```
// $Karthik Venkat <$kv39@zips.uakron.edu>

#ifndef STRING_VECTOR_HPP
#define STRING_VECTOR_HPP

#include "string.hpp"
#include "memory.hpp"
#include "test.hpp"
#include <algorithm>
#include <initializer_list>
using namespace std;

struct String_vector
{
  private:
  String *base; //pointer to starting point of vector
  String *last; //pointer to last object in vector
  String *limit; //pointer to end of vector

  public:
  using iterator = String*;
  using const_iterator = String const*;

  iterator begin()
  {
     return base;
   }
  iterator end()
  {
    return last;
  }
  const_iterator begin() const
  {
    return base;
  }
  const_iterator end() const
  {
    return last;
  }

  //default constructor
  String_vector();
  //initializer_list constructor.
  String_vector(std::initializer_list<String>);
  //Copy constructor
  String_vector(String_vector const&);
  //destructor, release memory.
  ~String_vector();

  //Overloads for subscript operator
  String& operator[](std::size_t n)
  {
    assert(n < size()); return base[n];
  }

  String operator[](std::size_t n) const
  {
    assert(n < size()); return base[n];
  }

  String* data() const
  {
```



```
     return base;
  }
  void reserve(std::size_t n);
  bool empty() const; //Checks if vector is empty
  std::size_t size() const; //Returns last-base
  std::size_t capacity() const; //Returns limit-base
  void push_back(String const&);
  void pop_back();
  void resize(size_t n); //Resizes to size n
  void clear(); //Empties vector

  String_vector& operator = (String_vector const&);

};


//Equivalency and relational operators
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
```

```
// $Karthik Venkat <$kv39@zips.uakron.edu>

#include "string_vector.hpp"

//default constructor
String_vector::String_vector()
  :base(), last(), limit()
{}

//Constructor with initializer list
String_vector::String_vector(std::initializer_list<String> list)
  :base(), last(), limit()
{
  reserve(list.size());
  for (String const& s : list)
    push_back(s);
}

//Copy constructor
String_vector::String_vector(String_vector const& v)
  :base(allocate<String>(v.size())),
  last(uninitialized_copy(v.base, v.limit, base)),
  limit(base + v.size())
{}

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

//Checks if string is empty
bool String_vector::empty() const
{
  if (base == last) return true;
  return false;
}

//Returns last-base for the vector
std::size_t String_vector::size() const
{
  return last - base;
}

//Returns limit-base for the vector
std::size_t String_vector::capacity() const
{
   return limit-base;
}

//Reserves uninitialized memory for a vector
void String_vector::reserve(std::size_t n)
{
  if(n > capacity())
  {
    if(!base)
    {
      base = allocate<String>(n);
      last = base;
      limit = base + n;
    }
    else
    {
      //allocate new memory of size n
      String *p = allocate<String>(n); //new base
      String *q = p; //new last
      limit = p + n; //new limit
      String *i = uninitialized_copy(base, last, q);
      destroy(i);
      deallocate(base);
      base = p; //update base
      last = q; //update last
    }
  }
}

//resize the vector to a size greater than or less than current size
void String_vector::resize(size_t n)
{
  if(n > size()) while(size()!=n) push_back(String("")); //append "" to vec
    while(size()!=n) pop_back(); //pop out from back until new size is reac
  }

//Function to add to vector
void String_vector::push_back(String const& s)
{
  if(!base)
    reserve(20);
  else if(limit == last)
    reserve( 2 * capacity());
  construct(last++, s); //inplace construction
}

//Function to remove from vector
void String_vector::pop_back()
{
  assert(!empty());
  destroy(--last);
}

void String_vector::clear()
{
  initialized_destroy(base, last);
  last = base;
}

//Overload for assignment operator
String_vector& String_vector::operator = (String_vector const& s)
{
  if(this != &s) //Skips this bit if assigned to itself
  {
    clear(); //Clear out the contents
    deallocate(base);
    base = allocate<String>(s.size()); //allocate memory of size of the obj
    last = uninitialized_move(s.begin(), s.limit, begin());
    limit = base + s.size();
  }
  return *this;
}

//Equality overload
bool operator == (String_vector const& a, String_vector const& b)
{
```

```
   if (a.size() == b.size() && std::equal(a.begin(), a.end(), b.begin()))
    return true;
  return false;
 }

//Inequality overload
bool operator != (String_vector const &a, String_vector const &b)
{
  return !(a == b);
}

//Overload for less than
bool operator < (String_vector const &a, String_vector const &b)
{
  return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
}

//Overload for greater than
bool operator > (String_vector const &a, String_vector const &b)
{
   return !std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
 }

 //Overload for less than or equal to
bool operator <= (String_vector const &a, String_vector const &b)
{
   if (a == b || std::lexicographical_compare(a.begin(), a.end(),
  b.begin(), b.end())) //Split for neatness
    return true;
   return false;
}

//Overload for greater than or equal to
bool operator >= (String_vector const &a, String_vector const &b)
{
  if (a == b || !std::lexicographical_compare(a.begin(), a.end(),
  b.begin(), b.end())) //Split for neatness
    return true;
  return false;
}
```