5.2 委托构造函数

1. C++11 新标准扩展了构造函数初始值的功能，使得我们可以定义所谓的 *委托构造函数 delegating constructor*。一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些（或者全部）职责委托给了其他构造函数。

```
class Sales_data {
public:
    // 非委托构造函数
    Sales_data(int a) {
        // 实现a
    }

    // 委托构造函数，该函数会先调用实现a，然后再调用实现b
    Sales_data(): Sales_data(0) {
        // 实现b
    }
}
```

7.5.4 隐式的类类型转换

1. 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，称为 *转换构造函数converting constructor*。
2. 编译器值会自动地执行一步类型转换。如该调用需类型转换多次，则应使用显示类型转换。
3. 使用 `explicit` 关键字声明构造函数时，该构造函数只能以直接初始化的形式使用，且编译器将不会在自动转换过程中使用该构造函数。

```
std::string str;
Sales_data item1(str);  // 正确：直接初始化
Sales_data item2 = str; // 错误，不能用于拷贝形式的初始化过程。
```

4. 关键字 `explicit` 只对一个实参的构造函数有效。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。
5. 尽管编译器不会将 `explicit` 的构造函数用于隐式转换过程，但是我们可以使用 `Sales_data(str)` 或者 `static_cast<Sales_data>(str)` 这两种方式显示地强制进行转换。

7.5.5 聚合类

1. *聚合类 aggregate class* 只含有公有成员的类，并且没有类内初始值或者构造函数。聚合类的成员可以用花括号括起来的初始值列表进行初始化。

7.5.7 字面值常量类

1. 数据成员都是字面值类型的聚合类是字面值常量类。
2. 如果一个类不是聚合类，但是它符合下述要求，它也是一个字面值常量类：
   · 数据成员都必须是字面值类型。
   · 类必须至少含有一个 `constexpr` 构造函数。
   · 如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一条常量表达式；或者如果成员属于某种类类型，则初始值必须使用成员自己的 `constexpr` 构造函数。
   · 类必须使用析构函数的默认定义，该成员负责销毁类的对象。
3. `constexpr` 构造函数体一般来说应该是空。且必须初始化所有数据成员。

```
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; }
private:
    bool hw;
    bool io;
    bool other;
}

constexpr Debug io_sub(false, true, false);
constexpr Debug prod(false);
io_sub.any();
prod.any();
```

7.6 类的静态成员

1. 有的时候类需要它的一些成员与类本身直接相关，而不是与类的各个对象保持关联。

2. 类的静态成员存在于任何对象之外，对象中不包含任何于静态数据成员有关的数据。类的静态成员对象只有一个，且被所有该类对象共享。
3. 静态成员函数也不于任何对象绑定在一起，它们不包含 `this` 指针。不能声明成 `const` 。这一限制既适用于 `this` 的显示使用，也对调用 *非静态成员的隐式使用* 有效。
4. 因为静态数据成员不属于类的任何一个对象，所以它们并不是在创建类的对象时被定义的。这意味着它们不是由类的构造函数初始化的，必须在类的外部定义和初始化每个静态成员，且一个静态数据成员只能定义一次。一旦它被定义，就将一直存在于程序的整个生命周期中。
5. 要想确保对象只定义一次，最好的办法是把静态数据成员的定义与其他非内联函数的定义放在同一个文件中。
6. 通常类的静态成员不应该在类的内部初始化。然而，我们可以为静态成员提供 `const` 整数类型的类内初始值，不过要求静态成员必须是字面值常量类型的 `constexpr` ：

```
class Account {
private:
    static constexpr int period = 30;
}

// 即使在类内部被初始化了，通常情况下也应该在类的外不定义一下该成员。
constexpr int Account::period;
```

7. 静态数据成员的类型可以就是它所属的类类型。而非静态数据成员则受到限制，只能声明成它所属类的指针或引用。
8. 可以使用静态数据成员作为成员函数的默认实参。非静态数据成员则不能，因为非静态数据成员本身属于对象的一部分，无法提供一个对象以便从中获取成员的值，最终将引发错误。