shared\_ptr智能指针

什么是shared\_ptr智能指针？

通过增加引用计数来共享同一块堆内存，当引用计数为0时，会自动释放该空间。

shared\_ptr智能指针的创建？

1. 空shared\_ptr智能指针的引用计数为0。
2. 可以使用 make\_shared进行初始化。
3. 使用拷贝构造函数和移动构造函数（左值用来赋值给别人初始化，就是拷贝构造函数）
4. 普通的指针不能为多个shared\_ptr智能指针初始化，会发生异常。
5. 在初始化shared\_ptr智能指针的时候，可以自定义指针的释放规则。（shared\_ptr默认的释放不支持释放申请的动态数组，所以释放申请的动态数组的时候需要自定义释放规则）

#include <iostream>

#include <memory>

using namespace std;

void deleteInt(int \*p)

{

    delete []p;

}

int main()

{

    std::shared\_ptr<int> p;    //空智能指针，引用计数为0

    std::shared\_ptr<int> q(nullptr);    //空智能指针，引用计数为0

    std::shared\_ptr<int> a(new int(10));

    std::cout<<\*a<<std::endl;

    std::shared\_ptr<int> b = std::make\_shared<int>(10); //使用make\_shared进行初始化

    std::cout<<\*b<<std::endl;

    std::shared\_ptr<int> c(a); //a是左值，所以是拷贝构造函数

    std::cout<<\*c<<std::endl;

    std::shared\_ptr<int> d = std::move(a); //移动构造函数，将a的堆内存给了d，此时a是空智能指针

    //std::cout<<\*d<<\*a<<std::endl;

    int \*e = new int(20);    //普通的指针

    std::shared\_ptr<int> f(e);

    std::cout<<\*f<<std::endl;

    //std::shared\_ptr<int> g(e);

    //std::cout<<\*g<<std::endl;

    //自定义释放规则1:使用std::default\_delete

    std::shared\_ptr<int> h(new int[10], std::default\_delete<int[]>());

    //自定义释放规则2:使用自定义的函数

    std::shared\_ptr<int> i(new int[10], deleteInt);

    //自定义释放规则3:使用Lambda表达式

    std::shared\_ptr<int> j(new int[10], [](int \*k){delete []k;});

    return 0;

}

shared\_ptr智能指针的一些成员方法

1. bool()
2. unique()
3. use\_count()
4. get()
5. reset()
6. swap()

#include <iostream>

#include <memory>

using namespace std;

int main()

{

    std::shared\_ptr<int> p1(new int(10));

    //bool用来判断shared\_ptr是否是空智能指针，空指针返回0，非空指针返回1

    std::cout<<"p1是空指针吗："<<bool(p1)<<std::endl;

    std::shared\_ptr<int> p2 = std::move(p1);

    std::cout<<"p2是空指针吗："<<bool(p2)<<std::endl;

    std::cout<<"p1是空指针吗："<<bool(p1)<<std::endl;

    //unique用来判断p2指向的堆内存是否只有p2指向它而已，是返回1，不是返回0

    std::cout<<"p2指向的堆内存，是否只有p2指向它而已："<<p2.unique()<<std::endl;

    std::shared\_ptr<int> p3(p2);

    std::cout<<"p2指向的堆内存，是否只有p2指向它而已："<<p2.unique()<<std::endl;

    //use\_count用来返回当前有多少个指针指向一样的堆空间

    std::cout<<"当前有多少个指针指向和p2指向一样的堆空间："<<p2.use\_count()<<std::endl;

    int \*p4 = new int(20);

    std::cout<<"普通指针p4的地址："<<p4<<std::endl;

    std::shared\_ptr<int> p5(p4);

    //get用来返回智能指针指向普通指针的地址

    std::cout<<"获取p5包含的普通指针："<<p5.get()<<std::endl;

    std::shared\_ptr<int> p6(new int(30));

    std::cout<<"p6是空指针吗："<<bool(p6)<<std::endl;

    //reset当没有实参的时候，将当前对象置为空指针，当前对象所指的堆内存的引用计数减1

    p6.reset();

    std::cout<<"p6是空指针吗："<<bool(p6)<<std::endl;

    //reset当有实参的时候，当前对象获得实参的所有使用权，并且所指的堆内存的引用计数为1

    p6.reset(new int(40));

    std::cout<<"p6是空指针吗："<<bool(p6)<<std::endl;

    std::cout<<"p5是多少："<<\*p5<<"p6是多少："<<\*p6<<std::endl;

    std::shared\_ptr<int> p7(p5);

    std::cout<<"当前有多少个指针指向和p5指向一样的堆空间："<<p5.use\_count()<<"p7是多少："<<\*p7<<std::endl;

    //swap用来交换两个相同类型的shared\_ptr智能指针的内容，其它指向同一堆内存的智能指针的值也会跟着交换

    swap(\*p5, \*p6);

    std::cout<<"p5是多少："<<\*p5<<"p6是多少："<<\*p6<<std::endl;

    std::cout<<"当前有多少个指针指向和p5指向一样的堆空间："<<p5.use\_count()<<"p7是多少："<<\*p7<<std::endl;

    return 0;

}

weak\_ptr智能指针

使用weak\_ptr的好处：

因为使用shared\_ptr的时候，容易造成内存泄漏【两个类彼此持有对方的智能指针】，这时候就可以使用weak\_ptr

#include <iostream>

#include <memory>

using namespace std;

class A2;

class A1

{

public:

std::shared\_ptr<A2> p1;

~A1(){std::cout<<"A1析构"<<std::endl;}

};

class A2

{

public:

std::shared\_ptr<A1> p2;

~A2(){std::cout<<"A2析构"<<std::endl;}

};

int main() {

auto pA = std::shared\_ptr<A1>();

auto pB = std::shared\_ptr<A2>();

pA->p1 = pB;

pB->p2 = pA;

//pA指向类A1，pA的成员p1指向pB，pB指向类A2，pB的成员p2指向pA

//pA和pB离开作用域就析构了，这样外界就无法访问p1和p2了，但是它们还没有释放，所以内存泄漏

std::cout<<"pA的引用计数："<<pA.use\_count()<<std::endl;

std::cout<<"pB的引用计数："<<pB.use\_count()<<std::endl;

return 0;

}

#include <iostream>

#include <memory>

using namespace std;

class A2;

class A1

{

public:

std::shared\_ptr<A2> p1;

~A1(){std::cout<<"A1析构"<<std::endl;}

};

class A2

{

public:

std::weak\_ptr<A1> p2;

~A2(){std::cout<<"A2析构"<<std::endl;}

};

int main() {

auto PA = std::shared\_ptr<A1>();

auto PB = std::shared\_ptr<A2>();

PA->p1 = PB;

PB->p2 = PA;

return 0;

}

什么是weak\_ptr智能指针？

是一种弱引用，指向一块堆内存堆时候，不会增加引用计数。所以只能访问，不能修改。

weak\_ptr智能指针的创建？

1. 拷贝构造函数
2. 可以使用shared\_ptr进行初始化

int main() {

std::weak\_ptr<int> p1;

std::weak\_ptr<int> p2(p1);

std::weak\_ptr<int> p3 = std::shared\_ptr<int>(new int(10));

return 0;

}

weak\_ptr智能指针的一些成员方法

1、use\_count()

2、reset()

3、swap()

4、expired()

5、lock()

int main() {

std::shared\_ptr<int> p1(new int(10));

std::weak\_ptr<int> p2 = p1;

std::shared\_ptr<int> p3 = std::make\_shared<int>(20);

std::weak\_ptr<int> p4(p3);

std::shared\_ptr<int> p5(p1);

//判断当前 weak\_ptr 指针为否过期

std::cout<<p2.expired()<<std::endl;

//如果当前 weak\_ptr 已经过期，则该函数会返回一个空的 shared\_ptr 指针；反之，该函数返回一个和当前 weak\_ptr 指向相同的 shared\_ptr 指针。

auto p22 = p2.lock();

std::cout<<\*p22<<std::endl;

return 0;

}

unique\_ptr智能指针

什么是unique\_ptr智能指针？

使用堆空间的时候，独自占有不共享，引用计数都只能为1

unique\_ptr智能指针的创建？

1. 不能使用拷贝构造函数，因为是独占的。
2. 在初始化unique\_ptr智能指针的时候，可以自定义指针的释放规则，只能使用函数对象的形式

struct del{

void operator()(int \*p){

delete p;

}

};

int main() {

std::unique\_ptr<int> P1;

std::unique\_ptr<int, del> P2(new int(10), del());

}

unique\_ptr智能指针的一些成员方法

1、reset()

2、swap()

3、get()

4、bool()

5、get\_deleter()

6、release()

struct del{

int number;

void operator()(int \*p){

delete []p;

}

void print(){

std::cout<<"P2P number = "<<number<<std::endl;

}

};

int main() {

std::unique\_ptr<int> P1;

std::unique\_ptr<int, del> P2(new int(10), del());

// 获取当前 unique\_ptr 指针释放堆内存空间所用的规则。

del &P2P = P2.get\_deleter();

P2P.print();

// 释放当前 unique\_ptr 指针对所指堆内存的所有权，但该存储空间并不会被销毁。

P2.release();

}