smart pointers (boost) explained

What is the difference between the following set of pointers? When do you use each pointer in production code, if at all?

Examples would be appreciated!

1. `scoped_ptr`
2. `shared_ptr`
3. `weak_ptr`
4. `intrusive_ptr`

Do you use boost in production code?

c++ boost smart-pointers

edited Apr 1 '16 at 17:18
Null
1,455 ● 6 ● 17 ● 25

asked Feb 20 '09 at 14:42
Sasha

4 Answers

Basic properties of smart pointers

It's easy when you have properties that you can assign each smart pointer. There are three important properties.

- **no ownership at all**
- **transfer of ownership**
- **share of ownership**

The first means that a smart pointer cannot delete the object, because it doesn't own it. The second means that only one smart pointer can ever point to the same object at the same time. If the smart pointer is to be returned from functions, the ownership is transferred to the returned smart pointer, for example.

The third means that multiple smart pointers can point to the same object at the same time. This applies to a *raw pointer* too, however raw pointers lack an important feature: They do not define whether they are *owning* or not. A share of ownership smart pointer will delete the object if every owner gives up the object. This behavior happens to be needed often, so shared owning smart pointers are widely spread.

Some owning smart pointers support neither the second nor the third. They can therefore not be returned from functions or passed somewhere else. Which is most suitable for `RAII` purposes where the smart pointer is kept local and is just created so it frees an object after it goes out of scope.

Share of ownership can be implemented by having a copy constructor. This naturally copies a smart pointer and both the copy and the original will reference the same object. Transfer of ownership cannot really be implemented in C++ currently, because there are no means to transfer something from one object to another supported by the language: If you try to return an object from a function, what is happening is that the object is copied. So a smart pointer that implements transfer of ownership has to use the copy constructor to implement that transfer of ownership. However, this in turn breaks its usage in containers, because requirements state a certain behavior of the copy constructor of elements of containers which is incompatible with this so-called "moving constructor" behavior of these smart pointers.

C++1x provides native support for transfer-of-ownership by introducing so-called "move constructors" and "move assignment operators". It also comes with such a transfer-of-ownership smart pointer called `unique_ptr`.

Categorizing smart pointers

`scoped_ptr` is a smart pointer that is neither transferable nor sharable. It's just usable if you locally need to allocate memory, but be sure it's freed again when it goes out of scope. But it can still be

`shared_ptr` is a smart pointer that shares ownership (third kind above). It is reference counted so it can see when the last copy of it goes out of scope and then it frees the object managed.

`weak_ptr` is a non-owning smart pointer. It is used to reference a managed object (managed by a shared_ptr) without adding a reference count. Normally, you would need to get the raw pointer out of the shared_ptr and copy that around. But that would not be safe, as you would not have a way to check when the object was actually deleted. So, weak_ptr provides means by referencing an object managed by shared_ptr. If you need to access the object, you can lock the management of it (to avoid that in another thread a shared_ptr frees it while you use the object) and then use it. If the weak_ptr points to an object already deleted, it will notice you by throwing an exception. Using weak_ptr is most beneficial when you have a cyclic reference: Reference counting cannot easily cope with such a situation.

`intrusive_ptr` is like a shared_ptr but it does not keep the reference count in a shared_ptr but leaves incrementing/decrementing the count to some helper functions that need to be defined by the object that is managed. This has the advantage that an already referenced object (which has a reference count incremented by an external reference counting mechanism) can be stuffed into an intrusive_ptr - because the reference count is not anymore internal to the smart pointer, but the smart pointer uses an existing reference counting mechanism.

`unique_ptr` is a transfer of ownership pointer. You cannot copy it, but you can move it by using C++1x's move constructors:

```
unique_ptr<type> p(new type);
unique_ptr<type> q(p); // not legal!
unique_ptr<type> r(move(p)); // legal. p is now empty, but r owns the object
unique_ptr<type> s(function_returning_a_unique_ptr()); // legal!
```

This is the semantic that std::auto_ptr obeys, but because of missing native support for moving, it fails to provide them without pitfalls. unique_ptr will automatically steal resources from a temporary other unique_ptr which is one of the key features of move semantics. auto_ptr will be deprecated in the next C++ Standard release in favor of unique_ptr. C++1x will also allow stuffing objects that are only movable but not copyable into containers. So you can stuff unique_ptr's into a vector for example. I'll stop here and reference you to a fine article about this if you want to read more about this.

edited Dec 4 '11 at 14:20

answered Feb 20 '09 at 15:19

Johannes Schaub - litb

385k ● 89 ● 740 ● 1098

26 very educational, a great read. +1. – Doug T. Feb 20 '09 at 15:47

3 thanks for the praise dude. i appreciate it so you're going to get +1 now too :p – Johannes Schaub - litb Feb 20 '09 at 16:02

@litb: I've a doubt in "transfer of ownership"; I do agree there is no *real* transfer of ownership amongst objects in C++03, but for smart pointers can't this be done, by the *destructive copy* mechanism stated here informit.com/articles/article.aspx?p=31529&seqNum=5. – legends2k Mar 18 '10 at 18:12

3 fantastic answer. Note: `auto_ptr` is already deprecated (C++11). – DaddyM Jan 5 '12 at 12:37

2 "this in turn breaks its usage in containers, because requirements state a certain behavior of the copy constructor of elements of containers which is incompatible with this so-called "moving constructor" behavior of these smart pointers." Didn't get that part. – Raja Oct 17 '13 at 7:06

---

scoped_ptr is the simplest. When it goes out of scope, it is destroyed. The following code is illegal (scoped_ptrs are non-copyable) but will illustrate a point:

```
std::vector< scoped_ptr<T> > tPtrVec;
{
     scoped_ptr<T> tPtr(new T());
     tPtrVec.push_back(tPtr);
     // raw T* is freed
}
tPtrVec[0]->DoSomething(); // accessing freed memory
```

shared_ptr is reference counted. Every time a copy or assignment occurs, the reference count is incremented. Every time an instance's destructor is fired, the reference count for the raw T* is decremented. Once it is 0, the pointer is freed.

```
std::vector< shared_ptr<T> > tPtrVec;
{
     shared_ptr<T> tPtr(new T());
     // This copy to tPtrVec.push_back and ultimately to the vector storage
     // causes the reference count to go from 1->2
     tPtrVec.push_back(tPtr);
     // num references to T goes from 2->1 on the destruction of tPtr
}
tPtrVec[0]->DoSomething(); // raw T* still exists, so this is safe
```

weak_ptr is a weak-reference to a shared pointer that requires you to check to see if the pointed-to shared_ptr is still around

```
std::vector< weak_ptr<T> > tPtrVec;
{
     shared_ptr<T> tPtr(new T());
     tPtrVec.push_back(tPtr);
     // num references to T goes from 1->0
}
shared_ptr<T> tPtrAccessed =  tPtrVec[0].lock();
if (tPtrAccessed[0].get() == 0)
{
     cout << "Raw T* was freed, can't access it"
}
else
{
     tPtrVec[0]->DoSomething(); // raw
}
```

intrusive_ptr is typically used when there is a 3rd party smart ptr you must use. It will call a free function to add and decrement the reference count.See the link to boost documentation for more info.

edited Jun 5 '14 at 9:58

Maarten Bodewes
55.1k ● 9 ● 64 ● 152

answered Feb 20 '09 at 14:51

Doug T.
45.5k ● 19 ● 104 ● 180

The examples helped – Rajeshwar Mar 20 '13 at 5:34

isnt `if (tPtrAccessed[0].get() == 0)` suppose to be `if (tPtrAccessed.get() == 0)` ? – Rajeshwar Nov 23 '14 at 4:14

Don't overlook `boost::ptr_container` in any survey of boost smart pointers. They can be invaluable in situations where a e.g `std::vector<boost::shared_ptr<T> >` would be too slow.

answered Feb 20 '09 at 15:06

timday
19.3k ● 7 ● 56 ● 115

Actually, last time I tried it, benchmarking showed the performance gap had closed significantly since I originally wrote this, at least on typical PC HW! The more efficient ptr_container approach may still have some advantages in niche use-cases though. – timday Aug 8 '16 at 12:21

I second the advice about looking at the documentation. It is not as scary as it seems. And few short hints:

- `scoped_ptr` - a pointer automatically deleted when it goes out of scope. Note - no assignment possible, but introduces no overhead
- `intrusive_ptr` - reference counting pointer with no overhead of `smart_ptr`. However the object itself stores the reference count
- `weak_ptr` - works together with `shared_ptr` to deal with the situations resulting in circular dependencies (read the documentation, and search on google for nice picture ;)
- `shared_ptr` - the generic, most powerful (and heavyweight) of the smart pointers (from the ones offered by boost)
- There is also old `auto_ptr` , that ensures that the object to which it points gets destroyed automatically when control leaves a scope. However it has different copy semantics than the rest of the guys.
- `unique_ptr` - will come with C++0x

Response to edit: Yes

edited Feb 20 '09 at 16:04

answered Feb 20 '09 at 14:59

Anonymous
14.8k ● 2 ● 29 ● 56

2 what is `smart_ptr` ? – DaddyM Jan 5 '12 at 12:40

8 I came here because I found the boost documentation too scary. – Francois Botha May 29 '14 at 16:48

---

protected by Stefano Borini Jul 26 '12 at 14:02

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?