Correctness, Safety and Freedom

or, How I Learned to Stop Worrying and Trust Myself

(and the Compiler)

The bad ol' days...

```
if (__lc_codepage == 0)
{ /* code page was not specified */
    if ( __getlocaleinfo( LC_INT_TYPE,
             MAKELCID(__lc_id[LC_CTYPE].wLanguage, SORT_DEFAULT),
             LOCALE_IDEFAULTANSICODEPAGE, (char **)&__lc_codepage ) )
        goto error_cleanup;
}

/* ... */

/* cleanup and return success */
_free_crt (cbuffer);
return 0;

error_cleanup:
  _free_crt (newctype1);
  _free_crt (cbuffer);
  return 1;
}
```


boostpro computing

Look at how careful this person has been! But it's an illusion. Maintenance nightmare!

Is it correct?

- Have all resources been freed?
  - ... on all possible code paths?
  - ... in the presence of exceptions?
- Have our invariants been preserved?
  - ... in all possible failure scenarios?

Well, yes. The programmer was careful.

- But will it still be correct after maintenance?

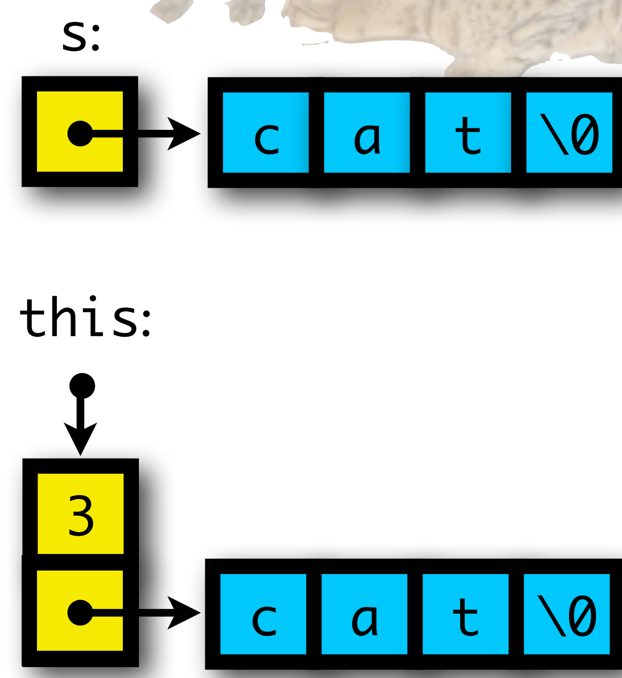
boostpro computing

Outline

- Contract Programming—*preconditions*, *postconditions*, and *invariants*
- Exception Safety—contractually speaking, techniques, and best practices
- Resource Management—Smart Pointers and RAII

String Toy

```
struct string
{
   string( char const* s )
   {
        len = std::strlen( s );
        buf = std::malloc( len+1 );
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=(
        string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```



boostpro computing

Contractually Speaking

Correct code is impossible without these concepts

- **Precondition**: <u>requirement on caller</u> by callee
- **Postcondition**: <u>guarantee to caller</u> by callee
- **Invariant**: condition that must always* hold
  - For a class, data structure, or program
  - At each iteration of a loop

  * except during mutation

Contractually Speaking

`string::string( char const* s ) { … }`

- **Precondition**: <u>requirement on caller</u> by callee

  s is null-terminated

- **Postcondition**: <u>guarantee to caller</u> by callee

  `!std::strcmp(s, &str[0])`

- **Invariant**: condition that must always* hold
  - For a class, data structure, or program
  - At each iteration of a loop

* except during mutation

`buf[len] == '\0'`

boostpro computing

Precondition: `s` is Null-Terminated

- This cannot be checked efficiently and safely in code
- But the caller *can* and *must* ensure null-termination
- Document the full precondition regardless of checkability!
- Q: Assuming we want to check something, what *can* we check for?

A: that `s` is non-NULL

boostpro computing

When a Precondition Check Fails

- Q: What does it mean?

  A: Your program is buggy.

- Q: What information do you have about the location of the bug?

  A: Zip. Zilch. None. Nada.

- Q: What is an appropriate response to a failed precondition check?

When a Precondition Check Fails

- Code somewhere is broken—we don't know where
- The assumptions upon which our program was written have been violated.
- Continuing to run is hazardous. *Do as little as possible.*
- Take emergency measures:
  - Capture stack trace / program state for debugging
  - Capture unsaved work for user
  - Shutdown (and optional restart)

boostpro computing

Checking Out the Precondition

```
struct string
{
   string( char const* s )
   {
        len = std::strlen( s );
        buf = std::malloc( len+1 );
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=(
        string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Checking Out the Precondition

```
struct string
{
   string( char const* s )
   {
        assert( s != 0 );
        len = std::strlen( s );
        buf = std::malloc( len+1 );
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=(
        string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Note: the code is not broken without this check!

Uh-oh. We'll come back to that.

boostpro computing

First, Some Improvements

```
struct string
{
   string( char const* s )
     : len( std::strlen( s ) )
     , buf( std::malloc( len+1 ) )
   {
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=(
        string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Guideline:
Isolate Mutating
Code

boostpro
computing

First, Some Improvements

```
struct string
{
   string( char const*const s )
      : len( std::strlen( s ) )
      , buf( std::malloc( len+1 ) )
   {
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=(
        string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Guideline: Use const wherever possible

boostpro computing

What About Memory Exhaustion?

- If we can’t get memory, can we satisfy our postcondition?
- Can we make “have sufficient memory” a precondition?
    - Ask yourself: “*is the caller in a position to check for or ensure that there is sufficient memory?*”
- How can we avoid a postcondition failure?

Two Ways to Avoid Postcondition Failures

1. Change the postcondition
   - In this case, allow an empty result
   - Makes error checking the client's problem
2. Throw an exception—the function never returns, thus postcondition never applies.
   - Better!

Throw an Exception

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( std::malloc( len+1 ) )
   {
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=( string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Throw an Exception

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( std::malloc( len+1 ) )
   {
        if ( buf == 0 ) throw std::bad_alloc();
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=( string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Throw an Exception

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( std::malloc( len+1 ) )
   {

        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=( string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Throw an Exception

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( new char[len+1] )
   {
        std::strcpy( buf, s );
   }

   ~string() { std::free( buf ); }

   string( string const& rhs );
   string& operator=( string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Guideline: Encapsulate throw-expressions

Throw an Exception

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( new char[len+1] )
   {
        std::strcpy( buf, s );
   }

   ~string() { delete [] buf; }

   string( string const& rhs );
   string& operator=( string const& rhs );
 private:
   std::size_t len;
   char* buf;
};
```

Copy Constructor

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( new char[len+1] )
   {
        std::strcpy( buf, s );
   }

   ~string() { delete [] buf; }

   string( string const& rhs )
     : len( rhs.len ),
       buf( new char[len+1] )
   {
        std::strcpy( buf, rhs.buf );
   }
```


boostpro computing

Copy Assignment

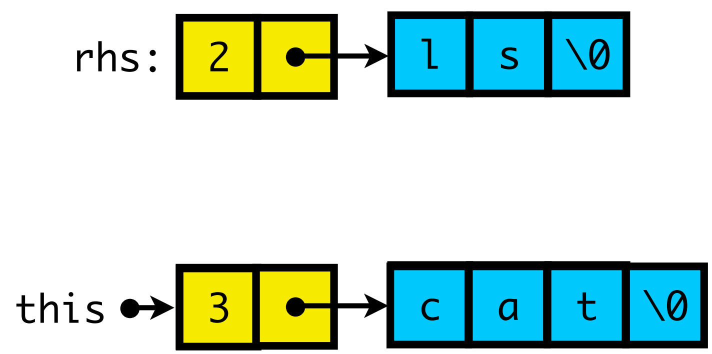
```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}
```

```
string& operator=( string const& rhs )
{
     if ( this == &rhs ) return *this;
     delete [] buf;
     len = rhs.len;
     buf = new char[len + 1];
     std::strcpy( buf, rhs.buf );
     return *this;
}
```

```
private:
```

Copy Assignment

```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}

string& operator=( string const& rhs )
{
     if ( this == &rhs ) return *this;
     delete [] buf;
     len = rhs.len;
     buf = new char[len + 1];
     std::strcpy( buf, rhs.buf );
     return *this;
}

private:
```



Invariant broken

Uh-oh!

Invariant restored

Exception Safety

- Safe code gives this **basic guarantee**:
    - All invariants are preserved
    - No resources leaked
    - Within that, arbitrary state changes allowed
- Not really specific to code that uses exceptions
- Handling errors correctly can be hard
- Exceptions can make it easier

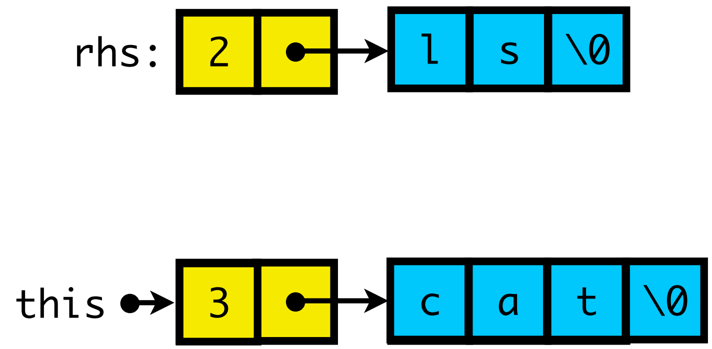
boostpro computing

2 can be seen as a consequence of #1

Nobody should ever teach about exception safety without talking about invariants

Restoring Safety, Take 1

```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}

string& operator=( string const& rhs )
{
     if ( this == &rhs ) return *this;
     delete [] buf;
     buf = 0; len = 0;
     buf = new char[rhs.len + 1];
     len = rhs.len;
     std::strcpy( buf, rhs.buf );
     return *this;
}
```



"Good state" restored

Uh-oh!

A Weakened Invariant

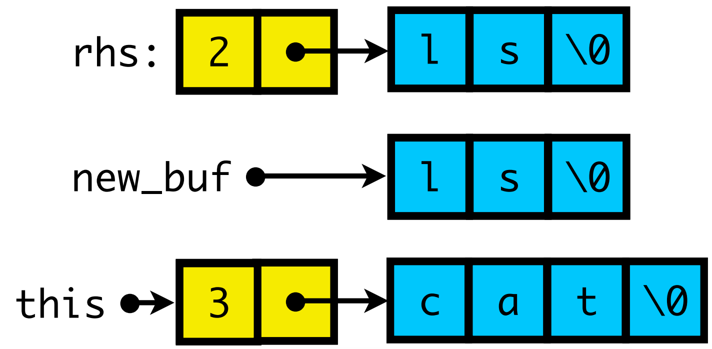
```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     if (rhs.buf) std::strcpy( buf, rhs.buf );
}

string& operator=( string const& rhs )
{
     if ( this == &rhs ) return *this;
     delete [] buf;
     buf = 0; len = 0;
     buf = new char[len + 1];
     len = rhs.len;
     if (rhs.buf) std::strcpy( buf, rhs.buf );
     return *this;
}
```

Prefer Strong Invariants

- Weaker invariants are:
  - Usually complicated to document
  - Much harder to reason about
  - Cause complexity to ripple through code

Keep the Invariant Strong

```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}
```




```
string& operator=( string const& rhs )
{
     if ( this == &rhs ) return *this;
     char* const new_buf = new char[rhs.len + 1];
     std::strcpy( new_buf, rhs.buf );
     delete [] buf;
     buf = new_buf;
     len = rhs.len;
     return *this;
}
```

First, the step(s) that can throw

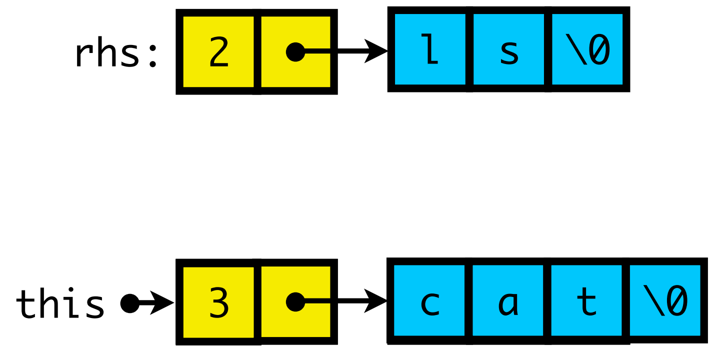
No: all data now "safe" from loss

Only later, the irreversible changes

boostpro computing

Atomicity!

```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}

string& operator=( string const& rhs )
{
     char* const new_buf = new char[rhs.len + 1];
     std::strcpy( new_buf, rhs.buf );
     delete [] buf;
     buf = new_buf;
     len = rhs.len;
     return *this;
}

private:
```



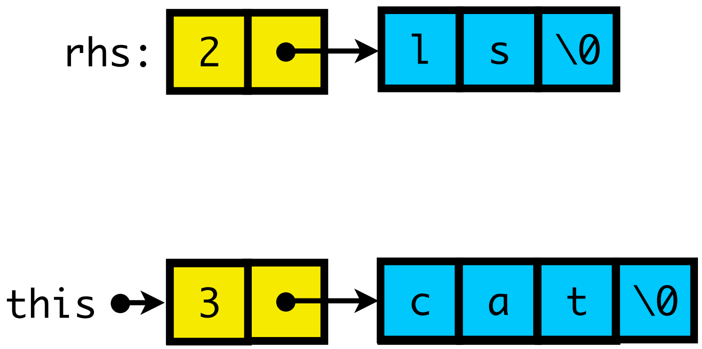
Q: If an exception is thrown, what will have been changed?

A: Nothing. Zip. Zilch. Nada… this is the "***strong guarantee***."

Note: Guarantee was not sought; it "fell out" of a fix.

Atomicity!

```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}

string& operator=( string const& rhs )
{
     char* const new_buf = new char[rhs.len + 1];
     std::strcpy( new_buf, rhs.buf );
     delete [] buf;
     buf = new_buf;
     len = rhs.len;
     return *this;
}

private:
```



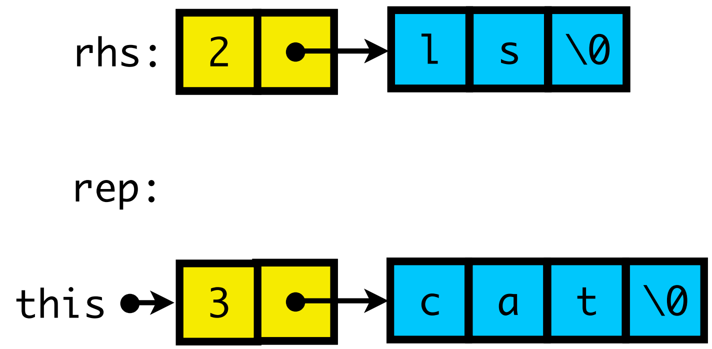
Q: If an exception is thrown, what will have been changed?

A: Nothing. Zip. Zilch. Nada… this is the "***strong guarantee***."

Note: Guarantee was not sought; it "fell out" of a fix.

Copy/Swap Idiom

```
string( string const& rhs )
  : len( rhs.len ),
    buf( new char[len+1] )
{
     std::strcpy( buf, rhs.buf );
}

string& operator=( string const& rhs )
{
     string rep( rhs );
     swap( *this, rep );
     return *this;
}

  friend void swap( string&, string& );
private:
  std::size_t len;
  char* buf;
```



Strong guarantee iff swap is non-throwing (but always correct).

Copy/Swap Idiom

```
  string( string const& rhs )
    : len( rhs.len ),
      buf( new char[len+1] )
  {
       std::strcpy( buf, rhs.buf );
  }

  string& operator=( string const& rhs )
  {
       string rep( rhs );
       swap( *this, rep );
       return *this;
  }

  friend void swap( string&, string& );
private:
  std::size_t len;
  char* buf;
```

Strong guarantee iff swap is non-throwing (but always correct).

boostpro computing

Copy/Swap Idiom

```
  string( string const& rhs )
    : len( rhs.len ),
      buf( new char[len+1] )
  {
       std::strcpy( buf, rhs.buf );
  }

  string& operator=( string rep )
  {
       swap( *this, rep );
       return *this;
  }

  friend void swap( string&, string& );
 private:
  std::size_t len;
  char* buf;
};
```

Copy will be <u>elided</u> for rvalue arguments

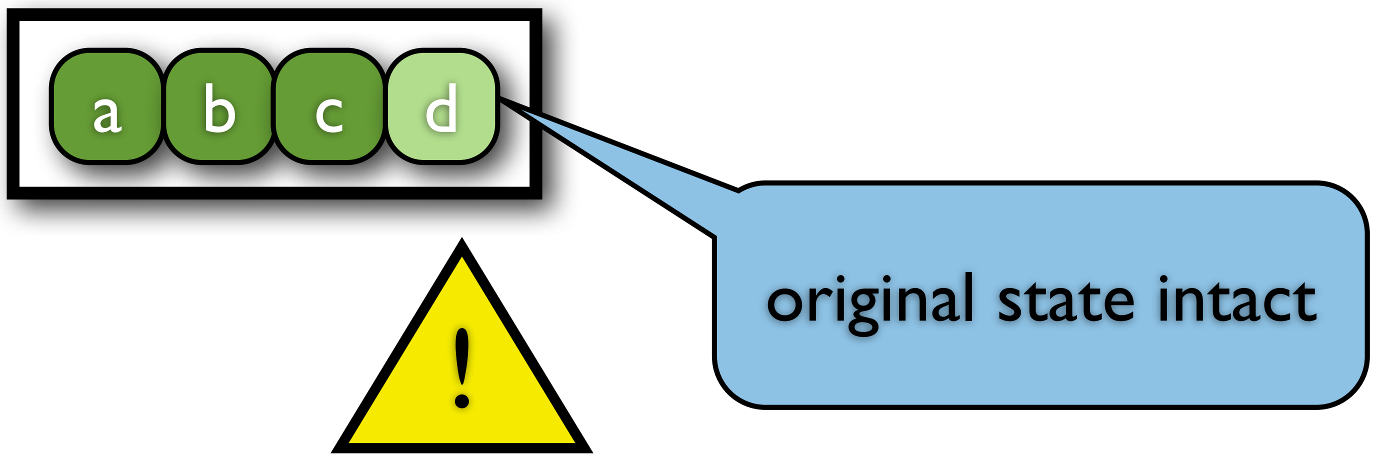
Strong guarantee iff swap is non-throwing (but always correct).

boostpro computing

A Little Perspective

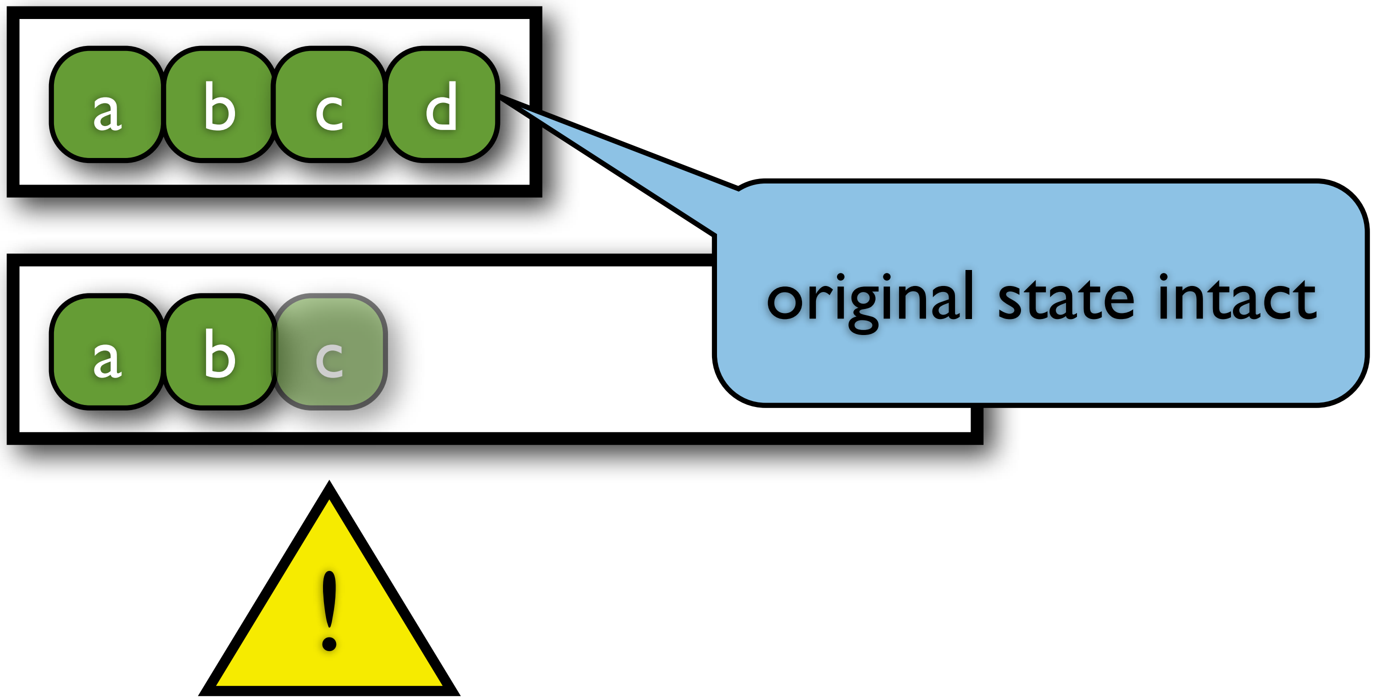
- The "strong guarantee" is not synonymous with safety
  - Basic guarantee often suffices, especially for stack objects
  - non-throwing operations are still needed for error handling
- Some operations can be "naturally strong," e.g. vector push_back
- Others only give the basic guarantee when implemented efficiently, e.g. vector assignment.
- <u>Don't copy/swap prematurely</u>, and code that only needs the basic guarantee won't be penalized.
- The one exception to that rule is in defining copy assignment. Why?
- Q: Is our copy/swap implementation of string assignment optimal?

Appending to a vector



code

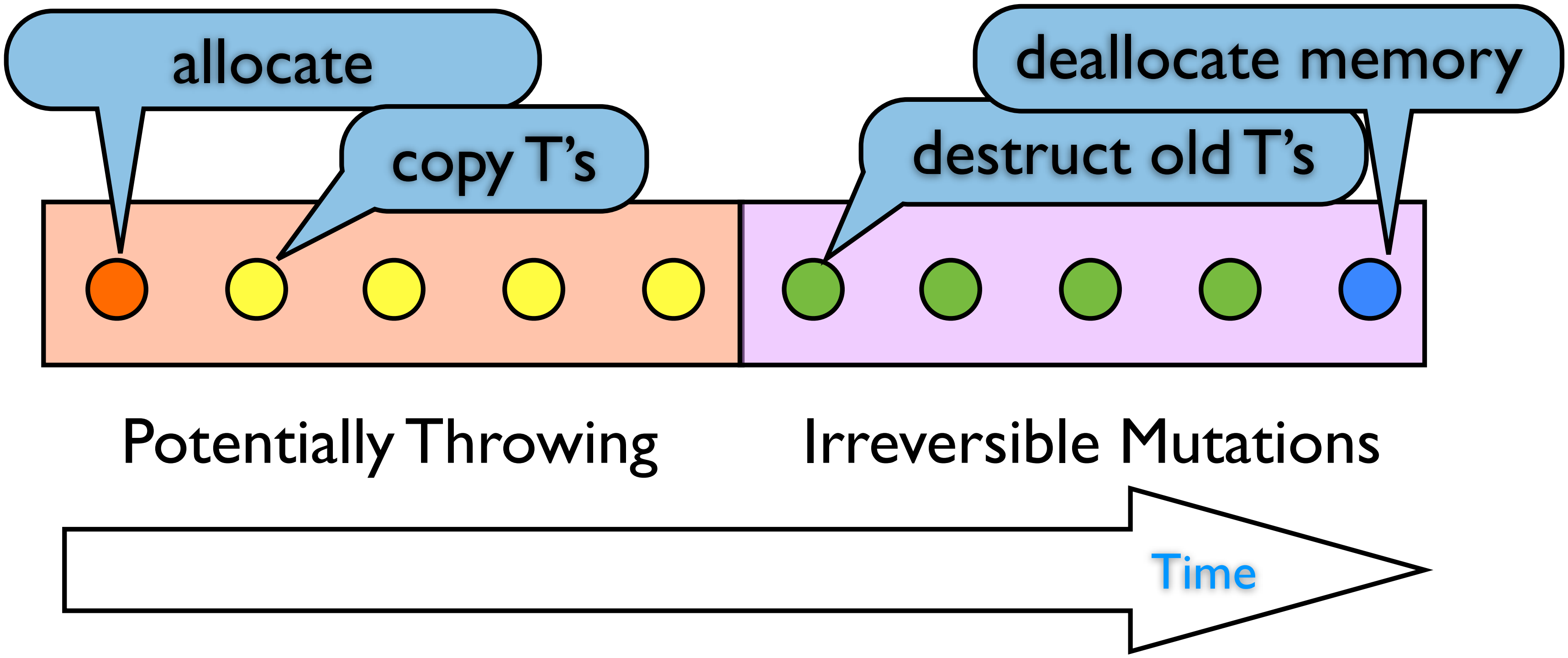
Appending to a vector



code

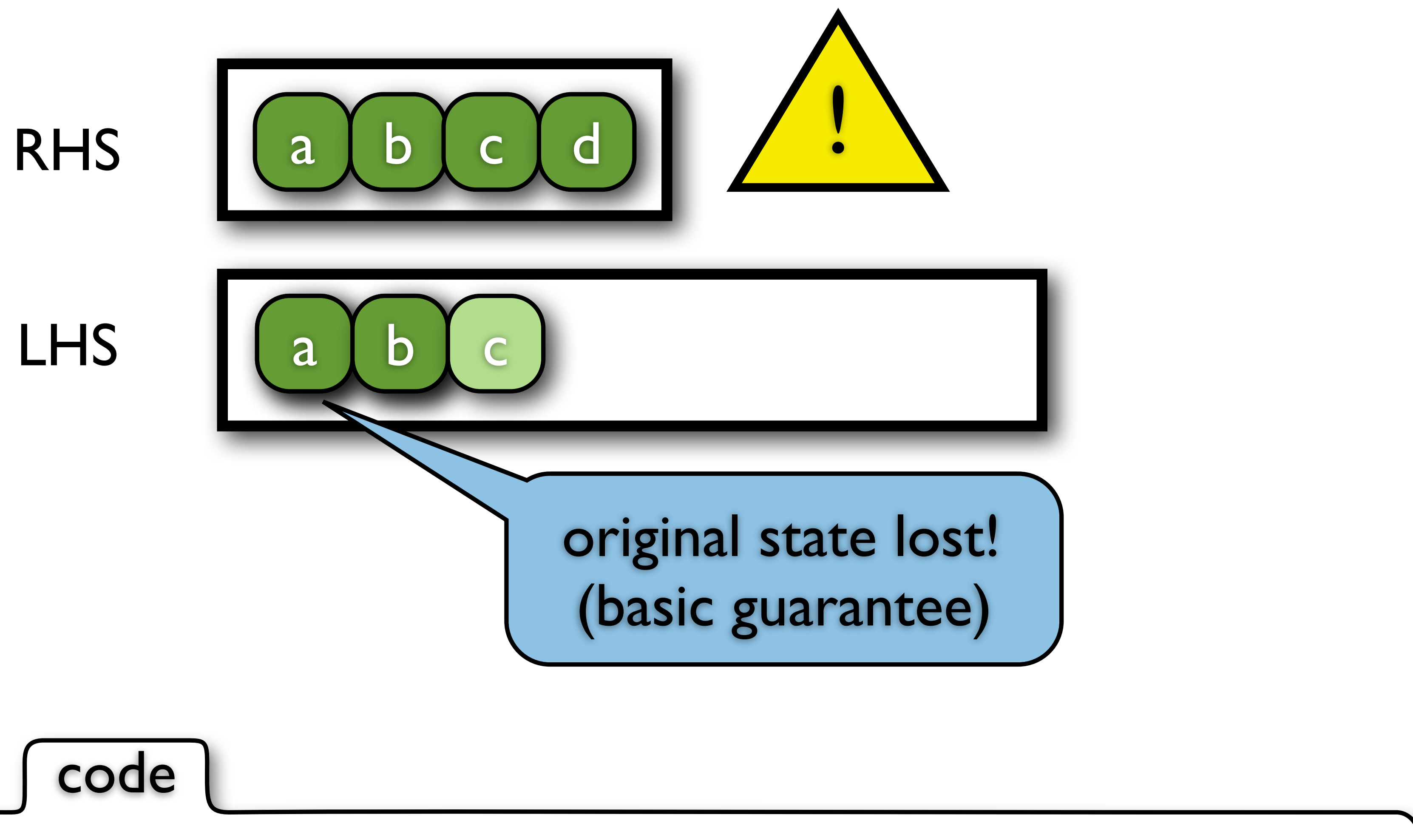
Observation

The *strong guarantee* depends on a partitioning in the sequence of sub-operations



vector<T>::push_back is "naturally strong." There's (basically) no other way to implement it

Assigning std::vector



Why not copy / swap assignment for std::vector?

Constructors that Throw?

```
struct string
{
   string( char const*const s )
     : len( std::strlen( s ) )
     , buf( new char[len+1] )
   {
        std::strcpy( buf, s );
   }

   ~string() { delete [] buf; }

   string( string const& rhs )
     : len( rhs.len ),
       buf( new char[len+1] )
   {
        std::strcpy( buf, rhs.buf );
   }
```

Q: What if we avoid throwing from constructors?

A: Invariants are weakened (need an empty state)

Q: But how can this report failure?

A Match Made in Heaven

Why? Because without exceptions, we'd have to weaken some class invariants

boostpro computing

Beware “2-phase” construction!

Destructors that Throw?

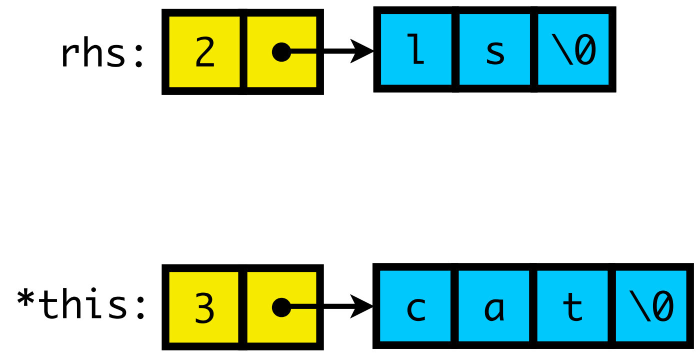
```
struct X
{
    ~X() { something_that_might_throw(); }
};

void g( std::string message );

void f()
{
    X a;
    g( "this is the end" );
    a.~X();
}
```

Might throw

If this throws too? Doom.

A Missed Opportunity

```
  string( string const& rhs )
    : len( rhs.len ),
      buf( new char[len+1] )
  {
       std::strcpy( buf, rhs.buf );
  }

  string& operator=( string rep )
  {
       swap( *this, rep );
       return *this;
  }

  friend void swap( string&, string& );
private:
  std::size_t len;
  char* buf;
};
```



"Composite Ownership"

```
// reactor.hpp
class Controller;
class TSensor;

class Reactor
{
 public:
   Reactor();
   ~Reactor();
   run();
 private:
   // copy ctor, copy assignment
   // go here
   TSensor* s;
   Controller* c;
};
```

```
// reactor.cpp
class Controller { … };
class TSensor { … };

Reactor::~Reactor()
{
   delete c;
   delete s;
}

Reactor::Reactor()
 : s( new TSensor ),
   c( new Controller(s) )
{}
```


Note careful deletion order

“Composite Ownership”

```
// reactor.hpp
class Controller;
class TSensor;

class Reactor
  : boost::noncopyable
{
 public:
   Reactor();
   ~Reactor();
   run();
 private:
   TSensor* s;
   Controller* c;
};
```

```
// reactor.cpp
class Controller { … };
class TSensor { … };

Reactor::~Reactor()
{
   delete c;
   delete s;
}

Reactor::Reactor()
 : s( new TSensor ),
   c( new Controller(s) )
{}
```


Note careful deletion order

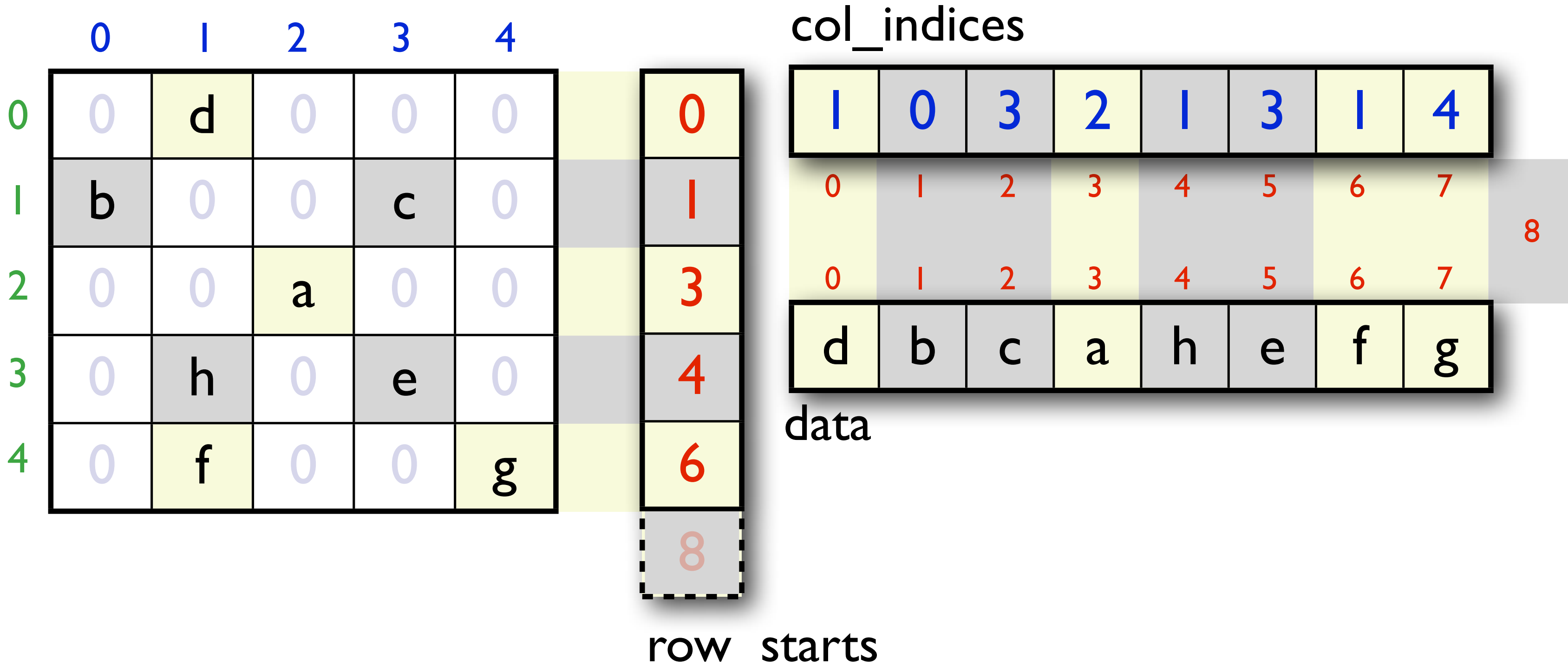
Fixing the Leak

```
// reactor.hpp
class Controller;
class TSensor;

class Reactor
  : boost::noncopyable
{
 public:
   Reactor();
   ~Reactor();
   run();
 private:
   TSensor* s;
   Controller* c;
};
```

```
// reactor.cpp
class Controller { … };
class TSensor { … };

Reactor::~Reactor()
{ delete c; delete s; }

Reactor::Reactor()
  : s ( new TSensor ), c( 0 )
{
   try {
        c = new Controller( s );
   }
   catch(…) { delete s; throw; }
}
```


Multiphase construction
Reminiscent of the bad ol' days
Careful, yet fragile under maintenance.
Why didn't our string have this problem?

Exercise: Compressed Sparse Row Matrix



Exercise: Implement Sparse Matrix

- http://www.filetolink.com/c8adce0a
  - Wed/sparse_eh.cpp
- allocate memory with new T[n]
- release memory with delete[] p
- Make it exception-safe
- What guarantee does your operator= offer?

boostpro computing

Raw Pointers

+1

- Easy to come by
- Efficient

−1

- Uninitialized
- Dangling
- Leaks
- Double delete
- Ownership semantics
- Array semantics
- Delete slicing
- Deletion of incomplete type
- Deletion in wrong heap

boostpro computing

Overcoming fear of failure

- Principle: Ownership
  - Every resource is owned by an object whose sole responsibility is to free the resource
- Resource acquisition is initialization (RAII)
  - Constructor acquires resource
  - Destructor frees resource

Deleting an Incomplete Type

```
#include <set>

class Z;      // forward declaration
std::set<Z*> known_zs;

destroy(Z* a)
{
    std::set<Z*>::iterator p = known_zs.find(a);
    known_zs.erase(p);
    delete a;
}
```

Boost solution: `checked_delete( z );`

Deletion in Wrong Heap

```
#include "X.hpp"
X* producer()
{
    return new X;
}
```

Statically-linked C++ runtime

```
#include "X.hpp"
void consumer(X* x)
{
    delete x;
}
```

Statically-linked C++ runtime

```
int main()
{
    consumer( producer() );
}
```

boostpro computing

“What we’ve got here is…

…failure to communicate.”

Smart Pointers

- Class types—usually instances of class templates
- “Like built-in pointer types” but with attached semantics, especially *destruction semantics*
- Usually a subset of the built-in pointer interface
- Make dynamic memory safe(r) to handle
- Communicate intent to readers/maintainers

boostpro computing

Basics of `std::auto_ptr<T>`

- One data member: `T* px;`
- Dereference semantics ("pointer-ness"):

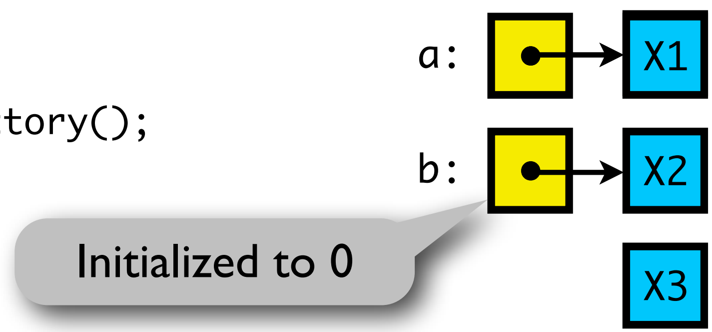
```
T& operator*() const { return *px; }
T* operator->() const { return px; }
```

- Destructor semantics: `delete px;` ("smartness")
- ***Transfer-of-ownership*** on copy and assignment
- ***Not an array pointer*** (no indexing, arithmetic, or order comparison)

Also has weird and wonderful transfer-of-ownership

auto_ptr Examples

```
#include "X.hpp"
#include <memory>
std::auto_ptr<X> factory()
{
    return std::auto_ptr<X>( new X ); // ok
}
}

void dump( std::auto_ptr<X> );

int main()
{
    std::auto_ptr<X> a = factory();
    std::auto_ptr<X> b;
    b = a;
    dump( b );
    a.reset( new X );
    delete a.release();
    b = factory();
    b.reset( new X );
}
```



boostpro computing

auto_ptr Caveats

- Can't place in std containers
- Can't use with std algorithms
- That goes for classes with `auto_ptr` members and default copy/assignment too

```
template <class T>
struct auto_ptr
{
               auto_ptr( auto_ptr& rhs );
    auto_ptr& operator=( auto_ptr& rhs );
    ~auto_ptr() { delete px; }
    bool operator==( auto_ptr const& rhs);
    ...
```

Transfer-of-ownership mutates rhs

T could be incomplete here!

boostpro computing

boost::scoped_ptr<T>

```
template <class T> struct scoped_ptr : noncopyable
{
   typedef T element_type;
   explicit scoped_ptr(T* p = 0)    : px(p) {}
   ~scoped_ptr()                   { boost::checked_delete(px); }
   void reset(T * p = 0)           { scoped_ptr(p).swap(*this); }
   T & operator*() const           { return *px; }
   T * operator->() const          { return px; }
   T * get() const                 { return px; }
   void swap(scoped_ptr & b)       { std::swap(this->px, b.px); }


   friend void swap( scoped_ptr& a, scoped_ptr& b)     { a.swap(b); }
  private:
   T* px;
};
```

boost::scoped_array<T>

```
template <class T> struct scoped_array : noncopyable
{
   typedef T element_type;
   explicit scoped_array(T * p = 0) : px(p) {}
   ~scoped_array()              { boost::checked_array_delete(px); }
   void reset(T * p = 0)        { scoped_array(p).swap(*this); }
   T & operator*() const        { return *px; }
   T * operator->() const       { return px; }
   T * get() const              { return px; }
   void swap(scoped_array& b) { std::swap(this->px, b.px); }
   T& operator[](std::ptrdiff_t i) const   { return px[i]; }

   friend void swap( scoped_array& a, scoped_array& b) { a.swap(b); }
  private:
   T* px;
};
```

Fixing the Leak

```
// reactor.hpp
class Controller;
class TSensor;

class Reactor
  : boost::noncopyable
{
 public:
   Reactor();
   ~Reactor();
   run();
 private:
   TSensor* s;
   Controller* c;
};
```

```
// reactor.cpp
class Controller { … };
class TSensor { … };

Reactor::~Reactor()
{ delete c; delete s; }

Reactor::Reactor()
  : s ( new TSensor ), c( 0 )
{
   try {
        c = new Controller( s );
   }
   catch(…) { delete s; throw; }
}
```


Multiphase construction
Reminiscent of the bad ol' days
Careful, yet fragile under maintenance.
Why didn't our string have this problem?

A Cleaner Fix

```
// reactor.hpp
class Controller;
class TSensor;


class Reactor
{
 public:
   Reactor();
   ~Reactor();
   run();
 private:
   boost::scoped_ptr<TSensor> s;
   boost::scoped_ptr<Controller> c;
};
```

```
// reactor.cpp
class Controller { … };
class TSensor { … };

Reactor::~Reactor()
{}

Reactor::Reactor()
  : s ( new TSensor )
  , c( new Controller(s.get()) )
{}
```


Can we get rid of the dtor? It's empty
A: no, but the compiler will catch us if we do.
A2: we could move the class definitions into the header.

Resource Management Gotcha

```
void g();

int f( std::auto_ptr<X> a, int b );


int a = f( std::auto_ptr<X>(new X), g() );
```

boostpro computing

The Dimov Directive

```
void g();

int f( std::auto_ptr<X> a, int b );

std::auto_ptr<X> new_x(new X);

int a = f( new_x, g() );
```

Assign ownership of every resource, immediately upon allocation, to a **named** manager object that manages no other resources

Exercise: ReImplement Sparse Matrix

- allocate memory with new T[n]
- release memory with delete[] p
- Make it exception-safe
- What guarantee does your operator= offer?
- Reimplement it using boost::scoped_array
- Did that simplify your code?

Exercise: ReImplement Sparse Matrix

- allocate memory with new T[n]
- release memory with delete[] p
- Make it exception-safe
- What guarantee does your operator= offer?
- Reimplement it using boost::scoped_array
- Reimplement it using std::vector
- Did that simplify your code?

`boost::scoped_ptr`
Solution Checklist

- ✓ Leaks
- ✓ Double Delete
- ✓ Uninitialized
- ✓ Array Semantics
- ✓ Ownership Semantics
- ✓ Deletion of Incomplete Type

- ➡ Delete Slicing
- ➡ Use in std::containers
- ➡ Deletion in wrong heap
- ➡ Dangling

boostpro computing

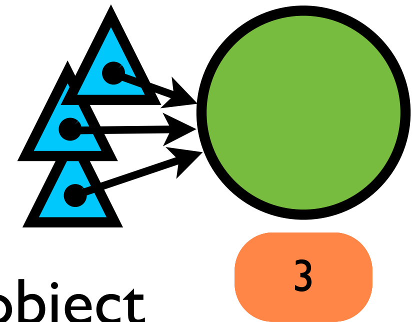
boost::scoped_ptr
Delete Slicing

```
struct Base
{
    ~Base(); // non-virtual
};

struct Derived : Base
{
    ~Derived();
};

scoped_ptr<Derived> x( new Derived );

scoped_ptr<Base> y( x );

scoped_ptr<Base> x( new Derived );
```

✓ Compile-time error: no such conversion

X Destruction ⇒ undefined behavior

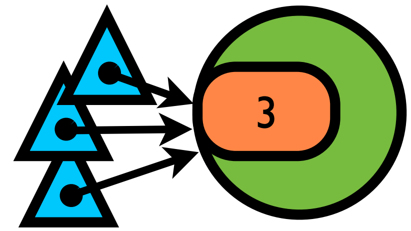
Reference Counted Smart Pointer



- Basic strategy:
  - Count ptrs that refer to each object
  - Delete object when count goes to zero
- Implements shared ownership model
- Has proper *value semantics*: can be copied, put in containers, used with algorithms, etc.

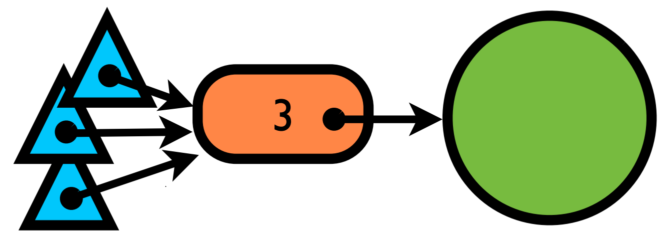
boostpro computing

Intrusive Ref-Counting



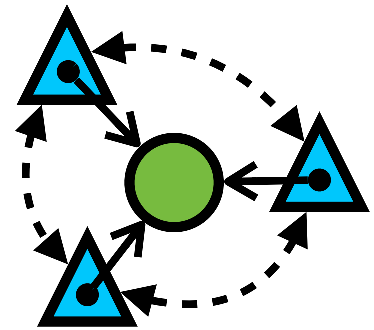
- \+ Extremely lightweight
- \+ Raw ⇒ smart pointer conversion *always* safe
- \+ Boost's `intrusive_ptr` is très cool
- \- Reference count "intrudes" on design of pointee
- \- Doesn't address delete slicing, deletion in wrong heap, or dangling

Indirect Ref-Counting



- \+ Non-intrusive
- \- Costs an allocation
- \- Can only convert raw ⇒ smart pointer once
- \- Can't support derived-to-base conversions
- \- Doesn't address deletion in wrong heap, dangling

Cyclic Ref-Counting

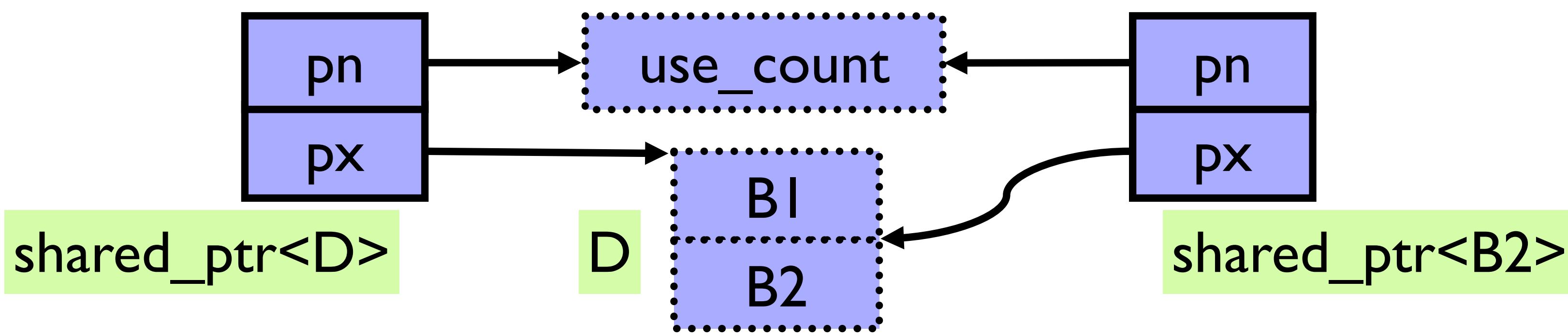


- \+ Non-intrusive
- \+ No count allocation needed
- \- Can only convert raw ⇒ smart pointer once
- \- Can't support derived-to-base conversions
- \- Doesn't address deletion in wrong heap, dangling
- \- Very hard / expensive to make thread-safe

boostpro computing

boost::shared_ptr
Basic Strategy

- Store pointers to count and pointee separately
- Allows derived-to-base conversion



boostpro computing

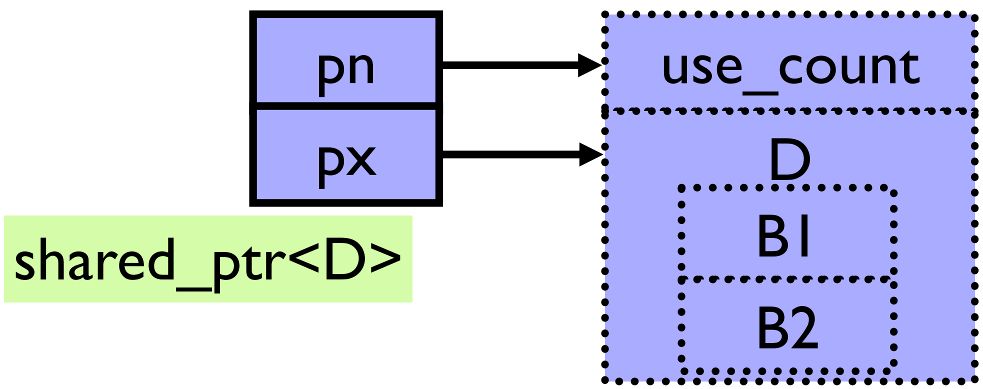
Gotcha, Revisited

```
void g();

int f( boost::shared_ptr<D> a, int b );

int a = f( boost::shared_ptr<D>( new D("foo", 3) ), g() );
```

Gotcha, Revisited

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

void g();

int f( boost::shared_ptr<D> a, int b );

int a = f( boost::make_shared<D>("foo", 3), g() );
```

No unmanaged resources exposed!



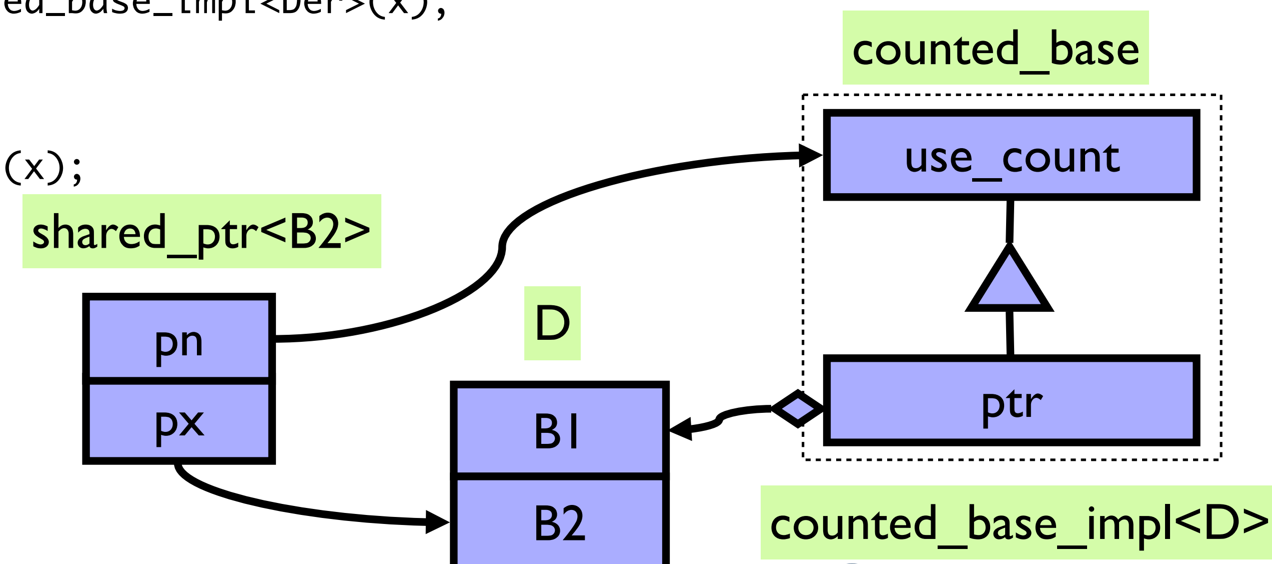
No separate use_count allocation!

boostpro computing

Solving the delete slicing problem

```
template <class Base>
template <class Der>
shared_ptr<Base>::shared_ptr(Der* x)
  : px(x)
{
    try
    {
        pn = new counted_base_impl<Der>(x);
    }
    catch(…)
    {
        checked_delete(x);
        throw;
    }
}
```

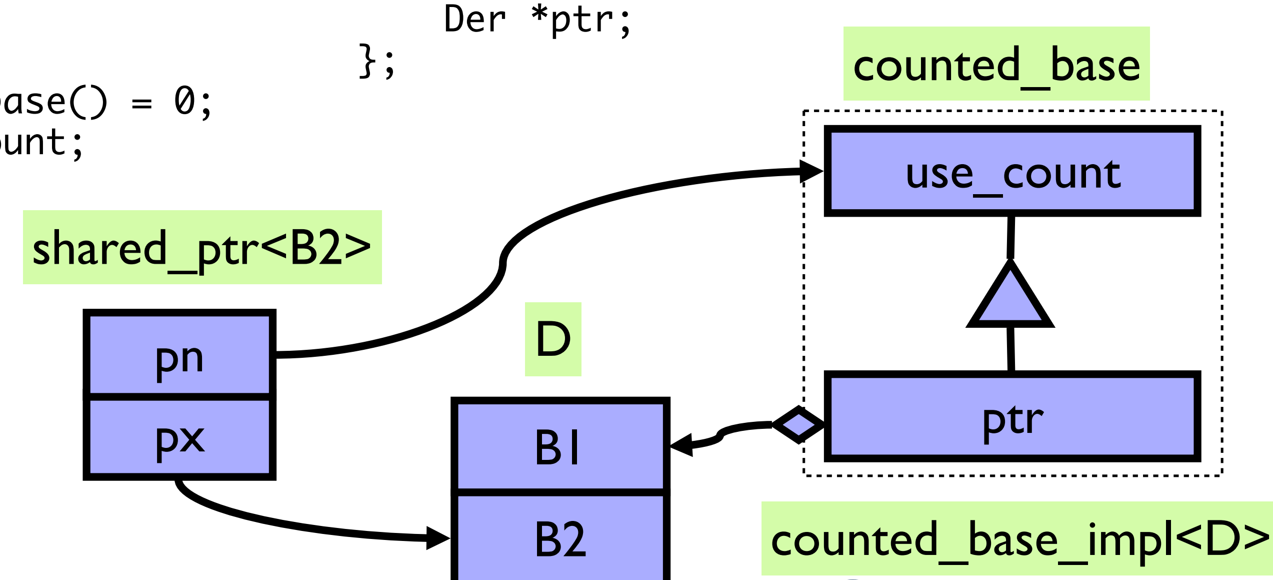
```
shared_ptr<B2>
factory()
{
  return shared_ptr<B2>(new D);
}
```



Solving the delete slicing problem

```
template <class Base>
shared_ptr<Base>::~shared_ptr()
{
    if (--pn->use_count == 0)
        delete pn;
}

struct counted_base
{
    virtual ~counted_base() = 0;
    std::size_t use_count;
};
```

```
template <class Der>
struct counted_base_impl
  : counted_base
{
    ~counted_base_impl()
    { delete ptr; }

    Der *ptr;
};
```



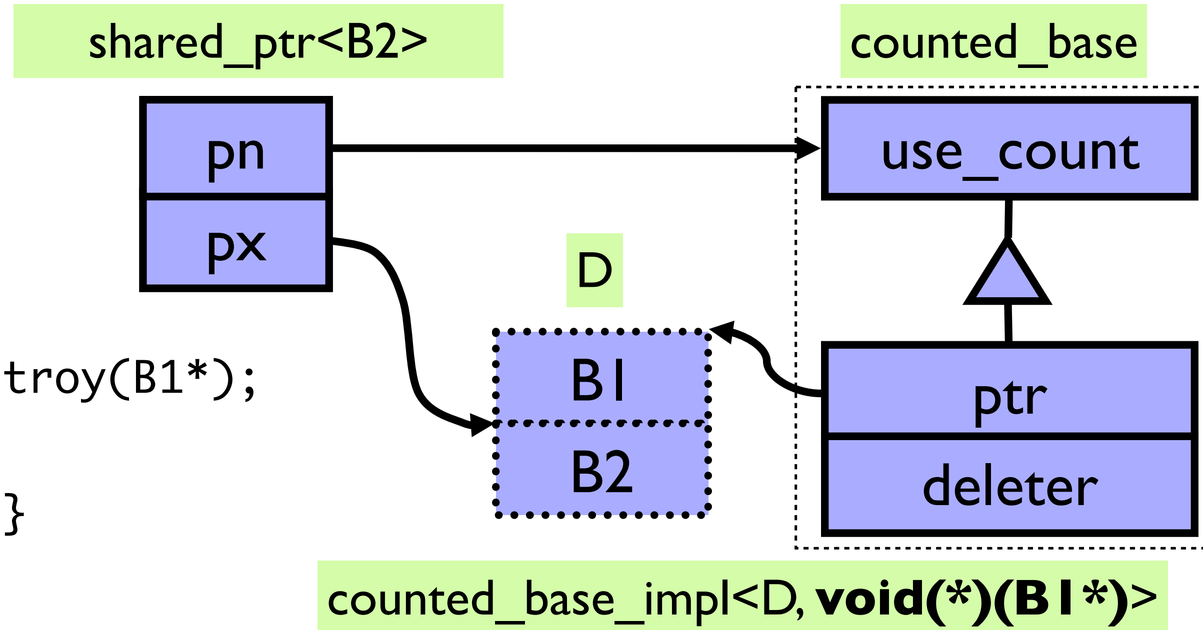
shared_ptr doesn’t delete px!

What have we gained?

- Derived⇒base conversion loses no destruction info (no more delete slicing)!
- Type to delete *and* deletion heap determined at moment of initialization (no more deletion in wrong heap)!
- Template parameter may be an incomplete type at `shared_ptr` destruction!
- Manage life of everything with `shared_ptr<void>`

boostpro computing

Low-hanging fruit: Custom Deleters




```
struct B1
{
    static void destroy(B1*);
 private:
    virtual ~B1() {}
};
```

```
B1 b1;                   // error, private destructor!
D d;                     // ditto
shared_ptr<B2> p(new D, B1::destroy); // OK
```

`boost::shared_ptr` Solution Checklist

- ✓ Leaks
- ✓ Double Delete
- ✓ Uninitialized
- ✓ Array Semantics
- ✓ Ownership Semantics
- ✓ Deletion of incomplete type ✓
- ✓ Delete Slicing ✓
- ✓ Use in std:: containers
- ✓ Deletion in wrong heap
- ➡ Dangling

boostpro computing

Solving the Dangling Problem

- Problem: battlefield simulation
- In each timeslice, GameObjects may:
    - Acquire a target, or
    - Fire on the target acquired last slice (if still alive)
- Target acquisition must not keep the target alive (shared_ptr<> won't work)
- Some other GameObject might kill the target before us (raw pointer won't work)

boostpro computing

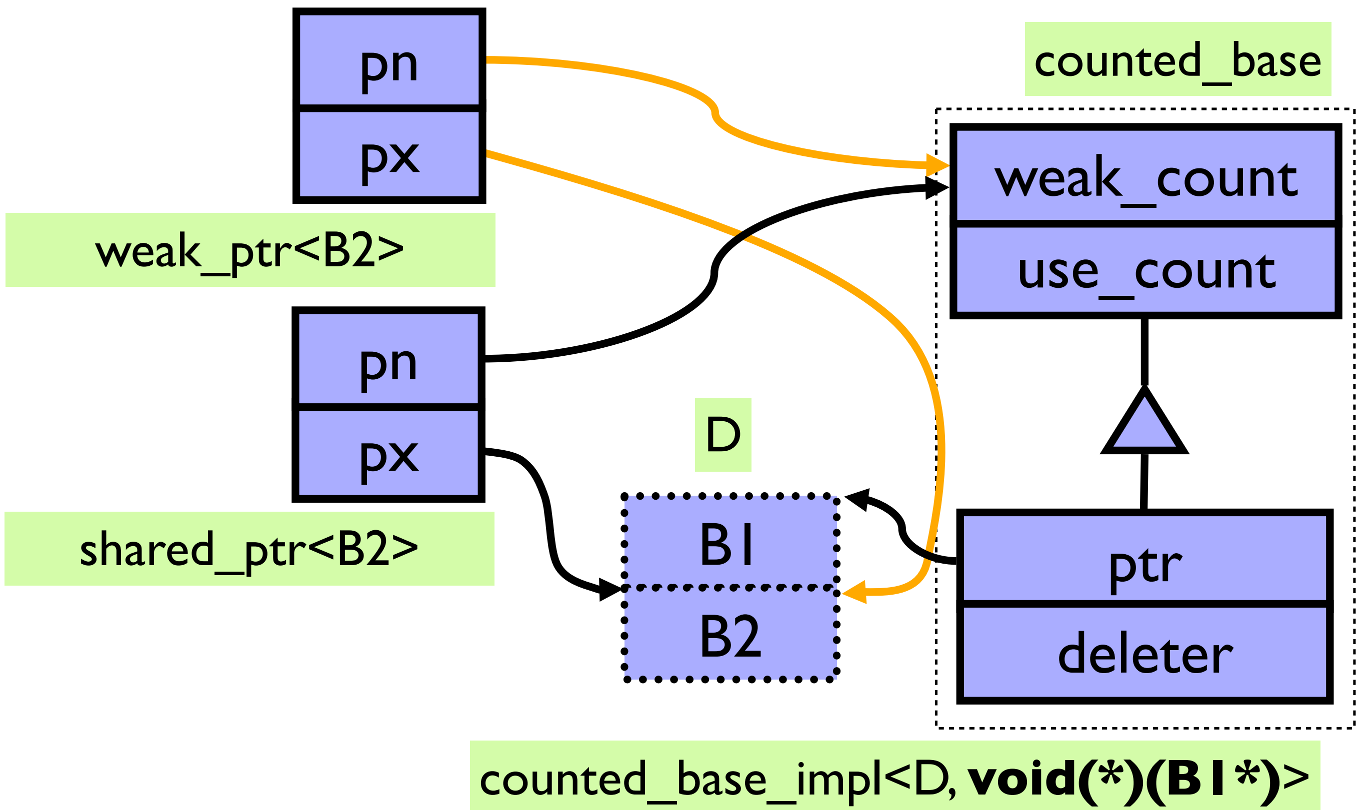
boost::weak_ptr<T> overview

- Implicit conversion/assignment from shared_ptr<T>
- Can be converted to shared_ptr<T>
  - explicit construction: throws if target deleted
  - w.lock(): NULL if target deleted
- No dereference operations! (Not a ptr?)
- Good for child-to-parent pointers

Solving the Dangling Problem

```
std::set< shared_ptr<GameObject> > all_targets;

struct Tank : GameObject
{
    virtual void time_slice()
    {
        shared_ptr<GameObject> p = this->target.lock();

        if (p)
            p->damage(500);
        else
            this->target = this->select_target();
    }

    shared_ptr<GameObject> select_target();

 private:
    weak_ptr<GameObject> target;
};
```

weak_ptr implementation



boostpro computing

`boost::shared_ptr` Solution Checklist

- ✓ Leaks
- ✓ Double Delete
- ✓ Uninitialized
- ✓ Array Semantics
- ✓ Ownership Semantics
- ✓ Deletion of incomplete type ✓
- ✓ Delete Slicing ✓
- ✓ Use in std:: containers
- ✓ Deletion in wrong heap
- ✓ Dangling

Reference Cycles

- Usually there's a natural ownership relationship

```
struct tree_node
{
    void add_child(shared_ptr<tree_node> c)
    {
        this->children.push_back( c );
        c->parent = this;
    }

 private:
    weak_ptr<tree_node> parent;        // OK
    std::list<shared_ptr<tree_node> > children;
};
```

Reference Cycles

```
struct tree_node
{
    void add_child(shared_ptr<tree_node> c)
    {
        this->children.push_back( c );
        c->parent = ???;
    }

 private:
    weak_ptr<tree_node> parent;
    std::list<shared_ptr<tree_node> > children;
};
```

Reference Cycles

```
struct tree_node
{
    static void parent_child(
      shared_ptr<tree_node> p, shared_ptr<tree_node> c)
    {
        parent->children.push_back( c );
        c->parent = p;
    }

 private:
    weak_ptr<tree_node> parent;
    std::list<shared_ptr<tree_node> > children;
};
```

Raw ⇒ Shared

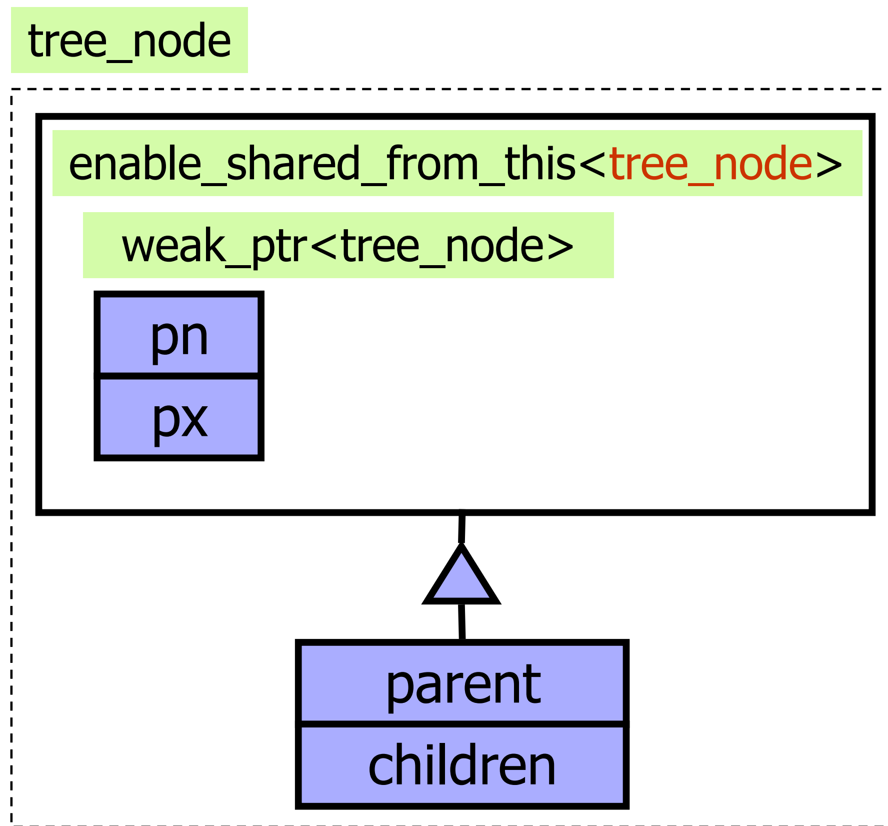
```
struct tree_node
{
    void add_child(shared_ptr<tree_node> c)
    {
        this->children.push_back( c );
        c->parent = ???;
    }

 private:
    weak_ptr<tree_node> parent;
    std::list<shared_ptr<tree_node> > children;
};
```

boostpro computing

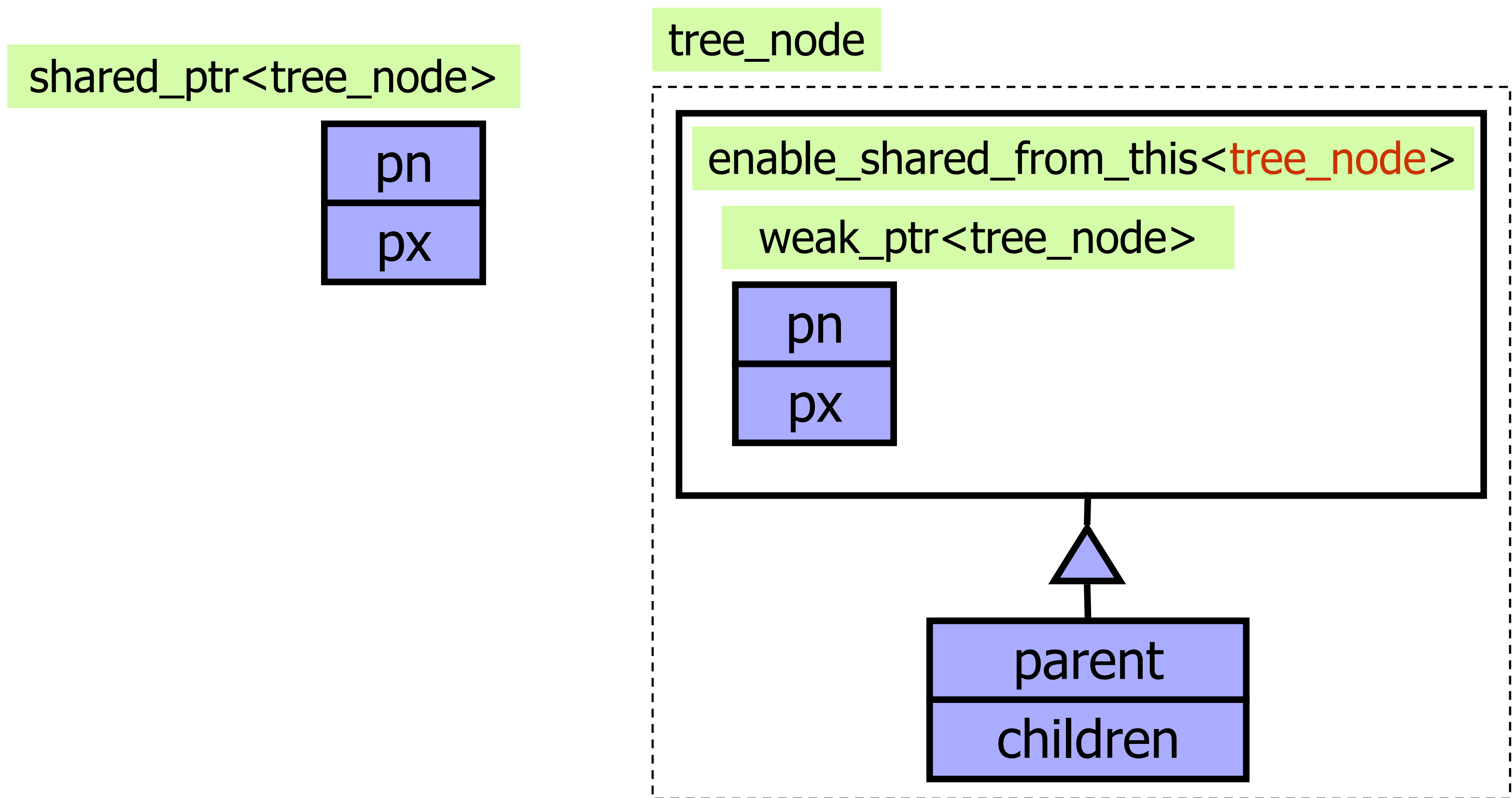
Raw ⇒ Shared

```
struct tree_node
  : boost::enable_shared_from_this<tree_node>
{
    void add_child(shared_ptr<tree_node> c)
    {
        this->children.push_back( c );
        c->parent = shared_from_this();
    }

 private:
    weak_ptr<tree_node> parent;
    std::list<shared_ptr<tree_node> > children;
};
```

Raw ⇒ Shared

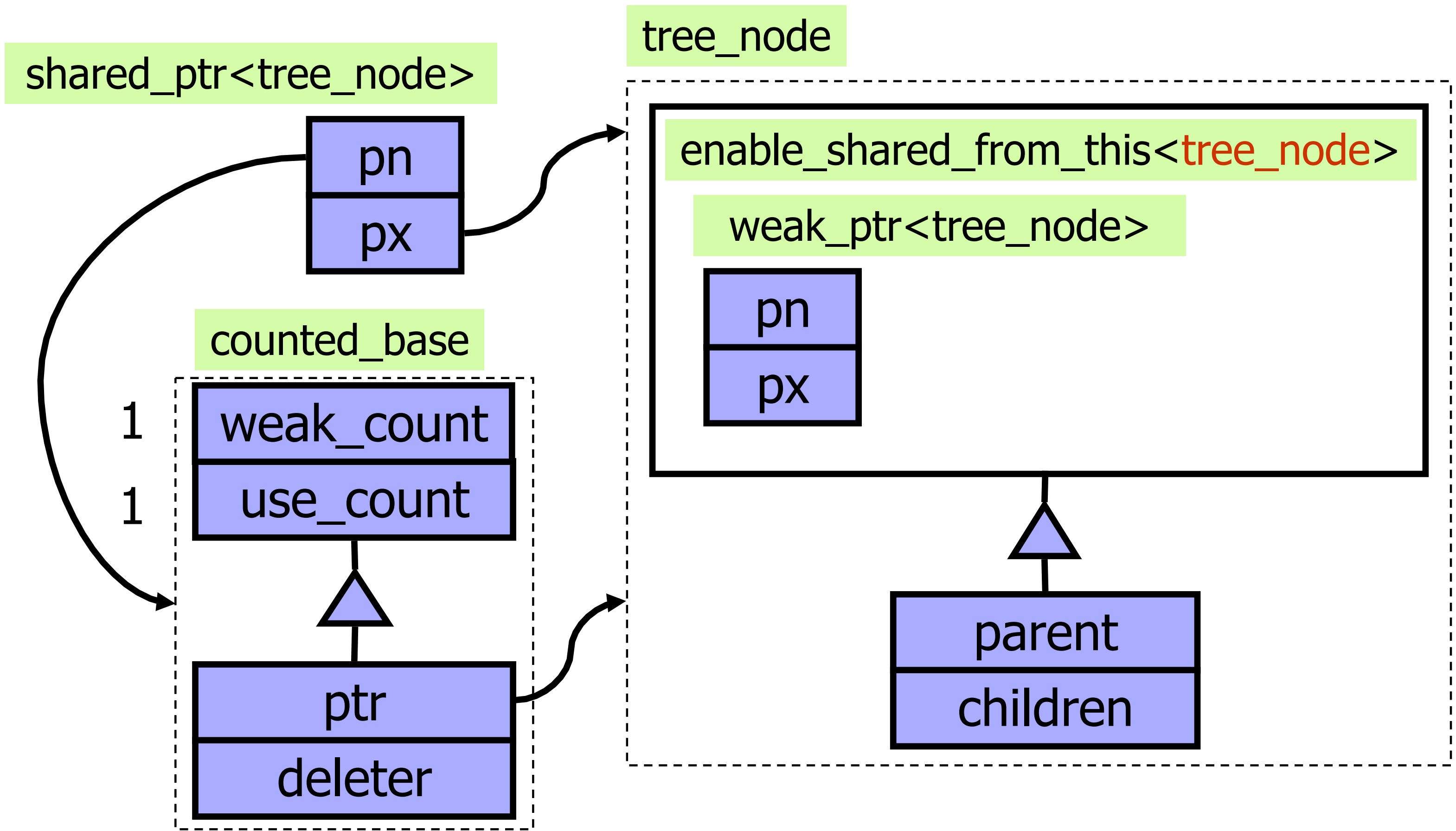

tree_node
enable_shared_from_this<tree_node>
weak_ptr<tree_node>
pn
px
parent
children

Raw ⇒ Shared



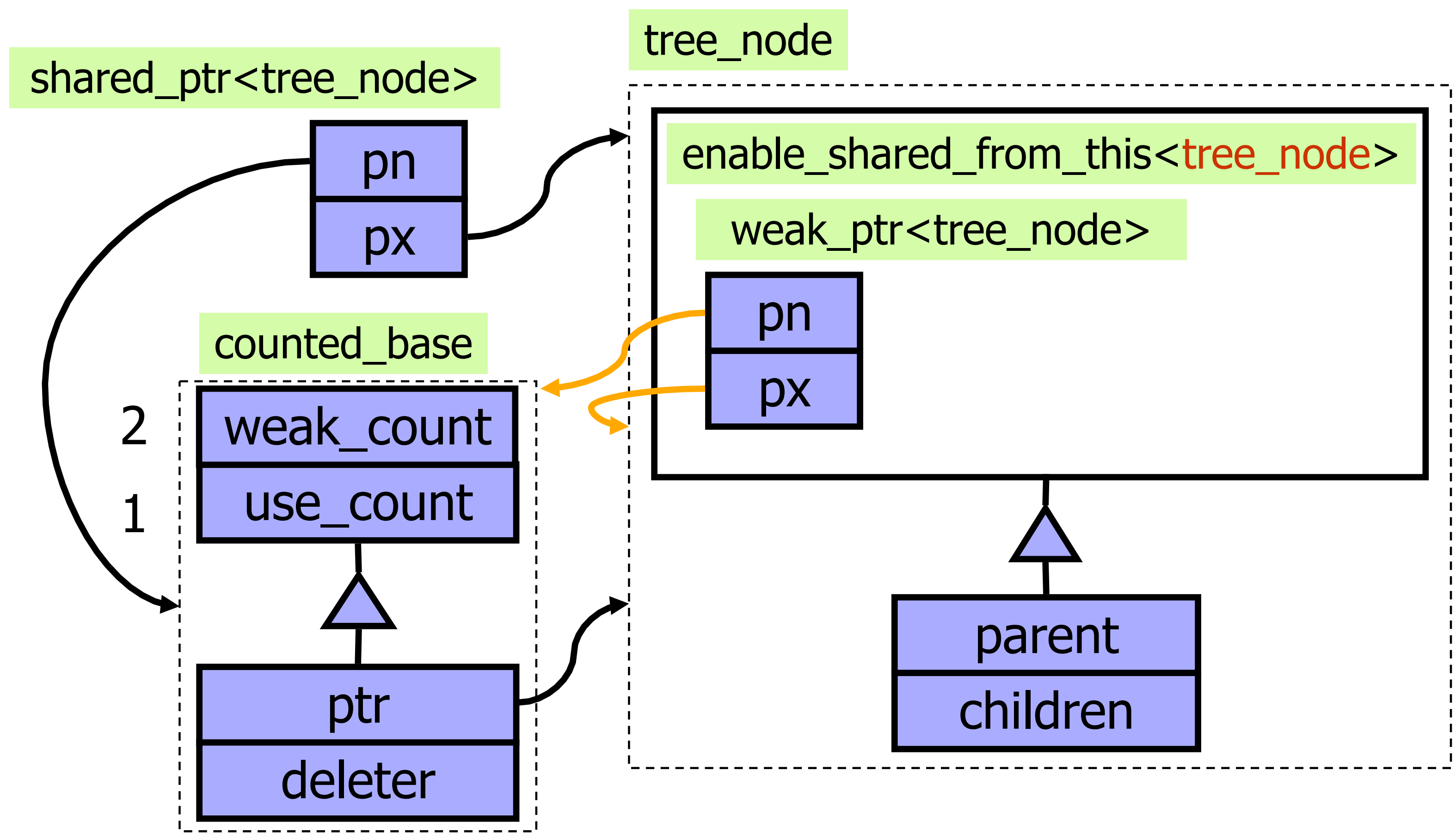
shared_ptr<tree_node>
pn
px
tree_node
enable_shared_from_this<tree_node>
weak_ptr<tree_node>
pn
px
parent
children

Raw ⇒ Shared



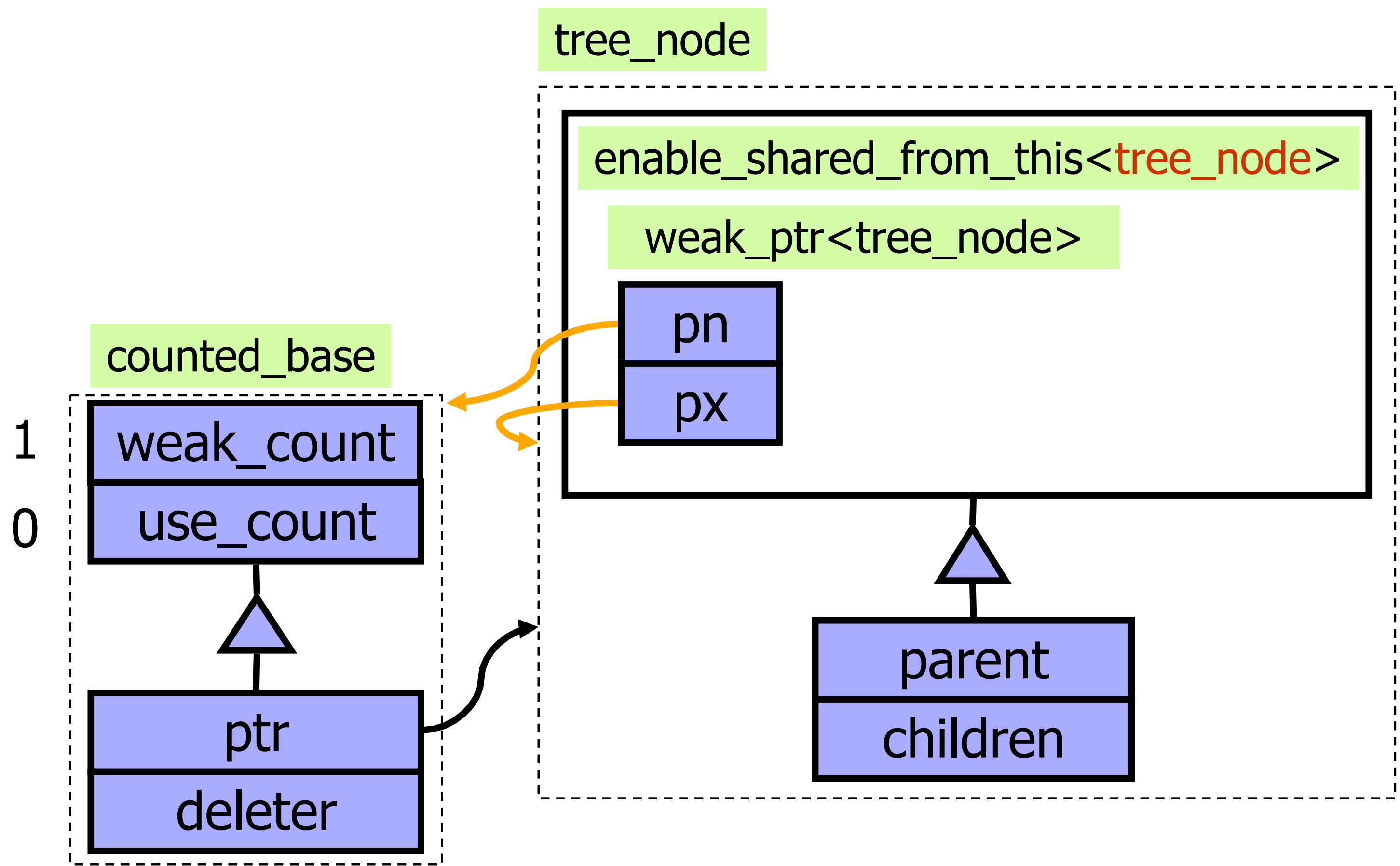
shared_ptr<tree_node>
pn
px
counted_base
1
weak_count
1
use_count
ptr
deleter
tree_node
enable_shared_from_this<tree_node>
weak_ptr<tree_node>
pn
px
parent
children

Raw ⇒ Shared



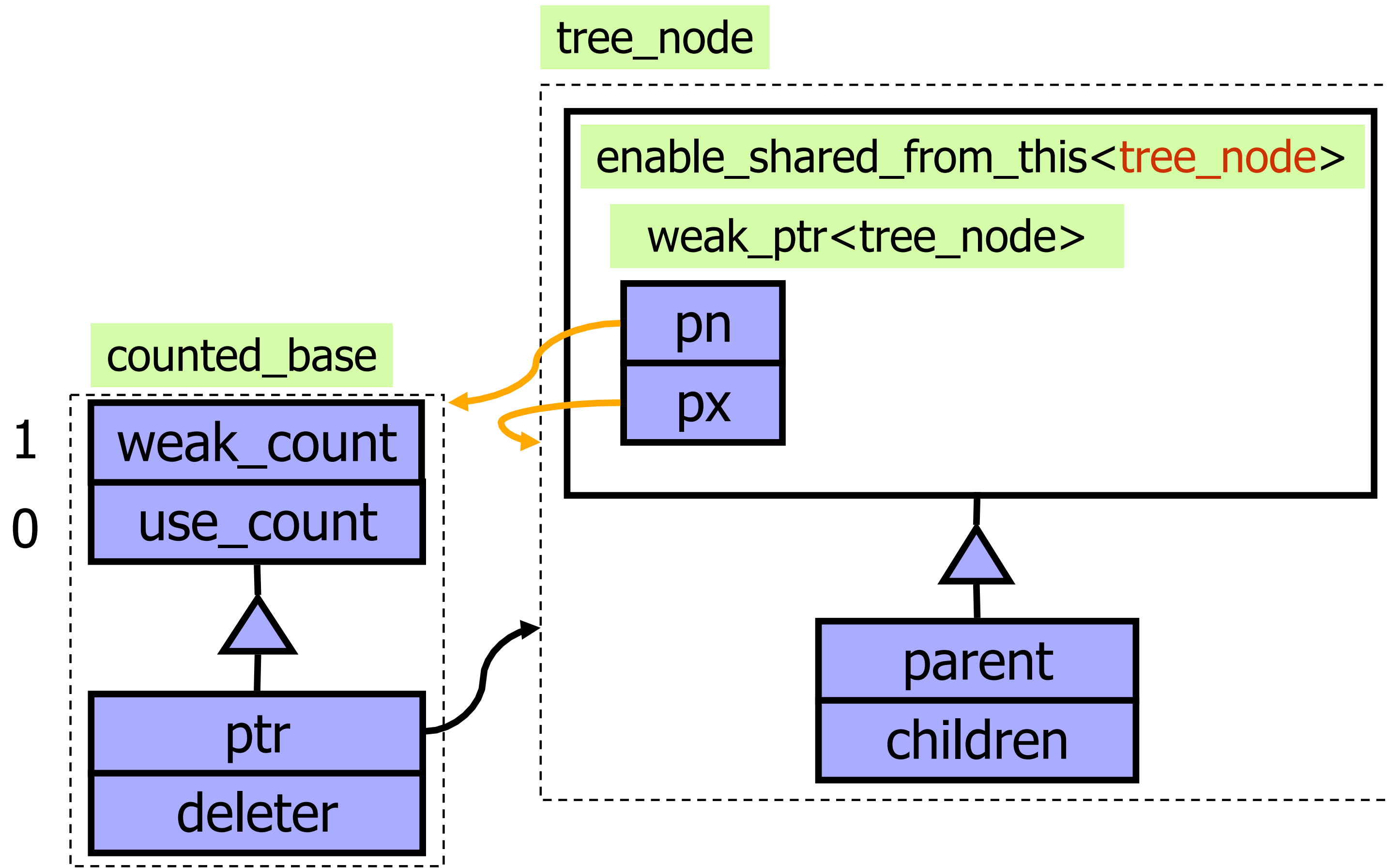
boostpro computing

Raw ⇒ Shared



boostpro computing

Raw ⇒ Shared


tree_node
enable_shared_from_this<tree_node>
weak_ptr<tree_node>
pn
px
counted_base
1
0
weak_count
use_count
ptr
deleter
parent
children

Exercise 5: GameObject Simulation

- In each timeslice, GameObjects:
  - Acquire a target, or
  - Fire on acquired target
- Firing on acquired target may destroy it
- Multiple GameObjects may acquire the same target
- Extra credit: Using a deleter, make it impossible for anyone to destroy a Tank except through a shared_ptr