

# C++ Exception Safety

Ali Çehreli

acehrel@yahoo.com

- Object lifetimes
- Exceptions
- RAII idiom
- Smart pointers
- Exception safety

# Resources

- Guru of the Week puzzles by Herb Sutter (archived at **GotW.ca**)
- “**Exceptional C++**” books by Herb Sutter, which are based on the GotW
- For a family of smart pointers: **boost.org**
- For policy-based smart pointers (and others):  
Andrei Alexandrescu's Loki library and his book  
“**Modern C++ Design**”

# Types of objects and their lifetimes

**Automatic object:** Lifetime is determined by the compiler; local objects, class members, temporary objects

**Dynamic object:** Lifetime is determined by the programmer; starts with `new` and ends with `delete`

**Static object:** Lifetime is determined by the compiler; ignored in this talk

# Automatic and dynamic objects

```
void foo()  
{  
    A a0;  
  
    D * d = new D;    // dynamic  
  
    {  
        A a1;  
    } // <-- a1 ends automatically  
  
} // <-- a0 ends automatically
```

# Object Construction Steps by the compiler

- 1) Construct virtual bases
- 2) Construct other bases
- 3) Initialize members in the order that they were defined
- 4) Execute constructor body

# Object Destruction Steps by the compiler

- 1) Execute the destructor body
- 2) Destroy members and bases in the reverse order of construction

Note: Destructors of fundamental types are thought to be “empty”, i.e. `ints` are not “finalized” with any special value, and plain pointers don't `delete` the objects that they point to

# When Construction Fails, the compiler...

- 1) destroys all that is constructed so far, in the reverse order
- 2) if the object was being “new'ed”, returns the memory that was allocated for the object

**Note: The destructor of *this object* is not called.**

# An object is not an object ...

... unless it's fully constructed...

```
class MyClass
{
    /* ... */

    MyClass(/* ... */)

        : /* base and member initializations */

    {
        /* further construction */

    } // <-- only now an object
};
```

# Exceptions

are ...

- a) only for important errors?
- b) for any error?
- c) for anything that's out of the ordinary?

- **try** to do something
- an error may be **thrown**,
- which some scope may **catch**.

**Unwinding the program stack:** exiting scopes recursively until a scope **catches** that type of exception; automatic *objects* are destructed as usual

# Stack Unwinding

If another exception is thrown during stack unwinding, meaning that an exception is already in flight; the program will **abort**.

**Guideline:** Never allow exceptions escape from destructors.

# A try/catch example

```
void foo()
{
    try {
        do_something();
        possibly_not_executed();
    }
    catch (const SomeTypeOfError & error) {

        /* error handling */

    }
}

void do_something()
{
    if (some_condition) {
        throw SomeTypeOfError();
    }

    possibly_not_executed_either();
}
```

# A safe way of managing resources in C

```
int bar(Resource ** in_out)
{
    int err = 0;

    Resource * r0 = NULL;
    Resource * r1 = NULL;

    err = allocate_resource(&r0);
    if (err) goto finally;

    err = allocate_resource(&r1);
    if (err) goto finally;

    /* use r0 and r1 */

    if (err) goto finally;

    /* transfer ownership */
    *in_out = r0;
    r0 = NULL;

finally:

    deallocate_resource(&r1);
    deallocate_resource(&r0);

    return err;
}
```

# A safe way of managing resources in C

```
int bar(Resource ** in_out)
{
    int err = 0;

    Resource * r0 = NULL;
    Resource * r1 = NULL;

    err = allocate_resource(&r0);
    if (err) goto finally;

    err = allocate_resource(&r1);
    if (err) goto finally;

    /* use r0 and r1 */

    if (err) goto finally;

    /* transfer ownership */
    *in_out = r0;
    r0 = NULL;

finally:

    deallocate_resource(&r1);
    deallocate_resource(&r0);

    return err;
}
```

# A safe way of managing resources in C

```
int bar(Resource ** in_out)
{
    int err = 0;

    Resource * r0 = NULL;
    Resource * r1 = NULL;

    err = allocate_resource(&r0);
    if (err) goto finally;

    err = allocate_resource(&r1);
    if (err) goto finally;

    /* use r0 and r1 */

    if (err) goto finally;

    /* transfer ownership */
    *in_out = r0;
    r0 = NULL;

finally:

    deallocate_resource(&r1);
    deallocate_resource(&r0);

    return err;
}
```

# A safe way of managing resources in C++

```
Resource bar()  
{  
  
    Resource r0(/* ... */);  
    Resource r1(/* ... */);  
  
  
    /* use r0 and r1 */  
  
    /* transfer ownership */  
    return r0;  
  
}
```

# Explicit resource deallocation ...

```
void foo()  
{  
    Resource * p = allocate();  
    /* ... */  
    deallocate(p);          // <-- explicit  
}
```

... is not guaranteed to work in C++.

```
void foo()  
{  
    Resource * p = allocate();  
    /* ... */           // may throw  
    deallocate(p);      // may not be executed  
}
```

# RAII idiom

(Resource Acquisition Is Initialization)

- ∴ deallocation is destruction
- ∴ deallocation belongs in the destructor

**Guideline:** No explicit deallocation in code; deallocation happens in the destructor of a managing class.

# RAII example

```
void foo_wishful()
{
    Resource * p = allocate();
    /* ... */ // may throw
    deallocate(p); // may not be executed
}
```

```
void foo_RAII()
{
    ResourceManager p(allocate());
    /* ... */ // may throw; fine...
}
```

# Smart pointers

- RAII objects that delete the objects that they own
- Can be used *almost* seamlessly as plain pointers
- May have additional smarts

# A smart pointer implementation

```
class SmartPointer                                     SmartPointer p(new MyType());
{
    MyType * ptr;

public:

    explicit SmartPointer(MyType * p = 0)
        :
        ptr(p)
    {}

    ~SmartPointer()
    {
        delete ptr;
    }

};
```

# A smart pointer implementation

```
class SmartPointer                                     SmartPointer p(new MyType());
{
    MyType * ptr;                                     bar(p.get());

public:

    explicit SmartPointer(MyType * p = 0)
        :
        ptr(p)
    {}

    ~SmartPointer()
    {
        delete ptr;
    }

    MyType * get() const { return ptr; }

};
```

# A smart pointer implementation

```
class SmartPointer
{
    MyType * ptr;

public:
    explicit SmartPointer(MyType * p = 0)
        :
        ptr(p)
    {}

    ~SmartPointer()
    {
        delete ptr;
    }

    MyType * get() const { return ptr; }

    MyType & operator* () const { return *ptr; }

};

SmartPointer p(new MyType());

bar(p.get());

MyType & r = *p;
r.i = 7;
```

# A smart pointer implementation

```
class SmartPointer
{
    MyType * ptr;

public:
    explicit SmartPointer(MyType * p = 0)
        :
        ptr(p)
    {}

    ~SmartPointer()
    {
        delete ptr;
    }

    MyType * get() const { return ptr; }

    MyType & operator* () const { return *ptr; }

    MyType * operator->() const { return ptr; }

};

SmartPointer p(new MyType());

bar(p.get());

MyType & r = *p;
r.i = 7;

p->i = 42;
```

# A smart pointer implementation

```
class SmartPointer
{
    MyType * ptr;

public:
    explicit SmartPointer(MyType * p = 0)
        :
        ptr(p)
    {}

    ~SmartPointer()
    {
        delete ptr;
    }

    MyType * get() const { return ptr; }

    MyType & operator* () const { return *ptr; }

    MyType * operator->() const { return ptr; }

    bool operator! () const { return ptr == 0; }
};

SmartPointer p(new MyType());

bar(p.get());

MyType & r = *p;
r.i = 7;

p->i = 42;

if (!p) {
    /* ... */
}
```

# A smart pointer implementation

```
class SmartPointer
{
    MyType * ptr;

public:
    explicit SmartPointer(MyType * p = 0)
        :
        ptr(p)
    {}

    ~SmartPointer()
    {
        delete ptr;
    }

    MyType * get() const { return ptr; }

    MyType & operator* () const { return *ptr; }

    MyType * operator->() const { return ptr; }

    bool operator! () const { return ptr == 0; }

private:
    SmartPointer(SmartPointer const &);
    SmartPointer & operator= (SmartPointer const &);
};

SmartPointer p(new MyType());

bar(p.get());

MyType & r = *p;
r.i = 7;

p->i = 42;

if (!p) {
    /* ... */
}

SmartPointer p1(p); // ERROR
p2 = p0; // ERROR
```

# boost::scoped\_ptr

```
#include <boost/scoped_ptr.hpp>

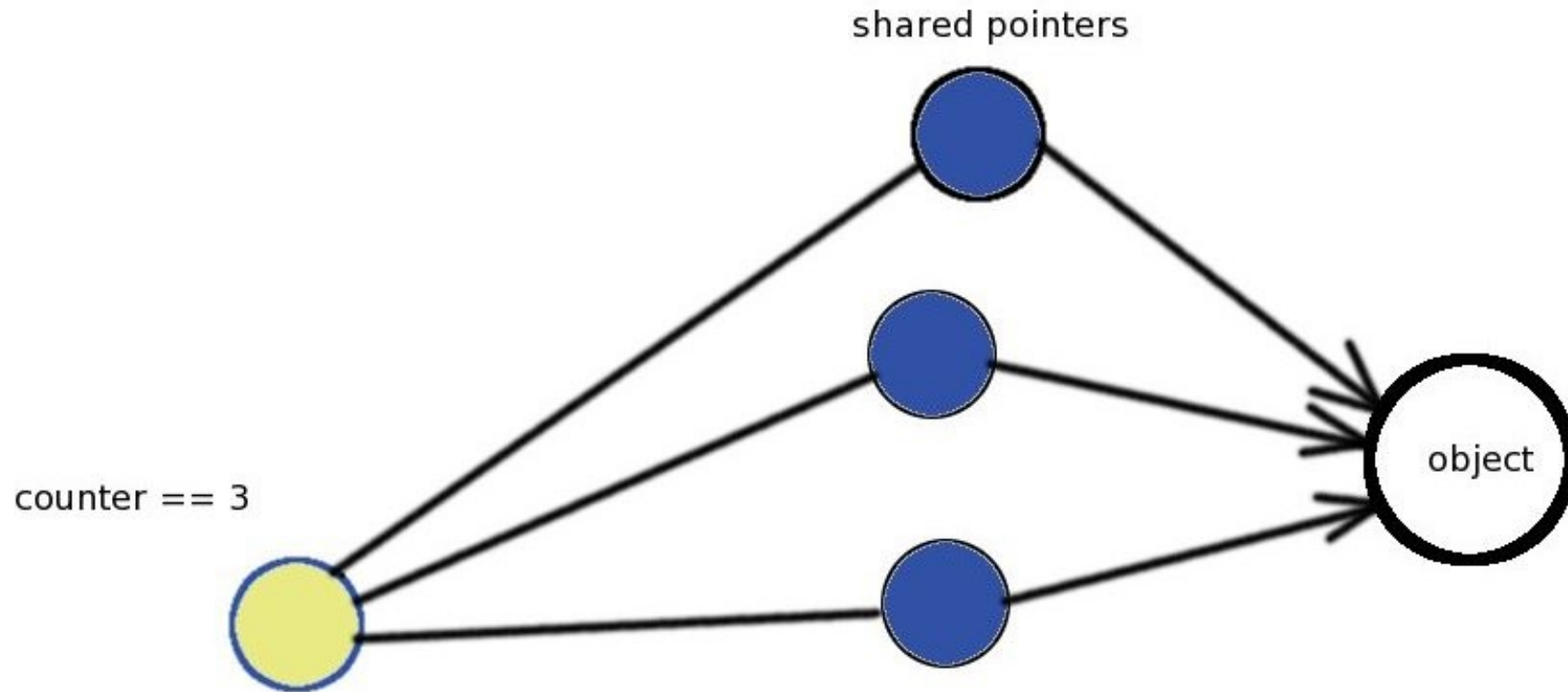
typedef boost::scoped_ptr<Animal> AnimalPtr;

void foo()
{
    AnimalPtr animal(new Cat);

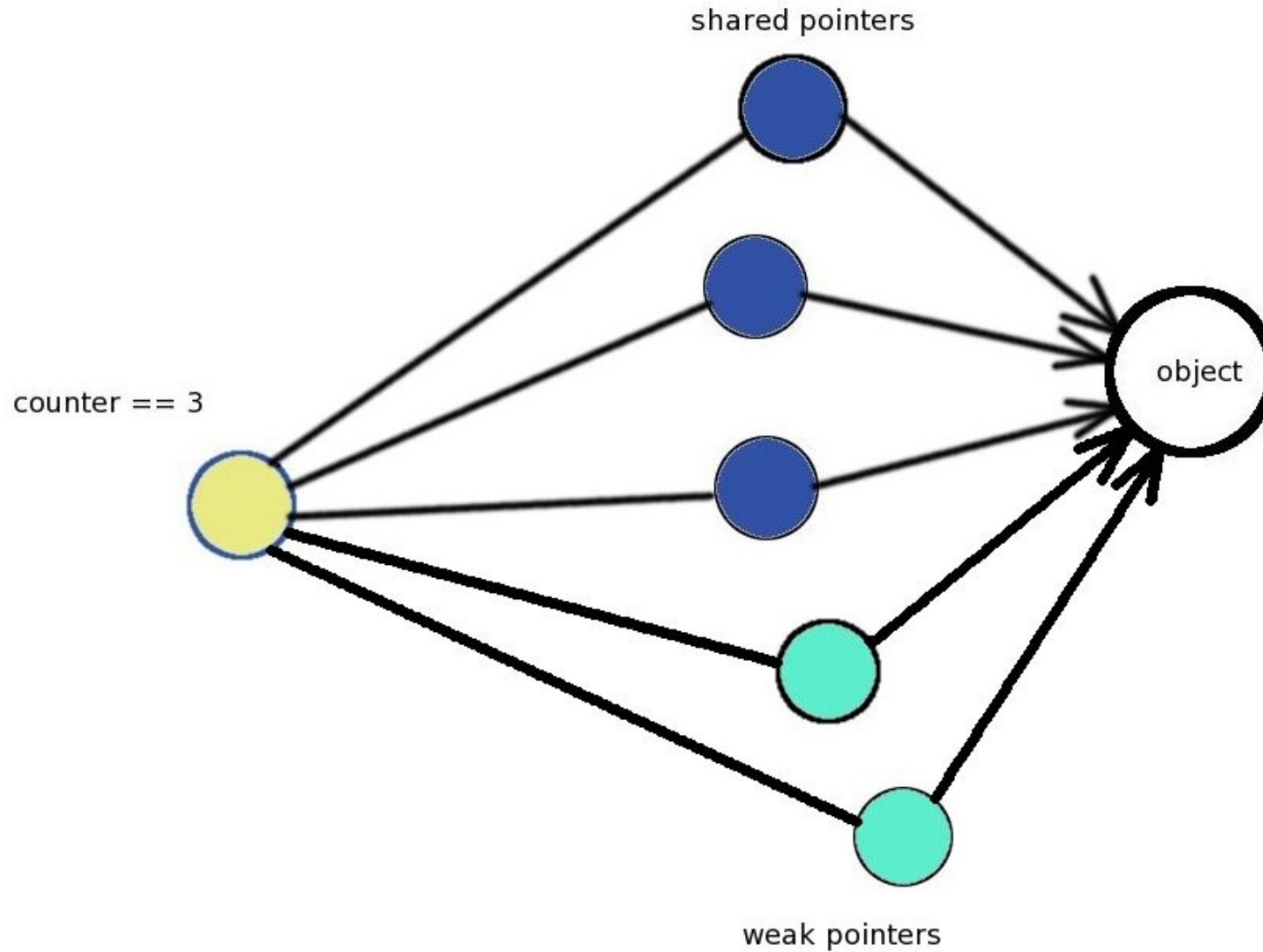
    animal->talk();

    /* ... */
} // Cat is deleted here
```

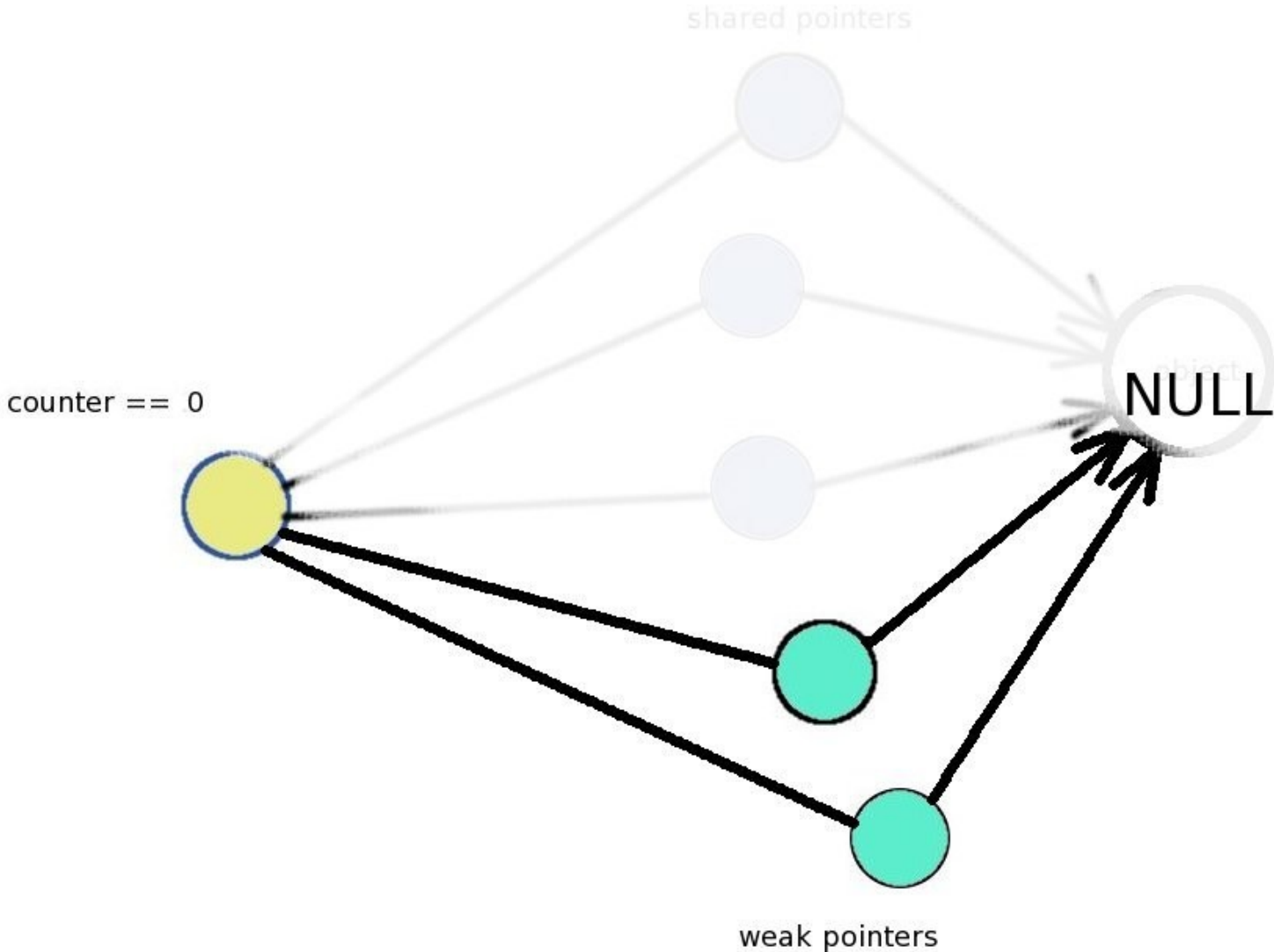
# boost::shared\_ptr



# boost::weak\_ptr



# Object goes, boost::weak\_ptr remains



# boost::intrusive\_ptr

- The type must already have a possibly more efficient way of reference counting
- The memory footprint is the same as the plain T\*
- Two free functions must be defined:

```
void intrusive_ptr_add_ref(T *);  
void intrusive_ptr_release(T *);
```

# boost::scoped\_array

- Like boost::scoped\_ptr but calls delete [ ]
- Instead, consider  
`std::vector<MySharedPtr>`

# boost::shared\_array

- Like boost::shared\_ptr but calls delete [ ]
- Instead, consider  
`std::vector<MySharedPtr>`

# `std::auto_ptr`

(might have been named `std::transfer_ptr`  
equally usefully)

- The object is owned by a single `auto_ptr` at a given time
- When copied or assigned, the ownership changes hand: destination is the new owner, source is “nulled”
- Can not be used in standard containers (e.g. `vector`)

*C++ Technical Report 1* added some of Boost's smart pointers to the C++ standard library

- `std::tr1::shared_ptr`
- `std::tr1::weak_ptr`
- others?

# boost::ptr\_vector

- Owns the objects of the pointers
- Calls `delete` individually for the pointers in the vector

# Some History

- 1994: Tom Cargill's challenge to the C++ community, *Exception Handling: A False Sense of Security* (“Not possible to write a strongly exception safe Stack class template!”)
- 1997: Herb Sutter's *Guru of the Week #8* as the first complete solution after long discussions on `comp.lang.c++.moderated` newsgroup

# Tom Cargill's Challenge: Stack::pop() can not be made strongly exception safe if T's copy constructor throws

```
template <class T>
class Stack
{
    int count_;
    T * stack_;

    /* ... */

public:

    void push(const T &);

    T pop()
    {
        /* ... */

        T result = stack_[count_ - 1];
        --count_;
        return result;
    }
};
```

# An exception-safe Stack interface

```
template <class T>
class Stack
{
    int count_;
    T * stack_;

    /* ... */

public:

    void push(const T &);

    T & top()
    {
        return stack_[count_ - 1];
    }

    void pop()
    {
        --count_;
    }
};
```

**Guideline:** Design exception-safe interfaces

# The Rule of The Big Three

If you need to define at least one of

- the destructor
- the copy constructor
- the assignment operator

chances are, you need to at least declare the other(s) private.

# Destructive Sense of Security

## Is this class safe?

```
class WishfulManager : private boost::noncopyable
{
    One * one_;
    Two * two_;

public:

    WishfulManager()
        :
        one_(new One()),
        two_(new Two())
    {
        /* further construction */
    }

    ~WishfulManager()
    {
        delete two_;
        delete one_;
    }
};
```

# Destructive Sense of Security

## Not safe!

```
class WishfulManager : private boost::noncopyable
{
    One * one_;
    Two * two_;

public:

    WishfulManager()
        :
        one_(new One()),
        two_(new Two())    <- 1: new may throw, <- 2: Two() may throw
    {
        /* further construction */ <- 3: something else may throw
    }

    ~WishfulManager()
    {
        delete two_;
        delete one_;
    }
};
```

# Same class, this time copyable and assignable.

## Is it safe now?

```
class DubiousManager
{
    OneManager one_;    // Assume that copying and assigning these
    TwoManager two_;    // objects are "safe" operations; so it is safe
                        // to use their compiler-generated defaults
public:

    DubiousManager()
        :
        one_(new One()),
        two_(new Two())
    {
        /* further construction */
    }

    // No destructor needed...

};
```

# It depends!

Not safe at least if the members have regular assignment semantics e.g. like `std::string`

```
class DubiousManager
{
    std::string one_;
    std::string two_;

public:

    DubiousManager()
        :
        one_("one"),
        two_("two")
    {
        /* further construction */
    }

    // No destructor needed...

    // This class may be left in a half-assigned state if assignment
    // of two_ fails in the default operator=
}; // (one_ would be assigned but not two_)
```

# The Rule of The Big One

If the class has more than one member, you may need to at least declare

- the assignment operator

`private.`

# Broken Definition of operator=

```
class MyClass
{

/* ... */

MyClass & operator= (const MyClass & other)
{
    if (this != &other) {
        // destroy the current state of this
        // copy new state from the other <-- may throw; bad...
    }
    return *this;
}

};
```

# New Canonical Definition of operator=

```
class SafeManager
{
    OnePtr one_;
    TwoPtr two_;
    string three_;

    /* ... */

    SafeManager & operator= (const SafeManager & other)
    {
        SafeManager temp(other); // <-- try copying on the side first
        this->swap(temp);         // <-- swap this state with the copy

        return *this;

    } // <-- old state is released along with temp

    // Does not throw
    void swap(SafeManager & other)
    {
        one_.swap(other.one_);
        two_.swap(other.two_);
        three_.swap(other.three_);
    }
};
```

# Exception Safety Guarantees

**Basic Guarantee:** No resources leaked, object(s) are still usable (consistent but not necessarily predictable state)

**Strong Guarantee:** Program state remains unchanged

**Nothrow Guarantee:** The function will not emit an exception (e.g. `swap( )`)

# Summary: Exception Safety Guidelines

- Never allow exceptions escape from destructors
- Never deallocate explicitly in code (RAII)
- Do the work on the side first, then modify state
- Never make exception safety an afterthought
- Prefer cohesion (`top()` and `pop()`)

# Refresher: Smart Pointers

boost::**scoped\_ptr**: simple resource guard; non-copyable

boost::**shared\_ptr**: reference counted ownership

boost::**weak\_ptr**: no ownership; observer of shared\_ptr

std::**auto\_ptr**: transfer of ownership

etc...