8/2/2010

Traits

GERMAN
CANCER RESEARCH CENTER
IN THE HELMHOLTZ ASSOCIATION

What traits are and what they are not

- Traits are NOT
  - A special keyword
  - A predefined construct or property
- Traits ARE
  - A technique used in template programming
  - A convention among C++ programmers
  - A way to achieve at compile time what could be achieved at runtime in another way
  - *“Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine "policy" or "implementation details".”*
    - Bjarne Stroustrup

Why traits

- Writing efficient code sometimes requires knowing things about your template parameter
- You can not insert that information into build in types
- Preferably that information should be available at compile time

An example

- Consider the following function working on iterators

```
template < typename IterT, typename DistT >
void advance( IterT& iter, DistT d )
{
  while ( d-- ) //assume d to be always positive
    ++iter;
}
```

- Inefficient for random access iterators

An example

- Instead we would like something like this

```
template < typename IterT, typename DistT >
void advance( IterT& iter, DistT d )
{
  if( iter is a random access iterator )
  {
    iter += d;
  }
  else {
    while ( d-- ) //assume d to be always positive
      ++iter;
  }
}
```

How to implement

- Nesting information for a type is not sufficient as information can not be nested in pointers
- Convention is to implement traits as structs

```
template < typename IterT >
struct iterator_traits {
  typedef typename IterT::iterator_category iterator_category;
  …
}
```

- Partial specialization for pointers

```
template < typename T >
struct iterator_traits <T*>{
  typedef random_access_iterator_tag iterator_category;
  …
}
```

How to implement

- User defined iterators must contain corresponding typedef

```
class iterator{
public:
  typedef bidirectional_iterator_tag iterator_category;
  ...
};
```

How to implement

- Overload functions based on traits

```
template< typename IterT, typename DistT >
void doAdvance( IterT& iter, DistT d, std::random_access_iterator_tag )
{
  iter += d;
}

template< typename IterT, typename DistT >
void doAdvance( IterT& iter, DistT d, std::bidirectional_iterator_tag )
{
  while( d-- )
    ++iter;
}
```

How to implement

- Call overloaded functions using the traits struct

```
template< typename IterT, typename DistT >
void advance( IterT& iter, DistT d )
{
  doAdvance( iter, d,
    typename std::iterator_traits< IterT >::iterator_category()
  );
}
```

Gain

- Now a call to advance will result in the appropriate efficient implementation of doAdvance being decided upon at compile time

Further information

- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. ISBN 0-321-33487-6
- An introduction to C++ Traits http://accu.org/index.php/journals/442