```
// Sam Borick <sb205@uakron.edu>

#ifndef STRING_VECTOR_HPP
#define STRING_VECTOR_HPP

#include "string.hpp"
#include "memory.hpp"
#include "test.hpp"

#include <initializer_list>


struct String_vector
{
  String_vector(std::initializer_list<String>);
  String* base = nullptr;
  String* last = nullptr;
  String* limit = nullptr;

  String_vector(){
   // reserve(8);
  }

  String_vector(const String_vector& v);

  String_vector& operator=(const String_vector & v){  //this is a neat optimizat
ion I found on stackoverflow.  I think it's
  //really elegant and now I understand the difference between copy construction
 and copy assingnment better
    String_vector p = v;
    swap(*this, p);
    return *this;
  }

  String& operator[](const size_t pos)const{
    assert(pos >=0);
    assert(pos < size());
    return base[pos];
  }

  ~String_vector();

  void clear();

  size_t size()const;

  void swap(String_vector & v1, String_vector & v2);

  void reserve(std::size_t n);

  void resize(std::size_t n);

  bool empty()const;

  void push_back(String const & s);

  void pop_back();

  size_t capacity()const;

  String const* data();

  using iterator = String*;
  using const_iterator = String*;

  iterator begin(){
    return base;
  }
```



```
  iterator end(){
    return last;
  }

  const_iterator begin()const{
    return base;
  }

  const_iterator end()const{
    return last;
  }

};

bool operator==(String_vector const &, String_vector const &);
bool operator!=(String_vector const &, String_vector const &);
bool operator<(String_vector const &, String_vector const &);
bool operator>(String_vector const &, String_vector const &);
bool operator<=(String_vector const &, String_vector const &);
bool operator>=(String_vector const &, String_vector const &);


#endif
```

```
// Sam Borick <sb205@uakron.edu>

#include "string_vector.hpp"

// Initialize the vector with a brace enclosed sequence of String values.
//
// Example:
//
//    String s {"a", "b", "c"};
String_vector::String_vector(std::initializer_list<String> list)
  : base(), last(), limit()
{
  reserve(list.size());
  for (String const& s : list)
    push_back(s);
}

String_vector:: String_vector(const String_vector& v){
    reserve(v.size());
    base = last;
    last = uninitialized_copy(v.base, v.last, base);
}

 String_vector:: ~String_vector(){
    initialized_destroy(base, last);
    deallocate(base);
  }

void String_vector::reserve(std::size_t n){
    if(!base){
      base = allocate<String>(n);
      last = base;
      limit = n + base;
    }else if(n <= capacity()){
    }else{
      String* p = allocate<String>(n);
      String* q = p;
      for(String*i = base; i != last; ++i){
        new(q)String(*i);
        ++q;
      }
      for(String*i = base; (i==last); ++i){
        i ->~String();
      }
      deallocate<String>(base);
      base = p;
      last = q;
      limit = base + n;
    }
  }

void String_vector::resize(std::size_t n){
    if(n == size()){
    }else if(n < size()){
      //int counter = size() - n;
      for(int counter= size() - n; counter > 0; --counter){
         destroy(--last);
      }
    }else{
      //int counter = n - size();
      for(int counter= n - size(); counter >= 0; --counter){
        push_back("");//yeah, gross
        //TODO: make this better with construct
      }
    }
  }

void String_vector::push_back(String const & s){
    if(!base){
      reserve(8);
    }else if(last == limit){
      reserve(2*capacity());
    }
    construct(last++, s);
  }

void String_vector::pop_back(){
    assert(!empty());
    destroy(--last);
  }

void String_vector::swap(String_vector & v1, String_vector & v2){
    std::swap(v1.base, v2.base);
    std::swap(v1.last, v2.last);
    std::swap(v1.limit, v2.limit);
  }

size_t String_vector:: size()const{
    return last - base;
  }

bool String_vector::empty()const{
    return (base == last);
  }

void String_vector:: clear(){
    resize(0);
  }

size_t String_vector::capacity()const{
    return limit - base;
  }

String const* String_vector::data(){
    return base;
  }

bool operator ==(String_vector const & v1, String_vector const & v2){
 /* std::size_t counter = 0;
  if(v1.size() != v2.size()){
    return false;
  }
  while (counter < v1.size()) {
    if(v1.base+counter != v2.base+counter){
      return false;
    }
    ++counter;
  }
  return true;
  */
  return std::equal(v1.base, v1.last, v2.base);
}

bool operator !=(String_vector const &v1, String_vector const & v2){
  return !(v1==v2);
}

bool operator<(String_vector const &v1, String_vector const & v2){
  return std::lexicographical_compare(v1.base, v1.last, v2.base, v2.last);
}

bool operator>(String_vector const& v1, String_vector const & v2){
  return std::lexicographical_compare(v2.base, v2.last, v1.base, v2.last);
}

bool operator<=(String_vector const& v1, String_vector const & v2){
  return !(v1>v2);
}
```

```
bool operator>=(String_vector const& v1, String_vector const & v2){
  return!(v1<v2);
}
```