



# Lecture 4

## *Boost Library 1 and 2*

Kenny Erleben

Department of Computer Science  
University of Copenhagen



# Shared Pointer 1

What are they good for?

- They store pointers to dynamically allocated objects
- and delete those objects at the right time

What are the benefits

- Handling shared ownership
- No leaking resources when throwing an exception
- Don't have to remember `delete`

Well it just get's easier!



# Shared Pointer 2

Basic idea is to keep a reference counter!

- null, means reference is zero
- “new” reference count equal to one
- copy-construction/assignment increment reference count
- deconstructor/reset (more on this later) decrement reference count by one
- if reference count equal to zero call “deleter” (more on this later)



# Shared Pointer 3

Subset of shared pointer interface

---

```
template<class T> class shared_ptr {
public:
    template<class Y> explicit shared_ptr(Y * p);
    template<class Y, class D> shared_ptr(Y * p, D d);
    ~shared_ptr(); // never throws
    shared_ptr(shared_ptr const & r); // never throws
    template<class Y> explicit shared_ptr(weak_ptr<Y> const & r);
    template<class Y> explicit shared_ptr(std::auto_ptr<Y> & r);
    shared_ptr & operator=(shared_ptr const & r); // never throws
    void reset(); // never throws
    T & operator*() const; // never throws
    T * operator->() const; // never throws
    T * get() const; // never throws
    bool unique() const; // never throws
    long use_count() const; // never throws
    operator unspecified-bool-type() const; // never throws
};
```

---



# The Basics

To create a new pointer write

---

```
class Dodah  
{  
    ...  
};
```

```
boost::shared_ptr<Dodah> ptr( new Dodah(...) );
```

---

Or

---

```
boost::shared_ptr<Dodah> ptr;  
ptr.reset( new Dodah(...) );
```

---

However `reset` is more tricky

---

```
ptr.reset();
```

---

Sets `ptr` to null.



# STL Containers 1

Say we have

---

```
std::list<Dodah *> A;  
Dodah * tmp = new Dodah(...);  
A.push_back(tmp);  
...  
Dodah * tmp = new Dodah(...);  
A.push_back(tmp);
```

---

Now if we want to clear the container we would have to write

---

```
for(std::list<Dodah*>::iterator it = A.begin(); it!=A.end(); ++it)  
    delete &(*it);  
A.clear();
```

---

Woah this is tedious (and error prone).



# STL Containers 2

Why not just write

---

```
std::list<boost::shared_ptr<Dodah> > A;  
boost::shared_ptr<Dodah> tmp( new Dodah(...) );  
A.push_back(tmp);  
...
```

---

Now we just write

---

```
A.clear();
```

---

Or let A run out of scope!



# Named pointer variables

## Programming Technique

- Always use a named smart pointer variable to hold the result of new

---

```
boost::shared_ptr<T> p(new Y);
```

---

- Nearly eliminates the possibility of memory leaks



# Named temporaries

Avoid using unnamed `shared_ptr` temporaries :

---

```
void f(boost::shared_ptr<int>, int);  
int g();
```

```
void ok()  
{  
    boost::shared_ptr<int> p(new int(2));  
    f(p, g());  
}
```

```
void bad()  
{  
    f(boost::shared_ptr<int>(new int(2)), g());  
}
```

---

- The function “ok” follows the guideline to the letter
- The function “bad” constructs the temporary, admitting the possibility of a memory leak



# this pointer

- Helper class template `enable_shared_from_this`

---

```
class A : public boost::enable_shared_from_this<A>
{
public:
    ...
    boost::shared_ptr<A> get_self()
    {
        return shared_from_this();
    }
};
```

---



# Pointers to static

In certain situations one may need to return a pointer to a statically allocated `x` instance.

- The solution is to use a custom deleter that does nothing

---

```
struct null_deleter
{
    void operator()(void const *) const
    {
    }
};

static X x;

shared_ptr<X> createX()
{
    shared_ptr<X> px(&x, null_deleter());
    return px;
}
```

---

- The same technique works for any object known to outlive the pointer.



# shared\_ptr from a raw pointer

Example:

---

```
void f(X * p)
{
    shared_ptr<X> px(???);
}
```

---

Inside `f`, we'd like to create a `shared_ptr` to `*p`.

- In the general case, this problem has no solution.
- One approach is to modify `f` to take a `shared_ptr`:

---

```
void f(shared_ptr<X> px);
```

---

- Or, if it's known that the `shared_ptr` created in `f` will never outlive the object, use a null deleter.



# Abstract classes for implementation hiding

## Abstract base class

---

```
class X
{
public:
    virtual void f() = 0;
    virtual void g() = 0;

protected:
    ~X() {}
};
```

---

Note the protected and nonvirtual destructor in the example above. The client code cannot, and does not need to, delete a pointer to X



# Abstract classes for implementation hiding

Now let us make an implementation

---

```
class Y : public X
{
private:
    Y(Y const &);
    Y & operator=(Y const &);

public:
    virtual void f()
    {
        // ...
    }

    virtual void g()
    {
        // ...
    }
};
```

---



# Abstract classes for implementation hiding

And finally we create a factory function

---

```
shared_ptr<X> createY()  
{  
    shared_ptr<X> px(new Y());  
    return px;  
}
```

---

- The allocation, construction, deallocation, and destruction details are captured at the point of construction.
- The client code cannot, and does not need to, delete a pointer to `x`; the `shared_ptr<X>` instance returned from `createY` will correctly call `~Y`.



# Cyclic Dependency Problem 1

An example of “direct” cyclic dependency

```
class A {
public:
    typedef boost::shared_ptr<A>    ptr_type;
public:
    char m_big_chunk[100*1024*1024];
    ptr_type m_ptr;
};
inline void test() {
    A::ptr_type ptr( new A());
    std::cout << ptr.use_count() << " == 0" << std::endl;
    ptr->m_ptr = ptr;
    std::cout << ptr.use_count() << " == 2" << std::endl;
    ptr.reset();
    std::cout << ptr.use_count() << " == 1?" << std::endl;
}
```

- Even when `ptr` is out of scope the application will still have allocated 100MB due to a dangling `m_ptr` pointer.



# Cyclic Dependency Problem 2

Output when using (VC7.1)

1 == 0

2 == 2

0 == 1?

Boost documentation

- States that reference count should be 1?
- However we do see “memory leak”.



# Cyclic Dependency Problem 3

Solution use `weak_ptr` to break cyclic dependence

---

```
class A
{
public:
    typedef boost::shared_ptr<A>    ptr_type;
    typedef boost::weak_ptr<A>     weak_ptr_type;
public:
    char m_big_chunk[100*1024*1024];
    weak_ptr_type m_ptr;
};
```

---

- Now no memory is eaten!



# Cyclic Dependency Problem 4

Cyclic dependencies are common:

---

```
class TreeNode;
typedef boost::shared_ptr<TreeNode> ptr_type;

class TreeNode
{
public:
    ptr_type m_parent;
    std::list<ptr_type> m_children;
};

class Tree
{
public:
    ptr_type m_root;
};
```

---

● How should we fix this?



# Cyclic Dependency Problem 5

One solution

---

```
class TreeNode;
typedef boost::shared_ptr<TreeNode> ptr_type;
typedef boost::weak_ptr<TreeNode>   weak_ptr_type;

class TreeNode
{
public:
    weak_ptr_type      m_parent;
    std::list<ptr_type> m_children;
};
```

---

- But what if children can be shared by parents?



# Weak Pointer 1

- Stores a “weak reference” to an object that’s already managed by a `shared_ptr`



# Weak Pointer 2

## Subset of Weak pointer interface

---

```
template<class T> class weak_ptr {  
    public:  
        template<class Y> weak_ptr(shared_ptr<Y> const & r);  
        weak_ptr(weak_ptr const & r);  
        template<class Y> weak_ptr(weak_ptr<Y> const & r);  
        ~weak_ptr();  
        weak_ptr & operator=(weak_ptr const & r);  
        template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);  
        template<class Y> weak_ptr & operator=(shared_ptr<Y> const & r);  
        long use_count() const;  
        bool expired() const;  
        shared_ptr<T> lock() const;  
        void reset();  
};
```

---



# Weak Pointer 3

## Example

---

```
shared_ptr<int> p(new int(5));  
weak_ptr<int> q(p);  
  
// some time later  
  
if(int * r = q.get())  
{  
    // use *r  
}
```

---

Imagine that after the if, but immediately before `r` is used, another thread executes the statement `p.reset()`. Now `r` is a dangling pointer.



# Weak Pointer 4

The solution: create a temporary `shared_ptr` from `q`:

---

```
shared_ptr<int> p(new int(5));  
weak_ptr<int> q(p);  
  
// some time later  
  
if(shared_ptr<int> r = q.lock())  
{  
    // use *r  
}
```

---

- Now `r` holds a reference to the object that was pointed by `q`
- By obtaining a `shared_ptr` to the object, we have effectively locked it against destruction



# Null pointer an error?

Consider

---

```
weak_ptr<Dodah> wp;  
shared_ptr<Dodah> p( wp );
```

---

- What happens?

Now

---

```
weak_ptr<Dodah> wp;  
shared_ptr<Dodah> p = wp.lock();
```

---

- What happens?



# Conversion

- `polymorphic_cast` automatically test for 0 and throw exception
- `polymorphic_downcast` uses `static_cast` in non-debug mode and `dynamic_cast` in debug mode
- `numeric_cast`
- `lexical_cast`



# Lexical Cast Example

---

```
template<typename T> std::string to_string( T const & arg)
{
    try
    {
        return boost::lexical_cast<std::string>( arg );
    }
    catch(boost::bad_lexical_cast & e)
    {
        return "";
    }
}
```

---