第二章 类与对象基础

面向对象思想

过程论认为：数据和逻辑是分离的、独立的，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。

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：

1. 数据，相当于现实世界中的属性，称为**数据成员**；
2. 对数据的操作，相当于现实世界中的行为，称为**成员函数**。

有些地方，会将类的数据成员和成员函数统称为类的**成员**。

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（某类的变量）。所以说类是对象的抽象，对象是类的实例。

由对象抽象出类

由类实例化出对象

C++中用关键字class来定义一个类，其基本形式如下：类的定义和声明

```
class MyClass{//类的定义
    //……
    void myFunc(){}  //成员函数
    int _a;          //数据成员
};//一定要有分号


//类也可以先声明，后完成定义
class MyClass2;//类的声明

class MyClass2{//类的定义
    //……
};//分号不能省略
```

访问修饰符

如下，我们定义好一个Computer的类，假设我们站在代工厂的视角，这个Computer类拥有两个属性——品牌与价格；两个行为——设置品牌与设置价格

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

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

    char _brand[20];
    float _price;
};
```

按之前的理解，现在我们自定义了一个新的类——Computer类，我们需要实例化出一个对象（特定的Computer），再通过这个对象来访问数据成员或调用成员函数，如下：

```
Computer pc;
pc.setPrice(10000); //error
pc._price; //error
```

结果发现都会报错，这是什么原因呢？事实上，class中的所有的成员都拥有自己的访问权限，分别可以用以下的三个访问修饰符进行修饰

public： //公有的访问权限，在类外可以通过对象直接访问公有成员

protected：//保护的访问权限，派生类中可以访问，在类外不能通过对象直接访问（后面学）

private：//私有的访问权限，在本类之外不能访问，比较敏感的数据设为private

注意：

- 类定义中访问修饰符的管理范围从当前行到下一个访问修饰符或类定义结束；
- class定义中如果在成员定义（或声明）之前没有任何访问修饰符，其**默认的访问权限为私有**。

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

Computer pc;
pc.setPrice(10000); //ok
pc._price; //error,因为_price是私有的
```

```
//定义类名遵循大驼峰规则
class Computer{
public:
    //类提供给外界的接口，告知了可以进行哪些操作
    //定义成员函数名遵循小驼峰规则
    void setBrand(const char * brand)
    {
        strcpy(_brand, brand);
    }

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

    void print(){
        cout << "brand:" << _brand << endl;
        cout << "price:" << _price << endl;
    }
private:
    //定义数据成员名前面加上下划线
    char _brand[20];
    float _price;
};

void test0(){
    /* int a; */
    Computer pc;
    pc.setBrand("Xiaomi");
    pc.setPrice(5699);
    pc.print();
```

struct与class的对比

学习了类的定义后，我们会发现它与C语言中的struct很相似。

- **C语言中的struct**

回顾一下C语言中struct的写法

```
1  struct Student{
2      int number;
3      char name[25];
4      int score;
5  };
6
7  void test0(){
8      struct Student s1;
9      struct Student s2;
10 }
```

采用typedef取别名后更像C++的class

```
typedef struct{
    int number;
    char name[25];
    int score;
} Student;

void test0(){
    Student s1;
    Student s2;
}
```

C中的struct只能是一些变量的集合体，可以封装数据但不能隐藏数据，而且成员不能是函数，要使用函数只能使用函数指针的方式。访问权限限制、继承性、构造析构都没有。事实上，C中struct的这种封装属于广义上的封装。面向对象的封装是指**隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别；将抽象得到的数据和操作数据的方法相结合，形成“类”。**

struct.c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    int number;
    char name[25];
    int score;
    //void print();//error
    void (*p)();
} Student;

void print(){
    printf("hello\n");
}

void test0(){
    Student s1 = {100,"bob",200,print};
    s1.p();
    Student s2;
    /* s2.number = 200; */
}
```

- **C++中的struct**

C++中的struct对C中的struct做了拓展，基本等同于class，默认访问权限是public.

- **C++中的class**

class默认访问权限是private.

成员函数的定义

- **成员函数定义的形式**

成员函数可以在类内部完成定义，也可以在类内部只进行声明，在类外部完成定义。

```
class Computer {
public:
    //成员函数
    void setBrand(const char * brand); //设置品牌

    void setPrice(float price); //设置价格

    void print();//打印信息
private:
    //数据成员
    char _brand[20];
    float _price;
};

void Computer::setBrand(const char * brand)
{
    strcpy(_brand, brand);
}
void Computer::setPrice(float price)
{
    _price = price;
}
```

实际开发中为什么采用成员函数声明和实现分离的写法?

当类中成员函数比较多（复杂），不容易看，如果只在类中进行成员函数的声明（同时配上注释），会方便理解。

```
class Computer {
public:
    //成员函数
    void setBrand(const char * brand);//设置品牌

    void setPrice(float price);//设置价格

    void print();//打印信息
private:
    //数据成员
    char _brand[20];
    float _price;
};
```

```
//虽然成员函数的定义放在了类之外
//但由于有作用域限定，仍视为类中
//返回类型  类名::成员函数名
void Computer::setBrand(const char * brand)
{
    strcpy(_brand, brand);
}
void Computer::setPrice(float price)
{
    _price = price;
}

void Computer::print(){
    cout << "brand:" << _brand << endl;
    cout << "price:" << _price << endl;
}
```

- **多文件联合编译时可能出现的错误**

为什么一般不在头文件中定义函数?

在头文件中定义一个函数时，如果多个源文件都包含了该头文件，那么在联合编译时会出现重定义错误。因为头文件的内容在每个源文件中都会被复制一份，而每个源文件都会生成对应的目标文件。在链接的阶段，会出现多个相同函数定义的情况，导致重定义错误。

对于成员函数，也存在这样的问题。

如果在头文件中采用成员函数声明和定义分离的形式，在类外部完成成员函数的实现，就会陷入这个错误。

解决方法1：在成员函数的定义前加上inline关键字，inline函数定义在头文件中是ok的

```cpp
//虽然成员函数的定义放在了类之外
//但由于有作用域限定，仍视为类中
//返回类型  类名::成员函数名
inline void Computer::setBrand(const char * brand)
{
    strcpy(_brand, brand);
}
inline void Computer::setPrice(float price)
{
    _price = price;
}

inline void Computer::print(){
    cout << "brand:" << _brand << endl;
    cout << "price:" << _price << endl;
}
```

解决方法2：将成员函数放到类内部进行定义（**说明类内部定义的成员函数就是inline函数**——默认效果）

Computer.hpp Computer1.cc Computer2.cc

```cpp
#include <iostream>//其次是C++的头文件，第三方库头文件放
using std::cout;
using std::endl;

class Computer {
public:
    //成员函数
    void setBrand(const char * brand){//设置品牌
        strcpy(_brand, brand);
    }

    void setPrice(float price){//设置价格
        _price = price;
    }

    void print(){//打印信息
        cout << "brand:" << _brand << endl;
        cout << "price:" << _price << endl;
    }
private:
    //数据成员
    char _brand[20];
    float _price;
};
```

解决方法3：**函数声明放在头文件，函数定义放在实现文件中**，就算有多个测试文件，也不会出现重定义（最常用的方式）。

之后遇到这种需求（定义一个非常复杂的类，多处都需要用到这个类）

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。

- 构造函数的作用：就是用来初始化数据成员的
- 构造函数的形式：

  没有返回值，即使是void也不能有；

  函数名与类名相同，再加上函数参数列表。

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)

对象的创建规则

1. 当类中没有显式定义构造函数时，编译器会自动生成一个默认（无参）构造函数，但并不会初始化数据成员；

   以Point类为例：

```
class Point {
public:
    void print()
    {
        cout << "(" << _ix
            << "," << _iy
            << ")" << endl;
    }
private:
    int _ix;
    int _iy;
};

void test0()
{
    Point pt;
    pt.print();
}
//运行结果显示，pt的_ix,_iy都是不确定的值
```

   Point pt；这种方式创建的对象，其数据成员没有被初始化，输出的会是不确定的值

```
class Point {
public:
    Point(){
        cout << "Point()" << endl;
    }

    void print()
    {
        cout << "(" << _ix
             << "," << _iy
             << ")" << endl;
    }
private:
    int _ix;
    int _iy;
};
```

2. 一旦当类中显式提供了构造函数时，编译器就不会再自动生成默认的构造函数；

```
class Point {
public:
    Point(){
        cout << "Point()" << endl;
        _ix = 0;
        _iy = 0;
    }
    void print()
    {
        cout << "(" << _ix
             << "," << _iy
             << ")" << endl;
    }
private:
    int _ix;
    int _iy;
};

void test0()
{
    Point pt;
    pt.print();
}
//这次创建pt对象时就调用了自定义的构造函数，而非默认构造函数
```

3. 编译器自动生成的默认构造函数是无参的，构造函数也可以接收参数，在对象创建时提供更大的自由度；

```
class Point {
public:
    Point(int ix, int iy){
```

```
        cout << "Point(int,int)" << endl;
        _ix = ix;
        _iy = iy;
    }
    void print()
    {
        cout << "(" << _ix
            << "," << _iy
            << ")" << endl;
    }
private:
    int _ix;
    int _iy;
};

void test0()
{
    Point pt;//error,没有默认的无参构造函数可供调用
    Point pt2(10,20);
    pt2.print();
}
```

4. 如果还希望通过默认构造函数创建对象， 则必须要手动提供一个默认构造函数;

```
class Point {
public:
    Point(){}

    Point(int ix, int iy){
        cout << "Point(int,int)" << endl;
        _ix = ix;
        _iy = iy;
    }
    void print()
    {
        cout << "(" << _ix
            << "," << _iy
            << ")" << endl;
    }
private:
    int _ix;
    int _iy;
};

void test0()
{
    Point pt;//ok
    Point pt2(10,20);
```

```
25      pt2.print();
26  }
```

5. 构造函数可以重载

如上，一个类中可以有多种形式的构造函数，说明构造函数可以重载。事实上，真实的开发中经常会给一个类定义各种形式的构造函数，以提升代码的灵活性（可以用多种不同的数据来创建出同一类的对象）。

```
class Point {
public:
    Point(){
        cout << "Point()" << endl;
    }

    Point(int x){
        cout << "Point(int)" << endl;
        _ix = x;
        _iy = 1000;
    }

    Point(int x,int y){
        cout << "Point(int,int)" << endl;
        _ix = x;
        _iy = y;
    }
```

```
Point pt;//创建对象时就会自动调用构造函数
/* pt.print(); */

Point pt2(1,2);
pt2.print();

Point pt3(100);
pt3.print();
```

对象的数据成员初始化

上述例子中，在构造函数的函数体中对数据成员进行赋值，其实严格意义上不算初始化（而是算赋值）。

在C++中，对于类中数据成员的初始化，推荐使用**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。

```
1   class Point {
2   public:
3       //...
4       Point(int ix = 0, int iy = 0)
5       : _ix(ix)
6       , _iy(iy)
7       {
8           cout << "Point(int,int)" << endl;
9       }
10      //...
11  };
```

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“对象的创建规则”示例代码中，有参的构造函数中 _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。

```
//初始化列表的形式,严格意义上的初始化
Point(int x,int y)
: _ix(x)
, _iy(y)
{
    cout << "Point(int,int)" << endl;
}
```

补充：**数据成员的初始化并不取决于其在初始化列表中的顺序，而是取决于声明时的顺序（与声明顺序一致）**。

- 构造函数的参数也可以按从右向左规则赋默认值，同样的，如果构造函数的声明和定义分开写，只用在声明或定义中的一处设置参数默认值，一般建议在声明中设置默认值。

```
class Point {
public:
    Point(int ix, int iy = 0);//默认参数设置在声明时
    //...
};

Point::Point(int ix, int iy)
: _ix(ix)
, _iy(iy)
{
    cout << "Point(int,int)" << endl;
}

void test0(){
    Point pt(10);
}
```

- C++11之后，普通的数据成员也可以在声明时就进行初始化。但一些特殊的数据成员初始化只能在初始化列表中进行，故一般情况下统一推荐在初始化列表中进行数据成员初始化。

```
class Point {
public:
    //...
    int _ix = 0;//C++11
    int _iy = 0;
};
```

- 数据成员初始化的顺序与其声明的顺序保持一致，与它们在初始化列表中的顺序无关（但初始化列表一般保持与数据成员声明的顺序一致）。

对象所占空间大小

之前在讲引用的知识点时，我们提过使用引用作为函数的返回值可以避免多余的复制。内置类型的变量最大也就是long double，占16个字节。但是现在我们学习了类的定义，自定义类型对象的大小可以非常大。

使用**sizeof**查看一个类的大小和查看该类对象的大小，得到的结果是相同的（**类是对象的模板**）；

```
void test0(){
    Point pt(1,2);
    cout << sizeof(Point) << endl;
    cout << sizeof(pt) << endl;
 }
```

成员函数并不影响对象的大小，对象的大小与数据成员有关（后面学习继承、多态，对象的内存布局会更复杂）；

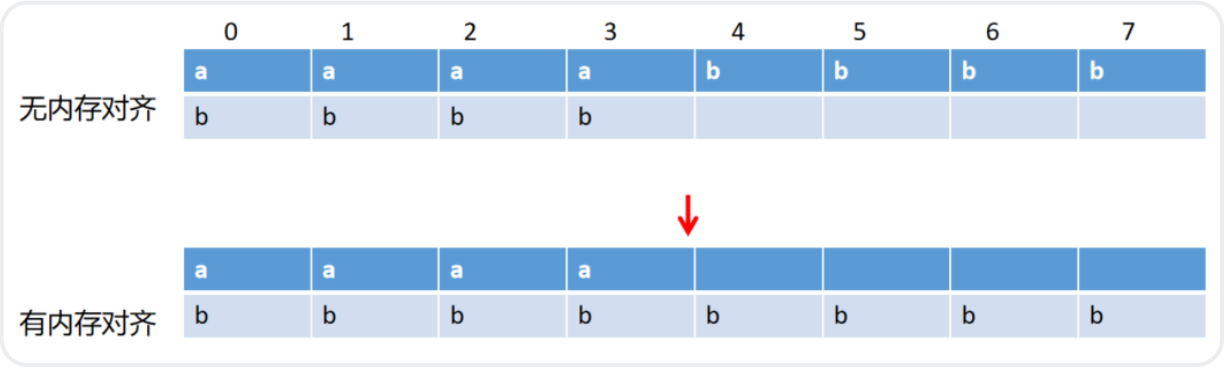
现阶段，在不考虑继承多态的情况下，我们做以下测试。发现有时一个类所占空间大小就是其数据成员类型所占大小之和，有时则不是，这就是因为有**内存对齐**的机制。

```
class A{
    int _num;
    double _price;
};
//sizeof(A) = 16

class B{
    int _num;
    int _price;
};
//sizeof(D) = 8

```

- **为什么要进行内存对齐?**

  1.平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

  2.性能原因：CPU 对内存的读取不是连续的，而是分成块读取的，块的大小只能是1、2、4、8、16 . . . 字节。若不进行内存对齐，可能需要做两次内存访问，性能会大打折扣；而进行过内存对齐仅需要一次访问。



64位系统默认以8个字节的块大小进行读取。

如果没有内存对齐机制，CPU读取_price时，需要两次总线周期来访问内存，第一次读取 _price数据前四个字节的内容，第二次读取后四个字节的内容，还要经过计算，将它们合并成一个数据。

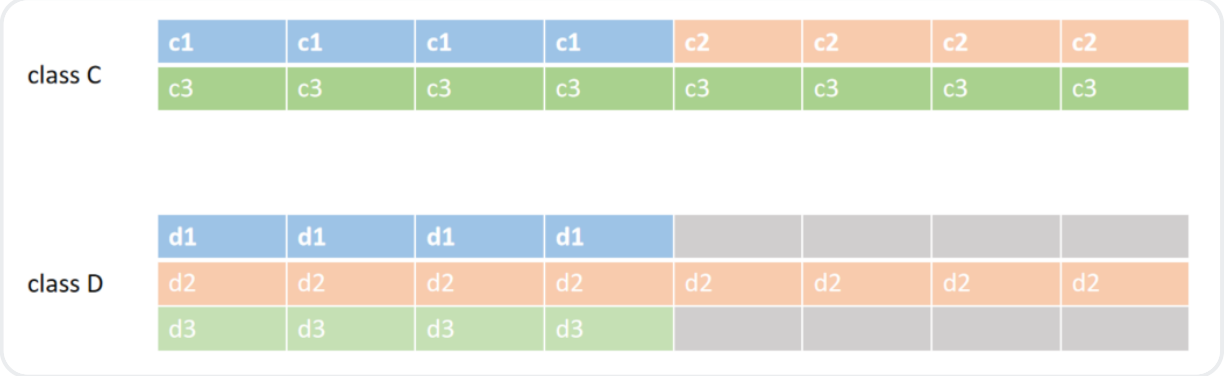
有了内存对齐机制后，以浪费4个字节的空间为代价，读取_price时只需要一次访问，所以编译器会隐式地进行内存对齐。

规则一：

按照类中占空间最大的数据成员大小的倍数对齐；

如果数据成员再多一些，我们发现自定义类型所占的空间大小还与这些数据成员的顺序有关

```
class C{
    int _c1;
    int _c2;
    double _c3;
};
//sizeof(C) = 16

class D{
    int _d1;
    double _d2;
    int _d3;
};
//sizeof(D) = 24
```



如果数据成员中有数组类型，会按照除数组以外的其他数据成员中最大的那一个的倍数对齐

```
class E{
    double _e;
    char _eArr[20];
    double _e1;
    int _e2;
};
//sizeof(E) = 48

class F{
    char _fArr[20];
};
//sizeof(F) = 20
```

再判断一下，G类所占的空间是多少?

```
class G{
    char _gArr[20];
    int _g1;
    double _g2;
};//32
```

在C语言的涉及的结构体代码中，我们可能会看到#pragma pack的一些设置，#pragma pack(n)即设置编译器按照n个字节对齐，n可以取值1,2,4,8,16.在C++中也可以使用这个设置，最终的对齐效果将按照 #pragma pack 指定的数值和类中最大的数据成员长度中，比较小的那个的倍数进行对齐。

总结:

除数组外，其他类型的数据成员中，以较大的数据成员所占空间的倍数去对齐。

内存对齐还与数据成员的声明顺序有关。

指针数据成员

类的数据成员中有指针时，意味着创建该类的对象时要进行指针成员的初始化，需要申请堆空间。

在初始化列表中申请空间，在函数体中复制内容。

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

```
    {
        strcpy(_brand,brand);
    }

private:
    char * _brand;
    double _price;
};

void test0(){
    Computer pc("Apple",12000);
}
```

思考一下，以上代码有没有问题?

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

    void print(){
        cout << "brand:" << _brand << endl;
        cout << "price:" << _price << endl;
    }

private:
    char * _brand;
    double _price;
};


void test0(){
    Computer pc("Apple",20000);
    pc.print();
}
```

代码运行没有报错，但使用memcheck工具检查发现发生了内存泄漏。有new表达式被执行，就要想到通过delete表达式来进行回收。如果没有对应的回收机制，对象被销毁时，它所申请的堆空间不会被回收，就会发生内存泄漏。

那么如何进行妥善的内存回收呢？这需要交给**析构函数**来完成。

对象的销毁

1. 析构函数：对象在销毁时，一定会调用析构函数
2. 析构函数的作用：清理对象的数据成员申请的资源（堆空间）—— 析构函数并不负责清理数据成员(系统自动完成)
3. 形式：【特殊的成员函数】
   - 没有返回值，即使是void也没有
   - 没有参数
   - 函数名与类名相同，在类名之前需要加上一个波浪号~
4. 析构函数只有一个（不能重载）
5. **析构函数默认情况下，系统也会自动提供一个**
6. **当对象被销毁时，会自动调用析构函数【非常重要】**

自定义析构函数

之前的例子中，我们没有显式定义出析构函数，但是没有问题，系统会自动提供一个默认的析构。

析构函数作为一个清理数据成员申请的堆空间的接口存在。

当数据成员中有指针时，创建一个对象，会申请堆空间，销毁对象时默认析构不够用了（造成内存泄漏），此时就需要我们自定义析构函数。在析构函数中定义堆空间上内存回收的机制，就不会发生内存泄漏。

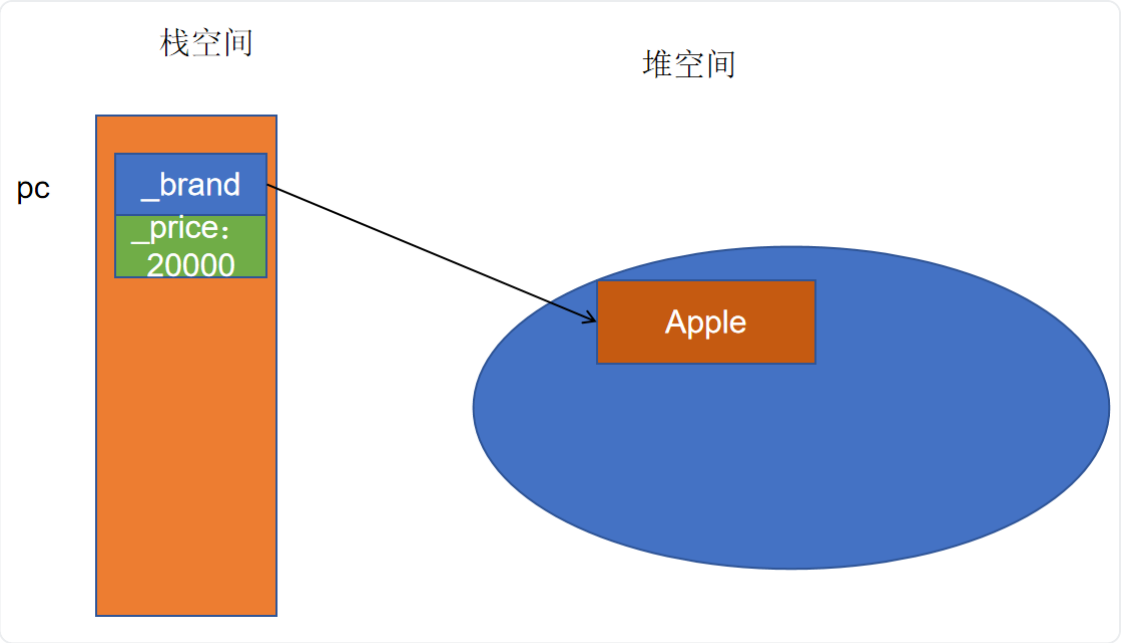
同样以Computer类为例

```
class Computer {
public:
    Computer(const char * brand, double price)
    : _brand(new char[strlen(brand) + 1]())
    , _price(price)
    {}
    ~Computer()
    {
        if(_brand){
            delete [] _brand;
            _brand = nullptr//设为空指针，安全回收
        }
        cout << "~Computer()" << endl;
    }
private:
    char * _brand;
```

```
    double _price;
};
```

```
//析构函数是用来清理数据成员所申请的堆空间资源的
//默认的析构函数并不能实现这个功能
//需要我们自己定义析构函数的内容
//析构函数是提供给我们的接口，用来清理
~Computer(){
    if(_brand){
        delete [] _brand;
        _brand = nullptr;
    }
    cout << "~Computer()" << endl;
}
```

析构函数：如果指针成员申请了堆空间，就回收这片空间，并将指针成员设为空指针，进行安全回收。



析构函数的规范写法为什么这样写呢？实际上，如果类中没有指针数据成员，即数据成员没有申请堆空间的情况下，默认的析构函数就够用了。

（1）如果没有进行安全回收这一步会引发很多问题，此时我们没有学习类与对象的更多知识，可以做个简单小实验，看看会发生什么情况，思考一下原因

```
~Computer()
{
    if(_brand){
        delete [] _brand;
        //_brand = nullptr//设为空指针，安全回收
       }
    cout << "~Computer()" << endl;
}

```

```
10  void test0(){
11      Computer pc("apple",12000);
12      pc.print();
13      pc.~Computer();//手动调用析构函数
14  }
```

```
~Computer(){
    if(_brand){
        delete [] _brand;
        /* _brand = nullptr; */
    }
    cout << "~Computer()" << endl;
}
```

```
Computer pc("Apple",20000);
cout << sizeof(pc) << endl;
pc.print();
//手动调用一次析构后，
//当pc被销毁时，还会自动调用一次析构
pc.~Computer();
```

```
ray@ubuntu:~/HaiBao/54th/day03$ ./a.out
Computer(const char *,double)
16
brand:Apple
price:20000
~Computer()
free(): double free detected in tcache 2
Aborted (core dumped)
```

——第一次手动调用析构函数时已经回收了这片堆空间，但是_brand存的地址值依然有效，当对象销毁时自动调用析构函数，依然会进入if语句，再一次试图回收这片空间，发生double free错误。

（2）如果没有对指针成员的判断，可能会有delete一个空指针的情况。尽管一些平台，delete本身会自动检查对象是否为空，如果为空就不做操作，但是在其他的一些平台这样做可能会导致风险，所以请按照规范去定义析构函数。

注意：对象被销毁，一定会调用析构函数；

调用了析构函数，对象并不会被销毁。

上述例子中手动调用了析构函数，发现之后又自动调用了一次析构函数。

那么在手动调用析构函数之后，再次调用print函数，看看会发生什么？

```
Computer pc("apple",12000);
pc.~Computer();
pc.print();
```

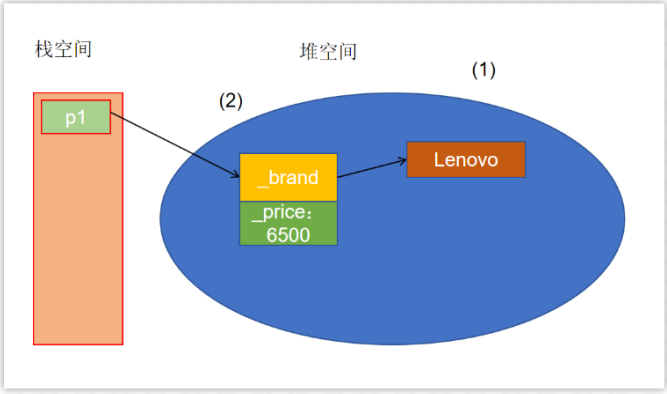
发现程序在print执行时尝试对char型空指针进行输出，导致程序中断。

```
//空指针指向的地址是不允许访问的
//输出流运算符遇到char*会自动尝试去访问
char * p3 = nullptr;
cout << p3 << endl;
```

结论：**不建议手动调用析构函数，因为容易导致各种问题，应该让析构函数自动被调用。**

构造函数和析构函数的调用时机（重点）

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 定义的静态对象**，当程序流程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 运算符创建的堆对象**，每当创建该对象时调用构造函数，**在使用 delete 删除该对象时，调用析构函数。**




```
LEAK SUMMARY:
   definitely lost: 16 bytes in 1 blocks
   indirectly lost: 7 bytes in 1 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

```
Computer * p1 = new Computer("Lenovo",6500);
p1->print();
delete p1;
p1 = nullptr;
```

本类型对象的复制

拷贝构造函数

对于内置类型而言，使用一个变量初始化另一个变量是很常见的操作

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

那么对于自定义类型，我们也希望能有这样的效果，如

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

发现这种操作也是可以通过的。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拷贝构造函数(复制构造函数)。

拷贝构造函数的定义

拷贝构造函数的形式是固定的：**类名(const 类名 &)**

1. 该函数是一个构造函数 —— 拷贝构造也是构造！
2. 该函数用一个已经存在的同类型的对象，来初始化新对象，即对对象本身进行复制

没有显式定义拷贝构造函数，这条复制语句依然可以通过，说明编译器自动提供了默认的拷贝构造函数。其形式是：

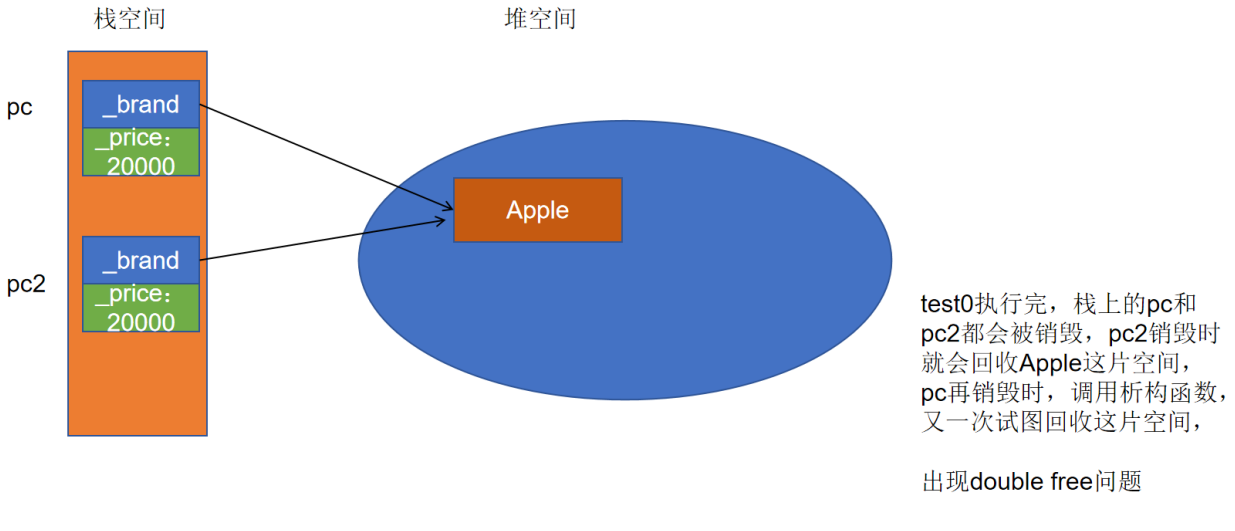
```
Point(const Point & rhs)
: _ix(rhs._ix)
, _iy(rhs._iy)
{}
```

```
Point(int,int)
Point(const Point &)
(18,45)
0x7fff729bd4d0
0x7fff729bd4d8
Point(const Point &)
(18,45)
~Point()
~Point()
~Point()
```

```
void test1(){
    Point pt1(18,45);
    Point pt2 = pt1;
    pt2.print();
    cout << &pt1 << endl;
    cout << &pt2 << endl;

    Point pt3(pt1);
    pt3.print();
}
```

拷贝构造函数看起来非常简单，那么我们尝试对Computer类的对象进行同样的复制操作。发现同样可以编译通过，但运行报错。思考一下为什么？

```
Computer pc("Acer",4500);
Computer pc2 = pc;//调用拷贝构造函数
```

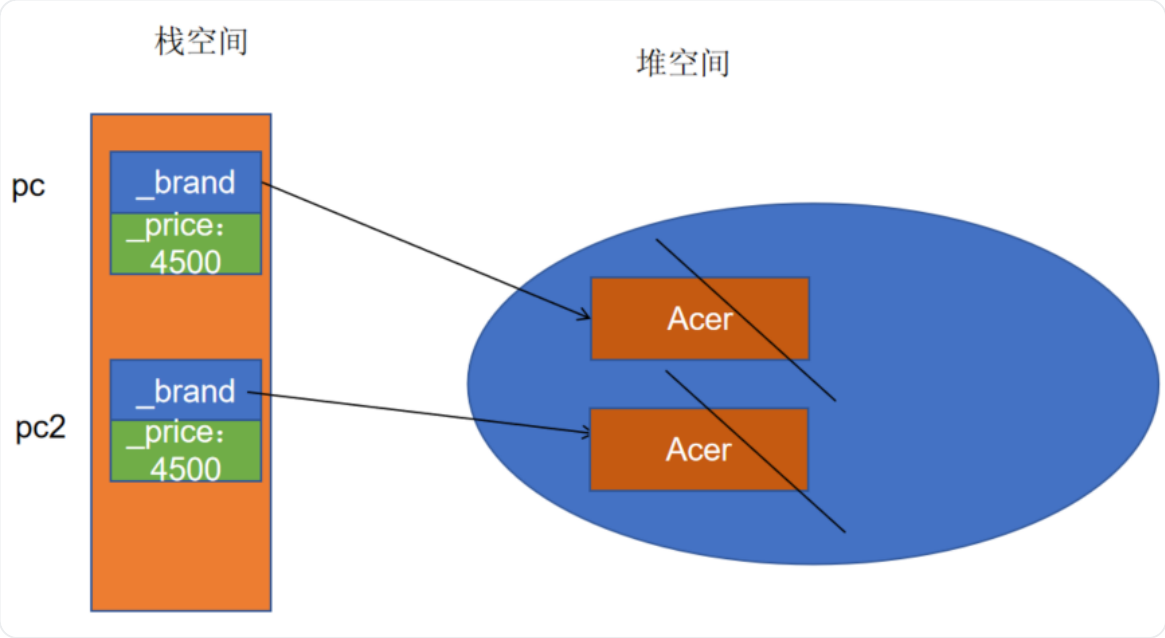


如果是默认的拷贝构造函数，pc2会对pc的_brand进行**浅拷贝**，指向同一片内存；pc2被销毁时，会调用析构函数，将这片堆空间进行回收；pc再销毁时，析构函数中又会试图回收这片空间，出现double free问题

```
Computer(const Computer & rhs)
//: _brand(rhs._brand) //_brand = rhs._brand;
: _brand(new char[strlen(rhs._brand) + 1]())
, _price(rhs._price)
{
    strcpy(_brand,rhs._brand);
    cout << "Computer(const Computer&)" << endl;
}
```

所以，如果拷贝构造函数需要显式写出时（该类有指针成员申请堆空间），在自定义的拷贝构造函数中要换成**深拷贝**的方式，先申请空间，再复制内容

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



拷贝构造函数的调用时机（重点）

1. 当使用一个已经存在的对象初始化另一个同类型的新对象时；
2. 当函数参数（实参和形参的类型都是对象），形参与实参结合时（实参初始化形参）；

   —— 为了避免这次不必要的拷贝，可以使用引用作为参数

```
//拷贝构造的第二种调用时机
//函数参数改成引用，可以避免复制
void func(Computer & rhs){
    rhs.print();
}

void test1(){
    Computer pc("apple",20000);
    func(pc);
}
```

3. 当函数的返回值是对象，执行return语句时（编译器有优化）。

   ——为了避免这次多余的拷贝，可以使用引用作为返回值，但一定要确保返回值的生命周期大于函数的生命周期

```
//拷贝构造的第三种调用时机
Computer pc3("Acer",5400);

Computer & func2(){
    return pc3;
}
```

第三种情况直接编译并不会显示拷贝构造函数的调用，但是底层实际是调用了的，加上去优化参数进行编译可以看到效果

```
g++ CopyComputer.cc -fno-elide-constructors
```

拷贝构造函数的形式探究*

思考1：拷贝构造函数是否可以去掉引用符号?

—— 类名(const 类名) 形式，首先编译器不允许这样写，直接报错

如果拷贝函数的参数中去掉引用符号，进行拷贝时调用拷贝构造函数的过程中会发生"实参和形参都是对象，用实参初始化形参"（拷贝构造第二种调用时机），会再一次调用拷贝构造函数。形成递归调用，直到栈溢出，导致程序崩溃。

假如Point类的拷贝构造函数形式：

```
Point(const Point rhs){
//……
}

Point pt(1,2);
Point pt2(pt);
```

用pt初始化pt2时，第一次调用拷贝构造，此时拷贝构造函数的形参、实参都是Point对象，实参初始化形参，触发拷贝构造的第二种调用时机，……，陷入拷贝构造的递归调用，直到栈溢出，程序崩溃

Point(const Point rhs);
Point(const Point rhs);
Point(const Point rhs);
Point(const Point rhs);
Point(const Point rhs);

构造函数是最特殊的成员函数，不是由对象来调用构造函数。而是，编译器在看到创建对象的语句时，会自动生成一段代码，在这段代码中调用构造函数，利用传入的参数来初始化对象。

思考2：拷贝构造函数是否可以去掉const?

—— 类名(类名 &) 形式　　编译器不会报错

加const的第一个用意：为了确保右操作数的数据成员不被改变

加const的第二个用意：为了能够复制临时对象的内容，因为非const引用不能绑定临时变量（右值）

```
Computer & rhs = Computer("apple",12000); //error
```

```
    cout << endl;
    //有时需用通过拷贝构造复制临时对象的内容
    Computer pc3 = Computer("ASUS",5000);
    pc3.print();
    cout << endl;
```

```
Computer(Computer & rhs)
//: _brand(rhs._brand) //_brand = rhs._brand;
: _brand(new char[strlen(rhs._brand) + 1]())
, _price(rhs._price)
{
    strcpy(_brand,rhs._brand);
    cout << "Computer(const Computer&)" << endl;
    /* rhs._price = 1; */
}
```

```
CopyComputer.cc:61:20: error: cannot bind non-const lvalue reference of typ
omputer&' to an rvalue of type 'Computer'
     Computer pc3 = Computer("ASUS",5000);
                    ^~~~~~~~~~~~~~~~~~~~
CopyComputer.cc:28:5: note:   initializing argument 1 of 'Computer::Compute
mputer&)'
     Computer(Computer & rhs)
     ^~~~~~~~
```

const引用可以绑定右值，非const引用不能绑定右值。

```
void test0(){
    int num = 1;
    &num;//可以取地址的变量称为左值

    int & ref = num;
    const int & ref2 = num;

    //临时变量、匿名变量、临时对象、匿名对象
    //&1;//不能取地址的变量称为右值
    //int & ref3 = 1;//error
    const int & ref4 = 1;
}
```

赋值运算符函数

赋值运算同样是一种很常见的运算，比如：

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

自定义类型当然也需要这种运算，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。

赋值运算符函数的形式

在上述例子中，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：

类名& operator=(const 类名 &)

对Point类进行测试时，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;
}
```

手动写出赋值运算符，再加上函数调用的提示语句。执行发现语句被输出，也就是说，**当对象已经创建时，将另一个对象的内容复制给这个对象，会调用赋值运算符函数。**

那么现在又产生了问题

首先，赋值号是一个双目运算符，如果把它视为一个函数，那么应该有两个参数。但是从赋值运算符函数的形式上看只接收了一个参数，为什么?

其次，赋值运算符函数返回类型是Point&，那么它的返回值是什么?

这两个问题引出了一个重要的知识点——**this指针**

```
//成员函数的参数列表首位会被编译器自动加上一个参数：this指针
//作用：指向本对象，准确访问本对象的成员
//形式：Point * const this
//位置：成员函数参数列表首位，不能显式写出
Point & operator=(const Point & rhs){
    this->_ix = rhs._ix;
    this->_iy = rhs._iy;
    cout << "Point& operator=(const Point &)" << endl;
    return *this;//返回本对象
}
```

this指针

- this指针的本质

this指针的本质是一个指针常量 Type* const pointer；它储存了调用它的对象的地址，不可被修改。这样成员函数才知道自己修改的成员变量是哪个对象的。

this是一个隐藏的指针，可以在类的成员函数中使用，它可以用来指向调用对象。当一个对象的成员函数被调用时，编译器会隐式地传递该对象的地址作为 this 指针。

this指针指向本对象

- this指针存在哪儿

编译器在生成程序时加入了获取对象首地址的相关代码，将获取的首地址存放在了寄存器中，这就是this指针。

- this指针的生命周期

this 指针的生命周期开始于成员函数的执行开始。当一个非静态成员函数被调用时，this 指针被自动设置为指向调用该函数的对象实例。在成员函数执行期间，this 指针一直有效。它可以被用来访问调用对象的成员变量和成员函数。this指针的生命周期结束于成员函数的执行结束。当成员函数返回时，this指针的作用范围就结束了。

要注意，this指针的生命周期与它所指向的对象的生命周期虽然并不完全相同，但是是相关的。

this指针本身只在成员函数执行期间存在，但它指向的对象可能在成员函数执行前就已经存在，并且在成员函数执行后继续存在。

如果成员函数是通过一个已经销毁或未初始化的对象调用的，this指针将是悬挂的，它的使用将会是未定义行为。

```
    void print()
    {
        cout << "(" << this->_ix
             << "," << this->_iy
             << ")" << endl;
    }
```

```
void test1(){
    Point * p = new Point(1,2);
    delete p;
    p->print();
    (*p).print();
}
```

理解以下问题：

1. 对象调用函数时，是如何找到自己本对象的数据成员的？ —— 通过this指针
2. this指针代表的是什么？ —— 指向本对象
3. this指针在参数列表中的什么位置？（默认自动加入） —— 参数列表的第一位
4. this指针的形式是什么？（指针常量） —— 类名 * const this

```
Point & operator=(const Point & rhs){
    this->_ix = rhs._ix;
    this->_iy = rhs._iy;
    cout << "Point & operator=(const Point &)" << endl;
    return *this;
}
```

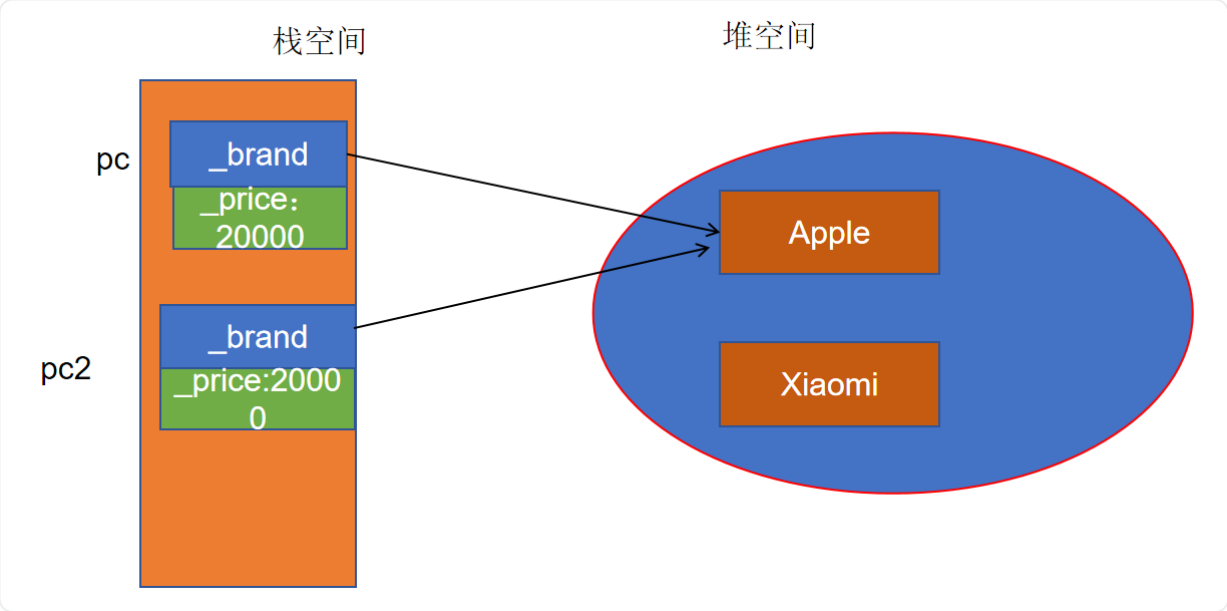
成员函数中可以加上this指针，展示本对象通过this指针找到本对象的数据成员。但是不要在参数列表中显式加上this指针，因为编译器一定会在参数列表的第一位加上this指针，如果显式再给一个，参数数量就不对了。

赋值运算符函数的定义

注意：如果对象的指针数据成员申请了堆空间，默认的赋值运算符函数就不够用了，以Computer类为例，默认的赋值运算符函数长这样

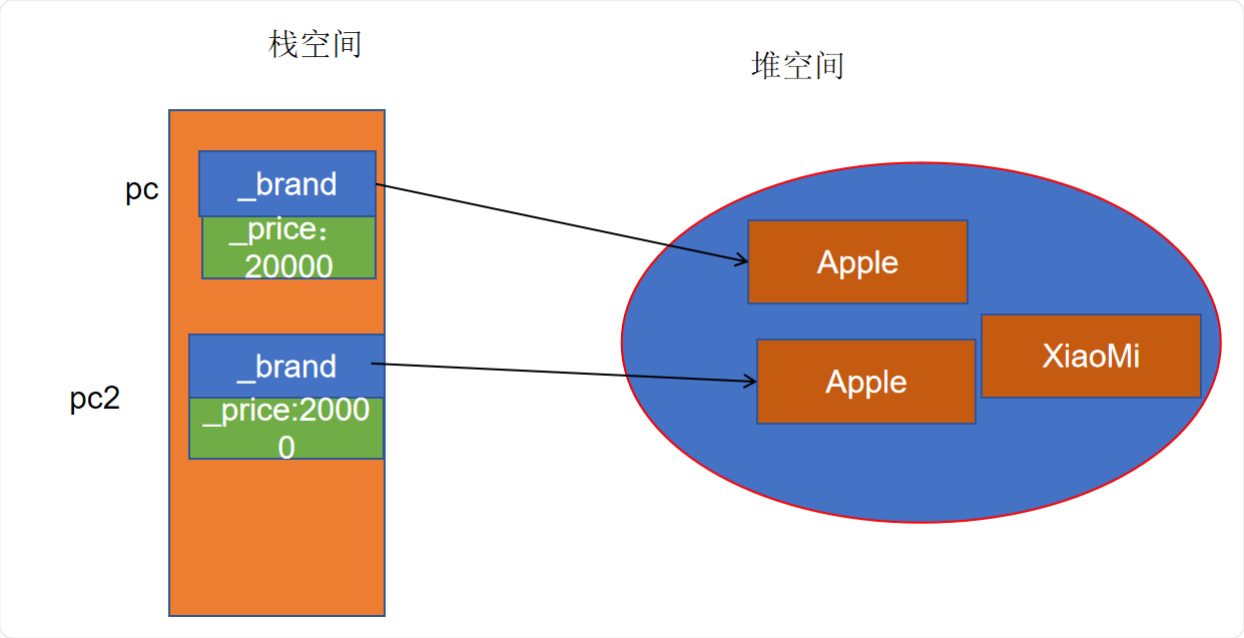
```
Computer & operator=(const Computer & rhs){
    this->_brand = rhs._brand;
    this->_price = rhs._price;
    return *this;
}
```

这里的指针成员_brand是进行的浅拷贝，会造成问题



思考：

如果直接进行深拷贝，可行吗?



会有内存泄露，需要回收掉pc2._brand原本申请的空间

如果用delete回收掉pc2._brand原本申请的空间，再进行深拷贝，是否可行？—— 还要考虑自复制

```
Computer & operator=(const Computer & rhs)
{
    //自定义的赋值运算符函数的流程
    if(this != &rhs){ //1.考虑自复制情况
        delete [] _brand; //2.回收原本的空间
        _brand = new char[strlen(rhs._brand) + 1]();//3.深拷贝
        strcpy(_brand,rhs._brand);//3.
        _price = rhs._price;
        cout << "Computer & operator=(const Computer&)" << endl;
    }
    return *this;//4.返回本对象
}
```

总结步骤—— 四步走（重点）：

1. 考虑自复制问题
2. 回收左操作数原本申请的堆空间
3. 深拷贝（以及其他的数据成员的复制）
4. 返回*this

```
Computer & operator=(const Computer & rhs){
    if(this != &rhs){
        delete [] _brand;
        _brand = new char[strlen(rhs._brand)]();
        strcpy(_brand,rhs._brand);
        _price = rhs._price;
    }
    return *this;
}
```

赋值运算符函数的形式探究*

关于赋值运算符函数的形式探究也是面试中比较可能出现的问题，以下提出四个思考：

1. 赋值运算符函数的返回必须是一个引用吗?

```
Computer operator=(const Computer & rhs)
{
    ……
    return *this;
}
```

—— 会造成一次多余的拷贝，增加不必要的开销

2. 赋值操作符函数的返回类型可以是void吗?

```
void operator=(const Computer & rhs)
{
    ……
}
```

—— 无法处理连续赋值

3. 赋值操作符函数的参数一定要是引用吗?

```
Computer & operator=(const Computer rhs)
{
    ……
    return *this;
}
```

—— 会造成一次多余的拷贝，增加不必要的开销

注意：此时讨论的是赋值运算符函数的参数形式，前提是拷贝构造是正常的。

4. 赋值操作符函数的参数必须是一个const引用吗?

```
Computer & operator=(Computer & rhs)
{
    ……
    return *this;
}
```

—— 无法避免在赋值运算符函数中修改右操作的内容，不合理

三合成原则

三合成原则很容易在面试时被问到：

拷贝构造函数、赋值运算符函数、析构函数，如果需要手动定义其中的一个，那么另外两个也需要手动定义。

特殊的数据成员

在 C++ 的类中，有4种比较特殊的数据成员，分别是常量成员、引用成员、类对象成员和静态成员，它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，在构造函数体内对const 数据成员赋值是非法的，**const数据成员需在初始化列表中进行初始化**（C++11之后也允许在声明时就初始化）。

普通的const常量必须在声明时就初始化，初始化之后就不再允许修改值；

const成员初始化后也不再允许修改值。

```
class Point {
public:
    Point(int ix, int iy)
    : _ix(ix)
    , _iy(iy)
    {}
private:
    const int _ix;
    const int _iy;
};
```

```
Point(int x,int y)
: _ix(x)  //const int _ix = x;
, _iy(y)
{
    /* _ix = x; */
    /* _iy = y; */
    cout << "Point(int,int)" << endl;
}

Point(const Point & rhs)
: _ix(rhs._ix) //const int _ix = rhs._ix;
, _iy(rhs._iy)
{
    cout << "Point(const Point &)" << endl;
}
```

引用数据成员

引用数据成员在初始化列表中进行初始化，C++11之后允许在声明时初始化（绑定）。

之前的学习中，我们知道了引用要绑定到已经存在的变量，引用成员同样如此。

```
class Point {
public:
    Point(int ix, int iy)
    : _ix(ix)
    , _iy(iy)
    , _iz(_ix)
    {}
private:
    const int _ix;
    const int _iy;
    int & _iz;
};
```

思考：构造函数再接收一个参数，用这个参数初始化引用成员可以吗？

```
Point(int x,int y)
: _ix(x)
, _iy(y)
, _iz(_ix) //int & _iz = _ix;
//, _iz(z) //要确保_iz绑定到左值
{
    cout << "Point(int,int)" << endl;
}
```

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

_iz绑定传入的z，看起来虽然是确定的值，但是由于值传递会进行复制，所以实际上是去绑定一个临时变量，临时变量的生命周期只有当前行，等到绑定的时候就是不确定的值了。

_iz绑定 _ix，因为数据成员的初始化顺序与声明顺序一致，此时 _ix已经完成了初始化，是一个确定的值，就没有问题。

对象成员

有时候，一个类对象会作为另一个类对象的数据成员被使用。比如一个直线类Line对象中包含两个Point对象。

对象成员必须在初始化列表中进行初始化。

注意：

1. 不能在声明对象成员时就去创建。
2. 初始化列表中写的是需要被初始化的对象成员的名称，而不是对象成员的类名。

```
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;
};
```

```
class Line {
public:
    Line(int x1, int y1, int x2, int y2)
    //调用Point的构造函数
    : _pt1(x1, y1) //Point _pt1(x1,y1)
    , _pt2(x2, y2) //Point _pt2(x2,y2)
    {
        cout << "Line(int,int,int,int)" << endl;
    }

    void printLine(){
        _pt1.print();
        cout << " -----> " ;
        _pt2.print();
        cout << endl;
    }
private:
    //_pt1/_pt2是Line类的对象成员
    //生成Line类对象，
    //这个对象会包含两个Point类型的成员子对象
    Point _pt1;
    Point _pt2;
};
```

注意：

如果在Line类的构造函数的初始化列表中没有显式地初始化Point类对象成员，编译器会自动去调用Point类型的默认无参构造；

如果不想用Point的无参构造，那么必须在Line类的初始化列表中对Point类的对象成员进行初始化

```
30 class Line {
31 public:
32     Line(int x1, int y1, int x2, int y2)
33     //调用Point的构造函数
34     /* : _pt1(x1, y1) //Point _pt1(x1,y1) */
35     /* , _pt2(x2, y2) //Point _pt2(x2,y2) */
36     {
37         cout << "Line(int,int,int,int)" << endl;
38     }
```

此例子中，创建一个Line类的对象，会首先调用Line的构造函数，在此过程中调用Point的构造函数完成Point类对象成员的初始化；

Line对象销毁时会先调用Line的析构函数，析构函数执行完后，再调用Point的析构函数。

```
Point(int,int)
Point(int,int)
Line(int,int,int,int)
~Line()
~Point()
~Point()
```

—— 与看起来的顺序有所不同。

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。**

静态数据成员被整个类的所有对象共享。

```
1  class Computer {
2  public:
3      //...
4  private:
5      char * _brand;
6      double _price;
7      //数据成员的类型前面加上static关键字
8      //表示这是一个static数据成员（共享）
9      static double _totalPrice;
10 };
11 double Computer::_totalPrice = 0;
```

静态成员规则：

1. private的静态数据成员无法在类之外直接访问（显然）
2. **对于静态数据成员的初始化，必须放在类外**（一般紧接着类的定义，这是规则1的特殊情况）
3. 静态数据成员初始化时不能在数据类型前面加static，在数据成员名前面要加上类名+作用域限定符
4. 如果有多条静态数据成员，那么它们的初始化顺序需要与声明顺序一致（规范）

```
private:
    char * _brand;
    double _price;
    static double _totalPrice;
};
double Computer::_totalPrice = 0;
```

```
  cout << pc3._price << endl;
  cout << pc3._totalPrice << endl;
  //static数据成员不依赖于特定的对象
  //被所有的Computer类对象共享
  //访问时也可以通过类名::
  cout << Computer::_totalPrice << endl;;
```

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

在某一个成员函数的前面加上static关键字，这个函数就是静态成员函数。静态成员函数具有以下特点：

（1）**静态成员函数不依赖于某一个对象；**

（2）静态成员函数可以通过对象调用，但更常见的方式是**通过类名加上作用域限定符调用**；

（3）静态成员函数没有this指针；

（4）**静态成员函数无法直接访问非静态的成员**，只能访问静态数据成员或调用静态成员函数（因为没有this指针）。

注：但是非静态的成员函数可以访问静态成员。

静态成员函数不能是构造函数/析构函数/赋值运算符函数/拷贝构造（因为这四个函数都会访问所有的数据成员，而static成员函数没有this指针）

```
class Computer {
public:
    Computer(const char * brand, double price)
    : _brand(new char[strcpy(brand) + 1]())
    , _price(price)
    {
        _totalPrice += _price;
    }
    //...
    //静态成员函数
    static void printTotalPrice()
    {
        cout << "totalPrice:" << _totalPrice << endl;
        cout << _price << endl;//error
    }
private:
    char * _brand;
    double _price;
    static double _totalPrice;
};
double Computer::_totalPrice = 0;
```

```
static void printTotalPrice(){
    //在静态成员函数中不能直接访问非静态成员
    /* cout << "price:" << _price << endl; */
    cout << "totalPrice:" << _totalPrice  << endl;
}
```

```
//pc.printTotalPrice();
//类名::的方式调用静态函数，更为常用
Computer::printTotalPrice();
```

想要完成Computer类的总价计算逻辑，除了构造函数之外，还需要做哪些补充呢？请结合前面学到的知识完成这个功能：无论是创建多个Computer对象，还是进行Computer对象的复制、赋值，Computer的总价始终能够正确输出。

const成员函数

之前已经介绍了 const 的应用，实际上， const 在类成员函数中还有种特殊的用法。在成员函数的参数列表之后，函数执行体之前加上const关键字，这个函数就是const成员函数。

形式： void func() const {}

```
class Computer{
public:
    //...
    void print const{
        cout << "brand:" << _brand << endl;
        cout << "price:" << _price << endl;
    }
    //...
};
```

特点：

1. const成员函数中，不能修改对象的数据成员；
2. 当编译器发现该函数是const成员函数时，会自动将this指针设置为双重const限定的指针；

```
//this指针变成了双重const限定的指针
void print() const
{
    //const成员函数中不允许对数据成员进行修改
    //_ix = 1; //error
    cout << "(" << this->_ix
        << "," << this->_iy
        << ")" ;
}
```

```
//原本的this指针类型    Point * const this
//const成员函数的this指针  const Point * const this

//前面一个const的作用是不能修改Point对象
//如果Point对象由int _ix / int _iy / int * _pint
//对于_pint,const属性是施加在指针层面，也就是说不能修改这个指针，代表着不能修改这个指针的指向，但是并不能限制它修改指向的值


//如果指针数据成员 const int * _pint的效果就是可以修改指向，不能修改指向的值
```

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，其机制与使用内置类型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(delete) 来创建(回收)堆对象。

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，就只有 const 成员函数了。因为 const 对象只能被创建、撤销和只读访问，写操作是不允许的。

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

const对象与const成员函数的规则：

1. 当类中有const成员函数和非const成员函数重载时，const对象会调用const成员函数，非const对象会调用非const成员函数；
2. 当类中只有一个const成员函数时，无论const对象还是非const对象都可以调用这个版本；
3. 当类中只有一个非const成员函数时，const对象就不能调用非const版本。

总结：如果一个成员函数中确定不会修改数据成员，就把它定义为const成员函数。

思考1：

一个类中可以有参数形式"完全相同"的两个成员函数（const版本与非const版本），既然没有报错重定义，那么它们必然是构成了重载，为什么它们能构成重载呢？

```
//this指针变成了双重const限定的指针
void print() const
{
    //const成员函数中不允许对数据成员进行修改
    //_ix = 1; //error
    cout << "(" << this->_ix
        << "," << this->_iy
        << ")" << endl;
    cout << "print const" << endl;
    cout << endl;
}
```

```
void print()
{
    cout << "(" << this->_ix
        << "," << this->_iy
        << ")" << endl;
    cout << "print" << endl;
    cout << endl;
}
```

—— 参数(this指针)是不同的。

思考2：

const成员函数中不允许修改数据成员，const数据成员初始化后不允许修改，其效果是否相同？请动手验证下面的问题

举例，如果有一个普通的指针成员，在const成员函数中，它被如何限制？

—— 在const成员函数中，如下效果：

```
/* _iy = 100; */
//const成员函数中不能修改指针成员的指向
//但是可以修改指针成员所指向的值
*_pint = 100;
/*  pint = new int(); */
```

如果这个指针成员是一个const成员，初始化之后，在一个普通的成员函数里，它被如何限制？

—— 在非const成员函数中的const数据成员，如下效果：

```
private:
    int _ix;
    int _iy;
    const int * _pint;
};
```

```
    *_pint = 100; //error
    _pint = new int(); //ok
}
```

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);
```

问题：定义好这些指针后，如何利用指针去调用Point类的成员函数print？请试验一下

```
p2->print();
(*p2).print();
```

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、初始化和使用等步骤。

- 对象数组的声明

```
Point pts[2];
```

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。

- 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}};
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，在拷贝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;
    pt2 = nullptr;
}
```

new/delete表达式的工作步骤

现在我们已经学习了new和delete的基本使用，在new/delete和malloc/free作对比时提到了二者的最本质区别 —— new/delete是表达式，而malloc/free是库函数。

那么new/delete表达式的底层工作步骤是怎样的呢？我们有必要进行了解，因为很多时候写出的bug就藏在这个工作步骤中。

new表达式的工作步骤

使用new表达式时发生的三个步骤：

1. 调用operator new标准库函数申请未类型化的空间
2. 在该空间上调用该类型的构造函数初始化对象
3. 返回指向该对象的相应类型的指针

delete表达式的工作步骤

使用delete表达式时发生的两个步骤:

1. 调用析构函数,回收数据成员申请的资源(堆空间)
2. 调用operator delete库函数回收本对象所在的空间

```
//默认的operator new
void * operator new(size_t sz){
    void * ret = malloc(sz);
    return ret;
}

//默认的operator delete
void operator delete(void * p){
    free(p);
}
```

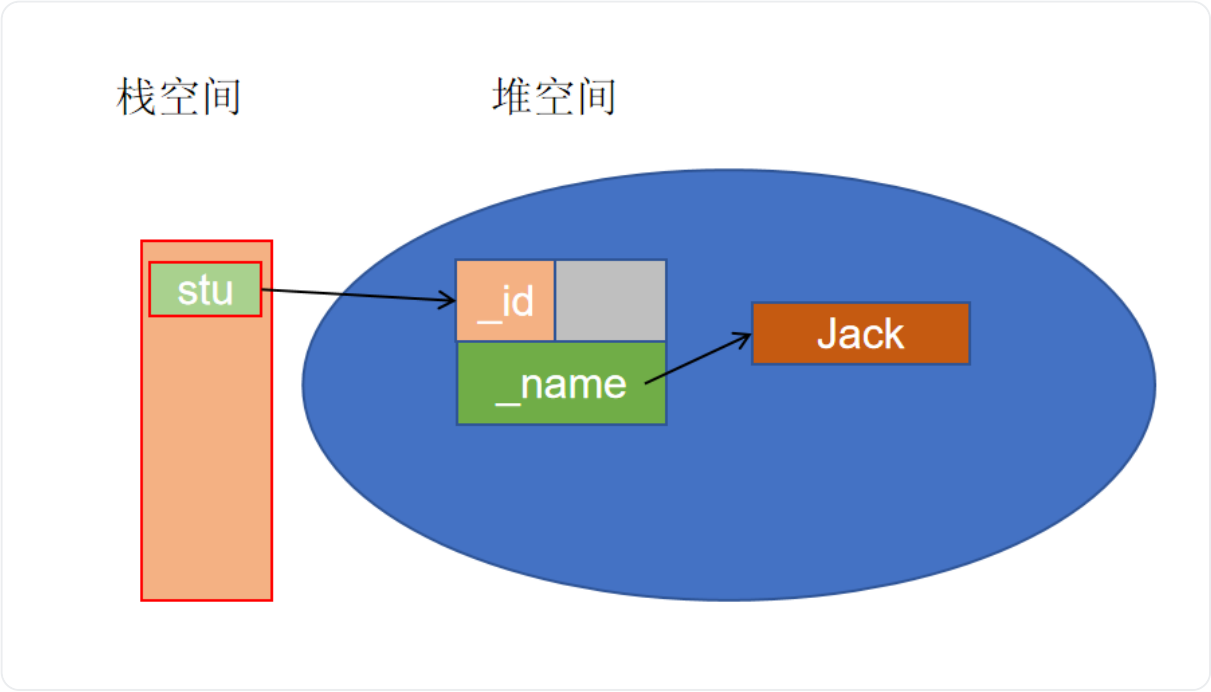
通过一个例子来认识这两个函数的用法

```
class Student
{
public:
    Student(int id, const char * name)
    : _id(id)
    , _name(new char[strlen(name) + 1]())
    {
        strcpy(_name, name);
        cout << "Student()" << endl;
    }

    ~Student()
    {
        delete [] _name;
        cout << "~Student()" << endl;
    }

    void * operator new(size_t sz)
    {
        cout << "operator new" << endl;
        void * ret = malloc(sz);
```

```
22          return ret;
23      }
24
25      void operator delete(void * pointer)
26      {
27          cout << "operator delete" << endl;
28          free(pointer);
29      }
30
31      void print() const
32      {
33          cout << "id:" << _id << endl
34              << "name:" << _name << endl;
35      }
36  private:
37      int _id;
38      char * _name;
39  };
40
41  void test0()
42  {
43      Student * stu = new Student(100, "Jackie");
44      stu->print();
45      delete stu;
46      return 0;
47  }
```

```
void test0()
{
    Student * stu = new Student(100, "Jackie");
    stu->print();
    delete stu;
}
```

```
operator new
Student()
id:100
name:Jackie
~Student()
operator delete
```



创建对象的探究

定义一个类，即使什么成员函数也不定义，依然可以创建栈对象和堆对象。之前我们知道了构造函数和析构函数会自动提供默认版本，那么能够创建堆对象、回收堆对象，说明会自动提供默认的operator new / operator delete函数。

默认的operator new / operator delete函数实际上就是通过malloc / free 实现的申请 / 回收堆空间。

请探究：

- **创建堆对象需要什么条件?**

思路：将创建、销毁对象过程中所调用到的函数——设为私有，私有的成员函数在类外就无法被直接调用了。

需要合法的operator new、operator delete、构造函数，对析构函数没有要求；在销毁堆对象的时候，才会调用析构函数。

- **创建栈对象需要什么条件?**

```
private:
    ~Student()
    {
        if(_name){
            delete [] _name;
            _name = nullptr;
        }
        cout << "~Student()" << endl;
    }
```

```
62 void test1(){
63     Student stu(190,"bob");
```

需要合法的构造函数、析构函数，对operator new/operator delete没有要求。

根据探究得出的结论，仍以Student类为例，想要实现以下需求，应该怎么做

- 只能生成栈对象，不能生成堆对象

可以将operator new/operator delete 设为私有。

- 只能生成堆对象，不能生成栈对象

可以将析构函数设为私有。

单例模式（重点*）

单例模式是23种常用设计模式中最简单的设计模式之一，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。

将单例对象创建在静态区

根据已经学过的知识进行分析：

1. 将构造函数私有；
2. 通过静态成员函数getInstance创建局部静态对象，确保对象的生命周期和唯一性；
3. getInstance的返回值设为引用，避免复制；

```
static Point & getInstance(){
    //当静态函数多次被调用
    //静态的局部对象只会被初始化一次
    //第一次调用时，静态对象会被初始化为一个对象实例
    //后续的调用中，静态局部对象已经存在，不会在初始化
    //而是直接返回已经初始化的对象实例
    static Point pt(1,2);
    return pt;
}
```

```
Point & pt = Point::getInstance();
Point & pt2 = Point::getInstance();
cout << &pt << endl;
cout << &pt2 << endl;
```

隐患：如果单例对象所占空间较大，可能会对静态区造成内存压力。

```
class Point
{
public:
    static Point & getInstance(){
        static Point pt(1,2);
        return pt;
    }

    void print() const{
        cout << "(" << this->_ix
            << "," << this->_iy
            << ")" << endl;
    }

private:
     Point(int x,int y)
    : _ix(x)
    , _iy(y)
    {
        cout << "Point(int,int)" << endl;
    }
private:
    int _ix;
    int _iy;
};

void test0(){
    Point & pt = Point::getInstance();
    pt.print();
```

```
30
31      Point & pt2 = Point::getInstance();
32      pt2.print();
33
34      cout << &pt << endl;
35      cout << &pt2 << endl;
36  }
```

将单例对象创建在堆区

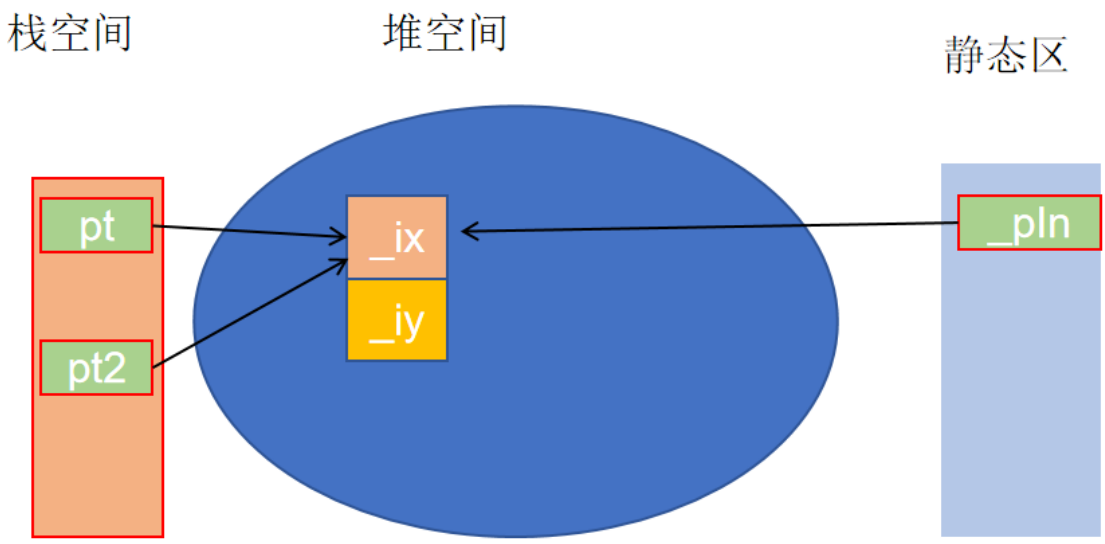
既然将单例对象创建在全局/静态区可能会有内存压力，那么为这个单例对象动态分配空间是比较合理的选择。请尝试实现代码：

分析：

1. 构造函数私有；
2. 通过静态成员函数getInstance创建堆上的对象，返回Point*类型的指针；
3. 通过静态成员函数完成堆对象的回收。

```
public:
    static Point * getInstance(){
        if(_pInstance == nullptr){
            _pInstance  = new Point(1,2);
        }
        return _pInstance;
    }

    static void destroy(){
        if(_pInstance){
            delete _pInstance;
            _pInstance = nullptr;
        }
    }
```

```
private:
    int _ix;
    int _iy;
    static Point * _pInstance;
};
Point * Point::_pInstance = nullptr;
```



单例模式的规范：不创建这些指针来接，每次只使用_pInstance来访问

```
//单例模式的规范
Point::getInstance()->print();
/* Point::getInstance()->destroy(); */
Point::destroy();
```

```
void init(int x,int y){
    _ix = x;
    _iy = y;
}
```

```
Singleton::getInstance()->init(3,9);
Singleton::getInstance()->print();
Singleton::getInstance()->init(30,90);
Singleton::getInstance()->print();
Singleton::getInstance()->init(300,900);
Singleton::getInstance()->print();
Singleton::destroy();
```

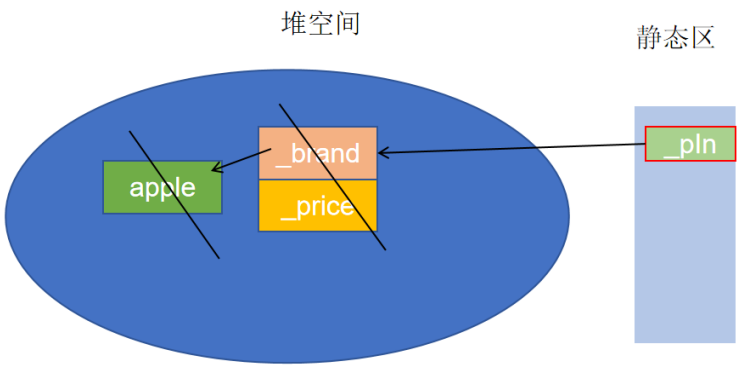
单例对象的数据成员申请堆空间

要求：实现一个单例的Computer类，包含品牌和价格信息。

```
static Computer * getInstance(){
    if(_pInstance == nullptr){
        _pInstance = new Computer("apple",20000);
    }
    return _pInstance;
}

void init(const char * brand,double price)
{
    if(_brand){
        delete [] _brand;
        _brand = nullptr;
    }
    _brand = new char[strlen(brand) + 1]();
    strcpy(_brand,brand);
    _price = price;
}
```

```
static void destroy(){
    if(_pInstance){
        delete _pInstance;
        _pInstance = nullptr;
    }
    cout << "heap delete" << endl;
}
```

```
~Computer(){
    if(_brand){
        delete [] _brand;
        _brand = nullptr;
    }
    cout << "~Computer()" << endl;
}
```



单例模式的应用场景

1、有频繁实例化然后销毁的情况，也就是频繁的 new 对象，可以考虑单例模式；

2、创建对象时耗时过多或者耗资源过多，但又经常用到的对象；

3、当某个资源需要在整个程序中只有一个实例时，可以使用单例模式进行管理（全局资源管理）。例如数据库连接池、日志记录器等；

4、当需要读取和管理程序配置文件时，可以使用单例模式确保只有一个实例来管理配置文件的读取和写入操作（配置文件管理）；

5、在多线程编程中，线程池是一种常见的设计模式。使用单例模式可以确保只有一个线程池实例，方便管理和控制线程的创建和销毁；

6、GUI应用程序中的全局状态管理：在GUI应用程序中，可能需要管理一些全局状态，例如用户信息、应用程序配置等。使用单例模式可以确保全局状态的唯一性和一致性。

C++字符串

有了类与对象的知识基础之后，我们可以来认识一下应用广泛的两种对象——C++字符串、C++动态数组。先来看看C++字符串：

字符串处理在程序中应用广泛， C 风格字符串是以 '\0' （空字符）来结尾的字符数组，在C++中通常用const char * 表示，用" "包括的认为是C风格字符串。

对字符串进行操作的 C 函数定义在头文件 <string.h> 或 < cstring > 中。常用的库函数如下：

```
//字符检查函数(非修改式操作)
size_t strlen(const char *str);//返回str的长度，不包括null结束符
//比较lhs和rhs是否相同。lhs等于rhs,返回0; lhs大于rhs，返回正数; lhs小于rhs，返回负数
int strcmp(const char *lhs, const char *rhs);
int strncmp(const char *lhs, const char *rhs, size_t count);
//在str中查找首次出现ch字符的位置；查找不到，返回空指针
char *strchr(const char *str, int ch);
//在str中查找首次出现子串substr的位置；查找不到，返回空指针
char *strstr(const char* str, const char* substr);
//字符控制函数(修改式操作)
char *strcpy(char *dest, const char *src);//将src复制给dest，返回dest
char *strncpy(char *dest, const char *src, size_t count);
char *strcat(char *dest, const char *src);//concatenates two strings
char *strncat(char *dest, const char *src, size_t count);
```

在使用时，程序员需要考虑字符数组大小的开辟，结尾空字符的处理，使用起来有诸多不便。

```
void test0()
{
    char str[] = "hello,";
    char * pstr = "world";
    //求取字符串长度
    printf("%d\n", strlen(str));

    //字符串拼接
```

```
    char * ptmp = (char*)malloc(strlen(str) + strlen(pstr) + 1);
    strcpy(ptmp, str);
    strcat(ptmp, pstr);
    printf("%s\n", ptmp);

    //查找子串
    char * p1 = strstr(ptmp, "world");
    free(ptmp);
}
```

C++风格字符串

C++ 提供了 std::string （后面简写为 string ）类用于字符串的处理。 string 类定义在 C++ 头文件< string > 中，注意和头文件 < cstring > 区分， < cstring > 其实是对 C 标准库中的 <string.h> 的封装，其定义的是一些对 C 风格字符串的处理函数。

尽管 C++ 支持 C 风格字符串，但在 C++ 程序中最好还是不要使用它们。这是因为 C 风格字符串不仅使用起来不太方便，而且极易引发程序漏洞，是诸多安全问题的根本原因。与 C 风格字符串相比， string不必担心内存是否足够、字符串长度，结尾的空白符等等。string 作为一个类出现，其集成的成员操作函数功能强大，几乎能满足所有的需求。从另一个角度上说，**完全可以把 string 当成是 C++ 的内置数据类型，放在和 int 、 double 等内置类型同等位置上。**

std::string标准库提供的一个自定义类类型basic_string，string 类本质上其实是 basic_string 类模板关于 char 型的实例化。使用起来不需要关心内存，直接使用即可。

string的构造

basic_string的常用构造——查看C++参考文档（cppreference-zh-20211231.chm）

```
basic_string(); //无参构造

basic_string( size_type count,
              CharT ch,
              const Allocator& alloc = Allocator() );  //count + 字符

basic_string( const basic_string& other,
              size_type pos,
              size_type count,
              const Allocator& alloc = Allocator() ); //接收一个basic_string对象

```

```
basic_string( const CharT* s,
              size_type count,
              const Allocator& alloc = Allocator() ); //接收一个C风格字符串
```

basic_string是一个模板类，它是std::string的基类。这里涉及到后面继承与模板的知识，现在我们掌握使用方法即可。

在创建字符串对象时，我们可以直接使用std::string作为类名，如std::string str = "hello"。这是因为C++标准库已经为我们定义了std::string这个类型的别名。

string对象常用的构造

```
string();//默认构造函数，生成一个空字符串
string(const char * rhs);//通过c风格字符串构造一个string对象
string(const char * rhs, size_type count);//通过rhs的前count个字符构造一个string对象
string(const string & rhs);//拷贝构造函数
string(size_type count, char ch);//生成一个string对象，该对象包含count个ch字符
string(InputIt first, InputIt last);//以区间[first, last)内的字符创建一个string对象
```

```
auto it2 = str.begin();
auto it3 = str.end();
string str2(it2,it3);
cout << str2 << endl;
```

还可以用拼接的方式构造string

原理：basic_string对加法运算符进行了默认重载（后续会学到），其本质是通过+号进行计算后得到一个basic_string对象，再用这个对象去创建新的basic_string对象

```
//采取拼接的方式创建字符串
//可以拼接string、字符、C风格字符串
string str3 = str1 + str2;
string str4 = str2 + ',' + str3;
string str5 = str2 + ",world!";
```

```
    //使用拼接方式创建字符串
    string str6 = str2 + ',' + str3;
    string str7 = str2 + ",world";
    cout << str6 << endl;
    cout << str7 << endl;
```

string的常用操作

```
const CharT* data() const;
const CHarT* c_str() const; //C++字符串转为C字符串

bool empty() const; //判空

size_type size() const;//获取字符数
size_type length() const;

void push_back(CharT ch);  //字符串结尾追加字符

//在字符串的末尾添加内容，返回修改后的字符串
basic_string& append(size_type count, CharT ch); //添加count个字符
basic_string& append(const basic_string& str);  //添加字符串
basic_string & append(const basic_string& str,  //从原字符串的pos位置，添加字符串的count个字符
                      size_type pos,size_type count);
basic_string& append(const charT* s);      //添加C风格字符串

//查找子串
size_type find( const basic_string& str, size_type pos = 0 ) const;  //从C++字符串的pos位开始查找C++字符串
size_type find( CharT ch, size_type pos = 0 ) const;      //从C++字符串的pos位开始查找字符ch
size_type find( const CharT* s, size_type pos, size_type count ) const;  //从C++字符串的pos位开始，去查找C字符串的前count个字符
```

实践一下string的各种操作，体会C++字符串的遍历。

```
void test1(){
    string str("hello");
    //pos是从0开始的
    size_t pos = str.find('e');
    cout << pos << endl;

    string str2("lly");
    cout << str.find(str2) << endl;
}
```

```
1
18446744073709551615
```

补充：两个basic_string字符串比较，可以直接使用==等符号进行判断

原理：basic_string对==运算符进行了默认重载（后续会学到）

```
//非成员函数
bool operator==(const string & lhs, const string & rhs);
bool operator!=(const string & lhs, const string & rhs);
bool operator>(const string & lhs, const string & rhs);
bool operator<(const string & lhs, const string & rhs);
bool operator>=(const string & lhs, const string & rhs);
bool operator<=(const string & lhs, const string & rhs);
```

string的遍历（重点）

string实际上也可以看作是一种存储char型数据的容器，对string的遍历方法是之后对各种容器遍历的一个铺垫。

（1）通过下标遍历

string 对象可以使用下标操作符[ ]进行访问。

```
//使用下标遍历
for(size_t idx = 0; idx < str.size(); ++idx){
    cout << str[idx] << " ";
}
cout << endl;
```

需要注意的是操作符[ ]并不检查索引是否有效，如果索引超出范围，会引起未定义的行为。而函数 at() 会检查，如果使用 at()的时候索引无效，会抛出 out_of_range 异常

```
string str("hello");
cout << str.at(4) << endl;  //输出o
cout << str.at(5) << endl;  //运行时抛出异常
```

（2）增强for循环遍历

针对容器，可以使用增强for循环进行遍历其中的元素。增强for循环经常和auto关键字一起使用，auto关键字可以自动推导类型。

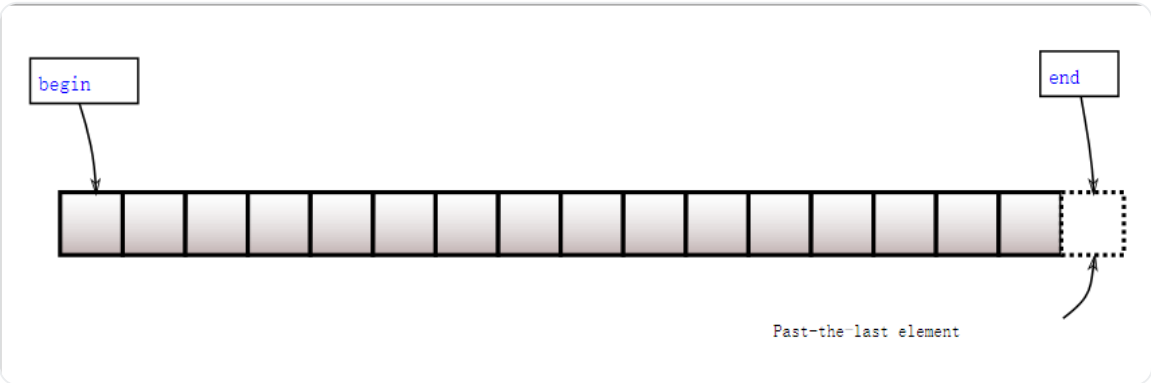
```
for(auto & ch : str){  //只要是str中的元素，就一一遍历
    cout << ch << " ";
}
cout << endl;
```

（3）迭代器方式进行遍历

string字符串利用连续空间存储字符，所以可以利用地址遍历。这里我们提出一个概念——迭代器。迭代器可以理解为是广义的指针。它可以像指针一样进行解引用、移位等操作。迭代器是容器用来访问元素的重要手段，容器都有相应的函数来获取特定的迭代器（此处可以简单理解为指向特定元素的指针）。在STL的阶段，我们会对迭代器进行更详细的讲解，现在我们只需要掌握它的基本使用即可。

begin函数返回首迭代器（指向首个元素的指针）；

end函数返回尾后迭代器（指向最后一个元素的后一位的指针）



如指针一样，迭代器也有其固定的形式。

```
//某容器的迭代器形式为 容器名::iterator
//此处auto推导出来it的类型为string::iterator
auto it = str.begin();
while(it != str.end()){
    cout << *it << " ";
    ++it;
}
cout << endl;
```

```
//2.增强for循环
//通常与auto关键字在一起使用
//auto能够自动推导出类型
//&表示引用，直接操作元素本身
//如果不加引用，操作的是元素的副本
for(auto & ch : str){
    cout << ch << " ";
    /* ch++; */
}
cout << endl;
/* cout << str << endl; */

//3.迭代器方式
//迭代器在目前阶段就理解成广义的指针
/* auto it = str.begin(); //首迭代器 */
string::iterator it = str.begin();
while(it != str.end()){
    cout << *it << " ";
    ++it;
}
cout << endl;
```

C++动态数组

C++中，std::vector（向量）是一个动态数组容器，能存放任意类型的数据。

其动态性体现在以下几个方面：

1. 动态大小：`std::vector` 可以根据需要自动调整自身的大小。它在内部管理一个动态分配的数组，可以根据元素的数量进行自动扩容或缩减。当元素数量超过当前容量时，`std::vector` 会重新分配内存，并将元素复制到新的内存位置。这使得 `std::vector` 能够根据需要动态地增长或缩小容量，而无需手动管理内存。
2. 动态插入和删除：`std::vector` 允许在任意位置插入或删除元素，而不会影响其他元素的位置。当插入新元素时，`std::vector` 会自动调整容量，并将后续元素向后移动以腾出空间。同样地，当删除元素时，`std::vector` 会自动调整容量，并将后续元素向前移动以填补空缺。
3. 动态访问：`std::vector` 提供了随机访问元素的能力。可以通过索引直接访问容器中的元素，而不需要遍历整个容器。这使得对元素的访问具有常数时间复杂度（O(1)），无论容器的大小如何。

vector的构造

vector常用的几种构造形式：

（1）无参构造，仅指明vector存放元素的种类，没有存放元素；

```
vector <int> numbers;
```

（2）传入一个参数，指明vector存放元素的种类和数量，参数是存放元素的数量，每个元素的值为该类型对应的默认值；

```
vector<long> numbers2(10); //存放10个0
```

（3）传入两个参数，第一个参数为vetor存放元素的数量，第二个参数为每个元素的值（相同）；

```
vector<long> numbers2(10, 20); //存放10个20
```

（4）通过列表初始化vector，直接指明存放的所有元素的值

```
vector<int> number3{1,2,3,4,5,6,7};
```

（5）迭代器方式初始化vector，传入两个迭代器作为参数，第一个为首迭代器，第二个为尾后迭代器；

```
vector<int> number3{1,2,3,4,5,6,7};
vector<int> number4(number3.begin(),number3.end() - 3);//推测一下，number4中存了哪些元素
```

```
/* vector<int>::iterator it = nums5.begin(); */
vector<int> nums6(nums5.begin(),nums5.end() - 3);
for(auto & num : nums6){
    cout << num << " ";
}
cout << endl;

//这种构造方法，第一个参数是首迭代器
//第二个参数是尾后迭代器
/* vector<long> nums7(arr,arr + 5); */
vector<long> nums7(&arr[0],&arr[5]);
for(auto & num : nums7){
    cout << num << " ";
}
cout << endl;
```

vector的常用操作

```
iterator begin();  //返回首位迭代器
iterator end();  //返回尾后迭代器

bool empty() const; //判空

size_type size() const; //返回容器中存放的元素个数
size_type capacity() const; //返回容器容量（最多可以存放元素的个数）

void push_back(const T& value); //在最后一个元素的后面再存放元素

void pop_back(); //删除最后一个元素
void clear(); //清空所有元素，但不释放空间
void shrink_to_fit();  //释放多余的空间（可以存放元素但没有存放运算的空间）

void reserve(size_type new_cap);//申请空间，不存放元素
```

```
nums.pop_back();
nums.shrink_to_fit();
cout << "nums'size:" << nums.size() << endl;
cout << "nums'capacity:" << nums.capacity() << endl;

nums.clear();
cout << "nums'size:" << nums.size() << endl;
cout << "nums'capacity:" << nums.capacity() << endl;

cout << endl;
```

```
//reserve可以与无参构造一起使用
//如果确定了大概需要存放多少个元素
//可以使用reserve去申请空间，但是不会存放元素
//对比有参的构造，能够节省开销
vector<int> nums2;
nums2.reserve(10);
cout << "nums2'size:" << nums2.size() << endl;
cout << "nums2'capacity:" << nums2.capacity() << endl;
for(auto & num : nums2){
    cout << num << " ";
}
cout << endl;
```

vector不仅能够存放内置类型变量，也能存放自定义类型对象和其他容器

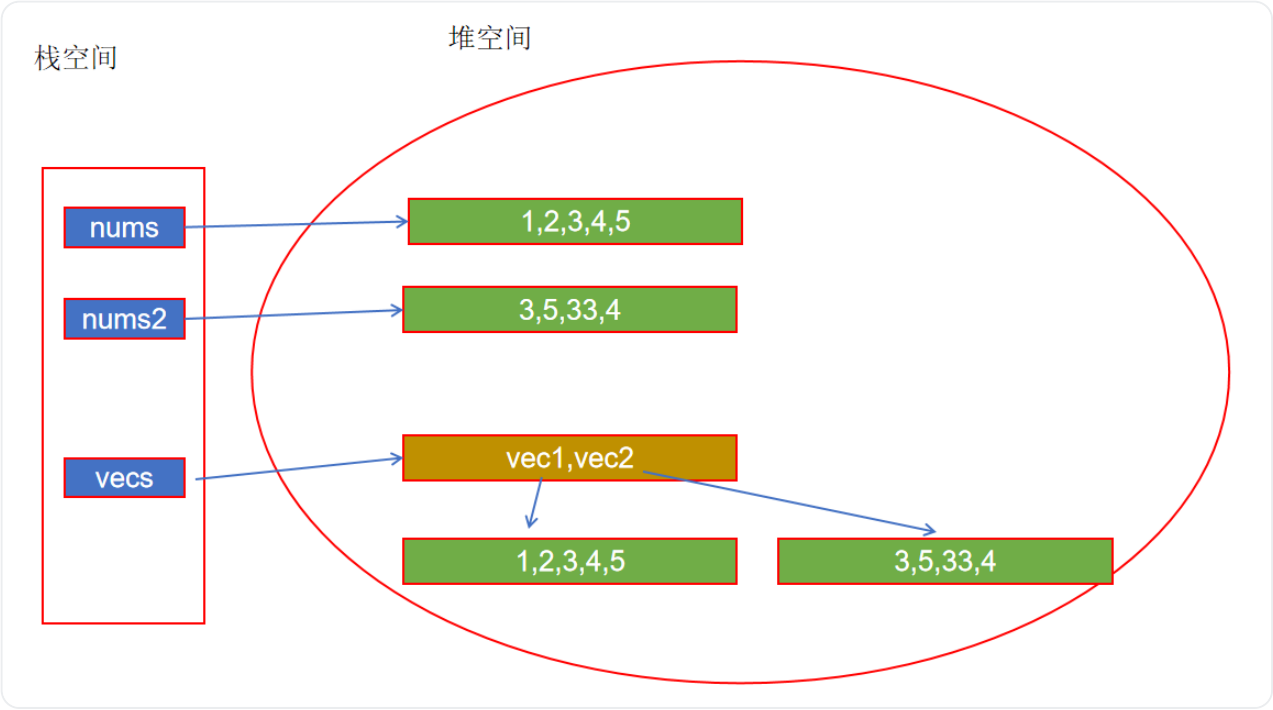
试着完成一下：

- vector中存放自定义类型Test的对象并遍历

- vector中存放string并遍历

- vector中存放vector

```
void test0(){
    vector<int> nums{1,2,3,4,5};
    vector<int> nums2{3,5,33,4};

    vector<vector<int>> vecs{nums,nums2};

    //遍历
    for(auto & vec : vecs){
        for(auto & ele : vec){
            cout << ele << " ";
        }
        cout << endl;
    }
```



vector的动态扩容

当vector存放满后，仍然追加存放元素，vector会进行自动扩容。

```
vector<int> numbers;
cout << numbers.size() << endl;
cout << numbers.capacity() << endl;

numbers.push_back(1);
cout << numbers.size() << endl;
cout << numbers.capacity() << endl;

numbers.push_back(1);
cout << numbers.size() << endl;
cout << numbers.capacity() << endl;

numbers.push_back(1);
cout << numbers.size() << endl;
cout << numbers.capacity() << endl;
//...
```

多追加一些元素，看看元素数量和容器容量的关系，思考一下vector的容量是如何增长的呢?

GCC发现vector是2倍的容量扩容机制：当vector存满后再添加新的元素，容量就会变成2倍，把新的元素存入其中。

VS上是1.5倍的扩容

—— 很多技术上具体的实现，在不同的平台上细节不同。C++标准给出功能的要求，各个编译器只需要实现此功能。

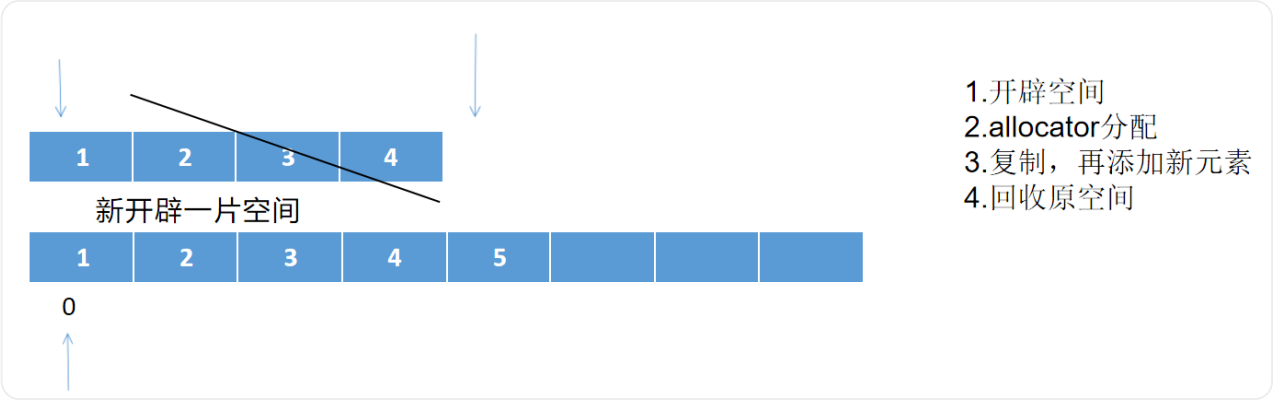
其工作步骤如下：

（1）开辟空间

（2）Allocator分配（后面STL阶段学习）

（3）复制，再添加新元素

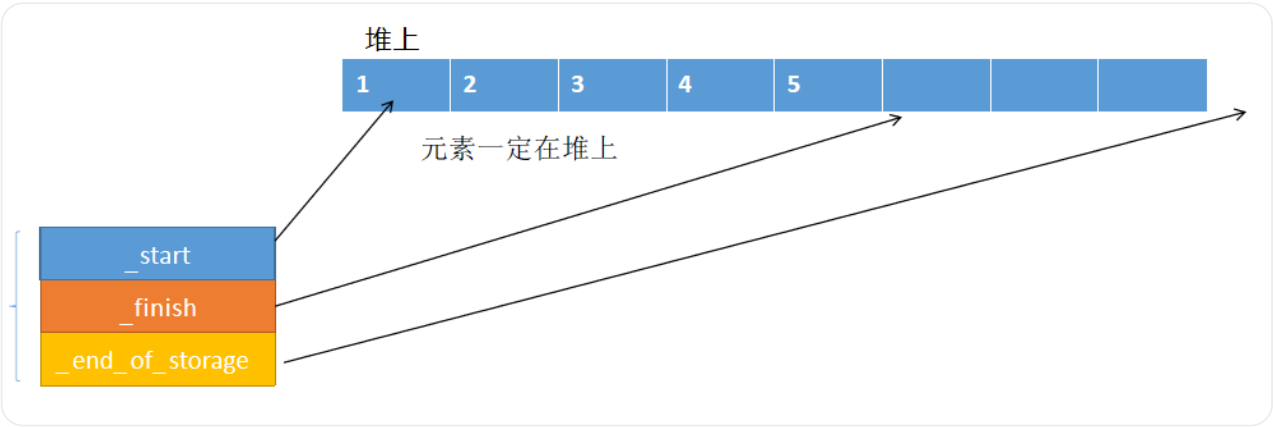
（4）回收原空间



vector的底层实现（重点*）

利用sizeof查看vector对象的大小时，发现无论存放的元素类型、数量如何，其大小始终为24个字节（64位环境）

因为vector对象是由三个指针组成



_start指向当前数组中第一个元素存放的位置

_finish指向当前数组中最后一个元素存放的下一个位置

_end_of_storage指向当前数组能够存放元素的最后一个空间的下一个位置

可以推导出

size()： _finish - _start

capacity()： _end_of_storage - start