Subject: Re: Impressive improvement in std::vector when dealing with raw memory.
Posted by Lance on Tue, 15 Nov 2022 14:58:41 GMT
View Forum Message <> Reply to Message

Story of std::basic_string (GLIBCXX implementation)

It's surprising that a basic_string<ch> would cause trouble (core dump etc) when treated as raw bytes. Digging into its implementation (in <bits/basic_string.h>), we have the data members

```
      // Use empty-base optimization: http://www.cantrip.org/emptyopt.html
      struct _Alloc_hider : allocator_type // TODO check __is_final
      {
#if __cplusplus < 201103L
 _Alloc_hider(pointer __dat, const _Alloc& __a = _Alloc())
 : allocator_type(__a), _M_p(__dat) { }
#else
 _Alloc_hider(pointer __dat, const _Alloc& __a)
 : allocator_type(__a), _M_p(__dat) { }

 _Alloc_hider(pointer __dat, _Alloc&& __a = _Alloc())
 : allocator_type(std::move(__a)), _M_p(__dat) { }
#endif

 pointer _M_p; // The actual data.
      };

      _Alloc_hider _M_dataplus;
      size_type  _M_string_length;

      enum { _S_local_capacity = 15 / sizeof(_CharT) };

      union
      {
 _CharT           _M_local_buf[_S_local_capacity + 1];
 size_type        _M_allocated_capacity;
      };
```

Ignore the Allocator and empty base optimization stuff, the member variables can be translated into

```
      enum { _S_local_capacity = 15 / sizeof(_CharT) };

    pointer _M_p;
    size_type _M_string_length;
    union
    {
 _CharT        _M_local_buf[_S_local_capacity + 1];
```

```
 size_type _M_allocated_capacity;
    };
```

Turns out, in the case the stored c-string can be fit in 15 bytes (one more for the null terminator), it stores the string locally and make _M_p point to it, thus created a class invariant that will break with raw move. It's disappointing that my copy raw bytes then do adjustment is less efficient than simply using move constructor, while it's logical that in the case when the object size is small comparing to adjustments that need to be made, adjusment-after-rawcopy will be more costly, let's try to blame somebody else.

Is basic_string has to be designed this way? I mean, for all it does, is the pointer to self action necessary? Indeed, it's not. We can make basic_string trivially relocatable without losing any functionality: just set _M_p to 1 when it's storing data locally!

A naive partial implementation of above idea looks like this

```
    pointer p;
    size_type _len;
    size_type _capacity;

    // fix: prepare for 16 - 2*sizeof(size_type) is 0!
    char _dummy[ 16 - 2* sizeof(size_type) ];
    enum{ local_capacity = 15 / sizeof(_CharT) };
    // if string is store locally, how difficult
    // is it to call strlen on a c-string with
    // length less than 15? we can certainly use
    // the space for _len for string storage too

    bool local()const{ return as_int(p) = 1; }

    // when no object in *this yet, ie, in constructor
    void store_a_strong_raw(const chT *s){
        assert( s!= nullptr );
        if( strlen(s) <= local_capacity )
        {
           copy_string_to_local_buff();
           as_int(p) = 1;
        }else{
           p = allocate_string_in_heap();
        }
    }
    void store_a_string(const char * s){
        if( !local() )
            delete [] p;
        store_a_strong_raw (s);
    }
```

```
~basic_string(){
    if( !local() )
       delete [] p;
}

size_type size()const{
    return local() ?
        getstrlen<chT>( local_storage_begin() ) :
        _len;
}

chT * local_storage_begin(){
    return reinterpret_cast<chT*>(
          &_len
        );
}
...
```

We have maintained the functionalities of original implementation, used less space, and most importantly, make basic_string objects trivially relocatable. Now basic_string objects will no longer be second class citizen in a vector/Vector world.

If you are more hackish: do we really need the full space of pointer p to determine if a string is stored locally? Depending on the endianness, we can potentially extract 7 more bytes on a 64-bit platform.
