1. auto\_ptr: in the <memory>

auto\_ptr<B> pInt1(new B(2)); /\*a B class auto\_ptr\*/

if (pInt1.get() != NULL)  /\*if the auto\_ptr binds with a pointer it returns the raw pointer that is new B(2)\*/

{

pInt1->func(); /\*access class member in three ways\*/

(\*pInt1).func();

pInt1.get()->func();

}

auto\_ptr<B> pInt2 = pInt1; /\*auto\_ptr can be owned by only one user，the pInt1 give it to pInt2\*/

/\*It is better not used the copy operation, it can’t be used as function parameter and in vector\*/

if (pInt1.get() != NULL);   /\*get() can’t detect copy case and lead to get a wild pointer\*/

{

//pInt1->func(); /\*it will lead a core dump for wild pointer\*/

}

if (pInt2.get() != NULL);

{

pInt2->func();

}

B\* t = pInt2.release();/\*release() do not free the memory just return the raw pointer and the ownship of pInt2, so now we need delete by my self\*/

if (pInt2.get() != NULL) /\*get() can detect release \*/

{

pInt2->func();

}

delete t;

auto\_ptr<B> pInt3(new B(4));

pInt3->func();

pInt3.reset(new B(5)); /\*giveup the owership of the poninter and delete it, set to the new one if there is a parameter\*/

if (pInt3.get() != NULL) /\* get() can detect reset \*/

{

pInt3->func();

}

conclusion:

a. auto\_ptr can be owned by only one user. With copy operation the old one lose the right to own it and can not be used.

b. It is better not used the assignment operation, so auto\_ptr can’t be used as function parameter and in vector.

c. release() do not free the memory just return the raw pointer and give up the ownship of the pointer, so the raw pointer need be deleted later

d. reset() giveup the owership of the poninter and delete it, set to the new one if there is a parameter

e. get() gets the raw pointer that the auto\_ptr bind to. if not it will return NULL. After copy operator the get() return a wild pointer. After reset or release it return NULL or new one.

2. unique\_ptr: in <tr1/memory> base on sdt::tr1 namespace

std::unique\_ptr<B> pInt1(new B(2));

                pInt1->func();

                /\*It explicitly prohibits copy operation\*/

                //unique\_ptr<B> pInt = pInt1;

                unique\_ptr<B> p2 = move(pInt1);//move the owership to p2

                assert(pInt1 == NULL);

                p2->func();

conclusion:

a. It explicitly prohibits copy operation, but can transfer the ownership through move()

b. There are also some operations. such as get(), reset(), release().

c. It provides == operation for scoped\_ptr. If it do not bind with any pointer, the unique\_ptr is equal to NULL

3. scoped\_ptr: base in boost. It explicitly prohibits copy operation

scoped\_ptr<B> pInt1(new B(2));

pInt1->func();

/\*It explicitly prohibits copy operation\*/

//scoped\_ptr<B> pInt = pInt1;

conclusion:

a. It explicitly prohibits copy operation

b. There are also some operations. such as get(), reset(), release().

c. It provides == operation for scoped\_ptr. If it do not bind with any pointer, the scoped\_ptr is equal to NULL

4. shared\_ptr: in <tr1/memory> base on sdt::tr1 namespace

shared\_ptr<B> pInt2(new B(2));

assert(pInt2.use\_count() == 1);   /\*it provides a good copy operation and a count function\*/

{

shared\_ptr<B> pInt1(new B(1));

assert(pInt1.use\_count() == 1);

pInt2->PrintSomething();

pInt2 = pInt1;

assert(pInt2 == pInt1);

pInt2->PrintSomething();

assert(pInt1.use\_count() == 2);

assert(pInt2.use\_count() == 2);

}

pInt2.reset(new B(100));  //destruct B(1), and create B(100) there is no release() in the share\_ptr

assert(pInt2.use\_count() == 1);

{

shared\_ptr<B> temp1(new B(3));

assert(temp1.use\_count() == 1);

shared\_ptr<B> temp2(temp1);

assert(temp2.use\_count() == 2);

assert(temp1.use\_count() == 2);

}

// int\* p = new int(14);

// shared\_ptr<int> temp1(p); //don't bind a raw pointer to diffirent shared\_ptr, it will lead to miss operation, delete the pointer p twice

//shared\_ptr<int> temp2(p);

conclusion:

a. it provides a good copy operation adn a count function

b. There are also some operations. such as get(), reset(), ==operation(), but no release()

c. It provides a count function use\_count().

5. weak\_ptr: in <tr1/memory> base on sdt::tr1 namespace

weak\_ptr<B> pIntw1;

{

shared\_ptr<B> pInts(new B(2));

pIntw1 = pInts;/\*weak\_ptr only can be constructed by shared\_ptr or weak\_ptr\*/

pInts->func();

assert(pInts.use\_count() == 1); // only 1

weak\_ptr<B> pIntw2 = pIntw1;

/\*pIntw->func();: weak ptr can not do this\*/

assert(pIntw1.use\_count() == 1); // only 1

shared\_ptr<B> pInts1 = pIntw2.lock();// enhance the weak to it's shared

assert(pIntw1.use\_count() == 2);  // only 1

}

assert(pIntw1.expired());

shared\_ptr<B> pInt = pIntw1.lock();/\* expired and can now enhance it\*/

asset(pInt.get() == NULL);

conclusion:

a. weak\_ptr is constructed base on shared\_ptr or weak\_ptr.

b. It is an observer for the shared\_ptr and do not add the count and can’t access the raw pinter by itself

c. weak\_ptr can be enhanced to shared\_ptr by lock(). if the weak\_ptr pointer is expired, it can not enhanced to shared\_ptr.

d. weak\_ptr usually is used to class member not shared\_ptr. see the example

class parent;

class children;

typedef shared\_ptr<parent> parent\_ptr;

typedef shared\_ptr<children> children\_ptr;

typedef weak\_ptr<parent> weak\_ptrs;

class parent

{

public:

parent() {cout <<"creating parent\n"; }

~parent() {cout <<"destroying parent\n"; }

public:

children\_ptr children;

};

class children

{

public:

children() { cout <<"creating children\n"; }

~children() { cout <<"destroying children\n"; }

public:

weak\_ptrs  parent;  //mostly member ptr should be weak one's, or the target of ptr will not be destroy until this class traget is destory. The target of ptr will nerver be destroy until there is no shared ptr point to it

//parent\_ptr parent;  /\*it leads to undestroying both parent and child\*/

};

weak\_ptr<parent> p;

weak\_ptr<children> c;

void test()

{

parent\_ptr father(new parent());

children\_ptr son(new children());

p = father;

c = son;

father->children = son;

son->parent = father;

}

int main()

{

cout<<"begin test...\n";

test();

cout<<"end test.\n";

printf("p count %ld\n", p.use\_count());

printf("c count %ld\n", c.use\_count());

return 0;

}