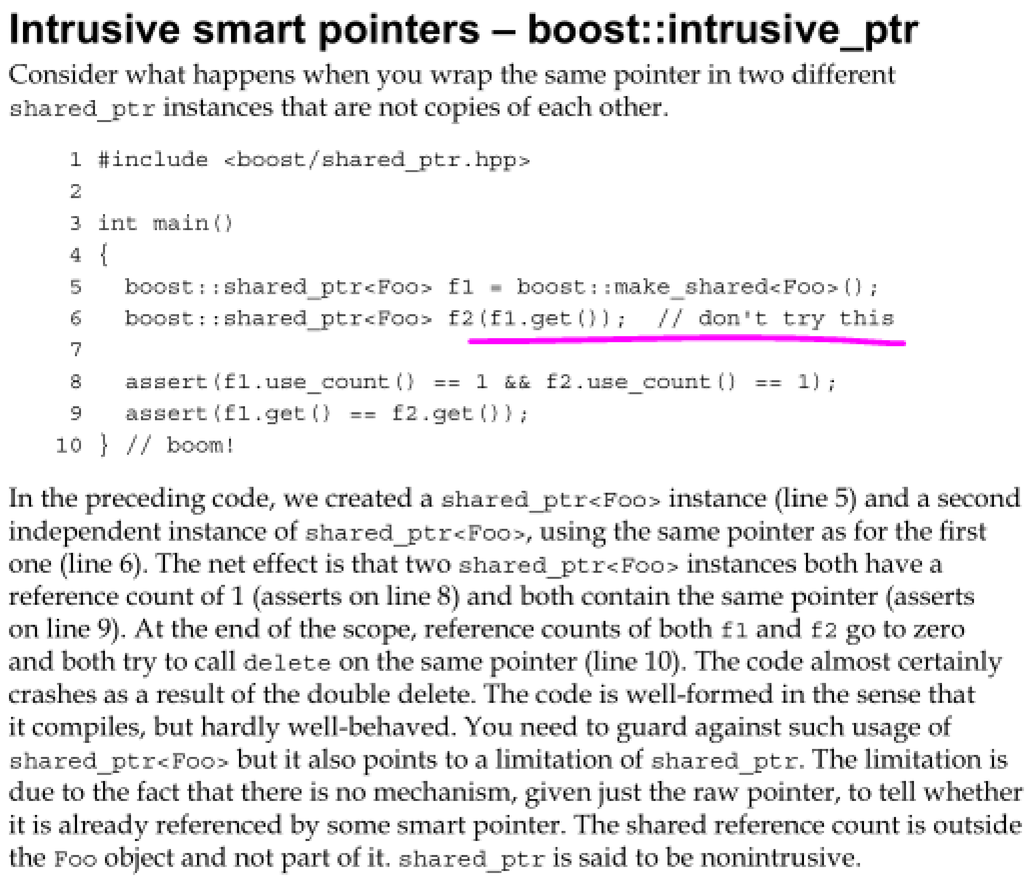
Boost Intrusive pointer

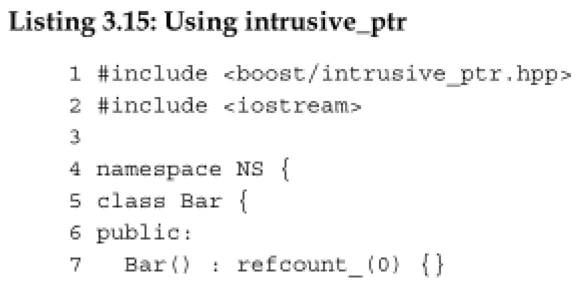


Intrusive smart pointers – boost::intrusive_ptr

Consider what happens when you wrap the same pointer in two different `shared_ptr` instances that are not copies of each other.

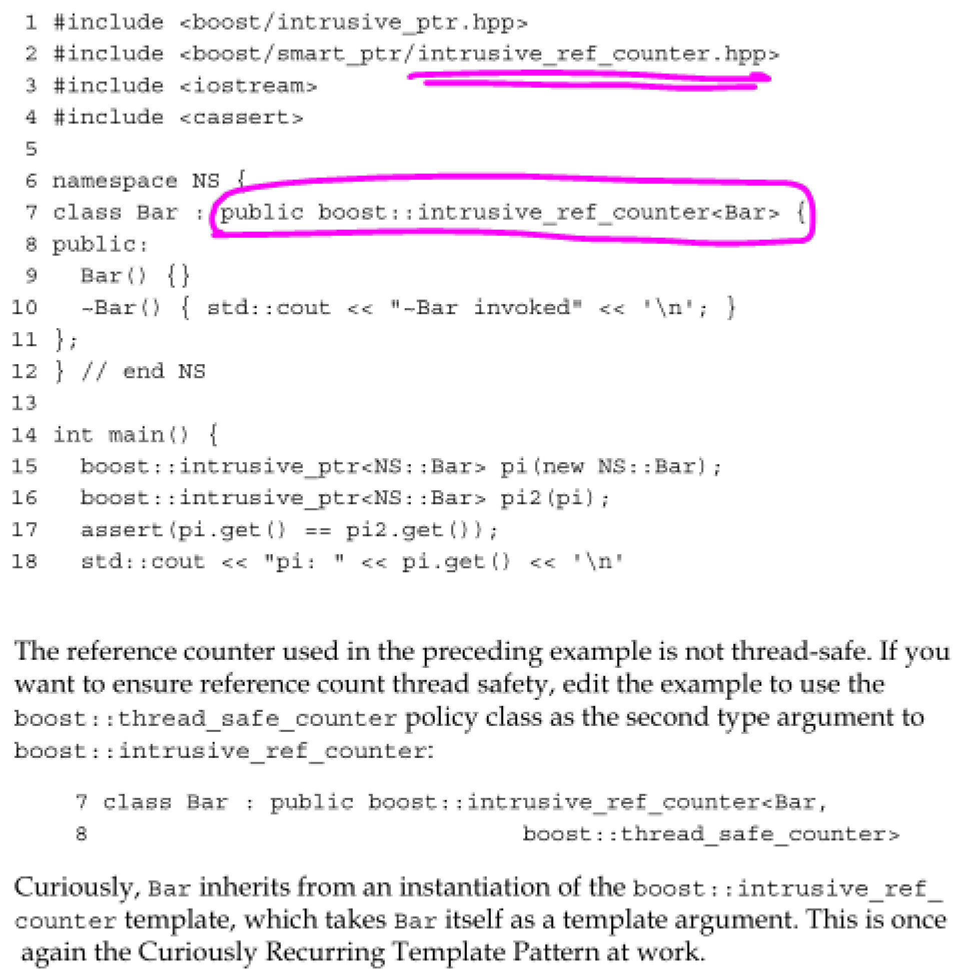
```
 1 #include <boost/shared_ptr.hpp>
 2
 3 int main()
 4 {
 5   boost::shared_ptr<Foo> f1 = boost::make_shared<Foo>();
 6   boost::shared_ptr<Foo> f2(f1.get());  // don't try this
 7
 8   assert(f1.use_count() == 1 && f2.use_count() == 1);
 9   assert(f1.get() == f2.get());
10 } // boom!
```

In the preceding code, we created a `shared_ptr<Foo>` instance (line 5) and a second independent instance of `shared_ptr<Foo>`, using the same pointer as for the first one (line 6). The net effect is that two `shared_ptr<Foo>` instances both have a reference count of 1 (asserts on line 8) and both contain the same pointer (asserts on line 9). At the end of the scope, reference counts of both `f1` and `f2` go to zero and both try to call `delete` on the same pointer (line 10). The code almost certainly crashes as a result of the double delete. The code is well-formed in the sense that it compiles, but hardly well-behaved. You need to guard against such usage of `shared_ptr<Foo>` but it also points to a limitation of `shared_ptr`. The limitation is due to the fact that there is no mechanism, given just the raw pointer, to tell whether it is already referenced by some smart pointer. The shared reference count is outside the `Foo` object and not part of it. `shared_ptr` is said to be nonintrusive.



Listing 3.15: Using intrusive_ptr

```
1 #include <boost/intrusive_ptr.hpp>
2 #include <iostream>
3
4 namespace NS {
5 class Bar {
6 public:
7   Bar() : refcount_(0) {}
```



```
#include <boost/intrusive_ptr.hpp>
#include <boost/smart_ptr/intrusive_ref_counter.hpp>
#include <iostream>
#include <cassert>

namespace NS {
class Bar : public boost::intrusive_ref_counter<Bar> {
public:
  Bar() {}
  ~Bar() { std::cout << "~Bar invoked" << '\n'; }
};
} // end NS

int main() {
  boost::intrusive_ptr<NS::Bar> pi(new NS::Bar);
  boost::intrusive_ptr<NS::Bar> pi2(pi);
  assert(pi.get() == pi2.get());
  std::cout << "pi: " << pi.get() << '\n'
```

The reference counter used in the preceding example is not thread-safe. If you want to ensure reference count thread safety, edit the example to use the `boost::thread_safe_counter` policy class as the second type argument to `boost::intrusive_ref_counter`:

```
class Bar : public boost::intrusive_ref_counter<Bar,
                                  boost::thread_safe_counter>
```

Curiously, `Bar` inherits from an instantiation of the `boost::intrusive_ref_counter` template, which takes `Bar` itself as a template argument. This is once again the Curiously Recurring Template Pattern at work.