Document number: *N4359*
Date: *2015-01-09*
Project: Programming Language C++,Library Evolution Working Group
Reply-to: *Jerry Liang 805600352@qq.com*

A Proposal to Add vector release method just like unique_ptr release method to the Standard Library

I. Table of Contents

II. Introduction

vector::release returns a direct pointer to the memory array used internally by the vector to store its owned elements,and then releases ownership of the elements by setting the internal status to null status. Difference between vector::release and vector::data is just like difference between unique_ptr::release and unique_ptr::get.

III. Motivation

The containers section of the standard library has become a familiar and valued tool over the years since standardisation, replacing low level manipulation of data structures and pointers with a consistent higher level interface.

However, standard library can't be used without conversion and data copy in interoperation between mordern C++ and C or other language or even some of C++ libraries. For example,from vector<char> to char[],there's no way to steal resource but copy.High level design usually causes overhead,but our(C++'s) goal is to achieve zero overhead.So why can't we think about to omit the data copy inside the conversion?

Remembering the old days without rvalue-reference,even within standard library,we do a lot of unnecessary copies to transfer data,while postbilt in D language with GC can omit all of them.Now we've realized that we need "move",but only move construct and assignment between two object of same type can be done.Why can't we step a little further and make stealing resource possible in more situation than just from one vector to another vector?unique_ptr already has one solution:unique_ptr::release,because unique_ptr designer knows that not every unique_ptr users won't need a raw pointer.So do vector users,and vector::data can't meet our need for stealing resources.

This proposal extends the vector class with one member function vector::release without any changes to anything else ,providing a way for stl users to steal the ownership of the memory originally owned by the vector.

IV. Impact On the Standard

This proposal is a extension to vector class. It only adds a member function to vector class,which modified standard library headers. It does not require any changes in the core language. It has been implemented in standard C++.

V. Design Decisions

Example and Typical case

```
// vector::release
#include <iostream>
#include <vector>

struct input_t
{
  int* statistics_data;
  int  statistics_size;
  //other fields ignored
};
extern void input(input_t in);
extern void set_callback(void (*fn)(int));//will call back after input
has been processed
void make_input ()
{
  std::vector<int> myvector;
  myvector.reserve(10000000);
  for(int i=0;i<50000000;++i)
    myvector.push_back(i*i);
  int n = myvector.size();
  int* p = myvector.release();
  input_t in;
  in.statistics_data = p;
  in.statistics_size = n;

  input(in);
}
int main ()
{
  set_callback([](int result){std::cout<<result<<endl;});
  make_input();
  wait_for_result();//some function to wait for result
  return 0;
}
```

In this example, myvector in make_input() firstly finished it’s job to collect data(data amount this job need in real world is dynamic depending on other conditions,so vector is chosen).Then the result should be input by calling function input(input_t) which will pass the data to another thread to use.The data don’t have to be copied there, just passing pointer is ok, if we don’t need touch these data anymore(giving up ownership) .

But unfortunately we used vector,and until now vector can only give up ownership of its elements to another vector. This proposal provides vector::release to solve this problem.After calling myvector.release(),myvector explicitly lost ownership of its elements and they are transfer to be used elsewhere and finally get dealocated via freeing struct input_t.

Destruction and deallocation issues

Just as when using unique_ptr::release you must use unique_ptr::get_deleter very often,when you’ve called vector::release,you’re on your own too.If you are handling vector<MyClass> ,you must call MyClass::~MyClass on the first size() elements, afterwards deallocate the chunk of memory. If you are handling vector<MyClass,MyAllocator>,you must also use get_allocator() to deallocate memory.

With destructor-less data types and default allocator,there are no such worries,while it’s just the typical case for vector::release.

VI. Technical Specifications

vector::release is designed to be a public member function of vector in this form:

```
value_type* release() noexcept;
```

Description

Returns a direct pointer to the memory array used internally by the vector to store its owned elements. ,and then releases ownership of the elements by setting the internal status to null status.
Because elements in the vector are guaranteed to be stored in contiguous storage locations in the same order as represented by the vector, the pointer retrieved can be offset to access any element in the array.

Parameters

None

Return value

A pointer to the first element in the array used internally by the vector.
Member type value_type is the type of the elements in the container, defined in vector as an alias of the first class template parameter (T).

Complexity

Constant.

Iterator validity

No changes.

Data races

The container is modified , concurrently accessing or modifying is unsafe.

Exception safety

No-throw guarantee: this member function never throws exceptions.

VII. Acknowledgements

- libstdc++@gcc.gnu.org
- Tim Shen
- Federico Terraneo
- Jonathan Wakely
- …

VII References

- [1] unique_ptr::release –C++ Reference
  http://www.cplusplus.com/reference/memory/unique_ptr/release/
- [2] vector::data –C++ Reference
  http://www.cplusplus.com/reference/vector/vector/data/