指针最重要的两点：指针本身的寿命，指向对象的寿命。

C++11中的两端类型的smart指针：

1. shared\_ptr：共享式拥有，多个smart指针指向同一个对象，该对象和相关资源会在“最后一个reference被销毁”时被释放。还有weak\_ptr, bad\_weak\_ptr等辅助类
2. unique\_ptr：独占式拥有，同一时间内只有一个smart指针指向该对象，可以移交拥有权。

shared\_ptr的使用：

例：程序stl\_test15

// 当最后一个拥有者被销毁时，会调用lambda函数

*shared\_ptr*<*string*> nico\_ptr(new *string*("nico"), [](*string*\* p) {

*cout* << "delete " << (\*p).*c\_str*() << *endl*;

delete p;

});

*vector*<*shared\_ptr*<*string*>> student;

student.*push\_back*(nico\_ptr);

student.*push\_back*(nico\_ptr);

student.*push\_back*(nico\_ptr);

// nico nico nico

for (auto ptr : student)

{

*cout* << (\*ptr).*c\_str*() << " ";

}

*cout* << *endl*;

nico\_ptr->*replace*(0, 1, "N");

// Nico Nico Nico

for (auto ptr : student)

{

*cout* << (\*ptr).*c\_str*() << " ";

}

*cout* << *endl*;

*cout* << "use count: " << student[0].*use\_count*() << *endl*; // 4

输出为：

nico nico nico

Nico Nico Nico

use count: 4

delete Nico

在程序中，在vector容器中插入了3个shared\_ptr对象，这样new string(“nico”)就有了4个smart指针指向它，当其中一个修改了string的值时，其它的都会变化。在程序结束时，当最后一个拥有者被销毁时，new string的指针被销毁，调用了lambda函数。

数组的shared\_ptr

shared\_ptr默认调用delete，而不是delete[]，通过“new建立起来的单一对象”，可以使用默认的delete，对于数组使用相同的方式，可以编译通过，但并不正确，所以需要定义自己的删除器deleter。

例：程序stl\_test15

// 数组

*shared\_ptr*<int> p(new int[5], [](int\* p) {

delete[] p;

});

weak\_ptr：在以下几种情况下会出现shared\_ptr不适用的情况

1. 环式指向。两个对象分别指向对方，如果使用shared\_ptr会出现资源无法释放的情况，因为use\_count始终为1的情况；
2. “明确想共享但不拥有”某对象，reference的寿命比其所指的对象寿命更长。shared\_ptr会在最末拥有者销毁时才会释放对象，其reference不会比所指对象的寿命更长。

例：程序stl\_test16

class Person

{

public:

Person(const *string*& name,

*shared\_ptr*<Person> mother = nullptr,

*shared\_ptr*<Person> father = nullptr)

: name\_(name), mother\_(mother), father\_(father)

{

}

~Person()

{

*cout* << "delete " << name\_.*c\_str*() << *endl*;

}

public:

*string* name\_;

*shared\_ptr*<Person> mother\_;

*shared\_ptr*<Person> father\_;

// 这里要使用弱smart指针

// 如果使用shared\_ptr，由于

// 存在环式指向，会造成

// 指向对象不释放的情况

*vector*<*weak\_ptr*<Person>> kids\_;

};

*shared\_ptr*<Person> InitFamily(const *string*& name)

{

*shared\_ptr*<Person> mom(new Person(name + "'s mom"));

*shared\_ptr*<Person> dad(new Person(name + "'s dad"));

*shared\_ptr*<Person> kid(new Person(name, mom, dad));

mom->kids\_.*push\_back*(kid);

dad->kids\_.*push\_back*(kid);

return kid;

}

int *main*(int argc, char\* argv[])

{

*shared\_ptr*<Person> ptr = InitFamily("nico");

*cout* << "nico is shared " << ptr.*use\_count*() << " times" << *endl*; // 1

*cout* << "name of 1st kid of nico's mom " <<

ptr->mother\_->kids\_[0].*lock*()->name\_.*c\_str*() << *endl*;

ptr = InitFamily("jim");

*cout* << "jim's family exists" << *endl*;

return 0;

}

输出：

nico is shared 1 times

name of 1st kid of nico's mom nico

delete nico

delete nico's dad

delete nico's mom

jim's family exists

delete jim

delete jim's dad

delete jim's mom

判断weak\_ptr背后的对象是否存在

例：程序stl\_test16

// 判断weak\_ptr背后的对象是否存在

try

{

*shared\_ptr*<*string*> sp(new *string*("hi"));

*weak\_ptr*<*string*> wp = sp;

sp.*reset*(); // 释放shared\_ptr，并释放指向的对象

// weak\_ptr背后的对象不存在

*cout* << wp.*use\_count*() << *endl*; // 0

*cout* << *boolalpha* << wp.*expired*() << *endl*; // true

*shared\_ptr*<*string*> p1 = wp.*lock*(); // p1为empty

*shared\_ptr*<*string*> p2(wp); // throw bad\_weak\_ptr

}

catch (const *exception*& e)

{

*cerr* << "exception: " << e.*what*() << *endl*;

}

输出：

0

true

exception: bad\_weak\_ptr

unique\_ptr：独占式资源smart指针，一个对象及其相应资源在同一个时刻只能被一个pointer拥有，一旦拥有者被销毁、变empty或拥有另一个对象，先前拥有的对象就会被销毁。

例：程序stl\_test17

*unique\_ptr*<*string*> up(new *string*("hello"));

// unique\_ptr同一时刻只允许存在一个拥有者

// unique\_ptr<string> up1(up); // error

// unique\_ptr<string> up2 = up; // error

// 可以通过move语义将拥有权转移

*unique\_ptr*<*string*> up3(*move*(up)); // ok

unique\_ptr的删除

例：程序stl\_test17

void Sink(*unique\_ptr*<*string*> up)

{

}

class Test

{

public:

Test() {}

~Test() { *cout* << "delete Test" << *endl*; }

};

*unique\_ptr*<Test> Source()

{

*unique\_ptr*<Test> ptr(new Test);

// 不加move也没关系，编译器会自动加上

return *move*(ptr);

}

void Func()

{

*unique\_ptr*<Test> ptr;

for (int i = 0; i < 5; ++i)

{

ptr = Source();

}

*cout* << "Unique Ptr test end" << *endl*;

}

*unique\_ptr*<*string*> up4(new *string*("welcome"));

// 需要使用move语义

Sink(*move*(up4));

Func();

输出：

delete Test

delete Test

delete Test

delete Test

Unique Ptr test end

delete Test

从输出结果看出：

当unique\_ptr被赋予新的对象时，之前拥有的会被删除。Func()函数结束时，最后一个ptr会被析构，其拥有的对象也会被删除。

unique\_ptr作为类成员：

如果普通的指针做为类成员，在构造函数中初始化，在析构函数中释放，这里有可能会产生内存泄漏，原因是如果构造函数时发生异常，则在析构函数就不会执行，已开辟的内存也不会释放。

使用unique\_ptr可以避免这类问题：

例：程序stl\_test18

class A

{

public:

A()

{

*cout* << "construct A" << *endl*;

}

A(const A& src)

{

*cout* << "copy construct A" << *endl*;

}

~A()

{

*cout* << "destruct A" << *endl*;

}

};

class Test

{

public:

// 如果在初始化ptr2时，new失败

// 那么ptr1也会释放已分配的内存

Test() : ptr1(new A), ptr2(new A)

{

}

// 拷贝构造函数是必要的

// unique\_ptr不能被默认拷贝

// Test也不能进行拷贝构造

Test(const Test& src) : ptr1(new A(\*src.ptr1)), ptr2(new A(\*src.ptr2))

{

}

// 赋值运算符重载是必要的

// unique\_ptr不能被默认赋值

const Test& operator= (const Test& src)

{

\*ptr1 = \*src.ptr1;

\*ptr2 = \*src.ptr2;

return \*this;

}

// 这里可以忽略析构函数

// unique\_ptr会自动释放开辟的资源

private:

*unique\_ptr*<A> ptr1;

*unique\_ptr*<A> ptr2;

};

int main(int argc, char\* argv[])

{

Test t1;

Test t2(t1);

return 0;

}

输出

construct A

construct A

copy construct A

copy construct A

destruct A

destruct A

destruct A

destruct A

unique\_ptr拥有数组

// 运行时会崩溃

// unique\_ptr调用delete，但其拥有的对象是数组

// 在C++里必须调用delete[]

// unique\_ptr<string> up(new string[10]); // runtime error

// 在这个偏特化版本中，unique\_ptr不能使用\*和->

// 可以使用[]

*unique\_ptr*<*string*[]> up(new *string*[5]); // ok

unique\_ptr自定义删除：

class ADeleter

{

public:

void operator() (A\* ptr)

{

*cout* << "call delete for class A object" << *endl*;

delete ptr;

}

};

// 自定义删除

// template第2个实参必须是函数的引用，函数指针或函数对象

*unique\_ptr*<int, function<void(int\*)>> up1(new int[10], [](int\* p) {

delete[] p;

});

*unique\_ptr*<int, void(\*)(int\*)> up2(new int[10], [](int\* p) {

delete[] p;

});

*unique\_ptr*<A, ADeleter> up3(new A);