C++:智能指针-TR1 的 shared_ptr 和 weak_ptr 使用介绍

shared_ptr: Based on a reference counter model, with the counter incremented each time a new shared pointer object points to the resource, and decremented when the object's destructor executes; when the counter gets to 0, the resource is released. This pointer is copy constructable and assignable; this makes it usable in STL containers. Moreover, the shared pointer works with polymorphic types and incomplete types. Its major drawback is the impossibility to detect cyclic dependencies, in which case the resources never get released (for example, a tree with nodes having (shared) pointers to children but also to the parent, in which case the parent and the children are referencing each other, in a cycle). To fix this issue, a second smart pointer was created:

weak_ptr: Points to a resource referred by a shared pointer, but does not participate in reference counting. When the counters gets to 0, the resource is released, regardless the number of weak pointers referring it; all these pointers are marked as invalid.

A sharted_ptr object has the ownership of an object if

- It was constructed with a pointer to that resource;
- It was constructed from a shared_ptr object that owns that resource;
- It was constructed from a weak_ptr object that points to that resource;

Ownership of that resource was assigned to it, either with shared_ptr

:operator= or by calling the member function shared_ptr::reset().

Can create a new shared_ptr from

- A pointer to any type T (including const T), having the possibility of specifying a deleter for the pointed resource;
- Another shared_ptr object;
- A weak_ptr object;
- An auto_ptr object;

目录

[隐藏]

1 1.get() 返回对象指针；use_count() 返回对象的引用计数

2 2.相比较 shared_ptr,auto_ptr 在赋值与别人后，是放弃对象引用的。

3 3.在 shared_ptr 构造函数中，行参指定构造对象和析构对象的函数

4 4.get() 返回对象指针,使用->调用成员函数

5 5.get() 返回对象指针,if 判断是否为 null

6 6.swap() 交换两个 shared_ptr 所指向的对象

7 7.使用等号赋值

8 8.unique() 判断当前对象的引用计数==1?

9 9.reset() 清空当前 shared 指针，并将所有基于该指针创建的 shared 指针的引用计数减 1

10 10.对引用计数的理解,在容器中使用 shared_ptr

11 11.多态情况下的 shared 指针使用（声明基类句柄，创建子类对象）

12 12.dynamic_cast,使用 dynamic_pointer_cast 将基类向下转型为子类

13 13.static cast，使用 static_pointer_cast 将 void 转型为 char，观察引用计数的变化

14 14.const cast,如果声明 std::tr1::shared_ptr<const int> csp，可以声明 std::tr1::shared_ptr<int> sp = std::tr1::const_pointer_cast<int>(csp);

15 15.weak_ptr 的 lock() 类似于 shared_ptr 的 get()

16 16.一个使用 shared_ptr 和 weak_ptr 的二叉树数据结构示例

[编辑]1.get() 返回对象指针；use_count() 返回对象的引用计数

```cpp
#include <iostream>
#include <tr1/memory>

class Foo
{
public:
    void print()
    {
        std::cout << " foo::print" << std::endl;
    }
};

/* When sp2 is created, sp1 increments the reference counter.
 * When the two shared pointer objects get out of scope, the last
 * one that is destroyed will release the resource.
 *
 * output:
 * foo::print
 * sp1 pointer: 0x90a7008
 * foo::print
 * sp1 pointer: 0x90a7008
 * sp2 pointer: 0x90a7008
 * counter sp1: 2
 * counter sp2: 2
 */
int main()
{
    std::tr1::shared_ptr<Foo> sp1(new Foo);
    sp1->print();
    std::cout << "sp1 pointer: " << sp1.get() << std::endl;

    std::tr1::shared_ptr<Foo> sp2(sp1);
    sp2->print();
    std::cout << "sp1 pointer: " << sp1.get() << std::endl;
    std::cout << "sp2 pointer: " << sp2.get() << std::endl;

    std::cout << "counter sp1: " << sp1.use_count() << std::endl;
    std::cout << "counter sp2: " << sp2.use_count() << std::endl;

    return 0;
}
```

[编辑]2.相比较 shared_ptr,auto_ptr 在赋值与别人后，是放弃对象引用的。

```
#include <iostream>
#include <tr1/memory>

class Foo
{
public:
    void print()
    {
        std::cout << " foo::print" << std::endl;
    }
};

/* The next sample shows a shared_ptr created from an auto_ptr object. The auto pointer gives up the ownership of the resource,
 * resetting its wrapped pointer to NULL.
 *
 * output:
 * foo::print
 * ap1 pointer: 0x99b8008
 * foo::print
 * ap1 pointer: 0
 * sp1 pointer: 0x99b8008
 */
int main()
{
    std::auto_ptr<Foo> ap1(new Foo);
    ap1->print();
    std::cout << "ap1 pointer: " << ap1.get() << std::endl;

    std::tr1::shared_ptr<Foo> sp1(ap1);  // 注意这里是 shared_ptr
    sp1->print();
    std::cout << "ap1 pointer: " << ap1.get() << std::endl;
    std::cout << "sp1 pointer: " << sp1.get() << std::endl;
    return 0;
}
```

[编辑]3.在 shared_ptr 构造函数中，行参指定构造对象和析构对象的函数

```
#include <iostream>
#include <tr1/memory>

class Foo
{
public:
    void print()
    {
        std::cout << " foo::print" << std::endl;
    }
};

class FooHandler
{
public:
    static Foo* alloc()
    {
        Foo* f = new Foo;
        std::cout << " a new foo was created" << std::endl;
        return f;
    }

    static void free(Foo* f)
    {
        delete f;
        std::cout << " foo destroyed" << std::endl;
    }
};

/*
 * Each time a new object is created or destroyed, a message is printed in the output
window (for simplicity, you will ignore the copy
 * construction or assignment). Function FooHandler::free can be provided as a delete
to the shared_ptr constructor. As a result,
 * when the resource is deleted a message is printed in the output window (you have
to run in debugger to see it).
 *
 * output:
 * a new foo was created
 * foo::print
 * foo destroyed
 */
int main()
{
    std::tr1::shared_ptr<Foo> ptr(FooHandler::alloc(), &FooHandler::free);
    ptr->print();

    return 0;
}
```

[编辑]4.get() 返回对象指针,使用->调用成员函数

```
#include <iostream>
#include <tr1/memory>

class Foo
{
public:
    void print()
    {
        std::cout << " foo::print" << std::endl;
    }
};

/*
 * Function get() returns the wrapped pointer to the resource (basically identical
to operator-> and available for compatibility
 * with auto_ptr).
 *
 * output:
 * foo::print
 */
int main()
{
    std::tr1::shared_ptr<Foo> sp(new Foo);
    Foo* f = sp.get();
    if (f)
        f->print();

    return 0;
}
```

[编辑]5.get() 返回对象指针,if 判断是否为 null

```
#include <iostream>
```

```
#include <tr1/memory>

/* Class shared_ptr defines a bool operator that allows shared pointers to be used
in boolean expressions.
 * With auto_ptr, that is not possible; you have to use function get() to access the
internal pointer and check it against NULL.
 */
class PtrUtil
{
public:
    static void is_empty(std::tr1::shared_ptr<std::string> ptr)
    {
        if (ptr)
            std::cout << "not empty" << std::endl;
        else
            std::cout << "is empty" << std::endl;
    }
};

/*
 * output:
 * is empty
 * not empty
 */
int main()
{
    std::tr1::shared_ptr<std::string> sp1;
    std::tr1::shared_ptr<std::string> sp2(new std::string("demo"));

    PtrUtil::is_empty(sp1);
    PtrUtil::is_empty(sp2);

    return 0;
}
```

[编辑]6.swap() 交换两个 shared_ptr 所指向的对象

```
#include <iostream>
#include <tr1/memory>

class PtrUtil
{
public:
    static void is_empty(std::tr1::shared_ptr<std::string> ptr)
    {
        if (ptr)
            std::cout << "not empty" << std::endl;
        else
            std::cout << "is empty" << std::endl;
    }
};

/* Method swap() : exchange the content of the shared pointers.
 *
 * output:
 * is empty
 * not empty
 * not empty
 * is empty
 */
int main()
{
    std::tr1::shared_ptr<std::string> sp1;
    std::tr1::shared_ptr<std::string> sp2(new std::string("demo"));

    PtrUtil::is_empty(sp1);
    PtrUtil::is_empty(sp2);

    sp1.swap(sp2);

    PtrUtil::is_empty(sp1);
    PtrUtil::is_empty(sp2);

    return 0;
}
```

[编辑]7.使用等号赋值

```
#include <iostream>
#include <tr1/memory>

/* operator= is overloaded so that a shared pointer can be assigned from another
shared_ptr or auto_ptr.
 *
 * output:
 * sp1 = 1
 * sp2 = 2
 * sp1 = 2
 */
int main()
```

```
{
    std::tr1::shared_ptr<int> sp1(new int(1));
    std::cout << "sp1 = " << *sp1 << std::endl;

    std::tr1::shared_ptr<int> sp2(new int(2));
    std::cout << "sp2 = " << *sp2 << std::endl;

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

    return 0;
}
```

[编辑]8.unique() 判断当前对象的引用计数==1？

```
#include <iostream>
#include <tr1/memory>

/* Method use_count() returns the number of references to the shared resource (pointed
by the current shared pointer object).
 * Method unique() indicates whether another shared pointed shares the ownership of
the same resource or not
 * (basically, it's identical to 1 == use_count()).
 *
 * output:
 * unique : true
 * counter : 1
 * unique : false
 * counter : 2
 */
int main()
{
    std::tr1::shared_ptr<std::string> sp1(new std::string("marius bancila"));
    std::cout << "unique : " << std::boolalpha << sp1.unique() << std::endl;
    std::cout << "counter : " << sp1.use_count() << std::endl;
    std::tr1::shared_ptr<std::string> sp2(sp1);
    std::cout << "unique : " << std::boolalpha << sp1.unique() << std::endl;
    std::cout << "counter : " << sp1.use_count() << std::endl;

    return 0;
}
```

[编辑]9.reset() 清空当前 shared 指针，并将所有基于该指针创建的 shared 指针的引用计数减 1

```cpp
#include <iostream>

#include <tr1/memory>


class Foo

{

public:

    void print()

    {

        std::cout << " foo::print" << std::endl;

    }

};


/*Function reset() decrements the shared reference counter. It then transforms the

shared pointer to an empty shared_ptr.

 *

 * output:

 * counter sp1: 1

 * counter sp1: 3

 * counter sp2: 3

 * counter sp3: 3

 * counter sp1: 0

 * counter sp2: 2

 * counter sp3: 2

 * counter sp1: 0

 * counter sp2: 0

 * counter sp3: 1

 */

int main()

{

    // a shared_ptr owns the resouce, counter is 1

    std::tr1::shared_ptr<Foo> sp1(new Foo);

    std::cout << "counter sp1: " << sp1.use_count() << std::endl;


    std::tr1::shared_ptr<Foo> sp2(sp1);

    std::tr1::shared_ptr<Foo> sp3(sp2);

    std::cout << "counter sp1: " << sp1.use_count() << std::endl;

    std::cout << "counter sp2: " << sp2.use_count() << std::endl;

    std::cout << "counter sp3: " << sp3.use_count() << std::endl;


    // first shared_ptr is reset, the counter decremented and the object becomes empty

    sp1.reset();

    std::cout << "counter sp1: " << sp1.use_count() << std::endl;

    std::cout << "counter sp2: " << sp2.use_count() << std::endl;

    std::cout << "counter sp3: " << sp3.use_count() << std::endl;
    sp2.reset();
    std::cout << "counter sp1: " << sp1.use_count() << std::endl;
    std::cout << "counter sp2: " << sp2.use_count() << std::endl;
    std::cout << "counter sp3: " << sp3.use_count() << std::endl;

    return 0;
}
```

[编辑]10.对引用计数的理解,在容器中使用 shared_ptr

```
#include <iostream>
#include <tr1/memory>
#include <vector>
#include <algorithm>

/* The following sample shows a vector of shared_ptr to int; a transformation is applied
on the elements of the vector,
 * doubling the value of the pointed objects.
 *
 * The program shows the reference counter to show that calling function double_it()
does not affect it, even though this function
 * returns a shared_ptr by value.
 */
std::tr1::shared_ptr<int> double_it(const std::tr1::shared_ptr<int>& sp)
{
    *sp *= 2;
    return sp;
}

/*
 * output:
 * initially
 * 1 (counter = 1)
 * 2 (counter = 1)
 * 3 (counter = 1)
 * after transformation
 * 2 (counter = 1)
 * 4 (counter = 1)
 * 6 (counter = 1)
 */
int main()
{
    std::vector<std::tr1::shared_ptr<int> > numbers;
    numbers.push_back(std::tr1::shared_ptr<int>(new int(1)));
    numbers.push_back(std::tr1::shared_ptr<int>(new int(2)));
    numbers.push_back(std::tr1::shared_ptr<int>(new int(3)));

    std::cout << "initially" << std::endl;
    for (std::vector<std::tr1::shared_ptr<int> >::const_iterator it =
            numbers.begin(); it != numbers.end(); ++it)
        std::cout << **it << " (counter = " << (*it).use_count() << ")"
                << std::endl;
    std::transform(numbers.begin(), numbers.end(), numbers.begin(), double_it);
    std::cout << "after transformation" << std::endl;
    for (std::vector<std::tr1::shared_ptr<int> >::const_iterator it =
            numbers.begin(); it != numbers.end(); ++it)
        std::cout << **it << " (counter = " << (*it).use_count() << ")"
                << std::endl;

    return 0;
}
```

[编辑]11.多态情况下的 shared 指针使用（声明基类句柄，创建子类对象）

```
#include <iostream>
#include <tr1/memory>
#include <vector>

/*shared_ptr can work with class hierarchies, so that shared<D> is convertible to
shared<B>, where D is a class (or struct) derived
 * from B. The following class hierarchy is used to demonstrate the concept.
 */
class Item
{
    std::string title_;
public:
    Item(const std::string& title) :
        title_(title)
    {
    }
    virtual ~Item()
    {
    }

    virtual std::string Description() const = 0;
    std::string Title() const
    {
        return title_;
```

```
   }
};

class Book: public Item
{
   int pages_;
public:
   Book(const std::string& title, int pages) :
      Item(title), pages_(pages)
   {
   }

   virtual std::string Description() const
   {
      return "Book: " + Title();
   }
   int Pages() const
   {
      return pages_;
   }
};

class DVD: public Item
{
   int tracks_;
public:
   DVD(const std::string& title, int tracks) :
      Item(title), tracks_(tracks)
   {
   }

   virtual std::string Description() const
   {
      return "DVD: " + Title();
   }
   int Tracks() const
   {
      return tracks_;
   }
};

/*
 * output:
 * Book: Effective STL
 * DVD: Left of the Middle
 */
int main()
{
    std::vector<std::tr1::shared_ptr<Item> > items;
    items.push_back(std::tr1::shared_ptr<Book>(new Book("Effective STL", 400)));
    items.push_back(std::tr1::shared_ptr<DVD>(new DVD("Left of the Middle", 14)));

    for (std::vector<std::tr1::shared_ptr<Item> >::const_iterator it =
            items.begin(); it != items.end(); ++it)
        std::cout << (*it)->Description() << std::endl;

    return 0;
}
```

[编辑]12.dynamic_cast,使用 dynamic_pointer_cast 将基类向下转型为子类

```
#include <iostream>
#include <tr1/memory>
#include <vector>

/*shared_ptr can work with class hierarchies, so that shared<D> is convertible to
shared<B>, where D is a class (or struct) derived
 * from B. The following class hierarchy is used to demonstrate the concept.
 */
class Item
{
    std::string title_;
public:
    Item(const std::string& title) :
        title_(title)
    {
    }
    virtual ~Item()
    {
    }

    virtual std::string Description() const = 0;
    std::string Title() const
    {
        return title_;
    }
};

class Book: public Item
```

```
{
   int pages_;
public:
   Book(const std::string& title, int pages) :
      Item(title), pages_(pages)
   {
   }

   virtual std::string Description() const
   {
      return "Book: " + Title();
   }
   int Pages() const
   {
      return pages_;
   }
};

class DVD: public Item
{
   int tracks_;
public:
   DVD(const std::string& title, int tracks) :
      Item(title), tracks_(tracks)
   {
   }

   virtual std::string Description() const
   {
      return "DVD: " + Title();
   }
   int Tracks() const
   {
      return tracks_;
   }
};

/* To convert back, from shared_ptr<B> to shared_ptr<D>, where D is a class (or structure)
derived from B,
 * you can use the cast function std::tr1::dynamic_pointer_cast.
 *
 * output:
 * spi counter: 1
 * Left of the Middle, 14 tracks
 * spi counter: 2
 * spb counter: 0
 * spd counter: 2
 */
int main()
{
   std::tr1::shared_ptr<Item> spi(new DVD("Left of the Middle", 14));
   std::cout << "spi counter: " << spi.use_count() << std::endl;
   std::tr1::shared_ptr<Book> spb = std::tr1::dynamic_pointer_cast<Book>(spi);
   if (spb)
      std::cout << spb->Title() << ", " << spb->Pages() << " pages" << std::endl;

   std::tr1::shared_ptr<DVD> spd = std::tr1::dynamic_pointer_cast<DVD>(spi);
   if (spd)
      std::cout << spd->Title() << ", " << spd->Tracks() << " tracks"   << std::endl;

   std::cout << "spi counter: " << spi.use_count() << std::endl;
   std::cout << "spb counter: " << spb.use_count() << std::endl;
   std::cout << "spd counter: " << spd.use_count() << std::endl;

   return 0;
}
```

[编辑]13.static cast，使用 static_pointer_cast 将 void 转型为 char，观察引用计数的变化

```
#include <iostream>
#include <tr1/memory>
#include <vector>

/* A second cast function is std::tr1::static_pointer_cast. It returns an empty
shared_ptr if the original object is empty,
 * or a shared_ptr<T> object that owns the resource that is owned by the original object.
The expression static_cast<T*>(r.get())
 * must be valid.
 *
 * In the next sample, a vector holds shared_ptr to void. The first element is statically
cast to shared_ptr<char>.
 * The cast is valid as long as the source is not empty, regardless of whether the
types are compatible or not.
 *
 * output:
 * after creating the shared pointer
 * -1    sp1 counter: 1
 * after adding to the vector
 * -2   sp1 counter: 2
 * A
 * after casting
 * -3   sp1 counter: 3
 * -4   spc counter: 3
 */
int main()
{
   std::vector<std::tr1::shared_ptr<void> > items;
   std::tr1::shared_ptr<char> sp1(new char('A'));
   std::tr1::shared_ptr<short> sp2(new short(66));
   std::cout << "after creating the shared pointer" << std::endl;
   std::cout << "-1   sp1 counter: " << sp1.use_count() << std::endl;
   items.push_back(sp1);
   items.push_back(sp2);
   std::cout << "after adding to the vector" << std::endl;
   std::cout << "-2   sp1 counter: " << sp1.use_count() << std::endl;
   std::tr1::shared_ptr<char> spc =
std::tr1::static_pointer_cast<char>(*(items.begin()));
   if (spc)
      std::cout << *spc << std::endl;
   std::cout << "after casting" << std::endl;
   std::cout << "-3   sp1 counter: " << sp1.use_count() << std::endl;
   std::cout << "-4   spc counter: " << spc.use_count() << std::endl;

   return 0;
}
```

[编辑]14.const cast,如果声明 std::tr1::shared_ptr<const int> csp，可以声明 std::tr1::shared_ptr<int> sp = std::tr1::const_pointer_cast<int>(csp);

```
#include <iostream>
#include <tr1/memory>

/* To modify the value of the pointer object the const specifier must be removed. This
is shown below.
 *
 * output:
 * csp counter: 1
 * 15
 * 15
 * csp counter: 2
 * sp counter: 2
 */
```

```
int main()
{
    std::tr1::shared_ptr<const int> csp(new int(5));
    std::cout << "csp counter: " << csp.use_count() << std::endl;

    std::tr1::shared_ptr<int> sp = std::tr1::const_pointer_cast<int>(csp);
    *sp += 10;

    std::cout << *csp << std::endl;
    std::cout << *sp << std::endl;

    std::cout << "csp counter: " << csp.use_count() << std::endl;
    std::cout << "sp counter: " << sp.use_count() << std::endl;

    return 0;
}
```

[编辑]15.weak_ptr 的 lock() 类似于 shared_ptr 的 get()

```
#include <iostream>
#include <tr1/memory>

/* The major weakness of shared_ptr is that it cannot detect cyclic dependencies. In
this case, the reference counter is incremented
 * more than it should actually be, so that the resources are no longer released when
the shared pointer objects go out of scope.
 * To fix this problem, a second smart pointer was created, weak_ptr, that points to
a resource owned by a shared_ptr but does not
 * affect the reference counter; it is a "weak reference." When the last shared_ptr
that owns the resource referred by a weak_ptr,
 * the resource is released and the weak pointer is marked as invalid. To check whether
a weak_ptr is valid or not, you can use
 * function expired() that returns true if the pointer was marked as invalid.
 */

/* Even though function get() (that provides direct access to the wrapped pointer)
is available, it's not recommended to use it even
 * in single-threaded applications. The safe alternative is function lock() that returns
a shread_ptr sharing the resource pointed by
 * the weak pointer.*/
void show(const std::tr1::weak_ptr<int>& wp)
{
    std::tr1::shared_ptr<int> sp = wp.lock();
    std::cout << *sp << std::endl;
}
```

```
/*
 * output:
 * 44
 * expired : true
 */
int main()
{
    std::tr1::weak_ptr<int> wp;
    {
        std::tr1::shared_ptr<int> sp(new int(44));
        wp = sp;
        show(wp);
    }
    std::cout << "expired : " << std::boolalpha << wp.expired() << std::endl;

    return 0;
}
```

[编辑]16.一个使用 shared_ptr 和 weak_ptr 的二叉树数据结构示例

```
#include <iostream>
#include <tr1/memory>

/* The following sample shows such a tree, but uses a weak_ptr to solve the cyclic
dependency.*/
class Node
{
    std::string value_;
    std::tr1::shared_ptr<Node> left_;
    std::tr1::shared_ptr<Node> right_;
    std::tr1::weak_ptr<Node> parent_;

public:
    Node(const std::string value) :
        value_(value)
    {
    }

    std::string Value() const
    {
        return value_;
    }
    std::tr1::shared_ptr<Node> Left() const
    {
        return left_;
    }
    std::tr1::shared_ptr<Node> Right() const
    {
        return right_;
    }
    std::tr1::weak_ptr<Node> Parent() const
    {
        return parent_;
    }

    void SetParent(std::tr1::shared_ptr<Node> node)
    {
        parent_.reset();
        parent_ = node;
    }

    void SetLeft(std::tr1::shared_ptr<Node> node)
    {
        left_.reset();
        left_ = node;
    }

    void SetRight(std::tr1::shared_ptr<Node> node)
    {
        right_.reset();
        right_ = node;
    }
};

std::string path(const std::tr1::shared_ptr<Node>& item)
{
    std::tr1::weak_ptr<Node> wparent = item->Parent();
    std::tr1::shared_ptr<Node> sparent = wparent.lock();

    if (sparent)
    {
        return path(sparent) + "\\" + item->Value();
    }

    return item->Value();
}

/*
```

```
 * output:
 * C:\dir1\dir11
 */
int main()
{
    std::tr1::shared_ptr<Node> root(new Node("C:"));

    std::tr1::shared_ptr<Node> child1(new Node("dir1"));
    std::tr1::shared_ptr<Node> child2(new Node("dir2"));

    root->SetLeft(child1);
    child1->SetParent(root);

    root->SetRight(child2);
    child2->SetParent(root);

    std::tr1::shared_ptr<Node> child11(new Node("dir11"));

    child1->SetLeft(child11);
    child11->SetParent(child1);

    std::cout << "path: " << path(child11) << std::endl;

    return 0;
```