BoundedList Template

Peter C. Chapin
Computer Information Systems
VERMONT TECH

November 4, 2013

Standard List

The C++ standard provides a list template `std::list<T>`.

- Can hold elements of any type.
- Can grow (and shrink) dynamically to any size.
- Provides $O(1)$ access to *both* ends.
- Provides $O(1)$ insert and erase operations.

Does not provide operations for random access. They would be $O(n)$.

Standard List Iterators

The type `std::list<T>::iterator` is a *bidirectional iterator* type.

```
template<typename Bidirectional>
void reverse( Bidirectional first, Bidirectional last )
{
  if( first == last ) return;
  --last;
  while( first != last ) {
    swap( *first, *last );
    ++first;
    if( first == last ) break;
    --last;
  }
}

std::list<int> my_list { 1, 2, 3, 4 };
reverse( my_list.begin( ), my_list.end( ) );
```

Constant Time Insert

Lists excel at insert (and erase) operations in the middle of the list.

```
void insert_before_dot( list<string> &lst,
                        const string &incoming )
{
  list<string>::iterator dot =
    std::find( lst.begin( ), lst.end( ), "." ); // O(n)

  if( dot != lst.end( ) ) {
    lst.insert( dot, incoming );                // O(1)
  }
}
```

The `std::find` algorithm searches a sequence for a particular item and returns an iterator to the first occurrence or, if the item is not found, an iterator just past the sequence.

The `insert` method inserts a new item into the list before the position indicated by the given iterator.

BoundedList

BoundedLists pre-allocate space for a fixed number of items.

- Maximum number of items given as constructor parameter.
- Space can't grow or shrink.
- List uses allocated space... can grow and shrink inside bound.
- Otherwise behaves similarly to the standard list.

The type `BoundedList<T>` is a bounded list holding items of type `T`.

The type `BoundedList<T>::iterator` is a bidirectional iterator type.

BoundedList Sample

Behaves like `std::list<T>`.

```
// Specify maximum element count during construction.
BoundedList<int> my_list( 16 );

// Assignment of an initializer list supported.
my_list = { 1, 2, 3, 4 };

// BoundedList iterators are bidirectional.
reverse( my_list.begin( ), my_list.end( ) );
```

Initializer list constructor not supported; its size would also fix the upper bound on the list (not appropriate).

The `reverse` algorithm is the same as presented earlier; works fine with `BoundedList`.

Why Bother?

What's the point of BoundedList?

- Memory sensitive applications can't allow one data structure to take memory away from others.
- If all memory is pre-allocated a careful accounting of memory can be made.
- Runaway growth is likely a sign of error and should not be supported.

However, there are obvious disadvantages.

- Application might consume more memory than it really needs.
- "Out of memory" errors still possible (if BoundedList fills up)... even with free memory available.

BoundedList is a *speciality* data structure.

Out of Memory

From the C++ 2011 standard:

> "The class `length_error` defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size."

```
BoundedList<int> my_list( 4 );

try {
  my_list = { 1, 2, 3, 4 };
  my_list.push_back( 5 );
  // Other operations on my_list...
}
catch( const std::length_error &ex ) {
  std::cout << "Caught length_error: " << ex.what( ); << "\n";
}
```

Implementation Notes

Use the style of the standard library.

- Provide “the usual nested types.”
- Provide a nested iterator class named `iterator`.
- Provide `begin` and `end` methods.
- Provide the same supporting methods (like `insert`).

Advantages:

- Plays well with standard algorithms.
- Plays well with standard template adaptors.
- Plays well with third party code.
- Very low learning curve.

The Usual Types

```
template< typename T >
class BoundedList {
public:
  typedef       T  value_type;
  typedef       T *pointer;
  typedef const T *const_pointer;
  typedef       T &reference;
  typedef const T &const_reference;
  typedef std::size_t    size_type;
  typedef std::ptrdiff_t difference_type;

  // ...
```

All library containers define nested types like above. Some algorithms depend on it. *Thus we should define these types also so we play well with the library.*

The Usual Types

- `BoundedList<T>::value_type` is the list element type.
- `BoundedList<T>::size_type` is a type suitable for measuring the size of a BoundedList.
- `BoundedList<T>::difference_type` is a type suitable for holding the difference between two iterators.

```
BoundedList<int> my_list( 16 );
...
BoundedList<int>::size_type list_size = my_list.size( );
```

The type `size_type` is some kind of unsigned integer; probably the same as `std::size_t` for most containers. Some "exotic" containers might use a larger unsigned integer.

Representation

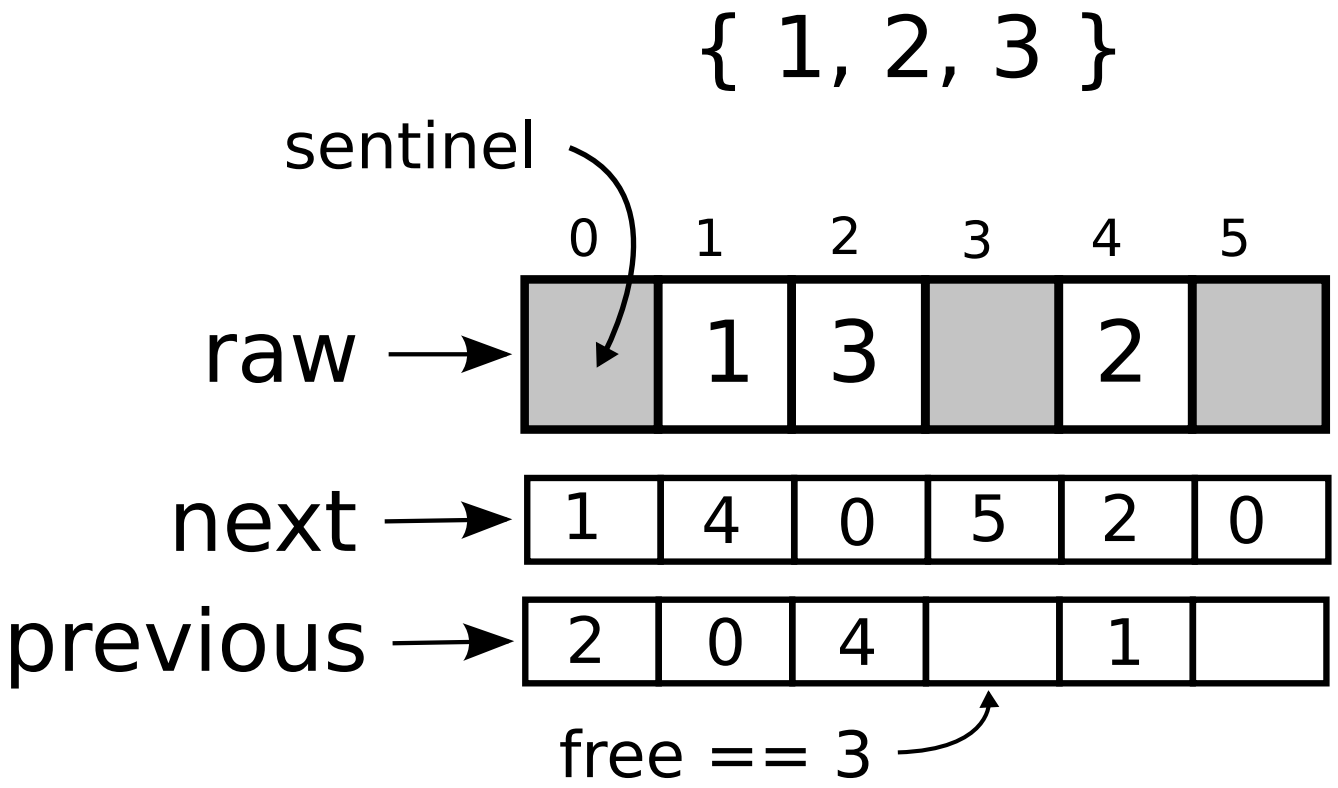
```
template< typename T >
class BoundedList {
private:
  T         *raw;        // Pre-allocated block of raw memory.
  size_type *next;       // Array of "next" indicies.
  size_type *previous;   // Array of "previous" indicies.
  size_type  count;      // Number of items in list.
  size_type  capacity;   // Size of pre-allocated block.
  size_type  free;       // Front of the free list.

  // ...
```

Three parallel arrays: one containing the list elements and two others containing the “next” and “previous” index values for each element.

Element zero of the arrays is for the *sentinel* node (used to mark both ends of the list). The sentinel is not part of the list data.

Memory Layout



Memory Trickery

Only certain slots in the raw array are really constructed `T` *objects.* It is necessary to manage construction and destruction manually.

```
template< typename T >
BoundedList<T>::BoundedList( size_type max_count )
{
  raw =
    reinterpret_cast<T *>(new char[(max_count + 1)*sizeof(T)]);

  next     = new size_type[max_count + 1];
  previous = new size_type[max_count + 1];
  count    = 0;
  capacity = max_count;
  free     = 0;
  // ...
```

Constructor allocates raw memory as an appropriately sized block of characters to avoid constructing unnecessary `T` objects. The cast changes interpretation of the resulting pointer.

Space for `max_count + 1` items allocated to account for sentinel node.

Wire the Free List

```
template< typename T >
BoundedList<T>::BoundedList( size_type max_count )
{
  // ...

  // Prepare the free list.
  for( size_type i = 1; i <= capacity; ++i ) {
    next[i] = i + 1;
  }
  if( capacity > 0 ) {
    next[capacity] = 0;
    free = 1;
  }

  // Configure the sentinel.
  next[0] = 0;
  previous[0] = 0;
}
```

The sentinel's `next` and `previous` pointers (indicies) point at itself.

Destructor Destroys Active List Elements

```
template< typename T >
BoundedList<T>::~BoundedList( )
{
  // Destroy all the active items.
  size_type current = next[0];
  while( current != 0 ) {
    raw[current].~T( );
    current = next[current];
  }

  // Release the memory.
  delete [] previous;
  delete [] next;
  delete [] reinterpret_cast< char * >( raw );
}
```

Red line shows explicit invocation of destructor. *Very unusual technique.*

Implementation of push_back

```
template< typename T >
void BoundedList<T>::push_back( const T &item )
{
    if( free == 0 )
      throw std::length_error( "BoundedList: full" );

    // Pull a slot off the free list.
    size_type new_item = free;
    free = next[free];

    // Construct a copy of the item in the desired slot.
    new ( &raw[new_item] ) T( item );

    next[previous[0]]  = new_item;
    previous[new_item] = previous[0];
    next[new_item]     = 0;
    previous[0]        = new_item;

    ++count;
}
```

Red line shows “placement new.” *Very unusual technique.*

Implementation of pop_back

```
template< typename T >
void BoundedList<T>::pop_back( )
{
  // Remove the last item (I assume one is present).
  size_type p = previous[previous[0]];
  raw[previous[0]].~T( );
  next[previous[0]] = free;
  free = previous[0];

  previous[0] = p;
  next[p] = 0;
  count--;
}
```

Red line shows explicit invocation of destructor.

BoundedList<T>::iterator

```
template< typename T >
class BoundedList {
public:
  class iterator :
    public std::iterator<std::bidirectional_iterator_tag, T> {
  public:
    iterator( );
    iterator( BoundedList *list, size_type index );
    iterator &operator++( );
    iterator &operator--( );
    bool      operator==( const iterator &other );
    bool      operator!=( const iterator &other );
    reference operator* ( );
    pointer   operator->( );
  };
    // ...
```

- Deriving from `std::iterator` template inherits certain types.
- An iterator knows which BoundedList it points into.
- The **`operator`**`->` is only useful if list elements are structures/classes.

Implementation of Iterator Inc/Dec

```
template< typename T >
BoundedList<T>::iterator &
  BoundedList<T>::iterator::operator++( )
{
  my_node = my_list->next[my_node];
  return( *this );
}

template< typename T >
BoundedList<T>::iterator &
  BoundedList<T>::iterator::operator--( )
{
  my_node = my_list->previous[my_node];
  return( *this );
}
```

An iterator knows its current list and node. It must be a **`friend`** of BoundedList to access the private section of the list.

Implementation of begin/end

```
template< typename T >
BoundedList<T>::iterator BoundedList<T>::begin( )
{
  return iterator( this, next[0] );
}
```

```
template< typename T >
BoundedList<T>::iterator BoundedList<T>::end( )
{
  return iterator( this, 0 );
}
```

These methods return an appropriate iterator from **`this`** object.

Full Details

- Complete implementation in `BoundedList.hpp`.
- Test program in `check/BoundedList_tests.cpp`.

Questions?

PChapin@vtc.vsc.edu