

# Embedded Software

---

Resource Handling - Improving our programs

# Challenge - Resource Handling

---

- Brainstorm - What do you consider a Resource Handling challenge?
  - ▶ What is a resource in sw “terms”?
  - ▶ In which situations do you foresee challenges?
- Team up 2 & 2 for 5 minutes

# The Challenge

---

- Design and implement a system that
  - ▶ That there is a garbage collection such that a resource is released when no one uses it (multi parties may share a resource)
  - ▶ It must be thread-safe
  - ▶ The resource itself is *NOT* protected, but its destruction must be

# Usage example

---

- Following usage scenario must work
  - ▶ Allocate resource in Thread A
  - ▶ Pass it to Thread B while keeping it in A
  - ▶ Relinquish usage in Thread B followed by Thread A
  - ▶ Resource is hereafter relinquished
- Could be ported to our Message Queues

# The exercises for this lecture

---

- RAII and SmartPointer
- Counted Smart Pointer
- Templated Counted Smart Pointer (OPTIONAL)
- `boost::shared_ptr<>`



Increased feature set

# Resource Handling

RAI for short repetition

# RAII

---



# RAII

---

- Managing memory is often a problem or challenge,
  - ▶ ensuring correctness - forgetting to deallocate (message etc.)
  - ▶ dealing with exceptions

# RAII

---

- Managing memory is often a problem or challenge,
  - ▶ ensuring correctness - forgetting to deallocate (message etc.)
  - ▶ dealing with exceptions
- ***Idiom : Resource Acquisition Is Initialization***
  - ▶ Wrap up all resources in their own object that handles their lifetime and put object on stack

# RAII Basic example

---

- Simple RAII class which also is a **SmartPointer**

```
template<typename T>
class RAII
{
public:
    explicit RAII( T* p = 0 ) : p_(p) {}

    ~RAII() { delete p_; }

    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }
private:
    T* p_;
};
```

- Usage example

```
{
    RAII<std::vector<int> > r( new std::vector<int>() );
    std::cout << r->size() << std::endl;
} // The std::vector<int> is automatically deallocated
```

# RAII Basic example

- Simple RAII class which also is a **SmartPointer**

```
template<typename T>
class RAII
{
public:
    explicit RAII( T* p = 0 ) : p_(p) {}

    ~RAII() { delete p_; }

    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }
private:
    T* p_;
};
```

RAII in effect

- Usage example

```
{
    RAII<std::vector<int> > r( new std::vector<int>() );
    std::cout << r->size() << std::endl;
} // The std::vector<int> is automatically deallocated
```

# RAII Basic example

- Simple RAII class which also is a **SmartPointer**

```
template<typename T>
class RAII
{
public:
    explicit RAII( T* p = 0 ) : p_(p) {}

    ~RAII() { delete p_; }

    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }
private:
    T* p_;
};
```

RAII in effect

Resource is deallocated

- Usage example

```
{
    RAII<std::vector<int> > r( new std::vector<int>() );
    std::cout << r->size() << std::endl;
} // The std::vector<int> is automatically deallocated
```

# RAII Basic example

- Simple RAII class which also is a **SmartPointer**

```
template<typename T>
class RAII
{
public:
    explicit RAII( T* p = 0 ) : p_(p) {}

    ~RAII() { delete p_; }

    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }

private:
    T* p_;
};
```

RAII in effect

Resource is deallocated

Accessing handled resource

- Usage example

```
{
    RAII<std::vector<int>> r( new std::vector<int>() );
    std::cout << r->size() << std::endl;
} // The std::vector<int> is automatically deallocated
```

`boost::shared_ptr<T>`

# Classic problem

---

- What is the problem here?

```
void f()
{
    Client* c = new Client;
    Data* d = acquireData(c);
    if(...)
        return;

    delete d;
    delete c;
}
```



# Classic problem

---

- What is the problem here?

```
void f()
{
    Client* c = new Client;
    Data* d = acquireData(c);
    if(...)
        return;

    delete d;
    delete c;
}
```

Exception what then?

# Classic problem

- What is the problem here?

```
void f()
{
    Client* c = new Client;
    Data* d = acquireData(c);
    if(...)
        return;

    delete d;
    delete c;
}
```

Exception what then?

Bad, where is the deallocation?

# Simple example

---

- `std::string` is automatically deallocated
  - Where exception is thrown
  - End of scope

```
void f()
{
    boost::shared_ptr<std::string> stringPtr(new std::string("Hello"));

    if (...) throw std::some_error("Bad number");

    std::cout << *stringPtr << std::endl;
}
```

Exit points

# Complex example

Stdout readout:			
	Pointer	Value	Use Count
1)	003756D0	42	1
2)	003756D0	43	2
3)	003756D0	43	3
4)	003756D0	43	1
5)	00000000	xx	0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

*Copy code and verify!*

# Complex example

Stdout readout:

	Pointer	Value	Use	Count
1)	003756D0	42		1
2)	003756D0	43		2
3)	003756D0	43		3
4)	003756D0	43		1
5)	00000000	xx		0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!

# Complex example

Stdout readout:

	Pointer	Value	Use	Count
1)	003756D0	42		1
2)	003756D0	43		2
3)	003756D0	43		3
4)	003756D0	43		1
5)	00000000	xx		0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!

# Complex example

Stdout readout:

	Pointer	Value	Use	Count
1)	003756D0	42		1
2)	003756D0	43		2
3)	003756D0	43		3
4)	003756D0	43		1
5)	00000000	xx		0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!

# Complex example

Stdout readout:			
	Pointer	Value	Use Count
1)	003756D0	42	1
2)	003756D0	43	2
3)	003756D0	43	3
4)	003756D0	43	1
5)	00000000	xx	0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!



# Complex example

Stdout readout:			
	Pointer	Value	Use Count
1)	003756D0	42	1
2)	003756D0	43	2
3)	003756D0	43	3
4)	003756D0	43	1
5)	00000000	xx	0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!

# Complex example

Stdout readout:

	Pointer	Value	Use	Count
1)	003756D0	42		1
2)	003756D0	43		2
3)	003756D0	43		3
4)	003756D0	43		1
5)	00000000	xx		0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!

# Complex example

Stdout readout:			
	Pointer	Value	Use Count
1)	003756D0	42	1
2)	003756D0	43	2
3)	003756D0	43	3
4)	003756D0	43	1
5)	00000000	xx	0

```
boost::shared_ptr<int> ee;
boost::shared_ptr<int> aa(new int(42));

1) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;
{
    boost::shared_ptr<int> bb(aa);
    ++(*bb);
2) std::cout << bb << "\\t" << *bb << "\\t" << bb.use_count() << std::endl;

    ee = bb;
3) std::cout << ee << "\\t" << *ee << "\\t" << ee.use_count() << std::endl;
}

ee.reset();
4) std::cout << aa << "\\t" << *aa << "\\t" << aa.use_count() << std::endl;

aa.reset();
5) std::cout << aa << "\\t" << "xx" << "\\t" << aa.use_count() << std::endl;
```

Copy code and verify!

# boost::shared\_ptr<T> - Properties

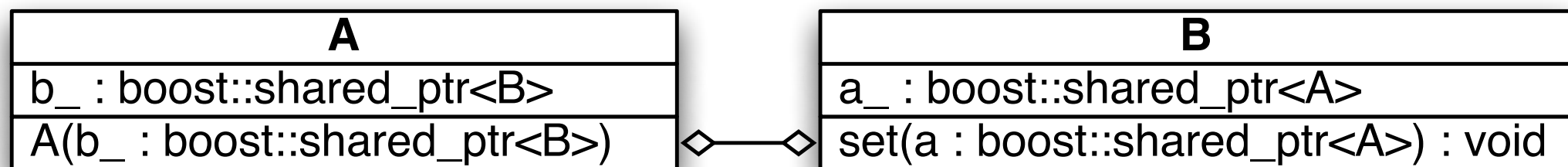
---

- A wrapping created with new T
- **Reference counted**
- **delete is guaranteed called when the last reference to the object dies or when the member function reset() is called**
- **The same thread safety as build-in types**
- **It is possible to supply an alternative functor that handles the “deallocation”**
- May be used in a container
- Implements the comparison operators
- Allows conversion from shared\_ptr<T> to shared\_ptr<U> if T\* implicit can be converted to U\* or if T is a specialization of U or U is void

`boost::weak_ptr<T>` - Cyclic dependencies

# Cyclic dependency - What?

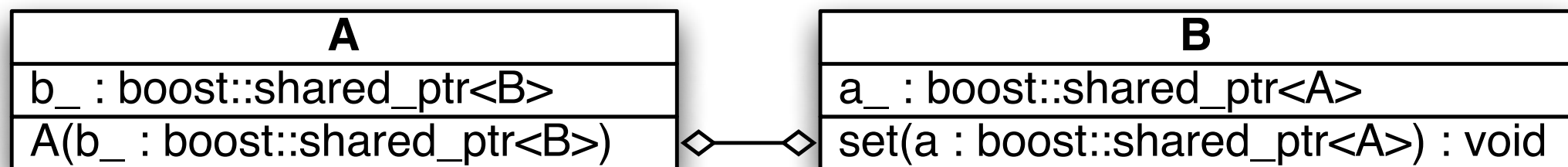
---



# Cyclic dependency - What?

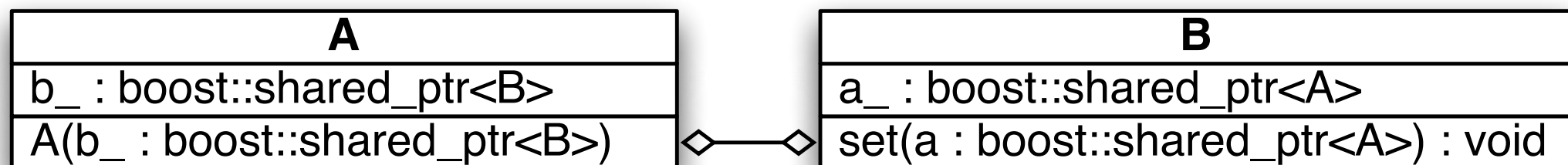
---

- Problem simple

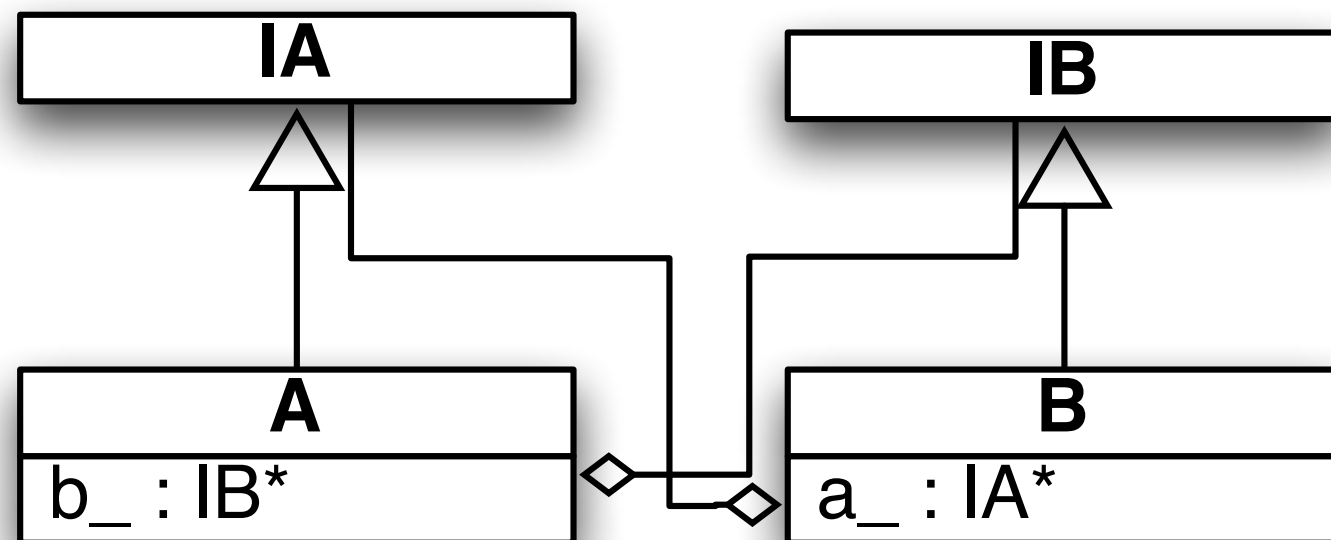


# Cyclic dependency - What?

- Problem simple



- More complex





# Usage – boost::weak\_ptr

---

- Breaking cyclic dependencies

```
struct A {  
    A(boost::shared_ptr<B> b)  
    : b_(b) {}  
  
    boost::shared_ptr<B> b_;  
};
```

```
struct B {  
    void set(boost::shared_ptr<A> a)  
    { a_ = a; }  
  
    boost::shared_ptr<A> a_;  
};
```

Test harness

```
boost::shared_ptr<B> tmpB(new B);  
boost::shared_ptr<A> tmpA(new A(tmpB));  
tmpB->set(tmpA); // Cyclic dependency introduced
```

# Usage – boost::weak\_ptr

- Breaking cyclic dependencies

```
struct A {  
    A(boost::shared_ptr<B> b)  
    : b_(b) {}  
  
    boost::shared_ptr<B> b_;  
};
```

```
struct B {  
    void set(boost::shared_ptr<A> a)  
    { a_ = a; }  
  
    boost::weak_ptr<A> a_;  
  
    void doStuff() {  
        boost::shared_ptr<A> a = a_.lock();  
        if(a) {  
            // Do stuff  
        }  
    }  
};
```

Test harness

```
boost::shared_ptr<B> tmpB(new B);  
boost::shared_ptr<A> tmpA(new A(tmpB));  
tmpB->set(tmpA); // Cyclic dependency introduced
```

# Properties – boost::weak\_ptr

---

- **Typical usage is breaking cyclic dependencies**
- **Must call .lock() to determine whether the object pointed to still lives before trying to access it**
- **Must be converted to a shared\_ptr when access the object again**
- Implements a "a less important" shared\_ptr
- A shared\_ptr without reference counting.
- Can be used in containers

`scoped_ptr<T>`

# Usage

## boost::scoped\_ptr

---

- Simple usage
  - ▶ std::string is automatically deallocated
    - ▶ Where exception is thrown
    - ▶ End of scope

```
void f()
{
    boost::scoped_ptr<std::string>
    stringPtr(new std::string("Hello"));

    if (...) throw std::some_error("Bad number");

    std::cout << *stringPtr << std::endl;
}
```

# Usage

## boost::scoped\_ptr

---

- Simple usage
  - ▶ std::string is automatically deallocated
    - ▶ Where exception is thrown
    - ▶ End of scope

```
void f()  
{  
    boost::scoped_ptr<std::string>  
    stringPtr(new std::string("Hello"));  
  
    if (...) throw std::some_error("Bad number");  
  
    std::cout << *stringPtr << std::endl;  
}
```

Exit points

# Properties

## `boost::scoped_ptr`

---

- **Holds a pointer to an element allocated with `new`**
- **Guarantees that `delete` is called at destruction time**
- Calls `delete` via a call to the member function `reset()`
- Cannot be used in a container, not copyable
- Alternative
  - ▶ `std::auto_ptr` const but it cannot be reset

# Boost::SmartPointer - Summary



# Boost SmartPointer summary

---

- ***boost::scoped\_ptr & boost::scoped\_array***
  - ▶ Objects with short lifespan – confined to function/object
  - ▶ Single object or an array of objects
  - ▶ Non-copyable (whole point)
- ***boost::shared\_ptr & boost::shared\_array***
  - ▶ A more general wrapping used in containers
  - ▶ Single object or an array of objects
  - ▶ You want it *all* and you are willing to pay for it
- ***boost::weak\_ptr***
  - ▶ Typically used to break circular references
- ***boost::intrusive\_ptr***
  - ▶ Where OS or framework implement reference counting