Generic Programming Techniques

This is an incomplete survey of some of the generic programming techniques used in the boost libraries.

Table of Contents

Introduction

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

As a simple example of generic programming, we will look at how one might generalize the `memcpy()` function of the C standard library. An implementation of `memcpy()` might look like the following:

```
void* memcpy(void* region1, const void* region2, size_t n)
{
  const char* first = (const char*)region2;
  const char* last = ((const char*)region2) + n;
  char* result = (char*)region1;
  while (first != last)
    *result++ = *first++;
  return result;
}
```

The `memcpy()` function is already generalized to some extent by the use of `void*` so that the function can be used to copy arrays of different kinds of data. But what if the data we would like to copy is not in an array? Perhaps it is in a linked list. Can we generalize the notion of copy to any sequence of elements? Looking at the body of `memcpy()`, the function's ***minimal requirements*** are that it needs to *traverse* through the sequence using some sort of pointer, *access* elements pointed to, *write* the elements to the destination, and *compare* pointers to know when to stop. The C++ standard library groups requirements such as these into ***concepts***, in this case the Input Iterator concept (for `region2`) and the Output Iterator concept (for `region1`).

If we rewrite the `memcpy()` as a function template, and use the Input Iterator and Output Iterator concepts to describe the requirements on the template parameters, we can implement a highly reusable `copy()` function in the following way:

```
template <typename InputIterator, typename OutputIterator>
OutputIterator
copy(InputIterator first, InputIterator last, OutputIterator result)
{
  while (first != last)
```

```
    *result++ = *first++;
  return result;
}
```

Using the generic `copy()` function, we can now copy elements from any kind of sequence, including a linked list that exports iterators such as `std::list` .

```
#include <list>
#include <vector>
#include <iostream>

int main()
{
  const int N = 3;
  std::vector<int> region1(N);
  std::list<int> region2;

  region2.push_back(1);
  region2.push_back(0);
  region2.push_back(3);

  std::copy(region2.begin(), region2.end(), region1.begin());

  for (int i = 0; i < N; ++i)
    std::cout << region1[i] << " ";
  std::cout << std::endl;
}
```

Anatomy of a Concept

A ***concept*** is a set of requirements consisting of valid expressions, associated types, invariants, and complexity guarantees. A type that satisfies the requirements is said to ***model*** the concept. A concept can extend the requirements of another concept, which is called ***refinement***.

- **Valid Expressions** are C++ expressions which must compile successfully for the objects involved in the expression to be considered *models* of the concept.
- **Associated Types** are types that are related to the modeling type in that they participate in one or more of the valid expressions. Typically associated types can be accessed either through typedefs nested within a class definition for the modeling type, or they are accessed through a traits class.
- **Invariants** are run-time characteristics of the objects that must always be true, that is, the functions involving the objects must preserve these characteristics. The invariants often take the form of pre-conditions and post-conditions.
- **Complexity Guarantees** are maximum limits on how long the execution of one of the valid expressions will take, or how much of various resources its computation will use.

The concepts used in the C++ Standard Library are documented at the C++ reference website.

Traits

A traits class provides a way of associating information with a compile-time entity (a type, integral constant, or address). For example, the class template `std::iterator_traits<T>` looks something like this:

```
template <class Iterator>
struct iterator_traits {
  typedef ... iterator_category;
  typedef ... value_type;
  typedef ... difference_type;
  typedef ... pointer;
```

```
  typedef ... reference;
};
```

The traits' `value_type` gives generic code the type which the iterator is "pointing at", while the `iterator_category` can be used to select more efficient algorithms depending on the iterator's capabilities.

A key feature of traits templates is that they're *non-intrusive*: they allow us to associate information with arbitrary types, including built-in types and types defined in third-party libraries, Normally, traits are specified for a particular type by (partially) specializing the traits template.

For an in-depth description of `std::iterator_traits`, see this page. Another very different expression of the traits idiom in the standard is `std::numeric_limits<T>` which provides constants describing the range and capabilities of numeric types.

Tag Dispatching

Tag dispatching is a way of using function overloading to dispatch based on properties of a type, and is often used hand in hand with traits classes. A good example of this synergy is the implementation of the `std::advance()` function in the C++ Standard Library, which increments an iterator `n` times. Depending on the kind of iterator, there are different optimizations that can be applied in the implementation. If the iterator is random access (can jump forward and backward arbitrary distances), then the `advance()` function can simply be implemented with `i += n`, and is very efficient: constant time. Other iterators must be `advanced` in steps, making the operation linear in `n`. If the iterator is bidirectional, then it makes sense for `n` to be negative, so we must decide whether to increment or decrement the iterator.

The relation between tag dispatching and traits classes is that the property used for dispatching (in this case the `iterator_category`) is often accessed through a traits class. The main `advance()` function uses the `iterator_traits` class to get the `iterator_category`. It then makes a call the the overloaded `advance_dispatch()` function. The appropriate `advance_dispatch()` is selected by the compiler based on whatever type the `iterator_category` resolves to, either `input_iterator_tag`, `bidirectional_iterator_tag`, or `random_access_iterator_tag`. A ***tag*** is simply a class whose only purpose is to convey some property for use in tag dispatching and similar techniques. Refer to this page for a more detailed description of iterator tags.

```
namespace std {
  struct input_iterator_tag { };
  struct bidirectional_iterator_tag { };
  struct random_access_iterator_tag { };

  namespace detail {
    template <class InputIterator, class Distance>
    void advance_dispatch(InputIterator& i, Distance n, input_iterator
      while (n--) ++i;
    }

    template <class BidirectionalIterator, class Distance>
    void advance_dispatch(BidirectionalIterator& i, Distance n,
       bidirectional_iterator_tag) {
      if (n >= 0)
        while (n--) ++i;
      else
        while (n++) --i;
    }

    template <class RandomAccessIterator, class Distance>
    void advance_dispatch(RandomAccessIterator& i, Distance n,
       random_access_iterator_tag) {
      i += n;
    }
  }

  template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n) {
    typename iterator_traits<InputIterator>::iterator_category categor
```

```
    detail::advance_dispatch(i, n, category);
  }
}
```

Adaptors

An *adaptor* is a class template which builds on another type or types to provide a new interface or behavioral variant. Examples of standard adaptors are std::reverse_iterator, which adapts an iterator type by reversing its motion upon increment/decrement, and std::stack , which adapts a container to provide a simple stack interface.

A more comprehensive review of the adaptors in the standard can be found here.

Type Generators

Note: The *type generator* concept has largely been superseded by the more refined notion of a *metafunction*. See *C++ Template Metaprogramming* for an in-depth discussion of metafunctions.

A *type generator* is a template whose only purpose is to synthesize a new type or types based on its template argument(s)[1]. The generated type is usually expressed as a nested typedef named, appropriately `type`. A type generator is usually used to consolidate a complicated type expression into a simple one. This example uses an old version of `iterator_adaptor` whose design didn't allow derived iterator types. As a result, every adapted iterator had to be a specialization of `iterator_adaptor` itself and generators were a convenient way to produce those types.

```
template <class Predicate, class Iterator,
    class Value = complicated default,
    class Reference = complicated default,
    class Pointer = complicated default,
    class Category = complicated default,
    class Distance = complicated default
        >
struct filter_iterator_generator {
    typedef iterator_adaptor<

        Iterator,filter_iterator_policies<Predicate,Iterator>,
        Value,Reference,Pointer,Category,Distance> type;
};
```

Now, that's complicated, but producing an adapted filter iterator using the generator is much easier. You can usually just write:

```
boost::filter_iterator_generator<my_predicate,my_base_iterator>::type
```

Object Generators

An *object generator* is a function template whose only purpose is to construct a new object out of its arguments. Think of it as a kind of generic constructor. An object generator may be more useful than a plain constructor when the exact type to be generated is difficult or impossible to express and the result of the generator can be passed directly to a function rather than stored in a variable. Most Boost object generators are named with the prefix "`make_`", after `std::make_pair(const T&, const U&)`.

For example, given:

```
struct widget {
  void tweak(int);
};
std::vector<widget *> widget_ptrs;
```

By chaining two standard object generators, `std::bind2nd()` and `std::mem_fun()`, we can easily tweak all widgets:

```
void tweak_all_widgets1(int arg)
{
   for_each(widget_ptrs.begin(), widget_ptrs.end(),
      bind2nd(std::mem_fun(&widget::tweak), arg));
}
```

Without using object generators the example above would look like this:

```
void tweak_all_widgets2(int arg)
{
   for_each(struct_ptrs.begin(), struct_ptrs.end(),
      std::binder2nd<std::mem_fun1_t<void, widget, int> >(
          std::mem_fun1_t<void, widget, int>(&widget::tweak), arg));
}
```

As expressions get more complicated the need to reduce the verbosity of type specification gets more compelling.

Policy Classes

A policy class is a template parameter used to transmit behavior. An example from the standard library is `std::allocator`, which supplies memory management behaviors to standard containers.

Policy classes have been explored in detail by Andrei Alexandrescu in this chapter of his book, *Modern C++ Design*. He writes:

> In brief, policy-based class design fosters assembling a class with complex behavior out of many little classes (called policies), each of which takes care of only one behavioral or structural aspect. As the name suggests, a policy establishes an interface pertaining to a specific issue. You can implement policies in various ways as long as you respect the policy interface.
>
> Because you can mix and match policies, you can achieve a combinatorial set of behaviors by using a small core of elementary components.

Andrei's description of policy classes suggests that their power is derived from granularity and orthogonality. Less-granular policy interfaces have been shown to work well in practice, though. This paper describes an old version of `iterator_adaptor` that used non-orthogonal policies. There is also precedent in the standard library: `std::char_traits`, despite its name, acts as a policies class that determines the behaviors of std::basic_string.

Notes

[1] Type generators are sometimes viewed as a workaround for the lack of “templated typedefs” in C++.