CSCI-1200 Data Structures
Test 1 — Practice Problem Solutions

1 Short Answer Round [/ 16]

For each of the following statements, write if it is true or false, and then write 1-2 *complete* sentences explaining why. Most of these statements are false.

1.1 *sizeof()* and Arrays [/4]

True or False Since *sizeof(x)* tells us how much memory the variable `x` takes, we can use *sizeof()* to find out how many elements are in an array.

Solution: False. Arrays are represented by a pointer and may be dynamic, so sizeof() will be a constant size for all arrays, whether they have 0, 10, 500, or some other number of elements. (Remember C-arrays are "dumb".)

1.2 l-values [/4]

True or False 500 can be used as an *l-value*.

Solution: False, you cannot assign a value into a constant (imagine 500 = 200;) ! 500 could certainly be used as an r-value (int x = 500;) .

1.3 Number Types [/4]

True or False The following code will compile and print "true":

```
int x = 5;
float y = 7.2;
x = y;
if(x==7) std::cout << "true";
```

Solution: True, the code will run and due to the float being truncated (decimal part being thrown away) during x=y, x will be 7.

1.4 Vector Usage [/4]

True or False The following code will compile and print "true" *Hint: The fill constructor arguments are correct, and you can assume the correct header files are included.*

```
std::vector<int> x(5,5);
std::vector<float> y(5,7.2);
x=y;
if(x[0]==7) std::cout << "true";
```

Solution: False, because the template types do not match for the two vectors (vector<int> and vector<float>) this will not compile.

2 Image Flood Fill [/ 24]

A popular operation not written in in Homework 1 is "floodfill" ("paint bucket"). floodfill takes a starting position (x,y), where x is the row and y is the column, a vector of strings that is the image, and a fill character. The function will change all pixels with the same value as (x,y) that can be reached by making a path starting from (x,y) and using only adjacent pixels (no diagonals) with the same value. Input pixel values will not be whitespace. Fill in the blanks in the code below to complete the *floodfill()* function.

```
starting_image:          floodfill(0,6,'Z',starting_image):          floodfill(3,2,'Z',starting_image):
   .....XX.                        .....ZZ.                                    .....XX.
   .....XX.                        .....ZZ.                                    .....XX.
   .XXX...X                        .XXX...X                                    .ZZZ...X
   ..XXXX..                        ..XXXX..                                    ..ZZZZ..
   ..X..X..                        ..X..X..                                    ..Z..Z..
```

Solution:

```
void floodfill(int x, int y, char fill_char, std::vector<std::string> &image) {
  // change the specified pixel to a temporary character
  char old_char = image[x][y];
  image[x][y] = ' ';
  // repeated loop over all pixels, looking for neighboring pixels
  while (1) {
    int tmp = 0;
    for (unsigned int i = 0; i < image.size(); i++) {
      for (unsigned int j = 0; j < image[0].size(); j++) {
        if (image[i][j] == old_char) {
          // for any character matching the original character, see if
          // it neighbors a temporarily marked pixel
          if (legal_and_match(image,i+1,j,' ') ||
              legal_and_match(image,i-1,j,' ') ||
              legal_and_match(image,i,j+1,' ') ||
              legal_and_match(image,i,j-1,' ')) {
            tmp++;
            image[i][j] = ' ';
          }
        }
      }
    }
    // if no pixels were changed, break out of the outer loop
    if (tmp == 0) break;
  }
  // replace all temporary pixels with the foreground char
  replace(' ',fill_char,image);
}
```

3 Laundry Baskets [/ 43]

For this problem you will be writing two classes, `Basket` and `Clothing`. The `Basket` holds zero or more pieces of `Clothing` and has a number to identify it. A `Clothing` object has an ID to identify it, and is either dirty or clean. All `Clothing` starts dirty. Here is an example code segment that uses the two classes:

```
Basket b1(1), b2(24);
PrintBasket(b1);
Clothing c1("white socks");
b1.addToBasket(c1);
b2.addToBasket(c1);
PrintBasket(b2);
b1.washClothes();
b1.addToBasket(Clothing("ugly christmas sweater"));
PrintBasket(b1);
PrintBasket(b2);
```

And here is the output:

```
Basket 1 has 0 clothes:
Added white socks to basket 1
Added white socks to basket 24
Basket 24 has 1 clothes:
  white socks (dirty)
Washing white socks
Added ugly christmas sweater to basket 1
Basket 1 has 2 clothes:
  white socks (clean)
  ugly christmas sweater (dirty)
Basket 24 has 1 clothes:
  white socks (dirty)
```

3.1 `Clothing` Declaration (*Clothing.h*) [/ 11]

Start by writing the header file for the `Clothing` class.

Solution:

```
class Clothing{
public:
    Clothing(const std::string& cid);

    const std::string& getID() const { return id; }
    bool isDirty() const { return dirty; }
    bool isClean() const { return !dirty; }

    void Wash();
private:
    std::string id;
    bool dirty;
};
```

3.2 `Clothing` Implementation (*clothing.cpp*) [/ 6]

Next write the implementation of the `Clothing` class.

Solution:

```
#include "Clothing.h"
Clothing::Clothing(const std::string& cid){
    id = cid;
    dirty = true;
}

const std::string& Clothing::getID() const{
    return id;
}

bool Clothing::isDirty() const{
    return dirty;
}

bool Clothing::isClean() const{
    return !isDirty();
}

void Clothing::Wash(){
    if(!dirty){
        std::cerr << id << " already clean" << std::endl;
        return;
    }
    dirty = false;
}
```

3.3 `Basket` Declaration (*Basket.h*) [/ 13]

Next write the header file for the `Basket` class.

Solution:

```
#include "Clothing.h"
class Basket{
public:
    Basket(int num);

    int getNumber() const { return basket_num; }
    const std::vector<Clothing>& getClothes() const { return clothes; }

    void washClothes();
    void addToBasket(const Clothing& c);
private:
    std::vector<Clothing> clothes;
    int basket_num;
};

void PrintBasket(const Basket& b);
```

3.4 `Basket` Member Functions (*basket.cpp*) [/ 8]

Write the implementation of the member functions of the `Basket` class.

Solution:

```
#include "Basket.h"

Basket::Basket(int num){
    basket_num = num;
}

void Basket::washClothes(){
    for(unsigned int i=0; i<clothes.size(); i++){
        if(clothes[i].isDirty()){
            clothes[i].Wash();
            std::cout << "Washing " << clothes[i].getID() << std::endl;
        }
    }
}

void Basket::addToBasket(const Clothing& c){
    clothes.push_back(c);
    std::cout << "Added " << c.getID() << " to basket "
              << basket_num << std::endl;
}
```

3.5 `Basket` Non-Member Functions (*basket.cpp*) [/ 5]

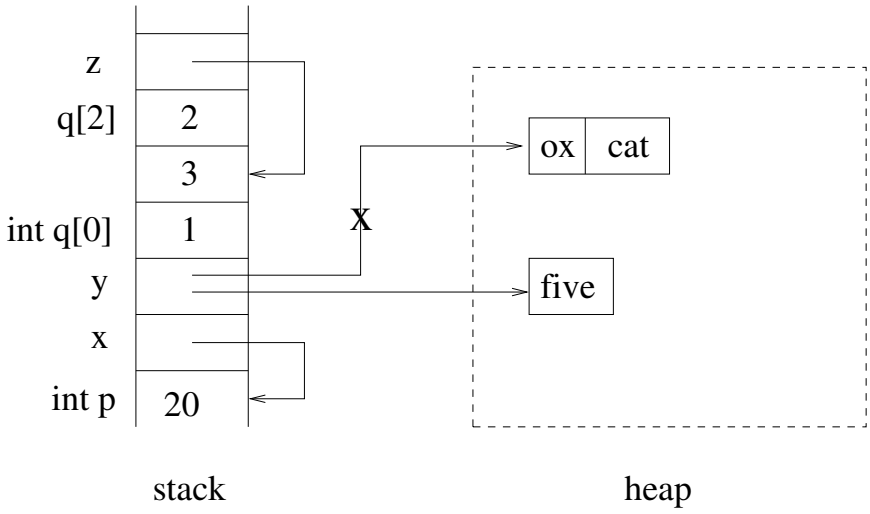
Finally, implement any non-member functions that were declared in *Basket.h* . You do not need to rewrite any `#include` statements written in Section 3.4 .

Solution:

```
void PrintBasket(const Basket& b){
    std::vector<Clothing> clothes = b.getClothes();
    std::cout << "Basket " << b.getNumber() << " has " << clothes.size()
              << " clothes:" << std::endl;
    for(unsigned int i=0; i<clothes.size(); i++){
        std::cout << "  " << clothes[i].getID() << " ";
        if(clothes[i].isClean()){
            std::cout << "(clean)";
        }
        else{
            std::cout << "(dirty)";
        }
        std::cout << std::endl;
    }
}
```

4 Memory Coding [/ 14]

Write code to produce the memory diagram shown on this page. An X over a line denotes a pointer that has been replaced by a more recent pointer. Some types have been left out.



Solution:

```
int p = 20;
int *x = &p;
std::string *y = new std::string[2];
y[0] = "ox";
y[1] = "cat";
y = new std::string;
*y = "five";
int q[3];
q[0] = 1;
q[1] = 3;
q[2] = 2;
int *z = &q[1];

Alternatives:
y = new std::string[1];
y[0] = "five"; //Can do this even without an array
int q[3] = {1,3,2}; //Initialization list
int *z = q+1;
```

After code is run to produce the memory diagram shown above, is it possible to clean up all dynamically allocated memory? If it is not, write 1-2 sentences explaining why. If it is possible, write the code that will clean up the heap.

Solution: It is not possible, because we do not have a pointer to the array of two strings (ox and cat), so this memory will be leaked.

5 Parcel Delivery [/ 35]

In the following problem you will finish the implementation of a program that is designed to keep track of several delivery drivers. Each driver is represented by a `Driver` object which has an ID, a name, a maximum capacity in kg that their vehicle can carry, and the packages they are currently carrying. Each package is represented by a `Parcel` object. For this problem, you can assume that all weights and capacities are integers, and that there will not be duplicate driver IDs.

First, here's `main.cpp` and the output that it produces:

```
#include "Driver.h"
#include "Parcel.h"
// print_drivers implemented, but not included in handout
// add_parcel written in 1.4

int main(){
   std::vector<Driver> drivers;
   drivers.push_back(Driver(124,"Chris",50));
   drivers.push_back(Driver(8,"Sam",150));
   drivers.push_back(Driver(35,"Taylor",200));

   print_drivers(drivers);
   add_parcel(drivers,Parcel("A7X",25),0);
   add_parcel(drivers,Parcel("A7X",25),8);
   add_parcel(drivers,Parcel("S41",126),8);
   add_parcel(drivers,Parcel("AK3",10),35);
   add_parcel(drivers,Parcel("P1",1),124);
   add_parcel(drivers,Parcel("P2",1),124);
   add_parcel(drivers,Parcel("P3",1),124);
   print_drivers(drivers);
   std::sort(drivers.begin(), drivers.end(), bySmallestWeight);
   print_drivers(drivers);
   return 0;
}
```

The output:

```
Driver Chris (#124) is carrying 0 of 50 kgs:
Driver Sam (#8) is carrying 0 of 150 kgs:
Driver Taylor (#35) is carrying 0 of 200 kgs:

Could not find driver #0
Added parcel A7X to driver #8
Failed to add parcel S41 to driver #8
Added parcel AK3 to driver #35
Added parcel P1 to driver #124
Added parcel P2 to driver #124
Added parcel P3 to driver #124
Driver Chris (#124) is carrying 3 of 50 kgs: #P1 (1) kg #P2 (1) kg #P3 (1) kg
Driver Sam (#8) is carrying 25 of 150 kgs: #A7X (25) kg
Driver Taylor (#35) is carrying 10 of 200 kgs: #AK3 (10) kg

Driver Chris (#124) is carrying 3 of 50 kgs: #P1 (1) kg #P2 (1) kg #P3 (1) kg
Driver Taylor (#35) is carrying 10 of 200 kgs: #AK3 (10) kg
Driver Sam (#8) is carrying 25 of 150 kgs: #A7X (25) kg
```

5.1 `Parcel` Class Declaration (`Parcel.h`) [/6]

Start by writing the class declaration in the `Parcel.h` file. The Parcel class should support the constructor used in `main.cpp`, and should have two accessors, *getWeight* and *getID*. For this problem, please do not use constructor initializer lists. You do not need to use include guards. Remember that one line functions can be written in the .h file.

Solution:

```
class Parcel{
public:
   Parcel(const std::string& id, int weight);
   int getWeight() const { return m_weight; }
   const std::string& getID() const { return m_id; }
private:
   int m_weight;
   std::string m_id;
};
```

5.2 `Parcel` Class Implementation (`Parcel.cpp`) [/5]

Now write the class implementation for the Parcel class in `Parcel.cpp`.

Solution:

```
#include "Parcel.h"

Parcel::Parcel(const std::string& id, int weight){
   m_weight = weight;
   m_id = id;
}
```

5.3 Completing the `Driver` Class Implementation (`Driver.cpp`) [/10]

Assume that the implementation of the Driver class is complete except for *getCurrentWeight* and *bySmallestWeight*. The .h file looks like this:

```
#include "Parcel.h"
class Driver{
public:
   Driver(int id, const std::string& name, int capacity);
   int getCapacity() const;
   const std::string& getName() const;
   int getID() const;
   const std::vector<Parcel>& getParcels() const;
   //getCurrentWeight definition would go here. Returns total weight the Driver is carrying.
   bool addParcel(const Parcel& p); //Returns true if parcel was added, false if it was too big.
private:
   int m_id;
   std::string m_name;
   int m_capacity;
   std::vector<Parcel> m_parcels;
};

//bySmallestWeight definition would go here. Used in main.cpp
```

Finish the .cpp file by implementing both of the missing functions:

Solution:

```
int Driver::getCurrentWeight() const{
   int ret = 0;
   for(unsigned int i=0; i<m_parcels.size(); i++){
      ret += m_parcels[i].getWeight();
   }
   return ret;
}

bool bySmallestWeight(const Driver& d1, const Driver& d2){
   return d1.getCurrentWeight() < d2.getCurrentWeight();
}
```

5.4 *add_parcel* Implementation (`main.cpp`) [/14]

Finally, write the *add_parcel* function that goes in `main.cpp`.

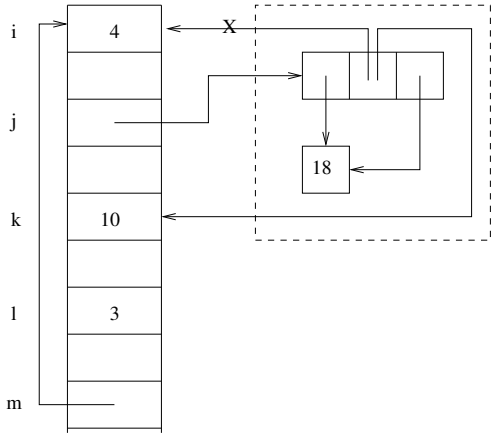
Solution:

```
void add_parcel(std::vector<Driver>& drivers, const Parcel& p, int driver_id){
   for(unsigned int i=0; i<drivers.size(); i++){
      if(drivers[i].getID() != driver_id) { continue; }
      if(drivers[i].addParcel(p)){
         std::cout << "Added parcel " << p.getID() << " to driver #"
                        << driver_id << std::endl;
         return;
      }
      else{
         std::cout << "Failed to add parcel " << p.getID() << " to driver #"
                        << driver_id << std::endl;
         return;
      }
   }
   std::cout << "Could not find driver #" << driver_id << std::endl;
}
```

6 Memory Diagramming [/ 26]

Consider the following code:

```
int i,**j,k,l,*m;
i = 0;
j = new int*[3];
j[0] = new int;
j[1] = &i;
m = *(j+1);
j[1] = &k;
k=10;
*(j[0]) = 5;
j[2] = j[0];
*(j[0]) = 18;
*m = 4;
l = 3;
```



6.1 Memory Diagram [/18]

First, draw a memory diagram for the above code. Do not erase lines for pointers that change, instead put an x over the middle of old line, and then draw the new pointer line.

6.2 Code Output [/8]

Next, write the output running this code will give:

```
std::cout << "i: " << i << std::endl;
std::cout << "j[0]: " << *j[0] << std::endl;
std::cout << "j[1]: " << *j[1] << std::endl;
std::cout << "j[2]: " << *j[2] << std::endl;
std::cout << "k: " << k << std::endl;
std::cout << "l: " << l << std::endl;
std::cout << "m: " << *m << std::endl;
```

Solution:

```
i: 4
j[0]: 18
j[1]: 10
j[2]: 18
k: 10
l: 3
m: 4
```

7 Short Answer Round [/ 16]

For each of the following statements, write if it is true or false, and then write 1-2 *complete* sentences explaining why.

7.1 Return Type [/4]

True or False If we are returning a string, we should always return using `const std::string&`.

Solution: False. If we are returning an automatic variable (non-member) then it will go out of scope so the reference would be invalid.

7.2 const Speed [/4]

True or False Using `const` types changes how fast the program runs.

Solution: False. This can produce compiler errors but has nothing to do with performance. const does not change if we copy a variable or not, and no run-time checking is done.

7.3 Reference Efficiency [/4]

True or False Passing by reference can be more efficient than passing by value.

Solution: True. Passing by reference means we just get a reference (like a pointer) to the original value. This means we don't have to make a second copy (saving time) or store a second copy (saving space).

7.4 const Members [/4]

True or False Every member function should have a `const` at the end of it. (e.g. `int get_var()` **const**).

Solution: False. Sometimes member functions need to alter the class's member variables.

8 Phrase Counting [/ 20]

In this problem you will write a function to count how many times a string appears inside a collection of other strings. Provided below is a code fragment which examines four strings: "banana", "bandana", "cabana", and "banabanbana". For this example, the output of the function is how many times each word had the letters "bana" consecutively in it. In this case, "bandana" has 0 instances of "bana" since the letter d gets in the way.

```
std::vector<std::string> words;
words.push_back("banana");
words.push_back("bandana");
words.push_back("cabana");
words.push_back("banabanabana");

std::vector<int> counts = count_phrase(words,"bana");
for(unsigned int i=0; i<counts.size(); i++){
  std::cout << words[i] << " contains \"bana\" " << counts[i]
            << " time(s)." << std::endl;
}
```

The expected output in this case:

```
banana contains "bana" 1 time(s).
bandana contains "bana" 0 time(s).
cabana contains "bana" 1 time(s).
banabanabana contains "bana" 3 time(s).
```

For this problem, the only STL string function you can use is *size()*. Do not use any C-style string functions (e.g. *strcmp()*).

Your answer should go in the box on the next page.

Write *count_phrase*:

Solution:

```
std::vector<int> count_phrase(const std::vector<std::string>& words, const std::string& phrase){
  std::vector<int> ret(words.size(),0);

  //Check each word
  for(unsigned int i=0; i<words.size(); i++){
    //Go letter by letter for starting position
    for(unsigned int j=0; j<words[i].size(); j++){
      unsigned int k;
      //Check if the substring is found starting at words[i][j+k]
      for(k=0; k<phrase.size() && j+k < words[i].size(); k++){
        if(words[i][j+k] != phrase[k]){
          break;
        }
      }

      //Found the whole phrase
      if(k==phrase.size()){
          ret[i]++;
      }
    }
  }

  return ret;
}
```

9 Movie Recommendations [/35]

If you have used Amazon, Netflix, Pandora, or Last.fm, you are probably familiar with recommendation systems. The idea is that based on some criteria like past user selections, the system will recommend new products. Recommendation systems can be quite sophisticated. In this problem, we will ask you to consider part of a very much simplified system which will determine similar users based on their movie recommendations.

In our system, viewers rate movies on a scale of 1 (terrible) to 5 (excellent). Not all users rated each movie. If a user didn't rate a movie, we will assume that user did not view the movie and the movie's rating is 0. There are many measures of similarity or dissimilarity that could be used. In cases where the data is somewhat sparse, a measure called cosine similarity is sometimes used. To measure the similarity between two users, we use cosine similarity, defined:

$$similarity(x, y) = \frac{x \bullet y}{||x||\,||y||}$$

$x \bullet y$ is the dot product of the ratings of users x and y. It is defined $x \bullet y = \sum_{i=0}^{movies.size()-1} x_i y_i$ where x_i is the rating that user x gave movie number i. That is, the dot product is the sum of the products of the ratings for the two viewers:

$$user_0\_movie_0\_rating \times user_1\_movie_0\_rating + user_0\_movie_1\_rating \times user_1\_movie_1\_rating...$$

$||x|| = \sqrt{\sum_{i=0}^{movies.size()-1} x_i^2}$ is the size of the rating vector. Similarity varies from 0 to 1. Users with similar taste should have a high similarity in their ratings.

The system will require two classes: a class for a movie name and its rating, and a class for movie viewers.

The Rating class is defined as:

```
class Rating {
public:
 Rating(const std::string& name, int rating) : movie_name_(name), rating_(rating)
  {}

  const std::string& getName() const {return movie_name_;}
  int getRating() const {return rating_;}

  void setRating(int rating) {rating_ = rating;}

private:
  std::string movie_name_;
  int rating_;
};
```

9.1 Movie Viewer Class [/11]

First create the class definition for the movie viewer class. The movie viewer class should hold a movie viewer's name and a vector of that viewer's movie ratings, i.e. a vector of *Rating* objects. It must have an accessor function to return the viewers name and a function that is passed a movie name and returns the viewer's numerical rating of that movie. If the viewer did not rate a movie, its rating is zero.

The viewer class must also have a function to add movie ratings. If the viewer has already rated a movie (viewers sometimes rate a movie more than once), the current rating is replaced by the new rating. If the viewer has not yet rated the movie, the new rating is added. You may define helper functions as needed. Don't worry about `#include` or `#define` statements.

Solution:

```
class MovieGoer {
public:
  MovieGoer(const std::string& name) {name_ = name;}

  //Accessors
  const std::string& getName() const {return name_;}
  int  getRating(const std::string& movie_name) const;

  // Modifiers
  void addRating(const Rating& rating);

private:
  std::string name_;
  std::vector<Rating> ratings_;
};
```

9.2 Movie Viewer Class Implementation [/14]

Now implement the constructors, member functions, and any helper functions as they would appear in the `.cpp` file.

Solution:

```
int MovieGoer::getRating(const std::string& movie_name) const {
  for (unsigned int i = 0; i < ratings_.size(); ++i) {
    if (ratings_[i].getName() == movie_name) {
      return ratings_[i].getRating();
    }
  }

  return 0;
}

void MovieGoer::addRating(const Rating& rating) {
  // arg doesn't have to be a Rating object,
  // could be a movie name and rating
  for (unsigned int i = 0; i < ratings_.size(); ++i) {
    if (ratings_[i].getName() == rating.getName()) {
```

```
      ratings_[i].setRating(rating.getRating());
      return;
    }
  }

  ratings_.push_back(rating);
}
```

9.3 `Similarity` Implementation [/10]

Now, implement the Similarity function. This function is not part of the movie viewer or rating class. It is passed two movie viewer objects and a vector of movie names. It should return a double representing the similarity between the two viewer's ratings.

Solution:

```
double similarity(const MovieGoer& mg1, const MovieGoer& mg2,
          const std::vector<std::string>& movies) {
  double n1 = 0;
  double n2 = 0;
  double dot = 0;
  for (unsigned int i = 0; i < movies.size(); ++i) {
    n1 += pow(mg1.getRating(movies[i]), 2);
    n2 += pow(mg2.getRating(movies[i]), 2);
    dot += mg1.getRating(movies[i]) * mg2.getRating(movies[i]);
  }

  return dot/(sqrt(n1) * sqrt(n2));
}
```

10 Parsing Symbols [/20]

A consultant has decided to make a new programming language. This is a pretty complicated task, so they have decided to ask for help from several people, including you. Since you're a Data Structures student, they think you can help them with a task called *parsing*, which is the process of reading in code and breaking it up into pieces the compiler understands. However, even the task of parsing is difficult, so he's asked you to just do a smaller set of problems that are about handling the results of the parser. One job of the parser is matching *opening* symbols like `(`, `[`, {, and *closing* symbols like `)`, `]`, and }.

To make the matching rules easy, the consultant has put all the opening symbols into one `vector` and all the closing symbols into another `vector`. The closing symbol that *matches* opening symbol i (the ith element in the opening symbol vector) is stored in position i of the closing symbol vector. For example, the two vectors for the symbols we've mentioned so far are illustrated below. Keep in mind that they may add more symbols, so you must write code that allows for *any* number of symbols.

```
opening_symbols:  ( { [
closing_symbols:  )  } ]
```

Our goal will be to interpret a *parses* data structure that has the indices of all pairs of matching symbols in a given line of input. We can assume that we're given partially processed input in the form of a `vector` of `string`s. Every entry in the vector is either an opening symbol, a closing symbol, or input we can ignore. The *parses* data structure is a 2-D vector. Each entry in the parses vector is a vector of unsigned integers (positions). The vector of positions always has an even length, with even indices $(0, 2, 4, \ldots)$ containing the position of an opening symbol in the input, and odd indices containing the position of the closing symbol that matched it. Note that each symbol may only match another symbol *once*, so there should be no duplicate entries in the parses vector. Additionally, an opening symbol can only match a closing symbol that is *further right* in the input vector.

As an example using the above `opening_symbols` and `closing_symbols`, the input vector (entries separated by spaces)

```
vector: [ 3 + 41 ] + ( 8 + 9 )  +  (  x  +  y  )  =  [  11 +  50 ]
index:  0 1 2 3  4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

Has a parse vector like:

```
(): 6 10 12 16
{}: <empty>
[]: 0 4 18 22
```

For this problem you may not use `std::find`. You are only allowed to use subscript indexing (e.g. `a[5]`) for accessing arrays and vectors. You should use no pointers and no iterators. You can always assume that the opening symbols and closing symbols two vectors will match in length.

10.1 Printing Output [/12]

The first order of business is to write a function called `PrintParses` that takes in a vector of opening symbols called `opens`, a vector of closing symbols called `closes`, a partially processed input vector, and a matches vector called `parses`. This function should loop through the parses vector, and write the output like shown below to standard output.

The example on the previous page will have the following output:

```
( )
6 10 8 + 9
12 16 x + y
{ }
[ ]
0 4 3 + 41
18 22 11 + 50
```

Solution:

```
//Could also use const std::vector<char>& for opens/closes. Need std::string for input
void PrintParses(const std::vector<std::string>& opens, const std::vector<std::string>& closes,
                 const std::vector<std::string>& input,
                 const std::vector<std::vector<unsigned int> >& parses){
    for(unsigned int i=0; i<opens.size(); i++){
        std::cout << opens[i] << " " << closes[i] << std::endl;
        for(unsigned int j=0; j<parses[i].size(); j+=2){
            std::cout << parses[i][j] << " " << parses[i][j+1];
            for(unsigned int k=parses[i][j]+1; k<parses[i][j+1]; k++){
                std::cout << " " << input[k];
            }
            std::cout << std::endl;
        }
    }
}
```

10.2 Vector Search [/8]

Before tackling the big problem, a more senior programmer on the parsing team has suggested you write a helper function for them called `WordInVector`. This function will take in four arguments: a vector of strings to search through, a string to look for, an unsigned integer we will store the position in, and an unsigned integer we will use as a start index. The function should return `true` if the string we're looking for is in the vector of strings, and the position is not smaller than the start index.

If the word cannot be found in the vector, or can only be found before the start index, the function should return `false` and the behavior of the position variable is undefined; in other words when returning `false`, the position can be set to any value. If the function returns true, the position variable should be set to the index of the first match that is on or after the start index.

For example:

```
//vec contains: Hello world goodbye world

unsigned int pos,pos2,pos3;
WordInVector(vec, "world", pos, 2);
WordInVector(vec, "world", pos2, 1);
WordInVector(vec, "there", pos3, 1);
std::cout << pos << " " << pos2 << " " << pos3 << std::endl;
```

This will print out: 3 1 <any unsigned integer>

Solution:

```
bool WordInVector(const std::vector<std::string>& vec, const std::string& word, unsigned int& position,
                  unsigned int start_position){
    for(unsigned int i=start_position; i<vec.size(); i++){
        if (vec[i] == word){
            position = i;
            return true;
        }
    }
    return false;
}
```

11 Bug Catching [/9]

The fragments of C++ code below contain errors. Your job is to first describe the error in one or two concise and well-written sentences and then provide an appropriate correction that eliminates the error.

```
int* p;
int* q = p;
p = new int;
*p = 55;
std::cout << *q << std::endl;
```

Solution: This code contains a dereference of an uninitialized pointer. This may cause a segmentation fault at runtime, or unexpected output. Add q = p before the cout statement to fix the problem, change *q to *p or put a value in q before new p.

```
std::vector<std::string> > pets;
pets.push_back("cat");
pets.push_back("dog");
pets.push_back("elephant");

std::cout << pets[1] << " " << pets[2] << " " << pets[3] << std::endl;
```

Solution: An attempt was made to reference a vector element that was not allocated. The solution is to reduce each index by 1. There was also extra > on the first line, removing the extra > was another solution.

```
std::cout << pets[0] << " " << pets[1] << " " << pets[2] << std::endl;
```

```
std::vector<std::string>& Vectorfy(const std::string& s) {
  std::vector<std::string> v;
  v.push_back(s);
  return v;
}
```

Solution: The function is returning a reference to a local variable. Return a copy.

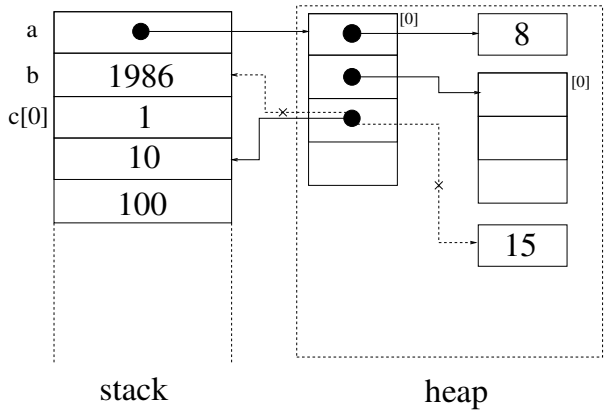
```
std::vector<std::string> Vectorfy(const std::string& s)
```

12 Memory [/24]

Consider the following code:

```
int **a;
a = new int*[4];
a[0] = new int;
a[1] = new int[3];
a[2] = new int;
int b = 25;
int c[3] = {1,10,100};

*(a[2]) = 15;
a[0][0] = 8;
a[2] = &b;
a[2][0] = 1986;
a[2] = &(c[1]);
```



12.1 Memory Diagram [/18]

First, draw a memory diagram for the above code. Do not erase lines for pointers that change, instead put an x over the middle of old line, and then draw the new pointer line.

12.2 Cleaning Up Memory [/6]

Right now the code leaks any memory still allocated. Write 3 `delete` statements that should go immediately after the provided code.

Solution:

```
delete a[0];
delete [] a[1];
delete [] a;
```

Is there another leak? If there is, how would you fix it?

Solution: Yes, `a[2] = &b;` will cause a leak. To fix it, just before this statement add the line of code below.

```
delete a[2];
```