Subject: I don't get some aspects of STL ... [pointless rant]
Posted by mr_ped on Wed, 18 Mar 2009 20:45:27 GMT
View Forum Message <> Reply to Message

I'm forced to use STL for some short piece of code, and I'm wondering whether this is really that great of language extension... (also I became so used to NTL naming and pick behavior, that it's a real pain now for me to watch how STL works [inefficiently])

At first I did need to put some strings into hash map, so I found out std::hash_map .. then I figured out it's renamed to something new in 4.3.x and finally part of default stlib, but in older compilers it's just some extension which may be not available (my case). At that point I was so sick of the situation, I simply took my old source with CCIT 16bit CRC which works as excellent 16b hash function for strings with minimal CPU usage, created 64k of vector<int>, add/get functions, and there we go, problem solved.

Then I did want to sort some strings and get the conversion data of old indices to new indices, so I can reorder also additional data in different structures. There's some clever sort variant in U++ doing exactly this (can't recall it's name), but to my surprise there's nothing like that in STL. Although after little bit of thinking I figured out something like this:

```
struct ParentClass::S_compare {
 const ParentClass & p;
 S_compare( const ParentClass & parent ) : p( parent ) {}
 bool operator() ( int i, int j ) {
  return ( /* desired compare on p.get(i) vs p.get(j) */ );
 }
};

void ParentClass::Sort( std::vector<int> & neworder )
{
 neworder.clear();
 for ( int oldid = 0; oldid < /*count of data*/; ++oldid )
  neworder.push_back( oldid );
 S_compare cmp(*this);
 sort( neworder.begin(), neworder.end(), cmp );
 //neworder contains old indices in proper order
}

//call Sort, and use the result in "neworder" to access your data in sorted manner
```

does what I need. A tad more code then simple 2~3 lines in U++, but not that bad, and actually it's quite powerful for extraordinary cases when you need some really crazy compare functions.

Then the third crash into STL today finally made me to write this rant here. I tried to use std::set_difference ... and there's no "in place" variant working on the range1 data as output too, which IMHO makes perfect sense when std::list are used and you don't need original data anymore. And as far as I can tell, there's no problem to write such algorithm (I will try in next minutes, so if there's some problem, I will learn the hard way).

And the vector_type::const_iterator b = v.begin(), e = v.end() is killing me. I'm all about verbose variable names (I rarely use "b" or "v", this is just fake example), but the "vector_type::const_iterator" is overkill for me, it makes me always to think how to not create a "for" loop, or rather use the int i variant, because it's too much writing.

I mean, the STL doesn't look that great to me. I can see how it saves some work in small applications, but I can easily imagine to take back some of that advantage with that verbosity of type names and definitions (see also the compare function embedded into struct). And for big things where performance matters the performance of STL makes it unusable. So far my real life experience with NTL made me always much more happy.

I either didn't understand the "spirit" of STL, or it sucks a bit. If anyone has some good comments to learn me more about it's spirit - so it will be less pain to use it effectively for me, I will be grateful. I will very likely still prefer NTL, but sometimes NTL is not available, while STL is.

Subject: Re: I don't get some aspects of STL ... [pointless rant]
Posted by cbpporter on Thu, 19 Mar 2009 08:33:56 GMT
View Forum Message <> Reply to Message

I'm sorry to disappoint you if you're looking for some enlightening comments that will help you appreciate STL better. Coming from NTL it will allays suck.

I never used STL for extended periods of time, and it's ugly design is probably the reason for it. At first I used Delphi for serious development, then MFC which had it's own containers. Then U++. And STL here and there in between.

But my latest effort in a small utility tool needs to use STL and I'm quite hatting it. I took me over an hour to load a file in memory. There is no LoadFile. I had to use a file stream, go to the end, take the position, compute the size, go to the start then allocate the buffer and read. I found that there is a flag you can use in the constructor so that the file cursor is positioned at the end of the file, so now I have take position, go to start, read. A lot shorter. The result of my efforts was an aptly named LoadFile function . Something like this should be part of the standard library. Loading files line by line is OK, but when creating parsers, loading the full file is often a must.

Also, other useful conversion functions seem to be missing. Try converting and int to a string, or vice-versa. You have basically two options. A stringstream and <<, or c string with a local buffer of fixed size and itoa. One would think that such basic conversion would be part of a general library, or at least string + int would work, seeing as strings don't convert to char*.

Iterators are also incredibly ugly and verbose. and the worst part is the lack of uniformity. I can navigate a vector or string by index, but that's about it. Delete for example needs and iterator. Even on vector. Deleting the fifth element is a simple as v.erase(v.begin() + 4).

Other gripes: wstring is again somewhere between Unicode 1 and 1.1 and wchar_t is 16 bit or 32 bits depending on platform. This wouldn't be a problem if STL would be Unicode aware, but it's not.

And please, include boost::format already. I hate cout<< because setting output options is again ugly and verbose. cout.width(10); cout << 100 << endl; cout.width(10); cout.flags (ios::right | ios::hex | ios::showbase); cout.width (20); cout << 100;
Yes... right... ahem...

But will all this, STL is really great. Looking both at the history of C/C++ and also current OSS projects, STL is a blessing. Every single C and even C++ project I have seen defines it's own int_t, int32_t, int32, int_32, __int32 or other variants of basic types. Also, prefixing it with the name of the library or the first letter is very popular: gint, fint, glint, axiom_int. If you want your programs to be portable and sizes of types are so different, than the standard should have a type that is guaranteed to accommodate some restrictions. You need a 32 bit integer? The standard says you need _____int_std____C__or_C___ESCP_ESCP_32_bits. There! Problem solved. This way not every single header will define these types. Actually, there is stdint.h, which offers a little more friendly names for types like int32_t and int_least32_t.

But redundant typedefs are only the tip of the iceberg. The problem is that every single C lib reinvents basic containers. Try building an application by combining open source libraries. In a project I ended up with 3 C string types and two C++ string types. Ironically the two C++ types were normal std::string, and normal std::string which support NULL values for DB interactions.

So basically, for every C++ library that doesn't invent it's own vector, STL gets on huge award and my gratitude. IMO it is a lot better if everyone uses the same ugly library, rather then them using separate libraries, which will always overlap in functionality and are often ugly to.

Subject: Re: I don't get some aspects of STL ... [pointless rant]
Posted by mr_ped on Thu, 19 Mar 2009 10:13:00 GMT
View Forum Message <> Reply to Message

Here are the "inplace" variants of set_difference for STL, written in a way mimicking the original code as much as possible.

Although I did hit two "major" problems which make these functions a bit different.
1) the original does not need the list container itself, while I need to do erase upon it, so the parameters are now not only from iterator family.
2) the return value points at the start of output range, not end. That's the way how I need it, and I wonder who needs end of range of new result anyway, I mean you need result usually to further use it, and usually the further usage starts to work at the start of result, so I simply can't figure out the reason why STL is returning end, except the simple "STL sucks and defends actively against being used in common programming problems in simple way" reason.
It's either me missing the spirit of STL and trying to abuse it in wrong way, or it's STL completely missing the common programming problems and instead solving unreal scenarios and helping with real ones only by accident and with difficulties.
Rant off, let's talk code .

set_algo.h

```
#ifndef _set_inplace_algo_h_
#define _set_inplace_algo_h_

#include <algorithm>

namespace std
{
  //for license of the original set_difference code see:
  //  .../c++/bits/stl_algo.h file.
  //If you find this different enough to be considered my code,
  //feel free to use it under BSD license. (my point of view)
  //If you feel this is derivative work of original, follow the
  //original license then.

  /**
   *  @brief Modify first range to return the difference of two sorted ranges.
   *  @param  first1  Start of first range.
   *  @param  last1   End of first range.
   *  @param  first2  Start of second range.
   *  @param  last2   End of second range.
   *  @param  list    List containing the first range
   *  @return  Start of modified first range.
   *  @ingroup setoperations
   *
   *  This operation mimicks behavior of set_difference, but the result is not
   *  copy of elements, instead the actual first range is modified (elements
   *  are erased from first range, if they are found in second range).
   *  The input ranges may not overlap, the output range is always within
   *  the first range.
   *  Return value is start (warning, original set_difference returns end) of
   *  modified first range (i.e. first unerased element or __last1).
  */
  template<typename _InputOutputIterator1, typename _InputIterator2,
           typename _ListContainer>
    _InputOutputIterator1
    set_difference_inplace(_InputOutputIterator1 __first1, _InputOutputIterator1 __last1,
                _InputIterator2 __first2, _InputIterator2 __last2,
                _ListContainer & __list)
    {
      // concept requirements
      __glibcxx_function_requires(_OutputIteratorConcept<_InputOutputIterator1,
            typename iterator_traits<_InputOutputIterator1>::value_type>)
      __glibcxx_function_requires(_InputIteratorConcept<_InputIterator2>)
      __glibcxx_function_requires(_SequenceConcept<_ListContainer>)
      __glibcxx_function_requires(_SameTypeConcept<
            typename iterator_traits<_InputOutputIterator1>::value_type,
            typename iterator_traits<_InputIterator2>::value_type>)
```

```
    __glibcxx_function_requires(_LessThanComparableConcept<
          typename iterator_traits<_InputOutputIterator1>::value_type>)
    __glibcxx_requires_sorted(__first1, __last1);
    __glibcxx_requires_sorted(__first2, __last2);

    _InputOutputIterator1 __ret = __first1;
    --__ret;      //move to safe place in case __first1 will be erased
    while (__first1 != __last1 && __first2 != __last2)
      if (*__first1 < *__first2)
        ++__first1;
      else if (*__first2 < *__first1)
        ++__first2;
      else
        {
          __first1 = __list.erase( __first1 );
          ++__first2;
        }
    return ++__ret;    //return the first unerased member of list
  }

/**
 *  @brief  Modify first range to return the difference of two sorted
 *  ranges using comparison functor.
 *  @param  first1  Start of first range.
 *  @param  last1   End of first range.
 *  @param  first2  Start of second range.
 *  @param  last2   End of second range.
 *  @param  comp    The comparison functor.
 *  @param  list    List containing the first range
 *  @return  Start of modified first range.
 *  @ingroup setoperations
 *
 *  This operation mimicks behavior of set_difference, but the result is
 *  not copy of elements, instead the actual first range is modified.
 *  Return value is start (warning, original set_difference returns end) of
 *  modified first range (i.e. first unerased element or __last1).
*/
template<typename _InputOutputIterator1, typename _InputIterator2,
         typename _Compare, typename _ListContainer>
  _InputOutputIterator1
  set_difference_inplace(_InputOutputIterator1 __first1, _InputOutputIterator1 __last1,
                _InputIterator2 __first2, _InputIterator2 __last2,
                _ListContainer & __list, _Compare __comp)
  {
    typedef typename iterator_traits<_InputOutputIterator1>::value_type
      _ValueType1;
    typedef typename iterator_traits<_InputIterator2>::value_type
      _ValueType2;
```

```
      // concept requirements
      __glibcxx_function_requires(_OutputIteratorConcept<_InputOutputIterator1,
                                  _ValueType1>)
      __glibcxx_function_requires(_InputIteratorConcept<_InputIterator2>)
      __glibcxx_function_requires(_SequenceConcept<_ListContainer>)
      __glibcxx_function_requires(_BinaryPredicateConcept<_Compare,
                                  _ValueType1, _ValueType2>)
      __glibcxx_function_requires(_BinaryPredicateConcept<_Compare,
                                  _ValueType2, _ValueType1>)
      __glibcxx_requires_sorted_set_pred(__first1, __last1, __first2, __comp);
      __glibcxx_requires_sorted_set_pred(__first2, __last2, __first1, __comp);

      _InputOutputIterator1 __ret = __first1;
      --__ret;      //move to safe place in case __first1 will be erased
      while (__first1 != __last1 && __first2 != __last2)
        if (__comp(*__first1, *__first2))
          ++__first1;
        else if (__comp(*__first2, *__first1))
          ++__first2;
        else
          {
            __first1 = __list.erase( __first1 );
            ++__first2;
          }
      return ++__ret;    //return the first unerased member of list
    }

} // namespace std

#endif
```

Warning, I didn't test this extensively yet, so there may be some bug hidden. If anyone is up to do a review, I will be really glad for it.

Subject: Re: I don't get some aspects of STL ... [pointless rant]
Posted by piotr5 on Mon, 11 May 2009 14:28:33 GMT
View Forum Message <> Reply to Message

I see stl less as a library than as a standard. usually the containers of stl are absolutely useless for me and I need to write my own. the way I do this is to just copy some container from the header-files of stl and modify it along my own wishes. for example a graph-container could be created by reusing the list-container code. another interesting aspect of stl is that quite useless algorithms are implemented in stl on a template-basis -- if you know a better implementation for some particular case you can always write it as a specialization, people will understand your code because you used exactly the same function-name as in stl. finally the strange choice to return the end-iterator after whatever operation is also an example for setting up a standard: there are

oftentimes cases with 2 or more possibilities for an interface, stl does tell us which to chose such that other people reading the sources wont be confused. but I agree, as a library stl certainly is just the last ressort when nothing else exists...
