CSCI-1200 Data Structures
Test 1 — Practice Problems

Note: This packet contains selected practice problems from three previous exams. Your exam will contain approximately one third to one half as many problems.

1 Opening a New Hair Salon [/32]

In this problem you will implement a simple class named `Customer` to keep track of customers at a hair salon. First, we create 6 `Customer` objects:

```
Customer betty("Betty");     Customer chris("Chris");     Customer danielle("Danielle");
Customer erin("Erin");       Customer fran("Fran");       Customer grace("Grace");
```

Then, we can track customers as they come to the salon for appointments on specific dates with one of the salon's stylists. We use the `Date` class we discussed in Lecture 2. You may assume the appointments are entered chronologically, with increasing dates.

```
betty.hairCut   (Date(1,15,2015), "Stephanie");
chris.hairCut   (Date(1,17,2015), "Audrey"   );
grace.hairCut   (Date(1,20,2015), "Stephanie");
danielle.hairCut(Date(1,28,2015), "Stephanie");
chris.hairCut   (Date(2, 5,2015), "Audrey"   );
betty.hairCut   (Date(2, 9,2015), "Stephanie");
fran.hairCut    (Date(2,12,2015), "Audrey"   );
danielle.hairCut(Date(2,18,2015), "Lynsey"   );
betty.hairCut   (Date(2,20,2015), "Stephanie");
```

In the system, each customer record will store the customer's preferred stylist. The preferred stylist is defined as a customer's most recent stylist. A message is printed to `std::cout` on each customer's first visit to the salon, or if a customer switches to a new stylist. Here is the output from the above commands:

```
Setting Stephanie as Betty's    preferred stylist.
Setting Audrey    as Chris's    preferred stylist.
Setting Stephanie as Grace's    preferred stylist.
Setting Stephanie as Danielle's preferred stylist.
Setting Audrey    as Fran's     preferred stylist.
Setting Lynsey    as Danielle's preferred stylist.
```

Next, we insert the customers into an STL `vector`:

```
std::vector<Customer> customers;
customers.push_back(betty);  customers.push_back(chris);  customers.push_back(danielle);
customers.push_back(erin);   customers.push_back(fran);   customers.push_back(grace);
```

And then sort & print them first alphabetically by stylist, and secondarily by most recent visit to the salon:

```
std::sort(customers.begin(),customers.end(),stylist_then_last_appointment);
for (int i = 0; i < customers.size(); i++) {
   std::cout << customers[i].getName() << " has had "
             << customers[i].numAppointments() << " appointment(s) at the salon";
   if (customers[i].numAppointments() > 0) {
      std::cout << ", most recently with " << customers[i].getStylist()
                << " on " << customers[i].lastAppointment(); }
   std::cout << "." << std::endl;
}
```

Which results in this output to the screen:

```
Erin     has had 0 appointment(s) at the salon.
Chris    has had 2 appointment(s) at the salon, most recently with Audrey    on  2/ 5/2015.
Fran     has had 1 appointment(s) at the salon, most recently with Audrey    on  2/12/2015.
Danielle has had 2 appointment(s) at the salon, most recently with Lynsey    on  2/18/2015.
Grace    has had 1 appointment(s) at the salon, most recently with Stephanie on  1/20/2015.
Betty    has had 3 appointment(s) at the salon, most recently with Stephanie on  2/20/2015.
```

Note: Don't worry about output formatting/spacing. You may assume that the `Date` class has an `operator<` to compare/sort dates chronologically and an `operator<<` to print/output `Date` objects.

1.1 `Customer` Class Declaration [/15]

Using the sample code on the previous page as your guide, write the class declaration for the `Customer` object. That is, write the *header file* (`customer.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`. Include prototypes for any related non-member functions. Save the implementation of all functions for the `customer.cpp` file, which is the next part.

sample solution: 14 line(s) of code

1.2 `Customer` Class Implementation [/17]

Now implement the member functions and related non-member functions of the class, as they would appear in the corresponding `customer.cpp` file.

sample solution: 26 line(s) of code

2 Color Analysis of HW1 Images [/21]

Write a function named `color_analysis`, that takes three arguments: `image`, an STL `vector` of STL `string`s representing a rectangular ASCII image (similar to HW1); an integer `num_colors`; and a character `most_frequent_color`. The function scans through the image and returns (through the 2nd & 3rd arguments) the number of different colors (characters) in the image & the most frequently appearing color.

sample solution: 27 line(s) of code

What is the order notation of your solution in terms of w & h, the width & height of the image, and c, the number of different colors in the image?

3 Power Matrix Construction [/16]

Write a function named `make_power_matrix` that takes in two arguments, `num_rows` and `num_columns`, and creates and returns a 2D matrix using STL `vector`s. Each element of the matrix $m_{r,c}$ stores the value r^c, that is, the row index raised to the power of the column index. For example, `make_power_matrix(5,7)` should produce this matrix:

```
1    0    0    0    0    0    0
1    1    1    1    1    1    1
1    2    4    8   16   32   64
1    3    9   27   81  243  729
1    4   16   64  256 1024 4096
```

Try to write this function *without using* the `pow` function.

sample solution: 13 line(s) of code

What is the order notation of your solution in terms of r, the number of rows, and c, the number of columns?

4 Diagramming Pointers & Memory [/15]

In this problem you will work with pointers and dynamically allocated memory. The fragment of code below allocates and writes to memory on both *the stack* and *the heap*. Following the conventions from lecture, draw a picture of the memory after the execution of the statements below.

```
char* cat;
char** dog;
char fish[2];
char horse;
dog = new char*[3];
dog[0] = new char;
fish[0] = 'b';
fish[1] = 'i';
dog[1] = &fish[1];
dog[2] = &horse;
cat = dog[0];
*cat = 'r';
horse = 'd';
```

Now, write a fragment a C++ code that cleans up all dynamically allocated memory within the above example so that the program will not have a memory leak.

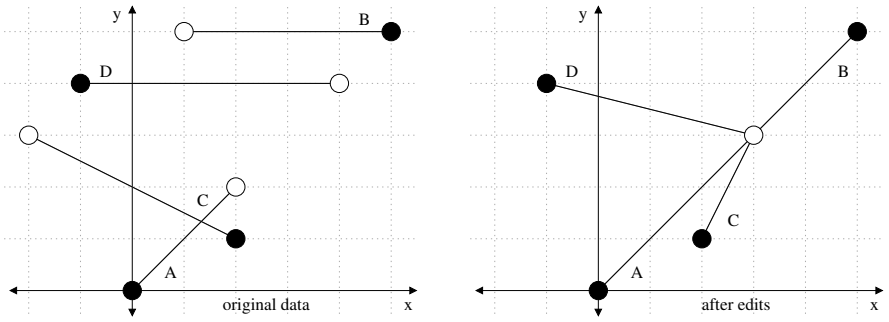
5 Classy Line Slopes [/28]

In this problem you will implement a simple class named `Line` to keep track of two dimensional lines. `Line`s are defined by two endpoints with integer coordinates. We will calculate the slope and y-axis intercept of each of the lines. Remember from algebra/geometry class that the equation for a line is $y = mx + b$, where m is the slope and b is the y-intercept. You may assume that the two endpoints are not the same point, and that the line is not exactly vertical (which would correspond to slope=∞).

In the example below we make four `Line` objects and put them in an STL `vector`. The diagram on the left shows the original position of each of the lines – the black dot is the "first" endpoint and the white dot is the "second" endpoint. The `Line` class allows us to edit the second endpoint of each line, as shown in the diagram on the right, and in the code below.

```
Line a ("A",   0, 0,   2, 2);
Line b ("B",   5, 5,   1, 5);
Line c ("C",   2, 1,  -2, 3);
Line d ("D",  -1, 4,   4, 4);

std::vector<Line> lines;
lines.push_back(a);
lines.push_back(b);
lines.push_back(c);
lines.push_back(d);
```



Here's a helper function that outputs information about each line stored in a `vector`:

```
void printLines(const std::vector<Line> &lines) {
  for (int i = 0; i < lines.size(); i++) {
    std::cout << "Line " << lines[i].getName()
              << std::fixed << std::setprecision(2)
              << " with slope=" << std::setw(5) << lines[i].getSlope()
              << " and y intercept=" << std::setw(5) << lines[i].getYIntercept()
              << std::endl;
  }
}
```

Here's a code fragment that will first sort the line collection by slope and then print the lines:

```
std::cout << "original data, sorted by slope" << std::endl;
sort(lines.begin(),lines.end(),by_slope);
printLines(lines);
```

Now we edit the second endpoint of each line to be the point (3,3) and then sort and print the data again:

```
for (int i = 0; i < lines.size(); i++) {
  lines[i].setNewSecondPoint(3,3);
}
std::cout << "after changing second point to (3,3)" << std::endl;
sort(lines.begin(),lines.end(),by_slope);
printLines(lines);
```

The code above results in this output to the screen:

```
original data, sorted by slope
Line C with slope=-0.50 and y intercept= 2.00
Line D with slope= 0.00 and y intercept= 4.00
Line B with slope= 0.00 and y intercept= 5.00
Line A with slope= 1.00 and y intercept= 0.00
after changing second point to (3,3)
Line D with slope=-0.25 and y intercept= 3.75
Line B with slope= 1.00 and y intercept= 0.00
Line A with slope= 1.00 and y intercept= 0.00
Line C with slope= 2.00 and y intercept=-3.00
```

5.1 `Line` Class Declaration [/13]

Using the sample code on the previous page as your guide, write the class declaration for the `Line` object. That is, write the *header file* (`line.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`. Include prototypes for any related non-member functions. Save the implementation of all functions for the `line.cpp` file, which is the next part.

sample solution: 12 line(s) of code

5.2 `Line` Class Implementation [/15]

Now implement the member functions and related non-member functions of the class, as they would appear in the corresponding `line.cpp` file.

sample solution: 28 line(s) of code

6 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) Uninitialized memory
B) Compile error: type mismatch
C) Accessing data beyond the array bounds
D) Infinite loop
E) Math error (incorrect answer)
F) Memory leak
G) Syntax error
H) Does not contain an error

[ ]
```
float* floating_pt_ptr = new float;
*floating_pt_ptr = 5.3;
floating_pt_ptr = NULL;
```

[ ]
```
unsigned int x;
for (x = 10; x >= 0; x--) {
  std::cout << x << std::endl;
}
```

[ ]
```
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);
```

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

[ ]
```
int balance = 100;
int withdrawal;
std::cin >> withdrawal;
if (withdrawal <= balance)
  balance -= withdrawal;
  std::cout << "success\n";
else
  std::cout << "failure\n";
```

[ ]
```
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 grades[4] = { 1, 2, 3, 4 };
std::cout << "grades" << grades[1]
          << " " << grades[2]
          << " " << grades[3]
          << " " << grades[4]
          << std::endl;
```

7 Detecting Compound Words [/18]

Write a C++ function that takes in a collection of English words stored as an STL `vector` of STL `strings`. The function should return a `vector` containing all *compound words* from the input collection. We define a compound word as two or three words joined together to make a different word. For example, given the input collection:

```
a back backlog backwoods backwoodsman cat catalog
less log man none nonetheless ship the woods woodsman
```

Your function should return (in any order):

```
backlog backwoods backwoodsman catalog nonetheless woodsman
```

sample solution: 21 line(s) of code

If there are n words in the input, what is the order notation of your solution?

8 Sorting by Vowels [/20]

Write a fragment of C++ code to read all the words stored in a file named "`input.txt`". You should then print the words to `std::cout` sorted by the number of vowels ('`a`', '`e`', '`i`', '`o`', or '`u`') that each contains, fewest first. If 2 words have the same number of vowels, output them in normal alphabetical order ('a' before 'z'). You may assume that all words in the file contain only lowercase letters. For example, if `input.txt` contains:

```
aspire  dog  zoological  cat  meow  grrr  utopia  audio
```

Your program should output:

```
grrr  cat  dog  meow  aspire  audio  utopia  zoological
```

You should write one or more simple helper functions as part of your solution.

sample solution: 21 line(s) of code

9 Navigating the City [/23]

Write a function named `give_directions` that takes in three arguments. The first argument is an STL `vector` of STL `vector`s of STL `string`s named `city` that contains the city block layout of stores in downtown. The city is a regular square grid, and each city block has exactly one store. Here is an example of the data stored in a city with 20 blocks (each row & column are labeled with numbers for clarity):

```
   0           1             2           3
0  bestbuy     lenscrafters  walmart     gap
1  claires     payless       starbucks   oldnavy
2  brookstone  teavana       sears       macys
3  cvs         tmobile       lowes       footlocker
4  dsw         jcrew         apple       zales
```

The second and third arguments to the function are the names of two stores that may or may not be in the grid. If both stores are in the city grid, your function should print to `std::cout` a set of directions to help someone walk through the city starting at the store named by the second argument and ending at the store named by the third argument. For example:

```
give_directions(city,"cvs","oldnavy");
```

Should result in this output:

```
walk from cvs to brookstone
walk from brookstone to claires
walk from claires to payless
walk from payless to starbucks
walk from starbucks to oldnavy
```

In each step of the directions we walk to one of the 4 adjacent neighboring city blocks. We cannot directly walk to one of the diagonal blocks. Also, there is usually more than one solution. Your program may output any correct shortest-path solution.

9.1 Store Location Helper Function [/6]

First, write a helper function to find the location of a single store in the city grid. You may assume there are no duplicate stores in the city.

sample solution: 9 line(s) of code

9.2 Providing Step-by-step Directions [/15]

Now, using the helper function you defined in the previous part, implement the `give_directions` function. Make sure your code does simple error checking and outputs a helpful error message to `std::cerr` if either the starting point or ending point does not exist in the city.

sample solution: 23 line(s) of code

9.3 Order Notation [/2]

If there are n blocks in the city downtown, what is the order notation of your solution?

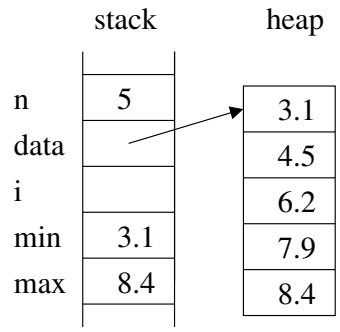
10 Minimum and Maximum Absolute Value [/20]

Finish the program below to read in a positive integer `n` followed by n floating point numbers from the keyboard (`std::cin`). The code prints out the absolute value of each of the input numbers as well as the minimum and maximum absolute values. If the user types:

```
5    3.1  -4.5  6.2  7.9  -8.4
```

The program should produce the memory diagram on the right, and output this to the console (`std::cout`):

```
absolute values: 3.1 4.5 6.2 7.9 8.4
min: 3.1
max: 8.4
```



The main function is responsible for input and output. A helper function will edit and process the data. *Note:* Make sure the finished program does not have any memory leaks.

```
int main() {
```

sample solution: 9 line(s) of code

```
   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;
```

sample solution: 1 line(s) of code

```
   return 0;
}
```

Now implement the helper function `find_min_and_max`:

sample solution: 10 line(s) of code

11 Olympic Medal Statistics [/34]

In this problem you will implement a simple class named `OlympicTeam` to keep track of the athletes on one country's team and the gold, silver, and bronze medals won by that team. Here's an example usage of the class. We create an `OlympicsTeam` object and enter data about the athletes and the medals as they are won.

```
OlympicsTeam US_Winter_2014;

US_Winter_2014.addAthlete("Sage_Kotsenburg");
US_Winter_2014.addAthlete("Hannah_Kearney");
US_Winter_2014.addAthlete("Gracie_Gold");
US_Winter_2014.addAthlete("Lindsey_Van");
US_Winter_2014.addAthlete("Shani_Davis");

US_Winter_2014.addMedal("Sage_Kotsenburg","gold");
US_Winter_2014.addMedal("Hannah_Kearney","bronze");
```

The `OlympicsTeam` object should do simple error checking on the input, making sure we do not add the same athlete multiple times, or try to count medals won by athletes not on this team, or try to add medals that are a color other than "`gold`", "`silver`" or "`bronze`". Each of the cases below results in a descriptive error message printed to `std::cerr`, with the data in the `OlympicsTeam` object unchanged.

```
US_Winter_2014.addAthlete("Gracie_Gold");
US_Winter_2014.addMedal("Michael_Phelps","gold");
US_Winter_2014.addMedal("Lindsey_Van","pewter");
```

We can output information stored in the `OlympicsTeam` object:

```
std::cout << "The US Winter Olympics team has " << US_Winter_2014.numAthletes()
          << " athletes and has won on average " << US_Winter_2014.averageMedalsPerAthlete()
          << " medals per athlete." << std::endl;

if (US_Winter_2014.hasWonGoldMedal("Sage_Kotsenburg")) {
  std::cout << "Sage_Kotsenburg has won a gold medal." << std::endl;
} else {
  std::cout << "Sage_Kotsenburg has not (yet) won a gold medal." << std::endl;
}
if (US_Winter_2014.hasWonGoldMedal("Lindsey_Van")) {
  std::cout << "Lindsey_Van has won a gold medal." << std::endl;
} else {
  std::cout << "Lindsey_Van has not (yet) won a gold medal." << std::endl;
}
```

The code above results in this output to the screen (`std::cout` and `std::cerr`):

```
ERROR: cannot add duplicate athlete 'Gracie_Gold'
ERROR: athlete 'Michael_Phelps' is not a member of this team
ERROR: unknown medal color 'pewter'
The US Winter Olympics team has 5 athletes and has won on average 0.4 medals per athlete.
Sage_Kotsenburg has won a gold medal.
Lindsey_Van has not (yet) won a gold medal.
```

11.1 `OlympicTeam` Class Declaration [/14]

Using the sample code on the previous page as your guide, write the class declaration for the `OlympicTeam` object. That is, write the *header file* (`olympicteam.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`. Don't include any of the member function implementations in this file: save the implementation of all functions for the next part.

sample solution: 13 line(s) of code

11.2 `OlympicTeam` Class Implementation [/20]

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

sample solution: 40 line(s) of code