类

成员函数

1. 定义在类内部的函数都是隐式的inline函数。
2. 成员函数通过一个名为this的隐式参数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地址初始化this**。任何对类成员的直接访问都被看作this的隐式引用。
3. this指针总是指向一个对象，所以是一个常量指针
4. const成员函数：this是成员函数的**隐式形参**，这个形参指向的类型是非常量的，所以const成员函数的作用是修改隐式的this指针使其指向常量类型，因此在该函数内部无法改变调用它的对象的内容。

```
class Test{
    int ele=1;
    public:
        void func() const{ ele=2;}  //这是错误的 不可以修改ele的值
        //上面的成员函数定义相当于
        void func(const Test *const this) {ele=2};
};
```

5. 编译器分两步处理类：首先编译成员的声明，然后才编译成员函数体。因此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。
6. *this 得到的是this指针指向的对象的引用

构造函数

1. 若类含有const对象且没有在声明时初始化，则**必须在构造函数的初始值列表中初始化const对象**，而不能在函数体中初始化。这个初始值列表是专门用于初始化类的数据成员的。
2. 定义了一个含有参数的构造函数之后编译器将不再提供默认构造函数。
3. 如果类的成员是const或者是引用的话，就必须进行初始化。或者当成员属于某种类的类型且该类没有定义默认构造函数时，也必须将这个成员初始化
4. 在很多类中，初始化和赋值的区别关系到底层效率的问题。初始化是直接初始化成员，而赋值是先初始化再赋值。
5. **类的成员的初始化顺序与它们在类定义中出现的顺序一致**：第一个成员先被初始化，然后是第二个，以此类推。初始化列表中成员出现的位置顺序不会影响实际的初始化顺序。
6. 委托构造函数：C++11新标准，一个委托构造函数的成员初始值列表只有一个唯一的入口，就是其他的构造函数。即使用其他构造函数在初始化列表中初始化类成员。

7. 定义使用默认构造函数进行初始化的对象时不用加括号()，否则如下：

```
class Test{
    public:
        void func(){}
};
Test obj(); //实际上obj是一个不接受任何参数的函数并且返回类型为Test
obj.func(); //编译出错 因为obj是一个函数
```

8. 转换构造函数：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。**即定义了一条从构造函数的参数的类型 向 该类的类型 隐式转换的规则**。不过编译器只允许一步类类型转换，如果代码需要连续使用两种隐式转换规则，则是错误的。
9. 可以使用关键字**explicit**声明构造函数使其不能用于隐式转换。该关键字只对只有一个实参的构造函数有效，而且**只能在类内声明构造函数时使用explicit关键字，在类外部定义时不应重复**。此外，**不能将explicit构造函数用于拷贝形式的初始化过程**

访问控制与封装

1. class和struct的唯一区别是默认的访问权限
2. 类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数称为它的友元，采用friend关键字开始的函数声明。**友元声明只能出现在类定义的内部，但是出现的具体位置不限**。友元不是类的成员也不受它所在区域访问控制级别的约束。友元的声明仅仅指定了访问的权限而非一个通常意义上的函数声明，如果我们希望类的用户能够调用某个友元函数，就必须在友元声明之外再专门对函数进行一次声明。此外，友元函数也能定义在类内。

```
class Test{
    int ele;
    public:
        Test():ele(2){}
    friend void printTest(Test);
};
void printTest(Test t){
    cout<<t.ele;
}
```

3. 如果一个类指定了友元类，则**友元类的成员函数可以访问此类的所有成员，并不是说友元类对象可以调用该类的所有成员**。但是友元关系不具有传递性，即朋友的朋友不一定是朋友。
4. 令成员函数作为友元：假设要令A的成员函数func作为类B的友元
    - 首先定义类A，声明func函数但不能定义；
    - 定义类B，包括对A::func的友元声明；
    - 最后定义A::func函数，这样func函数才能使用类B的成员
5. 类与非成员函数的声明**不是必须在它们的友元声明之前**。当一个名字第一次出现在一个友元声明中时，我们隐式地假定该名字在当前作用域中是可见的，然后友元本身不一定真的声明在当前作用

域中。就算类的内部定义该函数，我们也必须在类的外部提供相应的声明从而使得函数可见。

```
struct Test{
    friend void f(){}   //在类内定义
    Test(){f();}    //这是错的，因为f还没有被声明
    void g();
    void h();
}
void Test::g(){f();}    //这是错的，因为f还没有被声明
void f();
void Test::h(){f();}    //这次就对了，f在此之前已经被声明
```

不过有的编译器并不强制执行上述关于友元的限定规则。

其他特性

1. 可变数据成员：mutable。用于声明一个避免的数据成员，该成员即使在const成员函数中也能被修改。可以用来追踪每个成员函数调用的次数。
2. 类的定义分两步处理：首先编译成员的声明，直到类全部可见后才编译函数体。
   而一般程序的名字查找过程是：
   - 首先在名字所在的块中寻找其声明语句，只考虑在名字使用前出现的声明；
   - 若无找到则继续查找外层作用域；
   - 若最终没有找到匹配的声明则程序报错。
3. 在类中，若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型，则类不能在之后重新定义该名字。如下：

   ```
   typedef double M;
   class Test{
       public:
           M func(){ return ele; }
       private:
           typedef double M;   //不能重新定义M
           M ele;
   };
   ```

   不过有些编译器不为此负责，仍可以正常通过。
4. 成员函数中使用的名字按照下面的方式解析：
   - 首先在成员函数内查找该名字的声明，只有在函数使用之前出现的声明才被考虑
   - 如果在成员函数内没有找到则在类内继续查找，可以考虑所有成员
   - 如果类内没有找到该名字的声明，则在成员函数定义之前的作用域内继续查找

   但如下的代码会报错：可见类型的重命名的查找规则有所不同

```
class Test{
    public:
        M func(){}
    private:
        typedef int M;
};
```

聚合类

1. 当一个类满足以下条件时称为聚合类：
    - 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。

    可以使用花括号括起来的成员初始值列表来初始化其数据成员，要按照成员声明的顺序初始化。

类的静态成员

1. 类的静态成员是**与类本身相关**，而不是与类的各个对象相关。类的静态成员存在于任何对象之外。类似的，静态成员函数也不与任何对象绑定在一起，它们**不包含this指针**。
2. 静态成员函数不能在函数体内使用this指针，而由于类内函数限定符const是用来限定this指针的，故静态成员函数也不能声明成const。因此静态成员函数体内也不可以直接操作类的数据成员，但能访问声明为static的数据成员。
3. 可以用类的作用域运算符访问静态成员，也可以通过类对象访问静态成员。
4. 在类外定义静态成员函数时不能重复static关键字。
5. 类的静态成员不可以在类的内部初始化。不过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常量类型的constexpr，且初始值必须是常量表达式。
6. 静态成员和非静态成员的不同限制：
    - **静态数据成员可以就是它所属的类的类型，但是非静态数据成员则只能声明为它所属的类的指针或者引用**。
    - 静态成员可以作为默认实参，而非静态成员不可以。