Supporting alternative pointer types in STL containers

This document is some notes / lessons learned trying to write relative_ptr<T>, a class which is intended to behave like a standard T* to as much of a degree as is possible, but uses an alternative storage form. The intent was to enable this class to be used as the designated pointer type with an allocator, and to have the libstdc++ STL containers support this use of a non-standard pointer type.

Why?

The STL standard has the allocator type, which controls allocation, but the allocator also has a pointer typedef which defines the type to be used for pointers within the STL container. The standard (C++03) allows STL libraries to assume that the typedef of allocator<T>::pointer is T*, but 'encourages' libraries to support non-standard pointer types.

A key motivation for needing a non-standard pointer type is to support an alternative physical storage for the pointer, such as the offset_ptr<> from Boost.Interprocess. Alternative pointers could allow STL containers to not only be allocated within shared memory segments, but would also allow the STL containers to function correctly even when the different processes end up mapping the shared region at different virtual addresses.

Speculation: Because the memory allocation and deallocation mechanism is typically controlled by overriding the allocator type, the key reason for overriding the allocator::pointer typedef to something other than T* would be to change the physical storage characteristics of the pointer type.

Current State of Alternative Pointer Implementations

Compliance with the standard would be easy if the burden fell entirely to the alternative pointer. i.e. libstdc++ works with any allocator::pointer typedef so long as that type is syntactically and semantically equivalent to a pointer.

However, it does not seem to be possible to write a class that behaves in the same way as a standard pointer.

My first pass at trying to show that a non-standard pointer type could be used with the libstdc++ STL involving taking some existing pointer types and seeing how they behaved when used in this way. I experimented

with the Loki smart pointer, the Boost.interprocess offset_ptr, and the ACE libraries 'based_ptr_T' implementations. When compiling STL containers which used these pointer types, I encountered a number of situations that the pointer wasn't supporting (invalid cast errors), and also several cases where the compiler reported ambiguities in compiling certain expressions.

Implementing relative_ptr<T>

relative_ptr<T> was devised by experimenting with 'map' and 'const map', seeking to get them to compile completely with minimal code changes to the stl_tree.h code, using the relative_ptr<> type as the allocator::pointer typedef.

The appropriate design of the pointer seems to be:

* ptr<T>	equivalent to	T*;
* ptr<const T>	equivalent to	const T*;
* const ptr<T>	equivalent to	T * const;
* const ptr<const T>	equivalent to	const T * const;

Although it was tempting to try “const ptr<T>” as the equivalent of “const T*”, such a convention leaves you with no good way to represent “T * const”, and seemed to create even more compile errors and code difficulties than the convention described above.

The following sections explain some design challenges to help convey how 'tricky' it can be to write a class which behaves like a pointer...

Challenge #1: static cast between members of a class hierarchy

Simple case really: The relative_ptr<T> is statically cast to T2* where T and T2 are in the same class hierarchy.

The expression:

```
Relative_ptr<T>  ptr;
T2* ptr2 = static_cast<T2*>(ptr);     // invalid cast
```

The above could be made to work by provide a templatized conversion operator as part of relative_ptr

```
template <typename T2> operator T2*() const {
        return static_cast<T2*>( this->get() );
}
```

This method allows the previous expression to compile. The presence of just an operator T*() does not allow that expression to compile with gcc 4.4.0.

Design Challenge #2: const casts

Const_cast<> between relative_ptr<T> and relative_ptr<const T> is not considered valid to the compiler. Such a cast can be done using static_cast<>.

We can use the convention of constructor-based conversion when casting from T* to const T*.

```
Base_ptr ptr;
Const_Base_ptr temp(ptr);
Int result = foo( temp );
```

This notation works for relative_ptr<> and also standard pointers. However, if we try the reverse (construct Base_ptr from an instance of Const_Base_ptr) it works for relative_ptr<>, but causes a compile error with standard pointers. Standard pointers require an explicit const_cast<> to go from const to non-const.

The only solution I could devise to this paradox was to use an intermediate class to accomplish the const_cast. I chose to use Ion Gaztanaga's 'cast_to' template (from Boost.interproces offset_ptr.h) as the basis for a custom casting class, which works appropriately to the pointer type (via pointer specific specialization.)

The code in std_tree.h which relies on a const_cast is then altered as follows:

```
Base_ptr ptr;
Int result = foo(  cast_to<Const_Base_ptr>::using_const_cast(ptr)  );
```

Design Challenge #3: ambiguities when calling overloaded functions.

Adding the templated conversion operator from challenge #1 gives rise to ambiguities when calling overloaded functions. With a standard pointer, the compiler can easily figure out whether an implicit cast to const* or a base pointer type is called for. However, with a relative_ptr, it just sees two approaches involving a call to a conversion constructor, either of which will work.

```
Int foo ( T* arg ) { … };
Int foo ( T2* arg ) { … };
```

```
Relative_ptr<T>  ptr;
Int result = foo(ptr);       // ambiguity problem
```

The removal of the templated conversion operator would address this, but with the consequence of making the static_cast<T2> operations fail. This is a good example of a paradox where an attempt to fix one problem exacerbates the other.

The approach which was used to address this is explicit downcasting (via static_cast) to make the version of the overloaded function explicit.

Design Challenge #4: Const member functions.

Const member functions exacerbate the problems described previously, because they implicitly convert members of type relative_ptr<T> to const relative_ptr<T>, and not to the desired relative_ptr<const T>.

```
template <class Alloc>
class X {
public:
    typedef Alloc::pointer pointer;
    typedef Alloc::const_pointer const_pointer;

    // non-const version of begin
    pointer begin() {
        return _member;
    }

    // const version of begin
    const_pointer begin() const {
        return _member;    // problem with conversion here
    }

private:
    pointer _member;
};
```

The above is a common symptom of design of classes which assume that Alloc::pointer and Alloc::const_pointer are T* and const T* respectively. The code above finds itself (in the const version of begin()) trying to convert a “const pointer<T>” to a “relative_ptr<const T>” implicitly (in the const version of begin).

My solution to this: partially specialize relative_ptr for <const T>, and in that version, add a constructors allowing conversion from a const

relative_ptr<T>. Specifically relative_ptr <const T> has a non-explicit constructor that serves this purpose:

```
relative_ptr( const relative_ptr <T> &rarg );
```

This constructor is the equivalent of implicit casting from T* to const T*. A side effect of using this is that this relative pointer supports the implicit casing of T const * to const T*, which is not strictly 'legal'.

Design Challenge #5: Implicit const conversion ambiguities

Here's the second problem in code form:

```
class Base {
public:
   typedef relative_ptr<Base>  pointer;
   typedef relative_ptr<const Base> const_pointer;
   ...
   void foo( const_pointer rarg ) const;
   void foo( pointer rarg );
}

class Derived : public Base {
   ...
}

problem()
{
   Derived a;
   relative_ptr<Derived> d( new Derived() );

   a.foo( d );  // ambiguity
};
```

Originally in my pointer implementation, I had included templated constructors in the different pointer implementations which allowed conversion along multiple paths.

To reduce this ambiguity I had to comment out the following in the offset_ptr<const T> version:

```
template <typename T2>
relative_ptr( const relative_ptr<T2>& arg );
```

I did leave in place a version which takes const relative_ptr<const T2>& however. That version still allows implicit conversion from a const pointer of a different type in the same type hierarchy.

Making STL containers support alternative pointers:

Support for non-standard pointer types involves a few changes to the STL container implementation:

- Internal structures have to become templated and accept an Allocator parameter, so that they have access to the pointer typedef.
- Instead of the direct use of const_cast<T>, static_cast<T>, etc. the code can be changed to cast_to<T>::using_const_cast, cast_to<T>::using_static_cast, and so on.
- Aside from the use of these templates, developers should otherwise be able to assume that the pointer typedef refers to the standard pointer type in all other cases.

The relative_ptr<> class can be refactored slightly into a generic pointer<> class that accepts a storage policy template parameter, which defines a simpler interface for storing and yielding a pointer value. In this way, pointer<> actually becomes reusable as a wrapper (facade) which addresses many of the ambiguity issues in the pointer interface, and simultaneously serves as a way to document the requirements of a class which works as a pointer type with libstdc++.

Test Suite:

The relative_ptr <> class would be included with the test suite, and a testing mode enabled which run the current tests with std::allocator using that pointer type. This confirms the ability to compile and run the bulk of the template code using that alternative pointer type. It also provides a means to confirm that this capability of libstdc++ isn't being compromised.

STL_tree.h case:

In the course of making this change, the _Rb_tree_node_base class had to become templated, so that the pointer type could be passed to it. While this is an internal class, its methods are built into the libstdc++ library and constitute a portion of the library's ABI.

To preserve ABI compatibility the structure _Rb_tree_node_base was preserved as the base of _Rb_tree_node<_Val,_Alloc> for cases where _Alloc is some form of std::allocator<T>. This allowed ABI compatibility to be preserved.

The only other things needed to Stl_tre.h to complete the process were:

- Const_cast<> had to be converted to cast_to<>::using_const_cast.
- There were some function calls which were depending on implicit conversion from _Link_type down to either _Const_Base_ptr or _Base_ptr, and this resulted in compiler ambiguities. To address this, some additional versions of some functions were created.