Template Policies and Traits By Example

Jim Fawcett

CSE687 – Object Oriented Design

Spring 2005

Template Policies

- A policy is a class designed to tailor behavior of a template class in some narrow specific way:
  - Locking policy for class that may be used in multi-threaded program:
    - template<typename T, typename LockPolicy> class queue;

    Allows designer to use Lock or noLock policy.
  - Enqueuing policy for a thread class:
    - template <typename queue> class thread;

    Allows designer to optionally add queue and queue operations as part of thread class's functionality.
  - HashTable policy for hashing table addresses:
    - template <typename key, typename value, typename Hash> class HashTable;

    Allows designer to provide hashing tailored for application long after HashTable class was designed.

Hashing Policy

```
template <typename T>
class HashBase
{
public:
   HashBase() : _size(0) {};
   void Size(long int size) { _size = size; }
   virtual long int operator() (const T& t) const = 0;
protected:
   long int _size;
};

class HashString : public HashBase<std::string>
{
public:
   long int operator() (const std::string& s) const;
};
```

Hashing Policy Class

- HashBase:
  - Provides protocol, e.g.: long int operator()(const T& t)
  - Has _size member and Size(long int) to allow HashTable to inform it about the table size. HashTable does this in its constructor.
- HashString:
  - Provides the hashing algorithm for its std::string type.
- These hash classes are functors – function objects. The operator()(T t) provides function call syntax:
  - HashString hash(size);

    long int loc = hash.operator()(key)   ⇔   long int loc = hash(key)

Why Use Non-Polymorphic HashBase?

- The HashBase class has two significant roles to play:
  - Prevent users from creating hash table for type without a suitable hash function. Results in compile-time failure.
  - Provide table size to user-defined class without the user needing to figure out how to get the table size from the container, or even remembering to do so. HashBase and HashTable conspire to take care of that detail.

Traits are Types and Values that Support Generic Programming

- Traits types are introduced by typedefs:
  - typedef key key_type;
  - typedef value value_type;
  - typedef HashIterator iterator;

- Traits allow a template parameterized class to be used in a function that is not aware of the parameter types.
  - The function simply uses the "standard" name for the type provided by the class's traits

Trait Types: value_type, iterator

```
//----< sum values in container >--------------------------
//
//  Demonstrates use of container traits
//    - Sums values across all elements of container
//    - could just as easily sum keys using key_type
//      declaration
//
template <typename Cont>
Cont::value_type sum(Cont &c)
{
  Cont::value_type sum = Cont::value_type();
  Cont::iterator it = c.begin();
  while(it != c.end())
  {
    sum += (it->Value());
    ++it;
  }
  return sum;
}
```

Traits Used in Cont::value_type sum(Cont &c)

- In the HashTable module – sum function:
  - Container’s real value type, unknown to sum:

    ```
    std::string value;
    ```

    Trait type, provided by all containers that use sum:

    ```
    Cont::value_type value;
    ```

  - Container’s real iterator type:

    ```
    HashIterator<double,string,HashDouble> it
    ```

    Trait type:

    ```
    Cont::iterator it
    ```

End Of Traits and Policies

- Policies allow a designer of a library to make the behavior of a class configurable at application design time.
  - To lock or not to lock
  - To enqueue or not enqueue
  - Which way to hash

- Traits provide common type aliases used by all containers so functions that operate on the containers can be written to apply to every one of them without modification.
  - std::string ↔ value_type
  - std::string& ↔ reference_type
  - std::string* ↔ pointer_type
  - HashIterator<double,string,HashDouble> ↔ iterator