Homework #7

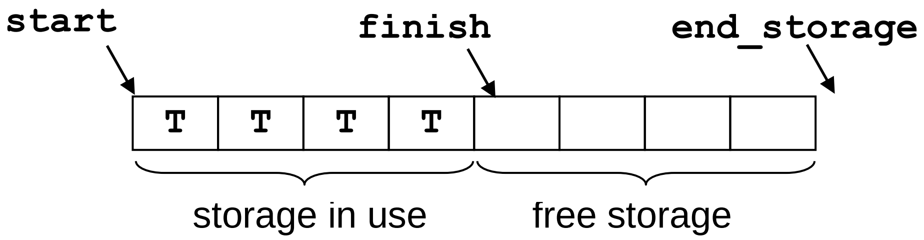
Demo date: 5/31 Lab

STL vector

In this homework, you are asked to implement a portion of STL class template **vector** by yourself. The definitions of the class and most simple member functions are given below.

```
template<typename T>
class vector {
public:
// types
   typedef size_t size_type;
   typedef T& reference;
   typedef const T& const_reference;
   typedef T* iterator;
   typedef const T* const_iterator;
// ctor/copy/dtor
   vector();
   explicit vector(size_type);          // C++11
   vector(size_type,const T&);          // C++11
// explicit
// vector(size_type,const T& =T());     // C++03
   vector(const vector&);
   vector(vector&&);
   vector& operator=(const vector&);
   vector<T>& operator=(vector<T>&&); // <T> optional
   vector(initializer_list<T>);         // <T> required
   ~vector();
// iterators
   iterator begin() { return start; }
   const_iterator begin() const { return start; }
   iterator end() { return finish; }
   const_iterator end() const { return finish; }
   const_iterator cbegin() const     // C++11
   { return start; }
   const_iterator cend() const       // C++11
   { return finish; }

// modifiers
// iterator insert(iterator,const T&);          // C++03
   iterator insert(const_iterator,const T&);    // C++11
   iterator insert(const_iterator,T&&);
// iterator erase(iterator);         // C++03
   iterator erase(const_iterator);   // C++11
   void push_back(const T& val)
   { insert(cend(),val); }
   void push_back(T&& val)
   { insert(cend(),std::move(val)); }
   void pop_back() { erase(cend()-1); }

// capacity
   size_type size() const { return finish-start; }
   size_type capacity() const
   { return end_storage-start; }
   bool empty() const { return size()==0; }

// element access
   reference operator[](size_type n)
   { return start[n]; }
   const_reference operator[](size_type n) const
   { return start[n]; }
   reference back() { return *(end()-1); }
   const_reference back() const { return *(end()-1); }

private:
   T *start,*finish,*end_storage;
};
```

Note that a vector is represented by three pointers:



Part A (70%)

Define all the blue-colored member functions, subject to the storage allocation strategy specified below.

Storage allocation strategy

`vector();`

1 construct an empty vector

2 size() = capacity() = 0

`explicit vector(size_type n);`

1 construct a vector with **`n`** elements initialized with **`T()`**

2 size() = capacity() = **`n`**

`vector(size_type n,const T& val);`

1 construct a vector with **`n`** copies of **`val`**

2 size() = capacity() = **`n`**

`vector(initializer_list<T> init);`

1 construct a vector by copying the elements of the initializer list **`init`** to it
N.B. The elements of an initializer list are constant and shan't be moved.

2 size() = capacity() = **`init`**.size()

Note
Initializer lists are currently supported by g++47 and so you have to test your program under g++47.

`vector<T>::vector(const vector<T>& rhs);`

1 construct a vector by copying the vector `rhs` to it

2 size() = `rhs`.size()
capacity() = `rhs`.capacity()

`vector<T>::vector(vector<T>&& rhs)`

1 construct a vector by moving the resource of vector `rhs` to it

2 empty the vector `rhs`

`vector<T>& operator=(vector<T>&& rhs);`

1 destroy the resource of `*this`

2 move the resource of vector `rhs` to `*this`

3 empty the vector `rhs`

`vector<T>& operator=(const vector<T>& rhs);`

1 capacity() < `rhs`.size()

a) destroy the original vector

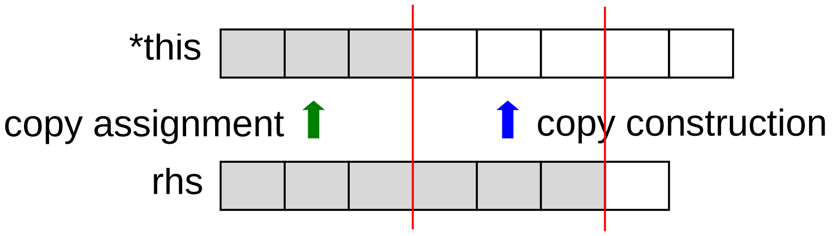
b) allocate a new vector with capacity() = `rhs`.capacity()

c) copy the vector `rhs` to the new vector

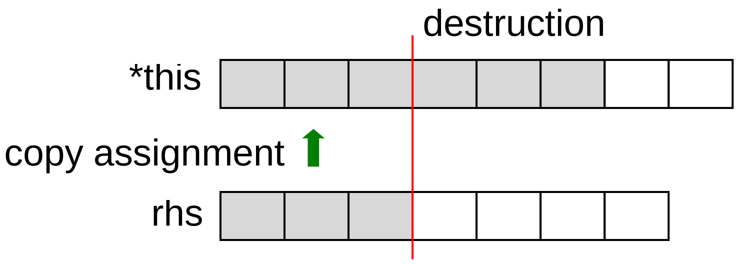
2 capacity() ≥ `rhs`.size()

a) first of all, leave the capacity unchanged.

b) case 1: size() ≤ `rhs`.size()

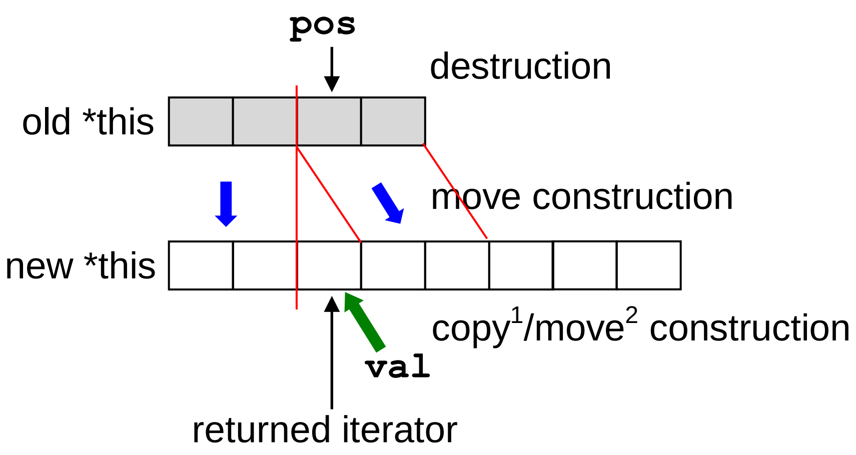


c) case 2: size() > `rhs`.size()

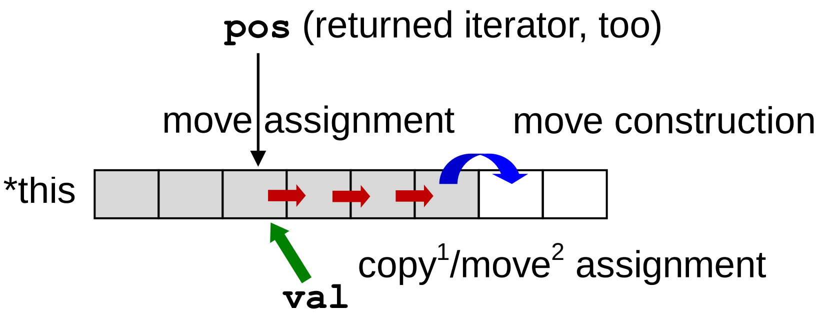


`iterator insert(const_iterator pos,const T& val);` [1]
`iterator insert(const_iterator pos,T&& val);` [2]

1. copy[1]/move[2] **`val`** to the position before **`pos`**
2. return an iterator pointing to the inserted element
3. the behavior is undefined if **`pos`** isn't in the range [begin(),end()]
4. case 1: capacity() = size()
   In this case the vector has no free storage, double its capacity.
   (If capacity = 0, set capacity = 1.)



5. case 2: capacity() > size()

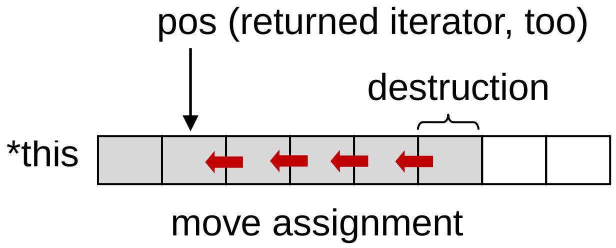


```
iterator erase(const_iterator pos);
```

1 erase the element pointing to by **`pos`**

2 return an iterator pointing to the element immediately following the element just erased;
if the vector becomes empty after the erasion, return end()

3 the behavior is undefined if **`pos`** isn't in the range [begin(),end())



File organization requirement

As before, your program shall be organized in two files.

1 Vector implementation file (say, vector7.h)
This file contains the implementation of the class template **`vector`**.

2 Application file (say, hw7.cpp)
This file includes vector7.h and contains a set of sample tests.

Sample test

Suffice it to run the sample test given in file hw7.cpp, together with the implementation file vector7.h, as follows:

```
bsd2> g++47 -std=c++11 -rpath=/usr/local/lib/gcc47 hw7.cpp
bsd2> ./a.out
```

Sample run

```
Test 1 ...
1 2 3 4 5 6 7 8 9
9 16
8 6 4 2
5 16
```

```
Test 2 ...
5 5
5 5 5 6
7 7

Test 3 ...
Snoopy   copy-constructed
Garfield copy-constructed
Snoopy   copy-constructed
Garfield copy-constructed
Garfield destructed
Snoopy   destructed

Test 4 ...
Snoopy   move-constructed
Pluto    copy-constructed
Garfield move-constructed
 destructed
 destructed

Test 5 ...
Pluto    move-assigned to Snoopy
Garfield move-assigned to Snoopy
Snoopy   destructed
```

Part B (25%)

1 Consider the part-2 test

```
vector<vector<int> > v(2,vector<int>(3,5));
v[0].pop_back();
v[1].push_back(6);
v.push_back(vector<int>(2,7));
```

Draw a picture showing the internal structure of the vector **v**. (10%)

2 Rewrite the nested **`for`** loops of part-2 test by range-based **`for`** loops. (10%)

3 Part 5 output is incorrect, due to the incorrect implementation of the move assignment operator of the **`string`** class by g++47. What should be the correct output? (5%)