SAND REPORT

SANDxxx-xxx
Unlimited Release
Printed ???

Teuchos::RefCountPtr

The Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Optimization/Uncertainty Estim

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SANDxxx-xxx
Unlimited Release
Printed ???

Teuchos::RefCountPtr

The Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Optimization/Uncertainty Estim
Sandia National Laboratories,* Albuquerque NM 87185 USA,

Abstract

This document builds on the material in the document "Teuchos::RefCountPtr Beginner's Guide: An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++". In this document we discuss the templated concrete C++ class `Teuchos::RefCountPtr<>` in more detail and discuss the use of extra data, deallocation policies and other topics. We expand on the proper use of this new data type and how it relates to built-in C++ data types and other user-defined data types. A variety of other issues related to `Teuchos::RefCountPtr<>` are also discussed.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Acknowledgment

The author would like to thank Carl Laird, Heidi Thornquist, Mike Heroux and Mazrio Sala for comments on earlier drafts of this document.

The format of this report is based on information found in [7].

Contents

Appendix

Figures

Teuchos::RefCountPtr

The Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

1 Introduction to `RefCountPtr<>`

Described herein is the templated class `RefCountPtr<>` (which exists in the namespace `Teuchos`) and a set of global template functions for implementing automatic garbage collection in C++ using smart reference-counted pointers. This document builds on the material in the beginner's guide [1]. The beginner's guide is considered an integral part of this document and this document is incomplete without it.

This discussion assumes that the reader is knowledgeable of C++ and has read books such as [11] and [8]. While understanding and appreciating the advanced features of `RefCountPtr<>` requires advanced C++ knowledge and experience, the basic use (as described in the beginner's guide [1]) requires only basic C++ skills. By "coding by example" even beginner C++ programmers can immediately realize the benefits of `RefCountPtr<>`.

The commandments stated in Appendix B in the beginner's guide [1] are represented here in the context of a wider discussion.

The following class hierarchy is used to demonstrate this smart pointer design.

```
class A { public: virtual ~A(){} A& operator=(const A&){} virtual void f(){} };
class B1 : virtual public A {};
class B2 : virtual public A {};
class C : virtual public B1, virtual public B2 {};
class D {};
class E : public D {};
```

Note that the classes `A`, `B1`, `B2` and `C` are polymorphic (with multiple inheritance using virtual base classes) while the classes `D` and `E` are not. In the following description, all of the code examples are written as though there was a `using namespace Teuchos;` declaration is the current scope or as if all of the code resided in the `Teuchos` namespace.

The public C++ declarations for all of the code discussed here are shown in Appendix A in the beginner's guide [1]. A short quick-start and reference for `RefCountPtr<>` is contained in Appendix B in the beginner's guide [1].

A word of caution; note that the design of the reference-counted smart pointer class discussed in Item 29 of [9] in the subsection “Adding Reference Counting to Existing Classes” has a purpose very similar to `RefCountPtr<>` but the design of `RefCountPtr<>` is very different (see Appendix B). Since many of the issues involved in using smart pointers and reference counting discussed Meyers [9] are the same as with `RefCountPtr<>` this reference is worthy of study. However, the exact designs used in this reference are not used in `RefCountPtr<>` since they lead to less flexible software.

1.1 Initializing `RefCountPtr` objects

A smart reference-counted pointer to a dynamically allocated object (of type `A` for instance) is initialized as follows

```
RefCountPtr<A>             a_ptr   = rcp(new A); // A       *       a_ptr   = new A;
RefCountPtr<const A>       ca_ptr  = rcp(new A); // const A *       ca_ptr  = new A;
const RefCountPtr<A>       a_cptr  = rcp(new A); // A       * const a_cptr  = new A;
const RefCountPtr<const A> ca_cptr = rcp(new A); // const A * const ca_cptr = new A;
```

The above code shows how all the various combinations for `const` or non-`const` pointers to `const` or non-`const` objects are expressed (this is explained in more detail in Item 28 in [9]). There is no automatic conversion from raw pointers to smart pointers for many good reasons (again, see Item 28 in [9]). To allow such an implicit conversion almost guarantees memory allocation/deallocation errors will occur. Therefore, the following templated function

```
template<class T> RefCountPtr<T> rcp(T* p, bool owns_mem = true );
```

must be used to initialize a smart pointer object given a raw pointer. The reason for this is somewhat complicated but it is related to multiple inheritance and/or virtual base classes (see Appendix C).

Commandment 1 *Thou shall put a pointer for an object allocated with operator* `new` *into a* `RefCountPtr<>` *object only once. The best way to insure this is to call operator* `new` *directly in the call to* `rcp(...)` *to create a dynamically allocated object that is to be managed by a* `RefCountPtr<>` *object.*

The only case where the template function `rcp<>(...)` should not be used to create a smart pointer is the initialization to `NULL`. One way to initialize a null smart pointer is to just use the default constructor as

```
RefCountPtr<A> a_ptr;   // A a_ptr = NULL:
```

One of the big advantages of using `RefCountPtr<>` over raw pointers (independent of the reference counting and garbage collection features) is this default initialization to `NULL`. To be more explicit about the initialization to `NULL` (and for passing a null pointer into a function), a special constructor is included which takes the enumeration value `null`. For example

```
RefCountPtr<A> a_ptr = null; // i.e. A a_ptr = NULL:
```

Commandment 2 *Thou shall only create a `NULL` `RefCountPtr<>` object by using the default constructor or by using the `null` enum (and its associated special constructor). Trying to assign to `NULL` or `0` will not compile.*

Another situation that commonly occurs is when an object is allocated on the stack or statically, but must be passed to a client that only accepts a `RefCountPtr<>` object. For example, suppose that there is a class `Foo` that is declared as follows

```
class Foo {
public:
    Foo( const RefCountPtr<A>& a ) : a_(a) {}
    void f() { a_->f(); }
private:
    RefCountPtr<A>  a_;
};
```

The following code fragment shows how to allocate an object of concrete type `C` on the stack and then pass this object into the constructor for `Foo`

```
C c;
Foo foo(rcp(&c,false));
```

In this example, the templated function `rcp(...)` is used with the parameter `owns_mem` set to `false`. In this example, when the `foo` object goes out of scope (and its `Foo::a_` member is deleted), `delete` will not be called on the underlying pointer and no memory error will occur. For more specialized types of memory deallocation control see Section 2.3.

Commandment 3 *Thou shall only pass a raw pointer for an object that is <u>not</u> allocated by `new` (e.g. allocated on the stack) into a `RefCountPtr<>` object by using the templated function `rcp<T>(T* p, bool owns_mem)` and setting `owns_mem` to `false`.*

1.2 Accessing the underlying reference-counted object

Through the magic of the overloaded member operator functions `RefCountPtr<T>::operator->()` and `RefCountPtr<T>::operator*()`, the underlying object being reference counted can be accessed exactly the same as for raw pointers such as follows

```
a_ptr->f();
(*a_ptr).f();
```

Therefore, using a smart reference-counted pointer is very similar to using a raw C++ pointer. The use of smart pointers is very similar to raw pointers as shown below.

```
RefCountPtr<A>
  a_ptr1     = rcp(new A),   // Initialize them from a raw pointer from new
  a_ptr2     = rcp(new A);   // ""
A *ra_ptr1   = new A,        // ""
  *ra_ptr2   = new A;        // ""
a_ptr1       = rcp(ra_ptr2); // Assign from a raw pointer (only do this once!)
a_ptr2       = a_ptr1;       // Assign one smart pointer to another
a_ptr1       = rcp(ra_ptr1); // Assign from a raw pointer (only do this once!)
a_ptr1->f();                 // Access a member using ->
ra_ptr1->f();                // ""
*a_ptr1      = *a_ptr2;      // Dereference the objects and assign
*ra_ptr1     = *ra_ptr2;     // ""
```

What makes smart pointers different however is that the above piece of code does not create any memory leaks that would have otherwise occurred if `a_ptr1` and `a_ptr2` where raw C++ pointers.

However, these smart reference-counted pointers can not be used everywhere a raw pointer can. `RefCountPtr<>` purposefully does not support all raw pointer syntax. The goal of `RefCountPtr<>` is not to hide the fact that smart pointers are being used over raw pointers. Therefore, developers should not expect (or even want) to simply change the data type of raw pointers from `T*` to `RefCountPtr<T>` and then have a large body of code simply recompile without a wimper from the compiler. For instance, the following statements will not compile

```
a_ptr1++;           // Error, pointer arithmetic ++, --, +, - etc. not defined!
a_ptr1 == ra_ptr1;  // Error, comparision operators ==, !=, <=, >= etc. not defined!
```

Since a smart reference-counted pointer can not be used everywhere a raw C++ pointer can, there is a means for getting at the raw C++ pointer with the `RefCountPtr<T>::get()` function (same as with `std::auto_ptr<T>::get()`). For example, to check if two `RefCountPtr<>` objects contain pointers to the same underlying object one would check:

```
if( a_ptr1.get() == a_ptr2.get() )
    std::cout << "a_ptr1 and a_ptr2 point to the same object\n";
```

1.3 Conversion of `RefCountPtr<>` objects

The existing C++ conversion rules for raw C++ pointers ((e.g. from derived to base classes, from non-`const` to `const`)) are supported by `RefCountPtr<>` for the conversion of smart pointers. For

example, the compiler will implicitly cast a raw C++ pointer up its type's inheritance hierarchy (e.g. `B1` $\rightarrow$ `A` or `E` $\rightarrow$ `D`). For smart reference-counted pointers, these implicit conversions are just as easy thanks to a templated copy constructor (see Figure **??** and the discussion in Item 28 in []). For example, the below code compiles and runs just fine

```
RefCountPtr<C>  c_ptr  = rcp(new C);
RefCountPtr<A>  a_ptr  = c_ptr;
RefCountPtr<B1> b1_ptr = c_ptr;
RefCountPtr<D>  d_ptr  = rcp(new E);
```

To perform other non-implicit type conversions with pointers such as `static_cast<>`, `const_cast<>` and `dynamic_cast<>`, the namespace `Teuchos` contains the nonmember template functions `rcp_static_cast<>(...)`, `rcp_const_cast<>(...)` and `rcp_dynamic_cast<>(...)` respectively with the following prototypes:

```
template<class T2,
class T1> RefCountPtr<T2> rcp_static_cast(const RefCountPtr<T1>& p1);

template<class T2, class T1>
RefCountPtr<T2> rcp_const_cast(const RefCountPtr<T1>& p1);

template<class T2, class T1>
RefCountPtr<T2> rcp_dynamic_cast(const RefCountPtr<T1>& p1);
```

The usage of these conversion template functions looks very similar to the syntax used for raw C++ pointers. For example

```
RefCountPtr<const C>   c_ptr   = rcp(new C);
RefCountPtr<const A>   ca_ptr  = c_ptr;                              // Implicit!
RefCountPtr<const C>   cc_ptr1 = rcp_dynamic_cast<const C>(ca_ptr);  // Safe!
RefCountPtr<const C>   cc_ptr2 = rcp_static_cast<const C>(ca_ptr);   // Unsafe!
RefCountPtr<A>         a_ptr   = rcp_const_cast<A>(ca_ptr);          // Cast away const
```

Just like the built-in C++ conversion operators, some types of conversions will not compile. For example

```
RefCountPtr<C>       c_ptr1 = rcp(new C);
RefCountPtr<A>       a_ptr  = c_ptr;
RefCountPtr<const C> c_ptr2 = rcp_dynamic_cast<const C>(a_ptr);                      // Error!
RefCountPtr<const C> c_ptr3 = rcp_dynamic_cast<const C>(const_cast<const A>(a_ptr)); // Okay!
RefCountPtr<D>  d_ptr  = rcp(new E);
RefCountPtr<E>  e_ptr1 = rcp_dynamic_cast<E>(d_ptr); // Error, D and E are not polymorphic!
RefCountPtr<E>  e_ptr2 = rcp_static_cast<E>(d_ptr);  // Okay, but unchecked!
```

This brings us to another commandment.

Commandment 4 *Thou shalt only cast between* `RefCountPtr<>` *objects using the default copy constructor (for implicit conversions) and the nonmember template functions* `rcp_static_cast<>( ...)`, `rcp_const_cast<>(...)` *and* `rcp_dynamic_cast<>(...)` *described above.*

A developer should never never never try to convert between `RefCountPtr<>` objects using the `get()` member function (which violates Commandment 4). For example, the following piece of code

```
{
  ...
  RefCountPtr<A> a_ptr = rcp(new A());
  ...
  RefCountPtr<C> c_ptr = rcp(a_ptr.get());
  ...
}
```

will always (unless other steps are taken) result in a memory deallocation error when these smart pointer objects are deleted (i.e. the pointer being deleted twice).

2 Advanced features of `RefCountPtr<>`

In this section we describe some advanced features of `RefCountPtr<>` that most developers will never need to use and therefore may never need to know about. However, there are certain challenging use cases where this functionality is critical to the success of `RefCountPtr<>` and therefore are discussed here. It is the inclusion of these advanced features that makes `RefCountPtr<>` an almost universal "band aide" for the use of poorly designed (from a memory management standpoint) software.

2.1 Accessing reference-count information

In most situations, clients do not need to know anything about reference counts but there are use cases where it is helpful if clients can query reference-count information. For this purpose, `RefCountPtr<>` contains the member functions

```
template<class T> int  RefCountPtr<T>::count() const {...}
template<class T> bool RefCountPtr<T>::has_ownership() const {...}
```

which can be used to ascertain the status of a reference-counted object. Specifically, if `ptr` is a `RefCountPtr<>` object and `(ptr.count()==1 && ptr.has_ownership())==true` then the

client that owns `ptr` (where `ptr` is perhaps as a private data member) can infer that it is the only client that has a reference to the underlying reference-counted object. Therefore, such a client is free to change the object without worry that such a change will affect another client.

This type of information must be used very carefully but can be very helpful in simplifying the logic in certain types of use cases. If this type of logic is to be used then it must be documented very clearly since it is easy to falsely show a reference count higher than the number of clients that really own `RefCountPtr<>` objects to the reference-counted object (i.e. due to temporary object creation that allow implicit conversions).

2.2 Associating extra data with a `RefCountPtr<>` object

There are situations when a single `RefCountPtr<>` object is insufficient to completely handle the dynamic memory for a reference-counted object. This can occur, for instance, when trying to use classes that were written without dynamic memory allocation in mind. To demonstrate the issues involved, assume that the following classes are to be reused in a dynamic environment where the memory management is to be handled using `RefCountPtr<>` objects.

```
class Base {
public:
  virtual ~Base() {}
  ...
};

class Derived1 : public Base {
  ...
};

class Utility {
  ...
};

class Derived2 : public Base {
public:
  Derived2(const Utility& u) : u_(u) {}
  ...
private:
  const Utility& u_;
};
```

Above, every `Derived2` object must have an associated `Utility` object. The class `Derived2` assumes that an object of type `Utility` is allocated (presumably on the stack or as a global variable) by the client before it is used to construct a type `Derived2` object. For example, the designers of these classes may have expected that class objects would be used in a flat program such as:

```
void f(Base* b);
```

```
int main() {
  ...
  if(...) {
    Derived1 d1;
    f(d1);
  }
  else {
    ...
    Utility u;
    Derived2(u) d2;
    f(d2);
  }
  ...
  return 0;
}
```

In the above types of programs the flow of logic is linear from beginning to end and it is easy to allocate all of the objects on the stack.

Now suppose that we wish to create a factory function

```
RefCountPtr<Base> createBase(bool someFlag);
```

(see the “Abstract Factory” design pattern in [5]) that dynamically allocates a `Base` object (of concrete type `Derived1` or `Derived2`) given a runtime flag and return it using a `RefCountPtr<Base>` object. Now let us consider the following naive and incorrect first implementation of this function.

```
RefCountPtr<Base> createBase(bool someFlag)
{
  if(someFlag) {
    return rcp(new Derived1);
  }
  else {
    Utility u;
    return rcp(new Derived2(u));
  }
}
```

After some thought, the problem with the above function should be obvious. As soon as the block of code where `u` is allocated (on the stack) is exited, the created `u` object (and therefore also the `Derived2(u)` object) becomes invalid before the function even returns. A second naive and incorrect implementation of this function might be

```
RefCountPtr<Base> createBase(bool someFlag)
{
  if(someFlag) {
    return rcp(new Derived1);
```

```
  }
  else {
    Utility *u = new Utility;
    return rcp(new Derived2(*u));
  }
}
```

The above implementation will insure that the `Utility` object created in this function will remain valid and therefore the `Derived2` object returned from the function will remain valid but a memory leak will have been created. If the function `createBase(...)` is only called a few times in a program then this memory leak will most likely be harmless but if this function is called too many times, memory exhaustion will occur (resulting in a program crash in most cases).

The solution to this type of problem is to attach a `RefCountPtr<Utility>` object to the created `RefCountPtr<Derived2>` object so that when all of the `RefCountPtr<Derived2>` objects pointed to this object are deleted (along with its underlying `Derived2` object) the associated `Utility` object will also be deleted. The way to do this is to use the nonmember function

```
template<class T1, class T2>
void set_extra_data(
  const T1 &extra_data, const std::string& name
  ,RefCountPtr<T2> *p, bool force_unique = true
  ,EPrePostDestruction destroy_when = POST_DESTROY
  );
```

The function `set_extra_data(...)` adds a piece of extra data of any type[1] to an existing `RefCountPtr<T2>` object. A string `name` is associated with each piece of extra data to disambiguate it from other pieces of extra data with the same type and to facilitate later retrieval. The boolean argument `force_unique` (which has a default value of `true`) is used to allow or disallow overwriting existing data of the same type and name. The enumeration argument `destroy_when` can take on the values `POST_DESTROY` and `PRE_DESTROY` (and takes the default value of `POST_DESTROY`). For the class of use cases described here the default value of `destroy_when=POST_DESTROY` is the correct value. However, in other use cases, it is useful to allow pieces of extra data to be destroyed before a reference-counted object is destroyed. At its essences, the extra data facility allows a developer to allow some action or set of actions to be performed just before or just after an object is destroyed. This is a very powerful feature that is not supported in most memory management classes.

A piece of extra data is keyed both on the type of the extra data and its string name and access is granted by using one of the following nonmember functions.

```
template<class T1, class T2>
T1& get_extra_data( RefCountPtr<T2>& p, const std::string& name );

template<class T1, class T2>
const T1& get_extra_data( const RefCountPtr<T2>& p, const std::string& name );
```

[1] Note: the type `T1` used for extra data must support value semantics (i.e. have default and copy constructors defined)

See the Teuchos Doxygen documentation for more details on these functions.

In many cases, a client does not care to retrieve a set piece of extra data and therefore the string `name` really only needs to disambiguate the extra data being added from other pieces of extra data that may potentially already be added (perhaps by another unknown client). This allows multiple, potentially unrelated, clients to tack on extra data to a `RefCountPtr<>` object and retrieve that extra data without stomping on each other. It is important to remember that in order for a client to retrieve a piece of extra data that was added to a `RefCountPtr<>` object, the client must know the exact type (i.e. not a base type) of the extra data and its string name that where used with the call to the `set_extra_data()` function that added this data.

New let us see how to use the function `set_extra_data(...)` to correctly implement `createBase(...)`

```
RefCountPtr<Base> createBase(bool someFlag)
{
  if(someFlag) {
    return rcp(new Derived1);
  }
  else {
    RefCountPtr<Utility> u = rcp(new Utility);
    RefCountPtr<Base> d = rcp(new Derived2(*u));
    set_extra_data( u, "createBase::u", &d );
    return d;
  }
}
```

Now when the last `RefCountPtr<>` object to the dynamically allocated `Derived2` object is removed and the `Derived2` object is deleted, the associated dynamically allocated `Utility` object will also be deleted. Note that extra data associated with a `RefCountPtr<>` object by default is deleted after the reference-counted object is deleted. This is the correct behavior in use cases like described above where the destructor for `Derived2` may need to call functions on its aggregate `Utility` object and therefore the utility object must be valid at this point. Added pieces of extra data are destroyed in a first-in last-out manner (just as is done on the stack).

Note that the ability to add extra data to a `RefCountPtr<>` object is an advanced feature that that most programmers may never need to use. However, when backed into a corner, this functionality can be critical to a successful implementation.

The reason that `get_extra_data()` template functions are nonmember functions is that the C++ standard does not allow the explicit specification of a template argument in a member template function without the use of a verbose, nonintuitive syntax [11, C.13.6].

Even though `set_extra_data()` can deduce the template type from the input argument, there may be cases where a client may want to be specific about what type is being used to store extra data such as in the following case

```
set_extra_data<float>( 1e+5, "some-number", &a_ptr );
```

In the above example, if the type of the extra data `float` had not been explicitly specified, the compiler would have used `double` as dictated by the standard. However, the same result could have been achieved with

```
set_extra_data( static_cast<float>(1e+5), "some-number", &a_ptr );
```

which is more verbose. It is up to the developer to decide what form is more desirable.

2.3 Customized deallocation through templated `Dealloc` policy objects

The most common use of `RefCountPtr<>` is to manage the lifetime of objects allocated using operator `new` and deallocated using operator `delete`. For these use cases, the built-in behavior in `RefCountPtr<>` does exactly the right thing. However, there are situations when one can not simply call `delete` to deallocate an object. Some examples of these situations include:

1. When reference counts for objects are managed by requiring clients to explicitly call increment and decrement functions. This situation occurs when using CORBA [6] and COM [3] for instance. Such an approach is also presented in [9, Item 29] in the subsection “A Reference-Counting Base Class”. In these protocols, deallocation occurs automatically behind the scenes when this other reference count goes to zero and does not occur through an explicit call to operator `delete` as with the default behavior for `RefCountPtr<>`.
2. When objects are managed by certain types of object databases. In some object databases, an object that is grabbed from the database must be explicitly returned to the database in order to allow proper object deletion to take place later.
3. A different reference-counted pointer class is used to initially get access to the managed object. For example, suppose some piece of peer software works with `boost::shared_ptr<>` (see [2]) referenced-counted objects while the resident software works with `RefCountPtr<>` objects. It then becomes critical no object is deleted until all the clients using either of these smart pointer types remove their references (i.e. destroy their smart pointers) to this underlying object.

There are also other situations where one can not simply assume that calling operator `delete` is used to release an object. The bottom line is that in order to be general, one must allow arbitrary policies to be used to deallocate an object after clients are finished using the object.

`RefCountPtr<>` supports arbitrary deallocation polices using the concept of a template deallocator object through the following version of the nonmember function `rcp(...)`.

```
template<class T, class Dealloc_T>
RefCountPtr<T> rcp( T* p, Dealloc_T dealloc, bool owns_mem );
```

The implementation of the simpler version of `rcp(...)` first described in Section 1.1 uses the default deallocator class

```
template<class T>
class DeallocDelete
{
public:
  typedef T ptr_t;
  void free( T* ptr ) { if(ptr) delete ptr; }
};
```

All deallocator objects must support the typedef member `ptr_t` and function member `free()`. The concept of a template policy interface (also called a function object [11, Section 18.4]) should be familiar to semi-advanced users of the STL (part of the standard C++ library).

To demonstrate the use of a deallocator object, let us assume that we must wrap objects of type `A` managed by the the following object database

```
class ObjectADB {
  ...
  A& get(int id);
  void release(int id);
  ...
};
```

In the above object database, objects are accessed and released using an id. How this id is specified and determined is not important here. Now let us suppose that we want to define an abstract factory that returns objects of type `A` wrapped in `RefCountPtr<A>` objects using a database of type `ObjectADB` shown above. For this abstract factory, objects of type `A` will be allocated from a list of ids given to the factory. The outline of this abstract factory subclass looks like the following

```
class ObjectADBFactory : public AbstractFactory<A> {
public:
  ObjectADBFactory( ObjectADB *db, int num_ids, const int ids[] ) : db_(db), ids_(ids,ids+num_ids) {}
  RefCountPtr<A> create() { ... };  // Overridden from AbstractFactory
private:
  ObjectADB *db_;
  std::vector ids_;
};
```

The above abstract factory subclass `ObjectADBFactory` inherits from a generic `AbstractFactory` base class that defines a pure virtual method `create()`. In order to implement the override of the `create()` method, a deallocator class must be defined and used. For this purpose we define

```
template<class A>
class DeallocObjectADB
{
public:
  DeallocObjectADB( ObjectjADB* db, int id ) : db_(db), id_(id) {}
  typedef A ptr_t;
  void free( A* ptr ) { db_->release(id_); }
private:
  ObjectjADB* db_;
  int id_;
  DeallocObjectADB(); // not defined and not to be called!
};
```

Above, the default constructor is declared private and has no implementation to avoid accidental default construction (see Item 27 in [8]).

Now we can define the implementation of the `create()` function override below.

```
RefCountPtr<A> ObjectADBFactory::create()
{
  TEST_FOR_EXCEPTION( ids_.size()==0, std::runtime_error, "No ids are left!" );
  const int id = ids_.pop();
  return rcp(&db_->get(id),DeallocObjectADB(db_,id),true);
}
```

The following program shows the use of the above factory class

```
int main()
{
  // Create the object database and populate it (and save the ids)
  ObjectADB db;
  std::vector ids;
  ...
  // Create the abstract factory object
  ObjectADBFactory  ftcy(db,ids.size(),&ids[0])
  // Create some A objects and use them
  RefCountPtr<A> a_ptr1 = fcty.create();
  ...
  return 0;
}
```

In the above program, all of the objects of type `A` are created and removed seamlessly without the client code that interacts with `RefCountPtr<>` and `AbstractFactory<>` knowing anything about what is going on under the hood.

3 Guidelines for the usage of `RefCountPtr<>` and other C++ data types

After the basics of `RefCountPtr<>` are understood its use is quite straightforward. Furthermore,this class can be instantiated with any built-in or user-defined concrete or abstract data type, so one may ask why `RefCountPtr<>` should not be used for all objects. There is nothing stopping a developer from using `RefCountPtr<>` in this way but to do so would be overkill and counterproductive. The most compelling situation where `RefCountPtr<>` should be used is when a client needs to maintain a private data member to another object which is of an abstract type or is an object that is shared with other clients. As described in the beginner's guide [1] this is an example of a *persisting* relationship and `RefCountPtr<>` should be used for all *persisting* relationships. Note that `RefCountPtr<>` usually should not be used for built-in data types such as `int` or `double` unless there is a desire to allow one client to change the value and then have this value automatically updated in other clients (which is rarely a good idea to allow in general).

Note that the consistent use of `RefCountPtr<>` makes it possible to skip writing explicit destructors for most classes since the compiler-generated destructor does exactly the right thing.

4 Idioms for passing objects to and from functions

This section focuses on how to best write function prototypes using `RefCountPtr<>`. Understanding how to use `RefCountPtr<>` well also requires understanding how to use more standard concrete and abstract data types through raw C++ references and pointers.

Choosing how to write function prototypes for passing arguments to and from functions in languages like C and Fortran 77 (exception F77 does not have a concept of function prototypes) is simple since there are few choices to make. In C you can either pass an object by value or by reference (i.e. through a pointer). In Fortran 77 the choice is even easier (there is no choice) since every variable is passed by reference. Passing arrays of objects to and from functions in these languages is slightly more complicated, but not by much.

On the other hand, C++ adds the concepts of `const` and references (i.e. `type& obj`) for passing arguments in addition to C's concepts of pointers (i.e. `type* obj`) and pass-by-value (i.e. `type obj`). This makes deciding how to pass arguments in C++ much more complicated than in other languages. These features (which where added to C++ for very good technical reasons), when guided by commonsense idioms, can make it more-or-less clear how arguments should be passed to and from functions in C++ based on their purpose and can serve to clearly documument the purpose in many cases without having to write much extra documentation. This discussion presents some of these idioms and how `RefCountPtr<>` fits in.

In many respects C++ is a highly expressive programming language where if used carefully, in

many cases, the programmer's intent for an interface or a function prototype can be discerned just by looking at the C++ declaration. For example, if a reference or pointer argument passed into a function has the `const` qualifier, then a developer can rightly assume that the argument will not be changed within the function. The language tries to help enforce `const` but, in the end, it is up to the implementer of the function to honor the promise inherent in the `const` modifier in the function prototype (for a more detailed discussion of `const` in C++ see [11, Section 10.2.7.1]). In other cases, the exact intent of an argument passed to a function can not be discerned just be looking at the function prototype. For example, if an array argument is specified as `int a[]` it is not clear if this array is required (i.e. `a!=NULL` required) or is optional (i.e. `a==NULL` allowed). C++ is just not expressive enough to completely document every use case for an argument.

In addition to the idioms mentioned above, there are other C++ idioms that can be used so that the general nature of arguments passed to and from a function can be largely discerned just by looking at the C++ prototype for the function. The `const` qualifier just mentioned above is one of the language features which is self-documenting but there are others. Another example is passing arrays to and from functions, for example, by declaring an integer array argument as `int arg[]` instead of `int* arg`. When declaring an argument as `int arg[]` there is no question that an array is being passed but in the case of `int* arg` it is not clear if the address of the first element of an array is being passed or just the address to a single object. Appendix D in the beginner's guide [1] shows recommendations for how to pass objects to and from C++ functions in a way that is as self-documenting as possible.

The reasons why the declarations are the way they are in Appendix D in [1] can be found in the references [11] and [8]. Specifically, see the sections ???. The discussion below describes each of the use cases shown in Appendix D in [1] and how `RefCountPtr<>` fits in.

As a basic rule of thumb, `RefCountPtr<>` objects should be passed around as though they where raw C++ pointers to the underlying objects. The main difference between passing `RefCountPtr<>` and raw C++ pointers, however, is that it is usually a good idea to pass `RefCountPtr<>` objects by `const` reference instead of by value since passing `RefCountPtr<>` objects by value results in unnecessary calls to copy constructors and destructors that, if nothing else, make stepping through code in debugger more difficult.

4.1 Detailed discussion/justification for idioms

ToDo: Put in a lot more example code.

If the reader already accepts and is confortable with the idioms for function prototypes shown in [1, Appendix D] and overviewed in the previous section, then this section can be skipped until later. However, readers that do not agree or are curious why the idioms in [1, Appendix D] are emphasised in this document should read the following section and them make up their own minds. This material is not fundumental to the design of `RefCountPtr<>` but instead is more basic advise about good C++ practicies along with experienced guidelines for the use of `RefCountPtr<>`.

We now explain the reasoning behind the idiom recommendations given in Appendix D in [1] for how to pass objects to and from C++ functions and how `RefCountPtr<>` fits in with these idioms. First it should be stated that many different code projects do not follow these idioms. Usually this is because the developers where unaware of these idioms and why they are useful to follow. Even if a developer is familiar with these idioms for raw data types, the may not be as familiar with how `RefCountPtr<>` fits in.

As stated in the beginner's guide [1], it is recommended to use raw object references for passing non-persisting arguments (i.e. those that are only accessed during a function call) to a function. Here, the term reference is used in a more abstract sense where a reference can be implemented as a C++ reference such as `const A &a` or as a C++ pointer such as `A *a`. On the other hand, it is recommended to pass `RefCountPtr<>` wrapped object references (i.e. `const RefCountPtr<A> &a`) for any persisting arguments (i.e. those where a "memory" of the object in the server implementing the function persists after the function returns).

Passing `RefCountPtr<>` wrapped object references to and from functions will differ slightly from how raw object references are passed. As a rule of thumb, when thinking about how to use `RefCountPtr<>` to pass object references to and from functions, one should think of a `RefCountPtr<>` object as though it was really just a raw pointer. The main difference is that while raw pointers to single objects are usually passed by value (i.e. `A *a`) to a function (which is very efficient) a `RefCountPtr<>` wrapped single object on the other hand should usually be passed by `const` reference (i.e. `const RefCountPtr<A> &a`). It is important to pass single `RefCountPtr<>` object by `const` reference and not by non-`const` reference (i.e. `RefCountPtr<A> &a`) since the compiler will only perform automatic conversions for `const` references. However, passing `RefCountPtr<>` objects by value is harmless but passing `RefCountPtr<>` objects by `const` reference eliminates unnecessary (while quite trivial) constructor and destructor function calls. In any case, passing `RefCountPtr<>` objects by `const` reference as apposed to by value simplifies stepping through code in a debugger and should be preferred if only for this reason. On the other hand, when passing arrays of objects wrapped within `RefCountPtr<>` objects, it is recommended to pass an array of `RefCountPtr<>` objects (i.e. `const RefCountPtr<A> a[]`) and not an array of `RefCountPtr<>` references. This is discussed below.

Before getting into great detail about these recommendations and exactly how `RefCountPtr<>` fits in, consider the following example function prototype.

```
void foo1(
  const int             size_x
  ,const double         x[]
  ,const int            size_a
  ,const A*             a[]
  ,const int            size_y
  ,double               y[]
  ,double               *z
  );
```

By comparing the argument declarations (which use good argument names) in the above function

to the recommendations in [1, Appendix D], and without looking at any extra documentation (if it even exists), we already know a lot about the arguments being passed into and out of this function. First, it is clear that `x` is an array of non-mutable `double` objects with an presumed length of `size_x`. Similarly, we know that `y` is an array of mutable `double` objects that may be set or modified. We also know that `z` is a single mutable `double` object that may be changed inside of the function. What the above C++ declarations, however, do not tell us is whether the arguments `x`, `a`, `y` and `z` are optional (i.e. a `NULL` pointer can be passed in for them) or are required (i.e. a non-`NULL` pointer can be passed in for them). We also do not know if the mutable objects passed in `y` and `z` are only output arguments (i.e. the state of the object before the function call is not significant) or are input/output arguments (i.e. the state of the object before the function call is significant). This type of information must be specified in developer-supplied documentation. Note that while a required single non-mutable object such as `z` could be passed using a non-`const` reference, using a non-`const` reference for such an argument is usually not advised and this issue is addressed a little later.

Note that the above function `foo(...)` involved all non-persisting (i.e. only accessed with the function) arguments which is often the case. Now lets consider the use of persisting argument types as supported using `RefCountPtr<>`.

```
class SomeClass {
public:
  ...
  void foo2(
    const int                                  size_x
    ,double                                    x[]
    ,const RefCountPtr<std::vector<double> >   &y
    )
  {
    y_ = y;
    ...
  }
  ..
private:
  RefCountPtr<std::vector<double> >  y_;
};
```

The above example function `SomeClass::foo2(...)` is an example where a persisting argument, `y`, is passed into a function and is then "remembered". In this case, the `RefCountPtr<>` argument `y` is delivering a `std::vector<double>` object to be used as private data for the class object.

4.1.1 Passing non-mutable objects

Non-mutable non-persisting objects (i.e. objects that are passed into a function, are not modified and no memory of these objects is maintained) can be passed by value or by reference (i.e. through a C++ pointer or a C++ reference) and as either single objects or as an array of objects. When non-mutable objects are passed to a function using a C++ pointer or a C++ reference, the

declaration should always use the `const` modifier. Failure to use the `const` modifier in these cases goes against the design of the C++ language and is incompatible with the standard C++ library. The use of the `const` modifier both serves as documentation and helps the implementor of the function avoid accidentally changing the object. However, when a small concrete object of type `S` (where `S` is a `double` or `int` for instance) is being passed, it may also be passed by value (with or without the `const` modifier as shown in [1, Appendix D]). When a concrete object is passed by value, the `const` modifier does not in any way affect the client code that calls the function and the client's copy of the argument is guaranteed not to be changed regardless of whether the `const` modifier is used or not. On the other hand, the `const` modifier on a C++ pointer or C++ reference does not really guarantee that the object will not be changed. The `const` modifier used with pass-by-value only restricts the modification of the copied argument in the function implementation. Some see the use of `const` pass-by-value as a means to help function implementors from making mistakes while others see this as a unnecessary restriction on the freedom in the implementation of the function.

When a non-mutable input argument is required, it should be passed by value (i.e. `S s`) or by `const` C++ reference (i.e. `const S &s`). However, when a non-mutable input argument is optional, it must be passed using a `const` C++ pointer (i.e. `const S *s`) since this pointer is be allowed to be `NULL` for when the argument is not specified.

Arrays of non-mutable objects should either be passed as an array of objects (i.e. in the case where the object's type is a small concrete data type such as `double`) such as `S s[]` or `const S s[]`, or as an array of `const` C++ pointers (i.e. in the case where the type is a large object or the type has reference semantics) such as `const S* s[]`. Note that the syntax `const S s[]` (or `const A* a[]`) for passing an array of objects is to be preferred over the C++ pointer syntax `const S *s` when passing an array of objects or `const A** a` when passing an array of pointers. This is because that even though the raw C++ pointer syntax `const S *s` is perfectly legal and proper C++ usage, this form does not allow one to tell if this pointer is supposed to point to a single object or to an array of objects and therefore the syntax `const S s[]` is to be preferred. Also note that the syntax `const A* a[]` allows individual components of `a[i]` to be optional (i.e. `a[i]==NULL` allowed) or required (i.e. `a[i]!=NULL` required) in addition to the entire array being optional (i.e. `a==NULL` allowed) or required (i.e. `a!=NULL` required). In this case the preconditions for these types of array of pointer arguments must be spelled out in supplemental documentation.

Note that while it is possible to pass objects as an array of object references such as `const S& s[]`, it is generally very difficult to initialize the references right when the array is first created (as is required by the standard). Therefore it is not recommended to passed objects as an array of references but instead to use an array of pointers as shown in [1, Appendix D]

Also note that unlike in the case of passing single non-mutable objects, that there is no a convenient way to help differentiate whether an array of objects being passed to a function is optional or not. When passing arrays of objects, it must be documented whether the arrays are required or optional.

Non-mutable persisting objects (i.e. objects that are passed into a function, are not modified but

some memory of these objects is maintained after the function call) should be passed through `RefCountPtr<>` wrapped objects. Single object references of this type should be passed as `const RefCountPtr<const A> &a`. Note the presences of the `const` templated type `<const A>` which states that clients may not modify the underlying wrapped object (see [1, Appendix B] and Section 1.1 for an explanation of how to express combinations of `const` or non-`const` pointers to `const` or non-`const` objects using `RefCountPtr<>`). As stated earlier, the `RefCountPtr<>` object itself should be passed as a `const` reference to avoid unnecessary constructor and destructor calls. Note that unlike passing single raw `const` object reference to a function, there is no easy way at compile time to differentiate between a required and an optional argument when passing a `RefCountPtr<>` wrapped object. For the argument `const RefCountPtr<const A> &a`, if the underlying object is required, then the precondition `a.get()!=NULL` should be documented. Likewise, if the object is optional, then it should be documented that `a.get()==NULL` is allowed.

Passing an array of non-mutable persisting objects wrapped in `RefCountPtr<>` objects is a little different from arrays for raw C++ objects or object references. While it is recommended to pass single object references using `const RefCountPtr<>` C++ references, an array of `RefCountPtr<>` objects should be passed as an array of `RefCountPtr<>` objects as `const RefCountPtr<const A> a[]` rather than trying to pass an array of C++ references to `RefCountPtr<>` objects such as `const RefCountPtr<const A>& a[]`. As stated earlier, it is extremely difficult to work with arrays of references. Note that just as is the case when passing an array of raw object references, there is no good way to specify at compile time whether the entire array `RefCountPtr<>` objects is required or is optional and this instead must be explicitly documented. Note also that just as in the case for passing arrays of raw object references, the declaration `const RefCountPtr<const A> a[]` allows individual components of `a[i]` to be optional (i.e. `a[i].get()==NULL` allowed) or required (i.e. `a[i].get()!=NULL` required) in addition to the entire array being optional (i.e. `a==NULL` allowed) or required (i.e. `a!=NULL` required). The specific preconditions on such array arguments must be documented.

4.1.2 Passing mutable objects

Many of the issues involved with passing mutable objects (i.e. client maintained objects that are modified within a function) are the same as for passing non-mutable objects such as described above but there are some differences. First off, objects that are to have their state changed within a function must be passed by reference (i.e. C++ pointer or C++ reference) and can not be passed by value. In the case of mutable arguments, non-`const` instead of `const` C++ pointers are used to pass raw object references. Likewise, `RefCountPtr<A>` objects are used instead of `RefCountPtr<const A>` objects. Other than that, all of the issues are exactly the same as in the case of passing non-mutable objects.

One apparent area of controversy is the issue of whether non-`const` reference or non-`const` pointer arguments should be used for passing mutable objects to a function. If the argument is optional, then the choice is clear and the argument should be declared as a non-`const` pointer.

However, if the argument is required then some would say that a non-`const` reference should always be used. Interested readers should note, however, that references were added to C++ primarily to support operator overloading (see [10, Section 3.7]) and also for return values. If one is writing an overloaded operator function that modifies one of its arguments (such as the stream insertion and extraction operators `std::istream& operator<<(std::istream &i, ... )` and `std::ostream& operator>>(std::ostream &o, ... )` for instance) then a non-`const` reference argument must be used (and note the non-`const` reference return type as well). When the function is not an overloaded operator function then the choice is not so clear. Strustroup (the original inventor of C++ [10]) makes a case for using non-`const` pointers for mutable arguments in most non-operator functions in [11, Section 5.5]. In short, the use of non-`const` C++ references over non-`const` C++ pointer arguments for required mutable objects gives a false sense of security and in the end does not really guarantee better quality software. While that standard states that every C++ reference must be bound to a valid C++ object this is not guaranteed in any way. For example, consider the following program that will segfault when run.

```
#include <iostream>

void f( int& i )
{
  i = 5;
  std::cout << "\ni = "<<i<<std::endl;
}

void g( int* i )
{
  f(*i);
}

int main()
{
  int a  = 2, *b = NULL;
  g(&a);
  g(b);
  return 0;
}
```

The above program can be put into a file (`bad_ref.cpp` for instance) and be compiled and linked with no errors by any standard compliant C++ compiler (for instance using g++ as `g++ -o bad_ref.exe bad_ref.cpp`). However, when this program is run (as `./bad_ref.exe` for instance) the output will look something like:

```
#./bad_ref.exe

i = 5
Segmentation fault (core dumped)
```

The reason for the segmentation fault should be quite obvious. In the case of the above flat program, a very smart compiler might be able to produce a warning but in the general case, where the main program and the functions `f()` and `g()` are compiled in separate source files, there is no way a C++ compiler can possibly catch this error at compile time. This example just proves that the language does not guarantee that C++ references are bound to valid objects.

Since non-`const` C++ references don't really give any more guarantees than non-`const` C++ pointers, the chose of whether to use non-`const` C++ references over non-`const` C++ pointers should be made on other grounds. The primary justification for using non-`const` C++ pointer arguments for mutable objects is to force a different specification for non-mutable and mutable objects being passed into a C++ function that both give visual clues of what objects are going to be changed and provides some compile-time checking if arguments change from being non-mutable to mutable. For example, consider the following program

```
#include <iostream>

void f( const std::vector<int> &i, const std::vector<int> &j, int *k );

int main()
{
  std::vector<int>  i(5,0), j(5,1);
  int k;
  ...
  f( i, j, &k );
  ...
  std::foreach( j.begin(), j.end(), ... );
  ...
  return 0;
}
```

In the above program, even if you never even saw the prototype for the function `f()`, you can assume that the argument `k` is being modified by the function because of the visual clue `&k` in the statement

```
  f( i, j, &k );
```

while rightly assuming that the arguments `i` and `j` will not be modified. Later on in this program, the statement

```
  std::foreach( j.begin(), j.end(), ... );
```

uses the array `j`, which is assumed to still be in its initialized state. Now suppose that some overly clever developer decides that he or she can implement the function `f()` better if the array in `j` is used as workspace in addition to providing data for the function. This is a common practice

(especially in well-written Fortran programs). If non-`const` references are the norm for mutable arguments, then this developer may decide to change the prototype of `f()` to

```
void f( const std::vector<int> &i, std::vector<int> &j, int *k );
```

which now allows the function to use `j` as workspace. Removing a `const` modifier from a function like this may cause some calls to this function to not compile (which is a very good thing) but calls like in the above example program will still compile just fine. If this change is made to the function `f()` and the argument `j` is silently changed without any clue to the code in `main()`, then the behavior of the rest of the code that follows the call to `f()` may be silently changed (such as was the case in this function). This could be a very hard bug to track down.

On the other hand, if non-`const` C++ pointer arguments are always used for mutable objects, then the prototype for `f()` must be changed to

```
void f( const std::vector<int> &i, std::vector<int> *j, int *k );
```

if the argument `j` is to be used a workspace. In this case, the function call

```
  f( i, j, &k );
```

would not compile and the developer that maintains this code would have a great opportunity to change the calling code to cope with the fact that `j` is no longer a non-mutable argument. In this case, either a copy of `j` could be created to pass into `f()` or the logic after the call to `f()` could no longer assume that `j` is unchanged.

In summary, all things being equal, tracking down runtime errors associated with using raw non-`const` C++ pointer arguments is easier than tracking down runtime errors that result from using non-`const` C++ reference arguments and using non-`const` C++ pointer arguments give a visual clue as to what arguments are being changed and what arguments are being left unmodified. This is a subjective opinion but is one that is held by many esteemed C++ experts (including Strustroup himself).

However, while C++ references where not purposefully added to the language to support non-`const` pass-by-reference in non-operator functions, there are circumstances where using non-`const` reference arguments is to be preferred. An example might be one where the fact that an argument is going to be changed as a result of a function call is obvious from the name of the function or of the nature of the object being passed. This is the case, for instance, in the standard C++ function

```
template<class T1, class T2> void swap( T1 &t1, T2 &t2 );
```

where it is obvious that when called like:

```
void foo3()
{
  int a = 5, b = 6;
  std::swap(a,b);
}
```

that the objects `a` and `b` will be modified.

Another example of when it is reasonable to use a non-`const` reference is when working with objects such as standard streams (i.e. `std::ostream` and `std::istream`) where the `const` interface is almost worthless. For example, it is perfectly reasonable to declare a print function such as

```
class Foo2 {
public:
  ...
  void print(std::ostream &out) const;
  ...
};
```

where when called as

```
void foo4(const Foo2 &foo2) {
  foo2.print(std::cout);
}
```

it is obvious that the stream object `std::cout` will be modifed in the function call.

The general approach advocated here is to prefer non-`const` C++ pointer arguments in non-operator functions unless there is a compelling reason to use a non-`const` C++ reference instead such as when there is unlikely to be any confusion whether the argument is going to be changed and a couple of examples where given above. Making the determination of when to use a non-`const` C++ reference over a non-`const` C++ pointer requires experience, taste and in the end is a subjective decision.

While it was stated earlier that a `RefCountPtr<>` object should almost always be passed as a `const` reference (i.e. `const RefCountPtr<>&`), there is one use case where a pointer to a `const RefCountPtr<>` object (i.e. `RefCountPtr<>*`) should be passed instead. This occurs when the calling client code is to acquire a `RefCountPtr<>` to an object held by the server code, or in other words form persisting relation between the returned object and the client. When only a single `RefCountPtr<>` is to be acquired for a single object, then it usually best to just return a `RefCountPtr<>` object from the function as its return value such as in the case

```
RefCountPtr<Base> createBase(bool someFlag);
```

described in Section 2.2. However, when more than one `RefCountPtr<>` object is to be initialized and given to a client, then one should consider passing non-`const` pointers to `RefCountPtr<>` objects to a function (see the function `uninitialize()` in Appendix A).

5 Summary

This document expanded on the introductory discussion in the beginner's guide [1]. Some of the more advanced features of `RefCountPtr<>` where described in more detail. Suggestions for the usage of `RefCountPtr<>` and C++ itself was also discussed. The Appendices discuss some other topics related to `RefCountPtr<>` in detail such as the "separate construction and initialization" idiom (Appendix ???), the design of `RefCountPtr<>` (Appendix ???) and issues related to multiple inheritance and `RefCountPtr<>` (Appendix ???).

References

[1] Roscoe A. Bartlett. Teuchos::RefCountPtr 101 : A beginner's guide to the Trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++. Technical Report SAND2004-xxx, Sandia National Laboratories, 2004.

[2] BOOST. The BOOST library. http://www.boost.org.

[3] Micosoft Corporation. COM: Component object model. http://www.microsoft.com/com.

[4] M. Fowler and K. Scott. *UML Distilled, second edition*. Addison-Wesley, 2000.

[5] E. Gamma, R. Helm, R. Johnson, and John Vlissides. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Object Management Group. CORBA: Common object request broker architecture. http://www.corba.org.

[7] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[8] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[9] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[10] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, New York, 1994.

[11] B. Stroustrup. *The C++ Programming Language, special edition*. Addison-Wesley, New York, 2000.

A The “separate construction and initialization” idiom

Here we describe an idiom which works most effectively when using `RefCountPtr<>` for implementing concrete classes (that may be manipulated using reference (i.e. pointer) semantics) that maximizes flexibility and reusability. This is the concept of “separate construction and initialization”. The idea is that if an class object can be default constructed to a potentially “uninitialized” state and then at any time later can be separately initialized then such a class will have the maximum reusability and this can not be disputed. The idiom described here is more effective for concrete classes which are derived from abstract interface classes. It will be shown below that this “separate construction and initialization” idiom also trivially supports the “initialization on construction” idiom which is preferable in many cases and is advocated by many experts (for example, see [11, Section 10.2.3] and [9, Item 4]).

What many C++ experts fail to recognize, however, is that in many cases the entities that are best qualified to fully initialize and object are not in a position, nor should they be forced, to know the concrete subclass of an objects that they initialize. To force some kinds of entities to both allocate and initialize objects is to complicate the design and limit the reusability of such entities.

The basics of this “separate construction and initialization” idiom are demonstrated in the following example subclass.

```
class ConcreteClass1 : public AbstractClass {
public:
  ...
private:
  RefCountPtr<A>  a_;
  RefCountPtr<B>  b_;
};
```

In the above concrete class, `RefCountPtr<>` objects are maintained to objects of type `A` and `B`. Note that the types `A` and/or `B` may be abstract and in the overall design, `ConcreteClass1` may not be able to know the concrete types for these objects (this scenario is the basis for object-oriented design), therefore necessitating the use of `RefCountPtr<>`. Hence, the `A` and/or `B` objects must be allocated by an external entity and then given to a `ConcreteClass1` object in order to fully initialize it.

The idiom advocated here is to specify classes like `ConcreteClass1` as

```
class ConcreteClass1 : public AbstractClass {
public:
  ConcreteClass1() {}
  ConcreteClass1( const RefCountPtr<A>& a, const RefCountPtr<B>&  b )
    { initialize(a,b); }
  void initialize( const RefCountPtr<A>& a, const RefCountPtr<B>& b )
    { a_ = a; b_ = b; }
  void uninitialize( RefCountPtr<A> *a = NULL, RefCountPtr<B> *b = NULL )
    { if(a) *a = a_; if(b) *b = b_; a_ = null; b_ = null; }
```

```
  ...
private:
    RefCountPtr<A>  a_;
    RefCountPtr<B>  b_;
};
```

Note that in the above implementation of `ConcreteClass1` that an object can be default constructed to an “uninitialized” state or can be fully initialized on object creation (using the second constructor that calls the `initialize()` function). Therefore, this idiom trivially also supports the “initilaization on construction” use case as well.

A client can later uninitialize an object of type `ConcreteClass1` and take ownership of the aggregate `A` and `B` objects by calling the `uninitialize()` function as

```
ConcreteClass1 cc1;
...
RefCountPtr<A> a;
RefCountPtr<B> b;
cc1.uninitialize(&a,&b);
```

or could uninitialize a `ConcreteClass1` object and not take ownership as

```
ConcreteClass1 cc1;
...
cc1.uninitialize();
```

In the last example, if `cc1` was the only client maintaining `RefCountPtr<>` objects to its aggregate `A` and `B` objects, then these objects would be deallocated during this function call. This idiom also allows clients to optionally take ownership of some but not all of the aggregate objects on uninitialization. For example, a client could take ownership of the `B` but allow the `A` to be deleted as

```
ConcreteClass1 cc1;
...
RefCountPtr<B> b;
cc1.uninitialize(NULL,&b);
```

In summary, classes designed to use the “separate construction and initialization” idiom advocated above undeniably makes such classes more reusable than those designed to use the “initialization on construction” idiom. In addition, the above “separate construction and initializaation” is not any more inconvenient for client software to use since the “initialization on construction” use case is also supported.

B Design of `RefCountPtr<>`

Here we describe the basics of the C++ design of `RefCountPtr<>` and discuss what goes on under the hood. It is not necessary that developers using `RefCountPtr<>` know these details in order to successfully use the class but C++ experts and the otherwise curious will want to know this information. This information will also be very useful for those who need to debug through code that uses `RefCountPtr<>`. We, however, only provide an outline of the implementation and the interested reader is encouraged to study the actual C++ implementation and its test program to fill in the gaps left in this discussion. Note, the best way to browse the C++ code is through the doxygen generated HTML files that are part of the Teuchos package in the Trilinos documentation on the web (just do a web search for Teuchos).

Figure B.1 shows a UML [4] class diagram for the basic design. The templated class `RefCountPtr<T>` is the user-level type that this document describes. There are also other classes that are used behind the scenes to manage the reference count information, manage extra data and control deallocation. Reference-count information is managed primarily though an abstract non-templated shared `RefCountPtr_node` class object. This abstract class has only one subclass `RefCountPtr_node<ConcreteT,Dealloc_T>` but this subclass is templated on the actual concrete type `ConcreteT` of the underly reference-counted object and the deallocation policy (i.e. function) object type `Dealloc_T`. The purpose of these private utility classes will be make clear in the following discussion.

Every `RefCountPtr<T>` object maintains a typed pointer to the reference-counted object through a private data member `ptr` (of type `T*`) which is shown in the UML diagram as the association between `RefCountPtr<T>` and `ConcreteT` (the actual concrete type of the object being reference counted). Here, the type `T` must be a supported interface for the concrete type `ConcreteT` (i.e. `T` is base class of `ConcreteT` or `T` are `ConcreteT` are the same) and this is shown by the lollipop interface `T` connected to the type `ConcreteT`. It is through the private data member `ptr` that the functions `get()`, `operator*()` and `operator->()` are implemented.

As mentioned above, every `RefCountPtr<T>` object also maintains a pointer to a shared non-templated `RefCountPtr_node` object which is called `node`. This is shown using the association from `RefCountPtr<T>` to `RefCountPtr_node` with the role name `node`. The `RefCountPtr_node` `node` object maintains a reference count (`count`) for the number of `RefCountPtr<>` objects pointing to itself. The reference-count modification functions `incr_count()` and `deincr_count()` on the `RefCountPtr_node` object `node` are called by `RefCountPtr`'s constructors, destructors and assignment operators in order to maintain the proper count. The `RefCountPtr_node` `node` object also maintains an ownership flag `has_ownership` that determines if the deallocator will be used to deallocate the shared object when the reference count goes to zero (this is described later). This boolean is used to primarily support the case where objects allocated on the stack or statically must be wrapped by a `RefCountPtr<>` object in order to be passed to some piece of code.

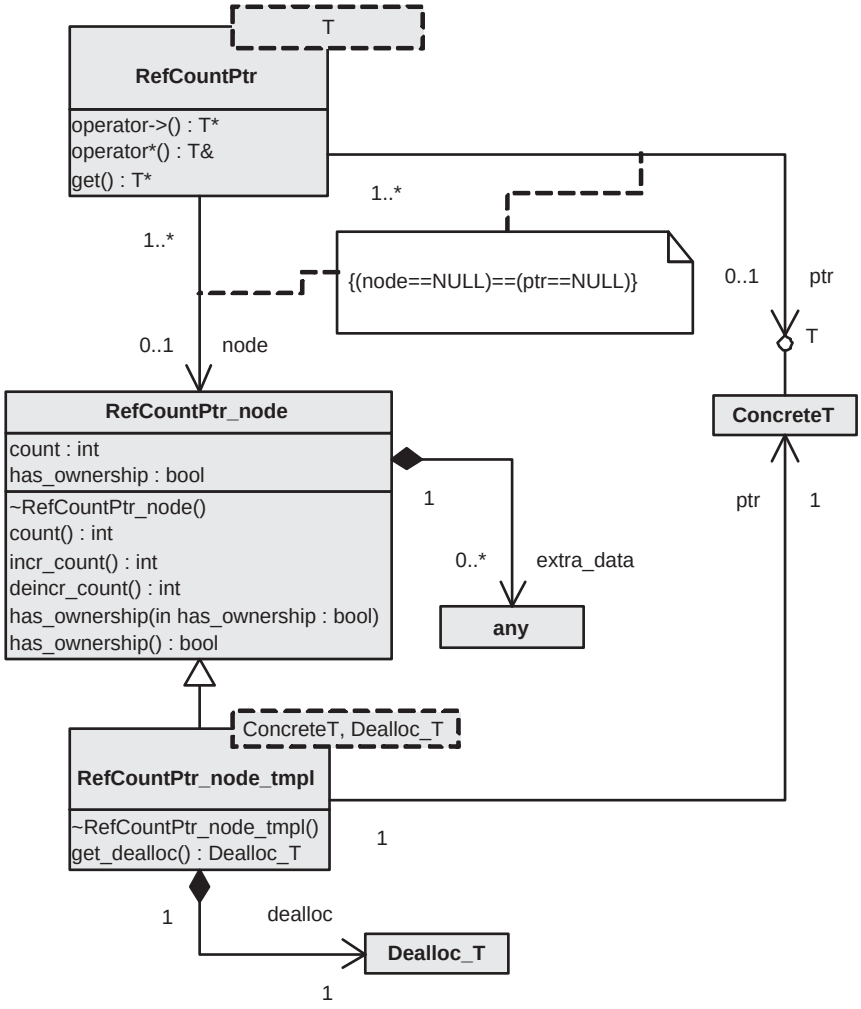

Figure B.1. UML class diagram : This shows the basic design of `RefCountPtr<>` with its reference-count node classes.

When a RefCountPtr<T> object is constructed to NULL, its ptr and node private pointer data members are set to NULL. This case is indicated in the UML class diagram by the multiplicity indicators 0..1 (where the 0 multiplicity is for the case where the object is in the NULL state). This is stated explicitly by the constraint {(node==NULL)==(ptr==NULL)} which will only be true if both node and ptr are NULL or non-NULL .

B.1 Construction, conversion and assignment of RefCountPtr<> objects

Some more of the details of the C++ design of RefCountPtr<> will be discussed in the context of the following snippet of user code that uses the classes from the class hierarchy first presented in Section 1.

```
if( ... ) {
  RefCountPtr<A>   a_ptr1 = rcp(new C);
  RefCountPtr<A>   a_ptr2 = a_ptr1;
  RefCountPtr<B1>  b1_ptr = rcp_dynamic_cast<B1>(a_ptr1);
  RefCountPtr<B2>  b2_ptr = rcp_dynamic_cast<B2>(a_ptr1);
  ...
}
```

In the above user code, a single concrete object of type C is dynamically allocated and put into a RefCountPtr<A> object. Three other RefCountPtr<> objects (of different interface types) are then created that reference this same object.

In the first statement

```
  RefCountPtr<A>   a_ptr1 = rcp(new C);
```

the first RefCountPtr<> object is created by the expression rcp(new C) which calls the following templated function

```
template<class T> RefCountPtr<T> rcp( T* p )
{
  return RefCountPtr<T>(p,true);
}
```

which calls the constructor

```
template<class T> RefCountPtr<T>::RefCountPtr( T* p, bool has_ownership )
  :ptr_(p)
  ,node_( p
     ? new PrivateUtilityPack::RefCountPtr_node_tmpl<T,DeallocDelete<T> >(p,DeallocDelete<T>(),has_ownership)
     : NULL )
{}
```

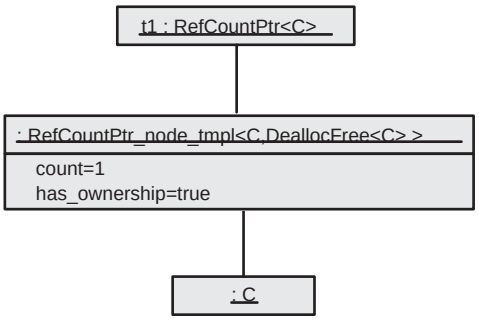

Figure B.2. Object diagram for the temporary returned from the expression `rcp(new C)`.

which is instantiated with the type `T = C`. As shown in the above constructor, the pointer `T* p` is passed into the constructor of the concrete `RefCountPtr_node_tmpl<C,DeallocDelete<C> >` node. This process results in the exact same typed address that was returned from `new C` being stored in the underlying `RefCountPtr_node_tmpl<C,DeallocDelete<C> >` node object and it is this pointer and address that will be used to delete the object later. It is this process involving the use of the templated function `rcp()` that guarantees that the same address returned from `new` is passed back to `delete` which addresses the problem described in Appendix C that occurs on some platforms. Figure B.2 shows a UML object diagram for the first `RefCountPtr<C>` object created by the statement `rcp(new C)`. This is a temporary object which is maintained by the compiler and we give it the name `t1` even through it really has no name. Note that the initial reference count on the `RefCountPtr_node_tmpl<C,DeallocDelete<C> >` is initialized by default as `count=1`.

After the first `RefCountPtr<C` object is created from `rcp(new C)`, as shown in Figure B.2, the rest of the statement

```
RefCountPtr<A>   a_ptr1 = rcp(new C);
```

is executed. This statement is executed in one of two ways based on how the compiler generates the code. The straightforward (non-optimized) approach is to default construct the `RefCountPtr<A>` object `a_ptr1` first and then call the assignment operator (see [???]). However, the assignment operator for `RefCountPtr<>` is not doubly templated so to assign a `RefCountPtr<C>` object to a `RefCountPtr<A>` object, the doubly templated copy constructor must be invoked to perform a conversion from a `RefCountPtr<C>` object to a `RefCountPtr<A>` object. The copy constructor invoked is as follows.

```
template<class T> template <class T2> RefCountPtr<T>::RefCountPtr(const RefCountPtr<T2>& r_ptr)
```

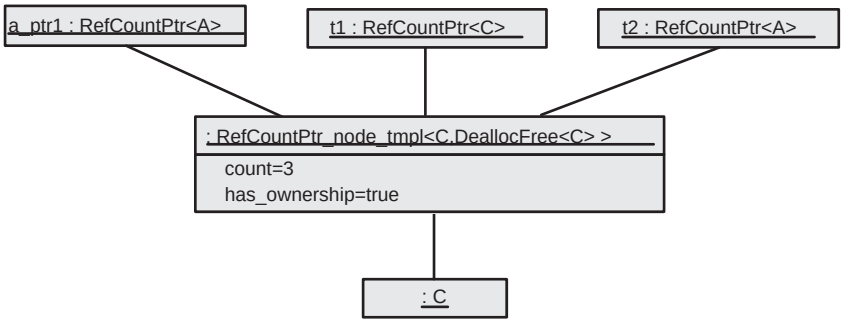

Figure B.3. Object diagram for the state just before the completion of the statement `RefCountPtr<A> a_ptr1 = rcp(new C);`.

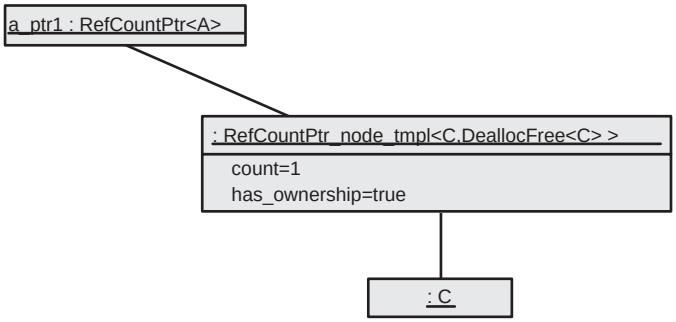

Figure B.4. Object diagram for the state just after the completion of the statement `RefCountPtr<A> a_ptr1 = rcp(new C);`.

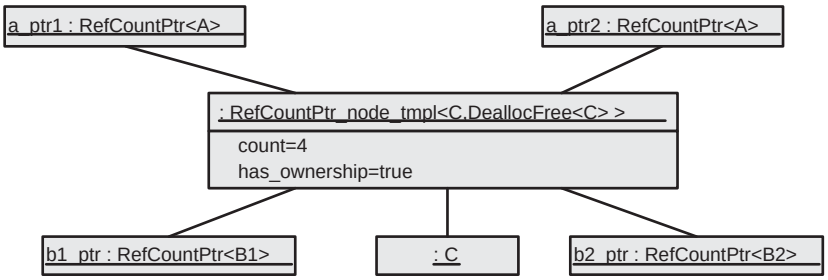

Figure B.5. Object diagram for the state after the creation of the four `RefCountPtr<>` objects `a_ptr1`, `a_ptr2`, `b1_ptr` and `b2_ptr`.

```
  : ptr_(r_ptr.get()) // will not compile if T2* is not convertible to T*
  , node_(r_ptr.access_node())
{
  if(node_) node_->incr_count();
}
```

Note how the reference count is explicitly incremented on the node.

This conversion generates another temporary that we will call `t2`. With this temporary `RefCountPtr<A> t2` created, the following assignment operator is then invoked.

```
template<class T>  RefCountPtr<T>& RefCountPtr<T>::operator=(const RefCountPtr<T>& r_ptr)
{
  if(node_) {
    if( r_ptr.node_ == node_ )
      return *this; // Assignment to self!
    if( !node_->deincr_count() )
      delete node_;
  }
  ptr_  = r_ptr.ptr_;
  node_ = r_ptr.node_;
  if(node_) node_->incr_count();
  return *this;
}
```

The above implementation of the assignment operator is a little more complex since it has to look out for assignment-to-self and has to remove its reference to an existing reference-counted object (if it has one). After the assignment operator is invoked and the statement

```
  RefCountPtr<A>   a_ptr1 = rcp(new C);
```

has completed and just before the compiler-generated temporaries are destroyed, the state of the objects are as shown in Figure B.3.

After the statement

```
RefCountPtr<A>   a_ptr1 = rcp(new C);
```

finishes executing and the compiler deletes the temporaries the object diagram looks like Figure B.4. At this point, only one `RefCountPtr<>` object is in existence and the reference count is back to one (i.e. `count=1`).

Note that through return value optimization [???] and the conversion of

```
RefCountPtr<A>   a_ptr1 = rcp(new C);
```

to

```
RefCountPtr<A>   a_ptr1(rcp(new C));
```

(which is allowed by the C++ standard [???]) that this statement will result in no temporary objects being created when optimizations are turned on at compile time.

After describing the details of the first statement above, we could also describe the remaining statements

```
RefCountPtr<A>   a_ptr2 = a_ptr1;
RefCountPtr<B1>  b1_ptr = rcp_dynamic_cast<B1>(a_ptr1);
RefCountPtr<B2>  b2_ptr = rcp_dynamic_cast<B2>(a_ptr1);
```

in same level of detail but the issues are largely the same so we do not discuss them. Interested readers are encouraged to look the implementation of the template function `rcp_dynamic_cast()` for instance to see how it works. After all of the above four `RefCountPtr<>` objects are constructed, the object diagram looks Figure B.5.

B.2 Destruction of reference-counted objects

Up to this point we have described how `RefCountPtr<>` objects are constructed and assigned and how reference counts are manipulated. Now it is equality important to describe how reference-counted objects maintained through `RefCountPtr<>` are deleted when all of the `RefCountPtr<>` objects referring to the underlying object are removed. Again, considering the snippet of user code

```
if( ... ) {
  RefCountPtr<A>   a_ptr1 = rcp(new C);
  RefCountPtr<A>   a_ptr2 = a_ptr1;
  RefCountPtr<B1>  b1_ptr = rcp_dynamic_cast<B1>(a_ptr1);
  RefCountPtr<B2>  b2_ptr = rcp_dynamic_cast<B2>(a_ptr1);
  ...
}
```

When the end of the `if(...) { ... }` block is reached, the automatic `RefCountPtr<>` objects `a_ptr1`, `a_ptr2`, `b1_ptr` and `b2_ptr` allocated on the stack by the compiler are popped off of the stack and destroyed in the opposite order that they where created. When each of these `RefCountPtr<>` objects is destroyed, the following destructor is called

```
template<class T> RefCountPtr<T>::~RefCountPtr()
{
  if(node_ && node_->deincr_count() == 0 ) delete node_;
}
```

The above destructor is quite simple. If the `RefCountPtr<>` object being destroyed is not `NULL` (i.e. `node_ != 0`) then the reference count is deincremented and if the reference count reaches zero, the node object is deleted. Therefore, in the above snippet of user code, the reference count drops from four to three when `b2_ptr` is destroyed, and from three to two when `b2_ptr` is destroyed and so on until `a_ptr1` is destroyed. When `a_ptr1` is destroyed the reference count goes to zero and the statement `delete node_;` is executed. This then calls the destructor

```
template<class T, class Dealloc_T>
class RefCountPtr_node_tmpl : public RefCountPtr_node {
public:
  ...
  ~RefCountPtr_node_tmpl()
  { if( has_ownership() ) dealloc_.free(ptr_); }
  ...
};
```

which will call on the template deallocator policy object to free the maneged object if it has ownership to do so. As mentioned earlier, because of the way that the very first `RefCountPtr<>` object was created, the pointer `ptr_` is exactly the same pointer returned from `new C` which addresses the problem discussed in Appendix C. In this case, the default deallocator

```
template<class T>
class DeallocDelete { public: void free( T* ptr ) { if(ptr) delete ptr; }  };
```

is used which calls `delete ptr;` to finally free the managed object.

B.3 Management of extra data

ToDo: Update this for new specification for set_extra_data(...)!

The last issue to discuss is how extra data is managed as described in Section 2.2. As shown in Figure B.1, each `RefCountPtr_node` object contains an array of `any` objects. The `any` class used here is an adaptation of the `boos::any` class [???] that is also contained in the Teuchos package and exists in the `Teuchos` namespace.

The class `any` uses RTTI which allows it to store any type of built-in or user-defined type that has value semantics (i.e. has a copy constructor and an assignment operators defined) and then to retrieve it upon request (assuming the client knows the correct type).

The templated extra-date manipulation functions

```
template<class T1, class T2>
int Teuchos::set_extra_data( const T1 &extra_data, Teuchos::RefCountPtr<T2> *p, int ctx )
{
  return p->access_node()->set_extra_data( extra_data, ctx );
}

template<class T1, class T2>
T1& Teuchos::get_extra_data( RefCountPtr<T2>& p, int ctx )
{
  return any_cast<T1>(p.access_node()->get_extra_data(ctx));
}
```

first described in Section 2.2 call on the non-templated functions

```
class RefCountPtr_node {
public:
  ...
  int set_extra_data( const any &extra_data, int ctx );
  any& get_extra_data( int ctx );
  const any& get_extra_data( int ctx ) const;
  ...
};
```

that actually manipulate the array of `any` objects.

Note that since `RefCountPtr<>` is itself a class that uses value semantics that it can be stored in an `any` object and therefore one or more `RefCountPtr<>` objects can be associated with an existing `RefCountPtr<>` object using `set_extra_data()` (this was described in Section 2.2).

C `RefCountPtr<>` and multiple inheritance and virtual base classes

In this section we discuss issues related to the proper deletion of objects that are from types that use virtual base classes and why it is important to follow Commandment 1 involving the use of the template function `rcp()` when using `new` to allocate an object to construct a `RefCountPtr<>` object. The issue considered here should be handled properly by ANSI C++ standard implementations and should not necessitate the use of template function `rcp()` but there are older C++ implementations that do not handle the deletion of these types of objects properly therefore requiring the use of `rcp()`. Here we describe what the issues are using a concrete example and describe why some C++ implementations have trouble getting this correct and how `rcp()` fixes the problem.

Consider the following class hierarchy:

```
class A {
  private: int A_g_, A_f_;
  public:  A(); virtual ~A(); virtual int A_g(); virtual int A_f();
};

class B1 : virtual public A {
  private: int B1_g_, B1_f_;
  public:  B1(); virtual int B1_g(); virtual int B1_f();
};

class B2 : virtual public A {
  private: int B2_g_, B2_f_;
  public:  B2(); virtual int B2_g(); virtual int B2_f();
};

class C : virtual public B1, virtual public B2
{
  private: int C_g_, C_f_;
  public:  C(); virtual int C_g(); virtual int C_f();
};
```

An object of type C should have one type B1 object, one type B2 object and one (since A is a virtual base class) type A object. Each of these objects must have at least storage for two `int` data members. On a 32 bit system, each `int` should require four bytes. In addition, each object should have a pointer to a virtual function table which on a 32 bit system is four bytes per pointer which is the same size as an `int`. Therefore, an object of type `C` should have a minimum size (as returned by `sizeof(C)`) of

$$(4\,\text{objects per C object})(3\,\text{ints per object})(4\,\text{bytes per int}) = 48\,\text{bytes per C object}.$$

While the layout of objects is compiler dependent, many C++ implementations sets up objects in memory in this manner.

A simple test program (this is the test program for `RefCountPtr<>` in Teuchos) was written to reveal the layout of an object of type `C` in memory. When run under the g++ version 3.2 the test program output reveals the following structure

```
Size of C = 48
Base address of object of type C        = 0xa040948
Offset to address of object of type C   = 0
Offset of B1 object in object of type C = 12
Offset of B2 object in object of type C = 36
Offset of A object in object of type C  = 24
```

This printout shows that an object of type `C` is laid out in memory as shown in Figure 6.

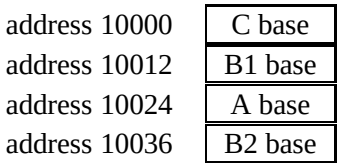

Figure 6. Layout of object of type `C` using g++ version 3.2 with a base address of (base-10) 10000 for example.

Why is the layout in Figure 6 significant? To understand why this is important, consider how a C++ implementation is likely to allocate and delete objects of type `C`. The way that an object of type `C` is usually constructed by a C++ implementation is to call `malloc(sizeof(C))` first to allocate storage for the object and then the sub-objects in `C` are constructed according to the standard (i.e. base-class objects first, followed by subclass objects).

Now consider the following program:

```
void delete_A(A* a_ptr)
{
    delete a_ptr;
}

void main()
{
    A *a_ptr = new C;
    delete_A(a_ptr);
}
```

Some older C++ implementations (most notably Microsoft Visual C++ version 6.0) implement the above `delete` call in the function `delete_A()` as follows:

```
a_ptr->~a_ptr();
free((void*)a_ptr);
```

and the program would crash with a runtime error from `free()` complaining that this memory was not allocated from `malloc()`. The reason for the runtime error given the above (incorrect) implementation is obvious if one thinks about what is occurring. Let us assume that the runtime allocates memory for the object `C` on the line

```
A *a_ptr = new C;
```

with the address 10000 returned from `malloc()` as shown in Figure 6. When the pointer variable `a_ptr` is assigned, the address is changed to 100024 as also shown in Figure 6. When this object is passed into the function `delete_A()` the above (incorrect) implementation first performs

```
a_ptr->~a_ptr();
```

and the correct destructors are called because this utilizes the virtual function mechanism. However, when the next line

```
free((void*)a_ptr);
```

is executed it is equivalent to

```
free((void*)10024);
```

which is not the same address 10000 returned from `malloc()` and hence the runtime error.

A fully ANSI/ISO C++ standard implementation must therefore perform some type of runtime lookup of an object to determine its base address before it calls `free()`. Hence we see that a proper C++ implementation of `delete` for an object with a virtual function must always perform a runtime type lookup to execute correctly and this is unavoidable extra overhead.

The reason that the simplistic (incorrect) implementation (such as implemented in MS Visual C++ 6.0) shown above generally works just fine is that if virtual inheritance is not used, then the address for every interface in an object is the same as the base address for the entire object in almost every

C++ implementation (and certainly for MS Visual C++ 6.0). And therefore any interface supported by the object can be used to delete the object since the addresses are the same.

Now let us consider what a `RefCountPtr<>` implementation of the above program would look like and how it works even with the incorrect implementation of `delete` (e.g. as used in MS Visual C++ 6.0).

```
void delete_A_ptr(Teuchos::RefCountPtr<A>* a_ptr)
{
    *a_ptr = Teuchos::null;
}

void main()
{
    Teuchos::RefCountPtr<A> a_ptr = Teuchos::rcp(new C);
    delete_A_ptr(&a_ptr);
}
```

In the above example, the line

```
Teuchos::RefCountPtr<A> a_ptr = Teuchos::rcp(new C);
```

results in the templated function `rcp()` first creating a `RefCountPtr<C>` object which stores the exact same address as returned from `new C` in the underlying reference-count node and then the `RefCountPtr<C>` object is converted into a `RefCountPtr<A>` object before assignment is performed (or just a copy constructor in most C++ implementations). See Appendix B for a description of how this works. Then when the line

```
*a_ptr = Teuchos::null;
```

in the function `delete_A_ptr()` is executed, the destructor for the underlying reference-count node calls `delete` on the exact same address returned from `new C` which was set when the first `RefCountPtr<C>` object was constructed by `rcp(new C)`. Since this is the same address returned form `new` (and therefore the same returned from `malloc()`) the incorrect compiler implementation (i.e. MS VC++ 6.0) works just fine. The key to the use of the templated function `rcp()` is that the compiler automatically constructs a `RefCountPtr<>` with the proper type without the programmer having to think about it.

New versions of C++ compilers do not seem to suffer from this problem and the above referenced g++ version performs the above deletion correctly as well as several other compilers tested. At the time of this writing it is unclear what platforms still suffer from this problem and whether Commandment 1 and the use of `rcp()` is really necessary but the use of `rcp()` is safe in any case and is therefore still recommended.