类和对象

序言

在谈到面向对象时，大家可能第一反应就是面向对象的特征为封装、继承、多态，这其实是一些编程语言在宣传过程中，为了让大家迅速记住语言的特点而提出来的。面向对象开发不会从无数使用早期技术而失败的软件项目的灰烬中自发地产生，它不是与早期方法断然决裂的，实际上，它是建立在以前技术的最佳思想之上的。

当回首相对简单、但又多姿多彩的软件工程历史时，会注意到如下两个趋势：

- 关注点从小规模编程向大规模编程转变
- 高级程序设计语言的演进。

这其实说明程序世界是在不断演变的，演变的过程中，就出现了两种认识世界的观点，这就是面向过程和面向对象。但不论是过程论还是对象论，都承认一点，程序世界本质上只有两种东西：数据和逻辑。数据天性喜静，构成了程序世界的本体和状态；逻辑天性好动，作用于数据，推动程序世界的演进和发展。但是在数据和逻辑的存在形式和演进形式上，过程论和对象论的观点截然不同。

过程论认为：数据和逻辑是分离的、独立的，各自形成程序世界的一个方面（Aspect）。所谓世界的演变，是在逻辑作用下，数据做改变的一个过程。这种过程有明确的开始、结束、输入、输出，每个步骤有着严格的因果关系。过程是相对稳定的、明确的和预定义的，小过程组合成大过程，大过程还可以组合成更大的过程。所以，程序世界本质是过程，数据作为过程处理对象，逻辑作为过程的形式定义，世界就是各个过程不断进行的总体。

对象论认为：数据和逻辑不是分离的，而是相互依存的。相关的数据和逻辑形成个体，这些个体叫做对象，世界就是由一个个对象组成的。对象具有相对独立性，对外提供一定的服务。所谓世界的演进，是在某个“初始作用力”作用下，对象间通过相互调用而完成的交互；在没有初始作用力下，对象保持静止。这些交互并不是完全预定义的，不一定有严格的因果关系，对象间交互是“偶然的”，对象间联系是“暂时的”。世界就是由各色对象组成，然后在初始作用力下，对象间的交互完成了世界的演进。

过程论和对象论**不是一种你死我活的绝对对立**，**而是一种辩证统一的对立**，**两者相互渗透、在一定情况下可以相互转化**，是一种“你中有我、我中有你”的对立。如果将对象论中的所有交互提取出来而撇开对象，就变成了过程论，而如果对过程论中的数据和逻辑分类封装并建立交互关系，就变成了对象论。

过程论相对确定，有利于明晰演进的方向，但当事物过于庞大繁杂，将很难理清思路。因为过程繁多、过程中又有子过程，容易将整个世界看成一个纷繁交错的过程网，让人无法看清。

对象论相对不确定，但是因为以对象为基本元素，即使很庞大的事物，也可以很好地分离关注，在研究一个对象的交互时，只需要关系与其相关的少数几个对象，不用总是关注整个流程和世界，**对象论更有助于分析规模较大的事物**。但是，对象论也有困难。例如，如何划分对象才合理？对于同一个驱动力，为什么不同情况下参与对象和交互流程不一样？如何确定？其实，这些困难也正是面向对象技术中的困难。

接下来，我们就开始讨论面向对象技术了。

类的产生

客观现实世界有的都是对象，万物皆对象，没有类。但大家都知道，在我们用Java、C++、C#等语言写代码时，都需要先定义一个类，之后再通过类创建对象。为什么呢？从认识论来说，首先有具体认知能力，而后才能有抽象认知能力，抽象认知能力是一种高层的，人类特有的认知能力，它使我们可以从大量具体认知中，舍弃个别的、非本质的属性，提取出共同的、本质的属性，是形成概念的必要手段。所

以从哲学角度说，是先有对象，然后才有类，类和对象是“一般和特殊”这一哲学原理在程序世界中的具体体现。

类可以帮助我们方便地认识和定义世界中的对象。这个作用是显而易见的。例如当今世界有76亿人，如果不会抽象思维，我们每遇到一个人，都要认知一遍：啊！这个对象有眼睛，有耳朵，有鼻子有嘴，有胳膊有腿……要是真这样，世界也太疯狂了。有了类的概念，我们就可以只记类的数据和逻辑，而对于具体对象，只要知道它属于什么“类”，一切就都知道了，所需要区分的只是不同对象的数据具有不同值而已。

接下来，我们就来一步一步地探索C++中有关类的一切。

C++中类的定义

C++用类来描述对象，类是对现实世界中相似事物的抽象，比如同是“双轮车”的摩托车和自行车，有共同点，也有许多不同点。“车”类是对摩托车、自行车、汽车等相同点的提取与抽象。

类的定义分为两个部分：数据（相当于属性）和对数据的操作（相当于行为）。
从程序设计的观点来说，**类就是数据类型**，是用户定义的数据类型，对象可以看成某个类的实例（某个类的变量）。所以说类是对象的封装，对象是类的实例。

C++中用关键字class来定义一个类，其基本形式如下：

```
class 类名 {
public:
    //公有数据成员和成员函数
protected:
    //保护数据成员和成员函数
private:
    //私有数据成员和成员函数
};  // 千万不要忘了这个分号
```

class内部可以拥有的是数据成员(属性)和成员函数(行为)，他们可以分别用三个不同的关键字进行修饰，public、protected、private. 其中public进行修饰的成员表示的是该类可以提供的接口、功能、或者服务；protected进行修饰的成员，其访问权限是开放给其子类；private进行修饰的成员是不可以在类之外进行访问的，只能在类内部访问，可以说封装性就是由该关键字来体现。下面以一台大家熟悉的计算机来举例:

```
class Computer {
public:
    //成员函数
    void setBrand(const char * brand)
    {
        strcpy(_brand, brand);
    }
    void setPrice(float price)
    {
        _price = price;
    }
private:
    //数据成员
    char _brand[20];
    float _price;
};

int main(int argc, char * argv[])
{
```

```
    Computer pc;
    pc.setBrand("Huawei Matebook14");
    pc.setPrice(5699);
    return 0;
}
```

在类中定义的成员函数，都是`inline`函数。除了可以在类内部实现外，成员函数还可以在类之外实现。在类定义的外部定义成员函数时，应使用作用域限定符（::）来标识函数所属的类，即有如下形式:

```
返回类型 类名::成员函数名(参数列表)
{
    //....
}
```

对于Computer中的两个成员函数，我们在类之外实现，其实现如下:

```
void Computer::setBrand(const char * brand)
{
    strcpy(_brand, brand);
}

void Computer::setPrice(float price)
{
    _price = price;
}
```

`class`与`struct`的区别

在`C++`中，与`C`相比，`struct`的功能已经进行了扩展。`class`能做的事儿，`struct`一样能做，他们之间唯一的区别，就是默认访问权限不同。`class`的默认访问权限是`private`，`struct`的默认访问权限是`public`

```
struct Computer {
//成员函数，其访问权限是public
    void setBrand(const char * brand)
    {
        strcpy(_brand, brand);
    }

    void setPrice(float price)
    {
        _price = price;
    }

//数据成员，其访问权限是public
    char _brand[20];
    float _price;
};

class Computer2 {
//成员函数，其访问权限是private
    void setBrand(const char * brand)
    {
        strcpy(_brand, brand);
```

```
    }
    void setPrice(float price)
    {
        _price = price;
    }
//数据成员，其访问权限是private
    char _brand[20];
    float _price;
};
```

对象的创建

在之前的`Computer`类中，通过自定义的公共成员函数`setBrand`和`setPrice`实现了对数据成员的初始化。实际上，`C++`为类提供了一种特殊的成员函数----构造函数来完成相同的工作。**构造函数**有一些独特的地方：

- 函数的名字与类名相同
- 没有返回值
- 没有返回类型，即使是`void`也不能有

构造函数在对象创建时自动调用，用以完成对象成员变量等的初始化及其他操作(如为指针成员动态申请内存等)；如果程序员没有显式定义它，系统会提供一个默认构造函数。下面我们用一个点`Point`来举例：

```
class Point {
public:
    //即使不写，编译器也会自动提供一个
    Point()
    {
        cout << "Point()" << endl;
        _ix = 0;
        _iy = 0;
    }
    void print()
    {
        cout << "(" << _ix
             << "," << _iy
             << ")" << endl;
    }
private:
    int _ix;
    int _iy;
};

int main(void)
{
    Point pt;
    pt.print();
    return 0;
}
```

编译器自动生成的缺省(默认)构造函数是无参的，实际上，构造函数可以接收参数，在对象创建时提供更大的自由度。我们在上面的`Point`类中可以加入一个新的构造函数

```
class Point {
public:
    //...
    Point(int ix, int iy)
    {
        cout << "Point(int,int)" << endl;
        _ix = ix;
        _iy = iy;
    }
    //....
};

int main(void)
{
    Point pt(1, 2);
    pt.print();

    Point pt2(11, 12);
    pt2.print();
    return 0;
}
```

上面的例子同时出现了无参构造函数，和有参构造函数，这说明了构造函数是可以重载的。

在上面，我们说编译器会自动给`Point`类生成一个默认构造函数，这是有前提条件的，就是类中没有定义任何构造函数。现在假设`Point`类中只显式定义了一个有参构造函数，则编译器不会再自动提供默认构造函数，如果还希望通过默认构造函数创建对象，则需要显式定义一个默认构造函数。

初始化表达式

在上面的例子中，构造函数对数据成员进行初始化时，都是在函数体内进行的。除此以外，还可以通过初始化列表完成。初始化列表位于构造函数形参列表之后，函数体之前，用冒号开始，如果有多个数据成员，再用逗号分隔，初始值放在一对小括号中。例子如下：

```
class Point {
public:
    //...
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    {
        cout << "Point(int = 0,int = 0)" << endl;
    }
    //...
};
```

如果没有在构造函数的初始化列表中显式地初始化成员，则该成员将在构造函数体之前执行默认初始化。如在“对象的创建”部分的两个构造函数中的`_ix`和`_iy`都是先执行默认初始化后，再在函数体中执行赋值操作。可能有同学会觉得在初始化列表中进行成员初始化不习惯，但有些时候成员必须在初始化列表中进行，否则会出现编译报错。

注意：每个成员在初始化列表之中只能出现一次，其初始化的顺序不是由成员变量在初始化列表中的顺序决定的，而是由成员变量在类中被声明时的顺序决定的。（举例说明）

```
class Foo {
public:
    Foo(int a)
    : _iy(a)   //在初始化列表中， _iy好像先被初始化
    , _ix(_iy)
    {
        cout << "Foo(int)" << endl;
    }
private:
    int _ix;  //在声明时，_ix在前
    int _iy;
};
```

对象的销毁

构造函数在创建对象时被系统自动调用，而析构函数(Destructor)在对象被撤销时被自动调用，相比构造函数，析构函数要简单的多。析构函数有如下特点：

- 与类同名，之前冠以波浪号，以区别于构造函数。
- 析构函数没有返回类型，也不能指定参数。因此，析构函数只能有一个，不能被重载。
- 对象超出其作用域被销毁时，析构函数会被自动调用。

析构函数在对象撤销时自动调用，用以执行一些清理任务，如释放成员函数中动态申请的内存等。如果程序员没有显式的定义它，系统也会提供一个默认的析构函数。例如:

```
class Point {
public:
    //...
    ~Point() {}
    //...
};
```

由于`Point`类比较简单，数据成员中没有需要进行清理的资源，所以即使不显式定义析构函数，也没关系。我们再举一个例子：

```
class Computer {
public:
    Computer(const char * brand, double price)
    : _brand(new char[strlen(brand) + 1]())
    , _price(price)
    {
        strcpy(_brand, brand);
    }

    ~Computer()
    {
        delete [] _brand;
        cout << "~Computer()" << endl;
    }
private:
    char * _brand;
    double _price;
};
```

以上的`Computer`中，有一个数据成员是指针，而该指针在构造函数中初始化时已经申请了堆空间的资源，则当对象被销毁时，必须回收其资源。此时，编译器提供的默认析构函数是没有做回收操作的，因此就不再满足我们的需求，我们必须显式定义一个析构函数，在函数体内回收资源。

析构函数除了在对象被销毁时自动调用外，还可以显式手动调用，但一般不建议这样使用。

析构函数在哪些时候会被调用呢?

1. 对于全局定义的对象，每当程序开始运行，在主函数`main`接受程序控制权之前，就调用构造函数创建全局对象，整个程序结束时，自动调用全局对象的析构函数。
2. 对于局部定义的对象，每当程序流程到达该对象的定义处就调用构造函数，在程序离开局部对象的作用域时调用对象的析构函数。
3. 对于关键字`static`定义的静态局部对象，当程序流程第一次到达该对象定义处调用构造函数，在整个程序结束时调用析构函数。
4. 对于用`new`运算符创建的对象，每当创建该对象时调用构造函数，当用`delete`删除该对象时，调用析构函数。

拷贝构造函数

`C++`中经常会使用一个变量初始化另一个变量，如

```
int x = 1;
int y = x;
```

我们希望这样的操作也能作用于自定义类类型，如

```
Point pt1(1, 2);
Point pt2 = pt1;
```

这两组操作是不是一致的呢？第一组好说，而第二组只是将类型换成了`Point`类型，执行`Point pt2 = pt1;`语句时，`pt1`对象已经存在，而`pt2`对象还不存在，所以也是这句创建了`pt2`对象，既然涉及到对象的创建，就必然需要调用构造函数，而这里会调用的就是复制构造函数，又称为拷贝构造函数。当我们进行测试时，会发现我们不需要显式给出拷贝构造函数，就可以执行第二组测试。这是因为如果类中没有显式定义拷贝构造函数时，编译器会自动提供一个缺省的拷贝构造函数。其原型是:

```
类名::类名(const 类名&);
```

那缺省（默认）的拷贝构造函数是如何实现的呢？很简单，我们来实现一下`Point`类的拷贝构造函数:

```
Point::Point(const Point & rhs)
: _ix(rhs._ix)
, _iy(rhs._iy)
{}
```

由于`Point`的成员比较简单，缺省的拷贝构造函数已经可以满足需求了，所以可以不显式定义。接下来，我们把目光转向`Computer`类，如果`Computer`类使用缺省拷贝构造函数，会发生什么问题呢？我们先来看看缺省的拷贝构造函数的实现

```
Computer::Computer(const Computer & rhs)
: _brand(rhs._brand)
, _price(rhs._price)
{}

//执行构造初始化
Computer pc1("Huawei Matebook14", 5699);
Computer pc2 = pc1;
```

从上面的定义来看，`pc1`与`pc2`对象的数据成员`_brand`都会指向同一个堆空间的字符串，这种只拷贝指针的地址的方式，我们称为**浅拷贝**。当两个对象被销毁时，就会造成`double free`的问题。显然，缺省拷贝构造函数不再满足需求，此时需要显式定义拷贝构造函数：

```
Computer::Computer(const Computer & rhs)
: _brand(new char[strlen(rhs._brand) + 1]())
, _price(rhs._price)
{
    strcpy(_brand, rhs._brand);
}
```

这种拷贝指针所指空间内容的方式，我们称为**深拷贝**。因为两个对象都拥有各自的独立堆空间字符串，一个对象销毁时就不会影响另一个对象。

拷贝构造函数的调用时机：

1. 当把一个已经存在的对象赋值给另一个新对象时，会调用拷贝构造函数。
2. 当实参和形参都是对象，进行实参与形参的结合时，会调用拷贝构造函数。
3. 当函数的返回值是对象，函数调用完成返回时，会调用拷贝构造函数。//-fno-elide-constructors

注意：拷贝构造函数的参数形式可以改变吗？

隐含的`this`指针

不知道大家是否有一个疑问：在上面的例子中，我们通过对象名调用成员函数后，都能准确的访问相应对象的数据成员，而不会出错，这到底是如何实现的呢？比如

```
Point pt(1, 2);
pt.print(); // (1, 2)
Point pt2(11, 12);
pt2.print();// (11, 12)
```

其实在类中定义的非静态成员函数中都有一个隐含的this指针，它代表的就是当前对象本身，它作为成员函数的第一个参数，由编译器自动补全。比如`print`函数的完整实现是：

```
void Point::print(/*Point * const this*/)
{
    cout << "(" << this->_ix
         << "," << this->_iy
         << ")" << endl;
}
```

对于类成员函数而言，并不是一个对象对应一个单独的成员函数体，而是此类的所有对象共用这个成员函数体。 当程序被编译之后，此成员函数地址即已确定。而成员函数之所以能把属于此类的各个对象的数据区别开, 就是靠这个this指针。函数体内所有对类数据成员的访问， 都会被转化为this->数据成员的方式。

赋值运算符函数

赋值运算是一种很常见的运算，比如：

```
int x = 1, y = 2;
x = y;
```

同样地， 我们也希望该操作能作用于自定义类类型，比如：

```
Point pt1(1, 2), pt2(3, 4);
pt1 = pt2;//赋值操作
```

在执行`pt1 = pt2;`该语句时， `pt1`与`pt2`都存在，所以不存在对象的构造，这要与`Point pt2 = pt1;`语句区分开，这是不同的。

在这里，当`=`作用于对象时，其实是把它当成一个函数来看待的。在执行`pt1 = pt2;`该语句时，需要调用的是赋值运算符函数。其形式如下：

```
返回类型 类名::operator=(参数列表)
{
    //...
}
```

当我们进行测试时，会发现我们不需要显式给出赋值运算符函数，就可以执行测试。这是因为如果类中没有显式定义赋值运算符函数时，编译器会自动提供一个缺省的赋值运算符函数。对于`Point`类而言，其实现如下:

```
Point & Point::operator=(const Point & rhs)
{
    _ix = rhs._ix;
    _iy = rhs._iy;
}
```

然而，当我们对Computer对象也执行赋值操作时，又会发生什么呢？先看`Computer`类的赋值运算符函数的实现:

```
Computer & Computer::operator=(const Computer & rhs)
{
    _brand = rhs._brand;
    _price = rhs._price;
    return *this;
}
```

很显然，这里默认情况下依然会采用浅拷贝的方式拷贝字符串。当两个对象被销毁时，同样会造成`double free`的问题，因此缺省赋值运算符函数不再满足需求，此时需要显式定义赋值运算符函数：

```
Computer & Computer::operator=(const Computer & rhs)
{
    if(this != &rhs) {
        delete [] _brand;

        _brand = new char[strlen(rhs._brand) + 1]();
        strcpy(_brand, rhs._brand);

        _price = rhs._price;
    }
    return *this;
}
```

注意： 赋值运算符函数的返回值类型可以改变吗?

特殊数据成员的初始化

在C++的类中，有4种比较特殊的数据成员，他们分别是常量成员、引用成员、类对象成员和静态成员，他们的初始化与普通数据成员有所不同。

常量数据成员

当数据成员用const关键字进行修饰以后，就成为常量成员。一经初始化，该数据成员便具有“只读属性”，在程序中无法对其值修改。事实上，在构造函数体内初始化const数据成员是非法的，它们只能在构造函数初始化列表中进行初始化。如：

```
class Point {
public:
    //错误写法
    Point(int ix = 0, int iy = 0)
    {
        _ix = ix;//error, 这是赋值语句，const成员不能修改
        _iy = iy;//error
        _iz = _ix;
    }

    //正确写法
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    , _iz(_ix)
    {}

private:
    const int _ix;
    const int _iy;
    int & _iz;
};
```

引用数据成员

和常量成员相同，引用成员也必须在构造函数初始化列表中进行初始化，否则编译报错。

类对象成员

当数据成员本身也是自定义类类型对象时，比如一个直线类`Line`对象中包含两个`Point`类对象，对Point对象的创建就必须要放在`Line`的构造函数的初始化列表中进行。如

```
class Line {
public:
    Line(int x1, int y1, int x2, int y2)
    : _pt1(x1, y1)
    , _pt2(x2, y2)
    {
        cout << "Line(int,int,int,int)" << endl;
    }
private:
    Point _pt1;
    Point _pt2;
};
```

当`Line`的构造函数没有在其初始化列表中初始化对象`_pt1`和`_pt2`时，系统也会自动调用`Point`类的默认构造函数，此时就会与预期的构造不一致。因此需要显式在`Line`的构造函数初始化列表中初始化`_pt1`和`_pt2`对象。

静态数据成员

`C++`允许使用`static`（静态存储）修饰数据成员，这样的成员在编译时就被创建并初始化的（与之相比，对象是在运行时被创建的），且其实例只有一个，被所有该类的对象共享，就像住在同一宿舍里的同学共享一个房间号一样。静态数据成员和之前介绍的静态变量一样，当程序执行时，该成员已经存在，一直到程序结束，任何该类对象都可对其进行访问，静态数据成员存储在全局/静态区，并不占据对象的存储空间。

下面以`Computer`为例，模拟购买电脑的过程，为了获取总价，我们定义一个静态变量`_totalPrice`：

```
class Computer {
public:
    Computer(const char * brand, double price)
    : _brand(new char[strlen(brand) + 1]())
    , _price(price)
    {
        strcpy(_brand, brand)
        _totalPrice += _price;
    }

    void print()
    {
        cout << "品牌:" << _brand << endl
             << "价格:" << _price << endl
             << "总价:" << _totalPrice << endl;
    }
    //...
private:
    char * _brand;
    double _price;
    static double _totalPrice;
};
```

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对象时被定义的。这意味着它们不是由类的的构造函数初始化的，一般来说，我们不能在类的内部初始化静态数据成员，必须在类的外部定义和初始化静态数据成员，且不再包含`static`关键字，格式如下:

```
类型 类名::变量名 = 初始化表达式; //普通变量
类型 类名::对象名(构造参数);      //对象变量
```

`Computer`中的静态变量`_totalPrice`的初始化如下:

```
double Computer::_totalPrice = 0;
```

特殊的成员函数

除了特殊的数据成员以外，`C++`类中还有两种特殊的成员函数：静态成员函数和`const`成员函数。我们先来看看静态成员函数。

静态成员函数

成员函数也可以定义成静态的，静态成员函数的特点:

1. 静态成员函数内部不能使用非静态的成员变量和非静态的成员函数
2. 静态成员函数内部只能调用静态数据成员和静态的成员函数

原因在于静态成员函数的参数列表中不含有隐含的`this`指针，如下:

```
class Computer {
public:
    Computer(const char * brand, double price)
    : _brand(new char[strlen(brand) + 1]())
    , _price(price)
    {
        strcpy(_brand, brand);
        _totalPrice += _price;
    }
    //...
    static void printTotalPrice()
    {
        cout << "总价:" << _totalPrice << endl;
    }
    //...
private:
    char * _brand;
    double _price;
    static double _totalPrice;
};
```

对于静态成员函数，还可以直接通过类名进行调用，这也是最常用的方式:

```
int main(void)
{
    Computer pc1("Huawei MateBook14", 5699);
    pc1.print();
    Computer::printTotalPrice();//通过类名直接调用
    return 0;
}
```

下面让我们再来看一个有意思的例子:

```
class NullPointCall {
public:
    static void test1();
    void test2();
    void test3(int test);
    void test4();
private:
    static int _static;
    int _test;
};

int NullPointCall::_static = 0;

void NullPointCall::test1()
{
    cout << "_static = " << _static << endl;
}

void NullPointCall::test2()
{
    cout << "coding is very cool!" << endl;
}

void NullPointCall::test3(int test)
{
    cout << "test = " << test << endl;
}

void NullPointCall::test4()
{
    cout << "_test = " << _test << endl;
}

int main(void)
{
    NullPointCall * pNull = nullptr;
    pNull->test1();
    pNUll->test2();
    pNull->test3(10);
    pNull->test4();
    return 0;
}
```

请问，上面这个程序会发生什么呢?

`const`成员函数

之前已经介绍了`const`在函数中的应用，实际上，`const`在类成员函数中还有种特殊的用法，把`const`关键字放在函数的参数表和函数体之间（与之前介绍的`const`放在函数前修饰返回值不同），称为`const`成员函数，其具有以下特点:

1. 只能读取类数据成员，而不能修改之。
2. 只能调用`const`成员函数，不能调用非`const`成员函数。

其格式为:

```
类型 函数名(参数列表) const
{
    函数体
}
```

比如当给`Computer`类添加一个`const`的打印函数后，则其实现如下:

```
class Computer {
public:
    //...
    void print() const
    {
        cout << "品牌:" << _brand << endl
             << "价格:" << _price << endl;
    }
    //...
};
```

注意：大家可以思考一下：当定义一个`const`成员函数时，到底发生了什么呢?

对象的组织

有了自己定义的类，或者使用别人定义好的类创建对象，其机制与使用`int`等创建普通变量几乎完全一致，同样可以创建`const`对象、创建指向对象的指针、创建对象数组，还可使用`new`和`delete`等创建动态对象。

`const`对象

类对象也可以声明为`const`对象，一般来说，能作用于`const`对象的成员函数除了构造函数和析构函数，便只有`const`成员函数了，因为`const`对象只能被创建、撤销以及只读访问，改写是不允许的。

```
const Point pt(1, 2);
pt.print();
```

指向对象的指针

对象占据一定的内存空间，和普通变量一致，`C++`程序中采用如下形式声明指向对象的指针：

```
类名 *  指针名 [=初始化表达式];
```

初始化表达式是可选的，既可以通过取地址（&对象名）给指针初始化，也可以通过申请动态内存给指针初始化，或者干脆不初始化（比如置为`nullptr`），在程序中再对该指针赋值。
指针中存储的是对象所占内存空间的首地址。针对上述定义，则下列形式都是合法的：

```
Point pt(1, 2);
Point * p1 = nullptr;
Point * p2 = &pt;
Point * p3 = new Point(3, 4);
Point * p4 = new Point[5];

p3->print();//合法
(*p3).print();//合法
```

对象数组

对象数组和标准类型数组的使用方法并没有什么不同，也有声明、初始化和使用3个步骤。

1. 对象数组的声明:

```
类名 数组名[对象个数];
```

这种格式会自动调用默认构造函数或所有参数都是缺省值的构造函数。

2. 对象数组的初始化：可以在声明时进行初始化。

```
Point pts[2] = {Point(1, 2), Point(3, 4)};
Point pts[] = {Point(1, 2), Point(3, 4)};
Point pts[5] = {Point(1, 2), Point(3, 4)};
//或者
Point pts[2] = {{1, 2}, {3, 4}};//注意是大括号，不是小括号
Point pts[] = {{1, 2}, {3, 4}};
Point pts[5] = {{1, 2}, {3, 4}};
```

堆对象

和把一个简单变量创建在动态存储区一样，可以用 `new` 和 `delete` 表达式为对象分配动态存储区，在复制构造函数一节中已经介绍了为类内的指针成员分配动态内存的相关范例，这里主要讨论如何为对象和对象数组动态分配内存。如：

```
void test()
{
    Point * pt1 = new Point(11, 12);
    pt1->print();
    delete pt1;
    pt1 = nullptr;

    Point * pt2 = new Point[5]();//注意
    pt2->print();
    (pt2 + 1)->print();

    delete [] pt2;
}
```

注意: 使用new表达式为对象数组分配动态空间时，不能显式调用对象的构造函数，因此，对象要么没有定义任何形式的构造函数（由编译器缺省提供），要么显式定义了一个（且只能有一个）所有参数都有缺省值的构造函数。

单例模式

单例模式是23种 `GOF` 模式中最简单的设计模式之一，属于创建型模式，它提供了一种创建对象的方式，确保只有单个对象被创建。这个设计模式主要目的是想在整个系统中只能出现类的一个实例，即一个类只有一个对象。根据我们已经学过的知识，其实现步骤有如下三步:

1. 将构造函数私有化
2. 在类中定义一个静态的指向本类型的指针变量
3. 定义一个返回值为类指针的静态成员函数

```
class Singleton {
public:
    static Singleton * getInstance()
```

```
    {
        if(nullptr == _pInstance)
        {
            _pInstance = new Singleton();
        }
        return _pInstance;
    }
private:
    Singleton() {   cout << "Singleton()" << endl;  }

private:
    Static Singleton * _pInstance;
};

Singleton * Singleton::_pInstance = nullptr;

int main(void)
{
    Singleton * p1 = Singleton::getInstance();
    Singleton * p2 = Singleton::getInstance();
    assert(p1 == p2);
    return 0;
}
```

注意：现在的实现，还有一些问题，你知道怎么改进吗？