G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_rope.h** 完整列表

```
/*
 * Copyright (c) 1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_ROPE_H
# define __SGI_STL_INTERNAL_ROPE_H

# ifdef __GC
#   define __GC_CONST const
# else
#   define __GC_CONST   // constant except for deallocation
# endif
# ifdef __STL_SGI_THREADS
#    include <mutex.h>
# endif

__STL_BEGIN_NAMESPACE

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM !=
_MIPS_SIM_ABI32)
#pragma set woff 1174
#endif

// The end-of-C-string character.
// This is what the draft standard says it should be.
template <class charT>
inline charT __eos(charT*) { return charT(); }

// Test for basic character types.
// For basic character types leaves having a trailing eos.
template <class charT>
inline bool __is_basic_char_type(charT *) { return false; }
template <class charT>
inline bool __is_one_byte_char_type(charT *) { return false; }
```

```
inline bool __is_basic_char_type(char *) { return true; }
inline bool __is_one_byte_char_type(char *) { return true; }
inline bool __is_basic_char_type(wchar_t *) { return true; }

// Store an eos iff charT is a basic character type.
// Do not reference __eos if it isn't.
template <class charT>
inline void __cond_store_eos(charT&) {}

inline void __cond_store_eos(char& c) { c = 0; }
inline void __cond_store_eos(wchar_t& c) { c = 0; }


// rope<charT,Alloc> is a sequence of charT.
// Ropes appear to be mutable, but update operations
// really copy enough of the data structure to leave the original
// valid.  Thus ropes can be logically copied by just copying
// a pointer value.
// The __eos function is used for those functions that
// convert to/from C-like strings to detect the end of the string.
// __compare is used as the character comparison function.
template <class charT>
class char_producer {
   public:
   virtual ~char_producer() {};
   virtual void operator()(size_t start_pos, size_t len, charT* buffer)
        = 0;
   // Buffer should really be an arbitrary output iterator.
   // That way we could flatten directly into an ostream, etc.
   // This is thoroughly impossible, since iterator types don't
   // have runtime descriptions.
};

// Sequence buffers:
//
// Sequence must provide an append operation that appends an
// array to the sequence.  Sequence buffers are useful only if
// appending an entire array is cheaper than appending element by element.
// This is true for many string representations.
// This should  perhaps inherit from ostream<sequence::value_type>
// and be implemented correspondingly, so that they can be used
// for formatted.  For the sake of portability, we don't do this yet.
//
// For now, sequence buffers behave as output iterators.  But they also
// behave a little like basic_ostringstream<sequence::value_type> and a
// little like containers.

template<class sequence, size_t buf_sz = 100
#   if defined(__sgi) && !defined(__GNUC__)
```

```
#   define __TYPEDEF_WORKAROUND
        ,class v = typename sequence::value_type
#   endif
      >
// The 3rd parameter works around a common compiler bug.
class sequence_buffer : public output_iterator {
   public:
#      ifndef __TYPEDEF_WORKAROUND
       typedef typename sequence::value_type value_type;
#  else
       typedef v value_type;
#  endif
   protected:
   sequence *prefix;
   value_type buffer[buf_sz];
   size_t buf_count;
   public:
   void flush() {
       prefix->append(buffer, buffer + buf_count);
       buf_count = 0;
   }
   ~sequence_buffer() { flush(); }
   sequence_buffer() : prefix(0), buf_count(0) {}
   sequence_buffer(const sequence_buffer & x) {
       prefix = x.prefix;
           buf_count = x.buf_count;
           copy(x.buffer, x.buffer + x.buf_count, buffer);
   }
   sequence_buffer(sequence_buffer & x) {
       x.flush();
       prefix = x.prefix;
       buf_count = 0;
   }
   sequence_buffer(sequence& s) : prefix(&s), buf_count(0) {}
   sequence_buffer& operator= (sequence_buffer& x) {
       x.flush();
       prefix = x.prefix;
       buf_count = 0;
       return *this;
   }
   sequence_buffer& operator= (const sequence_buffer& x) {
       prefix = x.prefix;
       buf_count = x.buf_count;
       copy(x.buffer, x.buffer + x.buf_count, buffer);
       return *this;
   }
   void push_back(value_type x)
   {
       if (buf_count < buf_sz) {
```

```
        buffer[buf_count] = x;
        ++buf_count;
      } else {
        flush();
        buffer[0] = x;
        buf_count = 1;
      }
    }
    void append(value_type *s, size_t len)
    {
      if (len + buf_count <= buf_sz) {
        size_t i, j;
        for (i = buf_count, j = 0; j < len; i++, j++) {
          buffer[i] = s[j];
        }
        buf_count += len;
      } else if (0 == buf_count) {
        prefix->append(s, s + len);
      } else {
        flush();
        append(s, len);
      }
    }
    sequence_buffer& write(value_type *s, size_t len)
    {
      append(s, len);
      return *this;
    }
    sequence_buffer& put(value_type x)
    {
      push_back(x);
      return *this;
    }
    sequence_buffer& operator=(const value_type& rhs)
    {
      push_back(rhs);
      return *this;
    }
    sequence_buffer& operator*() { return *this; }
    sequence_buffer& operator++() { return *this; }
    sequence_buffer& operator++(int) { return *this; }
};

// The following should be treated as private, at least for now.
template<class charT>
class __rope_char_consumer {
    public:
    // If we had member templates, these should not be virtual.
    // For now we need to use run-time parametrization where
```

```
    // compile-time would do.  Hence this should all be private
    // for now.
    // The symmetry with char_producer is accidental and temporary.
    virtual ~__rope_char_consumer() {};
    virtual bool operator()(const charT* buffer, size_t len) = 0;
};

//
// What follows should really be local to rope.  Unfortunately,
// that doesn't work, since it makes it impossible to define generic
// equality on rope iterators.  According to the draft standard, the
// template parameters for such an equality operator cannot be inferred
// from the occurence of a member class as a parameter.
// (SGI compilers in fact allow this, but the result wouldn't be
// portable.)
// Similarly, some of the static member functions are member functions
// only to avoid polluting the global namespace, and to circumvent
// restrictions on type inference for template functions.
//

template<class CharT, class Alloc=__ALLOC> class rope;
template<class CharT, class Alloc> struct __rope_RopeConcatenation;
template<class CharT, class Alloc> struct __rope_RopeLeaf;
template<class CharT, class Alloc> struct __rope_RopeFunction;
template<class CharT, class Alloc> struct __rope_RopeSubstring;
template<class CharT, class Alloc> class __rope_iterator;
template<class CharT, class Alloc> class __rope_const_iterator;
template<class CharT, class Alloc> class __rope_charT_ref_proxy;
template<class CharT, class Alloc> class __rope_charT_ptr_proxy;

//
// The internal data structure for representing a rope.  This is
// private to the implementation.  A rope is really just a pointer
// to one of these.
//
// A few basic functions for manipulating this data structure
// are members of RopeBase.  Most of the more complex algorithms
// are implemented as rope members.
//
// Some of the static member functions of RopeBase have identically
// named functions in rope that simply invoke the RopeBase versions.
//

template<class charT, class Alloc>
struct __rope_RopeBase {
    typedef rope<charT,Alloc> my_rope;
    typedef simple_alloc<charT, Alloc> DataAlloc;
    typedef simple_alloc<__rope_RopeConcatenation<charT,Alloc>,
Alloc> CAlloc;
```

```
    typedef simple_alloc<__rope_RopeLeaf<charT,Alloc>, Alloc> LAlloc;
    typedef simple_alloc<__rope_RopeFunction<charT,Alloc>, Alloc> FAlloc;
    typedef simple_alloc<__rope_RopeSubstring<charT,Alloc>, Alloc> SAlloc;
    public:
    enum { max_rope_depth = 45 };
    enum {leaf, concat, substringfn, function} tag:8;
    bool is_balanced:8;
    unsigned char depth;
    size_t size;
    __GC_CONST charT * c_string;
             /* Flattened version of string, if needed.  */
             /* typically 0.                             */
             /* If it's not 0, then the memory is owned  */
             /* by this node.                            */
             /* In the case of a leaf, this may point to */
             /* the same memory as the data field.    */
#   ifndef __GC
#       if defined(__STL_WIN32THREADS)
       long refcount;      // InterlockedIncrement wants a long *
#   else
       size_t refcount;
#   endif
    // We count references from rope instances
    // and references from other rope nodes.  We
    // do not count const_iterator references.
    // Iterator references are counted so that rope modifications
    // can be detected after the fact.
    // Generally function results are counted, i.e.
    // a pointer returned by a function is included at the
    // point at which the pointer is returned.
    // The recipient should decrement the count if the
    // result is not needed.
    // Generally function arguments are not reflected
    // in the reference count.  The callee should increment
    // the count before saving the argument someplace that
    // will outlive the call.
#   endif
#   ifndef __GC
#       ifdef __STL_SGI_THREADS
       // Reference counting with multiple threads and no
       // hardware or thread package support is pretty awful.
       // Mutexes are normally too expensive.
       // We'll assume a COMPARE_AND_SWAP(destp, old, new)
       // operation, which might be cheaper.
#          if __mips < 3 || !(defined (_ABIN32) || defined(_ABI64))
#              define __add_and_fetch(l,v) add_then_test((unsigned long *)l,v)
#          endif
       void init_refcount_lock() {}
       void incr_refcount ()
```

```
        {
          __add_and_fetch(&refcount, 1);
        }
        size_t decr_refcount ()
        {
          return __add_and_fetch(&refcount, (size_t)(-1));
        }
#       elif defined(__STL_WIN32THREADS)
        void init_refcount_lock() {}
            void incr_refcount ()
            {
                InterlockedIncrement(&refcount);
            }
            size_t decr_refcount ()
            {
                return InterlockedDecrement(&refcount);
            }
#   elif defined(__STL_PTHREADS)
        // This should be portable, but performance is expected
        // to be quite awful.  This really needs platform specific
        // code.
        pthread_mutex_t refcount_lock;
        void init_refcount_lock() {
          pthread_mutex_init(&refcount_lock, 0);
        }
        void incr_refcount ()
            {
          pthread_mutex_lock(&refcount_lock);
                ++refcount;
          pthread_mutex_unlock(&refcount_lock);
            }
            size_t decr_refcount ()
            {
          size_t result;
          pthread_mutex_lock(&refcount_lock);
                result = --refcount;
          pthread_mutex_unlock(&refcount_lock);
                return result;
            }
#   else
        void init_refcount_lock() {}
        void incr_refcount ()
        {
          ++refcount;
        }
        size_t decr_refcount ()
        {
          --refcount;
          return refcount;
```

```
      }
#     endif
#   else
    void incr_refcount () {}
#   endif
    static void free_string(charT *, size_t len);
             // Deallocate data section of a leaf.
             // This shouldn't be a member function.
             // But its hard to do anything else at the
             // moment, because it's templatized w.r.t.
             // an allocator.
             // Does nothing if __GC is defined.
#   ifndef __GC
     void free_c_string();
     void free_tree();
             // Deallocate t. Assumes t is not 0.
     void unref_nonnil()
     {
        if (0 == decr_refcount()) free_tree();
     }
     void ref_nonnil()
     {
        incr_refcount();
     }
     static void unref(__rope_RopeBase* t)
     {
        if (0 != t) {
          t -> unref_nonnil();
        }
     }
     static void ref(__rope_RopeBase* t)
     {
        if (0 != t) t -> incr_refcount();
     }
     static void free_if_unref(__rope_RopeBase* t)
     {
        if (0 != t && 0 == t -> refcount) t -> free_tree();
     }
#   else /* __GC */
     void unref_nonnil() {}
     void ref_nonnil() {}
     static void unref(__rope_RopeBase* t) {}
     static void ref(__rope_RopeBase* t) {}
     static void fn_finalization_proc(void * tree, void *);
     static void free_if_unref(__rope_RopeBase* t) {}
#   endif

    // The data fields of leaves are allocated with some
    // extra space, to accomodate future growth and for basic
```

```
    // character types, to hold a trailing eos character.
    enum { alloc_granularity = 8 };
    static size_t rounded_up_size(size_t n) {
        size_t size_with_eos;

        if (__is_basic_char_type((charT *)0)) {
             size_with_eos = n + 1;
         } else {
            size_with_eos = n;
    }
#       ifdef __GC
          return size_with_eos;
#   else
          // Allow slop for in-place expansion.
          return (size_with_eos + alloc_granularity-1)
                 &~ (alloc_granularity-1);
#   endif
    }
};

template<class charT, class Alloc>
struct __rope_RopeLeaf : public __rope_RopeBase<charT,Alloc> {
  public:  // Apparently needed by VC++
    __GC_CONST charT* data;    /* Not necessarily 0 terminated. */
                  /* The allocated size is     */
                  /* rounded_up_size(size), except */
                  /* in the GC case, in which it    */
                  /* doesn't matter.       */
};

template<class charT, class Alloc>
struct __rope_RopeConcatenation : public
__rope_RopeBase<charT,Alloc> {
  public:
    __rope_RopeBase<charT,Alloc>* left;
    __rope_RopeBase<charT,Alloc>* right;
};

template<class charT, class Alloc>
struct __rope_RopeFunction : public __rope_RopeBase<charT,Alloc> {
  public:
    char_producer<charT>* fn;
#   ifndef __GC
      bool delete_when_done;    // Char_producer is owned by the
                  // rope and should be explicitly
                  // deleted when the rope becomes
                  // inaccessible.
#   else
      // In the GC case, we either register the rope for
```

```
     // finalization, or not.  Thus the field is unnecessary;
     // the information is stored in the collector data structures.
#   endif
};
// Substring results are usually represented using just
// concatenation nodes.  But in the case of very long flat ropes
// or ropes with a functional representation that isn't practical.
// In that case, we represent the result as a special case of
// RopeFunction, whose char_producer points back to the rope itself.
// In all cases except repeated substring operations and
// deallocation, we treat the result as a RopeFunction.
template<class charT, class Alloc>
struct __rope_RopeSubstring: public
__rope_RopeFunction<charT,Alloc>,
                public char_producer<charT> {
  public:
    __rope_RopeBase<charT,Alloc> * base;  // not 0
    size_t start;
    virtual ~__rope_RopeSubstring() {}
    virtual void operator()(size_t start_pos, size_t req_len,
                charT *buffer) {
    switch(base -> tag) {
       case function:
       case substringfn:
         {
         char_producer<charT> *fn =
              ((__rope_RopeFunction<charT,Alloc> *)base) -> fn;
         __stl_assert(start_pos + req_len <= size);
         __stl_assert(start + size <= base -> size);
         (*fn)(start_pos + start, req_len, buffer);
         }
         break;
       case leaf:
         {
         __GC_CONST charT * s =
              ((__rope_RopeLeaf<charT,Alloc> *)base) -> data;
         uninitialized_copy_n(s + start_pos + start, req_len,
                      buffer);
         }
         break;
       default:
         __stl_assert(false);
    }
    }
    __rope_RopeSubstring(__rope_RopeBase<charT,Alloc> * b, size_t s,
size_t l) :
    base(b), start(s) {
#      ifndef __GC
       refcount = 1;
```

```
        init_refcount_lock();
        base -> ref_nonnil();
#       endif
    size = l;
    tag = substringfn;
    depth = 0;
    c_string = 0;
    fn = this;
    }
};


// Self-destructing pointers to RopeBase.
// These are not conventional smart pointers.  Their
// only purpose in life is to ensure that unref is called
// on the pointer either at normal exit or if an exception
// is raised.  It is the caller's responsibility to
// adjust reference counts when these pointers are initialized
// or assigned to.  (This convention significantly reduces
// the number of potentially expensive reference count
// updates.)
#ifndef __GC
  template<class charT, class Alloc>
  struct __rope_self_destruct_ptr {
    __rope_RopeBase<charT,Alloc> * ptr;
    ~__rope_self_destruct_ptr()
{ __rope_RopeBase<charT,Alloc>::unref(ptr); }
#   ifdef __STL_USE_EXCEPTIONS
    __rope_self_destruct_ptr() : ptr(0) {};
#   else
    __rope_self_destruct_ptr() {};
#   endif
    __rope_self_destruct_ptr(__rope_RopeBase<charT,Alloc> * p) :
ptr(p) {}
    __rope_RopeBase<charT,Alloc> & operator*() { return *ptr; }
    __rope_RopeBase<charT,Alloc> * operator->() { return ptr; }
    operator __rope_RopeBase<charT,Alloc> *() { return ptr; }
    __rope_self_destruct_ptr & operator=
(__rope_RopeBase<charT,Alloc> * x)
    { ptr = x; return *this; }
  };
#endif

// Dereferencing a nonconst iterator has to return something
// that behaves almost like a reference.  It's not possible to
// return an actual reference since assignment requires extra
// work.  And we would get into the same problems as with the
// CD2 version of basic_string.
template<class charT, class Alloc>
```

```
class __rope_charT_ref_proxy {
   friend class rope<charT,Alloc>;
   friend class __rope_iterator<charT,Alloc>;
   friend class __rope_charT_ptr_proxy<charT,Alloc>;
#  ifdef __GC
   typedef __rope_RopeBase<charT,Alloc> * self_destruct_ptr;
#  else
       typedef __rope_self_destruct_ptr<charT,Alloc>
self_destruct_ptr;
#  endif
   typedef __rope_RopeBase<charT,Alloc> RopeBase;
   typedef rope<charT,Alloc> my_rope;
   size_t pos;
   charT current;
   bool current_valid;
   my_rope * root;    // The whole rope.
 public:
   __rope_charT_ref_proxy(my_rope * r, size_t p) :
   pos(p), root(r), current_valid(false) {}
   __rope_charT_ref_proxy(my_rope * r, size_t p,
          charT c) :
   pos(p), root(r), current(c), current_valid(true) {}
   operator charT () const;
   __rope_charT_ref_proxy& operator= (charT c);
   __rope_charT_ptr_proxy<charT,Alloc> operator& () const;
   __rope_charT_ref_proxy& operator= (const
__rope_charT_ref_proxy& c) {
   return operator=((charT)c);
   }
};

template<class charT, class Alloc>
class __rope_charT_ptr_proxy {
   friend class __rope_charT_ref_proxy<charT,Alloc>;
   size_t pos;
   charT current;
   bool current_valid;
   rope<charT,Alloc> * root;    // The whole rope.
 public:
   __rope_charT_ptr_proxy(const
__rope_charT_ref_proxy<charT,Alloc> & x) :
   pos(x.pos), root(x.root), current_valid(x.current_valid),
   current(x.current) {}
   __rope_charT_ptr_proxy(const __rope_charT_ptr_proxy & x) :
   pos(x.pos), root(x.root), current_valid(x.current_valid),
   current(x.current) {}
   __rope_charT_ptr_proxy() {}
   __rope_charT_ptr_proxy(charT * x) : root(0), pos(0) {
   __stl_assert(0 == x);
```

```
    }
    __rope_charT_ptr_proxy& operator= (const
__rope_charT_ptr_proxy& x) {
    pos = x.pos;
    current = x.current;
    current_valid = x.current_valid;
    root = x.root;
    return *this;
    }
    friend bool operator== __STL_NULL_TMPL_ARGS
              (const __rope_charT_ptr_proxy<charT,Alloc> & x,
               const __rope_charT_ptr_proxy<charT,Alloc> & y);
    __rope_charT_ref_proxy<charT,Alloc> operator *() const {
    if (current_valid) {
       return __rope_charT_ref_proxy<charT,Alloc>(root, pos,
current);
    } else {
       return __rope_charT_ref_proxy<charT,Alloc>(root, pos);
    }
    }
};

// Rope iterators:
// Unlike in the C version, we cache only part of the stack
// for rope iterators, since they must be efficiently copyable.
// When we run out of cache, we have to reconstruct the iterator
// value.
// Pointers from iterators are not included in reference counts.
// Iterators are assumed to be thread private.  Ropes can
// be shared.

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM !=
_MIPS_SIM_ABI32)
#pragma set woff 1375
#endif

template<class charT, class Alloc>
class __rope_iterator_base:
  public random_access_iterator<charT, ptrdiff_t> {
  friend class rope<charT, Alloc>;
  public:
    typedef __rope_RopeBase<charT,Alloc> RopeBase;
    // Borland doesnt want this to be protected.
  protected:
    enum { path_cache_len = 4 }; // Must be <= 9.
    enum { iterator_buf_len = 15 };
    size_t current_pos;
    RopeBase * root;     // The whole rope.
    size_t leaf_pos;   // Starting position for current leaf
```

```
__GC_CONST charT * buf_start;
         // Buffer possibly
         // containing current char.
__GC_CONST charT * buf_ptr;
         // Pointer to current char in buffer.
         // != 0 ==> buffer valid.
__GC_CONST charT * buf_end;
         // One past last valid char in buffer.
// What follows is the path cache.  We go out of our
// way to make this compact.
// Path_end contains the bottom section of the path from
// the root to the current leaf.
const RopeBase * path_end[path_cache_len];
int leaf_index;    // Last valid pos in path_end;
              // path_end[0] ... path_end[leaf_index-1]
         // point to concatenation nodes.
unsigned char path_directions;
           // (path_directions >> i) & 1 is 1
           // iff we got from path_end[leaf_index - i - 1]
           // to path_end[leaf_index - i] by going to the
           // right. Assumes path_cache_len <= 9.
charT tmp_buf[iterator_buf_len];
         // Short buffer for surrounding chars.
         // This is useful primarily for
         // RopeFunctions.  We put the buffer
         // here to avoid locking in the
         // multithreaded case.
// The cached path is generally assumed to be valid
// only if the buffer is valid.
static void setbuf(__rope_iterator_base &x);
                  // Set buffer contents given
                  // path cache.
static void setcache(__rope_iterator_base &x);
                  // Set buffer contents and
                  // path cache.
static void setcache_for_incr(__rope_iterator_base &x);
                  // As above, but assumes path
                  // cache is valid for previous posn.
__rope_iterator_base() {}
__rope_iterator_base(RopeBase * root, size_t pos):
       root(root), current_pos(pos), buf_ptr(0) {}
__rope_iterator_base(const __rope_iterator_base& x) {
if (0 != x.buf_ptr) {
   *this = x;
} else {
   current_pos = x.current_pos;
   root = x.root;
   buf_ptr = 0;
}
```

```
    }
    void incr(size_t n);
    void decr(size_t n);
  public:
    size_t index() const { return current_pos; }
};

template<class charT, class Alloc> class __rope_iterator;

template<class charT, class Alloc>
class __rope_const_iterator : public
__rope_iterator_base<charT,Alloc> {
    friend class rope<charT,Alloc>;
  protected:
    __rope_const_iterator(const RopeBase * root, size_t pos):
           __rope_iterator_base<charT,Alloc>(
             const_cast<RopeBase *>(root), pos)
           // Only nonconst iterators modify root ref count
    {}
  public:
    typedef charT reference;   // Really a value.  Returning a
reference
                  // Would be a mess, since it would have
                  // to be included in refcount.
    typedef const charT* pointer;

  public:
    __rope_const_iterator() {};
    __rope_const_iterator(const __rope_const_iterator & x) :
                  __rope_iterator_base<charT,Alloc>(x) { }
    __rope_const_iterator(const __rope_iterator<charT,Alloc> & x);
    __rope_const_iterator(const rope<charT,Alloc> &r, size_t pos) :
    __rope_iterator_base<charT,Alloc>(r.tree_ptr, pos) {}
    __rope_const_iterator& operator= (const __rope_const_iterator &
x) {
    if (0 != x.buf_ptr) {
        *this = x;
    } else {
        current_pos = x.current_pos;
        root = x.root;
        buf_ptr = 0;
    }
    return(*this);
    }
    reference operator*() {
    if (0 == buf_ptr) setcache(*this);
    return *buf_ptr;
    }
    __rope_const_iterator& operator++() {
```

```
    __GC_CONST charT * next;
    if (0 != buf_ptr && (next = buf_ptr + 1) < buf_end) {
        buf_ptr = next;
        ++current_pos;
    } else {
        incr(1);
    }
    return *this;
    }
    __rope_const_iterator& operator+=(ptrdiff_t n) {
    if (n >= 0) {
        incr(n);
    } else {
        decr(-n);
    }
    return *this;
    }
    __rope_const_iterator& operator--() {
    decr(1);
    return *this;
    }
    __rope_const_iterator& operator-=(ptrdiff_t n) {
    if (n >= 0) {
        decr(n);
    } else {
        incr(-n);
    }
    return *this;
    }
    __rope_const_iterator operator++(int) {
    size_t old_pos = current_pos;
    incr(1);
    return __rope_const_iterator<charT,Alloc>(root, old_pos);
    // This makes a subsequent dereference expensive.
    // Perhaps we should instead copy the iterator
    // if it has a valid cache?
    }
    __rope_const_iterator operator--(int) {
    size_t old_pos = current_pos;
    decr(1);
    return __rope_const_iterator<charT,Alloc>(root, old_pos);
    }
    friend __rope_const_iterator<charT,Alloc> operator-
__STL_NULL_TMPL_ARGS
    (const __rope_const_iterator<charT,Alloc> & x,
     ptrdiff_t n);
    friend __rope_const_iterator<charT,Alloc> operator+
__STL_NULL_TMPL_ARGS
    (const __rope_const_iterator<charT,Alloc> & x,
```

```
     ptrdiff_t n);
    friend __rope_const_iterator<charT,Alloc> operator+
__STL_NULL_TMPL_ARGS
    (ptrdiff_t n,
     const __rope_const_iterator<charT,Alloc> & x);
    reference operator[](size_t n) {
    return rope<charT,Alloc>::fetch(root, current_pos + n);
    }
    friend bool operator== __STL_NULL_TMPL_ARGS
    (const __rope_const_iterator<charT,Alloc> & x,
     const __rope_const_iterator<charT,Alloc> & y);
    friend bool operator< __STL_NULL_TMPL_ARGS
    (const __rope_const_iterator<charT,Alloc> & x,
     const __rope_const_iterator<charT,Alloc> & y);
    friend ptrdiff_t operator- __STL_NULL_TMPL_ARGS
    (const __rope_const_iterator<charT,Alloc> & x,
     const __rope_const_iterator<charT,Alloc> & y);
};

template<class charT, class Alloc>
class __rope_iterator : public __rope_iterator_base<charT,Alloc> {
    friend class rope<charT,Alloc>;
  protected:
    rope<charT,Alloc> * root_rope;
    // root is treated as a cached version of this,
    // and is used to detect changes to the underlying
    // rope.
    // Root is included in the reference count.
    // This is necessary so that we can detect changes reliably.
    // Unfortunately, it requires careful bookkeeping for the
    // nonGC case.
    __rope_iterator(rope<charT,Alloc> * r, size_t pos):
        __rope_iterator_base<charT,Alloc>(r -> tree_ptr, pos),
        root_rope(r) {
         RopeBase::ref(root);
        }
    void check();
  public:
    typedef __rope_charT_ref_proxy<charT,Alloc>  reference;
    typedef __rope_charT_ref_proxy<charT,Alloc>* pointer;

  public:
    rope<charT,Alloc>& container() { return *root_rope; }
    __rope_iterator() {
    root = 0;  // Needed for reference counting.
    };
    __rope_iterator(const __rope_iterator & x) :
    __rope_iterator_base<charT,Alloc>(x) {
    root_rope = x.root_rope;
```

```
    RopeBase::ref(root);
    }
    __rope_iterator(rope<charT,Alloc>& r, size_t pos);
    ~__rope_iterator() {
    RopeBase::unref(root);
    }
    __rope_iterator& operator= (const __rope_iterator & x) {
    RopeBase *old = root;

    RopeBase::ref(x.root);
    if (0 != x.buf_ptr) {
        *this = x;
    } else {
        current_pos = x.current_pos;
        root = x.root;
        root_rope = x.root_rope;
        buf_ptr = 0;
    }
    RopeBase::unref(old);
    return(*this);
    }
    reference operator*() {
    check();
    if (0 == buf_ptr) {
        return __rope_charT_ref_proxy<charT,Alloc>(root_rope,
current_pos);
    } else {
        return __rope_charT_ref_proxy<charT,Alloc>(root_rope,
                                  current_pos, *buf_ptr);
    }
    }
    __rope_iterator& operator++() {
    incr(1);
    return *this;
    }
    __rope_iterator& operator+=(difference_type n) {
    if (n >= 0) {
        incr(n);
    } else {
        decr(-n);
    }
    return *this;
    }
    __rope_iterator& operator--() {
    decr(1);
    return *this;
    }
    __rope_iterator& operator-=(difference_type n) {
    if (n >= 0) {
```

```
        decr(n);
    } else {
        incr(-n);
    }
    return *this;
    }
    __rope_iterator operator++(int) {
    size_t old_pos = current_pos;
    incr(1);
    return __rope_iterator<charT,Alloc>(root_rope, old_pos);
    }
    __rope_iterator operator--(int) {
    size_t old_pos = current_pos;
    decr(1);
    return __rope_iterator<charT,Alloc>(root_rope, old_pos);
    }
    reference operator[](ptrdiff_t n) {
    return __rope_charT_ref_proxy<charT,Alloc>(root_rope,
current_pos + n);
    }
    friend bool operator== __STL_NULL_TMPL_ARGS
    (const __rope_iterator<charT,Alloc> & x,
     const __rope_iterator<charT,Alloc> & y);
    friend bool operator< __STL_NULL_TMPL_ARGS
    (const __rope_iterator<charT,Alloc> & x,
     const __rope_iterator<charT,Alloc> & y);
    friend ptrdiff_t operator- __STL_NULL_TMPL_ARGS
    (const __rope_iterator<charT,Alloc> & x,
     const __rope_iterator<charT,Alloc> & y);
    friend __rope_iterator<charT,Alloc> operator-
__STL_NULL_TMPL_ARGS
    (const __rope_iterator<charT,Alloc> & x,
     ptrdiff_t n);
    friend __rope_iterator<charT,Alloc> operator+
__STL_NULL_TMPL_ARGS
    (const __rope_iterator<charT,Alloc> & x,
     ptrdiff_t n);
    friend __rope_iterator<charT,Alloc> operator+
__STL_NULL_TMPL_ARGS
    (ptrdiff_t n,
     const __rope_iterator<charT,Alloc> & x);

};

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM !=
_MIPS_SIM_ABI32)
#pragma reset woff 1375
#endif
```

```
template <class charT, class Alloc>
class rope {
   public:
   typedef charT value_type;
   typedef ptrdiff_t difference_type;
   typedef size_t size_type;
   typedef charT const_reference;
   typedef const charT* const_pointer;
   typedef __rope_iterator<charT,Alloc> iterator;
   typedef __rope_const_iterator<charT,Alloc> const_iterator;
   typedef __rope_charT_ref_proxy<charT,Alloc> reference;
   typedef __rope_charT_ptr_proxy<charT,Alloc> pointer;

   friend class __rope_iterator<charT,Alloc>;
   friend class __rope_const_iterator<charT,Alloc>;
   friend struct __rope_RopeBase<charT,Alloc>;
   friend class __rope_iterator_base<charT,Alloc>;
   friend class __rope_charT_ptr_proxy<charT,Alloc>;
   friend class __rope_charT_ref_proxy<charT,Alloc>;
   friend struct __rope_RopeSubstring<charT,Alloc>;

   protected:
   typedef __GC_CONST charT * cstrptr;
#     ifdef __STL_SGI_THREADS
      static cstrptr atomic_swap(cstrptr *p, cstrptr q) {
#            if __mips < 3 || !(defined (_ABIN32) || defined(_ABI64))
               return (cstrptr) test_and_set((unsigned long *)p,
                                 (unsigned long)q);
#      else
               return(cstrptr) __test_and_set((unsigned long *)p,
                                (unsigned long)q);
#      endif
         }
#     elif defined(__STL_WIN32THREADS)
      static cstrptr atomic_swap(cstrptr *p, cstrptr q) {
       return (cstrptr) InterlockedExchange((LPLONG)p, (LONG)q);
      }
#  elif defined(__STL_PTHREADS)
      // This should be portable, but performance is expected
      // to be quite awful.  This really needs platform specific
      // code.
      static pthread_mutex_t swap_lock;
      static cstrptr atomic_swap(cstrptr *p, cstrptr q) {
       pthread_mutex_lock(&swap_lock);
       cstrptr result = *p;
       *p = q;
       pthread_mutex_unlock(&swap_lock);
       return result;
         }
```

```
#  else
       static cstrptr atomic_swap(cstrptr *p, cstrptr q) {
              cstrptr result = *p;
              *p = q;
        return result;
       }
#      endif

   static charT empty_c_str[1];

         typedef simple_alloc<charT, Alloc> DataAlloc;
         typedef
simple_alloc<__rope_RopeConcatenation<charT,Alloc>, Alloc> CAlloc;
         typedef simple_alloc<__rope_RopeLeaf<charT,Alloc>, Alloc>
LAlloc;
         typedef simple_alloc<__rope_RopeFunction<charT,Alloc>,
Alloc> FAlloc;
         typedef simple_alloc<__rope_RopeSubstring<charT,Alloc>,
Alloc> SAlloc;
   static bool is0(charT c) { return c == __eos((charT *)0); }
   enum { copy_max = 23 };
        // For strings shorter than copy_max, we copy to
        // concatenate.

   typedef __rope_RopeBase<charT,Alloc> RopeBase;
   typedef __rope_RopeConcatenation<charT,Alloc>
RopeConcatenation;
   typedef __rope_RopeLeaf<charT,Alloc> RopeLeaf;
   typedef __rope_RopeFunction<charT,Alloc> RopeFunction;
   typedef __rope_RopeSubstring<charT,Alloc> RopeSubstring;

   // The only data member of a rope:
   RopeBase *tree_ptr;

   // Retrieve a character at the indicated position.
   static charT fetch(RopeBase * r, size_type pos);

#  ifndef __GC
       // Obtain a pointer to the character at the indicated position.
       // The pointer can be used to change the character.
       // If such a pointer cannot be produced, as is frequently the
       // case, 0 is returned instead.
       // (Returns nonzero only if all nodes in the path have a refcount
       // of 1.)
       static charT * fetch_ptr(RopeBase * r, size_type pos);
#  endif

   static bool apply_to_pieces(
                // should be template parameter
```

```
                __rope_char_consumer<charT>& c,
                const RopeBase * r,
                size_t begin, size_t end);
                // begin and end are assumed to be in range.

#  ifndef __GC
    static void unref(RopeBase* t)
    {
       RopeBase::unref(t);
    }
    static void ref(RopeBase* t)
    {
       RopeBase::ref(t);
    }
#     else /* __GC */
    static void unref(RopeBase* t) {}
    static void ref(RopeBase* t) {}
#     endif


#     ifdef __GC
      typedef __rope_RopeBase<charT,Alloc> * self_destruct_ptr;
#      else
      typedef __rope_self_destruct_ptr<charT,Alloc>
self_destruct_ptr;
#  endif

   // Result is counted in refcount.
   static RopeBase * substring(RopeBase * base,
                   size_t start, size_t endp1);

   static RopeBase * concat_char_iter(RopeBase * r,
                      const charT *iter, size_t slen);
       // Concatenate rope and char ptr, copying s.
       // Should really take an arbitrary iterator.
       // Result is counted in refcount.
   static RopeBase * destr_concat_char_iter(RopeBase * r,
                           const charT *iter, size_t slen)
       // As above, but one reference to r is about to be
       // destroyed.  Thus the pieces may be recycled if all
       // relevent reference counts are 1.
#     ifdef __GC
        // We can't really do anything since refcounts are unavailable.
        { return concat_char_iter(r, iter, slen); }
#     else
        ;
#     endif

   static RopeBase * concat(RopeBase *left, RopeBase *right);
```

```
        // General concatenation on RopeBase.  Result
        // has refcount of 1.  Adjusts argument refcounts.

    public:
     void apply_to_pieces( size_t begin, size_t end,
                   __rope_char_consumer<charT>& c) const {
        apply_to_pieces(c, tree_ptr, begin, end);
     }


    protected:

     static size_t rounded_up_size(size_t n) {
        return RopeBase::rounded_up_size(n);
     }

     static size_t allocated_capacity(size_t n) {
        if (__is_basic_char_type((charT *)0)) {
          return rounded_up_size(n) - 1;
        } else {
          return rounded_up_size(n);
        }
     }

     // s should really be an arbitrary input iterator.
     // Adds a trailing NULL for basic char types.
     static charT * alloc_copy(const charT *s, size_t size)
     {
        charT * result = DataAlloc::allocate(rounded_up_size(size));

        uninitialized_copy_n(s, size, result);
        __cond_store_eos(result[size]);
        return(result);
     }

     // Basic constructors for rope tree nodes.
     // These return tree nodes with a 0 reference count.
     static RopeLeaf * RopeLeaf_from_char_ptr(__GC_CONST charT *s,
                              size_t size);
        // Takes ownership of its argument.
        // Result has refcount 1.
        // In the nonGC, basic_char_type  case it assumes that s
        // is eos-terminated.
        // In the nonGC case, it was allocated from Alloc with
        // rounded_up_size(size).

     static RopeLeaf * RopeLeaf_from_unowned_char_ptr(const charT *s,
                                    size_t size) {
        charT * buf = alloc_copy(s, size);
```

```
        __STL_TRY {
          return RopeLeaf_from_char_ptr(buf, size);
        }
        __STL_UNWIND(RopeBase::free_string(buf, size))
    }


    // Concatenation of nonempty strings.
    // Always builds a concatenation node.
    // Rebalances if the result is too deep.
    // Result has refcount 1.
    // Does not increment left and right ref counts even though
    // they are referenced.
    static RopeBase * tree_concat(RopeBase * left, RopeBase * right);

    // Result has refcount 1.
    // If delete_fn is true, then fn is deleted when the rope
    // becomes inaccessible.
    static RopeFunction * RopeFunction_from_fn
             (char_producer<charT> *fn, size_t size,
              bool delete_fn);

    // Concatenation helper functions
    static RopeLeaf * leaf_concat_char_iter
              (RopeLeaf * r, const charT * iter, size_t slen);
         // Concatenate by copying leaf.
         // should take an arbitrary iterator
         // result has refcount 1.
#   ifndef __GC
      static RopeLeaf * destr_leaf_concat_char_iter
              (RopeLeaf * r, const charT * iter, size_t slen);
      // A version that potentially clobbers r if r -> refcount == 1.
#       endif

    // A helper function for exponentiating strings.
    // This uses a nonstandard refcount convention.
    // The result has refcount 0.
    struct concat_fn;
    friend struct rope<charT,Alloc>::concat_fn;

    struct concat_fn
         : public binary_function<rope<charT,Alloc>, rope<charT,Alloc>,
                                  rope<charT,Alloc> > {
         rope operator() (const rope& x, const rope& y) {
            return x + y;
         }
    };

        friend rope identity_element(concat_fn) { return rope<charT,Alloc>(); }
```

```
    static size_t char_ptr_len(const charT * s);
             // slightly generalized strlen

    rope(RopeBase *t) : tree_ptr(t) { }


    // Copy r to the CharT buffer.
    // Returns buffer + r -> size.
    // Assumes that buffer is uninitialized.
    static charT * flatten(RopeBase * r, charT * buffer);

    // Again, with explicit starting position and length.
    // Assumes that buffer is uninitialized.
    static charT * flatten(RopeBase * r,
                   size_t start, size_t len,
                   charT * buffer);

    static const unsigned long min_len[RopeBase::max_rope_depth + 1];

    static bool is_balanced(RopeBase *r)
         { return (r -> size >= min_len[r -> depth]); }

    static bool is_almost_balanced(RopeBase *r)
         { return (r -> depth == 0 ||
                r -> size >= min_len[r -> depth - 1]); }

    static bool is_roughly_balanced(RopeBase *r)
         { return (r -> depth <= 1 ||
                r -> size >= min_len[r -> depth - 2]); }

    // Assumes the result is not empty.
    static RopeBase * concat_and_set_balanced(RopeBase *left,
                             RopeBase *right)
    {
       RopeBase * result = concat(left, right);
       if (is_balanced(result)) result -> is_balanced = true;
       return result;
    }

    // The basic rebalancing operation.  Logically copies the
    // rope.  The result has refcount of 1.  The client will
    // usually decrement the reference count of r.
    // The result isd within height 2 of balanced by the above
    // definition.
    static RopeBase * balance(RopeBase * r);

    // Add all unbalanced subtrees to the forest of balanceed trees.
    // Used only by balance.
```

```
    static void add_to_forest(RopeBase *r, RopeBase **forest);

    // Add r to forest, assuming r is already balanced.
    static void add_leaf_to_forest(RopeBase *r, RopeBase **forest);

    // Print to stdout, exposing structure
    static void dump(RopeBase * r, int indent = 0);

    // Return -1, 0, or 1 if x < y, x == y, or x > y resp.
    static int compare(const RopeBase *x, const RopeBase *y);

   public:
    bool empty() const { return 0 == tree_ptr; }

    // Comparison member function.  This is public only for those
    // clients that need a ternary comparison.  Others
    // should use the comparison operators below.
    int compare(const rope &y) const {
        return compare(tree_ptr, y.tree_ptr);
    }

    rope(const charT *s)
    {
        size_t len = char_ptr_len(s);

        if (0 == len) {
          tree_ptr = 0;
        } else {
          tree_ptr = RopeLeaf_from_unowned_char_ptr(s, len);
#         ifndef __GC
            __stl_assert(1 == tree_ptr -> refcount);
#         endif
        }
    }

    rope(const charT *s, size_t len)
    {
        if (0 == len) {
          tree_ptr = 0;
        } else {
          tree_ptr = RopeLeaf_from_unowned_char_ptr(s, len);
        }
    }

    rope(const charT *s, charT *e)
    {
        size_t len = e - s;

        if (0 == len) {
```

```
        tree_ptr = 0;
      } else {
        tree_ptr = RopeLeaf_from_unowned_char_ptr(s, len);
      }
    }

    rope(const const_iterator& s, const const_iterator& e)
    {
      tree_ptr = substring(s.root, s.current_pos, e.current_pos);
    }

    rope(const iterator& s, const iterator& e)
    {
      tree_ptr = substring(s.root, s.current_pos, e.current_pos);
    }

    rope(charT c)
    {
      charT * buf = DataAlloc::allocate(rounded_up_size(1));

      construct(buf, c);
      __STL_TRY {
          tree_ptr = RopeLeaf_from_char_ptr(buf, 1);
          }
          __STL_UNWIND(RopeBase::free_string(buf, 1))
    }

    rope(size_t n, charT c);

    // Should really be templatized with respect to the iterator type
    // and use sequence_buffer.  (It should perhaps use sequence_buffer
    // even now.)
    rope(const charT *i, const charT *j)
    {
      if (i == j) {
        tree_ptr = 0;
      } else {
        size_t len = j - i;
        tree_ptr = RopeLeaf_from_unowned_char_ptr(i, len);
      }
    }

    rope()
    {
      tree_ptr = 0;
    }

    // Construct a rope from a function that can compute its members
    rope(char_producer<charT> *fn, size_t len, bool delete_fn)
```

```
{
    tree_ptr = RopeFunction_from_fn(fn, len, delete_fn);
}

rope(const rope &x)
{
    tree_ptr = x.tree_ptr;
    ref(tree_ptr);
}

~rope()
{
    unref(tree_ptr);
}

rope& operator=(const rope& x)
{
    RopeBase *old = tree_ptr;
    tree_ptr = x.tree_ptr;
    ref(tree_ptr);
    unref(old);
    return(*this);
}

void push_back(charT x)
{
    RopeBase *old = tree_ptr;
    tree_ptr = concat_char_iter(tree_ptr, &x, 1);
    unref(old);
}

void pop_back()
{
    RopeBase *old = tree_ptr;
    tree_ptr = substring(tree_ptr, 0, tree_ptr -> size - 1);
    unref(old);
}

charT back() const
{
    return fetch(tree_ptr, tree_ptr -> size - 1);
}

void push_front(charT x)
{
    RopeBase *old = tree_ptr;
    RopeBase *left;

    left = RopeLeaf_from_unowned_char_ptr(&x, 1);
```

```
        __STL_TRY {
          tree_ptr = concat(left, tree_ptr);
          unref(old);
              unref(left);
            }
        __STL_UNWIND(unref(left))
    }

    void pop_front()
    {
        RopeBase *old = tree_ptr;
        tree_ptr = substring(tree_ptr, 1, tree_ptr -> size);
        unref(old);
    }

    charT front() const
    {
        return fetch(tree_ptr, 0);
    }

    void balance()
    {
        RopeBase *old = tree_ptr;
        tree_ptr = balance(tree_ptr);
        unref(old);
    }

    void copy(charT * buffer) const {
        destroy(buffer, buffer + size());
        flatten(tree_ptr, buffer);
    }

    // This is the copy function from the standard, but
    // with the arguments reordered to make it consistent with the
    // rest of the interface.
    // Note that this guaranteed not to compile if the draft standard
    // order is assumed.
    size_type copy(size_type pos, size_type n, charT *buffer) const
 {
        size_t sz = size();
        size_t len = (pos + n > sz? sz - pos : n);

        destroy(buffer, buffer + len);
        flatten(tree_ptr, pos, len, buffer);
        return len;
    }

    // Print to stdout, exposing structure.  May be useful for
    // performance debugging.
```

```
    void dump() {
        dump(tree_ptr);
    }

    // Convert to 0 terminated string in new allocated memory.
    // Embedded 0s in the input do not terminate the copy.
    const charT * c_str() const;

    // As above, but lso use the flattened representation as the
    // the new rope representation.
    const charT * replace_with_c_str();

    // Reclaim memory for the c_str generated flattened string.
    // Intentionally undocumented, since it's hard to say when this
    // is safe for multiple threads.
    void delete_c_str () {
        if (0 == tree_ptr) return;
        if (RopeBase::leaf == tree_ptr -> tag
          && ((RopeLeaf *)tree_ptr) -> data == tree_ptr -> c_string)
{
          // Representation shared
          return;
        }
#       ifndef __GC
          tree_ptr -> free_c_string();
#       endif
        tree_ptr -> c_string = 0;
    }

    charT operator[] (size_type pos) const {
        return fetch(tree_ptr, pos);
    }

    charT at(size_type pos) const {
       // if (pos >= size()) throw out_of_range;
       return (*this)[pos];
    }

    const_iterator begin() const {
        return(const_iterator(tree_ptr, 0));
    }

    // An easy way to get a const iterator from a non-const container.
    const_iterator const_begin() const {
        return(const_iterator(tree_ptr, 0));
    }

    const_iterator end() const {
        return(const_iterator(tree_ptr, size()));
```

```
    }

    const_iterator const_end() const {
        return(const_iterator(tree_ptr, size()));
    }

    size_type size() const {
        return(0 == tree_ptr? 0 : tree_ptr -> size);
    }

    size_type length() const {
        return size();
    }

    size_type max_size() const {
        return min_len[RopeBase::max_rope_depth-1] - 1;
        //  Guarantees that the result can be sufficirntly
        //  balanced.  Longer ropes will probably still work,
        //  but it's harder to make guarantees.
    }

#    ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
       typedef reverse_iterator<const_iterator>
const_reverse_iterator;
#    else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
    typedef reverse_iterator<const_iterator, value_type,
const_reference,
                  difference_type>  const_reverse_iterator;
#    endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

    const_reverse_iterator rbegin() const {
        return const_reverse_iterator(end());
    }

    const_reverse_iterator const_rbegin() const {
        return const_reverse_iterator(end());
    }

    const_reverse_iterator rend() const {
        return const_reverse_iterator(begin());
    }

    const_reverse_iterator const_rend() const {
        return const_reverse_iterator(begin());
    }

    friend rope<charT,Alloc>
        operator+ __STL_NULL_TMPL_ARGS (const rope<charT,Alloc>
&left,
```

```
                                const rope<charT,Alloc> &right);

    friend rope<charT,Alloc>
        operator+ __STL_NULL_TMPL_ARGS (const rope<charT,Alloc>
&left,
                                const charT* right);

    friend rope<charT,Alloc>
        operator+ __STL_NULL_TMPL_ARGS (const rope<charT,Alloc>
&left,
                                charT right);

    // The symmetric cases are intentionally omitted, since they're
presumed
    // to be less common, and we don't handle them as well.

    // The following should really be templatized.
    // The first argument should be an input iterator or
    // forward iterator with value_type charT.
    rope& append(const charT* iter, size_t n) {
        RopeBase* result = destr_concat_char_iter(tree_ptr, iter, n);
        unref(tree_ptr);
        tree_ptr = result;
        return *this;
    }

    rope& append(const charT* c_string) {
        size_t len = char_ptr_len(c_string);
        append(c_string, len);
        return(*this);
    }

    rope& append(const charT* s, const charT* e) {
        RopeBase* result =
              destr_concat_char_iter(tree_ptr, s, e - s);
        unref(tree_ptr);
        tree_ptr = result;
        return *this;
    }

    rope& append(const_iterator s, const_iterator e) {
        __stl_assert(s.root == e.root);
        self_destruct_ptr appendee(substring(s.root, s.current_pos,
                            e.current_pos));
        RopeBase* result = concat(tree_ptr, (RopeBase *)appendee);
        unref(tree_ptr);
        tree_ptr = result;
        return *this;
    }
```

```
    rope& append(charT c) {
       RopeBase* result = destr_concat_char_iter(tree_ptr, &c, 1);
       unref(tree_ptr);
       tree_ptr = result;
       return *this;
    }

    rope& append() { return append(charT()); }

    rope& append(const rope& y) {
       RopeBase* result = concat(tree_ptr, y.tree_ptr);
       unref(tree_ptr);
       tree_ptr = result;
       return *this;
    }

    rope& append(size_t n, charT c) {
       rope<charT,Alloc> last(n, c);
       return append(last);
    }

    void swap(rope& b) {
       RopeBase * tmp = tree_ptr;
       tree_ptr = b.tree_ptr;
       b.tree_ptr = tmp;
    }


    protected:
    // Result is included in refcount.
    static RopeBase * replace(RopeBase *old, size_t pos1,
                              size_t pos2, RopeBase *r) {
       if (0 == old) { ref(r); return r; }
       self_destruct_ptr left(substring(old, 0, pos1));
       self_destruct_ptr right(substring(old, pos2, old -> size));
       RopeBase * result;

       if (0 == r) {
         result = concat(left, right);
       } else {
         self_destruct_ptr left_result(concat(left, r));
         result = concat(left_result, right);
       }
       return result;
    }

    public:
    void insert(size_t p, const rope& r) {
```

```
    RopeBase * result = replace(tree_ptr, p, p,
                          r.tree_ptr);
    unref(tree_ptr);
    tree_ptr = result;
}

void insert(size_t p, size_t n, charT c) {
    rope<charT,Alloc> r(n,c);
    insert(p, r);
}

void insert(size_t p, const charT * i, size_t n) {
    self_destruct_ptr left(substring(tree_ptr, 0, p));
    self_destruct_ptr right(substring(tree_ptr, p, size()));
    self_destruct_ptr left_result(concat_char_iter(left, i, n));
    RopeBase * result =
               concat(left_result, right);
    unref(tree_ptr);
    tree_ptr = result;
}

void insert(size_t p, const charT * c_string) {
    insert(p, c_string, char_ptr_len(c_string));
}

void insert(size_t p, charT c) {
    insert(p, &c, 1);
}

void insert(size_t p) {
    charT c = charT();
    insert(p, &c, 1);
}

void insert(size_t p, const charT *i, const charT *j) {
    rope r(i, j);
    insert(p, r);
}

void insert(size_t p, const const_iterator& i,
                const const_iterator& j) {
    rope r(i, j);
    insert(p, r);
}

void insert(size_t p, const iterator& i,
                const iterator& j) {
    rope r(i, j);
    insert(p, r);
```

```
    }

    // (position, length) versions of replace operations:

    void replace(size_t p, size_t n, const rope& r) {
        RopeBase * result = replace(tree_ptr, p, p + n,
                                    r.tree_ptr);
        unref(tree_ptr);
        tree_ptr = result;
    }

    void replace(size_t p, size_t n, const charT *i, size_t i_len)
{
        rope r(i, i_len);
        replace(p, n, r);
    }

    void replace(size_t p, size_t n, charT c) {
        rope r(c);
        replace(p, n, r);
    }

    void replace(size_t p, size_t n, const charT *c_string) {
        rope r(c_string);
        replace(p, n, r);
    }

    void replace(size_t p, size_t n, const charT *i, const charT *j)
{
        rope r(i, j);
        replace(p, n, r);
    }

    void replace(size_t p, size_t n,
                 const const_iterator& i, const const_iterator& j) {
        rope r(i, j);
        replace(p, n, r);
    }

    void replace(size_t p, size_t n,
                 const iterator& i, const iterator& j) {
        rope r(i, j);
        replace(p, n, r);
    }

    // Single character variants:
    void replace(size_t p, charT c) {
        iterator i(this, p);
        *i = c;
```

```
}

void replace(size_t p, const rope& r) {
   replace(p, 1, r);
}

void replace(size_t p, const charT *i, size_t i_len) {
   replace(p, 1, i, i_len);
}

void replace(size_t p, const charT *c_string) {
   replace(p, 1, c_string);
}

void replace(size_t p, const charT *i, const charT *j) {
   replace(p, 1, i, j);
}

void replace(size_t p, const const_iterator& i,
               const const_iterator& j) {
   replace(p, 1, i, j);
}

void replace(size_t p, const iterator& i,
               const iterator& j) {
   replace(p, 1, i, j);
}

// Erase, (position, size) variant.
void erase(size_t p, size_t n) {
   RopeBase * result = replace(tree_ptr, p, p + n, 0);
   unref(tree_ptr);
   tree_ptr = result;
}

// Erase, single character
void erase(size_t p) {
   erase(p, p + 1);
}

// Insert, iterator variants.
iterator insert(const iterator& p, const rope& r)
     { insert(p.index(), r); return p; }
iterator insert(const iterator& p, size_t n, charT c)
     { insert(p.index(), n, c); return p; }
iterator insert(const iterator& p, charT c)
     { insert(p.index(), c); return p; }
iterator insert(const iterator& p )
     { insert(p.index()); return p; }
```

```
    iterator insert(const iterator& p, const charT *c_string)
         { insert(p.index(), c_string); return p; }
    iterator insert(const iterator& p, const charT *i, size_t n)
         { insert(p.index(), i, n); return p; }
    iterator insert(const iterator& p, const charT *i, const charT
*j)
         { insert(p.index(), i, j);  return p; }
    iterator insert(const iterator& p,
              const const_iterator& i, const const_iterator& j)
         { insert(p.index(), i, j); return p; }
    iterator insert(const iterator& p,
              const iterator& i, const iterator& j)
         { insert(p.index(), i, j); return p; }

    // Replace, range variants.
    void replace(const iterator& p, const iterator& q,
             const rope& r)
         { replace(p.index(), q.index() - p.index(), r); }
    void replace(const iterator& p, const iterator& q, charT c)
         { replace(p.index(), q.index() - p.index(), c); }
    void replace(const iterator& p, const iterator& q,
             const charT * c_string)
         { replace(p.index(), q.index() - p.index(), c_string); }
    void replace(const iterator& p, const iterator& q,
             const charT *i, size_t n)
         { replace(p.index(), q.index() - p.index(), i, n); }
    void replace(const iterator& p, const iterator& q,
             const charT *i, const charT *j)
         { replace(p.index(), q.index() - p.index(), i, j); }
    void replace(const iterator& p, const iterator& q,
             const const_iterator& i, const const_iterator& j)
         { replace(p.index(), q.index() - p.index(), i, j); }
    void replace(const iterator& p, const iterator& q,
             const iterator& i, const iterator& j)
         { replace(p.index(), q.index() - p.index(), i, j); }

    // Replace, iterator variants.
    void replace(const iterator& p, const rope& r)
         { replace(p.index(), r); }
    void replace(const iterator& p, charT c)
         { replace(p.index(), c); }
    void replace(const iterator& p, const charT * c_string)
         { replace(p.index(), c_string); }
    void replace(const iterator& p, const charT *i, size_t n)
         { replace(p.index(), i, n); }
    void replace(const iterator& p, const charT *i, const charT *j)
         { replace(p.index(), i, j); }
    void replace(const iterator& p, const_iterator i, const_iterator
j)
```

```
        { replace(p.index(), i, j); }
    void replace(const iterator& p, iterator i, iterator j)
        { replace(p.index(), i, j); }

    // Iterator and range variants of erase
    iterator erase(const iterator &p, const iterator &q) {
           size_t p_index = p.index();
           erase(p_index, q.index() - p_index);
           return iterator(this, p_index);
       }
       iterator erase(const iterator &p) {
           size_t p_index = p.index();
           erase(p_index, 1);
           return iterator(this, p_index);
       }

    rope substr(size_t start, size_t len = 1) const {
       return rope<charT,Alloc>(
             substring(tree_ptr, start, start + len));
    }

    rope substr(iterator start, iterator end) const {
       return rope<charT,Alloc>(
             substring(tree_ptr, start.index(), end.index()));
    }

    rope substr(iterator start) const {
       size_t pos = start.index();
       return rope<charT,Alloc>(
             substring(tree_ptr, pos, pos + 1));
    }

    rope substr(const_iterator start, const_iterator end) const {
       // This might eventually take advantage of the cache in the
       // iterator.
       return rope<charT,Alloc>
         (substring(tree_ptr, start.index(), end.index()));
    }

    rope<charT,Alloc> substr(const_iterator start) {
       size_t pos = start.index();
       return rope<charT,Alloc>(substring(tree_ptr, pos, pos + 1));
    }

    size_type find(charT c, size_type pos = 0) const;
    size_type find(charT *s, size_type pos = 0) const {
       const_iterator result = search(const_begin() + pos,
const_end(),
                           s, s + char_ptr_len(s));
```

```
        return result.index();
    }

    iterator mutable_begin() {
        return(iterator(this, 0));
    }

    iterator mutable_end() {
        return(iterator(this, size()));
    }

#     ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
        typedef reverse_iterator<iterator> reverse_iterator;
#     else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
    typedef reverse_iterator<iterator, value_type, reference,
                  difference_type>  reverse_iterator;
#     endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

    reverse_iterator mutable_rbegin() {
        return reverse_iterator(mutable_end());
    }

    reverse_iterator mutable_rend() {
        return reverse_iterator(mutable_begin());
    }

    reference mutable_reference_at(size_type pos) {
        return reference(this, pos);
    }

#   ifdef __STD_STUFF
        reference operator[] (size_type pos) {
          return charT_ref_proxy(this, pos);
        }

        reference at(size_type pos) {
          // if (pos >= size()) throw out_of_range;
          return (*this)[pos];
        }

        void resize(size_type n, charT c) {}
        void resize(size_type n) {}
        void reserve(size_type res_arg = 0) {}
        size_type capacity() const {
          return max_size();
        }

      // Stuff below this line is dangerous because it's error prone.
      // I would really like to get rid of it.
```

```
        // copy function with funny arg ordering.
          size_type copy(charT *buffer, size_type n, size_type pos =
0)
                                       const {
         return copy(pos, n, buffer);
         }

        iterator end() { return mutable_end(); }

        iterator begin() { return mutable_begin(); }

        reverse_iterator rend() { return mutable_rend(); }

        reverse_iterator rbegin() { return mutable_rbegin(); }

#  else

        const_iterator end() { return const_end(); }

        const_iterator begin() { return const_begin(); }

        const_reverse_iterator rend() { return const_rend(); }

        const_reverse_iterator rbegin() { return const_rbegin(); }

#  endif

};

template <class charT, class Alloc>
inline bool operator== (const __rope_const_iterator<charT,Alloc> &
x,
             const __rope_const_iterator<charT,Alloc> & y) {
   return (x.current_pos == y.current_pos && x.root == y.root);
}

template <class charT, class Alloc>
inline bool operator< (const __rope_const_iterator<charT,Alloc> & x,
              const __rope_const_iterator<charT,Alloc> & y) {
   return (x.current_pos < y.current_pos);
}

template <class charT, class Alloc>
inline ptrdiff_t operator-(const __rope_const_iterator<charT,Alloc>
& x,
               const __rope_const_iterator<charT,Alloc> & y) {
   return x.current_pos - y.current_pos;
}
```

```
template <class charT, class Alloc>
inline __rope_const_iterator<charT,Alloc>
operator-(const __rope_const_iterator<charT,Alloc> & x,
     ptrdiff_t n) {
   return __rope_const_iterator<charT,Alloc>(x.root, x.current_pos
- n);
}

template <class charT, class Alloc>
inline __rope_const_iterator<charT,Alloc>
operator+(const __rope_const_iterator<charT,Alloc> & x,
     ptrdiff_t n) {
   return __rope_const_iterator<charT,Alloc>(x.root, x.current_pos
+ n);
}

template <class charT, class Alloc>
inline __rope_const_iterator<charT,Alloc>
operator+(ptrdiff_t n,
     const __rope_const_iterator<charT,Alloc> & x) {
   return __rope_const_iterator<charT,Alloc>(x.root, x.current_pos
+ n);
}

template <class charT, class Alloc>
inline bool operator== (const __rope_iterator<charT,Alloc> & x,
             const __rope_iterator<charT,Alloc> & y) {
   return (x.current_pos == y.current_pos && x.root_rope ==
y.root_rope);
}

template <class charT, class Alloc>
inline bool operator< (const __rope_iterator<charT,Alloc> & x,
             const __rope_iterator<charT,Alloc> & y) {
   return (x.current_pos < y.current_pos);
}

template <class charT, class Alloc>
inline ptrdiff_t operator-(const __rope_iterator<charT,Alloc> & x,
               const __rope_iterator<charT,Alloc> & y) {
   return x.current_pos - y.current_pos;
}

template <class charT, class Alloc>
inline __rope_iterator<charT,Alloc>
operator-(const __rope_iterator<charT,Alloc> & x,
     ptrdiff_t n) {
   return __rope_iterator<charT,Alloc>(x.root_rope, x.current_pos
- n);
```

```
}

template <class charT, class Alloc>
inline __rope_iterator<charT,Alloc>
operator+(const __rope_iterator<charT,Alloc> & x,
     ptrdiff_t n) {
   return __rope_iterator<charT,Alloc>(x.root_rope, x.current_pos
+ n);
}

template <class charT, class Alloc>
inline __rope_iterator<charT,Alloc>
operator+(ptrdiff_t n,
     const __rope_iterator<charT,Alloc> & x) {
   return __rope_iterator<charT,Alloc>(x.root_rope, x.current_pos
+ n);
}

template <class charT, class Alloc>
inline
rope<charT,Alloc>
operator+ (const rope<charT,Alloc> &left,
      const rope<charT,Alloc> &right)
{
   return rope<charT,Alloc>
        (rope<charT,Alloc>::concat(left.tree_ptr,
right.tree_ptr));
   // Inlining this should make it possible to keep left and
   // right in registers.
}

template <class charT, class Alloc>
inline
rope<charT,Alloc>&
operator+= (rope<charT,Alloc> &left,
       const rope<charT,Alloc> &right)
{
   left.append(right);
   return left;
}

template <class charT, class Alloc>
inline
rope<charT,Alloc>
operator+ (const rope<charT,Alloc> &left,
      const charT* right) {
   size_t rlen = rope<charT,Alloc>::char_ptr_len(right);
   return rope<charT,Alloc>
     (rope<charT,Alloc>::concat_char_iter(left.tree_ptr, right,
```

```
rlen));
}

template <class charT, class Alloc>
inline
rope<charT,Alloc>&
operator+= (rope<charT,Alloc> &left,
       const charT* right) {
    left.append(right);
    return left;
}

template <class charT, class Alloc>
inline
rope<charT,Alloc>
operator+ (const rope<charT,Alloc> &left, charT right) {
    return rope<charT,Alloc>
         (rope<charT,Alloc>::concat_char_iter(left.tree_ptr,
&right, 1));
}

template <class charT, class Alloc>
inline
rope<charT,Alloc>&
operator+= (rope<charT,Alloc> &left, charT right) {
    left.append(right);
    return left;
}

template <class charT, class Alloc>
bool
operator< (const rope<charT,Alloc> &left, const rope<charT,Alloc>
&right) {
    return left.compare(right) < 0;
}

template <class charT, class Alloc>
bool
operator== (const rope<charT,Alloc> &left, const rope<charT,Alloc>
&right) {
    return left.compare(right) == 0;
}

template <class charT, class Alloc>
inline bool operator== (const __rope_charT_ptr_proxy<charT,Alloc> &
x,
             const __rope_charT_ptr_proxy<charT,Alloc> & y) {
    return (x.pos == y.pos && x.root == y.root);
}
```

```
template<class charT, class Alloc>
ostream& operator<< (ostream& o, const rope<charT, Alloc>& r);

typedef rope<char, __ALLOC> crope;
typedef rope<wchar_t, __ALLOC> wrope;

inline crope::reference __mutable_reference_at(crope& c, size_t i)
{
    return c.mutable_reference_at(i);
}

inline wrope::reference __mutable_reference_at(wrope& c, size_t i)
{
    return c.mutable_reference_at(i);
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class charT, class Alloc>
inline void swap(rope<charT, Alloc>& x, rope<charT, Alloc>& y) {
  x.swap(y);
}

#else

inline void swap(crope x, crope y) { x.swap(y); }
inline void swap(wrope x, wrope y) { x.swap(y); }

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

// Hash functions should probably be revisited later:
__STL_TEMPLATE_NULL struct hash<crope>
{
  size_t operator()(const crope& str) const
  {
    size_t sz = str.size();

    if (0 == sz) return 0;
    return 13*str[0] + 5*str[sz - 1] + sz;
  }
};


__STL_TEMPLATE_NULL struct hash<wrope>
{
  size_t operator()(const wrope& str) const
  {
    size_t sz = str.size();
```

```
    if (0 == sz) return 0;
    return 13*str[0] + 5*str[sz - 1] + sz;
  }
};

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM !=
_MIPS_SIM_ABI32)
#pragma reset woff 1174
#endif

__STL_END_NAMESPACE

# include <ropeimpl.h>
# endif /* __SGI_STL_INTERNAL_ROPE_H */

// Local Variables:
// mode:C++
// End:
```