Note: This packet contains selected practice problems from Test 1 from three previous years. Your test will contain approximately one third to one half as many problems (totalling ~100 pts).

1 Searching for Symbols in ASCII Art [ / 21 ]

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 [ /5 ]

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.
1.2 Now Implement the `search_for_symbols` Function

*sample solution: 22 line(s) of code*
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.

```cpp
1 { 

2

3 if ( < ) 

4 

5 else 

6 

7 

8 } 

9 () { 

10 function main

11 x: 8

12 y: 3

13 tmp: 42

14 b: 3

15 a: 8

16 return address main@line13

function foo

17 answer: 42

18 b: 5

19 a: 7

20 return address foo@line6

function foo

21 answer: 42

22 b: 7

23 a: 6

24 return address foo@line6

function foo

25 answer: 42

26 std::cout << tmp << std::endl; 

27 }

28 }
```
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

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

std::vector<std::string> temperature;
temperature.push_back(43.5);

float* floating_pt_ptr = new float;
floating_pt_ptr = NULL;

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;

int balance = 100;
int withdrawal;
std::cin >> withdrawal;
if (withdrawal <= balance)
    balance -= withdrawal;
else
    std::cout << "failure"

```
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):

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

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:

```cpp
std::cout << "Runners who have completed the half marathon distance: \n";
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 << ")";
}
```

Should result in this output:

```plaintext
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
/* ONE LINE OF CODE OMITTED: YOU WILL FILL IN THIS LINE IN PART 1 */

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;
}

std::cout << std::endl;
```

Should result in the following output:

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

4.1 Omitted Line of Code [  /4]

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

sample solution: 13 line(s) of code
4.3 RunTracker Class Implementation [ /16]

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

*sample solution: 25 line(s) of code*
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() {
    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");
king.addChild("Chris");
king.addPet("Buddy");
king.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." << std::endl;
}
king.print();
```

And here’s the output from this code:

```
The King family has 2 children.
The family does not have a pet named Socks.
King Family
   children: Chris Sally
   pets: Buddy
```

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.
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);
}

std::sort(families.begin(), families.end());
for (int i = 0; i < families.size(); i++) {
    families[i].print();
}
```

Sample solution: 19 line(s) of code
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 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 `family.cpp` file, which is the next part.

**Sample solution:** 15 line(s) of code
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.*
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.

```
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.");
```
Write code to produce the memory structure shown in the diagram to the right.

*sample solution: 12 line(s) of code*

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

*sample solution: 1 line(s) of code*

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

*sample solution: 3 line(s) of code*
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:

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

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

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

Now the `selected` vector will contain: dog horse goat hedgehog

*sample solution: 14 line(s) of code*
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.addSpoon();
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 [ /15]

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.

sample solution: 19 line(s) of code
10.2 Dishwasher Class Implementation

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

Sample solution: 31 line(s) of code
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 5LØP 5L1P 5L3PT 5L4P DØG 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 DØG 3ND 5L4P 5L3PT 5L1P 5LØP

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.

sample solution: 14 line(s) of code
Write code to produce the memory structure shown in the diagram to the right.

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

Sample solution: 21 line(s) of code