sylar笔记

部分笔记在代码内

一、智能指针

1

2陷阱

3 make_shared

4

1.1 左值与右值

左值：可以出现在 operator= 左侧的值；

右值：可以出现在 operator= 右侧的值；

当然这并不是全部正确的，但百分之90多都是这种情况，但有例外：

std::string();类似这种一个类加括号也就是临时变量都是右值；

函数的形参永远是左值；

1.2智能指针的三个常用函数

1.2.1.**get**() 获取智能指针托管的指针地址

```
// 定义智能指针
auto_ptr<Test> test(new Test);

Test *tmp = test.get();      // 获取指针返回
cout << "tmp->debug: " << tmp->getDebug() << endl;
```

但我们一般不会这样使用，因为都可以直接使用智能指针去操作，除非有一些特殊情况。
函数原型:

```
_NODISCARD _Ty * get() const noexcept
{   // return wrapped pointer
    return (_Myptr);
}
```

1.2.2.**release**() 取消智能指针对动态内存的托管

```
// 定义智能指针
auto_ptr<Test> test(new Test);

Test *tmp2 = test.release();    // 取消智能指针对动态内存的托管
delete tmp2;    // 之前分配的内存需要自己手动释放
```

也就是智能指针不再对该指针进行管理，改由管理员进行管理!
函数原型:

```
_Ty * release() noexcept
{   // return wrapped pointer and give up ownership
    _Ty * _Tmp = _Myptr;
    _Myptr = nullptr;
    return (_Tmp);
}
```

1.2.3.**reset**() 重置智能指针托管的内存地址，如果地址不一致，原来的会被析构掉

```
// 定义智能指针
auto_ptr<Test> test(new Test);

test.reset();           // 释放掉智能指针托管的指针内存，并将其置NULL

test.reset(new Test()); // 释放掉智能指针托管的指针内存，并将参数指针取代之
```

reset函数会将参数的指针(不指定则为NULL)，与托管的指针比较，如果地址不一致，那么就会析构掉原来托管的指针，然后使用参数的指针替代之。然后智能指针就会托管参数的那个指针了。

函数原型：

```
void reset(_Ty * _Ptr = nullptr)
{   // destroy designated object and store new pointer
    if (_Ptr != _Myptr)
        delete _Myptr;
    _Myptr = _Ptr;
}
```

1.3 auto_ptr

使用建议:

1. 尽可能不要将auto_ptr 变量定义为全局变量或指针；

```
// 没有意义，全局变量也是一样
auto_ptr<Test> *tp = new auto_ptr<Test>(new Test);
```

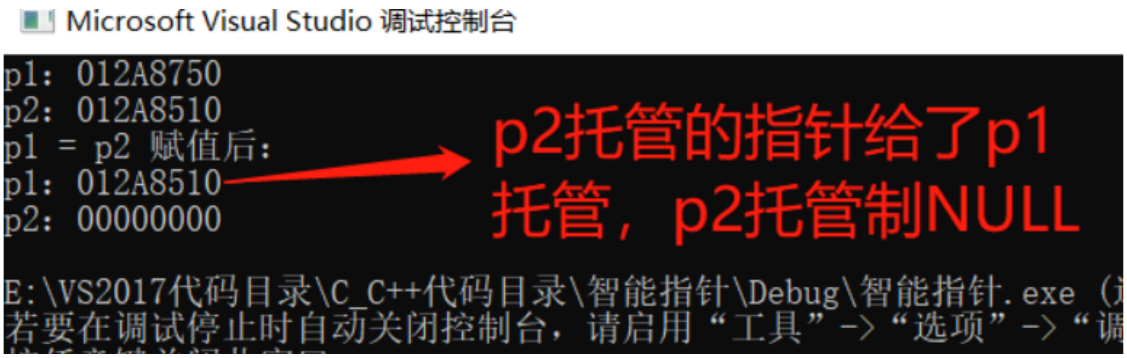
2. 除非自己知道后果，不要把auto_ptr 智能指针赋值给同类型的另外一个 智能指针；

```
auto_ptr<Test> t1(new Test);
auto_ptr<Test> t2(new Test);
t1 = t2;    // 不要这样操作...
```

3. C++11 后auto_ptr 已经被“抛弃”，已使用unique_ptr替代！C++11后不建议使用auto_ptr。
4. **auto_ptr 被C++11抛弃的主要原因**

   a.复制或者赋值都会改变资源的所有权

```
// auto_ptr 被C++11抛弃的主要原因
auto_ptr<string> p1(new string("I'm Li Ming!"));
auto_ptr<string> p2(new string("I'm age 22."));

cout << "p1: " << p1.get() << endl;
cout << "p2: " << p2.get() << endl;

// p2赋值给p1后，首先p1会先将自己原先托管的指针释放掉，然后接收托管p2所托管的指针，
// 然后p2所托管的指针制NULL，也就是p1托管了p2托管的指针，而p2放弃了托管。
p1 = p2;
cout << "p1 = p2 赋值后: " << endl;
cout << "p1: " << p1.get() << endl;
cout << "p2: " << p2.get() << endl;
```



b.在STL容器中使用auto_ptr存在着重大风险，因为容器内的元素必须支持可复制和可赋值

```
vector<auto_ptr<string>> vec;
auto_ptr<string> p3(new string("I'm P3"));
auto_ptr<string> p4(new string("I'm P4"));

// 必须使用std::move修饰成右值，才可以进行插入容器中
vec.push_back(std::move(p3));
vec.push_back(std::move(p4));

cout << "vec.at(0): " <<  *vec.at(0) << endl;
cout << "vec[1]: " <<  *vec[1] << endl;


// 风险来了:
vec[0] = vec[1];    // 如果进行赋值，问题又回到了上面一个问题中。
cout << "vec.at(0): " << *vec.at(0) << endl;
cout << "vec[1]: " << *vec[1] << endl;

//访问越界
```

c.不支持对象数组的内存管理

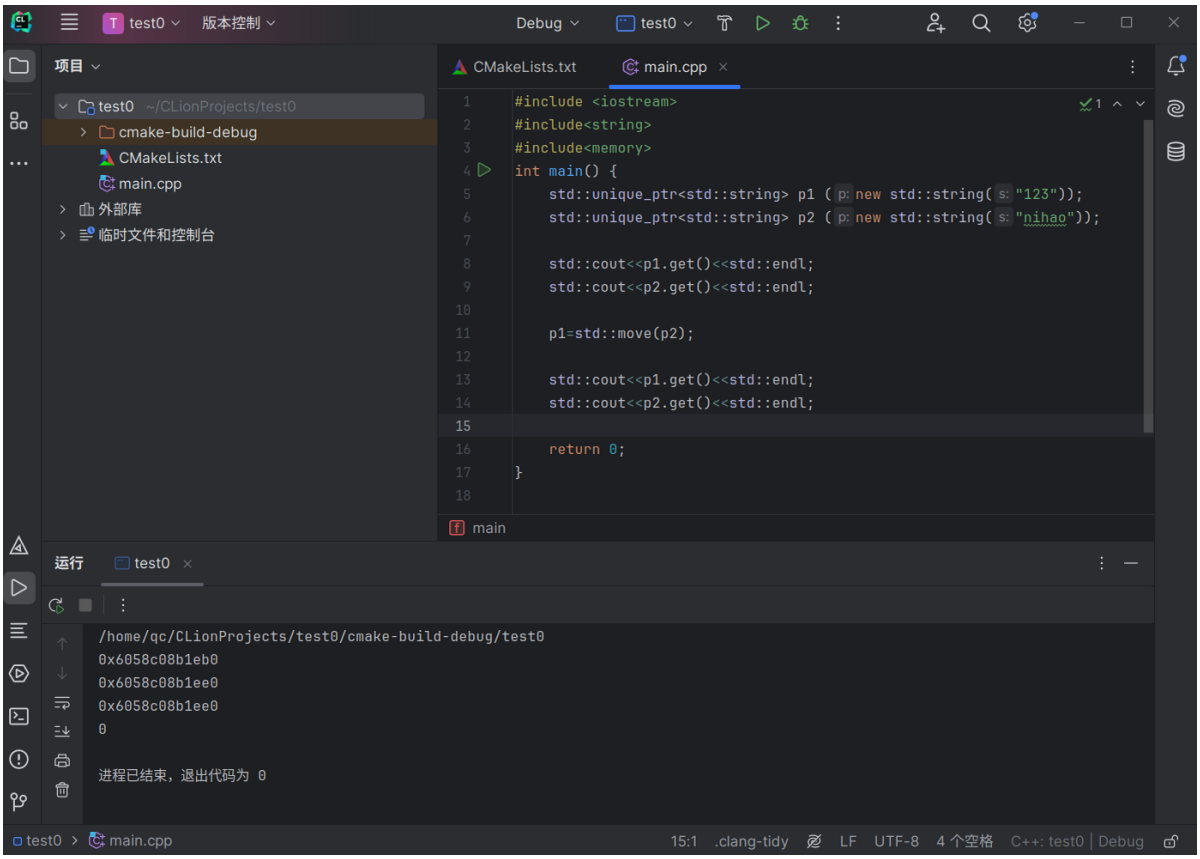
```
auto_ptr<int[]> array(new int[5]);  // 不能这样定义
```

1.4 unique_ptr

1.4.1**unique_ptr特性**

1. 基于排他所有权模式：两个指针不能指向同一个资源
2. 无法进行左值unique_ptr复制构造，也无法进行左值复制赋值操作，但允许临时右值赋值构造和赋值
3. 保存指向某个对象的指针，当它本身离开作用域时会自动释放它指向的对象。
4. 在容器中保存指针是安全的

1.**move()**使用 2.赋值操作后会托管原指针，原指针置0



A. 无法进行左值复制赋值操作，但允许临时右值赋值构造和赋值

```
unique_ptr<string> p1(new string("I'm Li Ming!"));
unique_ptr<string> p2(new string("I'm age 22."));

cout << "p1: " << p1.get() << endl;
cout << "p2: " << p2.get() << endl;

p1 = p2;                    // 禁止左值赋值
unique_ptr<string> p3(p2);  // 禁止左值赋值构造

unique_ptr<string> p3(std::move(p1));
p1 = std::move(p2); // 使用move把左值转成右值就可以赋值了，效果和auto_ptr赋值一样

cout << "p1 = p2 赋值后: " << endl;
cout << "p1: " << p1.get() << endl;
cout << "p2: " << p2.get() << endl;
```

B. 在 STL 容器中使用unique_ptr，不允许直接赋值

```
vector<unique_ptr<string>> vec;
unique_ptr<string> p3(new string("I'm P3"));
unique_ptr<string> p4(new string("I'm P4"));

vec.push_back(std::move(p3));
vec.push_back(std::move(p4));
```

```
cout << "vec.at(0): " << *vec.at(0) << endl;
cout << "vec[1]: " << *vec[1] << endl;

vec[0] = vec[1];    /* 不允许直接赋值 */
vec[0] = std::move(vec[1]);     // 需要使用move修饰，使得程序员知道后果

cout << "vec.at(0): " << *vec.at(0) << endl;
cout << "vec[1]: " << *vec[1] << endl;

//当然，运行后是直接报错的，因为vec[1]已经是NULL了，再继续访问就越界了。
```

C. 支持对象数组的内存管理

```
// 会自动调用delete [] 函数去释放内存
unique_ptr<int[]> array(new int[5]);    // 支持这样定义
```

1.4.2**auto_ptr 与 unique_ptr智能指针的内存管理陷阱**

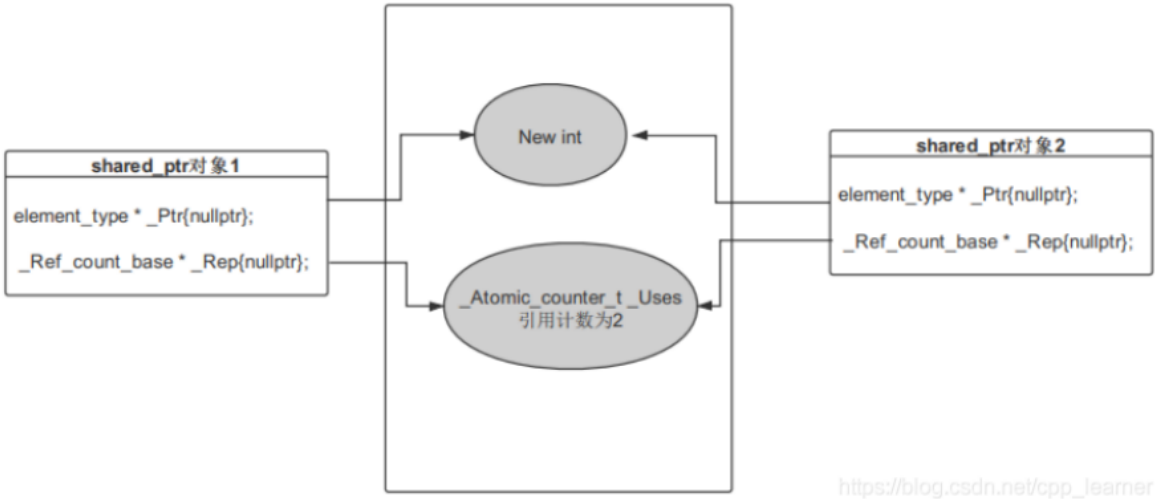
```
auto_ptr<string> p1;
string *str = new string("智能指针的内存管理陷阱");
p1.reset(str);  // p1托管str指针
{
    auto_ptr<string> p2;
    p2.reset(str);  // p2接管str指针时，会先取消p1的托管，然后再对str的托管
}

// 此时p1已经没有托管内容指针了，为NULL，在使用它就会内存报错!
cout << "str: " << *p1 << endl;
```

这是由于auto_ptr 与 unique_ptr的排他性所导致的!
为了解决这样的问题，我们可以使用shared_ptr指针指针!

1.5 shared_ptr

熟悉了unique_ptr 后，其实我们发现unique_ptr 这种排他型的内存管理并不能适应所有情况，有很大的局限！如果需要多个指针变量共享怎么办?

如果有一种方式，可以记录引用特定内存对象的智能指针数量，当复制或拷贝时，引用计数加1，当智能指针析构时，引用计数减1，如果计数为零，代表已经没有指针指向这块内存，那么我们就释放它！这就是 shared_ptr 采用的策略!

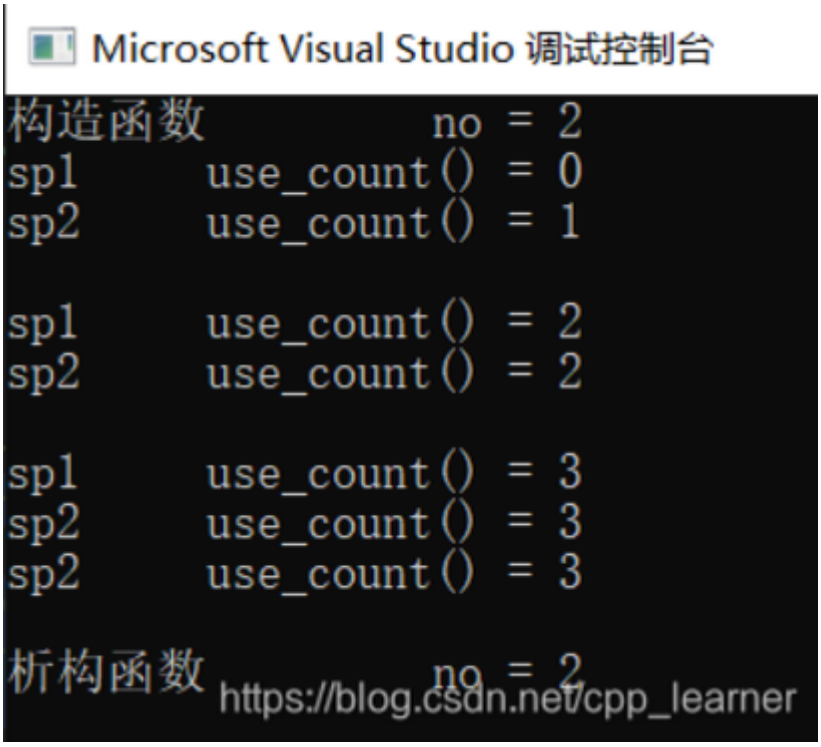


引用计数的使用

```
shared_ptr<Person> sp1;

shared_ptr<Person> sp2(new Person(2));

// 获取智能指针管控的共享指针的数量 use_count(): 引用计数
cout << "sp1    use_count() = " << sp1.use_count() << endl;
cout << "sp2    use_count() = " << sp2.use_count() << endl << endl;

// 共享
sp1 = sp2;

cout << "sp1    use_count() = " << sp1.use_count() << endl;
cout << "sp2    use_count() = " << sp2.use_count() << endl << endl;

shared_ptr<Person> sp3(sp1);
cout << "sp1    use_count() = " << sp1.use_count() << endl;
cout << "sp2    use_count() = " << sp2.use_count() << endl;
cout << "sp2    use_count() = " << sp3.use_count() << endl << endl;
```

如上代码，sp1 = sp2; 和 shared_ptr< Person > sp3(sp1);就是在使用引用计数了。

sp1 = sp2; --> sp1和sp2共同托管同一个指针，所以他们的引用计数为2；
shared_ptr< Person > sp3(sp1); --> sp1和sp2和sp3共同托管同一个指针，所以他们的引用计数为3；



1.5.1理解make_shared

为什么要使用make_shared

- shared_ptr：可以指向特定类型的对象，用于自动释放所指的对象
- make_shared：功能是在动态内存中分配一个对象并初始化它，返回指向此对象的`shared_ptr`；

```
shared_ptr<string> p1 = make_shared<string>(10, '9');

shared_ptr<string> p2 = make_shared<string>("hello");

shared_ptr<string> p3 = make_shared<string>();
```

从上面可以看出

1）make_shared是一个模板函数；
2）make_shared必须**显式指定想要创建的对象类型**，如上题所示make_shared(10, 9),如果不传递显式模板实参string类型，make_shared无法从(10, '9')两个模板参数中推断出其创建对象类型。
3）make_shared在**传递参数格式是可变的**，参数传递为生成类型的构造函数参数，因此在创建shared_ptr对象的过程中调用了类型T的某一个构造函数。

1.5.2make_shared VS std::shared_ptr区别

为什么不用std::shared_ptr 的构造函数的构造函数，而是要用std::make_shared呢?

优点

效率更高

`shared_ptr`需要维护引用计数的信息。

- 强引用，用来记录当前有多少个存活的`shared_ptr`正在持有该对象，共享的对象会在最后一个强引用离开的时候销毁（也可能释放）
- 弱引用，用来记录当前有多少个正在观察该对象的`weak_ptr`，当最后一个弱引用离开的时候，共享的内部信息控制块会被销毁和释放（共享的对象也会被释放，如果还没有释放的话）

如果你通过使用原始的`new`表示分配对象，然后传递给`shared_ptr`（也就是shared_ptr的构造函数）的话，`shared_ptr`的实现没有办法选择, 而只能单独的分配控制块:

```
auto p = new widget();
shared_ptr sp1{ p }, sp2{ sp1 };
```

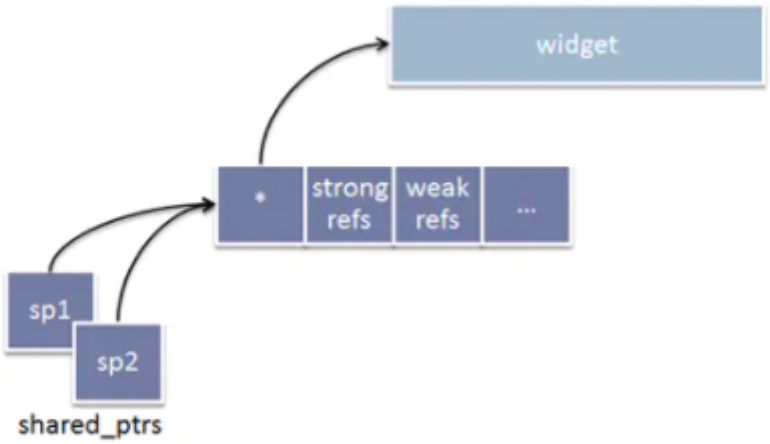


Figure 2(a): Approximate memory layout resulting from the code
auto p = new widget();
shared_ptr<widget> sp1{ p }, sp2{ sp1 };

使用shred_ptr初始化 CSDN @OceanStar的学习笔记

如果选择使用 make_shared 的话, 情况就会变成下面这样:

```
auto sp1 = make_shared(), sp2{ sp1 };
```

Figure 2(b): Approximate memory layout resulting from the code
auto sp1 = make_shared<widget>(), sp2{ sp1 };

使用make_shared CSDN @OceanStar的学习笔记

从上面可以看出，内存分配的动作，可以一次性完成，这减少了内存分配的次数，而内存分配是代价很高的操作。

总结：

```
struct A;
std::shared_ptr<A> p1 = std::make_shared<A>();
std::shared_ptr<A> p2(new A);
```

上面两者的区别：

- std::shared_ptr构造函数会执行两次内存申请，而`std::make_shared`则执行一次
- `std::shared_ptr`在实现的时候使用`refcount`技术，因此内部会有一个计数器(控制块)用来管理数据和一个指针。因此在`std::shared_ptr<A> p2(new A)`的时候，首先会申请数据的内存，然后申请内控制块，因此是两次内存申请，而`std::make_shared<A>()`则是只执行一次内存申请，将数据和控制块的申请放到一起。那这一次和两次的区别会带来什么不同的效果呢?

看下面的代码

```
void F(const std::shared_ptr<Lhs>& lhs, const std::shared_ptr<Rhs>& rhs) { /* ...
*/ }

F(std::shared_ptr<Lhs>(new Lhs("foo")),
  std::shared_ptr<Rhs>(new Rhs("bar")));
```

C++是不保证参数求值顺序，以及内部表示的求值顺序的，所以可能的执行顺序如下：

1. new Lhs(“foo”))
2. new Rhs(“bar”))
3. std::shared_ptr
4. std::shared_ptr

好了, 现在我们假设在第 2 步的时候, 抛出了一个异常 (比如 out of memory, 总之, Rhs 的构造函数异常了), 那么第一步申请的 Lhs 对象内存泄露了. 这个问题的核心在于, shared_ptr 没有立即获得裸指针.

我们可以用如下方式来修复这个问题.

```
auto lhs = std::shared_ptr<Lhs>(new Lhs("foo"));
auto rhs = std::shared_ptr<Rhs>(new Rhs("bar"));
F(lhs, rhs);
```

当然, 推荐的做法是使用 std::make_shared 来代替:

```
F(std::make_shared<Lhs>("foo"), std::make_shared<Rhs>("bar"));
```

make_ptr的优点是：**shared_ptr的构造函数会申请两次内存，而make_ptr会申请一次内存。一方面效率提交了，一方面保证了异常安全**

缺点

构造函数是保护或者私有时，无法使用make_shared

当我想要创建的对象没有共有的构造函数时，make_shared就无法使用了。当然，我们可以使用一些小技巧来解决这个问题

```
#include <memory>

class A
{
public:
    static std::shared_ptr<A> create()
    {
        struct make_shared_enabler : public A {};

        return std::make_shared<make_shared_enabler>();
    }

private:
    A() {}
};
```

对象的内存可能无法及时回收

因为make_shared只申请一次内存，因此控制块和数据块在一起，只有但控制块中不再使用时，内存才会释放。但是如果还有`weak_ptr`指向该块对象所在的内存，存放管理对象的部分内存仍然不会被释放，因而导致在所有其他`weak_ptr`销毁前整块内存（尽管被管理对象已经析构了）将不会进入系统的内存池循环使用

1.5.3shared_ptr陷阱

这篇文章写的挺好陷阱

请看下面例程:

```
#include <iostream>
#include <memory>

class Test1 {
public:
    // 无参构造函数
    Test1() = default;
    // 析构函数
    ~Test1() {std::cout << "~Test1" << std::endl;}
};

void test1(const std::shared_ptr<Test1> p) {
    std::cout << p.use_count() << std::endl;
}
```

```
void test2(const std::shared_ptr<Test1> &p) {
    std::cout << p.use_count() << std::endl;
}

void test3(const std::shared_ptr<const Test1> &p) {
    std::cout << p.use_count() << std::endl;
}

int main(int argc, const char * argv[]) {
    auto t1 = std::make_shared<Test1>();
    test1(t1);
    test2(t1);
    test3(t1);

    return 0;
}
```

请问，test1，test2和test3函数分别会打印几？运行一下可能大出所料：

```
2
1
2
~Test1
```

我们来一个一个解释。首先test1，参数是个智能指针，只不过加上了const，这个不要紧，p仍然是个新的指针，因此在函数体中，p和主函数中的t1这两个指针会同时指向同一个对象，所以引用计数为2，这个应该好理解。

再来看test2，参数是个智能指针的引用，而把t1传进来的时候，这个引用绑定了t1，因此p只是t1的引用，而并不是新的指针，因此，自始至终只有一个指针指向对象，所以，引用计数为1，这个应该也不难理解。

令人费解的就是这个test3了，明明这里p被声明成了引用，可为什么引用计数还是增加了呢？原因就在于，我们给Test1前面加个这个const。当这种情况的时候，大家一定不能再傻乎乎把智能指针无脑地当指针来对待了，虽然说它叫智能指针，但它本质上来说，还是个模板类，只不过起到了指针的作用罢了。既然是模板类，那么在传入参数时会被实体化成一个具体的类。因此，shared_ptr<Test>和shared_ptr<const Test>是两个不同的类，照理说，假如AB两个类没有任何关系的话，A的引用是不能用来接收B的对象的。然而此时却能够接收，那么此时必有蹊跷。

其实，在shared_ptr类中有类似这样一个定义：

```
template <typename T>
shared_ptr<const T>(const shared_ptr<T> &);
```

换句话说，我们可以用Test的指针构造出const Test的指针，然而，这个构造函数并没有用explicit修饰，因此允许隐式转换。而又因为很碰巧地，const引用除了可以作为引用来使用外，还可以作为普通变量来使用，比如说用常引用绑定常量时，相当于一个普通变量：

```
const int &a = 4; // 用常引用绑定常量
const int a = 4; // 和上面写法等价，都会占用一个int的空间
const int &b = a; // 用常引用绑定同类型的变量，是指针的语法糖（可以理解为别名）
```

正是因为【shared_ptr<Test>可以隐式构造shared_ptr<const Test>】以及【常引用有时等价于普通变量】这两件事情凑到一起，就造成了现在test3中的情况。首先，由于shared_ptr<Test>和shared_ptr<const Test>是不同的类型，因此，shared_ptr<const Test>不能直接用来绑定t1。但是，shared_ptr<Test>可以用来隐式构造shared_ptr<const Test>，因此，常引用const shared_ptr<const Test> &p作为了普通变量来使用，其构造参数就是t1，因此，这里p是一个新的对象，也就是另一个不同于t1的指针（要想验证这个说法，可以打印一下&p和&t1，它们地址是不同的），于是，引用计数变成了2。总结一下test3，其实等价成了下面的操作：

```
test3(t1); // 非同类型绑定，常引用作为新的普通变量来使用
// 上面的等价于下面的
test3(std::shared_ptr<const Test>(t1));
// 或者可以等价于下面的
const std::shared_ptr<const Test> tmp = t1;
test3(tmp); // 同类型绑定，常引用作为普通的引用来使用
```

所以这里的深坑就在于，shared_ptr<const T>和shared_ptr<T>要作为两个不同的类型来对待，而并不能当做同一类型的const和非const来对待。所以在使用常引用的时候，要小心类似的问题。

```
#include <iostream>
#include <memory>

class Test2;
class Test1 {
public:
    // 无参构造函数
    Test1() = default;
    // 析构函数
    ~Test1() {std::cout << "~Test1" << std::endl;}
    std::shared_ptr<Test2> t2; // 有个成员，是另一个类的智能指针
};

class Test2 {
public:
    // 无参构造函数
    Test2() = default;
    // 析构函数
    ~Test2() {std::cout << "~Test2" << std::endl;}
    std::shared_ptr<Test1> t1; // 有个成员，是另一个类的智能指针
};

int main(int argc, const char * argv[]) {
    auto t1 = std::make_shared<Test1>();
    auto t2 = std::make_shared<Test2>();
    t1->t2 = t2;
    t2->t1 = t1;

    return 0;
}
```

乍一看好像没什么问题，但我们执行后就会发现，t1和t2的析构函数都没有被调用过。说明两个对象都没有得到释放，发生了内存泄漏。这是为什么呢？是这样的，我们在创建两个对象时（我们不妨称Test1的这个对象叫对象1，Test2类型的这个对象叫对象2），t1指向对象1，t2指向对象2，之后，两个赋值语句结束以后，指向对象1的指针有t1和t2->t1，指向对象2的指针有t2和t1->t2，此时两个对象的引用计数都为2。当主函数结束后，t1和t2被释放，但对象1还有t2->t1在指向，对象2还有t1->t2在指向，因此两个对象的引用计数都是1，所以无法被释放。

> 这就是引用计数原理当中非常经典的循环引用问题，由于C++语言并不存在垃圾回收机制中可达性分析的机制，因此，这种循环引用问题是无法避免的。那么解决途径就是，让其中一个指针不去影响引用计数，这样就不会循环引用。所以，比较容易想到的做法就是把其中一个智能指针改成普通的指针。
>
> 但这样做虽然可以解决循环问题，但是，这个裸指针放在这里总是很眨眼，毕竟不属于智能指针体系，而且，裸指针是不安全的（比如我们可能会直接`delete`它这种风险操作），所以，`STL`还提供了另一种智能指针，这就是`weak_ptr`，用`weak_ptr`引用是不影响引用计数的，并且，`weak_ptr`还有一个功能就是，如果指向的对象被释放了，它会自动置空（也就是说不会出现野指针问题），所以这显然是优于裸指针的。因此，把上面例程可以这样更改：

```
#include <iostream>
#include <memory>

class Test2;
class Test1 {
public:
    // 无参构造函数
    Test1() = default;
    // 析构函数
    ~Test1() {std::cout << "~Test1" << std::endl;}
    std::shared_ptr<Test2> t2; // 有个成员，是另一个类的智能指针
};

class Test2 {
public:
    // 无参构造函数
    Test2() = default;
    // 析构函数
    ~Test2() {std::cout << "~Test2" << std::endl;}
    std::weak_ptr<Test1> t1; // 有个成员，是另一个类的智能指针
};

int main(int argc, const char * argv[]) {
    auto t1 = std::make_shared<Test1>();
    auto t2 = std::make_shared<Test2>();
    t1->t2 = t2;
    t2->t1 = t1;

    return 0;
}
```

只需要把其中的一个shared_ptr改成weak_ptr就可以完美解决所有问题。

但是，weak_ptr和shared_ptr在使用上还是有区别的，比如说weak_ptr不能直接使用*和->运算，也不能和nullptr进行比较。那么，我们应该怎么操作呢？

1.6 weak_ptr

weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作, 它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。 同时 weak_ptr 没有重载*和->但可以使用 lock 获得一个可用的 shared_ptr 对象。

1. 弱指针的使用;
   weak_ptr wpGirl_1; // 定义空的弱指针
   weak_ptr wpGirl_2(spGirl); // 使用共享指针构造
   wpGirl_1 = spGirl; // 允许共享指针赋值给弱指针

```
#include <iostream>
#include <memory>

class Frame {};

int main()
{
  std::shared_ptr<Frame> f(new Frame());
  std::weak_ptr<Frame> f1(f);                       // shared_ptr直接构造
  std::weak_ptr<Frame> f2 = f;                      // 隐式转换
  std::weak_ptr<Frame> f3(f1);                      // 拷贝构造函数
  std::weak_ptr<Frame> f4 = f1;                     // 拷贝构造函数
  std::weak_ptr<Frame> f5;
  f5 = f;                                           // 拷贝赋值函数
  f5 = f2;                                          // 拷贝赋值函数
  std::cout << f.use_count() << std::endl;          // 1

  return 0;
}
```

2. 弱指针也可以获得引用计数;
   wpGirl_1.use_count()
3. w.reset(...): 重置weak_ptr
4. 弱指针不支持 * 和 -> 对指针的访问;
   在必要的使用可以转换成共享指针 lock();

```
shared_ptr<Girl> sp_girl;
sp_girl = wpGirl_1.lock();

// 使用完之后，再将共享指针置NULL即可
sp_girl = NULL;
```

5.需要注意，**weak_ptr绑定到一个shared_ptr不会改变shared_ptr的引用计数。**

既然weak_ptr并不改变其所共享的shared_ptr实例的引用计数，那就可能存在weak_ptr指向的对象被释放掉这种情况。这时，就不能使用weak_ptr直接访问对象。那么如何判断weak_ptr指向对象是否存在呢？C++中提供了lock函数来实现该功能。如果对象存在，lock()函数返回一个指向共享对象的shared_ptr(引用计数会增1)，否则返回一个空shared_ptr。weak_ptr还提供了expired()函数来判断

所指对象是否已经被销毁。

1.7 make_unique 与make_shared

二、cmake使用

三、stringstream类

1.stringstream 是什么？

`stringstream` 是 `C++` 提供的专门用于处理字符串的 输入输出 流类。

这里稍微提一下c++中 “流” 的概念。在C++中，将数据从一个对象到另一个对象的流动抽象为“流”。流在使用前要被创建，使用后要被删除。数据的输入与输出是通过 `I/O` 流实现的，`cin` 和 `cout` 是c++预定义的流类对象。

`stringstream` 定义于头文件 `<sstream>`，它其实是个别名，具体定义如下：

typedef basic_stringstream stringstream;

类模板 `std::basic_stringstream` 实现基于字符串的流上的输入与输出操作。它等效地存储一个 `std::basic_string` 的实例，并在其上进行输入与输出操作。继承图如下：

2.、stringstream 的用法

✨构造函数

`stringstream` 的构造函数有很多，这里列举最为常用的两个构造函数：

1. 创建一个对象，向对象输入字符串：

```
// 创建一个 string类  对象 s
string s("hello stringstream");
// 创建一个 stringstraeam类 对象 ss
stringstream ss;

// 向对象输入字符串 : "<<" 表示向一个对象中输入
ss << s;
cout << ss.str() << endl;
```

2. 在创建对象的时候使用字符串初始化：

```
// 创建一个 stringstraeam类 对象 ss
stringstream ss("hello stringstream");

cout << ss.str() << endl;
```

两种方式都可以创建对象，但创建后的对象用法不一样，详见后面的示例。

✨输出字符串

stringstream 可以将存储于内部的字符串输出，需要调用 str() 函数，不可直接输出：

```
std::cout << ss.str() << std::endl;

// std::cout << ss << std::endl;         // 错误不可直接输出

 注意：cout << ss << endl;    是错误的，不可以直接输出
```

✨两种构造函数带来的不同

> 上面阐述了两种构造函数，利用不同的构造函数创建对象，对象具体的操作也不同：

1. 第一种构造方式

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    stringstream ss1;
    ss1 << "fre";
    ss1 << "gre";
    cout << ss1.str() << endl;


return 0;

}

/*
输出:
fregre
*/
```

可以发现，两个字符串直接拼接在了一起

2. 第二种构造方式

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    string str("asd");
    // 第二种构造
    stringstream ss2(str);
    cout << ss2.str() << endl;


// 第一种构造
ss2 << "r";
cout << ss2.str() << endl;
```

```
ss2 << "13";
cout << ss2.str() << endl;

ss2 << "hy";
cout << ss2.str() << endl;

return 0;



}

/*
输出:
asd
rsd
r13
r13hy
*/
```

可以发现，利用第一种构造函数创建对象时，输入字符串后直接进行字符串拼接，而第二种构造方式，在进行字符串拼接时，首先把原本的字符串覆盖掉，之后再进行拼接。

如果不想原来的字符串被覆盖，则需要换一种构造方式，如下：

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    ostringstream ss("1 2 3 4 ", std::ios_base::ate);   // append 方式追加
    cout << ss.str() << endl;

    ss << "5 3 4";
    cout << ss.str() << endl;

    return 0;

}
/*
输出:
1 2 3 4
1 2 3 4 5 3 4
*/
```

✨修改、清空 stringstream 内容

`stringstream` 的内容可以通过 `str()` 函数进行修改、清空：

```
#include <iostream>
#include <sstream>
```

```
using namespace std;

int main()
{
    stringstream ss("hello string");
    cout << ss.str() << endl;

    // 修改内容
    ss.str("hello stringstream");
    cout << ss.str() << endl;

    // 清空内容
    ss.str("");
    cout << ss.str() << endl;

    return 0;
}

/*
输出:
hello string
hello stringstream

*/
```

3、stringsteam 的用途

✨ **利用 stringstream 去除字符串空格**

`stringstream` 默认是以空格来分割字符串的，利用 `stringstream` 去除字符串空格非常方便:

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    stringstream ss("hello string and stringstream");
    cout << ss.str() << endl;

    cout<< endl;

    string str;
    // 注意:  stringstream 是一个单词一个单词 ”流入“ string 的
    while (ss >> str)
    {
        cout << str << endl;
    }

    return 0;
}

/*
输出:
hello string and stringstream
```

```
hello
string
and
stringstream
*/
```

✨ 利用 stringstream 指定字符分割字符串

```
      /* 上面描述了利用 stringstream 去除字符串空格，其实就是利用空格来分割字符串，同样，也可以指定其他字符对字符串进行分割，这需要与 getline() 函数搭配使用，下面以逗号分割字符串为例：

getline 详解：
getline()的原型是istream& getline ( istream &is , string &str , char delim );

    其中 istream &is 表示一个输入流，
    例如，可使用cin；
    string str ; getline(cin ,str)
    也可以使用 stringstream
    stringstream ss(“test#”) ; getline(ss,str)
    char delim表示遇到这个字符停止读入，通常系统默认该字符为’\n’，也可以自定义字符
*/
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    string source = "abc,123,<!>";
    stringstream ss(source);
    cout << ss.str() << endl;

    cout<< endl;

    string str;
    while (getline(ss, str, ','))
    {
        cout << str << endl;
    }

    return 0;
}

/*
输出：
abc,123,<!>

abc
123
<!>
*/

 //上述代码以逗号作为分割依据来分割字符串，同样的还可以扩展到其他字符。
```

✨ 类型转换

使用 stringstream 进行类型转换

 以下是一个使用 stringstream 将数字转换为字符串的例子：

```
#include <sstream>
#include <iostream>
#include <string>

int main() {
    int num = 123;
    std::stringstream ss;
    ss << num; // 将整数放入流中
    std::string str = ss.str(); // 使用str()函数 从流中提取字符串
    std::cout << str << std::endl; // 输出：123
}
```

反过来，也可以将字符串转换为数值类型：

```
#include <sstream>
#include <iostream>
#include <string>

int main() {
    std::string str = "456";
    std::stringstream ss(str); // 初始化stringstream
    int num;
    ss >> num; // 从流中提取整数
    std::cout << num << std::endl; // 输出：456
}
```

4、常考面试题

✨计算字符串中的单词个数：

```
输入：“hello world c plus plus”
输出：5

#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main() {
    string str = "hello world c plus plus";
    int count = 0;
    stringstream ss(str);
    string word;
    while (ss >> word)
        count++;
    cout << count << endl;

    system("pause");
    return 0;
```

```
}
```

✨ 反转字符串中的单词 （重点）

链接：151. 反转字符串中的单词

```
class Solution {
public:
    string reverseWords(string s)
    {
         string res,temp;
         stringstream ss(s);
         while(ss>>temp)
         {
            res = temp + " " + res;
         }
         if(!res.empty())
         {
            res.pop_back();
         }
         return res;
    }
};
```

四、宏定义与lambda函数

示例：

宏定义

```
const char* LogLevel::ToString(LogLevel::Level level) {
    switch(level) {
#define XX(name) \    /*在 C 和 C++ 的宏定义中，反斜杠 (\) 用于指示宏定义的续行符。它告诉预处理器当前行尚未结束，下一行仍是本行的一部分。这样可以将长宏定义分成多行，增强代码的可读性。*/
    case LogLevel::name: \
        return #name; \
        break;

        XX(DEBUG);
        XX(INFO);
        XX(WARN);
        XX(ERROR);
        XX(FATAL);
#undef XX
            default:
                return "UNKNOW";
        }
        return "UNKNOW";
    }
}
```

这段代码中使用了宏定义和switch-case语句来实现日志级别转换为字符串的功能。具体来说，代码中的写法通过宏定义简化了case分支的重复代码，下面我将详细解释其中的机制。

1. **宏定义**：

```
#define XX(name) \
    case LogLevel::name: \
        return #name; \
        break;
```

这段宏定义定义了一个名为`XX`的宏，它接受一个参数`name`，并生成相应的`case`语句。例如，`XX(DEBUG)`将展开为：

```
case LogLevel::DEBUG:
    return "DEBUG";
    break;
```

2. **switch-case 语句**：

```
switch(level) {
    XX(DEBUG);
    XX(INFO);
    XX(WARN);
    XX(ERROR);
    XX(FATAL);
    default:
        return "UNKNOW";
}
```

通过调用宏`XX`，这段代码为每个日志级别生成一个`case`分支，并返回相应的字符串。

这种写法的优势在于：

- **减少重复代码**：宏定义减少了重复的`case`分支代码，提高了代码的可维护性。
- **易于扩展**：如果需要增加新的日志级别，只需在宏调用中添加相应的日志级别即可，无需重复编写`case`语句。

`#undef`用法

```
cpp
复制代码
#undef XX
```

在使用完宏定义后，使用`#undef XX`取消宏定义，以免后续代码中再使用这个宏，避免潜在的命名冲突或意外行为。

整体代码结构可以更直观地理解为：

```
const char* LogLevel::ToString(LogLevel::Level level) {
    switch(level) {
        case LogLevel::DEBUG:
            return "DEBUG";
            break;
```

```
        case LogLevel::INFO:
            return "INFO";
            break;
        case LogLevel::WARN:
            return "WARN";
            break;
        case LogLevel::ERROR:
            return "ERROR";
            break;
        case LogLevel::FATAL:
            return "FATAL";
            break;
        default:
            return "UNKNOW";
    }
    return "UNKNOW";
}
```

通过使用宏定义，实际代码变得更简洁且更易于维护。

```
static std::map<std::string, std::function<FormatItem::ptr(const std::string&
str)>> s_format_items = {
#define XX(str, C) \                /*在 C 和 C++ 的宏定义中，反斜杠 (\) 用于指示宏定义
的续行符。它告诉预处理器当前行尚未结束，下一行仍是本行的一部分。这样可以将长宏定义分成多行，增强代
码的可读性。*/
        {#str, [](const std::string& fmt) { return FormatItem::ptr(new C(fmt));}}

                XX(m, MessageFormatItem),           //m:消息
                XX(p, LevelFormatItem),             //p:日志级别
                XX(r, ElapseFormatItem),            //r:累计毫秒数
                XX(c, NameFormatItem),              //c:日志名称
                XX(t, ThreadIdFormatItem),          //t:线程id
                XX(n, NewLineFormatItem),           //n:换行
                XX(d, DateTimeFormatItem),          //d:时间
                XX(f, FilenameFormatItem),          //f:文件名
                XX(l, LineFormatItem),              //l:行号
                XX(T, TabFormatItem),               //T:Tab
                XX(F, FiberIdFormatItem),           //F:协程id
                XX(N, ThreadNameFormatItem),        //N:线程名称
#undef XX
        };
```

宏定义 `#define` 是 C 和 C++ 预处理器中的一种指令，用于定义宏。宏可以使代码更简洁、更易读，并且在编译时会进行替换。具体到你的代码中的 `#define` 和 `XX`，我们可以这样解释：

1. **宏定义 `#define`**：
   宏定义 `#define` 可以用来定义常量、函数宏等。其语法通常是：

   ```
   #define 宏名 替换文本
   ```

   宏定义可以在代码的其他地方用来代替复杂的表达式或重复的代码段。

2. **宏 `XX`**:
   在你的代码中，`#define XX(str, C)` 定义了一个宏 `XX`，它带有两个参数 `str` 和 `C`。这个宏定义后面的代码会将 `XX(str, C)` 替换为：

   ```
   {#str, [](const std::string& fmt) { return FormatItem::ptr(new C(fmt));}}
   ```

   其中 `#str` 将 `str` 变成一个字符串，即将 `str` 替换为 `str` 的字符串字面量。

3. **使用 `XX` 宏**:
   宏 `XX` 在后续代码中被多次使用，每次使用时会替换为之前定义的文本。例如：

   ```
   XX(m, MessageFormatItem)  // 替换为 {"m", [](const std::string& fmt) { return FormatItem::ptr(new MessageFormatItem(fmt)); }}
   XX(p, LevelFormatItem)    // 替换为 {"p", [](const std::string& fmt) { return FormatItem::ptr(new LevelFormatItem(fmt)); }}
   ```

   依此类推，每个 `XX` 宏调用会生成一个键值对，键是 `str` 的字符串，值是一个返回 `FormatItem` 智能指针的 lambda 函数。

结合以上解释，你的代码中的宏定义和 `XX` 的作用是简化和自动化生成一系列键值对，并将其插入到 `std::map<std::string, std::function<FormatItem::ptr(const std::string& str)>>` 这个映射中。这样可以避免重复代码，提高代码的可读性和维护性。

在 C 和 C++ 预处理器中，`#name` 是一种称为“字符串化（stringizing）”的操作符。它将宏参数 `name` 转换为一个字符串字面量。

具体来说，当你在宏定义中使用 `#name` 时，预处理器会将宏参数 `name` 转换为一个用双引号括起来的字符串。这在创建键值对时特别有用，因为你可以将宏参数直接转换为字符串形式。下面是一些示例和详细解释：

#name

假设有以下宏定义：

```
#define STRINGIZE(name) #name
```

然后在代码中使用这个宏：

```
STRINGIZE(hello)
```

预处理器会将其展开为：

```
"hello"
```

这里，`#name` 将 `hello` 转换为了 `"hello"`。

在代码中，宏定义 `#define XX(str, C)` 中使用了 `#str`：

```
#define XX(str, C) \
    {#str, [](const std::string& fmt) { return FormatItem::ptr(new C(fmt)); }}
```

当你调用这个宏时，例如：

```
XX(m, MessageFormatItem)
```

预处理器会将其展开为：

```
{"m", [](const std::string& fmt) { return FormatItem::ptr(new
MessageFormatItem(fmt)); }}
```

这里，`#str` 将宏参数 `m` 转换为了字符串 `"m"`，从而生成了一个键值对，其中键是 `"m"`，值是一个 lambda 表达式。

详细解释

1. `#str`：
   `#str` 将宏参数 `str` 转换为字符串字面量。例如，如果 `str` 是 `m`，那么 `#str` 会被替换为 `"m"`。
2. **宏展开**：
   使用 `XX(m, MessageFormatItem)` 时，宏 `XX` 展开为：

   ```
   {"m", [](const std::string& fmt) { return FormatItem::ptr(new
   MessageFormatItem(fmt)); }}
   ```

这样，你可以方便地创建一系列键值对，并将其插入到 `std::map<std::string, std::function<FormatItem::ptr(const std::string& str)>>` 中，每个键值对都包含一个标识符字符串和一个用于创建 `FormatItem` 对象的 lambda 函数。这使得代码更加简洁和易于维护。

匿名函数--lambda函数

1.匿名函数的基本语法

```
[捕获列表](参数列表) mutable(可选) 异常属性 -> 返回类型 {
    // 函数体
}
```

语法规则：lambda表达式可以看成是一般函数的函数名被略去，返回值使用了一个 -> 的形式表示。唯一与普通函数不同的是增加了“捕获列表”。

```
// lambda_test lambda_test.cc
#include <iostream>

using namespace std;

void test01()
{
    cout << "test01" << endl;
    auto Add = [](int a, int b) -> int {
        return a + b;
    };

    cout << Add(1, 2) << endl;
}
```

```
int main(int argc,char **argv)
{
    test01();

    return 0;
}
```

编译（要指定-std=c++11）：

```
g++ -o lambda_test lambda_test.cc -std=c++11
```

输出：

```
$ ./lambda_test
test01
3
```

一般情况下，编译器可以自动推断出lambda表达式的返回类型，所以我们可以不指定返回类型，即：

```
// lambda_test lambda_test.cc
#include <iostream>

using namespace std;

void test02()
{
    cout << "test02" << endl;
    auto Add = [](int a, int b){
        return a + b;
    };

    cout << Add(1, 2) << endl;
}

int main(int argc,char **argv)
{
    //test01();
    test02();
    return 0;
}
```

但是如果函数体内有多个return语句时，编译器无法自动推断出返回类型，此时必须指定返回类型。

捕获列表

有时候，需要在匿名函数内使用外部变量，所以用捕获列表来传递参数。根据传递参数的行为，捕获列表可分为以下几种：

2.1、值捕获

与参数传值类似，值捕获的前提是变量可以拷贝，不同之处则在于，**被捕获的变量在 lambda表达式被创建时拷贝**，而不是在调用时才拷贝：

```
// lambda_test lambda_test.cc
#include <iostream>

using namespace std;

void test03()
{
    cout << "test03" << endl;
    int c = 20;
    int d = 30;
    auto Add = [c,d](int a, int b) {
        cout << "d = "<< d << endl;
        return c;
    };

    d = 10; // 在这里修改 d 的值，会改变 Add里的 d 值吗?

    cout << Add(1, 2) << endl;
}


int main(int argc,char **argv)
{

    //test01();
    //test02();

    test03();

    return 0;
}
```

执行结果：

```
$ ./lambda_test
test03
d = 30
20
```

2.2、引用捕获

与引用传参类似，引用捕获保存的是引用，值会发生变化。

```
#include <iostream>

using namespace std;
void test04()
{
    cout << "test04" << endl;
    int c = 20;
    int d = 30;
    auto Add = [c, &d](int a, int b) {
        cout << "c = " << c << endl;
        cout << "d = " << d << endl;
        return c;
    };

    d = 10;//在这里修改d的值，会改变Add里的d值吗?

    cout << Add(1, 2) << endl;
}


int main(int argc,char **argv)
{

    //test01();
    //test02();
    //test03();
    test04();

    return 0;
}
```

执行结果：

```
$ ./lambda_test
test04
c = 20
d = 10
20
```

2.3、隐式捕获

手动书写捕获列表有时候是非常复杂的，这种机械性的工作可以交给编译器来处理，这时候可以在捕获列表中写一个 **& 或 =** 向编译器声明采用**引用捕获或者值捕获**。编译器会将外部变量全部捕获。

```
#include <iostream>

using namespace std;
```

```
void test05()
{
    cout << "test05" << endl;
    int c = 20;
    int d = 30;
    auto Add = [&](int a, int b) {
        cout << "c = " << c << endl;
        cout << "d = " << d << endl;
        return c;
    };

    d = 10;//在这里修改d的值，会改变Add里的d值吗?

    cout << Add(1, 2) << endl;
}

void test06()
{
    cout << "test06" << endl;
    int c = 20;
    int d = 30;
    auto Add = [=](int a, int b) {
        cout << "c = " << c << endl;
        cout << "d = " << d << endl;
        return c;
    };

    d = 10;//在这里修改d的值，会改变Add里的d值吗?

    cout << Add(1, 2) << endl;
}

int main(int argc,char **argv)
{

    //test01();
    //test02();
    //test03();
    test05();
    test06();

    return 0;
}
```

输出：

```
$ ./lambda_test
test05
c = 20
d = 10
20
test06
c = 20
d = 30
20
```

2.4、空捕获列表

捕获列表'[]'中为空，表示Lambda不能使用所在函数中的变量。

```
void test07()
{
    cout << "test07" << endl;
    int c = 20;
    int d = 30;
    auto Add = [](int a, int b) {
        cout << "c = " << c << endl; // 编译报错
        cout << "d = " << d << endl; // 编译报错
        return c;                    // 编译报错
    };

    d = 10;

    cout << Add(1, 2) << endl;
}
```

编译报错：

```
lambda_test.cc:95:14: note: the lambda has no capture-default
  auto Add = [](int a, int b) {
             ^
lambda_test.cc:93:6: note: ‘int c’ declared here
  int c = 20;
      ^
lambda_test.cc:97:21: error: ‘d’ is not captured
   cout << "d = " << d << endl;
```

2.5、表达式捕获

上面提到的值捕获、引用捕获都是已经在外层作用域声明的变量，因此这些捕获方式捕获的均为左值，而不能捕获右值。

C++14之后支持捕获右值，允许捕获的成员用任意的表达式进行初始化，被声明的捕获变量类型会根据表达式进行判断，判断方式与使用 auto 本质上是相同的：

```
#include <iostream>
#include <memory>
```

```
using namespace std;

void test08()
{
    cout << "test08" << endl;

    auto important = make_unique<int>(1);
    auto Add = [v1 = 1, v2 = std::move(important)](int a, int b)->int{

        return a + b + v1 + (*v2);
    };

    cout << Add(1, 2) << endl;
}

int main(int argc,char **argv)
{

    test08();

    return 0;
}
```

执行结果:

```
$ ./lambda_test
test08
5
```

2.6、泛型 Lambda

在C++14之前，lambda表示的形参只能指定具体的类型，没法泛型化。从 C++14 开始， Lambda 函数的形式参数可以使用 auto关键字来产生意义上的泛型。
简单点说，就是通过auto使lambda自适应参数类型:

```
#include <iostream>

using namespace std;

void test09()
{
    cout << "test09" << endl;
    auto Add = [](auto a, auto b) {
        return a + b;
    };

    cout << Add(1, 2) << endl;
    cout << Add(1.1, 2.2) << endl;
}

int main(int argc,char **argv)
```

```
{

    test09();

    return 0;
}
```

执行结果：

```
./lambda_test
test09
3
3.3
```

2.7、可变lambda

（1）采用值捕获的方式，lambda不能修改其值，如果想要修改，使用mutable修饰。
（2）采用引用捕获的方式，lambda可以直接修改其值。

```
#include <iostream>

using namespace std;

void test10()
{
    cout << "test10" << endl;
    int v = 10;
    // 值捕获方式，使用mutable修饰，可以改变捕获的变量值
    auto tes = [v]() mutable {
        return ++v;
    };

    v = 5;
    auto a = tes();// a=11;
    cout << a << endl;
}

void test11()
{
    cout << "test11" << endl;
    int v = 10;
    auto Add = [&v]{
        return v++;
    };
    v = 6;
    cout << Add() << endl;
}

int main(int argc,char **argv)
{

    test10();
    test11();
```

```
    return 0;
}
```

执行结果：

```
$ ./lambda_test
test10
11
test11
6
```

2.8、混合捕获

1. 要求捕获列表中第一个元素必须是隐式捕获（&或=）。
2. 混合使用时，若隐式捕获采用引用捕获（&）则显式捕获的变量必须采用值捕获的方式。
3. 若隐式捕获采用值捕获（=），则显式捕获的变量必须采用引用捕获的方式。

```
#include <iostream>

using namespace std;

void test12()
{
    cout << "test12" << endl;

    int c = 12;
    int d = 30;
    int e = 30;
    // auto Add = [&, d, e](int a, int b)
    auto Add = [=, &c](int a, int b) -> int {
        c = a;
        cout << "d=" << d << ", e=" << e << endl;
        return c;
    };
    d = 20;
    cout << Add(1, 2) << endl;
    cout << "c:" << c << endl;
}

int main(int argc,char **argv)
{

    test12();

    return 0;
}
```

测试结果：

```
$ ./lambda_test
test12
d=30, e=30
1
c:1
```

2.10、Lambda捕获列表总结

捕获	含义
[]	空捕获列表，Lambda不能使用所在函数中的变量。
[names]	names是一个逗号分隔的名字列表，这些名字都是Lambda所在函数的局部变量。默认情况下，这些变量会被拷贝，然后按值传递，名字前面如果使用了&，则按引用传递
[&]	隐式捕获列表，Lambda体内使用的局部变量都按引用方式传递
[=]	隐式捕获列表，Lanbda体内使用的局部变量都按值传递
[&,identifier_list]	identifier_list是一个逗号分隔的列表，包含0个或多个来自所在函数的变量，这些变量采用值捕获的方式，其他变量则被隐式捕获，采用引用方式传递，identifier_list中的名字前面不能使用&。
[=,identifier_list]	identifier_list中的变量采用引用方式捕获，而被隐式捕获的变量都采用按值传递的方式捕获。identifier_list中的名字不能包含this，且这些名字面前必须使用&。

五、左值右值与move、 ref

在 C++ 中，左值（lvalue）和右值（rvalue）是两种主要的表达式类别，它们与对象存储、表达式的持久性、以及它们能否被赋值等属性有关。理解左值和右值对于深入学习 C++ 的内存管理、资源控制、以及现代 C++ 中的移动语义至关重要。

左值（Lvalue）

- **定义**：左值（lvalue, locator value）指的是一个表达式（不一定是变量）结束后仍然存在的持久对象。左值可以位于赋值表达式的左侧，这意味着你可以对其赋值。
- **特性**：左值通常表示对象的身份（它们在内存中的位置），而不仅仅是对象的值。例如，变量、数组的元素、对象的属性、返回引用的函数等。
- **例子**：变量名、数组的元素、对象的引用、返回引用的函数调用等。

```
int x = 10;  // x 是一个左值
x = 20;      // 正确，因为 x 是左值
```

右值（Rvalue）

- **定义**：右值（rvalue, read value）指的是临时的、非持久的对象，通常在表达式结束时就不再存在。右值可以位于赋值表达式的右侧，但通常不能在左侧（即你通常不能将一个值赋给一个右值）。
- **特性**：右值通常用来描述一个临时的值，这个值在表达式求值后就不再需要了。这包括字面量、运算表达式的结果、返回非引用类型的函数的调用结果等。
- **例子**：字面量（如 `42`、`"hello"`）、运算表达式的结果（如 `x + y`）、返回非引用的函数调用等。

```
int x = 10;
int y = x + 5;  // x + 5 是一个右值
// x + 5 = 10;  // 错误，因为 x + 5 是右值
```

右值的扩展：纯右值与将亡值

在 C++11 之后，为了支持移动语义，右值被进一步细分为**纯右值**（prvalue, pure rvalue）和**将亡值**（xvalue, expiring value）：

- **纯右值（prvalue）**：表示不与任何存储位置关联的表达式，如 `42` 或 `x + y`。
- **将亡值（xvalue）**：是一个右值，表示一个对象，即将销毁，且其资源可以被移动。例如通过 `std::move()` 转换的左值或返回右值引用的函数。

这些细分有助于在支持移动语义的语境下更精确地处理对象的状态，特别是在函数参数传递、返回值优化、以及泛型编程中。理解和正确使用这些概念可以显著提升程序性能和资源利用效率。

`std::ref` 在 C++ 中的主要作用是创建一个引用包装器，这对于在需要引用但是语境又只允许按值传递的情况下非常有用。下面，我将详细说明 `std::ref` 的几个主要应用场景和它如何发挥作用。

move使用

移动语义（Move semantics）是 C++11 版本中引入的一种特性，它允许资源（如内存、文件句柄等）在对象之间转移，而不是复制。这种语义显著提高了程序的性能，尤其是在处理大型数据结构和容器时。

为什么需要移动语义?

在 C++11 之前，对象间的数据通常通过复制实现，这意味着创建数据的副本。对于大型对象，如字符串、向量、或其他容器，这种复制操作可能会非常耗时和低效。例如，一个包含数百万元素的向量，复制它将涉及分配新内存并复制所有元素到新位置。

移动语义如何工作?

移动语义通过引入**右值引用**（一个新的引用类型）来实现。右值引用绑定到即将销毁的对象（也称为临时对象），允许开发者安全地从这些对象"窃取"资源，而不必复制它们。这种窃取是安全的，因为这些临时对象不会再被使用，它们的资源即将被销毁。

关键概念

- **右值引用**：用 `&&` 符号表示，比如 `int&&` 表示一个指向 `int` 的右值引用。它可以绑定到一个右值（临时对象），从而允许资源的转移。

- `std::move`：这是一个函数模板，用于将其参数显式转换为右值引用，使得资源可以被移动。这不是实际移动数据，而是创建一个状态，通过这个状态，资源可以被安全地窃取。

示例

假设有一个包含大量数据的 `std::vector`：

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v2 = std::move(v1);  // 使用 move 将 v1 的数据“移动”到 v2
```

在这个例子中，`v1` 的内部数据（例如指向堆上数据的指针）会被转移给 `v2`，而 `v1` 将处于一个安全的、有效的但未定义的状态。`v1` 不再拥有原来的数据，而 `v2` 现在拥有这些数据。这个操作非常快，因为它避免了复制大量数据。

总之，移动语义是现代 C++ 中提高性能的关键技术之一，特别是在处理大型数据和资源管理方面。

ref使用

1. 保持引用语义

在 C++ 中，通常参数和对象是通过值传递的，这意味着会发生数据复制。在处理大型数据结构或需要直接修改传入数据的场景中，按值传递可能不高效或不符合需求。`std::ref` 通过创建一个引用包装器，使得即使在按值传递的环境中，也能保持引用语义，从而避免不必要的复制并允许直接修改原始数据。

2. 线程函数参数传递

在多线程编程中，特别是使用 `std::thread` 时，如果你想让线程函数能够修改其主线程中的数据，直接传递引用是不行的，因为 `std::thread` 的构造函数默认以值传递方式接收参数。这时，使用 `std::ref` 可以将变量以引用方式传递给线程函数。

3. 使用标准库算法

大多数 C++ 标准库算法，例如 `std::for_each`、`std::transform` 等，都是以值传递方式接收参数。如果需要这些算法直接修改容器中的元素或传入的参数，可以使用 `std::ref` 来确保参数以引用方式传递，从而直接对原数据进行操作。

4. 与 `std::bind` 一起使用

在使用 `std::bind` 绑定函数调用时，如果希望绑定的参数保持引用形式而不是被复制，应使用 `std::ref`。这对于确保绑定后的函数可以修改其参数的原始数据非常重要。

示例代码

使用线程

```
#include <iostream>
#include <thread>

void add(int& number) {
    number += 5;
}

int main() {
    int x = 10;
    std::thread t(add, std::ref(x));
```

```
    t.join();
    std::cout << "x after thread: " << x << std::endl; // 输出: x after thread: 15
    return 0;
}
```

这个示例中，`std::ref` 确保线程函数 `add` 能够修改主函数中的变量 `x`。

使用标准库算法

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <functional>

void increment(int &n) {
    ++n;
}

int main() {
    std::vector<int> v = {1, 2, 3, 4};
    std::for_each(v.begin(), v.end(), std::ref(increment));  // 使用 std::ref
    for (auto i : v) {
        std::cout << i << " ";  // 输出: 2 3 4 5
    }
    return 0;
}
```

在这个例子中，`std::for_each` 使用 `std::ref` 保证 `increment` 函数能够修改向量 `v` 中的每个元素。

通过这些应用，我们可以看到 `std::ref` 在 C++ 中的实际用途和重要性，特别是在需要维护引用语义而又处于值传递上下文中的情况。

enable_shared_from_this（看项目代码）

enable_shared_from_this是一个模板类，定义于头文件，其原型为：

```
template< class T > class enable_shared_from_this;
```

std::enable_shared_from_this 能让一个对象（假设其名为 t，且已被一个 std::shared_ptr 对象 pt 管理）安全地生成其他额外的 std::shared_ptr 实例（假设名为 pt1, pt2, ... ），它们与 pt 共享对象 t 的所有权。

若一个类 T 继承 std::enable_shared_from_this，则会为该类 T 提供成员函数： shared_from_this 。当 T 类型对象 t 被一个为名为 pt 的 std::shared_ptr 类对象管理时，调用 T::shared_from_this 成员函数，将会返回一个新的 std::shared_ptr 对象，它与 pt 共享 t 的所有权。

一.使用场合

当类A被share_ptr管理，且在类A的成员函数里需要把当前类对象作为参数传给其他函数时，就需要传递一个指向自身的share_ptr。

1.为何不直接传递this指针

使用智能指针的初衷就是为了方便资源管理，如果在某些地方使用智能指针，某些地方使用原始指针，很容易破坏智能指针的语义，从而产生各种错误。

2.可以直接传递share_ptr么?

答案是不能，因为这样会造成2个非共享的share_ptr指向同一个对象，未增加引用计数导对象被析构两次

六、使用c语言时间标准库

```
#define _CRT_SECURE_NO_WARNINGS //VS中必须定义,否则报错
#include<ctime>
#include<stdio.h>
int main()
{
    time_t nowtime;
    time(&nowtime); //获取1970年1月1日0点0分0秒到现在经过的秒数
    tm* p = localtime(&nowtime); //将秒数转换为本地时间,年从1900算起,需要+1900,月为0-11,所以要+1
    printf("%04d:%02d:%02d %02d:%02d:%02d\n", p->tm_year + 1900, p->tm_mon + 1, p->tm_mday,p->tm_hour,p->tm_min,p->tm_sec);
}
```

或者使用安全函数不需要定义宏（localtime_s只有在windows下使用）

```
#include<ctime>
#include<stdio.h>
int main()
{
    time_t nowtime;
    time(&nowtime); //获取1970年1月1日0点0分0秒到现在经过的秒数
    tm p;
    localtime_s(&p,&nowtime); //将秒数转换为本地时间,年从1900算起,需要+1900,月为0-11,所以要+1
    printf("%04d:%02d:%02d %02d:%02d:%02d\n", p.tm_year + 1900, p.tm_mon + 1, p.tm_mday,p.tm_hour,p.tm_min,p.tm_sec);
}
```

1.time函数

函数原型:

```
time_t time(
time_t* _Time //保存时间的参数
)
```

可以看到,该函数只需要一个名为time_t的参数

```
#define long long time_t
```

从源代码可以看到,time_t实则是long long 类型的别名

该函数的作用就是将1970年1月1日0点0分0秒到当前所经过的秒数放在该参数中

所以想要得到我们想要的年月日时分秒常见格式,还需要将该结果转换一下
2.localtime函数

函数原型

```
tm* localtime(
time_t* _Time
)
```

该函数的作用就是将秒数转化为对应的年月日,时分秒

需要的参数正是time函数返回的结果,转化后的结果为tm结构体.就可以通过tm变量获取想要的内容

注意: 如果是在VS环境下,使用该函数会报错,必须定义宏_CRT_SECURE_NO_WARNINGS在最前面

```
#define _CRT_SECURE_NO_WARNINGS
```

3.tm结构体

源代码(中文为我的备注,可能理解有偏差,建议参照英文):

```
struct tm
{
    int tm_sec;   // seconds after the minute - [0, 60] including leap second //一分钟的第几秒,从0开始计数
    int tm_min;   // minutes after the hour - [0, 59] //一小时的第几分种,从0开始计数
    int tm_hour;  // hours since midnight - [0, 23] //一天的第几个小时,从0开始计数
    int tm_mday;  // day of the month - [1, 31] //一个月的第几天,从1开始计数
    int tm_mon;   // months since January - [0, 11] //一年的第几个月,从0开始计数,即0代表1月,1代表2月
    int tm_year;  // years since 1900 //从1900年到现在经过的年分
    int tm_wday;  // days since Sunday - [0, 6] //一个星期的第几天,从0开始计数,即0代表星期一,依次类推
    int tm_yday;  // days since January 1 - [0, 365] //一年的第几天,从0计数,0代表第一天,依次类推
    int tm_isdst; // daylight savings time flag //夏令时标志
};
```

4.localtime_s函数

该函数为对应的安全函数,不用定义宏就可以使用

```
errno_t localtime_s(
tm*  _Tm,
time_t * _Time
)
```

可以看到,该安全函数需要两个参数,一个是tm,相当于localtime函数的返回值,另一个是time_t,就是time函数得到的结果

5.localtime_r

localtime_r也是用来获取系统时间，运行于linux平台下
函数原型为struct tm *localtime_r(const time_t *timep, struct tm *result);

```
#include <stdio.h>
#include <time.h>
int main()
{
    time_t time_seconds = time(0);    /*或者写成  time_t time_seconds
                                                  time(time_seconds)*/
    struct tm now_time;
    localtime_r(&time_seconds, &now_time);
    printf("%d-%d-%d %d:%d:%d\n", now_time.tm_year + 1900, now_time.tm_mon + 1,
        now_time.tm_mday, now_time.tm_hour, now_time.tm_min, now_time.tm_sec);
}
```

C 库函数 - strftime()

C标准库<time.h>

描述

C 库函数 size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr) 根据 format 中定义的格式化规则，格式化结构 timeptr 表示的时间，并把它存储在 str 中。

声明

下面是 strftime() 函数的声明。

```
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm
*timeptr)
```

参数

- str -- 这是指向目标数组的指针，用来复制产生的 C 字符串。
- maxsize -- 这是被复制到 str 的最大字符数。
- format -- 这是 C 字符串，包含了普通字符和特殊格式说明符的任何组合。这些格式说明符由函数替换为表示 tm 中所指定时间的相对应值。格式说明符是：

说明符	替换为	实例
%a	缩写的星期几名称	Sun

说明符	替换为	实例
%A	完整的星期几名称	Sunday
%b	缩写的月份名称	Mar
%B	完整的月份名称	March
%c	日期和时间表示法	Sun Aug 19 02:56:02 2012
%d	一月中的第几天（01-31）	19
%H	24 小时格式的小时（00-23）	14
%I	12 小时格式的小时（01-12）	05
%j	一年中的第几天（001-366）	231
%m	十进制数表示的月份（01-12）	08
%M	分（00-59）	55
%p	AM 或 PM 名称	PM
%S	秒（00-61）	02
%U	一年中的第几周，以第一个星期日作为第一周的第一天（00-53）	33
%w	十进制数表示的星期几，星期日表示为 0（0-6）	4
%W	一年中的第几周，以第一个星期一作为第一周的第一天（00-53）	34
%x	日期表示法	08/19/12
%X	时间表示法	02:50:06
%y	年份，最后两个数字（00-99）	01
%Y	年份	2012
%Z	时区的名称或缩写	CDT
%%	一个 % 符号	%

- timeptr -- 这是指向 tm 结构的指针，该结构包含了一个呗分解为以下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59             */
   int tm_min;         /* 分，范围从 0 到 59             */
   int tm_hour;        /* 小时，范围从 0 到 23            */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31   */
   int tm_mon;         /* 月份，范围从 0 到 11            */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6     */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365   */
   int tm_isdst;       /* 夏令时                          */
};
```

返回值

如果产生的 C 字符串小于 size 个字符（包括空结束字符），则会返回复制到 str 中的字符总数（不包括空结束字符），否则返回零。

实例

下面的实例演示了 strftime() 函数的用法。

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

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );

   strftime(buffer, 80, "%Y-%m-%d %H:%M:%S", info);
   printf("格式化的日期 & 时间 : |%s|\n", buffer );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

格式化的日期 & 时间 : |2018-09-19 08:59:07|

七、c++17 Any

```
在 `any` 类的实现中，`base_` 不能写成 `std::unique_ptr<Derive<T>>` 是因为 `any` 类的设计初衷是为了处理**任意类型**的对象，而不仅仅是特定模板类型 `T` 的派生类 `Derive<T>`。以下是详细原因：

### 1. **类型擦除机制的核心：**
`any` 类的核心是类型擦除，旨在存储和操作任意类型的对象而不关心它们的具体类型。在这个设计中，基类 `Base` 提供了一个抽象接口，而不同类型的对象通过各自的派生类（比如 `Derive<T>`，其中 `T` 是实际存储的类型）具体实现。当 `any` 需要处理不同类型的对象时，`std::unique_ptr<Base>` 可以指向任何派生自 `Base` 的对象，保持类型的灵活性。
```

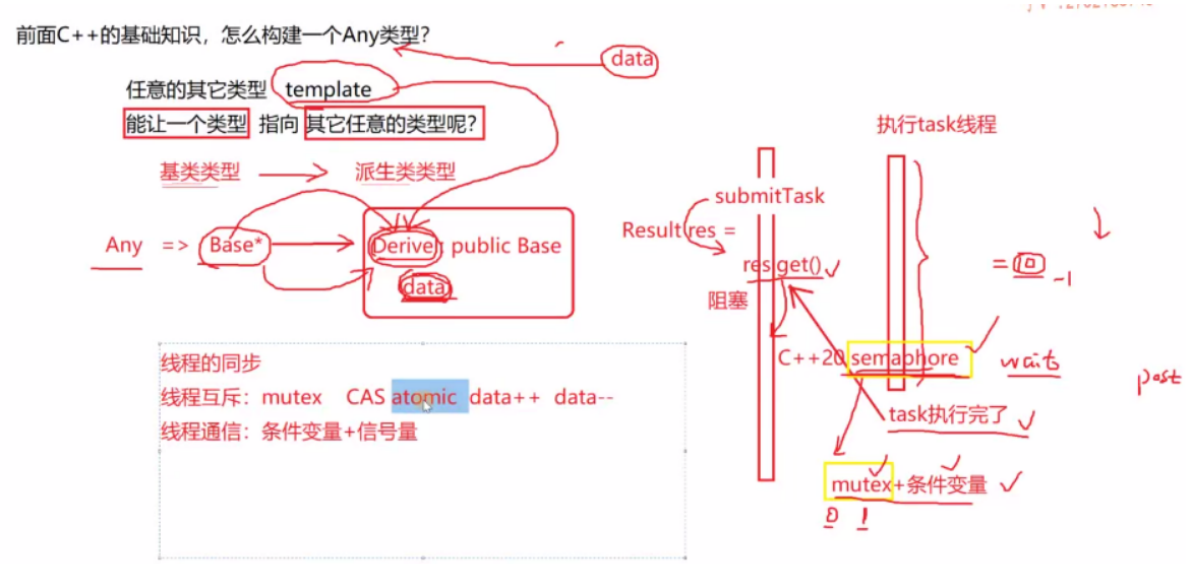
如果将 `base_` 改为 `std::unique_ptr<Derive<T>>`，你就只能存储 `T` 类型的对象，这违背了 `any` 类存储和处理任意类型的设计目标。

2. **模板类型 `T` 的限制：**
`std::unique_ptr<Derive<T>>` 会让 `base_` 绑定到具体类型 `T`。在 `any` 类中，你希望它能够容纳不同类型的对象。如果用 `Derive<T>`，这意味着你必须为每种类型显式地指定一个模板类型 `T`，而不是像类型擦除那样通用地管理所有类型。

举例：
- `std::unique_ptr<Base>` 可以存储派生自 `Base` 的任何类型，比如 `Derive<int>`，`Derive<std::string>`，等等。
- `std::unique_ptr<Derive<T>>` 则只能存储 `Derive<T>`，即只能存储一种具体的类型 `T`，导致泛化能力丧失。

3. **运行时 vs 编译时的类型处理：**
使用 `std::unique_ptr<Derive<T>>` 会在编译时固定类型 `T`，这意味着每次使用 `any` 时都需要为特定类型的 `Derive<T>` 重新实例化，这与类型擦除的运行时灵活性相违背。通过 `std::unique_ptr<Base>`，你可以在运行时存储和管理任何派生类型，而不必在编译时明确指定。

总结：
`any` 类的设计初衷是为任意类型的对象提供存储和操作能力，使用 `std::unique_ptr<Base>` 能够实现这种灵活性。相反，`std::unique_ptr<Derive<T>>` 会将 `any` 限制在某一个具体类型 `T` 上，无法实现类型擦除的目标。因此，`base_` 使用 `std::unique_ptr<Base>` 才能够让 `any` 类处理任意类型的对象。



八、packaged_task与future

`std::packaged_task` 是 C++ 标准库中提供的一种工具，用于将一个可调用对象（函数、lambda 表达式、函数对象等）包装为一个异步任务，允许将该任务的执行与获取结果分离。其典型用法包括将任务提交给线程池或延迟执行某个任务。

`std::packaged_task` 的核心是为一个可调用对象的结果创建一个 `std::future`，并通过这个 `future` 在任务完成后获取返回值。

1. `std::packaged_task` 的定义

`std::packaged_task` 是一个模板类，定义如下：

```
template <class T>
class packaged_task;
```

其中，`T` 是被包装的可调用对象的类型，它定义了该任务的签名，比如 `int(int, int)` 代表该任务接受两个 `int` 参数并返回一个 `int`。

2. 使用步骤

使用 `std::packaged_task` 通常可以分为以下几个步骤：

1. **创建 `std::packaged_task` 实例并绑定一个可调用对象。**
2. **获取与该任务关联的 `std::future` 对象。**
3. **启动任务（通常由线程或线程池异步执行）。**
4. **通过 `std::future` 获取任务的结果。**

3. 简单例子

基本使用：

下面的例子展示了如何使用 `std::packaged_task` 创建异步任务并获取结果：

```
#include <iostream>
#include <future>
#include <thread>
#include <functional>

// 定义一个简单的函数
int add(int a, int b) {
    return a + b;
}

int main() {
    // 1. 创建一个 std::packaged_task 实例，绑定可调用对象
    std::packaged_task<int(int, int)> task(add);

    // 2. 获取与该任务关联的 std::future
    std::future<int> result = task.get_future();

    // 3. 启动任务（使用线程异步执行）
    std::thread t(std::move(task), 2, 3);  // 传入参数 2 和 3

    // 4. 获取任务的结果
    std::cout << "Result: " << result.get() << std::endl;

    // 等待线程执行完毕
    t.join();

    return 0;
```

```
}
```

解释：

- `std::packaged_task<int(int, int)> task(add);`：创建一个 `packaged_task`，该任务包装了 `add` 函数，该函数接收两个 `int` 参数并返回一个 `int`。
- `task.get_future()`：获取与 `packaged_task` 关联的 `std::future` 对象，`std::future` 可以用于在未来获取异步任务的结果。
- `std::thread t(std::move(task), 2, 3);`：启动一个线程来执行 `task`，并传递参数 `2` 和 `3`。注意这里需要使用 `std::move`，因为 `std::packaged_task` 不允许复制，但可以移动。
- `result.get()`：阻塞等待任务完成，并获取任务的返回结果。

4. 与 `std::thread` 结合使用

`std::packaged_task` 的常见用法是将它与 `std::thread` 结合，来实现简单的异步任务处理。上面的例子已经展示了这种用法。

你也可以不使用 `std::thread`，而是手动调用任务对象。例如：

```
#include <iostream>
#include <future>
#include <functional>

int multiply(int x, int y) {
    return x * y;
}

int main() {
    // 创建 std::packaged_task 并绑定 multiply 函数
    std::packaged_task<int(int, int)> task(multiply);

    // 获取关联的 future
    std::future<int> result = task.get_future();

    // 手动调用 task（同步执行）
    task(4, 5);

    // 获取并输出结果
    std::cout << "Result: " << result.get() << std::endl;

    return 0;
}
```

在这个例子中，`task(4, 5);` 直接在主线程中调用任务，而不是通过线程异步执行。

5. 与 `std::async` 和 `std::promise` 的比较

- `std::packaged_task`：
  - 将可调用对象包装成一个任务，并提供 `std::future` 用来获取任务结果。
  - 需要手动启动任务（例如通过线程执行）。
- `std::async`：

- 提供一个更高级的接口来启动异步任务，并直接返回 `std::future`。
- 自动管理线程的创建和任务执行，不需要手动启动任务。

```
auto fut = std::async(std::launch::async, add, 2, 3);
```

- **`std::promise`**：
  - 提供一个更低级的接口，允许手动设置任务结果，可以和 `std::future` 配合使用。
  - 你可以在任何地方设置结果，并不强制和某个函数绑定。

6. 常见应用场景

- **线程池**：`std::packaged_task` 可以与线程池一起使用，允许将任务提交给线程池，然后通过 `std::future` 来获取结果。
- **异步计算**：它允许异步计算任务，尤其是在函数调用和结果获取之间存在延迟的场景。
- **任务调度**：任务可以在以后调用，也可以调度到其他线程执行。

7. 错误处理

`std::packaged_task` 的 `std::future` 也支持异常传递。如果任务在执行过程中抛出异常，`std::future::get` 会捕获该异常并重新抛出。

```
#include <iostream>
#include <future>
#include <exception>
#include <stdexcept>
#include <thread>

int divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero!");
    }
    return a / b;
}

int main() {
    std::packaged_task<int(int, int)> task(divide);
    std::future<int> result = task.get_future();

    std::thread t(std::move(task), 10, 0); // b = 0, will throw exception

    try {
        int res = result.get();  // Will throw
        std::cout << "Result: " << res << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
    }

    t.join();

    return 0;
}
```

输出：

```
Caught exception: Division by zero!
```

在这个例子中，任务抛出的异常会通过 `std::future::get` 被捕获并重新抛出。

总结

`std::packaged_task` 允许将可调用对象打包为一个任务，并通过 `std::future` 获取其结果。它与 `std::thread` 结合使用，允许在不同线程中异步执行任务，也可以手动执行。`std::packaged_task` 提供了灵活的任务管理方式，并在任务执行完毕后提供结果处理和异常处理的功能。

其他小记

一、c++元组 tuple 和 pair

二、宏写法

三、#include

```
m_filestream << m_formatter->format(logger,level,event);       //这种用法学习
```

四、c_str

在 C++ 中，`c_str()` 是 `std::string` 类的一个成员函数，用于返回一个指向以空字符终止的字符串的指针，这个字符串包含和 `std::string` 相同的数据。这是非常有用的，尤其是在需要将 C++ 字符串与那些期望以 C 风格字符串（即 `const char*` 类型）作为输入的 C API 或库进行互操作时。

用法和功能

```
std::string s = "Hello, world!";
const char* p = s.c_str();
```

在上述代码中，`s.c_str()` 返回指向字符串 "Hello, world!" 的常量字符指针。这是安全的，因为返回的指针指向 `std::string` 内部管理的内存，而这块内存会在 `std::string` 对象 `s` 生命周期结束时被自动释放。

注意事项

- **内存管理**：`c_str()` 返回的指针应当只在它的源 `std::string` 对象存活期间内使用。如果 `std::string` 对象被销毁或修改（如通过赋值、添加字符等），返回的指针可能会变得无效。
- **线程安全**：虽然 `std::string` 本身不是线程安全的，但只要没有同时修改字符串，多个线程读取 `c_str()` 返回的指针是安全的。
- **使用限制**：返回的指针是只读的，尝试修改通过 `c_str()` 获取的指针所指向的内容将导致未定义行为。

实际应用

`c_str()` 在与老旧的 C 库或 API 交互时尤其重要。许多 C 函数库，如标准 I/O 库（`printf`、`fopen` 等），都需要 `const char*` 类型的参数。`std::string` 的 `c_str()` 方法提供了一种简单的方式，将 C++ 字符串转换成 C 字符串，从而可以被这些函数接受。

```
std::string filename = "example.txt";
FILE* file = fopen(filename.c_str(), "r");
if (file != nullptr) {
    // 文件操作
    fclose(file);
} else {
    std::cerr << "文件打开失败\n";
}
```

在这个例子中，我们使用 `filename.c_str()` 将 `std::string` 对象转换为 C 风格字符串，以便 `fopen` 函数可以接受它作为参数。

总结，`c_str()` 是 `std::string` 提供的一个非常实用的功能，它允许 C++ 程序与需要传统 C 风格字符串的函数或库无缝集成。

五、

C++析构函数为什么要为虚函数?

A:基类指针可以指向派生类的对象（多态性），如果删除该指针delete []p；就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全,形成了删除一半形象,造成内存泄漏,。所以，将析构函数声明为虚函数是十分必要的。

项目代码

```
class LogLevel{
        //日志级别
    public:
        enum Level{
            UNKNOW = 0,
            DEBUG = 1,
            INFO = 2,
            WARN = 3,
            ERROR = 4,
            FATAL = 5
        };

        static const char* ToString(LogLevel::Level level);  //用static的作用
};
```

在 C++ 中，将类成员函数声明为 `static` 的做法是有其特定的用途和优点的。对于 `LogLevel` 类中的 `ToString` 方法来说，使用 `static` 关键字确实是有必要的。下面是使用 `static` 的几个关键理由：

1. **无需实例化**：`static` 成员函数不依赖于类的实例。这意味着你可以在不创建类对象的情况下调用该函数。例如，你可以直接通过 `LogLevel::ToString(LogLevel::DEBUG)` 来获取日志级别的字符串表示，而无需首先创建一个 `LogLevel` 对象。这在日志系统中特别有用，因为通常你需要在很多地方，包括一些可能还没有具体日志对象的上下文中，将日志级别转换为字符串。

2. **全局可访问性**：由于 `static` 函数不需要类的实例，它们可以被看作是与类关联的全局函数。这使得它们在处理不需要类实例状态的功能时非常方便和有效。对于 `ToString` 方法来说，它仅仅根据输入的枚举值返回对应的字符串，不需要访问或修改任何实例变量。

3. **资源共享**：在某些情况下，`static` 方法可以用来管理共享资源或执行对所有实例都相同的操作。尽管在 `ToString` 的例子中不直接涉及资源管理，但这一点在处理类似的静态数据时很重要。

4. **效率和简洁性**：使用 `static` 方法通常可以提高代码的效率和简洁性。因为不需要创建对象，所以可以减少内存和处理时间的开销。同时，`static` 方法可以直接通过类名调用，这使得代码更简洁，调用方式也更明确。

综上所述，为 `LogLevel` 类中的 `ToString` 方法加上 `static` 是合理的，因为这个方法的功能仅仅是将枚举值转换为对应的字符串，不需要访问或依赖类的实例状态。这样的设计使得方法的使用更为灵活和高效。

```
class Logger : public std::enable_shared_from_this<Logger>
{//////////////////////
public:
    typedef std::shared_ptr<Logger> ptr;

    Logger(const std::string& name = "root");
    void log(LogLevel::Level level, LogEvent::ptr event);

    void debug(LogEvent::ptr event);
    void info(LogEvent::ptr event);
    void warn(LogEvent::ptr event);
    void error(LogEvent::ptr event);
    void fatal(LogEvent::ptr event);

    void addAppender(LogAppender::ptr appender);
    void delAppender(LogAppender::ptr appender);

    LogLevel::Level getLevel()const {return m_level;}  //const用法   表示这些函数不会修改成员变量的状态
    void setLevel(LogLevel::Level val){m_level = val;}

    const std::string& getName() const {return m_name;}
private:
    std::string m_name;         //日志名称
    LogLevel::Level  m_level;   //日志级别
    std::list<LogAppender::ptr> m_Appenders; //Appender集合
    LogFormatter::ptr m_formatter;
    };


void Logger::log(LogLevel::Level level, LogEvent::ptr event){
    if(level>=m_level){
        auto self = shared_from_this();
        for(auto&  i : m_Appenders ){           // &使用引用   //循环遍历m_Appenders的用法
            /*
             for (declaration : expression) {}
            declaration: 用于声明一个变量，用来引用容器中的每个元素。可以使用自动类型推导（auto）或显式类型声明。
            expression: 一个返回可遍历的范围或容器的表达式。
             */
            i->log(self,level,event);
        }
    }
```

```
}
```