CS193D Handout 33
Autumn 2000 November 30, 2000

Templates Galore!

Introduction by Derek Poppink, partition text by Nihkil Sarin, code and practice by Jerry and Hiroshi.

Fighting Reality

As many of you know, dealing with run-time errors can be really frustrating, especially segmentation faults and bus errors. Those types of errors are not going to get much easier to deal with, so do your best to rigorously test your program throughout your programming. Design your program and your tests so that you can see your progress, incrementing a class's functionality little by little.

> "Item 46: Prefer compile-time and link-time errors to runtime errors."
>
> Other than in the few situations that cause C++ to throw exceptions, the notion of a runtime error is as foreign to C++ as it is to C. There's no detection of underflow, overflow, division by zero, no checking for array bounds violations, etc. Once your program gets past a compiler and linker, you're on your own – there's no safety net of any consequence. Much as with skydiving, some people are exhilarated by this state of affairs, others are paralyzed with fear. The motivation behind the philosophy, of course, is efficiency: without runtime checks, programs are smaller and faster.
>
> Never forget you are programming in C++. Whenever you can, push the detection of an error back from runtime to link-time, or, ideally, to compile-time."[1]

Templates and the STL

Shown on the following page is a function template `partition`. `partition` takes a range of elements (with beginning `start` and end `end`), and reorders the range such that all elements that satisfy the given `predicate` (the 'predicate' is a function that when passed an element, returns either `true` or `false`. Here, a pointer to this function is passed to the template `partition`) are collated at the front of range, and the elements that don't satisfy it are collected at the end of it. `partition` returns a pointer to that element in the range that is the first element (counting from the beginning of the range) of those that failed the `predicate`.

[1] Meyers, Scott. "Effective C++: 50 Specific Ways to Improve Your Programs and Designs", Addison-Wesley, Massachusetts, 1998.

```
template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator start,
                          ForwardIterator end,
                          Predicate pred)
{
   ForwardIterator next = start;
   while (start != end)
   {
      if (pred(*start))
      { // start belongs in front half
         swap(*next, *start);
         ++next;
      }
      ++start;
   }

   return next;
}
```

Note that we use the ubiquitous `swap` template as well.

Here's how we would use our `partition` template:

```
inline bool isEven(int a)
{
   return (a % 2 == 0);
}

int range[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *mid = partition(range, range + 10, isEven);
```

The result after `partition` is:

```
{ 2, 4, 6, 8, 10, 3, 7, 1, 9, 5 };
```

and `mid` points to the entry in the `range` array that contains 3.

What constraints does `partition` impose on its parameter types? Not too many; in fact the only constraints on `ForwardIterator` are imposed by `swap`, namely that whatever we get by dereferencing a `ForwardIterator` have a copy constructor and an assignment operator defined. What is more interesting is the need for the increment operator to work correctly on `ForwardIterators`, i.e. incrementing should take us to the next item in the range. Since we have used an array in our example, this works just fine.

However, if our iterator type had been set to address an element of a linked list, such as

```
struct cell{
   int value;
   cell *next;
}
```

and the range was specified by two pointers into a linked list, then `partition` would not do such a good job. In this case, we would need to overload the increment operator for pointers to `cell` structs. The increment operator would cause a pointer to take on the value of the `next` field of the structure it points to.

It turns out that `partition` is already a part of the C++ Standard Library. In addition, the SGI version of the STL provides a singly-linked list container class called `slist`. Associated with `slist` is an iterator type which meets the requirements of a `ForwardIterator`. So you could do something like:

```
slist<int> sl;
for(int i=0; i<10; ++i)
  sl.push_front(i);

slist<int>::iterator ip =
   partition(sl.begin(), sl.end(), isEven);
```

The STL groups sets of requirements on iterators into a hierarchy of iterator concepts. `ForwardIterator` is one such concept. `InputIterator` is a set of requirements which cover iterators on read-only containers (e.g. and incoming network stream). There is a similar concept called `OutputIterator`. The requirements which form `Forward Iterator` contains all the requirements in `InputIterator` and `OutputIterator`, and also adds a few requirements of its own. It is said that `ForwardIterator` is a *refinement* of `InputIterator` and `Output Iterator`. The refinement relationship between concepts is similar to an inheritance relationship between classes. Therefore, just as we can have a hierarchy of classes, we can have a hierarchy of concepts.

When a class satisfies the requirements of an iterator concept, then that class is said to be a *model* of that concept. For example, `slist<int>::iterator` is a model of `ForwardIterator`. If a class is a model of a concept which is a refinement of another concept, then that class is also a model of that other concept. For example, `slist<int>::iterator` is also a model of both `InputIterator` and `OutputIterator`.

The other two important iterator concepts are `BidirectionalIterator`, which is a refinement of `ForwardIterator`, and `RandomAccessIterator`, which is a refinement of `BidirectionalIterator`. Every pointer is a model of

RandomAccessIterator. An iterator associated with a doubly-linked list would be a model of BidirectionalIterator.

Templatized Algorithms Practice

a.) The templatized remove_copy_if algorithm copies elements from the range [first, last) to a range beginning at result, except that elements for which pred is true are not copied. The return value is the end of the resulting range. This operation is stable, meaning that the relative order of the copied elements is the same as in the range [first, last).

```
/*
 * Templatized Function: remove_copy_if
 * ------------------------------------
 * Copies elements from the range [first, last) to the range beginning
 * at result, except for those elements for which the specified
 * predicate function is true.  The return value is the end of
 * the resulting range.  Note that remove_copy_if is already a part of
 * the STL.
 */

inline template <class InputIterator, class OutputIterator,
                 class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result,
                              Predicate pred)
{
```

b.) List the assumptions about the InputIterator, OutputIterator, and the base class referenced by the Input/OutputIterator that must be made in order for this algorithm to compile when expanded.

c.) Write a function `removeNegativeFractions`, which takes an array of `Fractions` and filters out those that are negative. The original array should be used, and the effective size of the array after filtering should be returned. You should iterate over the array only once, using the `remove_copy_if` algorithm to do so. You will need to write a helper predicate function.

```
/*
 * Function: removeNegativeFractions
 * ---------------------------------
 * Examines the specified array of Fraction objects and removes
 * those that are negative while preserving the order of those
 * that remain.  The filtering is performed in place, and the
 * effective size of the filtered array is returned.
 */

int removeNegativeFractions(Fraction array[], int n)
{
```

Answers

a.)

```
/*
 * Templatized Function: remove_copy_if
 * ------------------------------------
 * Copies elements from the range [first, last) to the range beginning
 * at result, except for those elements for which the specified
 * predicate function is true.  The return value is the end of
 * the resulting range.
 */

inline template <class InputIterator, class OutputIterator,
                 class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result,
                              Predicate pred)
{
   while (first != last) {
      if (!pred(*first)) {
         *result = *first;
         ++result;
      }
      ++first;
   }

   return result;
}
```

b.)

Whatever true-type gets bound to the templatized type `InputIterator` must respond to `operator!=()`, `operator*()`, and `operator++()`. `OutputIterator` must respond to `operator*()` and `operator++()`. The type returned by `OutputIterator`'s `operator*()` must be assignable to whatever type is returned by `InputIterator`'s `operator*()`, and finally, `pred` may either be a function pointer or a function object that can be called to return something convertible to a `bool`. The type returned by `InputIterator`'s `operator*()` must be convertible to the argument of the predicate.

c.)

```
static inline bool belowZero(const Fraction& f)
{
   return f < 0;
}

int removeNegativeFractions(Fraction array[], int n)
{
   Fraction* end = remove_copy_if(array, array + n, array, belowZero);
   return end - array;
}
```

Implementing the `istreamiterator`

a.) Most predefined iterators iterate over the elements of a container—a `vector<string>`, a `list<int>`, a `map<string, Fraction>`, and so forth—but this is by no means required. Iterator concepts are very general, and they can abstract iteration through any ordered set of values, not just values that happen to be stored in a container. An example of this is the `istreamiterator`, which performs input using C++'s stream I/O library.

An `istreamiterator` is an `InputIterator` that extracts consecutive objects of type `T` from a particular `istream`. When the end of the stream is reached, the `istreamiterator` takes on a special end-of-stream value, which is the past-the-end iterator. In order for our `istreamiterator` to be compatible with templatized algorithms such as `for_each`, `accumulate`, `transform`, and so forth, the `istreamiterator` must support all the same operations normally required of the more traditional iterators: `operator++`, `operator*`, `operator!=`, and so forth.

Example:

```
static inline bool wordTooLong(const string& word)
{
   return word.size() > 6;
}

int main()
{
   ifstream myVocaularyList(“vocab.txt”);
   ofstream myEasierVocabularyList(“easyvocab.txt”);
   remove_copy_if(istreamiterator<string>(myVocubularyList),
                  istreamiterator<string>(),
                  ostreamiterator<string>(myEasierVocabularyList, “\n”),
                  wordTooLong);
}
```

The above program effectively duplicates the `vocab.txt` file, save the fact that all words of length six or more would be missing. (The "`\n`" is simply the `string` that should be printed in between actual `string` instances produced by the `istreamiterator`)

```
dragon
merciful
magnet
imp
behavior
compute
…
…
flatten
```

vocab.txt

```
dragon
magnet
imp
…
…
garage
```

easyvocab.txt

Based on an understanding on the above program and the `remove_copy_if` algorithm you wrote in section last time, it should be clear a minimal set of `istreamiterator` members is as follows:

```
istreamiterator<T>::istreamiterator(istream& s);
```

The constructor that creates an `istreamiterator` reading values from the input stream **`s`**. When **`s`** reaches end of stream, the iterator will compare equal to the end-of-stream iterator created using the default constructor.

```
istreamiterator<T>::istreamiterator();
```

The default constructor that creates an end-of-stream iterator, which is the past-the-end iterator needed to mark the end of a complete range.

```
const T& istreamiterator<T>::operator*() const;
```

Operator returning the next object in the `istream`.

```
const istreamiterator<T>& istreamiterator<T>::operator++();
const istreamiterator<T>  istreamiterator<T>::operator++(int);
```

Both versions of increment advancing the iterator to the next object in the `istream`.

```
bool operator==(const istreamiterator<T>& lhs,
                const istreamiterator<T>& rhs);
bool operator!=(const istreamiterator<T>& lhs,
                const istreamiterator<T>& rhs);
```

The equality and inequality operators.

Define and implement the `istreamiterator` template class to support these seven operations. Inline implementations within the class header whenever possible, but don't concern yourself with any compiler-specific workarounds.

Even More Generic Programming

Using your `remove_copy_if` algorithm from last week and your `istreamiterator` template, design and implement the `stripHTML` function, which takes a new `ifstream` referencing an HTML file, and a new `ofstream` referencing a presumably empty plain text file, and copies the HTML file content to the plain text file, but in doing so strips out all HTML tags except those in the specified `allowedTags` `set`.

Assume that the `set` class is templatized as follows:

```
template <class Key>
class set {
```

and that the only **`set`** class members of interest are:

```
/*
 * Function: begin, end
 * ---------------------
 * begin returns an iterator pointing to the beginning of the set.
 * end returns an iterator pointing just past-the-end of the set.
 */

   iterator begin() const;
   iterator end() const;

/*
 * Function: find
 * --------------
 * Returns an iterator pointing to the element matching the
 * specified key [in an operater==(const Key&, const Key&) sense],
 * or returns an iterator matching that returned by end() if
 * no such element can be found.
 */

   iterator find(const Key& k) const;
};
```

Assume tags stored within the `set` retain their delimiting **'<'** and **'>'**, and be sure to handle situations where tags appear side by side, as with

```
<b><i>Something in Bold Italics</i></b>
```

where `<b>` and `</b>` may be allowed but `<i>` and `</i>` may not be. You may not use any iteration constructs except that comprising `remove_copy_if`, you may traverse each `istream` one time and one time only, and you may not create any intermittent, temporary stream objects of any kind. *This is a difficult problem, so take your time.* My solution creates two very small helper classes: one is a subclass of `string` and another class overloads `operator()` to take one `const Key&` and return a `bool`.

```
/*
 * Function: stripHTML
 * -------------------
 * Copies the HTML text referenced by the specified instream
 * and copies it verbatim to the specified outstream, except that
 * tags not appearing the specified allowedTags set are removed.
 */

void stripHTML(ifstream& in, ofstream& out,
               const set<string>& allowedTags)
{
```

The Implementing the `istreamiterator` Part

```
template <class T>
class istreamiterator {

   public:
      istreamiterator() : in(NULL) {}
      istreamiterator(istream& instream) : in(&instream) { read() }

      const T& operator*() const { return value; }
      const istreamiterator& operator++() {
         read();
         return *this;
      }

      const istreamiterator operator++(int) {
         istreamiterator tmp(*this);
         read();
         return tmp;
      }

      friend bool operator!=(const istreamiterator<T>& rhs,
                             const istreamiterator<T>& lhs)
      { return rhs.in != lhs.in; }
      friend bool operator==(const istreamiterator<T>& rhs,
                             const istreamiterator<T>& lhs)
      { return rhs.in == lhs.in; }

   private:
      istream* in;
      T value;
      void read();
};

template <class T>
inline void istreamiterator<T>::read()
{
   if (in == 0) return;
   *in >> value;
   if (!*in) in = 0;
}
```

The Generic Programming Part

```
/*
 * Function: stripHTML
 * -------------------
 * Copies the HTML text referenced by the specified instream
 * and copies it verbatim to the specified outstream, except that
 * tags not appearing the specified allowedTags set are removed.
 */

void stripHTML(ifstream& in, ofstream& out,
               const set<string>& allowedTags)
{
   remove_copy_if(istream_iterator<HTMLToken>(in),
               istream_iterator<HTMLToken>(),
               ostream_iterator<string>(out, ""),
               FilterUnwantedTags(allowedTags));
}

class HTMLToken : public string {

   public:
      HTMLToken() {}
      HTMLToken(const char *str) : string(str) {}

      friend istream& operator>>(istream& in, HTMLToken& token) {
         if (in.peek() == EOF) return in;
         if (in.peek() == '<') {
            getline(in, token, '>');
            token += '>';
         } else {
            getline(in, token, '<');
         }
         return in;

      }
};

struct FilterUnwantedTags  {

   const set<string>& tagSet;
   FilterUnwantedTags(const set<string>& allowedTags) :
      tagSet(allowedTags) {}

   bool operator()(const HTMLToken& token) const
   {
      if (token.size() == 0) return false;   // will never happen, but...
      if (token[0] != '<') return false;  // not even a tail, so it's cool
      return tagSet.find(token) == tagSet.end();  // return true if the tag
   }                                              // isn't among those
}                                                 // allowed... remember true
                                                  // mean remove it
```