一、智能指针

1. 指针潜在问题

> c++ 把内存的控制权对程序员开放，让程序显式的控制内存，这样能够快速的定位到占用的内存，完成释放的工作。但是此举经常会引发一些问题，比如忘记释放内存。由于内存没有得到及时的回收、重复利用，所以在一些c++程序中，常会遇到程序突然退出、占用内存越来越多，最后不得不选择重启来恢复。造成这些现象的原因可以归纳为下面几种情况.

1. 野指针

> **出现野指针的有几个地方：**
>
> a. 指针声明而未初始化，此时指针的将会随机指向
>
> b. 内存已经被释放、但是指针仍然指向它。这时内存有可能被系统重新分配给程序使用，从而会导致无法估计的错误

```cpp
#include <iostream>

using namespace std;

int mian(){


    //1. 声明未初始化
    int *p1 ;
    cout << "打印p1: " << *p1 << endl;


    //2. 内存释放后，并没有置空 nullptr
    int *p = new int(55);

    cout << "释放前打印 :  " << *p << endl;

    delete  p ;
    cout << "释放后打印 :  " << *p << endl;

    return 0 ;
}
```

2. 重复释放

> 程序试图释放已经释放过的内存，或者释放已经被重新分配过的内存，就会导致重复释放错误.

```cpp
int main(){

    int *p = new int(4);

    //重复释放
    delete p;
    delete p;

    return 0 ;
}
```

3. 内存泄漏

> 不再使用的内存，并没有释放，或者忘记释放，导致内存没有得到回收利用。 忘记调用 delete

```
int main(){

    int *p = new int(4);


    //后面忘记调用delete p;

    return 0 ;
}
```

2. 智能指针

> 为了解决普通指针的隐患问题，c++在98版本开始追加了智能指针的概念，并在后续的11版本中得到了提升。
>
> 在98版本提供的 `auto_ptr` 在 c++11得到删除，原因是拷贝是返回左值、不能调用delete[]等。 c++11标准改用 `unique_ptr` | `shared_ptr` | `weak_ptr` 等指针来自动回收堆中分配的内存。智能指针的用法和原始指针用法一样，只是它多了些释放回收的机制罢了。
>
> 智能指针位于`` 头文件中，所以要想使用智能指针，还需要导入这个头文件 `#include`

1. unique_ptr

> `unique_ptr` 是一个独享所有权的智能指针，它提供了严格意义上的所有权。也就是只有这个指针能够访问这片空间，不允许拷贝，但是允许移动（转让所有权）。

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

using namespace std;

int main(){


        //1. 创建unique_ptr对象，包装一个int类型指针
        unique_ptr<int> p(new int(10));

    //2. 无法进行拷贝。编译错误
        //unique_ptr<int> p2 = p;
        cout << *p << endl;

    //3. 可以移动指针到p3. 则p不再拥有指针的控制权 p3 现在是唯一指针
        unique_ptr<int> p3 = move(p) ;
        cout << *p3 << endl;

    //p 现在已经无法取值了。
        cout << *p << endl;

    //可以使用reset显式释放内存。
        p3.reset();

    //重新绑定新的指针
        p3.reset(new int(6));

    //获取到曾经包装的int类型指针
        int *p4 = p3.get() ;

    //输出6
    cout << "指针指向的值是: " << *p4  << endl;

        return 0 ;
}
```

2. shared_ptr

> shared_ptr ：允许多个智能指针共享同一块内存，由于并不是唯一指针，所以为了保证最后的释放回收，采用了计数处理，每一次的指向计数 + 1， 每一次的reset会导致计数 -1 ，直到最终为0，内存才会最终被释放掉。 可以使用 use_cout 来查看目前的指针个数

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

class stu{

public:
    stu(){
        cout << "执行构造函数" <<endl;
    }

    ~stu(){
        cout << "执行析构函数" <<endl;
    }

};

int main(){

    shared_ptr<stu> s1 ( new stu());

   cout <<" cout = " << s1.use_count() <<endl; //查看指向计数

    shared_ptr<stu> s2  = s1;

    s1.reset();
    s2.reset();  // 至此全部解除指向 计数为0 。 会执行stu的析构函数

    return 0 ;
}
```

3. shared_ptr的问题

> 对于引用计数法实现的计数，总是避免不了循环引用（或环形引用）的问题，即我中有你，你中有我， shared_ptr 也不例外。 下面的例子就是，这是因为f和s内部的智能指针互相指向了对方，导致自己的引用计数一直为1，所以没有进行析构，这就造成了内存泄漏。

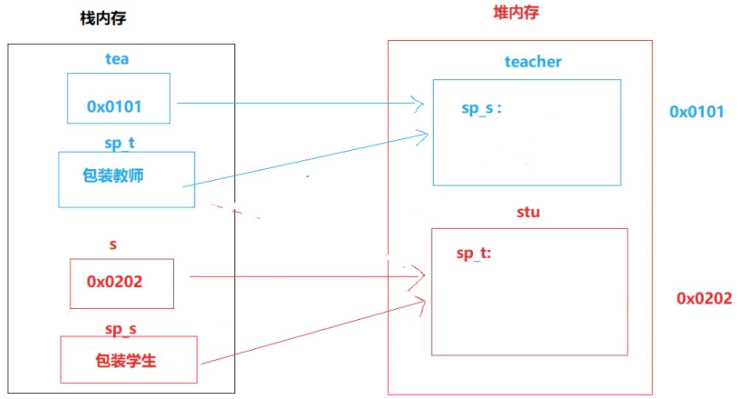
```
class father {
public:
    father(){cout <<"father 构造" << endl;}
    ~father(){cout <<"father 析构" << endl;}

    void setSon(shared_ptr<son> s) {
        son = s;
    }
private:
    shared_ptr<son> son;
};

class son {
public:
    son(){cout <<"son 构造" << endl;}
    ~son(){cout <<"son 析构" << endl;}
    void setFather(shared_ptr<father> f) {
        father = f;
    }
private:
    shared_ptr<father> father;
};


int main(){

    shared_ptr<father> f(new father());
    shared_ptr<son> s(new son());
    f->setSon(s);
    s->setFather(f);
}
```

```
class teacher{
public:
    shared_ptr<stu> sp_s;
    void setStu( shared_ptr<stu> sp){
        sp_s = sp;
    }
};
class stu{
public:
    shared_ptr<teacher> sp_t;
    void setTeacher( shared_ptr<teacher> sp){
        sp_t = sp;
    }
};
int main() {
    //1. 创建学生和教师 ， 指针
    teacher * tea = new teacher;
    stu * s = new stu;
    //2. 先把教师和学生包装成智能指针
    shared_ptr<teacher> sp_t(tea);
    shared_ptr<stu> sp_s(s);
    //3. 让教师关联学生，让学生关联教师
    tea->setStu(sp_s); //教师关联学生
    s->setTeacher(sp_t); //学生关联上教师/**/
    return 0;
```

问题： 析构不执行， 指针没销毁，计数不为 0

指向教师的智能指针： 1个

指向学生的智能指针： 1个

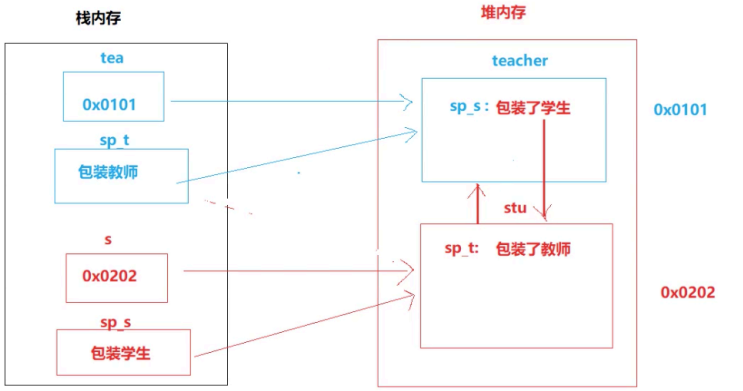



```
class teacher{
public:
    shared_ptr<stu> sp_s;
    void setStu( shared_ptr<stu> sp){
        sp_s = sp;
    }
};
class stu{
public:
    shared_ptr<teacher> sp_t;
    void setTeacher( shared_ptr<teacher> sp){
        sp_t = sp;
    }
};
int main() {
    //1. 创建学生和教师 ， 指针
    teacher * tea = new teacher;
    stu * s = new stu;
    //2. 先把教师和学生包装成智能指针
    shared_ptr<teacher> sp_t(tea);
    shared_ptr<stu> sp_s(s);
    //3. 让教师关联学生，让学生关联教师
    tea->setStu(sp_s); //教师关联学生
    s->setTeacher(sp_t); //学生关联上教师/**/
    return 0;
```

问题： 析构不执行， 指针没销毁，计数不为 0

指向教师的智能指针： 2个

指向学生的智能指针： 2个



4. weak_ptr

> 为了避免 shared_ptr 的环形引用问题，需要引入一个弱指针 weak_ptr ，它指向一个由 shared_ptr 管理的对象而不影响所指对象的生命周期，也就是将一个 weak_ptr 绑定到一个 shared_ptr 不会改变 shared_ptr 的引用计数。不论是否有 weak_ptr 指向，一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放。从这个角度看， weak_ptr 更像是 shared_ptr`的一个助手而不是智能指针。

```
class father {
public:
    father(){cout <<"father 构造" << endl;}
    ~father(){cout <<"father 析构" << endl;}
    void setSon(shared_ptr<son> s) {
        son = s;
    }
private:
    shared_ptr<son> son;
};

class son {
public:
    son(){cout <<"son 构造" << endl;}
    ~son(){cout <<"son 析构" << endl;}
    void setFather(shared_ptr<father> f) {
        father = f;
    }
private:
    //shared_ptr<father> father;
    weak_ptr<father> father;  //替换成weak_ptr 即可。
};


int main(){

    shared_ptr<father> f(new father());
    shared_ptr<son> s(new son());
    f->setSon(s);
    s->setFather(f);
}
```

5. 智能指针的应用场景

> 1.智能指针其实就是能够帮助我们去释放原始指针
>
> 2.一般使用在类里面的成员变量身上，如果这个变量是一个指针，那么这些指针大多数情况下都会被写成智能指针的写法。

二、动态内存

1. 内存分区

> 在C++中内存分为5个区，分别是 堆 、 栈 、 全局/静态存储区 和 代码|常量存储区 | 共享内存区 。

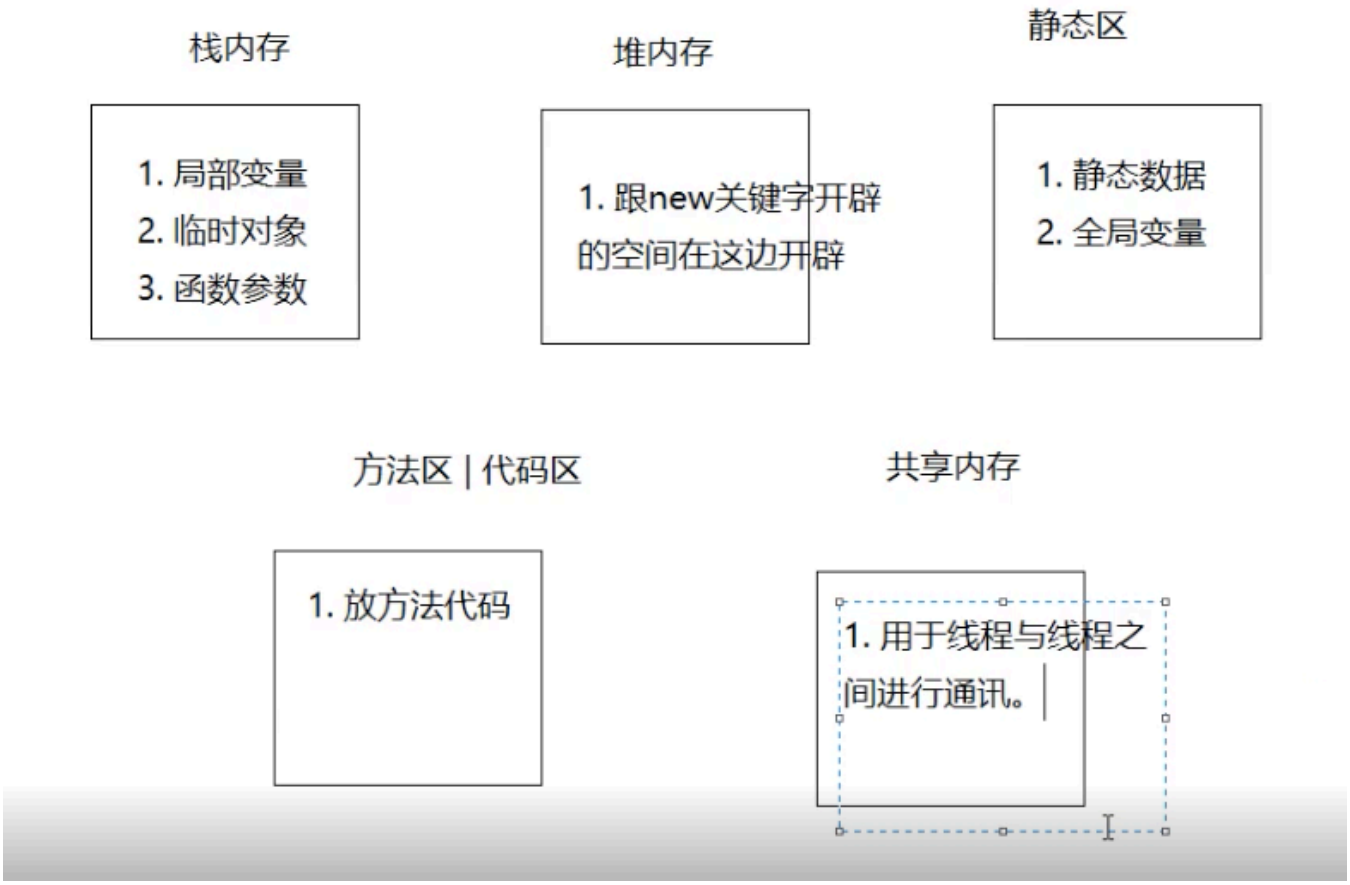
栈区：又叫堆栈，存储非静态局部变量、函数参数、 返回值等，栈是可以向下生长的

共享内存区：是高效的I/O映射方式，用于装载一个共享的动态内存库。用户可使用系统接口创建共享共享内 存，做进程间通讯

堆区：用于程序运行时动态内存分配，堆是可以向上增长的

静态区：存储全局数据和静态数据

代码区：存储可执行的代码、只读常量



2. new 和 delete

> 在 c++ 中， 如果要在堆内存中申请空间，那么需要借助 `new` 操作符，释放申请的空间，使用 `delete` 操作 。而c语言使用的是 `malloc` 和 `free` ， 实际上 `new` 和 `delete` 的底层实际上就是 `malloc` 和 `free` 。

1. new

> 在c++中， new是一个关键字，同时也是一个操作符，用于在堆区申请开辟内存，new的操作还具备以下几个特征：
>
> 1. 内存申请成功后，会返回一个指向该内存的地址。
> 2. 若内存申请失败，则抛出异常，
> 3. 申请成功后，如果是程序员定义的类型，会执行相应的构造函数

```cpp
#include <iostream>

using namespace std;

class stu{

    stu(){
        cout << "执行构造函数" <<endl;
    }

    ~stu(){
        cout << "执行析构函数" <<endl;
    }

}


int main(){
    int *a = new int();
    stu *s = new stu();

    //new的背后先创建
    return 0 ;
}
```

2. delete

> 在c++中， `delete` 和 `new` 是成对出现的，所以就有了 `no new no delete` 的说法。 `delete` 用于释放 `new` 申请的内存空间。 `delete` 的操作具备以下几个特征：
>
> 1. 如果指针的值是0，delete不会执行任何操作，有检测机制
> 2. delete只是释放内存，不会修改指针，指针仍然会指向原来的地址
> 3. 重复delete，有可能出现异常
> 4. 如果是自定义类型，会执行析构函数

```
int main(){

    int *p = new int(6);
    delete p ; // 回收数据

    *p = 18 ; //依然可以往里面存值，但是不建议这么做。
    return 0 ;
}
```

3. malloc 和 free

> malloc` 和 free 实际上是C语言 申请内存的语法，在C++ 也得到了保存。只是与 new 和 delete 不同的是， 它们

- malloc

> 1. malloc 申请成功之后， 返回的是void类型的指针。需要将void*指针转换成我们需要的类型。1.
> 2. malloc 要求制定申请的内存大小， 而new由编译器自行计算。
> 3. 申请失败，返回的是NULL， 比如： 内存不足。
> 4. 不会执行自定义类型的构造函数

```
int main(){

    int *p=(int *)malloc(int); //如果申请失败，返回的是NULL
    return 0 ;
}
```

- free

> free 和 malloc是成堆出现的， 所以也有了 no malloc no free的说法。 free 用于释放 mallo 申请的内存空间。
>
> 1. 如果是空指针， 多次释放没有问题， 非空指针， 重复释放有问题
> 2. 不会执行对应的析构
> 3. delete的底层执行的是free

```c
int main(){

    int *p=(int *)malloc(int); //如果申请失败，返回的是NULL

    free(p);
    return 0 ;
}
```

new delete

new [] delete[]