第五章 运算符重载

友元

- 什么叫友元?

一般来说，类的私有成员只能在类的内部访问，类之外是不能访问它们的。但如果将其他类/函数设置为类的友元，那么友元类/函数就可以在前一个类的类定义之外访问其私有成员了。**用friend关键字声明友元**。

将类比作一个家庭，类的private 成员相当于家庭的秘密，一般的外人当然不允许探听这些秘密的，只有 friend 才有资格探听这些秘密。

友元的三种形式：**普通函数、成员函数、友元类**

友元之普通函数形式

示例：程序中有Point类，需要求取两个点的距离。按照设想，我们定义一个普通函数distance，接收两个Point对象作为参数，通过公式计算这两个点之间的距离。但Point的_ix和 _iy是私有成员，在类外不能通过对象访问，那么可以将distance函数声明为Point类的友元函数，之后就可以在distance函数中访问Point的私有成员了。

```
class Point{
public:
    Point(int x, int y)
    : _ix(x)
    , _iy(y)
    {}

    friend
    float distance(const Point & lhs, const Point & rhs);
private:
    int _ix;
    int _iy;
};

float distance(const Point & lhs, const Point & rhs){
```

```
16      return sqrt((lhs._ix - rhs._ix)*(lhs._ix - rhs._ix) +
17                  (lhs._iy - rhs._iy)*(lhs._iy - rhs._iy));
18  }
```

```
class Point{
public:
    Point(int x,int y)
    : _ix(x)
    , _iy(y)
    {}

    friend
    float distance(const Point & lhs, const Point & rhs);
private:
    int _ix;
    int _iy;
};

//友元的普通函数形式
float distance(const Point & lhs, const Point & rhs){
    return sqrt(pow(lhs._ix - rhs._ix,2) +
                pow(lhs._iy - rhs._iy,2));
}
```

友元之成员函数形式

假设类A有一个成员函数，该成员函数想去访问另一个类B类中的私有成员变量。这时候则可以在第二个类B中，声明第一个类A的那个成员函数为类B的友元函数，这样第一个类A的某个成员函数就可以访问第二个类B的私有成员变量了。

我们试验一下，以另一种方式实现上面的需求，如果distance函数不再是一个普通函数，而是Line类的一个成员函数，也就是说需要在一个类（Line）的成员函数中访问另一个类（Point）的私有成员，那么又该如何实现呢？

- 如果将Point类定义在Line类之前，Line类的成员函数要访问Point类的私有成员，需要在Point类中将Line的这个成员函数设为友元函数——此时编译器并不认识Line类；
- 如果将Line类定义在Point类之前，那么distance函数需要接受两个const Point &作为参数——此时编译器不认识Point类；

解决方法：

——在Line前面做一个Point类的前向声明；

——但如果将distance的函数体写在Line类中，编译器虽然知道了有一个Point类，但并不知道Point类具体有什么成员，所以此时在函数体中访问_ix、 _iy都会报错，编译器并不认识它们；

思考一下，有什么办法可以解决这个问题呢？

```
//前向声明
class Point;

class Line{
public:
    float distance(const Point & lhs, const Point & rhs){
        return sqrt((lhs._ix - rhs._ix)*(lhs._ix - rhs._ix) + //error
                    (lhs._iy - rhs._iy)*(lhs._iy - rhs._iy));
    }
};

class Point{
public:
    Point(int x, int y)
    : _ix(x)
    , _iy(y)
    {}

    friend float Line::distance(const Point & lhs, const Point & rhs);
private:
    int _ix;
    int _iy;
};

```

```cpp
class Point;//前向声明

class Line{
public:
    //友元的成员函数形式
    float distance(const Point & lhs, const Point & rhs);
};

class Point{
public:
    Point(int x,int y)
        : _ix(x)
          , _iy(y)
    {}

    friend float Line::distance(const Point & lhs, const Point & rhs);
private:
    int _ix;
    int _iy;
};

float Line::distance(const Point & lhs, const Point & rhs){
    return sqrt(pow(lhs._ix - rhs._ix,2) +
                pow(lhs._iy - rhs._iy,2));
}

void test0(){
    Point pt(0,0);
    Point pt2(3,4);
    Line l;
    cout << l.distance(pt,pt2) << endl;
```

补充：

前向声明的用处：进行了前向声明的类，可以以引用或指针的形式作为函数的参数，只要不涉及到对该类对象具体成员的访问，编译器可以通过。

（让编译器认识这个类，但是注意如果只进行前向声明，这个类的具体实现没有的话，无法使用这个类的对象，无法创建）

注意：友元的声明要注意和函数的形式完全对应上。

友元类

如上的例子，假设类 Line 中不止有一个 distance 成员函数，还有其他成员函数，它们都需要访问Point 的私有成员，如果还像上面的方式一个一个设置友元，就比较繁琐了，可以直接将 Line 类设置为 Point 的友元类，在工作中这也是更常见的方法。

```cpp
class Point {
    //...
    friend class Line;
    //...
};
```

在Point类中声明Line类是本类的友元类，那么Line类中的所有成员函数中都可以访问Point类的私有成员。一次声明，全部解决。

```cpp
    //声明了Line类是Point类的友元类
    friend class Line;
private:
    int _ix;
    int _iy;
};

class Line{
public:
    float distance(const Point & lhs, const Point & rhs);
    void setX(Point & rhs,int x){
        rhs._ix = x;
    }
};

float Line::distance(const Point & lhs, const Point & rhs){
    return sqrt(pow(lhs._ix - rhs._ix,2) +
                pow(lhs._iy - rhs._iy,2));
}
```

不可否认，友元将类的私有成员暴露出来，在一定程度上破坏了信息隐藏机制，似乎是种“副作用很大的药”，但俗话说“良药苦口”。好工具总是要付出点代价的，拿把锋利的刀砍瓜切菜，总是要注意不要割到手指的。

友元的存在，使得类的接口扩展更为灵活，使用友元进行运算符重载从概念上也更容易理解一些，而且， C++ 规则已经极力地将友元的使用限制在了一定范围内。

友元的特点

1. **友元不受类中访问权限的限制**——可访问私有成员
2. **友元破坏了类的封装性**
3. **不能滥用友元 ， 友元的使用受到限制**
4. **友元是单向的**——A类是B类的友元类，则A类成员函数中可以访问B类私有成员；但并不代表B类是A类的友元类，如果A类中没有声明B类为友元类，此时B类的成员函数中并不能访问A类私有成员

5. **友元不具备传递性**——A是B的友元类，B是C的友元类，无法推断出A是C的友元类
6. **友元不能被继承**——因为友元破坏了类的封装性，为了降低影响，设计层面上友元不能被继承

运算符重载

运算符重载的介绍

C++ 预定义中的运算符的操作对象只局限于基本的内置数据类型，但是对于自定义的类型是没有办法操作的。当然我们可以定义一些函数来实现这些操作，但考虑到用运算符表达含义的方式很简洁易懂，当定义了自定义类型时，也希望这些运算符能被自定义类类型使用，以此提高开发效率，增加代码的可复用性。为了实现这个需求，C++提供了运算符重载。其指导思想是：**希望自定义类类型在操作时与内置类型保持一致**。

能够重载的运算符有42个

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	>>	<<	>>=
<<=	==	!=	>=	<=	&&
||	++	--	->*	->	,
[]	()	new	delete	new[]	delete[]

不能重载的运算符包括

.　成员访问运算符
.*　成员指针访问运算符
?:　三目运算符
::　作用域限定符
sizeof　长度运算符

记法：**带点的运算符不能重载，加上sizeof**

运算符重载的规则与形式（重点）

- **运算符重载有以下规则**

1. 运算符重载时，**其操作数类型必须要是自定义类类型或枚举类型**——不能是内置类型
2. 其优先级和结合性还是固定不变的　　a ==　b + c
3. **操作符的操作数个数是保持不变的**
4. **运算符重载时，不能设置默认参数**　——如果设置了默认值，其实也就是改变了操作数的个数
5. 逻辑与 && 逻辑或 || 就不再具备短路求值特性，进入函数体之前必须完成所有函数参数的计算，不推荐重载
6. 不能臆造一个并不存在的运算符　　@ $ 、

- **运算符重载的形式**

运算符重载的形式有三种：

1. **采用友元函数的重载形式**
2. 采用普通函数的重载形式
3. **采用成员函数的重载形式**

以加法运算符为例，认识这三种形式。

+运算符重载

需求：实现一个复数类，复数分为实部和虚部，重载+运算符，使其能够处理两个复数之间的加法运算（实部加实部，虚部加虚部）

友元函数实现

我们可以定义一个普通函数，接收两个复数类对象，在这个函数中定义计算逻辑。因为要在类外访问Complex的私有成员，故可以将这个普通函数设为Complex的友元函数

```
class Complex{
```

```cpp
    //...
    friend Complex add(const Complex & lhs, const Complex & rhs);
    //...
};

Complex add(const Complex & lhs, const Complex & rhs){
    //...
}

void test0(){
    Complex cx(1,2);
    Complex cx2(3,4);
    Complex cx3 = add(cx,cx2); //这样就可以计算两个Complex对象的加法了
}
```

还想要更直观、更简洁一些，那么可以定义一个相应的运算符重载函数（operator+），就可以直接使用+完成这两个对象的加法运算了

```cpp
class Complex{
    //...
    friend Complex operator+(const Complex & lhs, const Complex & rhs);
    //...
};

Complex operator+(const Complex & lhs, const Complex & rhs){
    //...
}

void test0(){
    Complex cx(1,2);
    Complex cx2(3,4);
    Complex cx3 = cx + cx2; //看上去和内置类型的计算一样了
    //Complex cx3 = operator(cx,cx2);   //本质上是调用了operator+函数
}
```

```
    }
    friend
    Complex operator+(const Complex & lhs, const Complex & rhs);
private:
    int _real;
    int _image;
};

Complex operator+(const Complex & lhs, const Complex & rhs){
    return Complex(lhs._real + rhs._real,
                   lhs._image + rhs._image);
}

void test0(){
    Complex c1(1,2);
    Complex c2(3,4);
    /* c1 = c2; */
    Complex c3 = c1 + c2;
    c3.print();
```

运算符重载的本质是定义一个运算符重载函数，步骤如下

1. 先确定这个函数的返回值是什么类型（加法运算返回值应该是一个临时的Complex对象，所以此处返回类型为Complex）
2. 再写上函数名（operator + 运算符，此处就是**operator+**）
3. 再补充参数列表（考虑这个运算符有几个操作数，此处加法运算应该有两个操作数，分别是两个Complex对象，因为加法操作不改变操作数的值，可以用const引用作为形参）
4. 最后完成函数体的内容（此处直接调用Complex构造函数创建一个新的对象作为返回值）。

——在定义的operator+函数中需要访问Complex类的私有成员，要进行友元声明

像加号这一类不会修改操作数的值的运算符，倾向于采用友元函数的方式重载。

普通函数实现

在一个普通函数中想要访问一个类的私有成员，也可以给这个类添加一些公有的get系列函数，因为这些成员函数是可以访问私有成员的，而在类外可以通过对象直接调用这些成员函数，也就能获取到私有成员了。

实际工作中不推荐使用，因为这样做几乎完全失去了对私有成员的保护。

```
class Complex {
public:
    //...
    double getReal() const { return _real; }
    double getImage() const { return _image; }
    //...
};

Complex operator+(const Complex & lhs, const Complex & rhs)
{
    return Complex(lhs.getReal() + rhs.getReal(),
                   lhs.getImage() + rhs.getImage());
}

void test0()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;//ok
}
```

```
    int getReal() const{
        return _real;
    }

    int getImage() const{
        return _image;
    }
private:
    int _real;
    int _image;
};

Complex operator+(const Complex & lhs, const Complex & rhs){
    return Complex(lhs.getReal() + rhs.getReal(),
                   lhs.getImage() + rhs.getImage());
}
```

成员函数实现

还可以将运算符重载函数定义为Complex类的成员函数

```
class Complex{
public:
    //...
    Complex operator+(const Complex & rhs)
    {
        return Complex(_real + rhs._real, _image + rhs._image);
    }
};
```

```
Complex operator+(const Complex & rhs){
    return Complex(_real + rhs._real,
                   _image + rhs._image);
}
```

```
Complex c1(1,2);
Complex c2(3,4);
/* c1 = c2; */
Complex c3 = c1 + c2;
c3.print();

Complex c4 = c1.operator+(c2);//本质
c4.print();
```

这种写法要注意的是，加法运算符的左操作数实际上就是this指针所指向的对象，在参数列表中只需要写上右操作数

```
Complex cp1(1,2);
Complex cp2(3,4);
Complex cp = cp1 + cp2;   //本质是Complex cp = cp1.operator+(cp2)
```

——思考，如果我们写出了这样的代码，是否可以通过呢？ —— 可以通过，但是要避免

```
class Complex{
public:
    //...
    Complex operator+(const Complex & rhs)
    {
        return Complex(_real - rhs._real, _image - rhs._image);
    }
};
```

明明是加操作符，但函数内却进行的是减法运算，这是合乎语法规则的，不过却有悖于人们的直觉思维，会引起不必要的混乱。

因此，除非有特别的理由，**尽量使重载的运算符与其内置的、广为接受的语义保持一致。**

+=运算符重载

如果要让Complex对象能够使用+=运算符进行计算，需要对+=运算符进行重载。

像+=这一类会修改操作数的值的运算符，倾向于采用成员函数的方式重载。

同样按照上述步骤来定义运算符重载函数，请尝试实现：

```
Complex & operator+=(const Complex & rhs){
    cout << "operator+=" << endl;
    _real += rhs._real;
    _image += rhs._image;
    return *this;
}
```

重载形式的选择（重要）

- 不会修改操作数的值的运算符，倾向于采用友元函数的方式重载
- 会修改操作数的值的运算符，倾向于采用成员函数的方式重载
- **赋值**=、**下标**[ ]、**调用**()、**成员访问**->、**成员指针访问**->* **运算符必须是成员函数形式重载**
- 与给定类型密切相关的运算符，如递增、递减和解引用运算符，通常应该是成员函数形式重载
- 具有对称性的运算符可能转换任意一端的运算对象，例如相等性、位运算符等，通常应该是友元形式重载

++运算符重载

自增运算符有前置++和后置++两种形式，依然按照内置类型先分析计算逻辑，再类比这个计算逻辑去定义运算符重载函数

int a = 5;

a++的操作是使a的值增为6，但是这个表达式的返回值却是一个临时变量（a的值改变前的副本，即5）

++a则是使a的值增加到6，直接返回变量a本身

类比Complex，写出++运算符重载函数。按照我们目前的认知，前置++和后置++都应该选择成员函数的形式进行重载。

但是前置形式和后置形式都是只有一个操作数（本对象），参数完全相同的情况下，只有返回类型不同不能构成重载。前置形式和后置形式的区分只能通过设计层面人为地加上区分。

```
    //前置++的形式
    Complex& operator++(){
        cout << "Complex & operator++()" << endl;
        ++_real;
        ++_image;
        return *this;
    }

    //后置++的形式
    //参数列表中要多加一个int
    //与前置形式进行区分
    Complex operator++(int){
        cout << "Complex operator++(int)" << endl;
        Complex tmp(*this);
        ++_real;
        ++_image;
        return tmp;
    }
```

```
//前置++
Complex & operator++(){
    cout << "operator++()" << endl;
    ++_real;
    ++_image;
    return *this;
}

//后置++运算符重载函数的参数列表中加入一个int
//与前置++进行区分
//返回类型是对象，不是引用
Complex operator++(int){
    cout << "operator++(int)" << endl;
    Complex temp(*this);
    ++_real;
    ++_image;
    return temp;
}
```

[ ]运算符重载

需求：定义一个CharArray类，模拟char数组，需要通过下标访问运算符能够对对应下标位置字符进行访问。

- 分析[ ]运算符重载函数的返回类型，因为通过下标取出字符后可能进行写操作，需要改变CharArray对象的内容，所以应该用char引用；
- [ ]运算符的操作数有两个，一个是CharArray对象，一个是下标数据，ch[0]的本质是ch.operator[] (0);

函数体实现需要考虑下标访问越界情况，若未越界则返回对应下标位置的字符，若越界返回终止符。

```
class CharArray{
public:
    CharArray(const char * pstr)
    : _capacity(strlen(pstr) + 1)
    , _data(new char[_capacity]())
    {
        strcpy(_data,pstr);
    }

    ~CharArray(){
        if(_data){
            delete [] _data;
            _data = nullptr;
        }
    }

    //"hello"来创建
    //capacity = 6
    //下标只能取到 4
    char & operator[](size_t idx){
        if(idx < _capacity - 1){
            return _data[idx];
        }else{
            cout << "out of range" << endl;
            static char nullchar = '\0';
            return nullchar;
        }
    }

    void print() const{
        cout << _data << endl;
    }
private:
    size_t _capacity;
```

```
35        char * _data;
36    };
```

思考，如果要禁止CharArray对象通过下标访问修改字符数组中的元素，应该怎么办?

```
//第一个const的效果， 函数的返回值是一个const引用
//调用函数得到结果不允许进行写操作
//第二个const的效果， 在函数中不能修改数据成员
//const对象只能调用const成员函数
const char & operator[](size_t idx) const{
    if(idx < _capacity - 1){
        /* _capacity = 100; */
        /* _data = new char[100](); */
        //_data[idx] = 'Y';  //可以修改
        return _data[idx];
    }else{
        cout << "out of range" << endl;
        static char nullchar = '\0';
        return nullchar;
    }
}
```

输入输出流运算符重载（重要）

输出流运算符 <<

在之前的例子中，我们如果想打印一个对象时，常用的方法是通过定义一个 print 成员函数来完成，但使用起来不太方便。我们希望打印一个对象，与打印一个整型数据在形式上没有差别(如下例子)，那就必须要重载 << 运算符。

需求：

对于Complex对象，希望像内置类型数据一样，使用输出流运算符可以对其进行输出

分析：

- 输出流运算符有两个操作数，左操作数是输出流对象，右操作数是Complex对象。如果将输出流运算符函数写成Complex的成员函数，会带来一个问题，成员函数的第一个参数必然是this指针，也就是说Complex对象必须要作为左操作数。这种方式完成重载函数后，只能cx << cout这样来使用，与内置类型的使用方法不同，所以**输出流运算符的重载采用友元形式。**
- cout << cx这个语句的返回值是cout对象，因为cout是全局对象，不允许复制，所以返回类型为ostream &;
- 参数列表中第一个是左操作数（cout对象），写出类型并给出形参名；第二个是右操作数（Complex对象），因为不会在输出流函数中修改它的值，采用const引用；

- 将Complex的信息通过连续输出语句全部输出给os，最终返回os（注意，使用cout输出流时通常会带上endl，那么在函数定义中就不加endl，以免多余换行）

```
1  class Point {
2  public:
3      //...
4      friend ostream & operator<<(ostream & os, const Point & rhs);
5
6  private:
7      int _x;
8      int _y;
9  };
10
11 ostream & operator<<(ostream & os, const Point & rhs)
12 {
13     os << "(" << rhs._x << "," << rhs._y << ")";
14     return os;
15 }
16
17 void test0(){
18     Point pt(1,2);
19     cout << pt << endl; //本质形式： operator<<(cout,pt) << endl;
20 }
```

——为了和内置类型的使用方式保持一致，输出流运算符重载采用友元形式

```
    friend
    ostream & operator<<(ostream & os, const Complex & rhs);
private:
    int _real;
    int _image;
};

ostream & operator<<(ostream & os, const Complex & rhs){
    os << rhs._real << "+" << rhs._image << "i";
    return os;
}


void test0(){
    Complex c1(1,2);
    cout << c1 << endl;
    operator<<(cout,c1) << endl;
```

——如果采用成员形式进行运算符重载，那么自定义类型对象必然会作为第一个参数

```
/* c1 << cout; */
/* c1.operator<<(cout); */
```

输入流运算符 >>

需求：对于Complex对象，希望像内置类型数据一样，使用输入流运算符可以对其进行输入

实现过程与输出流类似

```
class Point {
public:
    //...
    friend istream & operator>>(istream & is, Point & rhs);
private:
    int _x;
    int _y;
};

istream & operator>>(istream & is, Point & rhs)
{
    is >> rhs._x;
    is >> rhs._y;
    return is;
}
```

——如果不想分开输出实部和虚部，也可以直接连续输入，空格符、换行符都能作为分隔符

```
istream & operator>>(istream & is, Point & rhs)
{
    is >> rhs._x >> rhs._y;
    return is;
}
```

但是还有个问题需要考虑，使用输入流时需要判断是否是合法输入

——可以封装一个函数判断接收到的是合法的int数据，在>>运算符重载函数中调用，请结合前面输入流的知识试着实现

```
//第二个参数需要是引用形式，要确保写入的内容传给数据成员
void readInputInt(istream & is, int & number){
    cout << "please input a int number" << endl;
    //实际的输入操作在这里
    while(is >> number, !is.eof()){
        if(is.bad()){
            cout << "istream has broken" << endl;
            return;
        }else if(is.fail()){
            is.clear(); //恢复流的状态
            is.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
            cout << "please input a int number" << endl;
        }else{
            break;
        }
    }
}
```

```
istream & operator>>(istream & is, Complex & rhs){
    cout << "please input a real:" << endl;
    /* is >> rhs._real; */
    readInputInt(is,rhs._real);
    cout << "please input an image:" << endl;
    /* is >> rhs._image; */
    readInputInt(is,rhs._image);
    return is;
}
```

成员访问运算符

成员访问运算符包括箭头访问运算符 -> 和解引用运算符 * ，它们是指针操作最常用的两个运算符。我们先来看箭头运算符 ->

箭头运算符只能以成员函数的形式重载，其返回值必须是一个指针或者重载了箭头运算符的对象。来看下例子：

两层结构下的使用

例子：建立一个双层的结构，MiddleLayer含有一个Data*型的数据成员

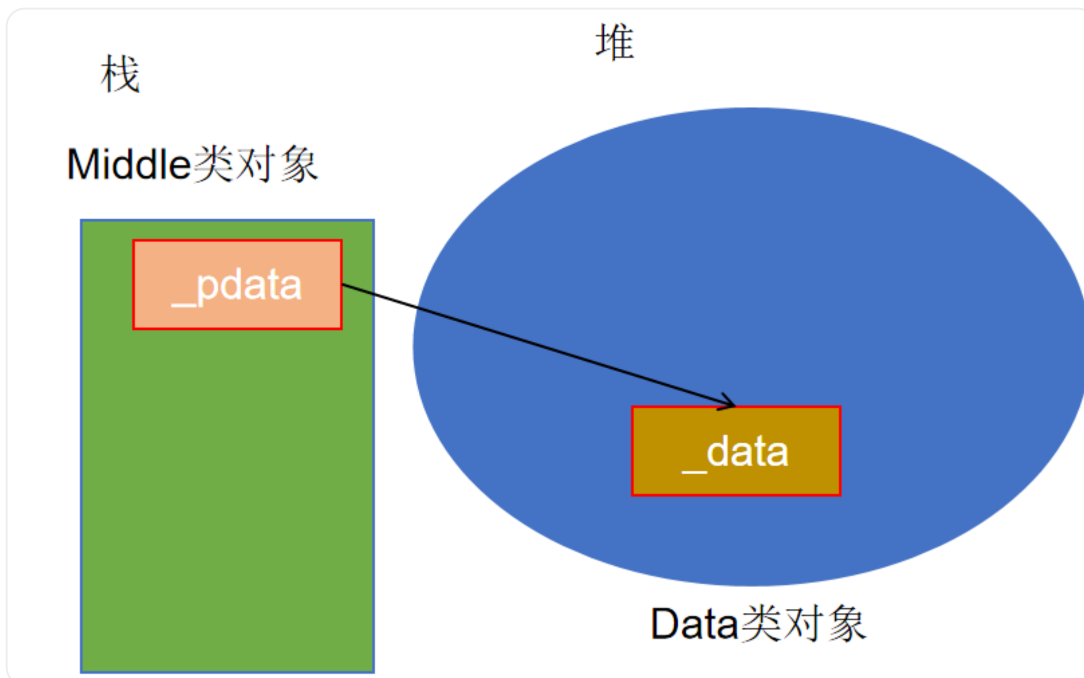
```cpp
class Data{
public:
    Data()
    { cout << "Data()" << endl; }

    Data(int x)
    : _data(x)
    { cout << "Data(int)" << endl; }

    int getData() const { return _data; }

    ~Data(){ cout << "~Data()" << endl; }
private:
    int _data = 10;
};
```

```cpp
class MiddleLayer{
public:
    MiddleLayer(Data * p)
    : _pdata(p)
    {
        cout << "MiddleLayer(Data*)" << endl;
    }
private:
    Data * _pdata;
};
```

Data*原生指针的用法如下，需要关注堆空间资源的回收

```cpp
Data * p = new Data();
p->getData();
(*p).getData();
delete p;
p = nullptr;
```

```cpp
33      Data * p1 = new Data();
34      p1->getData();
35
36      MiddleLayer ml(new Data());
37      ml->getData();
```



需求：希望实现一个这样的效果，创建MiddleLayer对象ml，让ml对象可以使用箭头运算符去调用Data类的成员函数getData

```
MiddleLayer ml(new Data);
cout << ml->getData() << endl;
```

箭头运算符无法应对MiddleLayer对象，那么可以定义箭头运算符重载函数。

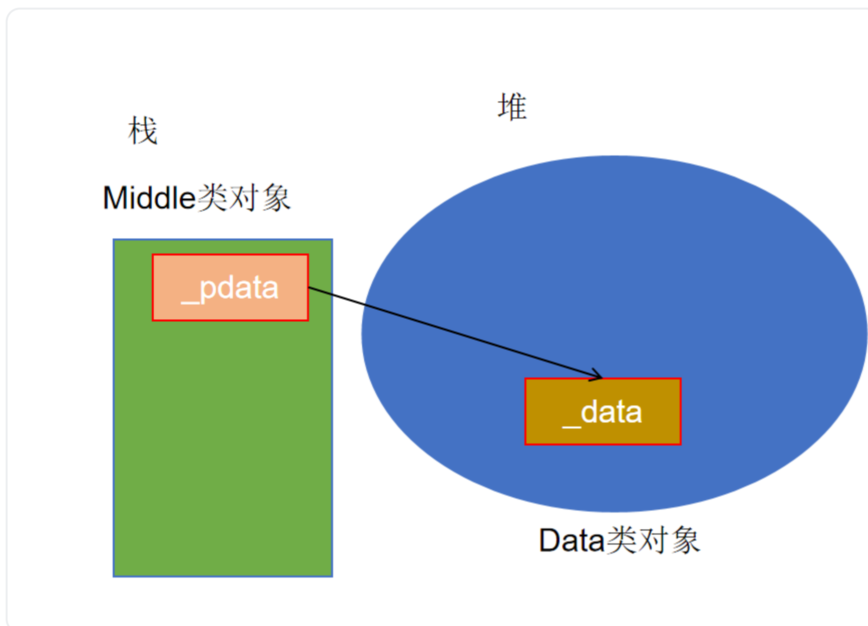
- 首先不用考虑重载形式，箭头运算符必须以成员函数形式重载；
- 然后考虑返回类型，返回值需要使用箭头运算符调用getData函数，而原生的用法只有Data* 才能这么用，所以返回值应该是一个Data* ，此时应该直接返回 _pdata；
- 同时考虑到一个问题：MiddleLayer的数据成员是一个Data*，创建MiddleLayer对象时初始化这个指针，让其指向了堆上的Data对象，那么还应该补充析构函数使MiddleLayer对象销毁时能够回收这片堆上的资源。

```
Data* operator->(){
    return _pdata;
}
```

思考，解引用运算符应该如何重载能够实现同样的效果呢？直接使用MiddleLayer对象模仿Data*指针去访问getData函数

```cpp
class MiddleLayer{
public:
    MiddleLayer(Data * p)
    : _pdata(p)
    { cout << "MiddleLayer(Data*)" << endl; }

    Data * operator->(){
        return _pdata;
    }

    Data & operator*(){
        return *_pdata;
    }

    ~MiddleLayer(){
        if(_pdata){
            delete _pdata;
            _pdata = nullptr;
        }
        cout << "~MiddleLayer()" << endl;
    }
private:
    Data * _pdata;
};
```

```cpp
void test0(){
    Data * p1 = new Data();
    p1->getData();
    (*p1).getData();
    delete p1;
    p1 = nullptr;

    MiddleLayer ml(new Data());
    cout << ml->getData() << endl;
    cout << (ml.operator->())->getData() << endl;

    cout << (*ml).getData() << endl;
    cout << (ml.operator*()).getData() << endl;
}
```



当我们完成了以上的需求后，还有一件“神奇”的事情，使用的语句中有new没有delete，但是检查发现并没有内存泄漏

原因：ml本身是一个局部对象，因为重载了箭头运算符和解引用运算符，所以看起来像个指针，也可以像指针一样进行使用，但是这个对象在栈帧结束时会自动销毁，自动调用析构函数回收了它的数据成员所申请的堆空间

实际上，这就是智能指针的雏形：其思想就是通过对象的生命周期来管理资源。

三层结构下的使用（难点）

- **拓展思考：那么如果结构再加一层，引入一个ThirdLayer类**

```
class ThirdLayer{
public:
    ThirdLayer(MiddleLayer * pml)
    : _ml(pml)
    { cout <<"ThirdLayer(MiddleLayer*)" << endl; }

    ~ThirdLayer(){
        cout << "~ThirdLayer()" << endl;
        if(_ml){
            delete _ml;
            _ml = nullptr;
        }
    }
private:
    MiddleLayer * _ml;
};
```

希望实现如下使用方式，思考一下应该如何对ThirdLayer进行对应的运算符重载

```
ThirdLayer tl(new MiddleLayer(new Data));
cout << tl->getData() << endl;
cout << (*(*tl)).getData() << endl;
```

```
/* Data * operator->(){ */
/*     return (*_ml)._pdata; */
/* } */

MiddleLayer & operator->(){
    return *_ml;
}

//两步解引用的方案二
/* MiddleLayer & operator*(){ */
/*     return *_ml; */
/* } */

//两步解引用的方案一
/* Data * operator*(){ */
/*     return (*_ml)._pdata; */
/* } */

//一步解引用的方案
Data & operator*(){
    return *((*_ml)._pdata);
}
```

```
void test1(){
    ThirdLayer tl(new MiddleLayer(new Data()));
    cout << tl->getData() << endl;
    //第一次调用ThirdLayer的箭头运算符重载函数
    //返回的是一个MiddleLayer对象
    cout << (tl.operator->())->getData() << endl;
    //因为之前已经在MiddleLayer中重载了箭头运算符
    //所以MiddleLayer对象可以调用本类的->运算符重载函数
    //返回的是一个Data*，就可以直接使用箭头运算符了
    ((tl.operator->()).operator->())->getData();

    cout << endl;
    //希望经过两步解引用访问getData
    //方案一：内层的*tl返回一个Data*
    //方案二：内存够的*tl返回一个MiddleLayer对象
    //(MiddleLayer已经进行过重载)
    //cout << (*(*tl)).getData() << endl;

    //*tl的返回值必须是Data对象才可以
    cout << (*tl).getData() << endl;
}
```

- 拓展思考：如果解引用的使用也希望和箭头运算符一样，一步到位

```
ThirdLayer tl(new MiddleLayer(new Data));
cout << (*tl).getData() << endl;
```

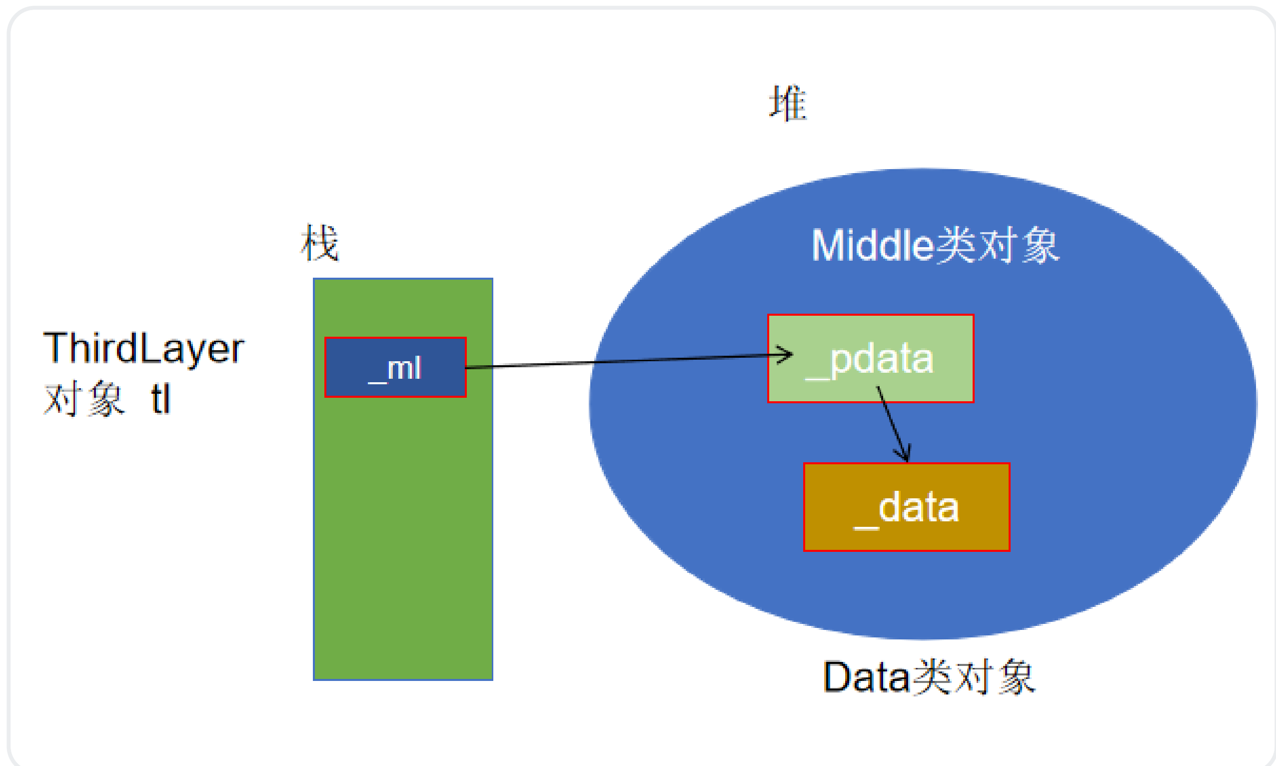
内存分析

三层的结构比较复杂，我们可以通过内存图的方式进行分析。

ThirdLayer对象的创建

```
ThirdLayer tl(new MiddleLayer(new Data));
```

实际上的内存结构如图



创建和销毁的过程：

创建tl对象时，调用ThirdLayer的构造函数，调用的过程中调用MiddleLayer的构造函数，在这个过程调用Data的构造。

Data构造完才能完成MiddleLayer的指针数据成员初始化，MiddleLayer创建完毕，才能完成ThirdLayer的指针数据成员初始化。

tl销毁时，马上调用ThirdLayer的析构，执行delete _ml时，第一步调用MiddleLayer的析构，在这个过程中，会delete _pdata，会调用Data的析构函数。

可调用实体

讲到调用这个词，我们首先能够想到**普通函数**和**函数指针**，在学习了类与对象的基础知识后，还增加了**成员函数**，那么它们都被称为**可调用实体**。事实上，根据其他的一些不同的场景需求，C++还提供了一些可调用实体，它们都是通过运算符重载来实现的。

普通函数执行时，有一个特点就是无记忆性。一个普通函数执行完毕，它所在的函数栈空间就会被销毁，所以普通函数执行时的状态信息，是无法保存下来的，这就让它无法应用在那些需要对每次的执行状态信息进行维护的场景。大家知道，我们学习了类与对象以后，有了对象的存在，对象执行某些操作之后，只要对象没有销毁，其状态就是可以保留下来的。

函数对象

想让对象像一个函数一样被调用

```
class FunctionObject{
    //...
};

void test0(){
    FunctionObject fo;
    fo();  //让对象像一个函数一样被调用
}
```

```
FunctionObject fo; //无参构造
//声明了一个返回值为FunctionObject对象的函数，函数名是fo2
FunctionObject fo2();

//无参构造创建堆上对象
FunctionObject * p = new FunctionObject();

//fo();  //让对象像一个函数一样被调用
//fo.operator()();
```

上面的代码看起来很奇怪，如果我们从运算符的视角出发，就是函数调用运算符()要处理FunctionObject对象，只需要实现一个**函数调用运算符重载**函数即可。

函数调用运算符必须以成员函数的形式进行重载

```cpp
class FunctionObject{
    void operator()(){
        cout << "void operator()()" << endl;
    }
};

void test0(){
    FunctionObject fo;
    fo();  //ok
}
```

在定义 "()" 运算符的语句中，第一对小括号总是空的，因为它代表着我们定义的运算符名称，第二对小括号就是函数参数列表了，它与普通函数的参数列表完全相同。对于其他能够重载的运算符而言，操作数个数都是固定的，**但函数调用运算符不同，它的参数是根据需要来确定的， 并不固定。**

重载了函数调用运算符的类的对象称为函数对象，由于参数列表可以随意扩展 ， 所以可以有很多重载形式（对应了普通函数的多种重载形式）

```cpp
class FunctionObject{
public:
    //第一个括号表示运算符（函数调用运算符）
    //第二个括号表示参数列表（无参形式）
    int operator()(){
        cout << "int operator()()" << endl;
        return 1;
    }

    //可以定义多种函数调用运算符重载函数
    void operator()(int x){
        cout << "void operator(int)" << endl;
        cout << x << endl;
    }
```

```cpp
FunctionObject fo; //无参构造
```

```cpp
    cout << fo() << endl;  //让对象像一个函数一样被调用
    //第一个括号应该与operator连在一起，作为函数名
    //fo这个对象调用了operator()这个函数
    //第二个括号是函数调用的形式
    cout << fo.operator()() << endl;
```

```cpp
class FunctionObject{
public:
    void operator()(){
        cout << "FunctionObject operator()()" << endl;
        ++ _count;
```

```cpp
    }

    int operator()(int x, int y){
        cout <<"operator()(int,int)" << endl;
        ++ _count;
        return x + y;
    }

    int _count = 0;//携带状态
};

void test0(){
    FunctionObject fo;

    cout << fo() << endl;
    cout << fo.operator()() << endl;//本质

    cout << fo(5,6) << endl;
    cout << fo.operator()(5,6) << endl;//本质

    cout << "fo._count:" << fo._count << endl;//记录这个函数对象被调用的次数
}


```

函数对象相比普通函数的优点：可以携带状态（函数对象可以封装自己的数据成员、成员函数，具有更好的面向对象的特性）

如上，可以记录函数对象被调用的次数，而普通函数只能通过全局变量做到（全局变量不够安全）。

```cpp
int cnt = 0;
void print(){
    //static int cnt = 0;
    cout << "hello" << endl;
    ++cnt;
}
```

函数指针

既然对象可以像一个函数一样去调用，那函数可不可以像一个对象一样去组织?

如果可以，那函数类型由什么决定呢，也就是说，如果把函数看作对象，如何从这些"对象"抽象出类来?

在C的阶段就学习过函数指针，定义函数指针时要明确使用这个指针指向一个什么类型的函数（返回类型、参数类型都要确定）

```cpp
void print(int x){
    cout << "print:" << x << endl;
}

void display(int x){
    cout << "display:" << x << endl;
}

int main(void){
    //省略形式
    void (*p)(int) = print;
    p(4);
    p = display;
    p(9);

    //完整形式
    void (*p2)(int) = &print;
    (*p2)(4);
    p2 = &display;
    (*p2)(9);
}
```

定义函数指针p后，可以指向print函数，也可以再指向display函数，并通过函数指针调用函数（两种方式——完整/省略）；

——那么其实可以抽象出一个函数指针类，这个类的对象就是这个特定类型的函数指针

p和p2可以抽象出一个函数指针类型**void(*)(int)** —— **逻辑类型，不能在代码中直接以这种形式写出**

```cpp
//将此类的函数指针的类型定名为Func
typedef void (*Func)(int);

void test1(){
    Func p = print;
    p(6);

    p = display;
    p(7);
}
```

以前我们使用typedef可以定义类型别名，这段程序中函数指针p、p2的类型是void（*）（int），但是C++中是没有这个类的（我们可以这样理解，但是代码不能这么写）

可以使用typedef定义这样的一个新类型

可以理解为是给void (*) (int) 取类型别名为Function

```
typedef void(*Function)(int);
```

Function类的对象可以这样使用，这个类的对象都是特定类型的函数指针，只能指向一种函数（这种函数的类型在定义函数指针类时就决定了）

```
    Function f;
    f = print;
    f(19);
    f = display;
    f(27);
```

成员函数指针

函数指针的用法熟悉后，顺势思考一个问题：成员函数能否也使用这种形式？如果可以，应该怎样定义一个成员函数指针

比如有这样一个类FFF，包含两个成员函数

```
class FFF
{
public:
    void print(int x){
        cout << "FFF::print:" << x << endl;
    }

    void display(int x){
        cout << "FFF::display:" << x << endl;
    }

    void test()
    {}
};
```

定义一个函数指针要明确指针指向的函数的返回类型、参数类型，那么**定义一个成员函数指针还需要确定的是这个成员函数是哪个类的成员函数（类的作用域）**

与普通函数指针不一样的是，**成员函数指针的定义和使用都需要使用完整写法**，不能使用省略写法，定义时要完整写出指针声明，使用时要完整写出解引用（解出成员函数后接受参数进行调用）。

另外，成员函数需要通过对象来调用，**成员函数指针也需要通过对象来调用**。

```
void (FFF::*p)(int) = &FFF::print;
FFF ff;
(ff.*p)(4);
```

——类比来写，也可以使用typedef来定义这种成员函数指针类,使用这个成员函数指针类的对象来调用FFF类的成员函数print

这里有一个要求，成员函数指针指向的成员函数需要是FFF类的公有函数

```
typedef void (FFF::*MemberFunction)(int); //定义成员函数类型MemberFunction

MemberFunction mf = &FFF::print; //定义成员函数指针
FFF fff;
(fff.*mf)(15);//通过对象调用成员函数指针
```

此时就出现了一个新的运算符 ".*" —— **成员指针访问运算符的第一种形式。**

FFF类对象还可以是一个堆上的对象

```
FFF * fp = new FFF();

(fp->*mf)(65);//通过指针调用成员函数指针
```

又引出了新的运算符 "->*" —— **成员指针访问运算符的第二种形式。**

```
//将此类的成员函数指针的类型定名为MemberFunc
typedef void (FFF::*MemberFunc)(int);
```

```
void test3(){
    //创建成员函数指针的时候就已经确定了
    //返回类型、参数、是哪个类的成员函数
    MemberFunc p = &FFF::print;
    FFF fff;
    //成员指针访问运算符的第一种形式
    //指针指的是p，这是一个成员函数指针
    (fff.*p)(10);

    p = &FFF::display;
    /* p = &DDD::print; */

    FFF * pff = new FFF();
    //成员指针访问运算符的第二种形式
    (pff->*p)(11);
```

成员函数指针的意义：

1. 回调函数：**将成员函数指针作为参数传递给其他函数**，使其他函数能够在特定条件下调用该成员函数；
2. 事件处理：将成员函数指针存储事件处理程序中，以便在特定事件发生时调用相应的成员函数；
3. 多态性：通过将成员函数指针存储在基类指针中，可以实现多态性，在运行时能够去调用相应的成员函数。

空指针的使用

接着上面的例子，我们来看一段比较奇怪的代码

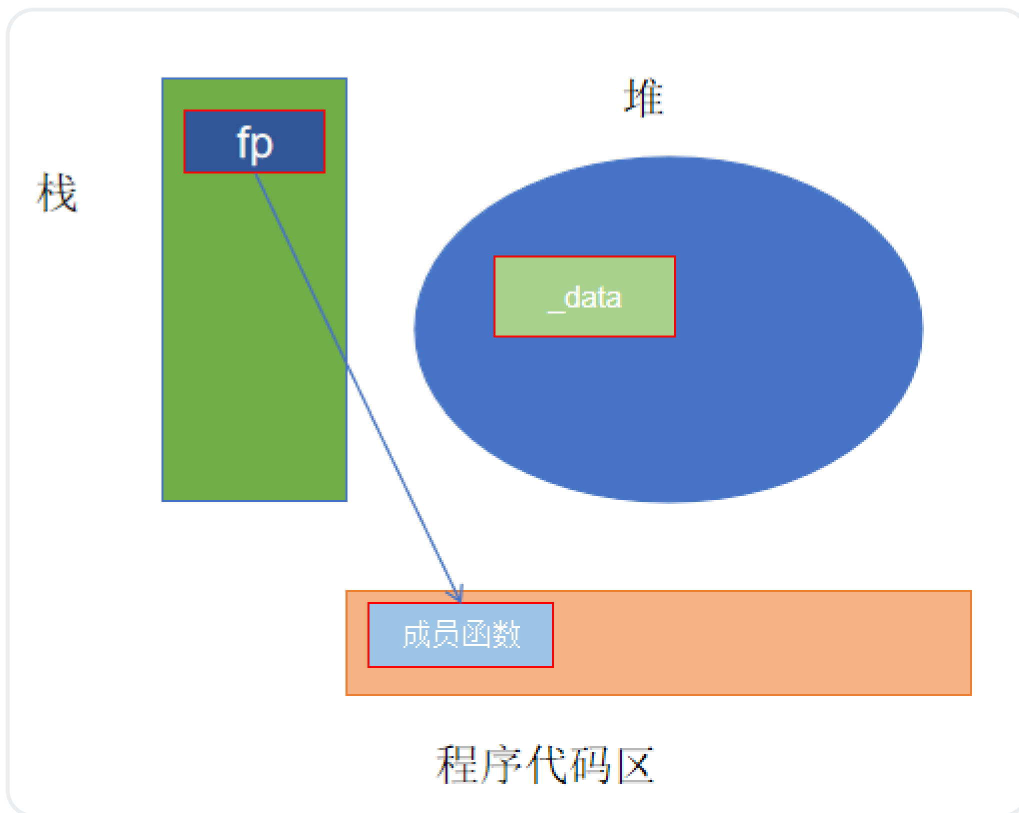
```
fp = nullptr;
(fp->*mf)(34);
```

发现竟然是可以通过的并输出了正常的结果。难道空指针去调用成员函数指针没有问题吗?

事实上，空指针去调用成员函数也好、成员函数指针也好，只要不涉及到访问该类数据成员，都是可以的。

```
class Bar{
public:
    void test0(){ cout << "Bar::test0()" << endl; }
    void test1(int x){ cout << "Bar::test1(): " << x << endl; }
    void test2(){ cout << "Bar::test2(): " << _data << endl; }

    int _data = 10;
};

void test0(){
    Bar * fp = nullptr;
    fp->test0();
    fp->test1(3);
    fp->test2(); //error
}
```

结合内存图来分析



空指针没有指向有效的对象。对于不涉及数据成员的成员函数，不需要实际的对象上下文，因此就算是空指针也可以调用成功。对于涉及数据成员的成员函数，空指针无法提供有效的对象上下文，因此导致错误。

总结：

C++中普通函数、函数指针、成员函数、成员函数指针、函数对象，可以将它们概括为可调用实体。

类型转换函数

以前我们认识了普通变量的类型转换，比如说 int 型转换为 long 型， double 型转换为 int 型，接下来我们要讨论下类对象与其他类型的转换。转换的方向有：

由其他类型向自定义类型转换

由自定义类型向其他类型转换

- **由其他类型向自定义类型转换**

由其他类型向定义类型转换是由构造函数来实现的，只有当类中定义了合适的构造函数时，转换才能通过。这种转换，一般称为**隐式转换**。

之前我们见识了隐式转换，当时的例子中能够进行隐式转换的前提是Point类中有相应的构造函数，编译器会看用一个int型数据能否创建出一个Point对象，如果可以，就创建出一个临时对象，并将它的值复制给pt

```
Point pt = 1;
//等价于Point pt = Point(1);
```

这种隐式转换是比较奇怪的，一般情况下，不希望这种转换成立，所以可以在相应的构造函数之前加上explicit关键字，禁止这种隐式转换。

而有些隐式转换使用起来很自然，比如：

```
string s1 = "hello,world";
```

这行语句其实也是隐式转换，利用C风格字符串构造一个临时的string对象，再调用string的拷贝构造函数创建s1

- **由自定义类型向其他类型转换——类型转换函数**

类型转换函数的形式是固定的：**operator 目标类型(){   }**

它有着如下的特征：

1.必须是成员函数

2.没有返回类型

3.没有参数

4.在函数执行体中必须要返回目标类型的变量

(1)自定义类型向内置类型转换

在Point类中定义这样的类型转换函数

```
class Point{
public:
   //...
    operator int(){
        cout << "operator int()" << endl;
        return _ix + _iy;
    }
    //...
};
```

使用时就可以写出这样的语句（与隐式转换的方向相反）

```
Point pt(1,2);
int a = 10;
//将Point类型对象转换成int型数据
a = pt;
cout << a << endl;
```

```
operator int(){
    cout << "operator int()" << endl;
    return _ix + _iy;
}
```

```
//隐式转换，从内置类型转换成自定义类型
/* Point pt = 1; */

Point pt2(1,2);
//利用类型转换函数，将自定义类型转换成了内置类型
int a = pt2;
int b = pt2.operator int();//本质
cout << a << endl;
cout << b << endl;
```

(2) 自定义类型向自定义类型转换

自定义类型可以向内置类型转换，还可以向自定义类型转换，但要**注意将类型转换函数设为谁的成员函数**

```
Point pt(1,2);
Complex cx(3,4);
cx = pt;
cx.print();
```

如上，想要让Point对象转换成Complex对象，并对cx赋值，应该在Point类中添加目标类型的类型转换函数

```
class Point
{
    //...
    operator Complex(){
        cout << "operator Complex()" << endl;
        return Complex(_ix,_iy);
    }
};
```

```
operator Complex(){
    cout << "operator Complex()" << endl;
    return Complex(_ix,_iy);
}
```

思考，可否用隐式转换的思路（即调用特定形式的构造函数），实现这种转换?

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

```
//隐式转换, cx = Complex(pt);
cx = pt;
cx.print();
```

附录：C++运算符优先级排序与结合性

1	::	作用域解析	从左到右
2	a++ a-- *type*() *type*{} a() a[] . ->	后缀自增与自减 函数风格转型 函数调用 下标 成员访问	
3	++a --a +a -a ! ~ (*type*) *a &a sizeof co_await new new[] delete delete[]	前缀自增与自减 一元加与减 逻辑非和逐位非 C 风格转型 间接（解引用） 取址 取大小[注 1] await 表达式 (C++20) 动态内存分配 动态内存分配	从右到左
4	.* ->*	成员指针	从左到右
5	a*b a/b a%b	乘法、除法与余数	
6	a+b a-b	加法与减法	
7	<< >>	逐位左移与右移	
8	<=>	三路比较运算符(C++20 起)	
9	< <= > >=	分别为 < 与 ≤ 的关系运算符 分别为 > 与 ≥ 的关系运算符	
10	== !=	分别为 = 与 ≠ 的关系运算符	
11	a&b	逐位与	
12	^	逐位异或（互斥或）	
13	|	逐位或（可兼或）	
14	&&	逻辑与	
15	||	逻辑或	
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	三元条件[注 2] throw 运算符 yield 表达式 (C++20) 直接赋值（C++ 类默认提供） 以和及差复合赋值 以积、商及余数复合赋值 以逐位左移及右移复合赋值 以逐位与、异或及或复合赋值	从右到左
17	,	逗号	从左到右

嵌套类

首先介绍两个概念：

- 类作用域（Class Scope）

类作用域是指在类定义内部的范围。在这个作用域内定义的成员（包括变量、函数、类型别名等）可以被该类的所有成员函数访问。类作用域开始于类定义的左花括号，结束于类定义的右花括号。在类作用域内，成员可以相互访问，无论它们在类定义中的声明顺序如何。

- 类名作用域（Class Name Scope）

类名作用域指的是可以通过类名访问的作用域。这主要用于访问类的静态成员、嵌套类型。类名必须用于访问静态成员或嵌套类型，除非在类的成员函数内部，因为它们不依赖于类的任何特定对象。以静态成员为例：

```
class MyClass
{
public:
    void func(){
        _a = 100;//类的成员函数内访问_a
    }
    static int _a;
};
int MyClass::_a = 0;

void test0(){
    MyClass::_a = 200;//类外部访问_a
}
```

在函数和其他类定义的外部定义的类称为**全局类**，绝大多数的 C++ 类都是全局类。我们在前面定义的所有类都在全局作用域中，全局类具有全局作用域。

与之对应的，一个类A还可以定义在另一类B的定义中，这就是**嵌套类**结构。A类被称为B类的**内部类**，B类被称为A类的**外部类**。

以Point类和Line类为例

```
class Line
{
public:
    class Point{
    public:
        Point(int x,int y)
        : _ix(x)
        , _iy(y)
```

```
        {}
    private:
        int _ix;
        int _iy;
    };
public:
    Line(int x1, int y1, int x2, int y2)
    : _pt1(x1,y1)
    , _pt2(x2,y2)
    {}
private:
    Point _pt1;
    Point _pt2;
};
```

Point类是定义在Line类中的内部类，无法直接创建Point对象，需要在Line类名作用域中才能创建

```
Point pt(1,2);//error
Line::Point pt2(3,4);//ok
```

A类是B类的内部类，并不代表A类的数据成员会占据B类对象的内存空间，**在存储关系上并不是嵌套的结构**。

只有当B类有A类类型的对象成员时，B类对象的内存布局中才会包含A类对象（成员子对象）。

```
class Line
{
public:
    class Point{
    public:
        Point(int x,int y)
        : _ix(x)
        , _iy(y)
        {}
    private:
        int _ix;
        int _iy;
    };
public:
    Line(int x1, int y1, int x2, int y2)
    : _pt1(x1,y1)
    , _pt2(x2,y2)
    {}
private:
    Point _pt1;
    Point _pt2;
    double length = 10;
};
```

（1）如果Line类中没有Point类的对象成员，sizeof(Line) = 8;

（2）如果Line类中有两个Point类的对象成员，sizeof(Line) = 24;

思考，如果想要使用输出流运算符输出上述的嵌套类对象，应该怎么实现？

嵌套类结构的访问权限

外部类对内部类的成员进行访问

内部类对外部类的成员进行访问

访问成员方式	不依赖对象直接访问	类名作用域访问	通过对象直接访问
外部类对内部类	无	内部类的静态成员 + 声明友元才ok	内部类的私有成员需要声明友元
内部类对外部类	外部类的静态成员	外部类的静态成员	即使是私有成员也ok

内部类相当于是定义在外部类中的外部类的友元类

pimpl模式（了解）

实际项目的需求：希望Line的实现全部隐藏，在源文件中实现，再将其打包成库文件，交给第三方使用。

（1）头文件只给出接口：

```
//Line.hpp
class Line{
public:
    Line(int x1, int y1, int x2, int y2);
    ~Line();
    void printLine() const;//打印Line对象的信息
private:
    class LineImpl;//类的前向声明
    LineImpl * _pimpl;
};
```

（2）在实现文件中进行具体实现，使用嵌套类的结构（LineImpl是Line的内部类，Point是LineImpl的内部类），Line类对外公布的接口都是使用LineImpl进行具体实现的

在测试文件中创建Line对象（最外层），使用Line对外提供的接口，但是不知道具体的实现

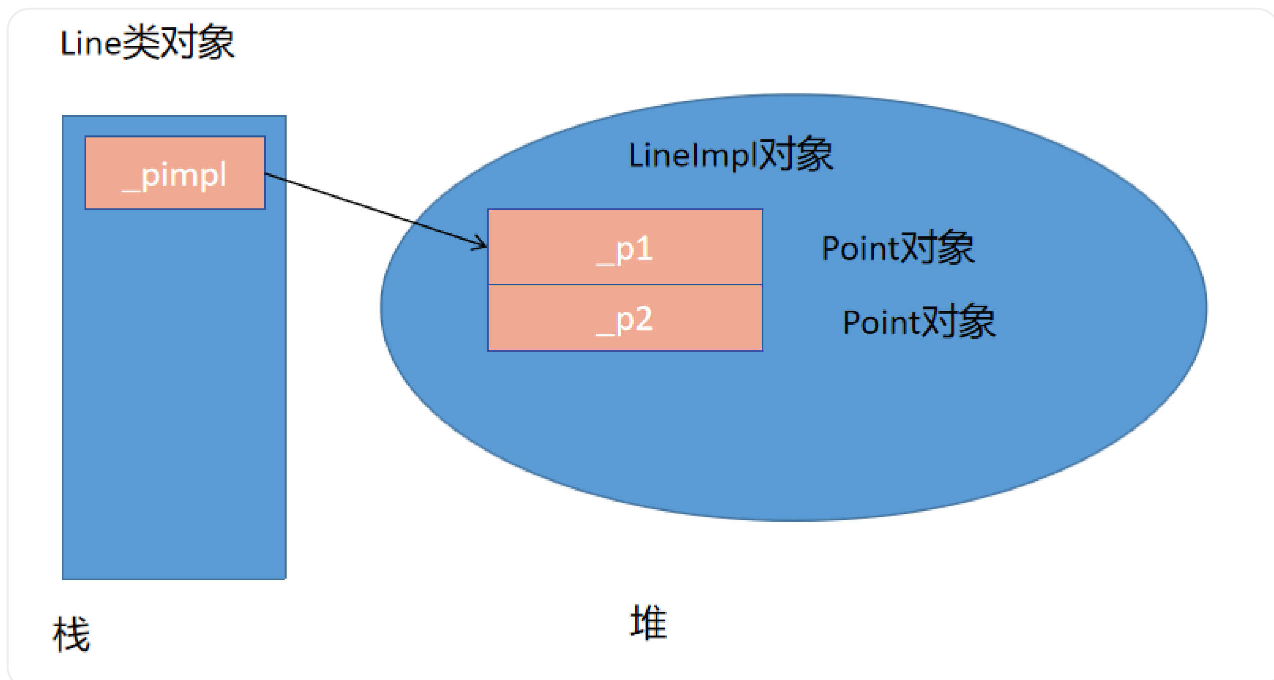
```
//LineImpl.cc
class Line::LineImpl
{
    class Point{
    public:
        Point(int x,int y)
            : _ix(x)
              , _iy(y)
        {}
        //...
    private:
        int _ix;
        int _iy;
    };
    //...
};

//Line.cc
```

```
19  void test0(){
20      Line line(10,20,30,40);
21      line.printLine();
22  }
```

（3）打包库文件，将库文件和头文件交给第三方

```
sudo apt install build-essential
g++ -c LineImpl.cc
ar rcs libLine.a LineImpl.o

生成libLine.a库文件
编译：g++ Line.cc(测试文件) -L(加上库文件地址) -lLine(就是库文件名中的lib缩写为l，不带后缀)
此时的编译指令为 g++ Line.cc -L. -lLine
```

内存结构



pimpl模式是一种减少代码依赖和编译时间的C++编程技巧，其基本思想是将一个外部可见类的实现细节（一般是通过私有的非虚成员）放在一个单独的实现类中，在可见类中通过一个私有指针来间接访问该类型。

好处：

1. 实现信息隐藏；
2. 只要头文件中的接口不变，实现文件可以随意修改，修改完毕只需要将新生成的库文件交给第三方即可；
3. 可以实现库的平滑升级。

单例对象自动释放（重点*）

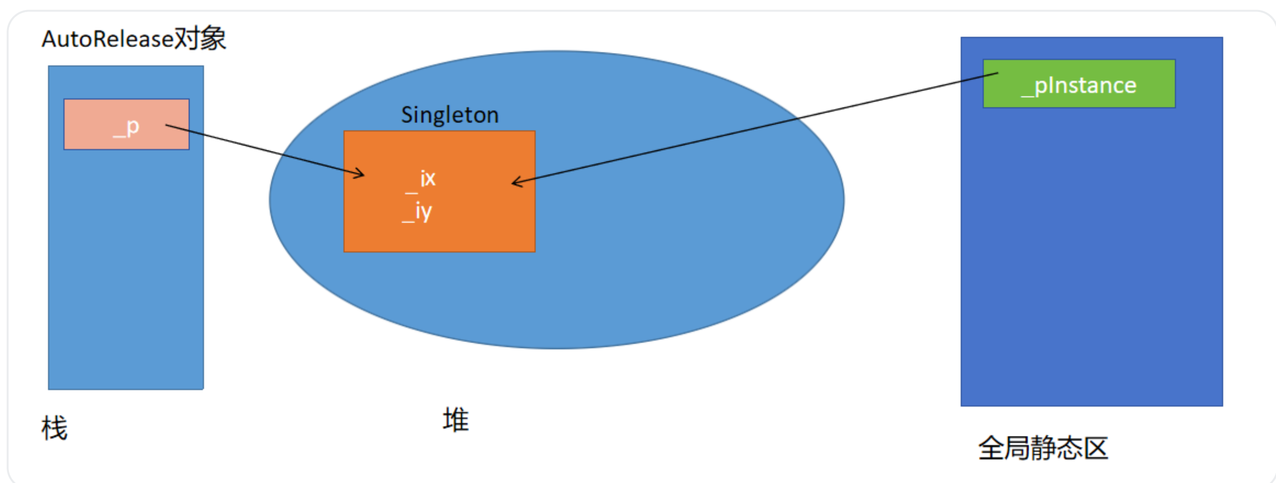
在类与对象的章节，我们学习了单例模式。单例对象由静态指针_pInstance保存，最终通过手动调用destroy函数进行释放。

现实工作中，单例对象是需要进行自动释放。程序在执行的过程中 ，需要判断有哪些地方发生了内存泄漏 ，此时需要工具valgrind的使用来确定。假设单例对象没有进行自动释放 ，那么valgrind工具会认为单例对象是内存泄漏。程序员接下来还得再次去确认到底是不是内存泄漏 ，增加了程序员的额外的工作。

那么如何实现单例对象的自动释放呢?

—— 看到自动就应该想到当对象被销毁时，析构函数会被自动调用。

方式一：利用另一个对象的生命周期管理资源



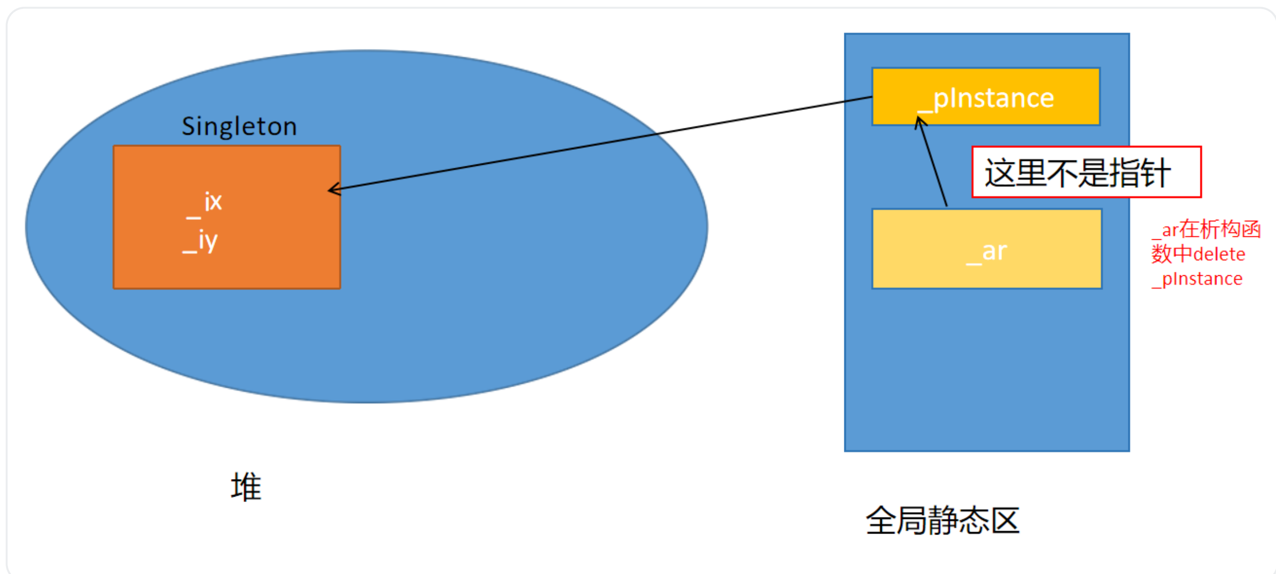
利用对象的生命周期管理资源析构函数（在析构函数中会执行delete _p），当对象被销毁时会自动调用。

要注意：如果还手动调用了Singleton类的destroy函数，会导致double free问题，所以可以删掉destroy函数，将回收堆上的单例对象的工作完全交给AutoRelease对象

```
class AutoRelease{
public:
    AutoRelease(Singleton * p)
    : _p(p)
    { cout << "AutoRelease(Singleton*)" << endl; }

    ~AutoRelease(){
        cout << "~AutoRelease()" << endl;
        if(_p){
            delete _p;
            _p = nullptr;
        }
```

```
13         }
14     private:
15         Singleton * _p;
16     };
17
18     void test0(){
19         AutoRelease ar(Singleton::getInstance());
20         Singleton::getInstance()->print();
21     }
```

方式二：嵌套类 + 静态对象



AutoRelease类对象_ar是Singleton类的对象成员，创建Singleton对象，就会自动创建一个AutoRelease对象（静态区），它的成员函数可以直接访问 _pInstance

```
1     class Singleton
2     {
3         class AutoRelease{
4         public:
5             AutoRelease()
6             {}
7             ~AutoRelease(){
8               //...
9             }
10        };
11        //...
12    private:
```

```cpp
    //...
    int _ix;
    int _iy;
    static Singleton * _pInstance;
    static AutoRelease _ar;
};
Singleton* Singleton::_pInstance = nullptr;
//使用AutoReleas类的无参构造对_ar进行初始化
Singleton::AutoRelease Singleton::_ar;


void test1(){
    Singleton::getInstance()->print();
    Singleton::getInstance()->init(10,80);
    Singleton::getInstance()->print();
}
```

```cpp
class Singleton {
    class AutoRelease{
    public:
        AutoRelease()
        {
            cout << "AutoRelease()" << endl;
        }

        ~AutoRelease(){
            cout << "~AutoRelease()" << endl;
            if(_pInstance){
                delete _pInstance;
                _pInstance = nullptr;
            }
        }
    };
```

```cpp
private:
    int _ix;
    int _iy;
    static Singleton * _pInstance;
    static AutoRelease _ar;
};
Singleton * Singleton::_pInstance = nullptr;
Singleton::AutoRelease Singleton::_ar;
```

程序结束时会自动销毁全局静态区上的_ar，调用AutoRelease的析构函数，在这个析构函数执行delete _pInstance的语句，这样又会调用Singleton的析构函数，再调用operator delete，回收掉堆上的单例对象。

我们利用嵌套类实现了一个比较完美的方案，不用担心手动调用了destroy函数。

方式三：atexit + destroy

很多时候我们需要在程序退出的时候做一些诸如释放资源的操作，但程序退出的方式有很多种，比如main()函数运行结束、在程序的某个地方用exit()结束程序、用户通过Ctrl+C操作来终止程序等等，因此需要有一种与程序退出方式无关的方法来进行程序退出时的必要处理。

方法就是用atexit函数来注册程序正常终止时要被调用的函数（C/C++通用）。

如果注册了多个函数，先注册的后执行。

```
class Singleton
{
public:
    static Singleton * getInstance(){
        if(_pInstance == nullptr){
            atexit(destroy);
            _pInstance = new Singleton(1,2);
        }
        return _pInstance;
    }
    //...
};
```

atexit注册了destroy函数，相当于有了一次必然会进行的destroy（程序结束时），即使手动调用了destroy，因为安全回收的机制，也不会有问题。

但是还遗留了一个问题，就是以上几种方式都无法解决**多线程安全**问题。以方式三为例，当多个线程同时进入if语句时，会造成单例对象被创建出多个，但是最终只有一个地址值会由_pInstance指针保存，因此造成内存泄漏。

可以使用**饿汉式解决**，但同时也可能带来内存压力（即使不用单例对象，也会被创建）

```
//对于_pInstance的初始化有两种方式

//饱汉式（懒汉式）—— 懒加载，不使用到该对象，就不会创建
Singleton* Singleton::_pInstance = nullptr;

//饿汉式 —— 最开始就创建（即使不使用这个单例对象）
Singleton* Singleton::_pInstance = getInstance();
```

方式四：atexit + pthread_once

Linux平台可以使用的方法（能够保证创建单例对象时的多线程安全）

pthread_once函数可以确保初始化代码只会执行一次，传给pthread_once函数的第一个参数比较特殊，形式固定，第二个参数需要是一个静态函数指针

```
SYNOPSIS
       #include <pthread.h>

       pthread_once_t once_control = PTHREAD_ONCE_INIT;

       int pthread_once(pthread_once_t *once_control, void (*init_routine) (void));

DESCRIPTION
       The  purpose  of  pthread_once is to ensure that a piece of initialization code is
       executed at most once. The once_control argument points  to  a  static  or  extern
       variable statically initialized to PTHREAD_ONCE_INIT.

       The first time pthread_once is called with a given once_control argument, it calls
       init_routine with no argument and changes the value of the  once_control  variable
       to record that initialization has been performed. Subsequent calls to pthread_once
       with the same once_control argument do nothing.
```

```cpp
class Singleton{
public:
    static Singleton * getInstance(){
        pthread_once(&_once,init_r);
        return _pInstance;
    }

    static void init_r(){
        _pInstance = new Singleton(1,2);
        atexit(destroy);
    }
    //...
private:
    int _ix;
    int _iy;
    static Singleton * _pInstance;
    static pthread_once_t _once;
};
Singleton* Singleton::_pInstance = nullptr;
pthread_once_t Singleton::_once = PTHREAD_ONCE_INIT;
```

注意：因为初始化（创建堆对象）的语句之后被执行一次，所以不能手动调用destroy函数，同时因为会使用atexit注册destroy函数实现资源回收，所以也不能将destroy删掉，应该将其私有，避免在类外手动调用。

std::string的底层实现*

我们都知道，std::string的一些基本功能和用法了，但它底层到底是如何实现的呢？其实在std::string的历史中，出现过几种不同的方式。

我们可以从一个简单的问题来探索，一个std::string对象占据的内存空间有多大，即sizeof(std::string)的值为多大？如果我们在不同的编译器（VC++，GCC，Clang++）上去测试，可能会发现其值并不相同；即使是GCC，不同的版本，获取的值也是不同的。

虽然历史上的实现有多种，但基本上有三种方式：

- Eager Copy(深拷贝)
- COW（Copy-On-Write 写时复制）
- SSO(Short String Optimization 短字符串优化)

  std::string的底层实现是一个高频考点，虽然目前std::string是根据SSO的思想实现的，但是我们最好能够掌握其发展过程中的不同设计思想，在回答时会是一个非常精彩的加分项。

首先，最简单的就是深拷贝。无论什么情况，都是采用拷贝字符串内容的方式解决，这也是我们之前已经实现过的方式。这种实现方式，在不需要改变字符串内容时，对字符串进行频繁复制，效率比较低下。所以需要对其实现进行优化，之后便出现了下面的COW的实现方式。

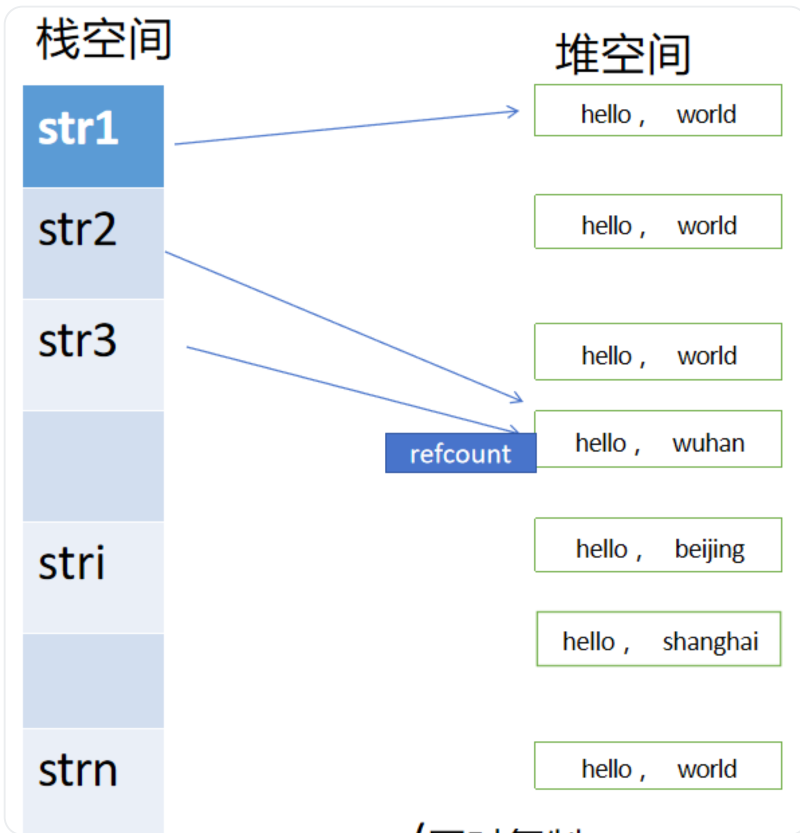
```
//如果string的实现直接用深拷贝
string str1("hello,world");
string str2 = str1;
```

如上，str2保存的字符串内容与str1完全相同，但是根据深拷贝的思想，一定要重新申请空间、复制内容，这样效率较低、开销较大。

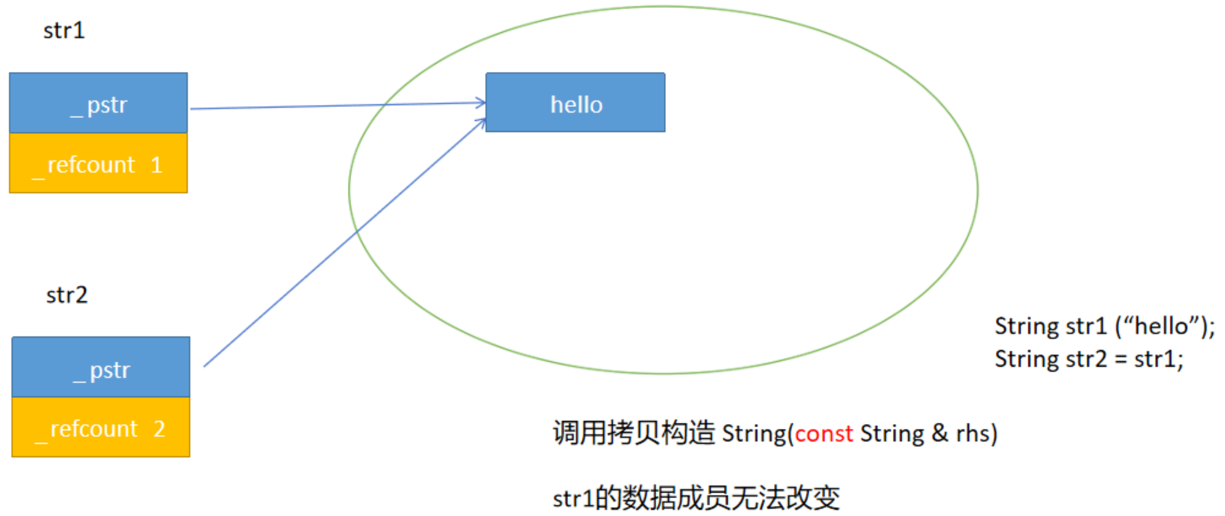
写时复制原理探究

Q1：当字符串对象进行复制控制时，可以优化为指向同一个堆空间的字符串，接下来的问题就是何时回收堆空间的字符串内容呢？

引用计数 refcount当字符串对象进行复制操作时，引用计数+1；当字符串对象被销毁时，引用计数-1；只有当引用计数减为0时，才真正回收堆空间上字符串

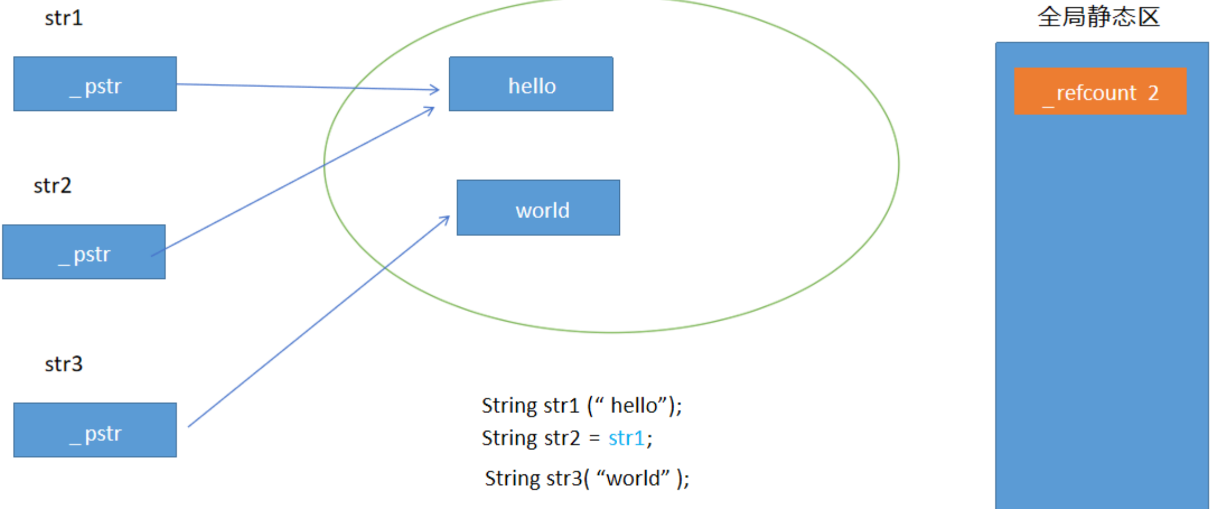


Q2：引用计数应该放到哪里？

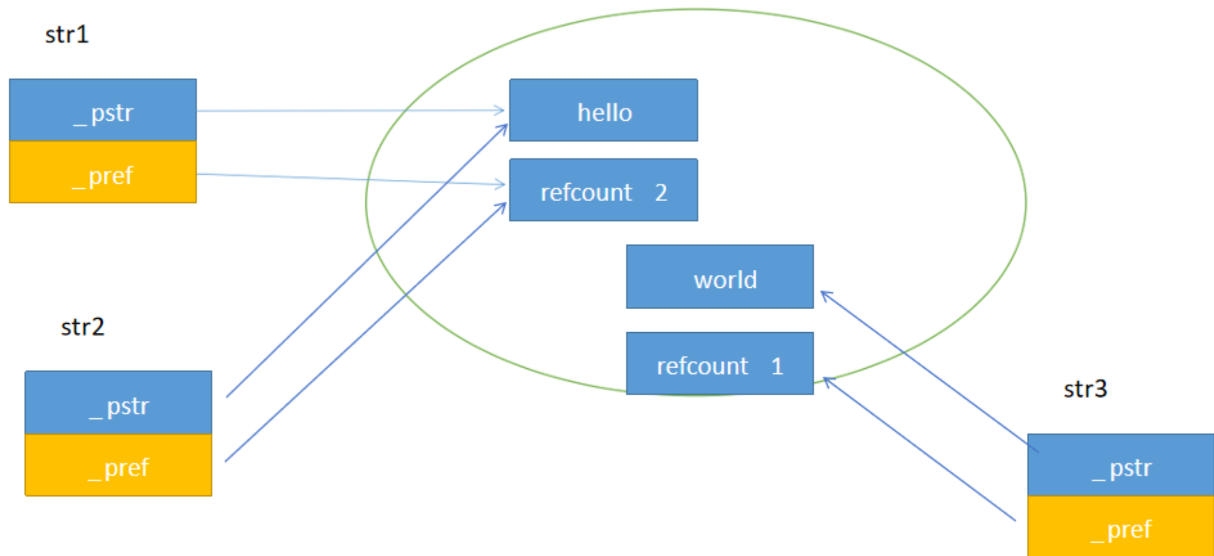


方案二：如果引用计数是一个静态数据成员

静态数据成员被这个类的所有对象共享，但是有些时机是需要多个refcount

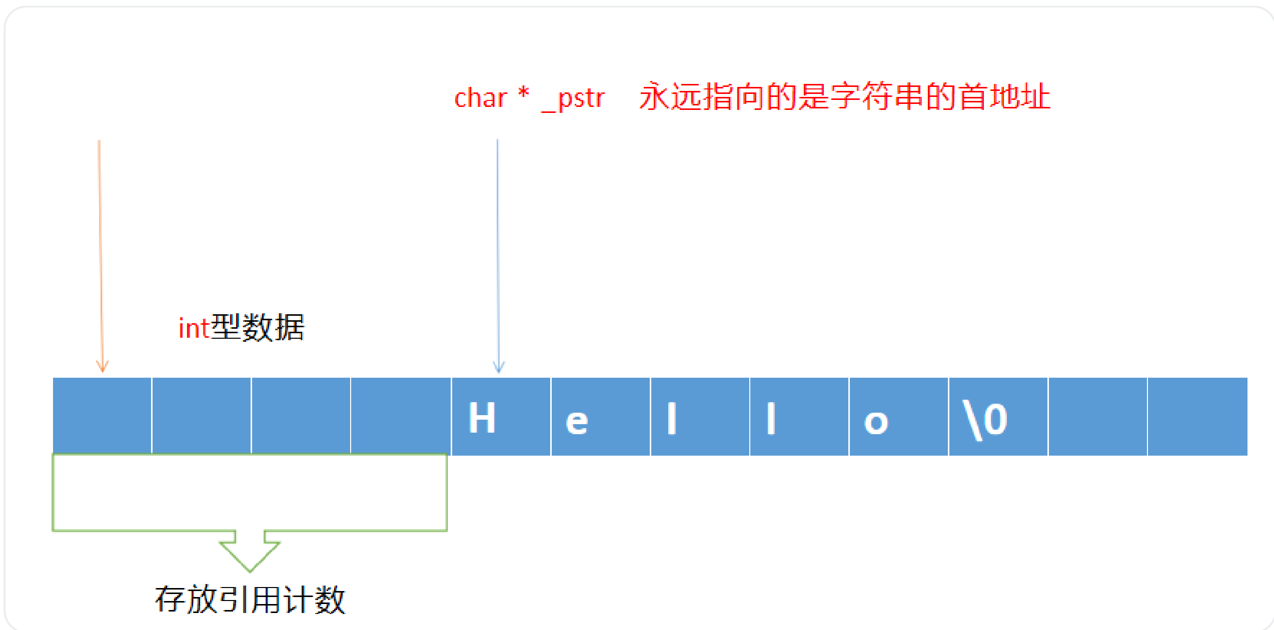


方案三：用堆空间来保存引用计数

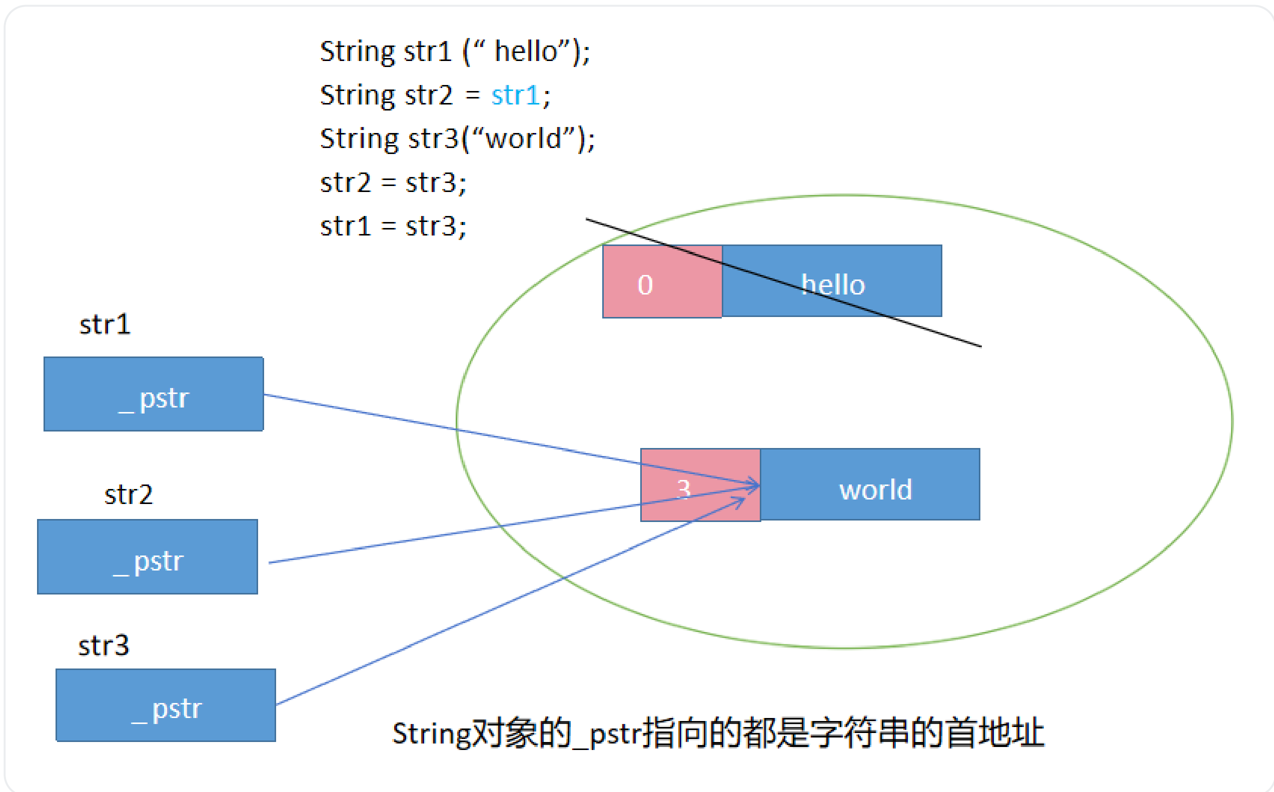


方案三可行，还可以优化一下

按常规的思路，需要使用两次new表达式（字符串、引用计数）；可以优化成只用一次new表达式，因为申请堆空间的行为一定会涉及系统调用，程序员要尽量少使用系统调用，提高程序的执行效率。



引用计数减到1，才真正回收堆空间



CowString代码初步实现

根据写时复制的思想来模拟字符串对象的实现，这是一个非常有难度的任务（源码级），理解了COW的思想后可以尝试实现一下

见CowString1.cc

在我们建立了基本的写时复制字符串类的框架后，发现了一个遗留的问题。

如果str1和str3共享一片空间存放字符串内容。如果进行读操作，那么直接进行就可以了，不用进行复制，也不用改变引用计数；如果进行写操作，那么应该让str1重新申请一片空间去进行修改，不应该改变str3的内容。

```
cout << str1[0] << endl; //读操作
str1[0] = 'H';       //写操作
cout << str3[0] << endl;//发现str3的内容也被改变了
```

我们首先会想到运算符重载的方式去解决。但是str1[0]返回值是一个char类型变量。

读操作 cout << char字符 << endl;

写操作 char字符 = char字符;

无论是输出流运算符还是赋值运算符，操作数中没有自定义类型对象，无法重载。而CowString的下标访问运算符的操作数是CowString对象和size_t类型的下标，也没办法判断取出来的内容接下来要进行读操作还是写操作。

—— 思路：创建一个CowString类的内部类，让CowString的operator[]函数返回是这个新类型的对象，然后在这个新类型中对<<和=进行重载，让这两个运算符能够处理新类型对象，从而分开了处理逻辑。

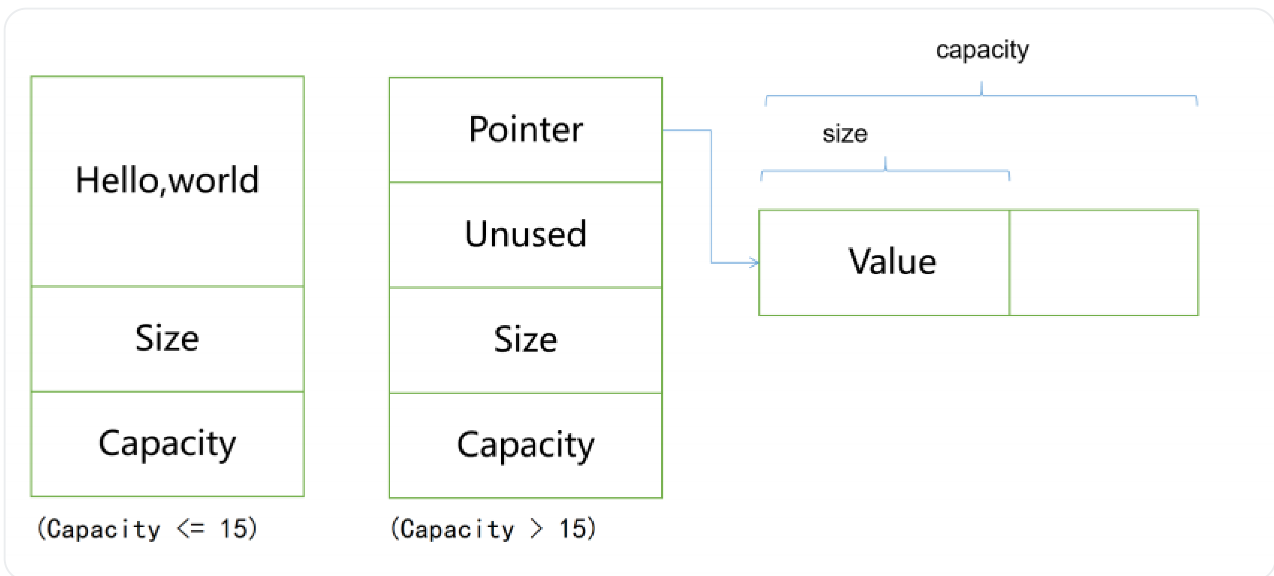
见CowString2.cc

对于读操作，还可以给CharProxy类定义类型转换函数来进行处理。

见CowString3.cc

短字符串优化（SSO）

当字符串的字符数小于等于15时， buffer直接存放整个字符串；当字符串的字符数大于15时， buffer 存放的就是一个指针，指向堆空间的区域。这样做的好处是，当字符串较小时，直接拷贝字符串，放在 string内部，不用获取堆空间，开销小。



union表示共用体，允许在同一内存空间中存储不同类型的数据。公用体的所有成员共享一块内存，但是每次只能使用一个成员。

```
class string {
    union Buffer{
        char * _pointer;
        char _local[16];
    };

    size_t _size;
    size_t _capacity;
    Buffer _buffer;
};
```

```
void test1(){
    string str1("helloworldaaaaa");
    string str2("helloworldaaaaaa");

    cout << "&str1:" << &str1 << endl;
    /* cout << &str1[0] << endl; */
    printf("%p\n",&str1[0]);
    cout << "&str2:" << &str2 << endl;
    printf("%p\n",&str2[0]);
}
```

```
&str1:0x7ffdbe2b92e0
0x7ffdbe2b92f0
&str2:0x7ffdbe2b9300
0x55843dca1e70
```

最佳策略

Fackbook提出的最佳策略，将三者进行结合：

因为以上三种方式，都不能解决所有可能遇到的字符串的情况，各有所长，又各有缺陷。综合考虑所有情况之后，facebook开源的folly库中，实现了一个fbstring，它根据字符串的不同长度使用不同的拷贝策略， 最终每个fbstring对象占据的空间大小都是24字节。

1. 很短的（0~22）字符串用SSO，23字节表示字符串（包括'\0'）,1字节表示长度
2. 中等长度的（23~255）字符串用eager copy，8字节字符串指针，8字节size，8字节capacity.
3. 很长的(大于255)字符串用COW，8字节指针（字符串和引用计数），8字节size，8字节capacity.