Instigate STL

Introduction

This is a concept proof implementation of STL algorithms to show that Instigate GP methodology can be used for implementing generic libraries for real-life applications. Algorithms themselves where taken from standard implementations (SGI, GNU, boost). The library contains most of standard STL algorithms, and some additions from boost.org and other generic libraries.

An important note on compatibility and completeness: the library provides only algorithms and function binders/adaptors. It does not provide any data types such as container classes. There are two reasons for this:

1. it is compatible with standard STL implementations, and algorithms/functors from Instigate STL can be used in combination with standard STL containers
2. we believe that truly Generic libraries should provide algorithms that would work with user data types.

Below are more details about Instigate's GP methodology.

The main goal is to implement truly generic algorithms, which do not impose any syntactical requirements on their argument types. For example the STL algorithm

```
UnaryFunctor for_each(Iterator b, Iterator e, UnaryFunctor f);
```

requires it's argument type "Iterator" to be model of concept "InputIterator", which in classical STL implies syntactical requirement of having dereference operator "operator*()", and it would not work on iterators, which use something like "get_current()" or "dereference()", although conceptually the class would still be an iterator if it had all the required operations of an iterator.

Instigate STL's "for_each" algorithm would work on any class T, which _conceptually_ is an InputIterator. Which means there must be some way to get its current value, not necessarily via operator*(), and some way to increment it, not necessarily operator++(), etc. Such genericity is achieved by using concept interfaces, which are template classes, providing interfaces to the models of given concept.

Theory

Generic algorithm is defined as an algorithm implemented in terms of Concept, as opposed to a function in Structured Programming being defined in terms of Types.

Concept is defined as set of types (Models) satisfying requirements (Specification) of the Concept.

Concept Specification consists of required Operations that should be supported for the **models of the concept**, as opposed to Structured Programming where Type is defined as tuple <V,O> of set of values V and set of operations O, here we have just set of operations O.

Concept specification also has to define **associated types**, because some operations may involve other supplementary types. For example the operation 'dereference' of a concept 'Input Iterator' involves an associated types 'Value Type' and 'Reference'.

Concept C' is called **refinement** of concept C if Specification of C is a subset of the specification of C', which is equivalent to set of models of C being superset of the set of models of C'.

Refinement is the analogue of inheritance in OOP, and thus comes need of **polymorphism** (in this case static), which is well covered in the literature and to save space we do not provide here (see tag-based dispatching techniques applied in original STL implementation).

Other similar projects

The STL library was standardized in '94 when this methodology and terminology was not fully developed, Alexander Stepanov's and David Musser's ideas had yet to evolve into the modern GP theory/methodology.

Currently there is no programming language truly supporting GP. In ISO/ANSI C++ '98 the "generic algorithms" are defined using function templates, and lack argument checking (model compliance to concept requirements). Concepts are defined only in documentation and models are defined by making proper interface, matching the documentation requirements. Industry and ISO/ANSI C++ team are working towards enhancement of C++ that would allow to fill this gap and provide necessary constructs for true GP in C++.

In the meantime community has developed series of tricks for implementing all aspects of the GP without relying on enhancements to '98 ISO/ANSI C++ standard. Some of these tricks, however, require very deep understanding of template syntax and more complex theory/notions than mentioned above, which complicates and slows down adoption of these methods in community and industry.

Instigate's GP technology is an attempt to summarise these tricks and the theory into a full-featured and yet simple enough set of techniques, that would allow to implement generic libraries.

Main advantage of this methodology over the one used in standard STL of '94 is that there are only semantical requirements on parameter types, and all syntactical requirements were practically eliminated. E.g. as mentioned in example above, the "for_each" would work on any iterator, even if it's dereference operation is not implemented as "operator*()" and increment is not done using “operator++()”. The only exception are notions of Default Constructible and Assignable, which due to specifics of the C++ language have to impose syntactical requirements on the models.

Technical details

True GP language/framework should allow to:

1. define a generic algorithm (e.g. A(T x)) operating on specific concepts (e.g. T must be Assignable)
2. define a concept by specifying its associated types and required operations
3. define a model T of a concept C. I.e. provide map of T's actual operators to operators required by the concept C's specification, even if syntactically they do not match. Example is iterator's dereference operation being called "get_current()" instead of "operator*()".
4. define concept C' being a refinement of concept C.
5. define polymorphic algorithm A(C) which would behave differently if actual argument passed to it is a model of some concept C' which is a refinement of C.

Instigate's GP approach suggests the following techniques to achieve above mentioned requirements.

For each concept C define an interface template structure, which is something like traits used in classical STL. However, it defines not only the associated types, but also operations of the Concept. For example, InputIterator concept can be defined as follows.

```
namespace input_iterator {
        template <typename T> struct interface {
                typedef typename
                std::iterator_traits<T>::value_type value_type;
                ...
                const value_type& dereference(T i) const
                {
                        return *i;
                }
                void increment(T& i)
                {
                        ++i;
                }
                ...
        }
};
```

Note that it doesn't matter what we write in the default implementation, as for each concrete model it can be specialized and operations can be remapped to something else. Our default implementation is provided so that it automatically works for standard STL types:

The C++ Template Class Specialization capability is used to define models. E.g. assume the class `MyInputIterator` is some old-fashioned iterator class with interface not matching requirements of standard STL. Then to make it a model of concept InputIterator one should define following:

```
namespace input_iterator {
        template <> struct interface<::MyInputIterator> {
                typedef int value_type;
                ...
                int dereference(const MyInputIterator& x)
                {
                        return x.get_current();
                }
                void increment( MyInputIterator& x)
                {
                        x = x.get_next();
                }
                ...
        }
}
```

The following structure is used to define concept requirements:

```
namespace input_iterator {
        template <typename T> struct requirements {
                requirements()
                {
                        &require_value_type;
                        &value_type_must_be_assignable;
                        // etc.
                }

                // will compile only if interface<T> provides nested
                // type name 'value_type'. Note that we don't put any
                // syntactical requirements on the type 'T' itself
                void require_value_type()
                {
                        typedef typename
                        input_iterator<T>::value_type REQUIRED;
                }

                // now as we are sure that typename 'value_type' is
                // defined we should also make sure it's a model of
                // concept Assignable
               void value_type_must_be_assignable
               {
                        typedef typename
                        input_iterator<T>::value_type VALUE_TYPE;
                        //macro
                        CHECK(assignable::requirements<VALUE_TYPE>);
               }
        }
};
```

To check the requirements from within the generic algorithms, use preprocessor macro CHECK:

```
template <typename InputIterator, typename UnaryFunction>
UnaryFunction for_each(InputIterator b,
                        InputIterator e, UnaryFunction f)
{
        CHECK(input_iterator::requirements<InputIterator>);
        CHECK(unary_function::requirements<UnaryFunction>);
        ...
}
```

To implement generic algorithms without imposing any syntactical requirement on its argument types use interfaces:

```
template <typename InputIterator, typename UnaryFunction>
UnaryFunction for_each(InputIterator b,
                       InputIterator e, UnaryFunction f)
{
        ... // CHECK requirements
        typedef input_iterator::interface<InputIterator> II;
        typedef unary_function::interface<UnaryFunction> UF;
        while (! II::equal( b, e) ) {
                UF::invoke(f, II::dereference(b));
                II::increment(b);
        }
        return f;
}
```

The only syntactical requirement here is that assignability is implemented via copy-constructor (due to specifics/limitations of C++ copy constructor syntax).

All the other requirements are completely semantical. For example the function doesn't have to be invoked using operator(). It could as well be something like `f->call(arg);` It still can be used as an argument to Instigate STL "for_each" algorithm.

Finally, the refinement is defined by simply inheriting base concept interfaces (both for specification and for models) and by inheriting base concept requirements (note that constructor of the base class will be invoked automatically and do all the checks of the base concept specification).