Teuchos::RCP

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett

Department 1411: Optimization and Uncertainty Estimation

Sandia National Laboratories
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- RCP: Nuts & bolts and avoiding trouble
- Summary and RCP philosophy
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- RCP: Nuts & bolts and avoiding trouble
- Summary and RCP philosophy
Dynamic Memory Management in C++

- C++ requires programmers to manage dynamically allocated memory themselves using `operator new` and `operator delete`.

- Problems with using raw `new` and `delete` at application programming level:
  - Very error prone (multiple deletes or memory leaks)
  - Difficult to know who’s responsibility it is to delete an object
  - Creates memory leaks when exceptions are thrown

- Reference counting to the rescue?

  ![UML object diagram]

  - Examples (C++ and other):
    - CORBA
    - COM

  - How is the reference count updated and managed?
  - When is the object deleted?
  - How is the object deleted?
Smart Pointers: A C++ Reference Counting Solution

- C++ supports development of “smart” pointer classes that behave a lot like raw pointers

- Reference counting + smart pointers = smart reference counted pointers

Advantages of Smart Pointer approach:
- Access to shared object uses pointer-like syntax
  - (*ptr).f() [operator*()]
  - ptr->f() [operator->()]
- Reference counts automatically updated and maintained
- Object automatically deleted when last smart pointer is destroyed

Examples of C++ smart reference counted pointer classes
- boost::shared_ptr: Part of the Boost C++ class library (created in 1999?)
  - Being considered to go into the next C++ standard
  - Does not throw exceptions
- Teuchos::RCP:
  - Originally developed as part of rSQP++ (Bartlett et. al.) in 1998
  - Does throw exceptions in some cases in debug mode and has addition features
  - Being used more and more extensively in many Trilinos packages such as Thyra, NOX/LOCA, Rythmos, Belos, Anasazi, ML, …
Outline

- Background on memory management in C++
  - Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
  - RCP: Nuts & bolts and avoiding trouble
- Summary and RCP philosophy
Introduction of Simple Example Program

  - Complete program listings in Appendix E and F
- Uses basic object-oriented programming
- Demonstrates the basic problems of using raw C++ pointers and delete for high-level memory management
- Provides context for differentiating “persisting” and “non-persisting” object associations
- Show step-by-step process of refactoring C++ code to use RCP
Utility interface and subclasses

```cpp
class UtilityBase {
public:
    virtual void f() const = 0;
};
class UtilityA : public UtilityBase {
public:
    void f() const {...}
};
class UtilityB : public UtilityBase {
public:
    void f() const {...}
};
```

Utility “abstract factory” interface and subclasses (see “Design Patterns” book)

```cpp
class UtilityBaseFactory {
public:
    virtual UtilityBase* createUtility() const = 0;
};
class UtilityAFactory : public UtilityBaseFactory {
public:
    UtilityBase* createUtility() const { return new UtilityA(); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
    UtilityBase* createUtility() const { return new UtilityB(); }
};
```
class ClientA {
public:
    void f(const UtilityBase &utility) const { utility.f(); }
};

class ClientB {
    UtilityBase *utility_;
public:
    ClientB() : utility_(0) {}
    ~ClientB() { delete utility_; }
    void initialize(UtilityBase *utility) { utility_ = utility; }
    void g(const ClientA &a) { a.f(*utility_); }
};

class ClientC {
    const UtilityBaseFactory *utilityFactory_;
    UtilityBase *utility_;
    bool shareUtility_;
public:
    ClientC(const UtilityBaseFactory *utilityFactory, bool shareUtility)
        :utilityFactory_(utilityFactory)
        ,utility_(utilityFactory->createUtility())
        ,shareUtility_(shareUtility) {}
    ~ClientC() { delete utilityFactory_; delete utility_; }
    void h(ClientB *b) {
        if(shareUtility_) b->initialize(utility_);
        else b->initialize(utilityFactory_->createUtility());
    }
};
```cpp
int main( int argc, char* argv[] )
{
    // Read options from the commandline
    bool useA, shareUtility;
    example_get_args(argc, argv, &useA, &shareUtility);
    // Create factory
    UtilityBaseFactory *utilityFactory = 0;
    if(useA) utilityFactory = new UtilityAFactory();
    else utilityFactory = new UtilityBFactory();
    // Create clients
    ClientA a;
    ClientB b1, b2;
    ClientC c(utilityFactory, shareUtility);
    // Do some stuff
    c.h(&b1);
    c.h(&b2);
    b1.g(a);
    b2.g(a);
    // Cleanup memory
    delete utilityFactory;
}
```

This program has memory usage problems!
Example Program Memory Usage Problem #1

class ClientC {
  ...
  public:
    ClientC( const UtilityBaseFactory *utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory),
      utility_(utilityFactory->createUtility()),
      shareUtility_(shareUtility) {}
    ~ClientC() { delete utilityFactory_; delete utility_; }
  ...
};

int main( int argc, char* argv[] )
{
  ...
  // Create factory
  UtilityBaseFactory *utilityFactory = 0;
  if(useA) utilityFactory = new UtilityAFactory();
  else    utilityFactory = new UtilityBFactory();
  // Create clients
  ...
  ClientC c(utilityFactory,shareUtility);
  // Do some stuff
  ...
  // Cleanup memory
  delete utilityFactory;
}
class ClientB {
    UtilityBase *utility_; 
public:
    ~ClientB() { delete utility_; }
    void initialize( UtilityBase *utility ) { utility_ = utility; }
};

class ClientC {
    const UtilityBaseFactory *utilityFactory_; 
    UtilityBase *utility_; 
    bool shareUtility_; 
public:
    ... 
    ~ClientC() { delete utilityFactory_; delete utility_; } 
    void h( ClientB *b ) { 
        if( shareUtility_ ) b->initialize(utility_); 
        else b->initialize(utilityFactory_->createUtility());
    }
};

int main( int argc, char* argv[] )
{
    ... 
    ClientB b1, b2; 
    ClientC c(utilityFactory,shareUtility); 
    c.h(&b1); 
    c.h(&b2); 
    ... 
}
Important points

• Fixing the memory management problems in this example program is not too hard
• However, writing complex OO software with independently developed modules without memory leaks nor multiple calls to delete is very hard!
  – Example: Epetra required major refactoring to address these problems!
• The designers of C++ never expected complex high-level code to rely on raw C++ memory management and raw calls to delete
• Almost every major C++ middleware software collection provides some higher-level support for dynamic memory management and wraps raw calls to delete
• Raw calls to delete are fragile and create memory leaks in the presents of exceptions

```cpp
void someFunction() {
    A *a = new A;
    a->f(); // memory leak on throw!
    delete a;
}
```

```cpp
void someFunction() {
    std::auto_ptr<A> a(new A);
    a->f(); // no leak on throw!
}
```

What is an alternative to using raw pointers for memory management?
• Smart Reference Counted Pointers! => Teuchos::RCP
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- RCP: Nuts & bolts and avoiding trouble
- Summary and RCP philosophy
RCP: Persisting and Non-Persisting Associations

- **Non-persisting association**: An object association that only exists within a single function call and no “memory” of the object persists after the function exits.
- **Persisting association**: An object association that exists beyond a single function call and where some “memory” of the object persists.
- **Examples**:

```cpp
class ClientA {
public:
    void f( const UtilityBase &utility ) const { utility.f(); }
};

class ClientB {
    UtilityBase *utility_
    public:
        ClientB() : utility_(0) {}
        ~ClientB() { delete utility_; }
        void initialize( UtilityBase *utility ) { utility_ = utility; }
        void g( const ClientA &a ) { a.f(*utility_); }
};
```

- **Non-persisting associations**: Use raw C++ references and pointers
- **Persisting associations**: Use RCP
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- RCP: Nuts & bolts and avoiding trouble
- Summary and RCP philosophy
Refactoring Example Program to use RCP : Part #1

Before Refactoring

class UtilityBaseFactory {
public:
    virtual UtilityBase* createUtility() const = 0;
};
class UtilityAFactory : public UtilityBaseFactory {
public:
    UtilityBase* createUtility() const { return new UtilityA(); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
    UtilityBase* createUtility() const { return new UtilityB(); }
};

After Refactoring

class UtilityBaseFactory {
public:
    virtual RCP.UtilityBase createUtility() const = 0;
};
class UtilityAFactory : public UtilityBaseFactory {
public:
    RCP.UtilityBase createUtility() const { return rcp(new UtilityA()); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
    RCP.UtilityBase createUtility() const { return rcp(new UtilityB()); }
};
Refactoring Example Program to use RCP: Part #2

Before Refactoring

class ClientA {
public:
    void f( const UtilityBase &utility ) const { utility.f(); } }

After Refactoring (no change)

class ClientA {
public:
    void f( const UtilityBase &utility ) const { utility.f(); } }

Before Refactoring

class ClientB {
    UtilityBase *utility_;
public:
    ClientB() : utility_(0) {}
    ~ClientB() { delete utility_; }
    void initialize( UtilityBase *utility ) { utility_ = utility; }
    void g( const ClientA &a ) { a.f(*utility_); }
};

After Refactoring

class ClientB {
    RCP<UtilityBase> utility_;
public:
    void initialize( const RCP<UtilityBase> &utility) { utility_=utility; }
    void g( const ClientA &a ) { a.f(*utility_); }
};

Constructor and Destructor are Gone!
Refactoring Example Program to use RCP : Part #3

Before Refactoring

class ClientC {
    const UtilityBaseFactory *utilityFactory_;
    UtilityBase *utility_;
    bool shareUtility_;

public:
    ClientC( const UtilityBaseFactory *utilityFactory, bool shareUtility )
        : utilityFactory_(utilityFactory),
          utility_(utilityFactory->createUtility()),
          shareUtility_(shareUtility) {};
    ~ClientC() { delete utilityFactory_; delete utility_; }
    void h( ClientB *b ) {
        if( shareUtility_ ) b->initialize(utility_);
        else b->initialize(utilityFactory_->createUtility());
    }
};

After Refactoring

class ClientC {
    RCP<const UtilityBaseFactory> utilityFactory_;
    RCP<UtilityBase> utility_;    
    bool shareUtility_;  

public:
    ClientC(const RCP<const UtilityBaseFactory> &utilityFactory, ... )
        : utilityFactory_(utilityFactory),
          utility_(utilityFactory->createUtility()),
          shareUtility_(shareUtility) {}
    void h( ClientB *b ) [...]  
};

Destructor is Gone!
Before Refactoring:

```c
int main( int argc, char* argv[] )
{
    // Read options from the commandline
    bool useA, shareUtility;
    example_get_args(argc, argv, &useA,
                     &shareUtility);
    // Create factory
    UtilityBaseFactory *utilityFactory = 0;
    if(useA)
        utilityFactory = new UtilityAFactory();
    else
        utilityFactory = new UtilityBFactory();
    // Create clients
    ClientA a;
    ClientB b1, b2;
    ClientC c(utilityFactory, shareUtility);
    // Do some stuff
    c.h(&b1);
    c.h(&b2);
    b1.g(a);
    b2.g(a);
    // Cleanup memory
    delete utilityFactory;
}
```

After Refactoring:

```c
int main( int argc, char* argv[] )
{
    // Read options from the commandline
    bool useA, shareUtility;
    example_get_args(argc, argv, &useA,
                     &shareUtility);
    // Create factory
    RCP<UtilityBaseFactory> utilityFactory;
    if(useA)
        utilityFactory = rcp(new UtilityAFactory());
    else
        utilityFactory = rcp(new UtilityBFactory());
    // Create clients
    ClientA a;
    ClientB b1, b2;
    ClientC c(utilityFactory, shareUtility);
    // Do some stuff
    c.h(&b1);
    c.h(&b2);
    b1.g(a);
    b2.g(a);
}
```

- New program runs without any memory problems
- New program will be easier to maintain
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- **RCP: Nuts & bolts and avoiding trouble**
  - More background
  - Construction, reinitialization, and object access
  - Explicit casting
  - Implicit casting
  - Common casting problems
- Summary and RCP philosophy
Outline

• Background on memory management in C++

• Simple motivating example using raw pointers

• Persisting and non-persisting associations

• Refactoring of simple example to use RCP

• RCP: Nuts & bolts and avoiding trouble
  – More background
  – Construction, reinitialization, and object access
  – Explicit casting
  – Implicit casting
  – Common casting problems

• Summary and RCP philosophy
Value Semantics vs. Reference Semantics

A. Value Semantics

class S {
public:
    S();               // Default constructor
    S(const S&);      // Copy constructor
    S& operator=(const S&); // Assignment operator
    ...
};

- Used for small, concrete datatypes
- Identity determined by the value in the object, not by its object address (e.g. obj==1.0)
- Storable in standard containers (e.g. std::vector<S>)
- Examples: int, bool, float, double, char, std::complex, extended precision ...

Important points!

- RCP class has value semantics
- RCP usually wraps classes with reference semantics

B. Reference Semantics

class A {
public:
    // Pure virtual functions
    virtual void f() = 0;
    ...
};

- Abstract C++ classes (i.e. has pure virtual functions) or for large objects
- Identity determined by the object’s address (e.g. &obj1 == &obj2)
- Can not be default constructed, copied or assigned (not storable in standard containers)
- Examples: std::ostream, any abstract base class, ...

Raw Pointers and RCP: const and non-const

Example:
```cpp
A a;
A* a_ptr = &a;
```

Important Point: A pointer object `a_ptr` of type `A*` is an object just like any other object with value semantics and can be const or non-const.

### Raw C++ Pointers

<table>
<thead>
<tr>
<th>Non-Const Pointer to Non-Const Object</th>
<th>Const Pointer to Non-Const Object</th>
<th>Non-Const Pointer to Const Object</th>
<th>Const Pointer to Const Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>typedef A* ptr_A;</code></td>
<td><code>typedef const A* ptr_const_A;</code></td>
<td><code>typedef A* ptr_A;</code></td>
<td><code>typedef const A* ptr_const_A;</code></td>
</tr>
<tr>
<td></td>
<td>equivalent to <code>RCP&lt;A&gt;</code></td>
<td>equivalent to <code>RCP&lt;A&gt;</code></td>
<td>equivalent to <code>const RCP&lt;A&gt;</code></td>
</tr>
<tr>
<td><code>an address</code></td>
<td><code>A's data</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><code>A* a_ptr;</code></td>
<td><code>a_ptr;</code></td>
<td>equivalent to <code>RCP&lt;A&gt;</code></td>
<td>equivalent to <code>const RCP&lt;A&gt;</code></td>
</tr>
<tr>
<td><code>A* const a_ptr;</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><code>const ptr_A</code></td>
<td><code>a_ptr;</code></td>
<td>equivalent to <code>const RCP&lt;A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td><code>A* const ptr_A</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><code>ptr_const_A</code></td>
<td><code>a_ptr;</code></td>
<td>equivalent to <code>RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td><code>const A*</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><code>const ptr_const_A</code></td>
<td><code>a_ptr;</code></td>
<td>equivalent to <code>const RCP&lt;const A&gt;</code></td>
<td><code>a_ptr;</code></td>
</tr>
<tr>
<td><code>const A* const</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
<td><code>a_ptr;</code></td>
</tr>
</tbody>
</table>

Remember this equivalence!
// Abstract hierarchy
class A {
public:
    virtual ~A(){}
    virtual void f(){} ...
};
class B1 : virtual public A {...};
class B2 : virtual public A {...};
class C : virtual public B1, virtual public B2 {...};

// Non-abstract hierarchy (no virtual functions)
class D {...};
class E : public D {...};

- Assume all these classes have reference semantics
Outline

• Background on memory management in C++

• Simple motivating example using raw pointers

• Persisting and non-persisting associations

• Refactoring of simple example to use RCP

• **RCP: Nuts & bolts and avoiding trouble**
  – More background
  – Construction, reinitialization, and object access
  – Explicit casting
  – Implicit casting
  – Common casting problems

• Summary and RCP philosophy
Constructing RCP Objects

C++ Declarations for constructing an RCP object

```cpp
template<class T>
class RCP {
public:
    RCP( ENull null_arg = null );
    explicit RCP( T* p, bool owns_mem = true );
    ...;
template<class T> RCP<T> rcp( T* p );
template<class T> RCP<T> rcp( T* p, bool owns_mem );
```

- **Initializing an RCP<> object to NULL**
  ```cpp
  RCP<C> c_ptr;  // Good for class data members!
  RCP<C> c_ptr = null;  // May help clarity
  ```

- **Creating an RCP object using new**
  ```cpp
  RCP<C> c_ptr(new C);  // or c_ptr = rcp(new C);
  ```

- **Initializing an RCP object to an object not allocated with new**
  ```cpp
  C c;
  RCP<C> c_ptr = rcp(&c,false);
  ```

- **Example**
  ```cpp
  void foo( const UtilityBase &utility )  // Non-persisting with utility
  {
    ClientB b;
    b.initialize(rcp(&utility,false));  // b lives entirely in foo()
    ...
  }
  ```
Commandment 1: Thou shall put a pointer for an object allocated with operator `new` into an RCP object only once. E.g.

```cpp
RCP<C> c_ptr(new C);
```

**Anti-Commandment 1:** Thou shall never give a raw C++ pointer returned from operator `new` to more than one RCP object.

**Example:**

```cpp
A *ra_ptr = new C;
RCP<A> a_ptr1(ra_ptr); // Okay
RCP<A> a_ptr2(ra_ptr); // no, No, NO !!!!
```

- `a_ptr2` knows nothing of `a_ptr1` and both will call delete!

**Anti-Commandment 2:** Thou shall never give a raw C++ pointer to an array of objects returned from operator `new[]` to an RCP object using `rcp(new C[n])`.

**Example:**

```cpp
RCP<std::vector<C> >
    c_array_ptr1(new std::vector<C>(N)); // Okay
RCP<C>
    c_array_ptr2(new C[n]); // no, No, NO!
```

- `c_array_ptr2` will call delete instead of `delete []`!
Reinitialization of RCP Objects

- The proper way to reinitialize an object with value semantics is to use the assignment operator! (boost::shared_ptr violates this principle!)

C++ Declarations for reinitializing an RCP Object

```cpp
template<class T>
class RCP {
public:
    RCP<T>& operator=(const RCP<T>& r_ptr);
    ...
};
```

- Resetting from a raw pointer
  ```cpp```
  RCP<A> a_ptr;
  a_ptr = rcp(new A());
  ```cpp```

- Resetting to null
  ```cpp```
  RCP<A> a_ptr(new A());
  a_ptr = null; // The A object will be deleted here
  ```cpp```

- Assigning from an RCP object
  ```cpp```
  RCP<A> a_ptr1;
  RCP<A> a_ptr2(new A());
  a_ptr1 = a_ptr2; // Now a_ptr1 and a_ptr2 point to the same A object
  ```cpp```
Access Underlying Reference-Counted Object

C++ Declarations for accessing referenced-counted object

```cpp
template<class T>
class RCP {
public:
    T* operator->() const; // Allows ptr->f(); [throws exception if NULL]
    T& operator*() const;   // Allows (*ptr).f() [throws exception if NULL]
    T* get() const;
    ...
};
template<class T> bool is_null(const RCP<T>& p);
template<class T> bool operator==(const RCP<T>& p, ENull);
template<class T> bool operator!=(const RCP<T>& p, ENull);
```

- **Access to object reference (debug runtime checked)**
  ```cpp
  C &c_ref = *c_ptr; // Throws exception if c_ptr.get()==NULL
  ```

- **Access to object pointer (unchecked, may return NULL)**
  ```cpp
  C *c_rptr = c_ptr.get(); // Never throws an exception
  ```

- **Access to object pointer (debug runtime checked, will not return NULL)**
  ```cpp
  C *c_rptr = &*c_ptr; // Throws exception if c_ptr.get()==NULL
  ```

- **Access of object’s member (debug runtime checked)**
  ```cpp
  c_ptr->f(); // Throws exception if c_ptr.get()==NULL
  ```

- **Testing for null**
  ```cpp
  if ( is_null(a_ptr) ) std::cout << “a_ptr is null
  ```

```cpp
if ( a_ptr==NULL ) std::cout << “a_ptr is null
```
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- RCP: Nuts & bolts and avoiding trouble
  - More background
  - Construction, reinitialization, and object access
  - Explicit casting
  - Implicit casting
  - Common casting problems
- Summary and RCP philosophy
Explicit Casting of RCP Objects: Template Functions

C++ Declarations for explicit casting of RCP Objects

```cpp
// implicit cast
template<class T2, class T1>
RCP<T2>& rcp_implicit_cast(const RCP<T1>& p1);

// static cast
template<class T2, class T1>
RCP<T2> rcp_static_cast(const RCP<T1>& p1);

// const cast
template<class T2, class T1>
RCP<T2> rcp_const_cast(const RCP<T1>& p1);

// dynamic cast
template<class T2, class T1>
RCP<T2> rcp_dynamic_cast(const RCP<T1>& p1,
                          bool throw_on_fail = false);
```
Explicit Casting of RCP Objects: Examples

Raw C++ Pointers

\[
\begin{align*}
D^* & \quad d = \text{new \ E;} \\
E^* & \quad e = \text{static\_cast\_E}^*(d);
\end{align*}
\]

RCP

\[
\begin{align*}
\text{RCP}\langle D \rangle & \quad d = \text{rcp(new \ E)}; \\
\text{RCP}\langle E \rangle & \quad e = \text{rcp\_static\_cast\_E}(d);
\end{align*}
\]

Constant cast

\[
\begin{align*}
\text{const A}^* & \quad \text{ca = new \ C} \\
A^* & \quad a = \text{const\_cast\_A}^*(ca);
\end{align*}
\]

Dynamic cast (runtime check, can return NULL on fail)

\[
\begin{align*}
A^* & \quad \text{a = new \ C;} \\
B1^* & \quad b1 = \text{dynamic\_cast\_B1}^*(a);
\end{align*}
\]

Dynamic cast (runtime check, can not return NULL on fail)

\[
\begin{align*}
A^* & \quad \text{a = new \ B1;} \\
B2^* & \quad b2 = (\text{a ? \&\&}\text{dynamic\_cast\_B2}^*\!(\text{a}) : \text{(B2\_*)NULL});
\end{align*}
\]

Note: In last dynamic cast, \text{rcp\_dynamic\_cast\_B2\_a(true)} throws exception with much better error message than \text{dynamic\_cast\_B2\_a}. See Teuchos::dyn\_cast<>().
Commandment 4: Thou shall only cast between RCP objects using the default copy constructor (for implicit conversions) and the nonmember template functions `rcp_implicit_cast<>(...), rcp_static_cast<>(...), rcp_const_cast<>(...) and rcp_dynamic_cast<>(...)`.  

Anti-Commandment 5: Thou shall never convert between RCP objects using raw pointer access.

Example:

```
RCP<A> a_ptr = rcp(new C);
RCP<B1> b1_ptr1 = rcp_dynamic_cast<B1>(a_ptr); // Yes : -)
RCP<B1> b1_ptr2 = rcp(dynamic_cast<B1*>(a_ptr.get())); // no, No, NO !!!!
```

- `b1_ptr2` knows nothing of `a_ptr` and both will call delete!
Outline

• Background on memory management in C++

• Simple motivating example using raw pointers

• Persisting and non-persisting associations

• Refactoring of simple example to use RCP

• **RCP: Nuts & bolts and avoiding trouble**
  – More background
  – Construction, reinitialization, and object access
  – Explicit casting
  – Implicit casting
  – Common casting problems

• Summary and RCP philosophy
// Typedefs
typedef A* ptr_A;
typedef const A* ptr_const_A;

// Takes pointer to A (by value)
void foo1( ptr_A a_ptr );

// Takes pointer to const A (by value)
void foo2( ptr_const_A a_ptr );

// Takes pointer to A (by const reference)
void foo3( const ptr_A &a_ptr );

// Takes pointer to A (by non-const reference)
void foo4( ptr_A &a_ptr );

void boo1()
{
    C* c = new C;
    A* a = c;
    foo1(c); // Okay, implicit cast to base class
    foo2(a); // Okay, implicit cast to const
    foo2(c); // Okay, implicit cast to base class and const
    foo3(c); // Okay, implicit cast to base class
    foo4(a); // Okay, no cast
    foo4(c); // Error, can not cast from (C&) to (A&*);
}
Implicit Casting of RCP Objects

C++ Declarations for implicit casting

```cpp
template<class T>
class RCP {
public:
    template<class T2>
    RCP(const RCP<T2>& r_ptr); // Temlated copy constructor!

    // ...
};
template<class T2, class T1>
RCP<T2> rcp_implicit_cast(const RCP<T1>& p1);
```

Raw C++ Pointers

**Implicit cast to base class**

```cpp
C* c_rptr = new C;
A* a_rptr = c_rptr;
RCP<C> c_ptr = rcp(new C);
RCP<A> a_ptr = c_ptr;
```

**Implicit cast to const**

```cpp
A* a_rptr = new A;
const A* ca_rptr = a_rptr;
RCP<A> a_ptr = rcp(new A);
RCP<const A> ca_ptr = a_ptr;
```

**Implicit cast to base class and const**

```cpp
C* c_rptr = new C;
const A* ca_rptr = c_rptr;
RCP<C> c_ptr = rcp(new C);
RCP<const A> ca_ptr = c_ptr;
```

Note: Templated copy constructor allows implicit conversion of RCP objects in almost every situation where an implicit conversion with raw C++ pointers would be allowed
Implicit Casting: Raw C++ Pointers versus RCP

Raw C++ Pointers

typedef A* ptr_A;
typedef const A* ptr_const_A;

void foo5(ptr_A a_ptr);
void foo6(ptr_const_A a_ptr);
void boo2()
{
    C* c = new C;
    A* a = c;
    foo5(c); // Okay, cast to base
    foo6(a); // Okay, cast to const
    foo6(c); // Okay, to base+const
}

RCP

void foo5(const RCP<A> &a_ptr);
void foo6(const RCP<const A> &a_ptr);
void boo3()
{
    RCP<C> c = rcp(new C);
    RCP<A> a = c;
    foo5(c); // Okay, cast to base
    foo6(a); // Okay, cast to const
    foo6(c); // Okay, to base+const
}

• Implicit conversions for RCP objects to satisfy function calls works almost identically to implicit conversions for raw C++ pointers and raw C++ references except for a few unusual cases:
  – Implicit conversions to call overloading functions (see example on next page)
  – ???
Outline

- Background on memory management in C++
- Simple motivating example using raw pointers
- Persisting and non-persisting associations
- Refactoring of simple example to use RCP
- **RCP: Nuts & bolts and avoiding trouble**
  - More background
  - Construction, reinitialization, and object access
  - Explicit casting
  - Implicit casting
  - Common casting problems
- Summary and RCP philosophy
Implicit Casting with RCP: Common Problems/Mistakes

Passing RCP by non-const reference instead of by const reference

```cpp
void foo7(RCP<A> &a);
void foo7(const RCP<A> &a);

void boo4()
{
    RCP<C> c(new C);
    RCP<A> a = c;
    foo7(a);  // Okay, no cast
    foo7(c);  // Error, can not cast involving non-const reference
    foo7(c);  // Okay, implicit case involving const reference okay
}
```

Failure to perform implicit conversion with overloaded functions

```cpp
RCP<A> foo9(const RCP<A> &a);
RCP<const A> foo9(const RCP<const A> &a);

RCP<A> boo5()
{
    RCP<C> c(new C);
    return foo9(c);  // Error, call is ambiguous!
    RCP<A> a = c;
    return foo9(a);  // Okay, calls first foo9(...)
    return foo9(rcp_implicit_cast<A>(c));  // Okay, calls first foo9(...)
}
```

A deficiency of smart pointers over raw pointers

Calls `foo9(A* a)` when using raw C++ pointers!
Outline

• Background on memory management in C++

• Simple motivating example using raw pointers

• Persisting and non-persisting associations

• Refactoring of simple example to use RCP

• RCP: Nuts & bolts and avoiding trouble

• Summary and RCP philosophy
Summary of RCP and Advanced Features Overview

- RCP combines concepts of “smart pointers” and “reference counting” to build an imperfect but effective “garbage collection” mechanism in C++

- **Smart pointers** mimic raw C++ pointer usage and syntax
  - Value semantics: i.e. default construct, copy construct, assignment etc.
  - Object dereference: i.e. \((*\text{ptr}).f()\)
  - Pointer member access: i.e. \(\text{ptr}->f()\)
  - Conversions:
    - Implicit conversions using templated copy constructor: i.e. \(\text{C}^*\) to \(\text{A}^*\), and \(\text{A}^*\) to \(\text{const A}^*\)
    - Explicit conversions: i.e. \(\text{rcp\_const\_cast<T>}(p)\), \(\text{rcp\_static\_cast<T>}(p)\), \(\text{rcp\_dynamic\_cast<T>}(p)\)

- **Reference counting**
  - Automatically deletes wrapped object when last reference (i.e. smart pointer) is deleted
  - **Watch out for circular references!** These create memory leaks!
    - Tip: Call \(\text{Teuchos\_setTracingActiveRCPNodes(true)}\) (with --enable-teuchos-debug)

- **RCP\(<T>** is not a raw C++ pointer!
  - Implicit conversions from \(\text{T}^*\) to \(\text{RCP}\(<T>\) and visa versa are not supported!
  - Failure of implicit casting and overload function resolution!
  - Other problems ...

- **Advanced Features (not covered here)**
  - Template deallocation policy object
    - Allows other an \texttt{delete} to be called to clean up
    - Allows one smart pointer (e.g., \texttt{boost::shared\_ptr}) to be embedded in an RCP
  - Extra data
    - Allows RCP to wrap classes that do not have good memory management (e.g. old Epetra)
    - Allows arbitrary events to be registered to occur before or after the wrapped object is deleted
The Philosophy of RCP

- Using RCP for only persisting associations increases the vocabulary of the C++ language and makes in more self documenting.
  
  ```cpp
  void foo( const A &a, const RCP<C> &c );
  ```

- Responsibilities of code that generates shared objects (e.g. factories)
  - Create and initialize the concrete object to be given away
  - Define the deallocation policy that will be used to deallocate the object
  ```cpp
  RCP<BaseClass>
  SomeFactory::create() const {
    ConcreteSubclass *instance; InitializeObject(&instance); // Initialize!
    new rcp(instance, DeallocConcreteSubclass(), true); // Define destruc. policy!
  }
  ```

- Responsibilities of code that maintains persisting associations with shared objects
  - Accept RCP objects that wrap shared objects
  - Maintain RCP objects to shared objects while in use
  - Destroy or assign to null RCP objects when it is finished using shared object
  ```cpp
  class SomeClient {
    RCP<A> a_; // Maintain A
  public:
    void accept(const RCP<A> &a) { a_ = a; } // Accept A
    void clearOut() { a_ = null; } // Forget/release A
  };
  ```

- RCP allows radically different types of software to be glued together to build a correct and robust memory management system
  - Use RCP to wrap objects not currently wrapped with RCP and use customized deallocation policies
  - Combine different smart pointer objects together (e.g., boost::shared_ptr, ...)
Ben Parker once said to Peter Parker:

"With great power comes great responsibility"

and the same can be said of the use of RCP and of C++ in general