# shared_ptr

## 构造

1、**普通构造函数**

测试代码如下:

```C++
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    // 使用智能指针管理一块 int 型的堆内存
    shared_ptr<int> ptr1(new int(520));
    cout << "ptr1管理的内存引用计数: " << ptr1.use_count() << endl;
    // 使用智能指针管理一块字符数组对应的堆内存
    shared_ptr<char> ptr2(new char[12]);
    cout << "ptr2管理的内存引用计数: " << ptr2.use_count() << endl;
    // 创建智能指针对象, 不管理任何内存
    shared_ptr<int> ptr3;
    cout << "ptr3管理的内存引用计数: " << ptr3.use_count() << endl;
    // 创建智能指针对象, 初始化为空
    shared_ptr<int> ptr4(nullptr);
    cout << "ptr4管理的内存引用计数: " << ptr4.use_count() << endl;

    return 0;
}
```

测试代码输出的结果如下:

```C++
ptr1管理的内存引用计数: 1
ptr2管理的内存引用计数: 1
ptr3管理的内存引用计数: 0
ptr4管理的内存引用计数: 0
```

2、**拷贝构造** 拷贝构造 无论显示隐式，都创建了一个对象 但是 移动构造并没有创建新对象

### 1.2 通过拷贝和移动构造函数初始化

当一个智能指针被初始化之后，就可以通过这个智能指针初始化其他新对象。在创建新对象的时候，对应的拷贝构造函数或者移动构造函数就被自动调用了。

C++

```cpp
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    // 使用智能指针管理一块 int 型的堆内存，内部引用计数为 1
    shared_ptr<int> ptr1(new int(520));
    cout << "ptr1管理的内存引用计数: " << ptr1.use_count() << endl;
    //调用拷贝构造函数
    shared_ptr<int> ptr2(ptr1);
    cout << "ptr2管理的内存引用计数: " << ptr2.use_count() << endl;
    shared_ptr<int> ptr3 = ptr1;
    cout << "ptr3管理的内存引用计数: " << ptr3.use_count() << endl;
    //调用移动构造函数
    shared_ptr<int> ptr4(std::move(ptr1));
    cout << "ptr4管理的内存引用计数: " << ptr4.use_count() << endl;
    std::shared_ptr<int> ptr5 = std::move(ptr2);
    cout << "ptr5管理的内存引用计数: " << ptr5.use_count() << endl;

    return 0;
}
```

测试程序输入的结果:

C++

```
ptr1管理的内存引用计数: 1
ptr2管理的内存引用计数: 2
ptr3管理的内存引用计数: 3
ptr4管理的内存引用计数: 3
ptr5管理的内存引用计数: 3
```

如果使用拷贝的方式初始化共享智能指针对象，这两个对象会同时管理同一块堆内存，堆内存对应的引用计数也会增加；如果使用移动的方式初始智能指针对象，只是转让了内存的所有权，管理内存的对象并不会增加，因此内存的引用计数不会变化。

3、**make_shared** 创建对象的同时，就初始化给智能指针 （智能指针 本质还是给你一个对象，并且可以在对象不再使用的时候，将其delete）

## 1.3 通过std::make_shared初始化

通过C++提供的std::make_shared() 就可以完成内存对象的创建并将其初始化给智能指针，函数原型如下：

C++

```
template< class T, class... Args >
shared_ptr<T> make_shared( Args&&... args );
```

- `T` ：模板参数的数据类型
- `Args&&... args` ：要初始化的数据，如果是通过make_shared创建对象，需按照构造函数的参数列表指定

```
class Test
{
public:
    Test()
    {
        cout << "construct Test..." << endl;
    }
    Test(int x)
    {
        cout << "construct Test, x = " << x << endl;
    }
    Test(string str)
    {
        cout << "construct Test, str = " << str << endl;
    }
    ~Test()
    {
        cout << "destruct Test ..." << endl;
    }
};

int main()
{
    // 使用智能指针管理一块 int 型的堆内存，内部引用计数为 1
    shared_ptr<int> ptr1 = make_shared<int>(520);
    cout << "ptr1管理的内存引用计数: " << ptr1.use_count() << endl;

    shared_ptr<Test> ptr2 = make_shared<Test>();
    cout << "ptr2管理的内存引用计数: " << ptr2.use_count() << endl;

    shared_ptr<Test> ptr3 = make_shared<Test>(520);
    cout << "ptr3管理的内存引用计数: " << ptr3.use_count() << endl;

    shared_ptr<Test> ptr4 = make_shared<Test>("我是要成为海贼王的男人!!!");
    cout << "ptr4管理的内存引用计数: " << ptr4.use_count() << endl;
    return 0;
}
```

使用std::make_shared()模板函数可以完成内存地址的创建，并将最终得到的内存地址传递给共享智能指针对象管理。如果申请的内存是普通类型，通过函数的（）可完成地址的初始化，如果要创建一个类对象，函数的（）内部需要指定构造对象需要的参数，也就是类构造函数的参数。

4、**reset初始化**

示例：

测试代码如下：

```C++
#include <iostream>
#include <string>
#include <memory>
using namespace std;

int main()
{
    // 使用智能指针管理一块 int 型的堆内存，内部引用计数为 1
    shared_ptr<int> ptr1 = make_shared<int>(520);
    shared_ptr<int> ptr2 = ptr1;
    shared_ptr<int> ptr3 = ptr1;
    shared_ptr<int> ptr4 = ptr1;
    cout << "ptr1管理的内存引用计数: " << ptr1.use_count() << endl;
    cout << "ptr2管理的内存引用计数: " << ptr2.use_count() << endl;
    cout << "ptr3管理的内存引用计数: " << ptr3.use_count() << endl;
    cout << "ptr4管理的内存引用计数: " << ptr4.use_count() << endl;

    ptr4.reset();
    cout << "ptr1管理的内存引用计数: " << ptr1.use_count() << endl;
    cout << "ptr2管理的内存引用计数: " << ptr2.use_count() << endl;
    cout << "ptr3管理的内存引用计数: " << ptr3.use_count() << endl;
    cout << "ptr4管理的内存引用计数: " << ptr4.use_count() << endl;

    shared_ptr<int> ptr5;
    ptr5.reset(new int(250));
    cout << "ptr5管理的内存引用计数: " << ptr5.use_count() << endl;

    return 0;
}
```

测试代码输入的结果:

测试代码输入的结果:

```C++
ptr1管理的内存引用计数: 4
ptr2管理的内存引用计数: 4
ptr3管理的内存引用计数: 4
ptr4管理的内存引用计数: 4

ptr1管理的内存引用计数: 3
ptr2管理的内存引用计数: 3
ptr3管理的内存引用计数: 3
ptr4管理的内存引用计数: 0

ptr5管理的内存引用计数: 1
```

对于一个未初始化的共享智能指针，可以通过reset方法来初始化，当智能指针中有值的时候，调用reset会使引用计数减1。

可用于ptr初始化：对于原先指向的对象内容（若有），其引用--；然后 ptr获得新的 对象的引用，并使得此对象引用+1;如果 reset没有传入参数（没有要剥夺的对象的指针引用，则引用为0）

5、**获取原始指针**

### 1.5 获取原始指针

通过智能指针可以管理一个普通变量或者对象的地址，此时原始地址就不可见了。当我们想要修改变量或者对象中的值的时候，就需要从智能指针对象中先取出数据的原始内存的地址再操作，解决方案是调用共享智能指针类提供的 `get()` 方法，其函数原型如下：

C++

```
T* get() const noexcept;
```

测试代码如下：

C++

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

int main()
{
    int len = 128;
    shared_ptr<char> ptr(new char[len]);
    // 得到指针的原始地址
    char* add = ptr.get();
    memset(add, 0, len);
    strcpy(add, "我是要成为海贼王的男人!!!");
    cout << "string: " << add << endl;

    shared_ptr<int> p(new int);
    *p = 100;
    cout << *p.get() << "  " << *p << endl;

    return 0;
}
```

```
shared_ptr<char> ptr6(new char[aaaaa]);
cout << "ptr6管理的内存引用计数: " << ptr6.use_count() << endl;
*ptr6 = 's';
char* ptr6_c=ptr6.get();
ptr6_c = (char*)"Hai zei wang de  Man";
cout << *ptr6 << endl;
cout << ptr6_c << endl;
```

```
ptr3管理的内存引用计数: 4
ptr4管理的内存引用计数: 4
ptr1管理的内存引用计数: 3
ptr2管理的内存引用计数: 3
ptr3管理的内存引用计数: 3
ptr4管理的内存引用计数: 0
ptr5管理的内存引用计数: 1
ptr6管理的内存引用计数: 1
s
Hai zei wang de  Man
```

# 删除

1、指定删除器

当智能指针管理的内存对应的引用计数变为0的时候，这块内存就会被智能指针析构掉了。另外，我们在初始化智能指针的时候也可以自己指定删除动作，这个删除操作对应的函数被称之为删除器，这个删除器函数本质是一个回调函数，我们只需要进行实现，其调用是由智能指针完成的。

```C++
#include <iostream>
#include <memory>
using namespace std;

// 自定义删除器函数，释放int型内存
void deleteIntPtr(int* p)
{
    delete p;
    cout << "int 型内存被释放了...";
}

int main()
{
    shared_ptr<int> ptr(new int(250), deleteIntPtr);
    return 0;
}
```

删除器函数也可以是lambda表达式，因此代码也可以写成下面这样：

```C++
int main()
{
    shared_ptr<int> ptr(new int(250), [](int* p) {delete p; });
    return 0;
}
```

在上面的代码中， **lambda表达式的参数就是智能指针管理的内存的地址，有了这个地址之后函数体内部就可以完成删除操作了。**

在C++11中使用shared_ptr管理动态数组时，需要指定删除器，因为 **std::shared_ptr的默认删除器不支持数组对象** ，具体的处理代码如下：

数组对象 一定要有自定义删除器（函数对象）

在C++11中使用shared_ptr管理动态数组时，需要指定删除器，因为 **std::shared_ptr的默认删除器不支持数组对象** ，具体的处理代码如下：

```C++
int main()
{
    shared_ptr<int> ptr(new int[10], [](int* p) {delete[]p; });
    return 0;
}
```

在删除数组内存时，除了自己编写删除器，也可以使用C++提供的 **std::default_delete<T>()** 函数作为删除器，这个函数内部的删除功能也是通过调用 **delete** 来实现的，要释放什么类型的内存就将模板类型T指定为什么类型即可。具体处理代码如下：

在删除数组内存时，除了自己编写删除器，也可以使用C++提供的 `std::default_delete<T>()` 函数作为删除器，这个函数内部的删除功能也是通过调用 `delete` 来实现的，要释放什么类型的内存就将模板类型T指定为什么类型即可。具体处理代码如下：

```C++
int main()
{
    shared_ptr<int> ptr(new int[10], default_delete<int[]>());
    return 0;
}
```

自定义 shared_ptr_array 嵌套 shared_ptr 返回的对象默认设置好：可删除数组的删除器 delete[]或者 设置 default_delete<T[]>() （原先的 default_delete<T>() 中的 模板T设置成 T[]）

另外，我们还可以自己封装一个make_shared_array方法来让shared_ptr支持数组，代码如下：

```C++
#include <iostream>
#include <memory>
using namespace std;

template <typename T>
shared_ptr<T> make_share_array(size_t size)
{
    // 返回匿名对象
    return shared_ptr<T>(new T[size], default_delete<T[]>());
}

int main()
{
    shared_ptr<int> ptr1 = make_share_array<int>(10);
    cout << ptr1.use_count() << endl;
    shared_ptr<char> ptr2 = make_share_array<char>(128);
    cout << ptr2.use_count() << endl;
    return 0;
}
```

# 独占式指针

unique_ptr<int>uptr1 不允许拷贝构造，但是可以 通过函数返回 和 转移move赋值

std::unique_ptr是一个独占型的智能指针，它不允许其他的智能指针共享其内部的指针，可以通过它的构造函数初始化一个独占智能指针对象，但是不允许通过赋值将一个unique_ptr赋值给另一个unique_ptr。

```C++
// 通过构造函数初始化对象
unique_ptr<int> ptr1(new int(10));
// error, 不允许将一个unique_ptr赋值给另一个unique_ptr
unique_ptr<int> ptr2 = ptr1;
```

std::unique_ptr不允许复制，但是可以通过函数返回给其他的std::unique_ptr，还可以通过std::move来转译给其他的std::unique_ptr，这样原始指针的所有权就被转移了，这个原始指针还是被独占的。

```C++
#include <iostream>
#include <memory>
using namespace std;

unique_ptr<int> func()
{
    return unique_ptr<int>(new int(520));
}

int main()
{
    // 通过构造函数初始化
    unique_ptr<int> ptr1(new int(10));
    // 通过转移所有权的方式初始化
    unique_ptr<int> ptr2 = move(ptr1);
    unique_ptr<int> ptr3 = func();

    return 0;
}
```

## reset 解除管理 或 初始化

unique_ptr独占智能指针类也有一个reset方法，函数原型如下：

```C++
void reset( pointer ptr = pointer() ) noexcept;
```

使用reset方法可以让unique_ptr解除对原始内存的管理，也可以用来初始化一个独占的智能指针。

```C++
int main()
{
    unique_ptr<int> ptr1(new int(10));
    unique_ptr<int> ptr2 = move(ptr1);

    ptr1.reset();
    ptr2.reset(new int(250));

    return 0;
}
```

- `ptr1.reset();` 解除对原始内存的管理
- `ptr2.reset(new int(250));` 重新指定智能指针管理的原始内存

## get 获取被管理的地址

如果想要获取独占智能指针管理的原始地址，可以调用get()方法，函数原型如下：

```C++
pointer get() const noexcept;
```

```C++
int main()
{
    unique_ptr<int> ptr1(new int(10));
    unique_ptr<int> ptr2 = move(ptr1);

    ptr2.reset(new int(250));
    cout << *ptr2.get() << endl;        // 得到内存地址中存储的实际数值 250

    return 0;
}
```

# 指定删除器时要指定函数类型 （返回值 参数）

## 2. 删除器

unique_ptr指定删除器和shared_ptr指定删除器是有区别的，unique_ptr指定删除器的时候需要确定删除器的类型，所以不能像shared_ptr那样直接指定删除器，举例说明：

```C++
shared_ptr<int> ptr1(new int(10), [](int*p) {delete p; });      // ok
unique_ptr<int> ptr1(new int(10), [](int*p) {delete p; });      // error

int main()
{
    using func_ptr = void(*)(int*);
    unique_ptr<int, func_ptr> ptr1(new int(10), [](int*p) {delete p; });

    return 0;
}
```

在上面的代码中第7行， `func_ptr` 的类型和 `lambda表达式` 的类型是一致的。在lambda表达式没有捕获任何变量的情况下是正确的，如果捕获了变量，编译时则会报错：

```C++
int main()
{
    using func_ptr = void(*)(int*);
    unique_ptr<int, func_ptr> ptr1(new int(10), [&](int*p) {delete p; });       // error
    return 0;
}
```

上面的代码中错误原因是这样的，在lambda表达式没有捕获任何外部变量时，可以直接转换为函数指针，一旦捕获了就无法转换了，如果想要让编译器成功通过编译，那么需要使用可调用对象包装器来处理声明的函数指针：


function<Return(Arg...)> 包装器：返回值类型(形参...)

## 二.function包装器

### 【1】function基本语法一览

```
std::function在头文件<functional>
// 类模板原型如下
template <class T> function;     // undefined
template <class Ret, class... Args>
class function<Ret(Args...)>;
模板参数说明：
Ret: 被调用函数的返回类型
Args…: 被调用函数的形参
```

### 【2】function解决可调用对象的类型问题——>把可调用对象包装器来，存放到数组中去

- function包装器 也叫作 **适配器**
- C++中的function本质是一个 **类模板**
- 在以往的学习中，面对不同的可调用对象，我们希望能把他们放到一个vector中方便调用，但是 类型不同显然做不到
- 而function包装器就恰好解决了这个问题（可调用对象的类型问题）

- 如在下面代码中，第一部分 `ret = func(x);`（可能是函数名？函数指针？函数对象(仿函数对象)？也有可能是lamber表达式对象）
- 我们 通过function语法即可成功把他们放到vector中

放到同一容器：前提是 写包装器时，返回对象和形参一致

```
int main()
{
    // 函数指针
    cout << useF(f, 11.11) << endl;

    // 函数对象
    cout << useF(Functor(), 11.11) << endl;

    // lambda表达式
    cout << useF([](double d)->double { return d / 4; }, 11.11) << endl;

    // 可调用对象存储到容器中
    //vector<>

    // 包装器 -- 可调用对象的类型问题
    //function<返回值类型(参数类型)>
    function<double(double)> f1 = f;// 函数名
    function<double(double)> f2 = [](double d)->double { return d / 4; };// 函数对象
    function<double(double)> f3 = Functor();// lamber表达式

    //vector<function<double(double)>> v = { f1, f2, f3 };//写法一
    //我们 通过function语法即可成功把他们放到vector中
    vector<function<double(double)>> v = { f, [](double d)->double { return d / 4; }, Functor() };//写法二

    double n = 3.3;
    for (auto f : v)
    {
        cout << f(n++) << endl;//遍历vector，每个元素是一个包装器
    }

    return 0;
}
```

## 三.包装器，解决模板的效率低下，同一函数模板实例化多份的问题

- 我们观察下面代码
- count 是一个静态局部变量，它确实存储在静态存储区域。
- 静态局部变量在程序生命周期内只被初始化一次，然后保留其值直到程序结束。因此，从理论上讲，count 应该在整个程序运行过程中保持唯一的值。然而，我们在 main 函数中使用了三个不同的函数对象（函数名、函数对象和 lambda 表达式），每个都调用了 useF 函数，**实例化了三份useF函数**，因此count值不会增加，还是1；

```cpp
template<class F, class T>
T useF(F f, T x)
{
    static int count = 0;
    cout << "count:" << ++count << endl;
    cout << "count:" << &count << endl;
    return f(x);
}
double f(double i)
{
    return i / 2;
}
struct Functor
{
    double operator()(double d)
    {
        return d / 3;
    }
};
int main()
{
    // 函数名
    cout << useF(f, 11.11) << endl;
    // 函数对象
    cout << useF(Functor(), 11.11) << endl;
    // lamber表达式
    cout << useF([](double d)->double { return d / 4; }, 11.11) << endl;
    return 0;
}
```

```
Microsoft Visual Studio 调试控
count:1
count:006EF4F0
5.555
count:1
count:006EF500
3.70333
count:1
count:006EF504
2.7775

C:\Users\86150\source\repos\2023.12.23\Debug\
按任意键关闭此窗口. . .
```

**包装器调用函数，只实例化一份模板函数对象**

- 经过包装器包装后，我们再来看这段代码：
- 我们发现，useF函数 **只被实例化成了一份**

```cpp
using namespace std;
template<class F, class T>
T useF(F f, T x)
{
    static int count = 0;
    cout << "count:" << ++count << endl;
    cout << "count:" << &count << endl;
    return f(x);
}
double f(double i)
{
    return i / 2;
}
struct Functor
{
    double operator()(double d)
    {
        return d / 3;
    }
};
int main()
{
    // 函数名
    std::function<double(double)> func1 = f;
    cout << useF(func1, 11.11) << endl;
    //函数对象
    std::function<double(double)> func2 = Functor();
    cout << useF(func2, 11.11) << endl;
    // lamber表达式
    std::function<double(double)> func3 = [](double d)->double { return d / 4; };
    cout << useF(func3, 11.11) << endl;
    return 0;
}
```

```
Microsoft Visual Studio 调试
count:1
count:0067F4F0
5.555
count:2
count:0067F4F0
3.70333
count:3
count:0067F4F0
2.7775

C:\Users\86150\source\repos\2023.12.23\Debug\
按任意键关闭此窗口. . .
```

地址相同

奥 其实，是模板函数在被调用时，会按照参数的不同，示例出不同的对象：

**对于模板函数 testTemp：传入参数不同 示例函数对象不同**

```cpp
template<typename T>
void testTemp(T t) {
    static int Count = 0;

    cout << "Count:" << Count++ << endl;

    return;
}
```

**两种类型的 t传入，会示例化出两种类型的 模板函数**

```
int Testtem = 0;
int Testa = 10;
testTemp(10);
testTemp("car");
testTemp("cal2");
testTemp(Testa);
```

**输出结果如下：**

```
Count:0
Count:0
Count:1
Count:1
```

而包装器会使得参数都统一成 包装器 类对象，所以只实例化一份模板函数对象

- 经过包装器包装后，我们再来看这段代码：
- 我们发现，useF函数 **只被实例化成了一份**

```
using namespace std;
template<class F, class T>
T useF(F f, T x)
{
    static int count = 0;
    cout << "count:" << ++count << endl;
    cout << "count:" << &count << endl;
    return f(x);
}
double f(double i)
{
    return i / 2;
}
struct Functor
{
    double operator()(double d)
    {
        return d / 3;
    }
};
int main()
{
    // 函数名
    std::function<double(double)> func1 = f;
    cout << useF(func1, 11.11) << endl;
    //函数对象
    std::function<double(double)> func2 = Functor();
    cout << useF(func2, 11.11) << endl;
    // lamber表达式
    std::function<double(double)> func3 = [](double d)->double { return d / 4; };
    cout << useF(func3, 11.11) << endl;
    return 0;
}
```

```
Microsoft Visual Studio 调试控
count:1
count:0067F4F0
5.555
count:2
count:0067F4F0
3.70333
count:3
count:0067F4F0
2.7775

C:\Users\86150\source\repos\2023.12.23\Debug\
按任意键关闭此窗口. . .
```

地址相同

**bind 配合包装器 调整参数顺序**

实现 5-10 和 10-5 p1和p2代表的是，rSub传参时，的参数1和2；然后 p1和p2对象所处的位置，按照顺序传给绑定的函数 Sub（p1，p2） Sub（p2，p1） 也就是说 Sub（被绑定函数）的参数顺序，就是按照 bind时，p1

和p2的位置参数顺序；但是 p1 和 p2的赋值是固定的 rSub的 **参1和参2** 赋值给 **p1和p2**

**【1】基本概念**

- std::bind函数定义在头文件中，是一个 函数模板，它就像一个函数包装器(适配器)，接受一个可调用对象（callable object），生成一个新的可调用对象来"适应"原对象的参数列表
- 一般主要应用于：**实现参数顺序调整等操作**

**【2】bind实现参数顺序调整的规则示意图**

- 如图中所示:
- 同样的 rSub（10，5） 通过变换bind 函数包装器 中 placeholders::_1, placeholders::_2，可以实现 10-5& 5-10

```
int Sub(int a, int b)
{
    return a - b;
}

int main()
{
    function<int(int, int)> rSub = bind(Sub, placeholders::_1, placeholders::_2);
    cout << rSub(10, 5) << endl;
```

```
int Sub(int a, int b)
{
    return a - b;
}

int main()
{
    function<int(int, int)> rSub = bind(Sub, placeholders::_2, placeholders::_1);
    cout << rSub(10, 5) << endl;
```



**绑定成员函数：静态函数 非静态函数bind 需要先获取被绑定函数地址 （其实前面 普通函数绑定时 函数名就是函数地址，只不过 这里不同类的函数，函数名可以一样，所以需要加上 类名做限定）**

$\color{red}{静态成员函数}$在 bind时，可以没有 类对象，但得表明 绑定的函数是 类中成员函数 &类名::函数名 $\color{red}{非静态成员函数}$ 第二个参数要传入 类对象 无论是**匿名对象**或**有名对象地址**

**【5】bind绑定成员函数（静态/非静态）**

主要方法分为下面三种：

1. 对于静态成员函数，直接取类的地址即可 `&SubType::sub`
2. 对于非静态成员函数，在直接取类的地址的基础上 `&SubType::sub`，法一：先实例化出一个类 `SubType st;`，取其地址 `&st`
3. 在直接取类的地址的基础上 `&SubType::sub`，法二：直接传入一个匿名对象 `SubType()`

```cpp
class SubType
{
public:
    static int sub(int a, int b)
    {
        return a - b;
    }

    int ssub(int a, int b, int rate)
    {
        return (a - b) * rate;
    }
};

int main()
{
//对于静态成员函数
    function<double(int, int)> Sub1 = bind(&SubType::sub, placeholders::_1, placeholders::_2);
        cout << Sub1(1, 2) << endl;
//对于非静态成员函数，法一
      SubType st;
    function<double(int, int)> Sub2 = bind(&SubType::ssub, &st, placeholders::_1, placeholders::_2, 3);
        cout << Sub2(1, 2) << endl;
//对于非静态成员函数，法二
    function<double(int, int)> Sub3 = bind(&SubType::ssub, SubType(), placeholders::_1, placeholders::_2,
        cout << Sub3(1, 2) << endl;

    return 0;
}
```

# 弱引用指针

## 1. 基本使用方法

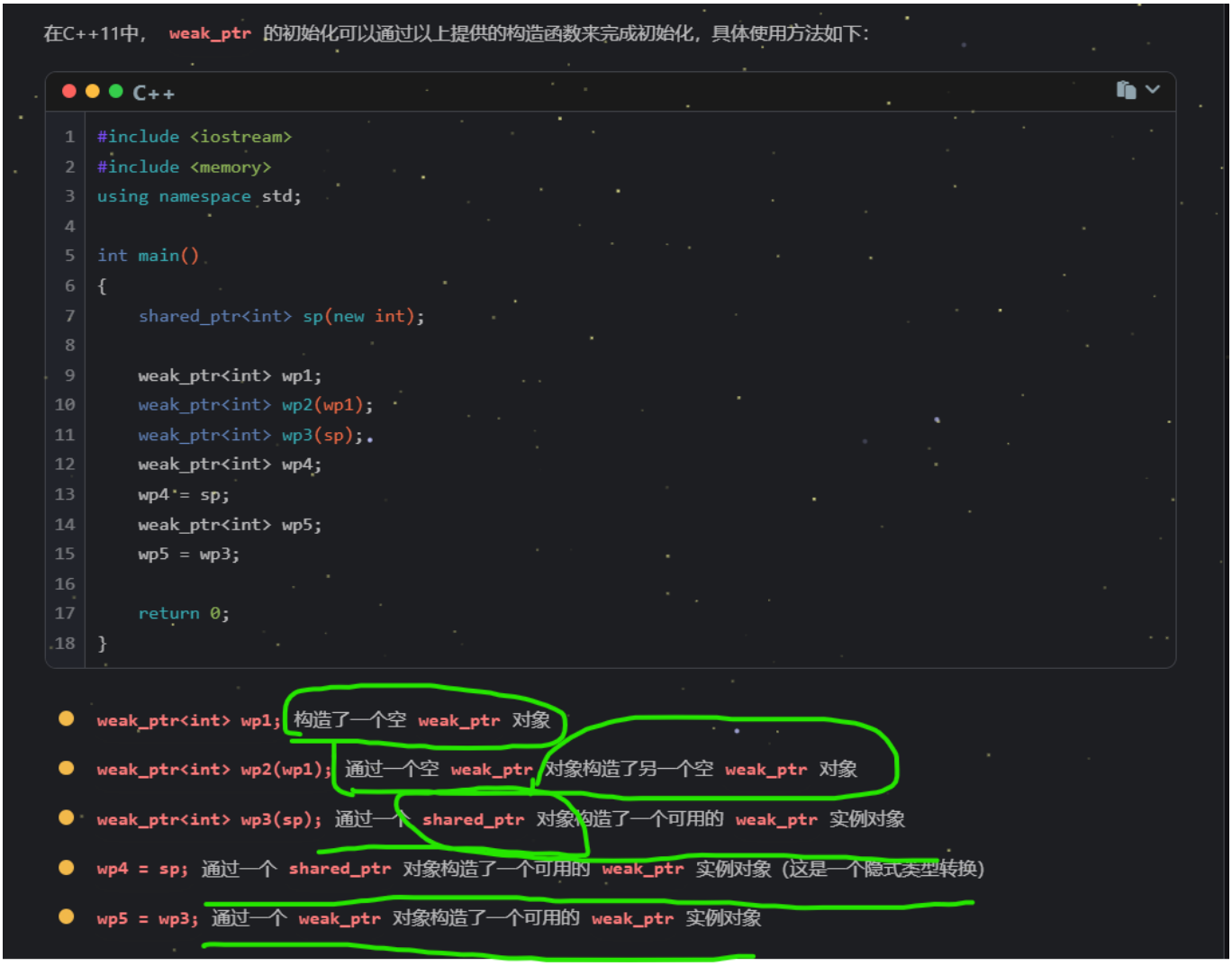
弱引用智能指针 `std::weak_ptr` 可以看做是 `shared_ptr` 的助手，它不管理 `shared_ptr` 内部的指针。 `std::weak_ptr` 没有重载操作符 `*` 和 `->` ，因为它不共享指针，不能操作资源，所以它的构造不会增加引用计数，析构也不会减少引用计数，它的主要作用就是作为一个旁观者监视 `shared_ptr` 中管理的资源是否存在。

### 1.1 初始化

C++

```cpp
// 默认构造函数
constexpr weak_ptr() noexcept;
// 拷贝构造
weak_ptr (const weak_ptr& x) noexcept;
template <class U> weak_ptr (const weak_ptr<U>& x) noexcept;
// 通过shared_ptr对象构造
template <class U> weak_ptr (const shared_ptr<U>& x) noexcept;
```

构造




**use_count() 返回监测的区域的引用计数**


weak 仅仅是监测资源，并不能引用使用资源 （注意 unique_ptr 都不允许其它人监测！！！） （weak_ptr的构造函数中，就没有关于 unique_ptr的）

```cpp
int main()
{
    shared_ptr<int> sp(new int);

    weak_ptr<int> wp1;
    weak_ptr<int> wp2(wp1);
    weak_ptr<int> wp3(sp);
    weak_ptr<int> wp4;
    wp4 = sp;
    weak_ptr<int> wp5;
    wp5 = wp3;

    cout << "use_count: " << endl;
    cout << "wp1: " << wp1.use_count() << endl;
    cout << "wp2: " << wp2.use_count() << endl;
    cout << "wp3: " << wp3.use_count() << endl;
    cout << "wp4: " << wp4.use_count() << endl;
    cout << "wp5: " << wp5.use_count() << endl;
    return 0;
}
```

测试程序输出的结果为:

```
use_count:
wp1: 0
wp2: 0
wp3: 1
wp4: 1
wp5: 1
```

通过打印的结果可以知道，虽然弱引用智能指针 `wp3` 、 `wp4` 、 `wp5` 监测的资源是同一个，但是它的引用计数并没有发生任何的变化，也进一步证明了 `weak_ptr`只是监测资源，并不管理资源 。

**expired() 返回true表示资源已经释放；否则 资源未被释放**

### 1.2.3 expired()

通过调用 `std::weak_ptr` 类提供的 `expired()` 方法来判断观测的资源是否已经被释放，函数原型如下：

```C++
// 返回true表示资源已经被释放，返回false表示资源没有被释放
bool expired() const noexcept;
```

函数的使用方法如下：

```C++
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    shared_ptr<int> shared(new int(10));
    weak_ptr<int> weak(shared);
    cout << "1. weak " << (weak.expired() ? "is" : "is not") << " expired" << endl;

    shared.reset();
    cout << "2. weak " << (weak.expired() ? "is" : "is not") << " expired" << endl;

    return 0;
}
```

测试代码输出的结果：

```C++
1. weak is not expired
2. weak is expired
```

**lock() 拷贝一份资源 返回值是 shared_ptr ， 所以需要 shared_ptr 去接收 （注意是拷贝，会导致 资源引用计数++）**

### 1.2.3 lock()

通过调用 `std::weak_ptr` 类提供的 `lock()` 方法来获取管理所监测资源的 `shared_ptr` 对象，函数原型如下：

```C++
shared_ptr<element_type> lock() const noexcept;
```

函数的使用方法如下：

```C++
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    shared_ptr<int> sp1, sp2;
    weak_ptr<int> wp;

    sp1 = std::make_shared<int>(520);
    wp = sp1;
    sp2 = wp.lock();
    cout << "use_count: " << wp.use_count() << endl;

    sp1.reset();
    cout << "use_count: " << wp.use_count() << endl;

    sp1 = wp.lock();
    cout << "use_count: " << wp.use_count() << endl;

    cout << "*sp1: " << *sp1 << endl;
    cout << "*sp2: " << *sp2 << endl;

    return 0;
}
```

测试代码输出的结果为：

```C++
use_count: 2
use_count: 1
use_count: 2
*sp1: 520
*sp2: 520
```

- `sp2 = wp.lock();` 通过调用 `lock()` 方法得到一个用于管理 `weak_ptr` 对象所监测的资源的共享智能指针对象，使用这个对象初始化 `sp2` ，此时所监测资源的引用计数为 `2`
- `sp1.reset();` 共享智能指针sp1被重置， `weak_ptr` 对象所监测的资源的引用计数减1
- `sp1 = wp.lock();` sp1重新被初始化，并且管理的还是 `weak_ptr` 对象所监测的资源，因此引用计数加1
- 共享智能指针对象 `sp1` 和 `sp2` 管理的是同一块内存，因此最终打印的内存中的结果是相同的，都是520

**reset() 不再监测任何资源**

### 1.2.4 reset()

通过调用 `std::weak_ptr` 类提供的 `reset()` 方法来清空对象，使其不监测任何资源，函数原型如下：

```C++
void reset() noexcept;
```

函数的使用是非常简单的，示例代码如下：

```C++
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    shared_ptr<int> sp(new int(10));
    weak_ptr<int> wp(sp);
    cout << "1. wp " << (wp.expired() ? "is" : "is not") << " expired" << endl;

    wp.reset();
    cout << "2. wp " << (wp.expired() ? "is" : "is not") << " expired" << endl;

    return 0;
}
```

测试代码输出的结果为：

```C++
1. wp is not expired
2. wp is expired
```

因为 不监测任何资源，所以 expired返回值为 true，所以 **其实，你可以把expired看作当前 weak指针，是否监测空资源**（shared_ptr不再指向的资源，会被释放！！变空）

`weak_ptr` 对象 `sp` 被重置之后 `wp.reset();` 变成了空对象，不再监测任何资源，因此 `wp.expired()` 返回 `true`

**可返回this 对象的 shared_ptr**

对于 shared_ptr而言 shared_ptr(this) 是创建一个新对象 并不是对同一个this对象进行引用

如果在一个类中编写了一个函数，通过这个得到管理当前对象的共享智能指针，我们可能会写出如下代码：

```C++
#include <iostream>
#include <memory>
using namespace std;

struct Test
{
    shared_ptr<Test> getSharedPtr()
    {
        return shared_ptr<Test>(this);
    }

    ~Test()
    {
        cout << "class Test is disstruct ..." << endl;
    }

};

int main()
{
    shared_ptr<Test> sp1(new Test);
    cout << "use_count: " << sp1.use_count() << endl;
    shared_ptr<Test> sp2 = sp1->getSharedPtr();
    cout << "use_count: " << sp1.use_count() << endl;
    return 0;
}
```

执行上面的测试代码，运行中会出现异常，在终端还是能看到对应的日志输出：

```C++
use_count: 1
use_count: 1
class Test is disstruct ...
class Test is disstruct ...
```

通过输出的结果可以看到 一个对象被析构了两次 ，其原因是这样的：在这个例子中使用同一个指针 `this` 构造了两个智能指针对象 `sp1` 和 `sp2` ，这二者之间是没有任何关系的，因为 `sp2` 并不是通过 `sp1` 初始化得到的实例对象。在离开作用域之后 `this` 将被构造的两个智能指针各自析构，导致重复析构的错误。

这个问题可以通过 `weak_ptr` 来解决，通过 `wek_ptr` 返回管理 `this` 资源的共享智能指针对象 `shared_ptr` 。C++11中为我们提供了一个模板类叫做 `std::enable_shared_from_this<T>` ，这个类中有一个方法叫做 `shared_from_this()` ，通过这个方法可以返回一个共享智能指针，在函数的内部就是使用 `weak_ptr` 来监测 `this` 对象，并通过调用 `weak_ptr` 的 `lock()` 方法返回一个 `shared_ptr` 对象。

修改代码，利用 内部是 weak_ptr 的lock方法的类 shared_from_this();

修改之后的代码为:

```C++
#include <iostream>
#include <memory>
using namespace std;

struct Test : public enable_shared_from_this<Test>
{
    shared_ptr<Test> getSharedPtr()
    {
        return shared_from_this();
    }
    ~Test()
    {
        cout << "class Test is disstruct ..." << endl;
    }
};

int main()
{
    shared_ptr<Test> sp1(new Test);
    cout << "use_count: " << sp1.use_count() << endl;
    shared_ptr<Test> sp2 = sp1->getSharedPtr();
    cout << "use_count: " << sp1.use_count() << endl;
    return 0;
}
```

测试代码输出的结果为:

```C++
use_count: 1
use_count: 2
class Test is disstruct ...
```

最后需要强调一个细节：在调用enable_shared_from_this类的shared_from_this()方法之前，必须要先初始化函数内部weak_ptr对象，否则该函数无法返回一个有效的shared_ptr对象（具体处理方法可以参考上面的示例代码）。

** 初始化的要求是 继承时说明 父类的模板的数据类型 为Test** 这也是 当所有的 基类是模板类时，子类继承时要做的事

测试代码输出的结果为:

```C++
use_count: 1
use_count: 2
class Test is disstruct ...
```

最后需要强调一个细节：在调用enable_shared_from_this类的shared_from_this()方法之前，必须要先初始化函数内部weak_ptr对象，否则该函数无法返回一个有效的shared_ptr对象（具体处理方法可以参考上面的示例代码）。

例：

```C++
template<typename T>
class Test1 {

};

class Test2 :public Test1<int> {

};
```

**解决循环引用问题**

智能指针如果循环引用会导致内存泄露，比如下面的例子：

```C++
#include <iostream>
#include <memory>
using namespace std;

struct TA;
struct TB;

struct TA
{
    shared_ptr<TB> bptr;
    ~TA()
    {
        cout << "class TA is disstruct ..." << endl;
    }
};

struct TB
{
    shared_ptr<TA> aptr;
    ~TB()
    {
        cout << "class TB is disstruct ..." << endl;
    }
};

void testPtr()
{
    shared_ptr<TA> ap(new TA);
    shared_ptr<TB> bp(new TB);
    cout << "TA object use_count: " << ap.use_count() << endl;
    cout << "TB object use_count: " << bp.use_count() << endl;

    ap->bptr = bp;
    bp->aptr = ap;
    cout << "TA object use_count: " << ap.use_count() << endl;
    cout << "TB object use_count: " << bp.use_count() << endl;
}

int main()
{
    testPtr();
    return 0;
}
```

相当于 四个指针 对两个对象指向 但是最后共享智能指针离开作用域后，每个对象的引用计数只能-1 ，所以对象无法释放 引用计数只能 -1的原因是：指针 **ap** 和 **bp** 的销毁，不会直接导致 对象的销毁，对象存在，则对象中的 **aptr** 和 **bptr** 依然存在； 对象被指针给指向，则对象又无法被销毁....因此 形成一个死循环，对象无法销毁！！！！

测试程序输出的结果如下:

```C++
TA object use_count: 1
TB object use_count: 1
TA object use_count: 2
TB object use_count: 2
```

在测试程序中，共享智能指针 `ap` 、 `bp` 对 `TA` 、 `TB` 实例对象的引用计数变为2， 在共享智能指针离开作用域之后引用计数只能减为1 ，这种情况下不会去删除智能指针管理的内存，导致类 `TA` 、 `TB` 的实例对象不能被析构，最终造成内存泄露。通过使用 `weak_ptr` 可以解决这个问题，只要将类 `TA` 或者 `TB` 的任意一个成员改为 `weak_ptr` ，修改之后的代码如下:

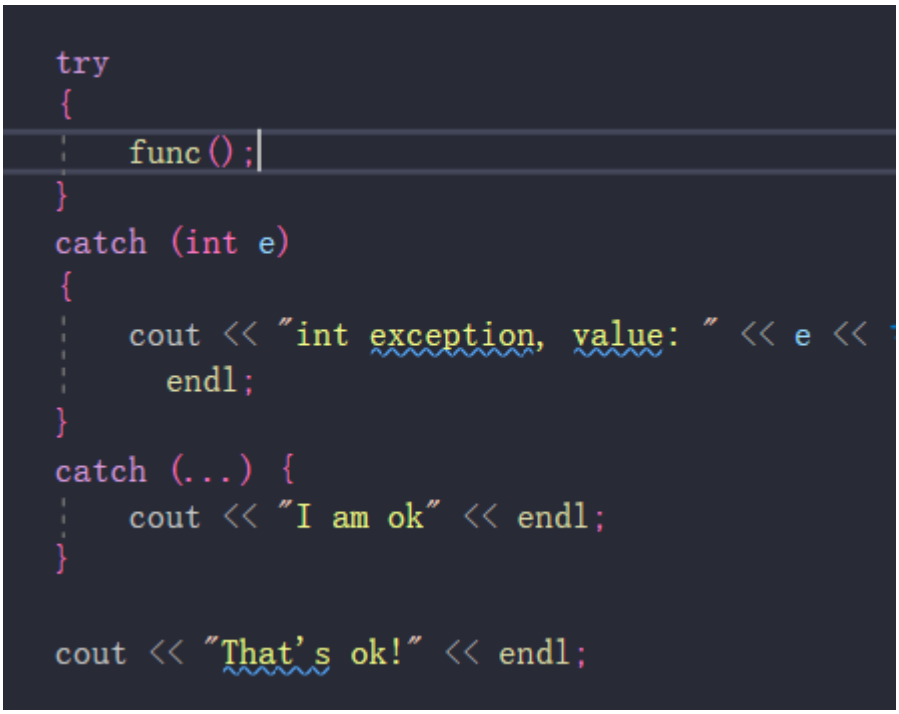
```C++
#include <iostream>
#include <memory>
using namespace std;

struct TA;
struct TB;

struct TA
{
    weak_ptr<TB> bptr;
    ~TA()
    {
        cout << "class TA is disstruct ..." << endl;
    }
};

struct TB
{
    shared_ptr<TA> aptr;
    ~TB()
    {
        cout << "class TB is disstruct ..." << endl;
    }
};
```


（上图也说明 TB对象销毁后，其内部的 aptr指针才会被接着析构）




try **内部的函数，相当于是保护哪一个函数，（接住 哪一个函数抛出的异常）**