# RESOURCE HANDLING

# AGENDA

- A resource ?
- Fundamental SW Design challenges
- Basic building block - RAII
- The full monty - boost::shared_ptr<>
- Exercise structure

# A RESOURCE ?

# A RESOURCE ?

- Brainstorm - What do you consider a Resource Handling challenge?
  - What is a resource in sw “terms”?
  - In which situations do you foresee challenges?

# FUNDAMENTAL SW

# DESIGN CHALLENGE

AARHUS
UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

# FUNDAMENTAL SW DESIGN CHALLENGE

- Design and implement a system that
  - Garbage collection exists and ensures that a resource is released when no one uses it (multiple parties may share a resource)
  - It must be thread-safe
    - The resource itself is NOT protected, but its destruction must be

# ABSTRACT 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
    - or other way round - *not known*
  - Resource is hereafter relinquished
- Usefulness?
  - Our Message Queues :)

# CONCRETE EXAMPLE

- Problem where?

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05    if(...)
06      return;
07
08    if(...)
09      throw SomeError("Bad input");
10
11    delete d;
12    delete c;
13  }
```

# CONCRETE EXAMPLE

- Problem where?
- Function throws an exception...

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05    if(...)
06      return;
07
08    if(...)
09      throw SomeError("Bad input");
10
11    delete d;
12    delete c;
13  }
```

# CONCRETE EXAMPLE

- Problem where?
- Function throws an exception...
- If statement true
  - return - leave function
  - *Memory not deallocated*

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05    if(...)
06      return;
07
08    if(...)
09      throw SomeError("Bad input");
10
11    delete d;
12    delete c;
13  }
```

# CONCRETE EXAMPLE

- Problem where?
- Function throws an exception...
- If statement true
  - return - leave function
  - *Memory not deallocated*
- If statement true
  - Exception thrown function left
  - *Memory not deallocated*

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05    if(...)
06      return;
07
08    if(...)
09      throw SomeError("Bad input");
10
11    delete d;
12    delete c;
13  }
```

# CONCRETE EXAMPLE

- Problem where?
- Function throws an exception...
- If statement true
  - return - leave function
  - *Memory not deallocated*
- If statement true
  - Exception thrown function left
  - *Memory not deallocated*
- *If* everything went well
  - The only path where things get cleaned up...

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05    if(...)
06      return;
07
08    if(...)
09      throw SomeError("Bad input");
10
11    delete d;
12    delete c;
13  }
```

# BASIC BUILDING BLOCK - RAII

# BASIC BUILDING BLOCK - 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 IN USE

```
01  template<typename T>
02  class RAII
03  {
04  public:
05    explicit RAII( T* p = 0 ) : p_(p) {}
06
07    ~RAII() { delete p_; }
08
09    // Smart pointer idiom mixed-in!
10    T& operator*() const { return *p_; }
11    T* operator->() const { return p_; }
12  private:
13    T*  p_;
14  };
```

```
01  {
02    RAII<std::vector<int> > r( new std::vector() );
03    std::cout << r->size() << std::endl;
04  }
```

# RAII IN USE

- RAII instance is constructed and passed instance

```
01  template<typename T>
02  class RAII
03  {
04  public:
05    explicit RAII( T* p = 0 ) : p_(p) {}
06
07    ~RAII() { delete p_; }
08
09    // Smart pointer idiom mixed-in!
10    T& operator*() const { return *p_; }
11    T* operator->() const { return p_; }
12  private:
13    T*  p_;
14  };
```

```
01  {
02    RAII<std::vector<int> > r( new std::vector() );
03    std::cout << r->size() << std::endl;
04  }
```

# RAII IN USE

- RAII instance is constructed and passed instance
- Arrow operator in use
  - Returns pointer to object -> method `size()` called on `std::vector<>`

```
01  template<typename T>
02  class RAII
03  {
04  public:
05    explicit RAII( T* p = 0 ) : p_(p) {}
06
07    ~RAII() { delete p_; }
08
09    // Smart pointer idiom mixed-in!
10    T& operator*() const { return *p_; }
11    T* operator->() const { return p_; }
12  private:
13    T*  p_;
14  };
```

```
01  {
02    RAII<std::vector<int> > r( new std::vector() );
03    std::cout << r->size() << std::endl;
04  }
```

# RAII IN USE

- RAII instance is constructed and passed instance
- Arrow operator in use
  - Returns pointer to object -> method `size()` called on `std::vector<>`
- Scope left
  - Destructor called
  - `std::vector<>` deleted

```
01  template<typename T>
02  class RAII
03  {
04  public:
05    explicit RAII( T* p = 0 ) : p_(p) {}
06
07    ~RAII() { delete p_; }
08
09    // Smart pointer idiom mixed-in!
10    T& operator*() const { return *p_; }
11    T* operator->() const { return p_; }
12  private:
13    T*  p_;
14  };
```

```
01  {
02    RAII<std::vector<int> > r( new std::vector() );
03    std::cout << r->size() << std::endl;
04  }
```

# THE FULL MONTY -
# BOOST::SHARED_PTR<>

# THE FULL MONTY - BOOST::SHARED_PTR<>

- Fixing concrete example
- More eleborate example
- Handling cyclic dependencies
- Summary (boost smart pointers)

# FIXING CONCRETE EXAMPLE

## OLD

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11
12    delete d;
13    delete c;
14  }
```

## NEW

```
01  void f()
02  {
03    boost::shared_ptr<Client*> c(new Client);
04    boost::shared_ptr<Data*> d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11  }
```

- Solved?

# FIXING CONCRETE EXAMPLE

## OLD

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11
12    delete d;
13    delete c;
14  }
```

## NEW

```
01  void f()
02  {
03    boost::shared_ptr<Client*> c(new Client);
04    boost::shared_ptr<Data*> d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11  }
```

- Solved?
- Both `c` and `d` are shared_ptr<>
- Both calls to delete removed (happens indirectly)

# FIXING CONCRETE EXAMPLE

## OLD

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11
12    delete d;
13    delete c;
14  }
```

## NEW

```
01  void f()
02  {
03    boost::shared_ptr<Client*> c(new Client);
04    boost::shared_ptr<Data*> d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11  }
```

- Solved?
- Both `c` and `d` are shared_ptr<>
- Both calls to delete removed (happens indirectly)
- if statement true...
  - Function returns
  - *shared_ptr<> cleans up for both `c` and `d`* -> **YES** :-)

# FIXING CONCRETE EXAMPLE

## OLD

```
01  void f()
02  {
03    Client* c = new Client;
04    Data* d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11
12    delete d;
13    delete c;
14  }
```

## NEW

```
01  void f()
02  {
03    boost::shared_ptr<Client*> c(new Client);
04    boost::shared_ptr<Data*> d = acquireData(c);
05
06    if(...)
07        return;
08
09    if(...)
10      throw SomeError("Bad input");
11  }
```

- Solved?
- Both `c` and `d` are shared_ptr<>
- Both calls to delete removed (happens indirectly)
- if statement true...
  - Function returns
  - *shared_ptr<> cleans up for both `c` and `d`* -> **YES** :-)

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42        1
04  2) 003756D0       43        2
05  3) 003756D0       43        3
06  4) 003756D0       43        1
07  5) 00000000       xx        0
```

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42         1
04  2) 003756D0       43         2
05  3) 003756D0       43         3
06  4) 003756D0       43         1
07  5) 00000000       xx         0
```

- Create a shared_ptr<>

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42         1
04  2) 003756D0       43         2
05  3) 003756D0       43         3
06  4) 003756D0       43         1
07  5) 00000000       xx         0
```

- Create a shared_ptr<>
- Print out stats

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42        1
04  2) 003756D0       43        2
05  3) 003756D0       43        3
06  4) 003756D0       43        1
07  5) 00000000       xx        0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42        1
04  2) 003756D0       43        2
05  3) 003756D0       43        3
06  4) 003756D0       43        1
07  5) 00000000       xx        0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42         1
04  2) 003756D0       43         2
05  3) 003756D0       43         3
06  4) 003756D0       43         1
07  5) 00000000       xx         0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42          1
04  2) 003756D0       43          2
05  3) 003756D0       43          3
06  4) 003756D0       43          1
07  5) 00000000       xx          0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats
- Create a copy (only *alive* in scope)

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42         1
04  2) 003756D0       43         2
05  3) 003756D0       43         3
06  4) 003756D0       43         1
07  5) 00000000       xx         0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats
- Create a copy (only *alive* in scope)

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42        1
04  2) 003756D0       43        2
05  3) 003756D0       43        3
06  4) 003756D0       43        1
07  5) 00000000       xx        0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats
- Create a copy (only *alive* in scope)
- `bb` is destroyed
- `ee` is reset

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42         1
04  2) 003756D0       43         2
05  3) 003756D0       43         3
06  4) 003756D0       43         1
07  5) 00000000       xx         0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats
- Create a copy (only *alive* in scope)
- `bb` is destroyed
- `ee` is reset
- Print out stats

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42        1
04  2) 003756D0       43        2
05  3) 003756D0       43        3
06  4) 003756D0       43        1
07  5) 00000000       xx        0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats
- Create a copy (only *alive* in scope)
- `bb` is destroyed
- `ee` is reset
- Print out stats
- `aa` is reset

# MORE ELEBORATE EXAMPLE

## CODE

```
01  boost::shared_ptr<int> ee;
02  boost::shared_ptr<int> aa(new int(42));
03
04  std::cout << aa << "\t" << *aa << "\t" << aa.use_count() << std::endl;
05  {
06    boost::shared_ptr<int> bb(aa);
07    ++(*bb);
08    std::cout << bb << "\t" << *bb << "\t" <<bb.use_count() <<std::endl;
09
10    ee = bb;
11    std::cout << ee << "\t" << *ee << "\t" <<ee.use_count() <<std::endl;
12  }
13
14  ee.reset();
15  std::cout << aa << "\t" << *aa << "\t" <<aa.use_count() <<std::endl;
16
17  aa.reset();
18  std::cout << aa << "\t" << "xx" << "\t" <<aa.use_count() <<std::endl;
```

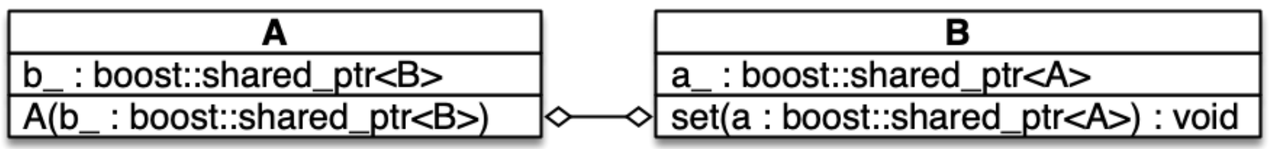
## OUTPUT

```
01  Stdout readout:
02     Pointer      Value  Use Count
03  1) 003756D0       42         1
04  2) 003756D0       43         2
05  3) 003756D0       43         3
06  4) 003756D0       43         1
07  5) 00000000       xx         0
```

- Create a shared_ptr<>
- Print out stats
- Create a copy
- Increment *shared* integer
- Print out stats
- Create a copy (only *alive* in scope)
- `bb` is destroyed
- `ee` is reset
- Print out stats
- `aa` is reset
- Print out stats

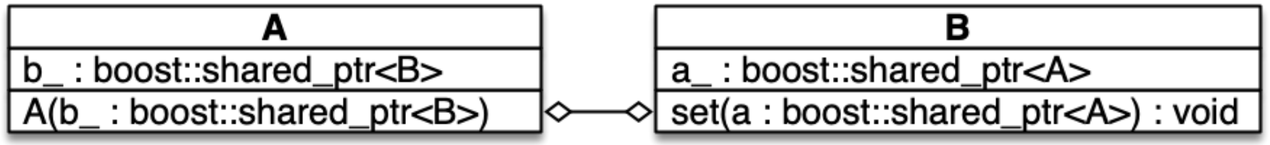
# HANDLING CYCLIC DEPENDENCIES

- Problem
  - When objects are inter-dependent via shared pointers
- Consequence
  - Leaving scope decrements shared pointers *but*
    - Inter-depencies keeps them *alive*
    - Memory though shared pointers are used :-(



# HANDLING CYCLIC DEPENDENCIES

## CODE EXEMPLICATION




```cpp
struct A
{
  A(std::shared_ptr<B> b)
    : b_(b)
  {  }

  std::shared_ptr<B> b_;
};
```

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

  std::shared_ptr<A> a_;
};
```

```cpp
{
  std::shared_ptr<B> tmpB(new B);
  std::shared_ptr<A> tmpA(new A(tmpB));
  tmpB->set(tmpA);
}
```

Leaves scope, but A points B and B to A => memory leak

# HANDLING CYCLIC DEPENDENCIES

## CODE EXEMPLICATION - SOLUTION

```
01  struct A
02  {
03    A(std::shared_ptr<B> b)
04      : b_(b)
05    {  }
06
07    std::shared_ptr<B> b_;
08  };
```

```
01  struct B {
02    void set(boost::shared_ptr<A> a)
03    { a_ = a; }
04
05    boost::weak_ptr<A>  a_;
06
07    void doStuff() {
08      boost::shared_ptr<A> a = a_.lock();
09      if(a) {
10        // Do stuff
11      }
12    };
13  };
```

# HANDLING CYCLIC DEPENDENCIES

## CODE EXEMPLICATION - SOLUTION

```
01  struct A
02  {
03    A(std::shared_ptr<B> b)
04      : b_(b)
05    {  }
06
07    std::shared_ptr<B> b_;
08  };
```

```
01  struct B {
02    void set(boost::shared_ptr<A> a)
03    { a_ = a; }
04
05    boost::weak_ptr<A>  a_;
06
07    void doStuff() {
08      boost::shared_ptr<A> a = a_.lock();
09      if(a) {
10        // Do stuff
11      }
12    };
13  };
```

- Create a weak_ptr<>
  - Does *NOT* increment reference counter!
    - Whole point

# HANDLING CYCLIC DEPENDENCIES

## CODE EXEMPLICATION - SOLUTION

```
01  struct A
02  {
03    A(std::shared_ptr<B> b)
04      : b_(b)
05    {  }
06
07    std::shared_ptr<B> b_;
08  };
```

```
01  struct B {
02    void set(boost::shared_ptr<A> a)
03    { a_ = a; }
04
05    boost::weak_ptr<A>  a_;
06
07    void doStuff() {
08      boost::shared_ptr<A> a = a_.lock();
09      if(a) {
10        // Do stuff
11      }
12    };
13  };
```

- Create a weak_ptr<>
  - Does *NOT* increment reference counter!
    - Whole point
- Convert weak_ptr<> to shared_ptr<>
  - Important to ensure it exists while using

# HANDLING CYCLIC DEPENDENCIES

## CODE EXEMPLICATION - SOLUTION

```
01  struct A
02  {
03    A(std::shared_ptr<B> b)
04      : b_(b)
05    {  }
06
07    std::shared_ptr<B> b_;
08  };
```

```
01  struct B {
02    void set(boost::shared_ptr<A> a)
03    { a_ = a; }
04
05    boost::weak_ptr<A>  a_;
06
07    void doStuff() {
08      boost::shared_ptr<A> a = a_.lock();
09      if(a) {
10        // Do stuff
11      }
12    };
13  };
```

- Create a weak_ptr<>
  - Does *NOT* increment reference counter!
    - Whole point
- Convert weak_ptr<> to shared_ptr<>
  - Important to ensure it exists while using
- Remember to verify that the shared_ptr<> actually points to something...

# SUMMARY (BOOST SMART POINTERS)

- **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::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::intrusive_ptr**
  - Where OS or framework implement reference counting

# EXERCISE STRUCTURE

# EXERCISE STRUCTURE

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

*Increase in feature set and complexity*