pointee and indirect_reference

Author: David Abrahams
Contact: dave@boost-consulting.com
Organization: Boost Consulting
Date: 2005-02-27

abstract: Provides the capability to deduce the referent types of pointers, smart pointers and iterators in generic code.

Overview

Have you ever wanted to write a generic function that can operate on any kind of dereferenceable object? If you have, you've probably run into the problem of how to determine the type that the object "points at":

```
template <class Dereferenceable>
void f(Dereferenceable p)
{
    what-goes-here? value = *p;
    ...
}
```

`pointee`

It turns out to be impossible to come up with a fully-general algorithm to do determine *what-goes-here* directly, but it is possible to require that `pointee<Dereferenceable>::type` is correct. Naturally, `pointee` has the same difficulty: it can't determine the appropriate `::type` reliably for all `Dereferenceables`, but it makes very good guesses (it works for all pointers, standard and boost smart pointers, and iterators), and when it guesses wrongly, it can be specialized as necessary:

```
namespace boost
{
  template <class T>
  struct pointee<third_party_lib::smart_pointer<T> >
  {
      typedef T type;
  };
}
```

`indirect_reference`

`indirect_reference<T>::type` is rather more specialized than `pointee`, and is meant to be used to forward the result of dereferencing an object of its argument type. Most dereferenceable types just return a reference to their pointee, but some return proxy references or return the pointee by value. When that information is needed, call on `indirect_reference`.

Both of these templates are essential to the correct functioning of `indirect_iterator`.

Reference

`pointee`

```
template <class Dereferenceable>
struct pointee
{
    typedef /* see below */ type;
};
```

Requires: For an object `x` of type `Dereferenceable`, `*x` is well-formed. If `++x` is ill-formed it shall neither be ambiguous nor shall it violate access control, and `Dereferenceable::element_type` shall be an accessible type. Otherwise `iterator_traits<Dereferenceable>::value_t` shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of `pointee`]

`type` is determined according to the following algorithm, where `x` is an object of type `Dereferenceable`:

```
if ( ++x is ill-formed )
{
    return ``Dereferenceable::element_type``
}
else if (``*x`` is a mutable reference to
         std::iterator_traits<Dereferenceable>::value_type)
{
    return iterator_traits<Dereferenceable>::value_type
}
else
{
    return iterator_traits<Dereferenceable>::value_type const
}
```

`indirect_reference`

```
template <class Dereferenceable>
struct indirect_reference
{
    typedef /* see below */ type;
};
```

Requires: For an object `x` of type `Dereferenceable`, `*x` is well-formed. If `++x` is ill-formed it shall neither be ambiguous nor shall it violate access control, and `pointee<Dereferenceable>::type&` shall be well-formed. Otherwise `iterator_traits<Dereferenceable>::reference` shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of `indirect_reference`]

`type` is determined according to the following algorithm, where `x` is an object of type `Dereferenceable`:

```
if ( ++x is ill-formed )
    return ‘‘pointee<Dereferenceable>::type&‘‘
else
    std::iterator_traits<Dereferenceable>::reference
```