>>>Vec Declaration & Implementation<<<

```
#ifndef Vec_h_
#define Vec_h_
/* This class is implemented using a dynamically allocated
array (of templated type T). We ensure that that m_size is
always <= m_alloc and when a push_back or resize call
would violate this condition, the data is copied to a larger
array. */
template <class T> class Vec {
public:
  // TYPEDEFS
  typedef T* iterator;
  typedef const T* const_iterator;
  typedef unsigned int size_type;
  // CONSTRUCTORS, ASSIGNMNENT OPERATOR, &
DESTRUCTOR
  Vec() { this->create(); }
  Vec(size_type n, const T& t = T()) { this->create(n, t); }
  Vec(const Vec& v) { copy(v); }
  Vec& operator=(const Vec& v);
  ~Vec() { delete [] m_data; }
  // MEMBER FUNCTIONS AND OTHER OPERATORS
  T& operator[] (size_type i) { return m_data[i]; }
  const T& operator[] (size_type i) const { return m_data[i]; }
  void push_back(const T& t);
  iterator erase(iterator p);
  void resize(size_type n, const T& fill_in_value = T());
  void clear() { delete [] m_data;  create(); }
  bool empty() const { return m_size == 0; }
  size_type size() const { return m_size; }
  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
private:
  // PRIVATE MEMBER FUNCTIONS
  void create();
  void create(size_type n, const T& val);
  void copy(const Vec<T>& v);
  // REPRESENTATION
  T* m_data;  // Pointer to first location
  size_type m_size;
  size_type m_alloc;
  m_size <= m_alloc;
};
// Create an empty vector (null pointers everywhere).
template <class T>  void Vec<T>::create() {
  m_data = NULL;
  m_size = m_alloc = 0;  // No memory allocated yet
}
// Create a vector with size n, each location having the given
value
template <class T> void Vec<T>::create(size_type n, const T&
val) {
  m_data = new T[n];
  m_size = m_alloc = n;
  for (T* p = m_data; p != m_data + m_size; ++p)
    *p = val;
}
// Assign one vector to another, avoiding duplicate copying.
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>&
v) {
  if (this != &v) {
    delete [] m_data;
    this -> copy(v);
  }
  return *this;
}
// Create the vector as a copy of the given vector.
template <class T> void Vec<T>::copy(const Vec<T>& v) {
  this->m_alloc = v.m_alloc;
  this->m_size = v.m_size;
  this->m_data = new T[this->m_alloc];

  // Copy the data
  for (size_type i = 0; i < this->m_size; ++i)
    this -> m_data[ i ] = v.m_data[ i ];
}
// Add an element to the end, resize if necesssary.
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    // Allocate a larger array, and copy the old values
    // Calculate the new allocation.  Make sure it is at least one.
    m_alloc *= 2;
    if (m_alloc < 1) m_alloc = 1;
   // Allocate and copy the old array
    T* new_data = new T[ m_alloc ];
    for (size_type i=0; i<m_size; ++i)
      new_data[i] = m_data[i];
    // Delete the old array and reset the pointers
    delete [] m_data;
    m_data = new_data;
  }
  // Add the value at the last location and increment the bound
  m_data[m_size] = val;
  ++ m_size;
}
// Shift each entry of the array after the iterator. Return the
iterator,
// which will have the same value, but point to a different
element.
template <class T> typename Vec<T>::iterator
Vec<T>::erase(iterator p) {
  // remember iterator and T* are equivalent
  for (iterator q = p; q < m_data+m_size-1; ++q)
    *q = *(q+1);
  m_size --;
  return p;
}
/* If n is less than or equal to the current size, just change the
size. If n is greater than the current size, the new slots must be
filled in with the given value. Re-allocation should occur only
if necessary.  push_back should not be used. */
template <class T> void Vec<T>::resize(size_type n, const T&
fill_in_value) {
  if (n <= m_size)
    m_size = n;
  else {
    // If necessary, allocate new space and copy the old values
    if (n > m_alloc) {
      m_alloc = n;
      T* new_data = new T[m_alloc];
      for (size_type i=0; i<m_size; ++i)
        new_data[i] = m_data[i];
      delete [] m_data;
      m_data = new_data;
    }
    // Now fill in the remaining values and assign the final size.
    for (size_type i = m_size; i<n; ++i)
      m_data[i] = fill_in_value;
    m_size = n;
  }
}
#endif
```

++

Erase invalidates **all iterators** after the point of erasure in **vectors**; **push back** and **resize** invalidate **ALL iterators** in a **vector** The value of any associated vector iterator must be re-assigned / re-initialized after these operations.

++

Here are several different ways to initialize a vector:
This "constructs" an empty vector of integers. Values must be placed in the vector using push_back.

```
std::vector<int> a;
```

This constructs a vector of 100 doubles, each entry storing the value 3.14. New entries can be created using push_back, but these will create entries 100, 101, 102, etc.

```
int n = 100;
std::vector<double> b( 100, 3.14 );
```

This constructs a vector of 10,000 ints, but provides no initial values for these integers. Again, new entries can be created for the vector using push_back. These will create entries 10000, 10001, etc.

```
std::vector<int> c( n*n );
```

This constructs a vector that is an exact copy of vector b.

```
std::vector<double> d( b );
```

This is a compiler error because no constructor exists to create an int vector from a double vector. These are different types.

```
std::vector<int> e( b );
```

++

const objects (usually passed into a function as parameters) can ONLY use const member functions. **Remember, you should only pass objects by value under special circumstances. In general, pass all objects by reference so they aren't copied, and by const reference if you don't want/need them to change.**

++

Sorting an Array
Arrays may be sorted using std::sort, just like vectors. Pointers are used in place of iterators. For example, if a is an array of doubles and there are n values in the array, then here's how to sort the values in the array into increasing order:

```
std::sort( a, a+n );
```

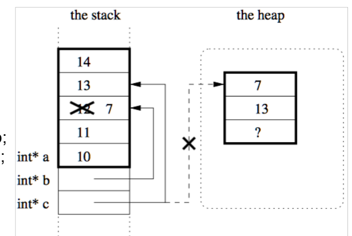
++

Dynamic Allocation of Two-Dimensional Arrays
To store a grid of data, we will need to allocate a top level array of pointers to arrays of the data. For example:

```
double** a = new double*[rows];
for (int i = 0; i < rows; i++) {
  a[i] = new double[cols];
  for (int j = 0; j < cols; j++) {
    a[i][j] = double(i+1) / double (j+1);
  }
}
```

++

Draw a diagram of the heap and stack memory for each segment of code below. Use a "?" to indicate that the value of the memory is uninitialized. Indicate whether there are any errors or memory leaks during execution of this code.

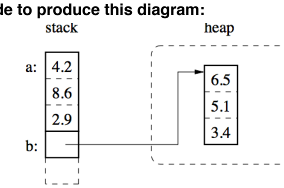



```
class Foo {
public:
   double x;
   int* y;
};
Foo a;
a.x = 3.14159;
Foo *b = new Foo;
(*b).y = new int[2];
Foo *c = b;
a.y = b->y;
c->y[1] = 7;
b = NULL;
int a[5] = { 10, 11, 12, 13, 14 };
int *b = a + 2;
*b = 7;
int *c = new int[3];
c[0] = b[0];
c[1] = b[1];
c = &(a[3]);
```

There is a memory leak of 3 ints in this program.

++

Write code to produce this diagram:




```
double a[3];
double *b = new double[3];
a[0] = 4.2;
a[1] = 8.6;
a[2] = 2.9;
b[0] = 6.5;
b[1] = 5.1;
b[2] = 3.4;
```

++

Opening a New Hair Salon

>>>Customer Class Declaration<<<

```
class Customer {
public:
  // CONSTRUCTOR
  Customer(const std::string& name);
  // ACCESSORS
  const std::string& getName() const;
  const std::string& getStylist() const;
  const Date& lastAppointment() const;
  int numAppointments() const;
  // MODIFIERS
  void hairCut(const Date &d,const std::string &stylist);
private:
  // REPRESENTATION
  std::string customer_name;
  std::string preferred_stylist;
  std::vector<Date> appointments;
};
// helper function for sorting
bool stylist_then_last_appointment(const Customer &c1, const
Customer &c2);
```

++

>>>Customer Class Implementation<<<

```
// CONSTRUCTOR
Customer::Customer(const std::string &name) {
  customer_name = name;
}
// ACCESSORS
const std::string& Customer::getName() const {
  return customer_name;
}
const std::string& Customer::getStylist() const {
  return preferred_stylist;
}
const Date& Customer::lastAppointment() const {
  return appointments.back();
}
int Customer::numAppointments() const {
  return appointments.size();
}
// MODIFIER
void Customer::hairCut(const Date &d,const std::string &stylist) {
  if (stylist != preferred_stylist) {
    std::cout << "Setting " << stylist << " as " << customer_name
<< "'s preferred stylist." << std::endl;
    preferred_stylist = stylist;
  }
  appointments.push_back(d);
}
// COMPARISON FUNCTION FOR SORTING
bool stylist_then_last_appointment(const Customer &c1, const
Customer &c2) {
  return (c1.getStylist() < c2.getStylist() ||
          (c1.getStylist() == c2.getStylist() && c1.lastAppointment() <
c2.lastAppointment()));
}
```

++

Color Analysis for HW1 Images

```
void color_analysis(const std::vector<std::string> &image, int
&num_colors, char &most_frequent_color) {
  // local variables to keep track of colors & counts
  std::vector<char> colors;
  std::vector<int> counts;
  // loop over every pixel in the image
  for (int i = 0; i < image.size(); i++) {
    for (int j = 0; j < image[i].size(); j++) {
      // add each pixel to the color counts
      bool found = false;
      for (int k = 0; k < colors.size() && !found; k++) {
        if (image[i][j] == colors[k]) {
          counts[k]++;
          found = true;
      }
    }
  }
      // if we haven't seen this color before...
      if (!found) {
        colors.push_back(image[i][j]);
        counts.push_back(1);
      }
    }
  }
```

```
  // loop over all of the colors to find the most frequent
  int max_count = 0;
  for (int k = 0; k < colors.size(); k++) {
    if (max_count < counts[k]) {
      max_count = counts[k];
      most_frequent_color = colors[k];
    }
  }
  // also set the num_colors "return value"
  num_colors = colors.size();
}
```

++

O(w * h * c). This function is a simple triply-nested loop. Thus, the order notation is a product of the controlling variables for each loop.

++

Power Matrix Construction

>>>Without pow function<<<

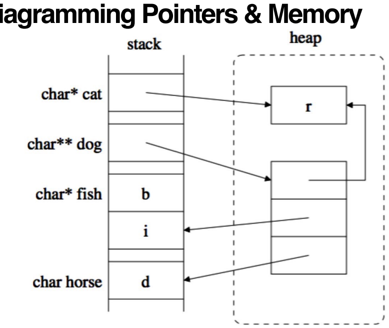
```
std::vector<std::vector<int> > make_power_matrix(int rows, int
cols) {
  std::vector<std::vector<int> > answer;
  for (int r = 0; r < rows; r++) {
    std::vector<int> helper;
    int val = 1;
    for (int c = 0; c < cols; c++) {
      helper.push_back(val);
val *= r; }
    answer.push_back(helper);
  }
  return answer;
}
```

++

To create this 2D vector structure, it will cost **O(r * c)**. Note this is true either using the constructor that creates an array of a specific size or with push back. Keeping a running prod- uct (multiplication) means that it is a constant amount of work per element, even without the pow function.

++

Diagramming Pointers & Memory



++

```
  delete [] dog;
  delete cat;
```

++

Classy Line Slop

>>>Line Class Declaration<<<

```
class Line {
public:
  // CONSTRUCTOR
  Line(const std::string &name, int x1, int y1, int x2, int y2);
  // ACCESSORS
  float getSlope() const;
  float getYIntercept() const;
  const std::string& getName() const;
  // MODIFIERS
  void setNewSecondPoint(int x2, int y2);
private:
  // REPRESENTATION
  std::string name_;
  int x1_,y1_,x2_,y2_;
};
bool by_slope (const Line &a, const Line &b);
```

++

>>>Line Class Implementation<<<

```
Line::Line(const std::string &name, int x1, int y1, int x2, int y2) {
  name_ = name;
  x1_=x1;
  y1_=y1;
  x2_=x2;
  y2_=y2;
  assert (x1_ != x2_);
}
const std::string& Line::getName() const {
  return name_;
}
float Line::getSlope() const {
  int rise = y2_-y1_;
  int run = x2_-x1_;
  return (float)rise/(float)run;
}
float Line::getYIntercept() const {
  float slope = getSlope();
  return y1_ - slope*x1_;
}
void Line::setNewSecondPoint(int x2, int y2) {
  x2_ = x2;
  y2_ = y2;
  assert (x1_ != x2_);
}
bool by_slope (const Line &a, const Line &b) {
  if (a.getSlope() < b.getSlope())
    return true;
  return false;
}
```

++

Detecting Compound Words

```
std::vector<std::string> compound_detector(const
std::vector<std::string> &words) {
  std::vector<std::string> answer;
  // loop over each word, testing to see if it is a compound word
  for (int w = 0; w < words.size(); w++) {
    bool found = false;
    for (int x = 0; !found && x < words.size(); x++) {
      for (int y = 0; !found && y < words.size(); y++) {
        // 2 word combinations
        if (words[w] == words[x]+words[y]) {
          answer.push_back(words[w]);
          found = true;
        }
        for (int z = 0; !found && z < words.size(); z++) {
          // 3 word combinations
          if (words[w] == words[x]+words[y]+words[z]) {
            answer.push_back(words[w]);
            found = true;
          }
        }
      }
    }
  }
  return answer;
}
```

++

To create compound words built from 3 words, we need a triple-nested loop. To see if combination is in the original list, we need another loop. The code above is **O(n^4).**

++

Sorting by Vowels

// HELPER FUNCTIONS

```
int num_vowels(const std::string &a) {
int answer = 0;
for (int i = 0; i < a.size(); i++) {
  if (a[i]=='a'||a[i]=='e'||a[i]=='i'||a[i]=='o'||a[i]=='u')
    answer++;
}
  return answer;
}
bool fewest_vowels(const std::string &a, const std::string &b) {
  int num_vowels_a = num_vowels(a);
  int num_vowels_b = num_vowels(b);
  return (num_vowels_a < num_vowels_b) ||
    (num_vowels_a == num_vowels_b && a < b);
}
// FRAGMENT OF CODE
std::ifstream istr("input.txt");
std::string tmp;
std::vector<std::string> words;
while (istr >> tmp) { words.push_back(tmp); }
sort(words.begin(),words.end(),fewest_vowels);
for (int i = 0; i < words.size(); i++) {
  std::cout << words[i] << "  ";
}
```

++

Navigating the City

>>>Store Location Helper Function<<<

```
bool location(const std::vector<std::vector<std::string> > &city,
const std::string &store_name, int &i, int &j) {
  for (i = 0; i < city.size(); i++) {
    for (j = 0; j < city[i].size(); j++) {
      if (city[i][j] == store_name) return true;
    }
  }
  return false;
}
```

++

>>>Providing Step-by-step Directions<<<

```
void give_directions(const std::vector<std::vector<std::string> >
&city, const std::string &start, const std::string &end) {
  int i,j;
  int end_i,end_j;
  if (!location(city,start,i,j)) {
    std::cerr << "ERROR: cannot find starting point " << start <<
std::endl;
return; }
  if (!location(city,end,end_i,end_j)) {
    std::cerr << "ERROR: cannot find end point " << end <<
std::endl;
    return;
  }
  while (i != end_i) {
    std::cout << "walk from " << city[i][j] << " to ";
    if (i < end_i) i++; else i--;
    std::cout << city[i][j] << std::endl;
  }
  while (j != end_j) {
    std::cout << "walk from " << city[i][j] << " to ";
    if (j < end_j) j++; else j--;
    std::cout << city[i][j] << std::endl;
  }
}
```

++

We need to locate each store, which is a linear scan through all n stores. Then we will take at most n steps in walking between the two stores = **O(n + n + n)**. Final simplified answer = **O(n)**.

++

Min and Max Absolute Value

>>>int main() { <<<

```
  int n;
  std::cin >> n;
  float *data = new float[n];
  int i;
  for (i = 0; i < n; i++) {
    std::cin >> data[i];
  }
  float min;
  float max;
  find_min_and_max(data,n,min,max);
  std::cout << "absolute values: ";
  for (i = 0; i < n; i++) { std::cout << data[i] << " "; }
  std::cout << std::endl;
  std::cout << "min: " << min << std::endl;
  std::cout << "max: " << max << std::endl;
```

++

```
  delete [] data;
```

++

>>>find_min_and_max<<<

```
void find_min_and_max(float data[], int n, float &min, float &max) {
  for (int i = 0; i < n; i++) {
    if (data[i] < 0)
      data[i] = -data[i];
    if (i == 0 || data[i] < min)
      min = data[i];
    if (i == 0 || data[i] > max)
      max = data[i];
  }
}
```

++

Olympic Medal Statistics

>>>OlympicTeam Class Declaration<<<

```
class OlympicsTeam {
public:
  // ACCESSORS
  int numAthletes() const;
  float averageMedalsPerAthlete() const;
  bool hasWonGoldMedal(const std::string& athlete) const;
  // MODIFIERS
  void addAthlete(const std::string &athlete);
  void addMedal(const std::string &athlete, const std::string
&color);
private:
  // REPRESENTATION
  std::vector<std::string> athletes;
  std::vector<std::string> gold;
  std::vector<std::string> silver;
  std::vector<std::string> bronze;
};
```

++

>>>OlympicTeam Class Implementation<<<

```
int OlympicsTeam::numAthletes() const {
  return athletes.size();
}
float OlympicsTeam::averageMedalsPerAthlete() const {
  return (gold.size() +  silver.size() + bronze.size()) / float
(athletes.size());
}
bool OlympicsTeam::hasWonGoldMedal(const std::string&
athlete) const {
  for (int i = 0; i < gold.size(); i++) {
    if (gold[i] == athlete) return true;
  }
  return false;
}
void OlympicsTeam::addAthlete(const std::string &athlete) {
  for (int i = 0; i < athletes.size(); i++) {
    if (athletes[i] == athlete) {
      std::cerr << "ERROR: cannot add duplicate athlete '" <<
athlete << "'" << std::endl;
      return;
    }
  }
  athletes.push_back(athlete);
}
void OlympicsTeam::addMedal(const std::string &athlete, const
std::string &color) {
  bool found = false;
  for (int i = 0; i < athletes.size(); i++) {
    if (athletes[i] == athlete) { found = true; }
  }
  if (found == false) {
    std::cerr << "ERROR: athlete '" << athlete << "' is not a member
of this team" << std::endl;
    return;
  }
  if (color == "gold") {
    gold.push_back(athlete);
  } else if (color == "silver") {
    silver.push_back(athlete);
  } else if (color == "bronze") {
    bronze.push_back(athlete);
  } else {
    std::cerr << "ERROR: unknown medal color '" << color << "'"<<
std::endl;
  }
}
```