In this problem we will search a large ASCII Art canvas for matches to a target pattern. For example, given the ASCII Art Tic Tac Toe game on the right, we can search for the 'X' symbol on the board. Or the 'O' symbol. Or the height=5 column of '|' characters that make the vertical bar of the Tic Tac Toe board. You will write a function named `search_for_symbols` that takes in 2 arguments, `canvas` and `symbol`. Both inputs are STL vectors of STL strings. The function should return all locations of that pattern or symbol on the canvas. The return type will be an STL vector of positions where each position is an STL vector of integers. Note: Your function program should be general, and work with any size canvas and any size symbol.

In the example below we search for all of the 'O' symbols on the board:

```cpp
std::vector<std::string> symbol;
symbol.push_back(" *** ");
symbol.push_back("* *");
symbol.push_back("* *");
symbol.push_back("* *");
symbol.push_back(" *** ");
std::vector<std::vector<int> > positions = search_for_symbols(tic_tac_toe_canvas,symbol);
for (int i = 0; i < positions.size(); i++) {
    assert (positions[i].size() == 2);
    std::cout << "An O is located at: " << positions[i][0] << "," << positions[i][1] << std::endl;
}
```

And this is the expected output:

```
An O is located at: 0,6
An O is located at: 12,12
```

### 1.1 Assumptions & Error Checking & Realistic Expectations

You will implement the `search_for_symbols` function on the next page. You will not need to do any error checking in your function. But let's pause and consider what assumptions you are making about the input and output. Write 3-4 well-written and concise sentences. You may find it easier to write the function on the next page, then return to answer this part.

Solution: This code will only find matches if all foreground and background characters match exactly between the canvas and the symbol strings. The code also assumes that both the canvas and the symbol are rectangular — that is, the width of each row is the same. Specifically we assume that the empty space at the end of the row is explicitly represented with space characters. The found matching locations may overlap with each other, this may or may not be the intended behavior.
1.2 Now Implement the search_for_symbols Function [  /16]

Solution:

```cpp
std::vector< std::vector<int> > search_for_symbols(const std::vector< std::string > &canvas,
                                                 const std::vector< std::string > &symbol) {
    std::vector< std::vector<int> > answer;
    for (int row = 0; row < canvas.size() - symbol.size() + 1; row++) {
        for (int col = 0; col < canvas[0].size() - symbol[0].size() + 1; col++) {
            bool match = true;
            for (int s_r = 0; s_r < symbol.size() && match == true; s_r++) {
                for (int s_c = 0; s_c < symbol[0].size(); s_c++) {
                    if (canvas[row + s_r][col + s_c] != symbol[s_r][s_c]) {
                        match = false;
                        break;
                    }
                }
            }
            if (match) {
                std::vector<int> position;
                position.push_back(row); position.push_back(col);
                answer.push_back(position);
            }
        }
    }
    return answer;
}
```

2 Stack Sleuthing [ /15]

Carefully examine the contents of the stack after program execution shown on the right. Your task for this problem is to complete the source code on the left so that it produces this stack.

Solution:

```cpp
int foo(int a, int b) {
    int answer;
    if (a < b)
        answer = a*b;
    else
        answer = foo(a-1,b+2);
    return answer;
}

int main() {
    int x = 8;
    int y = 3;
    int tmp = foo(x,y);
    std::cout << tmp << std::endl;
}
```

3 Common C++ Programming Errors [ /12]

For each code fragment below, choose the letter that best describes the program error. Hint: Each letter will be used exactly once.
A ) Accessing data beyond the array bounds
B ) Uninitialized memory
C ) Infinite loop
D ) Compile error: type mismatch
E ) Does not contain an error
F ) Memory leak
G ) Math error (incorrect answer)
H ) Syntax error

```c
int* apple;
int banana[5] = {1, 2, 3, 4, 5};
apple = &banana[2];
*apple = 6;
```

```c
float* floating_pt_ptr = new float;
*floating_pt_ptr = 5.3;
floating_pt_ptr = NULL;
```

```c
float a = 2.0;
float b = -11.0;
float c = 12.0;
float pos_root =
    -b + sqrt(b*b - 4*a*c) / 2*a;
float neg_root =
    -b - sqrt(b*b - 4*a*c) / 2*a;
```

```c
double x;
for (int i = 0; i < 10; i++) {
x += sqrt(double(i));
}
```

```c
std::vector<std::string> temperature;
temperature.push_back(43.5);
unsigned int x;
for (x = 10; x >= 0; x--) {
    std::cout << x << std::endl;
}
```

```
4 Classy Running [ /34]

In this problem you will implement a simple class named RunTracker to keep track of an athlete's running workouts. IMPORTANT: Read through all 3 pages of this problem before beginning your portion of the implementation.

First we create a number of runners, and store their recent runs by distance (in miles):

```c
RunTracker george("George", "Smith");
RunTracker sally("Sally", "Williams");
RunTracker chris("Chris", "Jones");
george.addRun(6.7); sally.addRun(3.1); chris.addRun(2.5); george.addRun(5);
sally.addRun(13.1); chris.addRun(2); george.addRun(9.6); chris.addRun(26.2);
sally.addRun(13.1); george.addRun(11.7);
```

```c
std::vector<RunTracker> runners;
runners.push_back(george); runners.push_back(sally); runners.push_back(chris);
```

We'd like to count how many times each runner ran a specific distance (or more). For example:

```c
std::cout << "Runners who have completed the half marathon distance: " << std::endl;
for (unsigned int i = 0; i < runners.size(); i++) {
    int num = runners[i].numRunsAtLeast(13.1);
    if (num >= 1)
        std::cout << " " << runners[i].getName() << " (" << num << ")" << std::endl;
}
Runners who have completed the half marathon distance:
  Williams, Sally (2)
  Jones, Chris (1)

Finally, we’d like to print all runners, ordered by total distance completed. The following code:

```cpp
for (unsigned int i = 0; i < runners.size(); i++) {
  std::cout << std::setw(18) << std::left << runners[i].getName() << " "
     << std::setw(4) << std::right << std::fixed << std::setprecision(1)
     << runners[i].totalDistance() << std::endl;
}
```

Should result in the following output:

```
Williams, Sally  29.3
Jones, Chris    30.7
Smith, George   33.0
```

4.1 Omitted Line of Code

What is the missing line of code above? It may depend on your answers on the next page(s).

Solution:

```cpp
std::sort(runners.begin(), runners.end(), byTotalDistance);
```

4.2 RunTracker Class Declaration

Using the sample code on the previous pages as your guide, write the class declaration for the RunTracker object. That is, write the header file (run_tracker.h) for this class. You don’t need to worry about the #include lines or other pre-processor directives. Focus on getting the member variable types and member and non-member function prototypes correct. Use const and call by reference where appropriate. Make sure you label what parts of the class are public and private. Save the implementation of all functions for the run_tracker.cpp file, which is the next part.

Solution:

```cpp
class RunTracker {
public:
  // CONSTRUCTOR
  RunTracker(const std::string &fname, const std::string &lname);
  // ACCESSOR
  std::string getName() const;
  float totalDistance() const;
  int numRunsAtLeast(float min) const;
  // MODIFIER
  void addRun(float distance);
private:
  // REPRESENTATION
  std::string fname_; 
  std::string lname_; 
  std::vector<float> runs_; 
};

// HELPER FUNCTION FOR SORTING
bool byTotalDistance(const RunTracker &a, const RunTracker &b);
```
4.3 RunTracker Class Implementation

Now implement all of the functions prototyped in the run_tracker.h file, as they would appear in the corresponding run_tracker.cpp file.

Solution:

// CONSTRUCTOR
RunTracker::RunTracker(const std::string &fname, const std::string &lname) {
    fname_ = fname;
    lname_ = lname;
}

// ACCESSORS
std::string RunTracker::getName() const {
    return lname_ +", " + fname_;}

float RunTracker::totalDistance() const {
    float answer = 0;
    for (int i = 0; i < runs_.size(); i++) { answer += runs_[i]; }
    return answer;
}

int RunTracker::numRunsAtLeast(float min) const {
    int answer = 0;
    for (int i = 0; i < runs_.size(); i++) {
        if (runs_[i] >= min) { answer++; }
    }
    return answer;
}

// MODIFIER
void RunTracker::addRun(float distance) {
    runs_.push_back(distance);
}

// NON-MEMBER HELPER FUNCTION FOR SORTING
bool byTotalDistance(const RunTracker &a, const RunTracker &b) {
    return a.totalDistance() < b.totalDistance();}
The code below is missing its variable declaration & initialization. Fill in these lines so the code compiles and runs without syntax errors, memory errors or leaks. Show your work by diagramming the memory.

```c++
int main() {
    char*** a = new char**;
    char*** b = a;
    *b = new char*[3];
    char** c = (*b);
    c[2] = new char[2];
    char* d[2];
    d[1] = new char;
    c[2][0] = 'R';
    (*a)[0] = NULL;
    (*b)[2][1] = 'P';
    c[1] = NULL;
    *(d[1]) = 'I';
    d[0] = NULL;
    // CREATE
    // MEMORY
    // DIAGRAM
    delete [] (*a)[2];
    delete b;
    delete [] c;
    delete d[1];
}
```

6 Keeping up with not-the-Kardashians

In this problem you will implement a simple class named `Family` to keep track of the children and pets that are members of a family. All members of a family have a common last name. IMPORTANT: Read through all 4 pages of this problem before beginning your portion of the implementation.

Here’s a simple example using the `Family` class:

```cpp
Family king("King");
kimg.addChild("Chris");
kimg.addPet("Buddy");
kimg.addChild("Sally");
std::cout << "The " << king.lastName() << " family has " << king.numChildren() << " children." << std::endl;
if (!king.isPet("Socks")) {
    std::cout << "The family does not have a pet named Socks."
}
kimg.print();
```

And here’s the output from this code:

```
The King family has 2 children.
The family does not have a pet named Socks.
```

We’ll also parse data on the children and pets of multiple families from a file. For example if the input file `family_input.txt` contains:
child Alice Williams
child Ellen Davis
child Frank Jones
pet Garfield Davis
child Henry Williams
pet Mittens Brown
child Ryan Jones
pet Spot Jones
pet Tweety Davis

We will use the Family class to organize, sort, and print this output:

Jones Family
   children: Frank Ryan
   pets: Spot
Williams Family
   children: Alice Henry
Davis Family
   children: Ellen
   pets: Garfield Tweety
Brown Family
   pets: Mittens

Note that the children and pets are grouped by last name. The families with the most children are printed first. Families with the same number of children are ordered by last name.

6.1 Using the Family Class [15]

Complete this fragment of code to read the input file and produce the output on the previous page.

```cpp
std::string filename = "family_input.txt";
std::ifstream istr(filename);
if (!istr.good()) {
    std::cerr << "ERROR: could not open " << filename << std::endl;
    exit(1);
}

Solution:
std::vector<Family> families;
std::string type, first, last;
while (istr >> type >> first >> last) {
    int found;
    for (found = 0; found < families.size(); found++) {
        if (families[found].lastName() == last) {
            break;
        }
    }
    if (found == families.size()) {
        families.push_back(Family(last));
    }
    if (type == "child") {
        families[found].addChild(first);
    } else {
        assert (type == "pet");
        families[found].addPet(first);
    }
}

std::sort(families.begin(), families.end());
for (int i = 0; i < families.size(); i++) {
    families[i].print();
}
```
Using the sample code on the previous pages as your guide, write the class declaration for the Family object. That is, write the header file (family.h) for this class. You don’t need to worry about the #include lines or other preprocessor directives. Focus on getting the member variable types and member and non-member function prototypes correct. Use const and call by reference where appropriate. Make sure you label what parts of the class are public and private. Save the implementation of all functions for the family.cpp file, which is the next part.

Solution:

class Family {
public:
    // CONSTRUCTORS
    Family(const std::string& n);
    // ACCESSORS
    const std::string& lastName() const;
    int numChildren() const;
    bool isPet(const std::string &n) const;
    // MODIFIERS
    void addChild(const std::string& n);
    void addPet(const std::string& n);
    // PRINT
    void print() const;
private:
    // REPRESENTATION
    std::string name;
    std::vector<std::string> children;
    std::vector<std::string> pets;
};

// SORTING HELPER FUNCTION
bool operator< (const Family &a, const Family &b);
6.3 Family Class Implementation

Now implement all of the functions prototyped in the `family.h` file, as they would appear in the corresponding `family.cpp` file. **NOTE:** You may omit the implementation of the `print()` function.

**Solution:**

```cpp
// CONSTRUCTOR
Family::Family(const std::string& n) {
    name = n;
}

// ACCESSORS
const std::string& Family::lastName() const {
    return name;
}
int Family::numChildren() const {
    return children.size();
}
bool Family::isPet(const std::string &n) const {
    for (int i = 0; i < pets.size(); i++) {
        if (pets[i] == n) return true;
    }
    return false;
}

// MODIFIERS
void Family::addChild(const std::string& n) {
    children.push_back(n);
}
void Family::addPet(const std::string& n) {
    pets.push_back(n);
}

// SORTING HELPER FUNCTION
bool operator<(const Family &a, const Family &b) {
    return (a.numChildren() > b.numChildren() ||
            (a.numChildren() == b.numChildren() && a.lastName() < b.lastName()));
}
```

7 Text Justification Redux

Write a function named `print_square` that takes in a single argument, an STL `string`, and reformats that text to fit in the smallest square box, surrounded by a border of stars. Unlike Homework 1, we won't worry about fitting complete words or hyphenation. Just break the words when you get to the end of the row. A few sample calls to the function are shown below, and the output to `std::cout` of each call is shown on the right.

```cpp
print_square("Here is an example.");
print_square("the quick brown fox jumped over the lazy dogs");
print_square("Twinkle, twinkle, little star, how I wonder what you are. Up above the " +
    "world so high, like a diamond in the sky.");
```
Solution:

```cpp
void print_square(const std::string& sentence) {
  // calculate dimensions of smallest square
  int dim = ceil(sqrt(sentence.size()));
  std::cout << std::string(dim+2,'*') << std::endl;
  // helper variable to select next character of the sentence
  int k = 0;
  for (int i = 0; i < dim; i++) {
    std::cout << "*";
    for (int j = 0; j < dim; j++) {
      // make sure we don't attempt to access characters beyond the end of the string
      if (k < sentence.size()) {
        std::cout << sentence[k];
        k++;
      } else {
        std::cout << " ";
      }
    }
    std::cout << "*" << std::endl;
  }
  std::cout << std::string(dim+2,'*') << std::endl;
}
```

8 Memory Diagramming [ /22]

Write code to produce the memory structure shown in the diagram to the right.

Solution:

```cpp
int* a = new int;
*a = 1;
int* b;
int*** c = new int**[2];
c[0] = &b;
c[1] = new int*;
int d[3];
d[0] = 6;
d[1] = 0;
d[2] = 2;
*c[1] = d;
b = &d[1];
```

Write code to print the current year to `std::cout` using ALL of the variables (a, b, c, and d).

Solution:

```cpp
std::cout << d[2] << *b << *a << **c[1] << std::endl;
```

Finally, write code to clean up the dynamically-allocated memory so we don't have any leaks.

Solution:

```cpp
delete a;
delete c[1];
delete [] c;
```
9 HasLetter [ /12]

For this problem you will write a function named HasLetter that accepts 3 arguments named words, letter, and selected. The function should examine the strings in the words vector, collecting all strings that contain the character letter in the selected vector.

If words contains these 8 words: dog bird cat fish turtle horse goat hedgehog
Then after executing this command:

```cpp
HasLetter(words,'r',selected);
```

The selected vector will contain: bird turtle horse
If we then execute:

```cpp
HasLetter(words,'o',selected);
```

Now the selected vector will contain: dog horse goat hedgehog

Solution:

```cpp
void HasLetter(const std::vector<std::string> &words, char letter, std::vector<std::string> &selected) {
    // clear out the vector of any previous answer
    selected.clear();
    for (int i = 0; i < words.size(); i++) {
        // use a boolean to check for the letter
        // (in case there are repeated letters)
        bool flag = false;
        for (int j = 0; j < words[i].size(); j++) {
            if (words[i][j] == letter) {
                flag = true;
                break;
            }
        }
        if (flag)
            selected.push_back(words[i]);
    }
}
```

10 Loading the Dishwasher [ /33]

In this problem you will implement a simple class named Dishwasher to keep track of the contents of a dishwasher as it’s being loaded with plates, cups, forks, knives, and spoons. The total volume of a Dishwasher is measured in number of plates. The diagram on the right shows a dishwasher that has room to hold 7 plates. If we want to wash a mix of plates and cups, we can load 3 cups in the space normally occupied by 2 plates. Note: There is no limit on space for forks, knives, or spoons. We never run out of space for utensils!

Here’s how we create a new dishwasher object, specifying the volume in number of plates:

```cpp
Dishwasher dw(7);
```

And here’s how we begin to load the dishwasher. Note that we perform error checking each time we add a cup or plate. None of the error checks in this portion of the example are triggered.

```cpp
if (!dw.addPlate("blue")) { std::cerr << "ERROR 1: cannot add another plate" << std::endl; }
if (!dw.addPlate("red")) { std::cerr << "ERROR 2: cannot add another plate" << std::endl; }
if (!dw.addPlate("green")) { std::cerr << "ERROR 3: cannot add another plate" << std::endl; }
if (!dw.addCup()) { std::cerr << "ERROR 4: cannot add another cup" << std::endl; }
if (!dw.addPlate("green")) { std::cerr << "ERROR 5: cannot add another plate" << std::endl; }
if (!dw.addPlate("red")) { std::cerr << "ERROR 6: cannot add another plate" << std::endl; }
if (!dw.addCup()) { std::cerr << "ERROR 7: cannot add another cup" << std::endl; }
dw.addSpoon(); dw.addFork(); dw.addKnife(); dw.addFork(); dw.addCup();
dw.addFork(); dw.addFork(); dw.addKnife(); dw.addSpoon(); dw.addFork();
```
Next, we print the contents of the dishwasher and the number of complete sets of utensils (1 fork + 1 knife + 1 spoon). It seems like we always have extra forks!

```cpp
dw.printContents();
std::cout << dw.completeUtensilSets() << " complete utensil set(s)" << std::endl;
```

Here is the output (to `std::cout`) after the above statements. Note that we print the colors of the plates in the order they were inserted.

```
5 plate(s): blue red green green red
2 cup(s)
10 utensil(s)
2 complete utensil set(s)
```

Finally, let’s explore what happens if we try to load more cups & plates into the dishwasher...

```cpp
if (!dw.addPlate("red")) { std::cerr << "ERROR 8: cannot add another plate" << std::endl; }
if (!dw.addCup()) { std::cerr << "ERROR 9: cannot add another cup" << std::endl; }
if (!dw.addCup()) { std::cerr << "ERROR 10: cannot add another cup" << std::endl; }
```

Two of these additions fail, as we can see by this output to `std::cerr`:

```
ERROR 8: cannot add another plate
ERROR 10: cannot add another cup
```

### 10.1 Dishwasher Class Declaration

Using the sample code on the previous page as your guide, write the class declaration for the `Dishwasher` object. That is, write the header file (`dishwasher.h`) for this class. You don’t need to worry about the `#include` lines or other pre-processor directives. Focus on getting the member variable types and member function prototypes correct. Use `const` and call by reference where appropriate. Make sure you label what parts of the class are `public` and `private`. Save the implementation of all functions > 1 line for the `dishwasher.cpp` file, which is the next part.

**Solution:**

```cpp
class Dishwasher {
public:
    // CONSTRUCTOR
    Dishwasher(int max_plates);
    // PRINT & ACCESSORS
    void printContents() const;
    int completeUtensilSets() const;
    // MODIFIERS
    bool addPlate(const std::string& color);
    bool addCup();
    void addFork() { forks++; }
    void addSpoon() { spoons++; }
    void addKnife() { knives++; }
private:
    // PRIVATE HELPER FUNCTION
    bool valid_counts(int p, int c) const;
    // REPRESENTATION
    int max_plates;
    std::vector<std::string> plates;
    int cups;
    int forks;
    int knives;
    int spoons;
};
```

### 10.2 Dishwasher Class Implementation

Now implement the member functions, as they would appear in the corresponding `dishwasher.cpp` file.

**Solution:**

```cpp
// CONSTRUCTOR
```
Dishwasher::Dishwasher(int max_p) {
    cups = forks = knives = spoons = 0;
    max_plates = max_p;
}

// PRINT & ACCESSORS
void Dishwasher::printContents() const {
    std::cout << plates.size() << " plate(s):";
    for (int i = 0; i < plates.size(); i++) { std::cout << " " << plates[i]; }
    std::cout << std::endl;
    std::cout << cups << " cup(s)" << std::endl;
    std::cout << forks+knives+spoons << " utensil(s)" << std::endl;
}

int Dishwasher::completeUtensilSets() const {
    return std::min(std::min(forks,knives),spoons);
}

// MODIFIERS
bool Dishwasher::addPlate(const std::string& color) {
    if (valid_counts(plates.size()+1,cups)) {
        plates.push_back(color);
        return true;
    }
    return false;
}

bool Dishwasher::addCup() {
    if (valid_counts(plates.size(),cups+1)) {
        cups++;
        return true;
    }
    return false;
}

// PRIVATE HELPER FUNCTION
bool Dishwasher::valid_counts(int p, int c) const {
    return (p + ceil(c/3.0)*2 <= max_plates);
}

11 Sorting L33T5P34K (a.k.a. LEETSPEAK) [15]

We saw in Homework 2 that the default sorting of STL string objects places numerical digits (0-9) before capital letters (A-Z). Therefore, following the STL defaults, this sequence of strings is considered sorted:

3ND 4ND 5L0 /P 5L1P 5L3PT 5L4P D0 /G D1G

Let’s instead view these strings as English words encoded by 1980’s hackers or texting teenagers in so-called LEETSPEAK, using these substitutions: 0↔O, 1↔I, 3↔E, 4↔A, and 5↔S. Therefore, this is a more appropriate alphabetization of these strings:

4ND D1G D0 /G 3ND 5L4P 5L3PT 5L1P 5L0 /P

Implement a function named leetspeak_sorter, that may be passed to the STL sort routine for STL vectors to alphabetize a collection of LEETSPEAK words. You may assume all characters of these words are either capital letters or numbers and use only the letter substitutions listed above.

Solution:

std::string leetspeak(const std::string &w) {
    std::string answer = w;
    for (int i = 0; i < w.size(); i++) {
        if (w[i] == '4') answer[i] = 'A';
        if (w[i] == '3') answer[i] = 'E';
        if (w[i] == '1') answer[i] = 'I';
        if (w[i] == '0') answer[i] = 'O';
        if (w[i] == '5') answer[i] = 'S';
    }
    return answer;
}
return answer;
}

bool leetspeak_sorter(const std::string &a, const std::string &b) {
    return leetspeak(a) < leetspeak(b);
}

12 Memory Diagramming [ /20]

Write code to produce the memory structure shown in the diagram to the right.

Some types have been omitted (marked with “???”).

Solution:

class Foo {
public:
    Foo(char* l);
private:
    char* letter;
    int values[2];
};

Foo::Foo(char *l) {
    static int id = 1;
    letter = l;
    values[0] = id;
    values[1] = id+1;
    id += 2;
}

int main() {
    char* w = new char;
    *w = 'a';
    Foo x(w);
    char y = 'b';
    Foo* z = new Foo(&y);
}