第五讲 类和对象

动态内存分配

动态内存分配具有以下特点：

➢ 不需要预先分配内存空间

➢ 分配的控件可根据程序需要扩大/缩小

➢ 动态内存分配的生存期由编程者决定

➢ 在堆中开辟，由程序员开辟和释放

new 的用法

当我们使用关键字 new 在堆上动态创建一个对象时，它实际上做了三件事：获得一块内存空间、调用构造函数、返回正确的指针。如果我们创建的是简单类型的变量，例如 int 或 float 类型，那么第二步会被省略。

使用 new 运算符时必须已知数据类型，new 运算符会向系统堆申请足够的存储空间。如果申请成功，就返回该内存块的首地址，如果申请不成功，则返回零值。其基本语法形式如下：

```
指针变量名=new 类型标识符;
或
指针变量名=new 类型标识符（初始值）;
或
指针变量名=new 类型标识符 [内存单元个数];
// 例如:
int *a=new int;          //动态分配一个int
int *pi=new int(1);      //动态分配一个int，初始化为1
int *pa=new int[1];      //动态分配一个数组，数组大小为1
```

格式 1 和格式 2 都是申请分配某一数据类型所占字节数的内存空间，其中前者只对变量进行申请，后者则同时将初值存放到该内存单元中；格式 3 可以同时分配若干个内存单元，相当于形成一个动态数组。

开辟数组空间

对于数组进行动态分配的格式为：

`指针变量名 = new 类型名[下标表达式]`

例如：

`int *prt; prt=new int[3];`

delete 的用法

当程序不再需要由 new 分配的内存空间时，可以使用 delete 释放这些内存空间。delete [] 的方括号中不需要填数组元素数，系统自知。即使写了，编译器也忽略。其语法形式如下：

删除单变量地址空间

`delete 指针变量名` 例如 `int *a = new int; delete a;`

删除数组空间

`int *a = new int[15]; delete [] a;`

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
    int *p = new int; // 开辟普通指针空间
    *p = 5;
    *p = *p + 5;
    cout << "p =" << p << endl;
    cout << "*p = " << *p << endl;
    delete p;
    p = new int[5]; // 开辟动态数组空间
    for (int i = 0; i < 5; i++)
    {
        p[i] = i + 1;
        printf("p[%d] = %d\n", i, p[i]);
    }
    delete[] p; //释放5个指针
    return 0;
}
// 结果:
// p =0x558d119b6eb0
// *p = 10
```

```
// p[0] = 1
// p[1] = 2
// p[2] = 3
// p[3] = 4
// p[4] = 5
```

面向对象的特征

类和对象

➢ 对象可以是一个有形的具体存在的事物，也可以是一个无形的，抽象的事物

➢ 对象一般具有两个因素：属性（attribute）和行为（behavior）。属性描述的是事物的静态特征，比如一个人的姓名，年龄，身份证号等信息；行为（或方法）描述事物具有的动态特征，比如一个人走路、说话，一辆汽车向前行驶等

➢ “类”是一组具有相同属性和行为的对象的抽象，类是一种自定义数据类型，对象是类的实例

面向对象程序设计将数据和数据的操作封装在一起，作为不可分割的整体，而各个对象之间通过传递不同的消息来实现相互协作。在 C++中，对象是由数据和函数组成，调用对象的函数就是向该对象发送一条消息，要求该对象完成某一功能。

封装

➢ 封装(encapsulation)是将抽象得到的数据和行为放在一起，形成一个实体——对象，并尽可能隐蔽对象的内部细节。

➢ 对象之间相互独立，互不干扰。对象只留有少量的接口供外部使用，数据和方法是隐藏的。

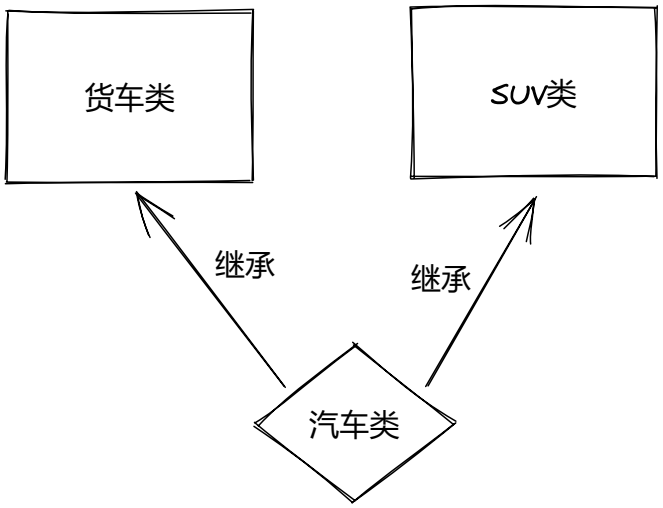
● 封装具有以下特点:

➢ 封装必须提供接口供外部使用，简化了对象的使用。

➢ 封装在内部的数据和方法对外不可见，其他对象不能直接使用。

➢ 封装可以通过继承机制实现代码重用。

继承



➢ 继承（inheritance）是指特殊类的对象拥有其一般类的全部属性与行为，被继承的类称为基类，通过继承得到的新类称为派生类。

➢ C++提供继承机制更符合现实世界的描述，继承机制具有传递性，可以被一层一层的不断继承下去，实现代码重用和可扩充性，减轻程序开发工作的强度，提高程序开发的效率。

多态

➢ 多态（polymorphism）是指不同的对象收到相同的信息时执行不同的操作。所谓消息是指对类的成员函数的调用，不同的操作是指不同的实现，也就是调用了不同的成员函数。

➢ C++语言支持两种多态性，即编译时的多态性（静态多态性）和运行时的多态性（动态多态性）。编译时的多态是在编译的过程中确定同名操作的具体操作对象，比如函数重载（包括运算符重载），根据参数的不同，执行不同的函数。运行时的多态性是在运行过程中才动态地确定所操作的具体对象，C++通过使用虚函数（virtual）来实现动态多态性。

➢ 这种确定具体操作对象的过程叫做绑定（也叫联编），绑定工作在编译链接阶段完成称为静态绑定，在程序运行阶段完成称为动态绑定。

消息

一个对象向另一个对象发出的服务请求被称为“消息”，也可以说是一个对象调用另一个对象的函数。当对象接收到消息时，就会调用相关的方法，执行相应的操作。

- 消息具有以下三个性质：

➢ 同一个对象可以接收不同形式的多个消息，作出不同的响应；

➢ 相同形式的消息可以传递给不同的对象，所作出的响应可以不同；

➢ 对消息的响应并不是必须的，对象可以响应消息，也可以不响应。

类和对象

类和定义

类是一种用户自定义数据类型，类的定义包含两部分内容：数据成员和成员函数（又称函数成员），数据成员表示该类所具有的属性，成员函数表示该类的行为，一般是对数据操作的函数，也称为方法。其定义格式如下：

```
class 类名
{
public:
    公有的数据成员和成员函数
protected:
    受保护的数据成员和成员函数
private:
    私有的数据成员和成员函数
};
```

➢ 定义类时使用关键字 class，类名必须符合标识符命名规范，一般类名的首字母大写

➢ 一个类包含类头和类体两部分，class <类名>称为类头

➢ public（公有）、protected（受保护）与 private（私有）为属性/方法的访问权限，用来控制类成员的存取。如果没有标识访问权限，默认为 private

➢ 三种访问控制权限在类定义中可按任意顺序出现多次，但一个成员只有一种访问权限

➢ 结束部分的分号不能省略

```
class Date
{
private:                                //以下是私有成员
    int year, month, day;               //用三个整型表示属性：年、月、日
public:                                 //以下是公有成员
    void setDate(int y, int m, int d) //函数成员：设置日期
    {
        year = y;
        month = m;
        day = d;
    }
    void show() //函数成员：显示日期
    {
        cout << year << "年" << month << "月" << day << "日" << endl;
```

```
    }
}; //类定义结束符
```

成员函数的实现

➢ 所有成员的声明都必须在类的内部，但是成员函数体的定义则既可以在类的内部也可以在类的外部。

➢ 当定义在类的外部时，函数名之前需要加上类名和作用域运算符（::）以显式的指出该函数所属的类。

➢ 在类外定义函数体的格式如下：

```
返回值类型 类名::成员函数名(形参表)
{
    函数体;
}
```

➢ 在类的外部定义成员函数时，必须同时提供类名和函数名，函数参数列表要和类内函数声明一致。其中，::表示类的作用域分辨符，放在函数名之前类名之后，返回类型在类的作用域之外。

对象的定义与使用

先定义类，再定义对象

`Date d1, d2, d3;`

已经有了类的定义，使用上述方式定义类的实例。

声明类的同时定义类对象

```
class Date
{
    ……..//数据成员和函数成员的实现
}    d1,d2,d3; //声明类型的同时定义三个对象
```

成员的访问权限

访问权限	含义	可存取对象
public	公开	该类成员、子类、友元及所有对象

访问权限	含义	可存取对象
protected	受保护	该类成员、子类、友元
private	私有	该类成员、友元

一般通过在类内定义的 public 成员函数对 private 成员进行存取，为 private 成员提供外界访问的接口

构造函数与析构函数

构造函数的声明与使用

```
class 类名
{
public:
    构造函数名（参数列表）
    {
        函数体
    }
};
```

构造函数名与类名相同，无返回类型。注意什么也不写，也不能用 void

```
#include <iostream>
using namespace std;
class Date
{
private:
    int year, month, day; //用三个整型表示年月日
public:
    Date() //定义不带参数的构造函数
    {
        year = 2019; //初始化数据成员
        month = 5;
        day = 20;
        cout <<"调用构造函数" << endl;
    }
    // Date():year(2019), month(5), day(20) { cout << "调用构造函数"
<< endl; }
    void setTime(int y, int m, int d)
```

```
    {
        year = y;
        month = m;
        day = d;
    }
    void show()
    {
        cout << year << "年" << month << "月" << day << "日" << endl;
    }
};
// Date::Date() : year(2019), month(5), day(20) { cout << "调用构造函数" << endl; };
int main()
{
    Date d; //自动调用构造函数，初始化数据成员
    d.show();
    d.setTime(2019, 9, 20);
    d.show();

    return 0;
}
// 结果:
// 调用构造函数
// 2019年5月20日
// 2019年9月20日
```

除了在函数体内对数据成员初始化，C++还提供另一种初始化方式：使用初始化列表来实现对数据成员的初始化，格式如下：

```
类名::构造函数名（参数表）：初始化列表
{
    //构造函数其他代码
}
```

➢ 初始化列表的形式如下：

`数据成员 1（参数名或常量），数据成员 2（参数名或常量）,数据成员 3（参数名或常量）`

➢ 例如，可将上述代码中的构造函数改成如下形式：

`Date(): year(2019), month(5), day(20) { }`

重载构造函数

在一个类中可以有多个构造函数，即可以重载构造函数，系统在调用构造函数时，根据参数类型和参数个数进行区分，选取合适的构造函数

析构函数

析构函数也是类的一种特殊成员函数，它的作用与构造函数相反，一般是执行对象的清理工作。当对象的生命周期结束时，通常需要做一些善后工作，例如：构造对象时，通过构造函数动态申请了一些内存单元，在对象消失之前就要释放这些内存单元

```
类名：：~析构函数名()
{
    函数体
}
```

➢ 析构函数名与类名相同，但在类名前加“~”字符

➢ 析构函数没有返回值，没有参数，不能被重载，一个类仅有一个析构函数。

➢ 析构函数一般由用户自定义，对象消失时系统自动调用。如果用户没有定义析构函数，系统将自动生成一个不作任何工作的析构函数

➢ 如果一个对象被定义在一个函数体内，则当该函数结束时，该对象的析构函数被自动调用

➢ 若一个对象是使用 new 运算符动态创建的（new 调用构造函数），在使用delete 运算符释放对象时，delete 将会自动调用析构函数

注意：对象消失时的清理工作并不是由析构函数完成，而是由用户在析构函数中添加的清理语句完成

拷贝构造函数

➢ 当程序中需要用一种已经定义的对象去创建另一个对象时，或将一个对象赋值给另一个对象，就需要用到拷贝构造函数。

➢ 拷贝构造函数的名称必须和类名称一致，没有返回类型，只有一个参数，该参数是该类型对象的引用。

➢ 拷贝构造函数的定义格式如下：

```
拷贝构造函数名（类名& 对象名）
{
```

```
    函数体 …
}
```

➢ 拷贝构造函数一般由用户自定义，如果用户没有定义拷贝构造函数，系统将自动生成一个默认的拷贝构造函数进行对象之间的拷贝。

➢ 如果用户自定义了拷贝构造函数，则在用一个类的对象初始化该类的另外一个对象时，自动调用自定义的拷贝构造函数。

➢ 拷贝函数的功能就是把有初始值对象的每个数据成员依次赋值给新创建的对象。

以下三种情况会调用拷贝构造函数:

➢ 当函数的形参为类的对象，将对象作为函数实参传递给函数的形参时。

➢ 当函数的返回值是类的对象，创建临时对象时。

➢ 当用类的一个对象初始化另外一个对象时

```
#include <iostream>
using namespace std;
class Student
{
private:
    char name[20]; //姓名
    char sex[10];  //性别
    int age;       //年龄
public:
    Student(char n[], char s[10], int a) //带参数的构造函数
    {
        strcpy_s(name, strlen(n) + 1, n);
        strcpy_s(sex, strlen(s) + 1, s);
        age = a;
        cout << "调用构造函数:" << name << endl;
    }
    Student(Student &s) //定义拷贝构造函数
    {
        strcpy_s(name, strlen(s.name) + 1, s.name);
        strcpy_s(sex, strlen(s.sex) + 1, s.sex);
        age = s.age;
        cout << "调用拷贝构造函数:" << name << endl;
    }
    ~Student() //定义析构函数
    {
```

```
        cout << "调用析构函数: " << name << endl;
    }
};
Student fun(Student a) //返回值为类类型的普通函数
{
    return a;
}
int main()
{
    Student stu1("Jane", "女", 20);
    Student stu2 = stu1;
    Student stu3("July", "女", 15);
    fun(stu3);
    return 0;
}
// 调用构造函数:Jane
// 调用拷贝构造函数:Jane
// 调用构造函数:July
// 调用拷贝构造函数:July          (传入实参的时候复制了一次)
// 调用拷贝构造函数:July          (返回的时候复制了一次)
// 调用析构函数:July
// 调用析构函数:July
// 调用析构函数:July
// 调用析构函数:Jane
// 调用析构函数:Jane
```

➢ 如果用户未定义拷贝构造函数，系统也会完成工作。C++把这种对象之间数据成员的简赋值称为“浅拷贝”，默认拷贝构造函数执行的也是浅拷贝。大多情况下“浅拷贝”已经能很好地工作，但是一旦对象存在了动态成员，那么浅拷贝就会出问题，通过如下一段代码加以说明。

```
#include <iostream>
#include <assert.h>
using namespace std;
class Book
{
private:
    char *name;   //图书名称
    double price; //图书价格
public:
    Book(char *n, double p)
```

```
    {
        int length = strlen(n);
        name = new char[length + 1];
        strcpy_s(name, length + 1, n);
        price = p;
        cout << "调用构造函数：" << name << endl;
    }
    ~Book()
    {
        if (name != NULL)
        {
            cout << "调用析构函数：" << name << endl;
            delete[] name; //释放分配的内存资源
            name = NULL;
        }
    }
};
int main()
{
    Book a("c++程序设计", 34);
    Book b(a);
    return 0;
}
```

这两个指针指向了堆里的同一个内存空间，如下图 2 (a)所示。对象 b 析构，释放内存，然后对象 a 析构，由于 a.name 和 b.name 所占用的是同一块内存，而同一块内存不可能释放两次，所以当对象 a 析构时，程序出现异常，无法正常执行和结束

➢ 当类中的数据成员是指针类型时，必须定义一个特殊的拷贝构造函数，该拷贝构造函数不仅可以实现对象之间数据成员的赋值，而且可以为新对象单独分配内存空间，这就是“深拷贝”。

友元

类具有封装和信息隐藏的特性，只有类的成员函数才能访问类的私有成员和受保护成员，类外的其他函数无法访问私有成员和受保护成员。

但有些情况下必须要访问某一些类的私有成员，此时就会为了这些特殊的少部分访问操作要把数据成员公有化，破坏私有成员的隐藏性。

为解决上述问题，C++提出一种使用友元（friend）的方案。

友元不是类的成员。友元是一种定义在类外部的普通函数或类，友元不是成员函数，但是它可以访问类中的保护和私有成员。
友元可以是另一个类的成员函数或者不属于任何一个类的普通函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类。

声明友元函数

友元函数可以是全局函数和其他类的成员函数，友元函数定义在类的外部，但它需要在类体内进行声明。为了与该类的成员函数加以区别，在声明时前面加以关键字 friend 修饰

`friend 返回值类型 友元函数名(参数表);`

将其他类的成员函数声明为友元的写法如下：

`friend 返回值类型 其他类的类名::成员函数名(参数表);`

➢ 友元函数是一个普通的函数，它不是本类的成员函数，因此在调用时不能通过对象调用。
➢ 友元函数可以在类内声明，类外定义。
➢ 友元函数对类成员的存取与成员函数一样，可以直接存取类的任何存取控制属性的成员；
➢ private、protected、public 访问权限与友元函数的声明位置无关，因此原则上，友元函数声明可以放在类体中任意部分，但为程序清晰，一般放在类定义的后面。
➢ 不能把其他类的私有成员函数声明为友元。

```
#include <iostream>
using namespace std;
class Boat
{
public:
    Boat(double w)
    {
        weight = w;
    }
    friend double totalWeight(Boat a)
    {
        return a.weight;
    }

private:
    double weight;
};
class Car
{
```

```
public:
    Car(double w)
    {
        weight = w;
    }
    friend double totalWeight(Car b)
    {
        return b.weight;
    }

private:
    double weight;
};
int main()
{
    Boat aa(300);
    Car bb(400);
    cout << "totalweight:" << totalWeight(aa) + totalWeight(bb) <<
endl;
    return 0;
}
// 结果:
// totalweight:700
```

声明友元函数

一个类的成员函数可以是另一个类的友元。例如，教师可以修改学生成绩（访问学生的私有成员），则可以将教师的成员函数声明为学生类的友元函数。

```
#include <iostream>
#include <string>
using namespace std;
class Student; //前向引用声明
class Teacher
{
public:
    void changeGrade(Student *s); //教师成员函数，修改学生成绩
};
class Student
{
```

```
public:
    Student(string name, int num, int grade[5]); //构造函数
    void show();

private:
    string name;
    int num;
    int grade[5];                                    //五门功课成绩
    friend void Teacher::changeGrade(Student *s); //将教师的成员函数声明为学生类的友元函数
};
Student::Student(string name, int num, int grade[])
{
    this->name = name,
    this->num = num;
    for (int i = 0; i < 5; i++)
        this->grade[i] = grade[i];
}
void Student::show()
{
    cout << name << "的成绩为：";
    for (int i = 0; i < 5; i++)
        cout << grade[i] << " ";
}
void Teacher::changeGrade(Student *s)
{
    int n, g; //分别表示要修改的课程编号和成绩
    cout << "\n请输入想要修改的课程编号：（1-5）";
    cin >> n;
    cout << "请重新输入成绩：";
    cin >> g;
    s->grade[n - 1] = g; //友元函数访问私有成员
    cout << "修改成功！\n";
}
int main()
{
    int grade[5];
    cout << "请输入五门功课的成绩：";
    for (int i = 0; i < 5; i++)
        cin >> grade[i];
    Teacher T1;
    Student S1("July", 90, grade);
```

```
    S1.show();
    T1.changeGrade(&S1);
    S1.show();
    return 0;
}

// 结果:
// 请输入五门功课的成绩:100 80 80 80 80
// July的成绩为:100 80 80 80 80
// 请输入想要修改的课程编号:(1-5)1
// 请重新输入成绩:50
// 修改成功!
// July的成绩为:50 80 80 80 80
```

声明友元类

友元除了前面讲过的函数以外，友元还可以是类。

一个类 A 可以将另一个类 B 声明为友元，则类 B 的所有成员函数就都可以访问类 A 的私有成员，这就意味着 B 类的所有成员函数都是 A 类的友元函数。

```
class A{
    ….
    friend class B;//声明B为A类的友元类
};
```

注意：在声明一个友元类时，该类必须已经存在。

```
#include <iostream>
#include <string>
using namespace std;
class Time //定义时间类
{
private:
    int hour;
    int minute;
    int second;

public:
    Time(int hour, int minute, int second)
```

```
    {
        this->hour = hour;
        this->minute = minute;
        this->second = second;
    }
    friend class Date; //声明友元类
};
class Date //定义日期类
{
private:
    int year;
    int month;
    int day;
    Time t;

public:
    Date(int y, int m, int d, int h, int mi, int s) : t(h, mi, s) //
通过初始化列表对成员对象赋值
    {
        year = y;
        month = m;
        day = d;
    }
    void showDate() // Time的友元函数
    {
        cout << "当前日期为: ";
        cout << year << "-" << month << "-" << day <<"\t";
        cout << t.hour << "时" << t.minute << "分" << t.second <<
"秒" << endl; //访问私有成员
    }
};
int main()
{
    Date date1(2019, 8, 10, 11, 56, 35);
    date1.showDate();
    return 0;
}

// 结果:
// 当前日期为: 2019-8-10   11时56分35秒
```

友元关系在类之间不能传递，即类 A 是类 B 的友元，类 B 是类 C 的友元，并不能导出类 A 是类 C 的友元。“咱俩是朋友，所以你的朋友就是我的朋友”这句话在 C++ 的友元关系上不成立。

➢ 友元关系不能被继承，B 类是 A 类的友元，C 类是 B 类的友元，C 类与 A 类之间，如果没有声明，就没有任何友元关系，不能进行数据共享。
➢ 友元关系是单向的，不具有交换性。若类 B 是类 A 的友元，类 A 不一定是类 B 的友元。
➢ 友元关系不具有传递性。若类 B 是类 A 的友元，类 C 是 B 的友元，类 C 不一定是类 A 的友元。
➢ 友元提高了数据的共享性，增强了函数与函数之间，类与类之间的相互联系，极大提高了程序的运行效率。但友元的存在破坏了类中数据的隐蔽性和封装性，使得程序的可维护性降低。

静态成员

C++提供 static 声明的静态成员，用以解决同一个类的不同对象之间数据成员和函数的共享问题。类的静态成员有两种：静态数据成员和静态函数成员。

静态数据成员

➢ 静态数据成员在内存中只保留一份拷贝，由该类的所有对象共同维护和使用，从而实现同一类中所有对象之间的数据共享。
➢ 静态数据成员的值可以被更新，一旦某一对象修改静态数据成员的值之后，其他对象再访问的是更新过之后的值。
➢ 静态数据成员属于类属性（class attribute），类属性是类的所有对象共同拥有的一个数据项，对于任何对象实例，它的属性值相同。

静态数据成员初始化

➢ 静态成员的定义或声明要加关键词 static，并且必须在类内声明，类外初始化。
➢ 类内声明静态数据成员的格式：

```
static 类型标识符 静态数据成员名;
```

➢ 在类外进行初始化格式为：

```
类型标识符 类名::静态数据成员名=初始值;
```

➢ 静态数据成员是所有对象所公有，不属于任何一个对象，独立占用一份内存空间。

➢ 静态数据成员的访问属性同普通数据成员一样，可以为 public、private 和 protected。

➢ 静态数据成员是一种特殊的数据成员，它表示类属性，而不是某个对象单独的属性。

➢ 静态数据成员使用之前必须初始化。

➢ 静态数据成员是静态存储的，它是静态生存周期。程序开始时就分配内存空间，而不是某个对象创建时分配；不随对象的撤销而释放，而是在程序结束时释放内存空间。

➢ 静态数据成员只是在类的定义中进行了引用性声明，因此必须在文件作用域的某个地方对静态数据成员用类名限定进行定义并初始化，即应在类体外对静态数据成员进行初始化。

➢ 静态数据成员初始化时前面不加 static 关键字，以免与一般静态变量或对象混淆。

➢ 静态数据成员初始化时必须使用类作用域运算符::来标明它所属的类。

静态数据成员的访问

➢ 静态数据成员本质上是全局变量，在程序中，即使不创建对象其静态数据成员也存在。因此，可以通过类名对其直接访问，一般格式为：

```
类名::静态数据成员;
```

➢ 静态数据成员在类内可以随意访问，如果在类外，通过类名与对象只能访问控制属性为 public 的成员。

静态成员函数

静态成员函数和静态数据成员一样，都属于类的成员，不属于某一个对象，是所有对象共享的成员函数，只要类存在，就可以使用静态成员函数

```
static 返回类型 静态成员函数名（形参表）;
```

静态成员函数可以在类内定义，也可以在类内声明，类外定义。在类外定义时不能在使用 static 作为关键字。

静态成员函数的调用形式有以下两种：

➢ 通过类名直接调用静态成员函数：

```
类名::静态成员函数名（参数表）;
```

➢ 通过对象调用静态成员函数：

```
对象.静态成员函数（参数表）;
```

➢ 公有的静态成员函数可通过类名或对象名进行调用，一般非静态成员函数只能通过对象名调用。

➢ 通过对象访问静态成员函数之前，对象已经创建。

➢ 静态成员函数可以直接访问类中静态数据成员和静态成员函数，但不能直接访问类中的非

静态成员。

➢ 由于静态成员是独立于对象而存在的，因此静态成员没有 this 指针。

➢ 类的非静态成员函数可以调用静态成员函数，反之不可以（动态可以调静态，静态不能调动态函数，因为可能没有创建）。

```
#include <iostream>
using namespace std;
class CRectangle
{
private:
    int w, h;               //矩形的长和高
    static int totalArea;   //矩形总面积
    static int totalNumber; //矩形总数
public:
    CRectangle(int w_, int h_);
    ~CRectangle();
    CRectangle(CRectangle &r);
    static void PrintTotal(CRectangle r);
};
CRectangle::CRectangle(int w_, int h_)
{
    w = w_;
    h = h_;
    totalNumber++;      //有对象生成则增加总数
    totalArea += w * h; //有对象生成则增加总面积
}
CRectangle::~CRectangle()
{
    totalNumber--;      //有对象消亡则减少总数
    totalArea -= w * h; //有对象消亡则减少总而积
}
CRectangle::CRectangle(CRectangle &r)
{
    totalNumber++;
    totalArea += r.w * r.h;
    w = r.w;
    h = r.h;
}
void CRectangle::PrintTotal(CRectangle r)
{
```

```
    cout << r.w << "," << r.h << endl;
    cout << totalNumber << "," << totalArea << endl;
}
int CRectangle::totalNumber = 0;
int CRectangle::totalArea = 0;
//必须在定义类的文件中对静态成员变量进行一次声明
int main()
{
    CRectangle r1(4, 6), r2(2, 5);
    // cout << CRectangle::totalNumber; //错误，totalNumber 是私有
    CRectangle::PrintTotal(r1);
    r1.PrintTotal(r1);
    return 0;
}

// 结果:
// 4,6
// 3,58
// 4,6
// 3,58
```

常成员与常对象

常对象

如果想要共享某一数据，但又不希望它被修改，可以用 const 修饰，表示该数据为常量。如果某个对象不允许被修改，也可以用 const 修饰，则该对象称为常对象。

const 用来限定类的数据成员和成员函数，分别称为类的常数据成员和常函数成员。

➢ 常对象可以调用常成员函数，不能调用非 const 成员函数；非 const 对象，可以调用普通成员函数和常成员函数。

➢ 定义常对象的一般格式如下：

```
类名 const 对象名（实参表）;
// 或者
const 类名 对象名（实参表）;
```

例如，定义一个日期对象始终为2019 年 8 月 11 号，则可以定义为：

`Date const d1(2019,8,11); //d1 为常对象`

定义常对象时必须赋初值

以下语句是错误的：

```
Date const d2; //d2为常对象
Date d3(2019,8,15); //d3为普通对象
d2=d3; //错误，常对象定义时必须赋初值
```

通过多态理解常成员函数：

```
class A{
private:
     int w,h;
public:
    int getValue() const
    {
         return w*h;
    }

     int getValue(){
         return w+h;
     }
     A(int x,int y)
     {
         w=x,h=y;
     }
     A(){}
};
void main()
{
     A const a(3,4);
     A c(2,6);
     cout<<a.getValue()<<c.getValue()<<"cctwlTest"; //输出12和8
}
```

常对象不能调用非常成员函数（除了系统自动调用的构造函数和析构函数），目的是防止这些函数修改对象中数据成员的值。如果有下面的语句：

```
Date const d4(2019,8,12);
d4.showDate();    //错误，试图调用常对象中的非常成员函数
```

常成员函数可以访问常对象中的数据成员，但不允许其修改常对象中数据成员的值。

如果想要修改常对象中某个数据成员的值，可以用 mutable 进行声明，例如：

`mutable int year;`

把 year 声明为可变的数据成员，就可以用常成员函数对其值进行修改。

常数据成员

用const修饰的数据成员称为常数据成员，其用法与常变量相似。常数据成员在定义时必须进行初始化，并且其值不能被更新（除非数据成员被mutable修饰时，可以被修改）。

```
数据类型 const 数据成员名;
// 或者
const 数据类型 数据成员名;
```

➢ 任何函数都不能对常数据成员赋值。

➢ 构造函数对常数据成员进行初始化时也只能通过初始化列表进行。

➢ 常数据成员在定义时必须赋值或必须初始化。

➢ 如果有多个构造函数，必须都初始化常数据成员

常数据成员不能在声明时赋初始值，必须在构造函数初始化列表进行初始化；普通数据成员在初始化列表和函数体中初始化均可

```
#include <iostream>
#include <string>
using namespace std;
class Student
{
private:
    const string name;       //常数据成员
    const int num;           //常数据成员
    static const int count; //静态常数据成员
public:
    Student(string i, int a) : name(i), num(a)
    {
```

```
        cout << "constructor!" << endl;
    }
    void display()
    {
        cout << name << "," << num << "," << count << endl;
    }
};
const int Student::count = 0; // 静态常数据成员在类外说明和初始化
int main()
{
    Student s1("jane", 1001);
    s1.display();
    return 0;
}

// 结果:
// constructor!
// jane,1001,0
```

常成员函数

`返回类型 函数名（参数列表） const;`

➢ const是加在函数说明后的类型修饰符，它是函数类型的一部分，在实现部分也要带该关键字，调用时不加const关键字。

➢ const关键字可以用于对重载函数的区分。

➢ 常成员函数不能更新任何数据成员，也不能调用该类中没有用const修饰的成员函数，只能调用常成员函数和常数据成员。

➢ 非常对象也可以调用常成员函数，但是当常成员函数与非常成员函数同名时（可以视为函数重载），非常对象是会优先调用非常成员函数。