Brandon Swanson
August 8th 2014
CS 165

Week 7 Assignment Report

EXERCISE COMPONENTS AND MATERIALS

Understanding

This week there was more on classes with constructors and operator overloads. An introduction to the standard template library with vectors. And some exercise components focused on exception handling. This week as well there was a subtle focus on algorithms in our exercise components. We were asked to solve a few problems that were a simple operation but with possible pitfalls and edge/corner cases. These problems like the nSuitors problems can be boiled down to a very well defined set of expected input and output, and tested stringently. This kind of activity will become invaluable when we develop complex systems like "The Game of Life" being able to test the algorithms that back the larger interactions in isolation and be assured of their veracity before using them to drive larger systems will help keep debugging efforts properly focused.

I am overjoyed that we are finally beginning to move into the standard template library. There are many great applications for simple types and arrays, but containers really begin to shine when we develop classes of our own. Particularly after manipulating these template data types like list and vector it starts become apparent how a more nebulous set of data, as in a collection of classes, could be similarly contained and operated upon through a defined interface. For instance a store inventory class could be developed that has as its core a vector or list of Items (like the ones developed for consignment) and then similar to the way that .pushBack() .front() .size() etc abstract our interaction with the data of a vector, methods could be defined for interaction with these Item classes.

Operator overload is a very exciting area of class development, It can allow for syntax sugar on our own objects. On an object like the consignment store item the << operator can be overloaded to display more than just one of its fields and to intelligibly build an output string from its parameters. For more complicated objects it becomes more nebulous how they would be added,subtracted or importantly compared for equality with ==. but operator overload can allow these operations to be performed on our objects in an encapsulated and information hiding way. In Java every object inherits the .toString() method but unless overridden will usually just output a memory address (or some other oblique information). I was admonished to always override .toString(), particularly to facilitate logging. It would seem a good maxim to always override << when creating objects in c++.

I will here describe each Exercise Component as each one this week is so dense and attempt to touch on Understanding/Design/Test/Reflection on each of them.

CALL DEPTH

In this exercise I ended up implementing two methods, because after the first I was surprised by the outcome. I used a recursive function with an int as the parameter. The user defined input number determined the end of the recursive calls and the tracked call depth was thrown to be caught by an exception handle catch statement that accepts an int. In the first method the main function catches the exception, this worked well because I needed to enclose the first call in a try block in case the call depth was 0. But what I noticed was that the trace statements I was using for testing that were after the recursive call (i.e. should have been executed when the stack was being popped back to the top level) were not being reached. I understand now that this is the nature of the catch statements; they surrender control of the program to the nearest frame with a catch statement. To contrast this I implemented a callDiverEXP function that caught its own exception, in this the call stack continued un-aborted after exception.

EXCEPTION

For this exercise I chose to modify the function so that it throws an exception of type

ItemNotFound (an empty class declared in this program) and the main program would attempt (try) to retrieve the product ID but if the try statement resulted in a thrown exception of type ItemNotFound the catch statement would output an error message.

This seemed an appropriate strategy, as opposed to having the function itself catch the exception. After discussing with others on the forums this seems like the biggest advantage of exception statements the ability to delegate the response to an edge case (ie exceptional circumstance) to be handled by some other agent in the program. If the function has a way of responding then it could well do so with If statements. Additionally this provides flexibility on how to handle this situation to the user of an object. One circumstance where the function is used might find an exception unrecoverable and need to terminate the program, but other cases might find it to be trivial. If the getProductID function decided that upon not finding an item it needed to terminate the program this would not be a very portable function, but by declaring that it throws and exception of type ItemNotFound (which would be even better if it was a static member of the class that throws this exception) it allows different implementations to handle this exception differently.

I was excitedly able to automate the testing of this little program, I used a rubric of test inputs that can be seen inputTest.txt that could be fed into the function and then output the result to another log file (see log.txt). This was achieved with the following shell command

```
./bin 2>log.txt 0<inputTest.txt
```

My first version of this test did not include the numbers specifying that stream 2 (stderr,cerr) should be redirected, but to make the output file easier to read I split this programs output into two streams so that the resulting log was only the function output, and not the drivers input prompt.

I had been reading about automated testing for a few weeks after some subtle hints in the assignments, I explored using command line arguments, as well as directly manipulating the source of the input stream within c++ using functions like stream.rdbuf(). But I have been pleased with the outcome of using Linux redirection and piping. While it is not a system portable solution it is able to supply input from a file and log the output without making modifications to the program which could introduce bugs upon insertion or upon removal when testing is complete.

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Item not found exception	Array samples shown, Size of arrays, Target product: “apple”	Prompt for input exit() if input is “EXIT”	“apple was not found”	“apple was not found”
Item Found	Array samples shown, Size of arrays, Target product: “computer”	Prompt for input exit() if input is “EXIT”	“Product ID for computer is:(x)”	“Product ID for computer is:4”

InputTest.txt	Log.txt

`apple`
`computer`
`mouse`
`computer`
`3432q1  lkjad`
`33.2`
`camera`
`Camera`
`caMeRa`
`EXIT`
	apple was not found
	Product ID for computer is:4
	Product ID for mouse is:8
	Product ID for computer is:4
	3432q1 lkjad was not found
	33.2 was not found
	Product ID for camera is:13
	Camera was not found
	caMeRa was not found

CONSIGNMENT

The hardest part for this one seemed to be how to handle the case in which certain items are not initialized. Many people on piazza had different strategies for this. I chose to have boolean values that track whether or not these values have been initialized and if not then they cannot be accessed. These Items can be set after construction and if they are set they can be accessed by passing in a variable by reference, these getters return false if the value is not available and doesn't modify the value passed in.

This is a bit of an unorthodox version of a getter function. But It is lightweight and works well for access statements latter on, it can be written as : if(item.hasAge(age)) cout << age;

This is a tactic I have seen in libraries I have used that perform operations on data like points (x,y) and the point is passed in by reference to a bool function that returns whether or not the specific operation was successful (because it might not always be possible). I suppose that this kind of design is very similar to exception handling, but I think that this is not a situation were a strong case can be made for using exceptions and as we have been told they are computationally expensive.

I made a small little store inventory interface where instances of the store items are created and added to a vector (of item pointers) and then can be displayed or Sold(). The sold function demonstrates one of the more interesting features of a mutator function, many of the functions we have written so far are simply SetX(x){ myX = x;} but setters can be used in a more abstract way where an action performed on them modifies various private variables in a way unknown to the outside. This sale function switches the isSold bool and records the price passed in.

The display Item function I built that outputs the fields of an item making use of their public methods would be a better fit as a member function, either as a .ToString() method or as an << overload. But I wanted to make sure to test the functioning of all of the getter methods.

NSUITORS

This was particularly fun to design. I went through a few different versions. My first attempt was running into seg faults and after tracing with my output function and GDB I discovered it was the case in which the vectors size was smaller than 3, this problem could only occur once but in another version of this problem where the princess could count using a different number it could happen multiple times. I resolved this with using mod (always handy for these scenarios). I collaborated with Lisa Percival on testing each others code and after looking at hers I suspected I was doing extraneous arithmetic. Both version of my algorithm can be seen in my code with the first version enclosed in block comment.

Before switching algorithms from one that was working to one that I hoped would work but just satisfy me more I wanted a way to test that my new algorithm worked. So I re factored my code so that the counting algorithm was contained within a function that takes a number of suitors as a parameter and then return the final suitor number. By doing this I could log the output for a set of test cases (1-50) and compare them after changing the algorithm.

SEE: suitorsTestALGORITHM1.txt & suitorsTestALGORITHM2.txt

Algorithm 1	Algorithm 2
`1 suitors = 1 selected`	`1 suitors = 1 selected`
`2 suitors = 2 selected`	`2 suitors = 2 selected`
`3 suitors = 2 selected`	`3 suitors = 2 selected`
`4 suitors = 1 selected`	`4 suitors = 1 selected`
`5 suitors = 4 selected`	`5 suitors = 4 selected`
`6 suitors = 1 selected`	`6 suitors = 1 selected`
`7 suitors = 4 selected`	`7 suitors = 4 selected`
`8 suitors = 7 selected`	`8 suitors = 7 selected`
.	.
.	.
.	.
`45 suitors = 43 selected`	`45 suitors = 43 selected`
`46 suitors = 46 selected`	`46 suitors = 46 selected`
`47 suitors = 2 selected`	`47 suitors = 2 selected`
`48 suitors = 5 selected`	`48 suitors = 5 selected`
`49 suitors = 8 selected`	`49 suitors = 8 selected`
`50 suitors = 11 selected`	`50 suitors = 11 selected`

GAME OF LIFE DESIGN

This discussion of my design plans will be disjointed but I will try to go over everything that has gone into my design considerations so far. I have done a little bit of skeleton work to help test out some of these concepts, they can be seen in ToyLife.cpp. I will discuss later the abstract classes that are defined in that source file.

Infinite World

The true version of Game of Life is on an infinite plane, Our implementation is only required to be 80x22. A version of this size could easily be stored in a simple 2 dimensional array and operated upon in a "brute force" iterative manner. While our implementation is finite I have sought to define and algorithm and data structure that would work in an infinite plane. I always try to keep my objects as un-scoped as possible and let my implementation define the boundaries. This seems like good design practice and seems to lead to more portability. And for the sake of this course work I'm hoping to learn as much as I can from the exercise, So I will seek to refrain from iterating over the extent of the given universe for the game.

Many times the initial design planning process for me resembles a trip down a binary tree, continually trying to boil down the next step into A/B decisions and weighing the two options. Following from there what new considerations must be made. The shortcoming though of presenting a final Design-Doc is that it fails to represent the full scope of considerations so this time I attempted to catalog some such considerations along the way.

- store generations terminal output as char array, string or just output to stream without storing.
- recreate set of live cells from scratch, or duplicate and then death/birth, Should the world of living cells be built up from the rules of life, or should I duplicate the board from this generation and then process deaths and new lives
- have each living cell be an instance of an object that has a .generation() method called by the controller class, OR have the controller class maintain a container of simple types and perform operations itself.

- How to avoid data race situations, what order to call different agents (since they are supposedly simultaneous) and if cells are automatons how do they coordinate their actions
- Is class wrappers for data set worth any efficiency/memory costs associated
- Who owns the World Object? If I want polymorphism than some controller must pass in to God the pointer to a world interface object, but then another class has access

One of the first big design considerations was whether or not each living cell should be an instance of an object. The synopsis of the game of life describes it as a cellular automaton and in simulations like this it seems appropriate to create an object that can instance with a starting position and then "set free" on the world. I have made little toy classes like this, bouncing balls, or crawling bugs, ones that after creation have no reference to them, no external class operating upon them. Instead they operate on some cycle mechanism (a timer that calls a method) and then use a broadcast/listener pattern to announce their location to the view responsible for displaying them. While this kind of independent agent object model seems tempting in this case, I don't think it is the appropriate model; for two large reasons. Each cell is not a great example of an object, it contains only two pieces of information, Its own existence, or that it is alive, and the second, its location in the world. The fact that this cell is alive could be also ascertained by its membership in a set, reducing it to only a single piece of information, its location. The cells cannot move, and would need a pretty sophisticated system of messaging to discover the quantity of neighbors and thereby instigate their own death (deletion). More importantly who would be responsible out of these cells for the creation of new life.

For this reason I have settled on a plan that involves a controller class managing a data set that describes the world each generation. It will then perform operations (or trigger an agent to do so) on this world each generation and carry out the rules of life.

Two worlds, this gen and next gen

Early on in my design process one of the first difficult problems to untangle was the simultaneous execution of the rules. The rules for living (2 neighbors) dying (!2 neighbors) and birth

(empty cell 3 neighbors) have to all be executed at once, a born cell does not count as a neighbor until the next turn and a dying cell is still counted as a neighbor for this turn. When drawing out the rules on paper this important feature became hard to calculate on a single sheet, as I would erase a cell that was fated to die I would realize I still needed to count that cell for the sustaining or generation of life for the cells around it. This led to two important realizations

- I would need to have two copies of my data set
    1. the world as it is this generation and
    2. the world as it will be next generation
- I needed to track (have some reference to) not only the cells that are alive but all cells that are a part of the Moore neighborhood (8 immediate neighbors) of any of the living cells.

So one copy of the data that express the world will be used to create the life,death conditions in the alternate one, and these two data sets will rotate places each generation. I think that the most elegant way to achieve such binary orbiting operations will be using 2 pointers pointing to two versions of the data, and swap these two pointers each generation.

Map<<x,y>,int> of neighbors and Set<<x,y>> of living

After a lot of consideration and a little bit of testing their functionality I decided that each generation should be stored as a map that has location as the key and number of neighbors as its value. And a set that expresses all the living cells, the set would be a set of locations and membership in this set implies life.

This seemed the best way to efficiently store and access the two pieces of information needed about the world in order to execute the rules. In an object oriented framework the map containing number of neighbors will be filled by each cell incrementing the 8 locations around it by one. In my specific implementation it will be done by the controller iterating once over the set of living cells and performing the incrementing.

The beauty of using these Map and Set combinations is that they are only as large as they need to be each generation. If a 2d array was used in an imaginary implementation (also considering a universe without number overflow) then as a glider moves across screen the array would need to grow, but this set and map can hold cells that are infinitely isolated from each other without growing beyond n (the number of cells) for the set of living and at max n8 for the map of all cells with living neighbors.

The particular ones I will be using will be ordered, which usually means that they will be searched with using a binary search. It is noted that these are slower than their unordered counterparts but they are advantageous in a (theoretically) large environment like this because they don't need as much memory (ie no hash table) as the un-ordered containers. As well the un-orderd containers are a c++ 11 feature, so another reason for using ordered in this implementation.

I would really like to use a struct that holds an x,y as the key for the map but this will require the writing of a compare (<) operation. I played with the map class a little bit and found that it was able to sort pairs with no extra function object parameter but passing around coordinates that are referenced as .first() and .second() seems very open to error. I will need to learn this coming week how to write the comparison functor but that is an implementation detail.

Class wrappers: GOD, WORLD, ANGELS

despite deciding to represent the data as collections of simple types I will still be using objects as data wrappers to abstract and obscure the interactions with the world and execution of rules. This way if I find myself to have been wrong about the choosing of my data structure I can change it while leaving the agents acting upon it untouched.

I usually am very formal in the naming of my object classes but as this game has the grandiose title “The Game of Life” I have decided to name my classes along a similar theme. I will have a God class controlling the interactions of the data set and its agents that perform the rules of life. The data set will be wrapped in a class called World that will allow for iteration over its data and for the execution of game rules, performing live(), die() or birth() on specific locations. And Finally an Angel of Life will be responsible for interacting with the world and killing or bringing to life cells.

In this way the algorithm, the data set, or their interactions can be modified in isolation. If the Angel was calling map operations directly or used a map<,> iterator, then changing the data structure of the World class would require rewriting the Angels operations.

See the abstract class for world developed in ToyLife.cpp that show the undefined methods for accessing and setting the life and death of its cells, also look at the implemented generation() methods for God and Angel. By writing this abstract World class I am able to define the algorithm for the Angel in very simple, readable terms. (please see flow diagram attached at end)

No Angel of Death

In the first consideration of the algorithm for the rules I imagined the necessity for an angle of death, or specifically the operation of performing the rule for a dead cell on the set of living cells. Upon further consideration it seemed extraneous, as the rules for life were going to require the checking of a larger number of cells, having a second agent iterate over a subset and mark for elimination seemed less efficient that having the other agent mark all that are fated for life.

Two factors were analyzed, isolated cells that have no other cells as their neighbor will not be in map of neighbor quantities, but these will already be fated for death, so if the absence of being marked for life signified death, then only iterating over one of the worlds containers would be necessary.

After analyzing the complexity and number of operations between the two possible strategies (angel of life and death, or only angel of life) I decided it would be better to refresh the set of living every time (start with an empty set) and have the agent that process the rules for sustained or new life (the AngelofLife class in my case) implicitly executing the rules for death.

Attached is a sketch to the two algorithms, from analysis the sole angel of life would iterate once over the map and search the set once each time {max n*8*(accesses + searches)}. Whereas if the set was duplicated and then cells were removed, sustained, or generated it would require the previously described iteration+search as well as an iteration over the living set with a search for the number of neighbors at each evaluated cell

{max n*8*(accesses + searches[n]) + n*(access + searches[max n*8])}

While this is not a strict big O analysis of complexity it is quite apparent which is more efficient both because adding the death agent involves a second iteration over a set but because the set of all cells in the Moore Neighborhood of all all living cells is always larger than the set of all cells; it seems better to be searching across the smaller set.

USER INTERACTION

I think that for starting positions I will allow the user to select from a few pre-designed patterns that include stable designs or from randomly generated patterns. I think it would enjoyable to try to make a randomizing algorithm that is not evenly weighted, because evenly dispersed cells will probably just lead to a quickly empty board, but if they could be clustered it should make an interesting pattern. Of course this would have to be implemented after all else was function, it is a “cherry on top” spec, and has a very low priority.

The second part of user interaction will be advancing through the generations. I would like to have a version where the game plays out and the user can pause it at will, but without a keyboard buffer through a system specific header like connio.h or terminos.h this wont be possible. So I will either have the user enter how many generations they wish to see (with a user set delay between) or use getchar() to advance through them by only pushing return.

TESTING AND IMPLEMENTING PLAN

STUBBING AND MODEL CLASSES

I will start with making a very small scale version of the World class with a simple data set, a small 2d array maybe, and simple output method. I can test a few specific test cases and (literal) edge and corner cases. It is mentioned that we have not covered file operations and wont be allowed to use files for input, But I will still want to use file output as a testing technique.

As I mentioned the abstract class defined for World will allow me to supply different implementations to the God controller. I will make one that is a small universe and outputs to file stream allowing me to check that the angels interactions are happening appropriately. I can use a more extended language than the binary cell no cell output of '*' or ' ' to help demonstrate operations being preformed. I could output the number of neighbors in each cell, and then output sustained life with a different symbol than generated life. Then multiple generations from multiple starting positions can be output to a file and compared to my expected outcome test cases.

See below my attached test patterns.

Images appendix

1. game of life first object and messaging design
2. analysis of Angel of Life building new world from empty set vs Angel of live executing Birth if 3 neighbors rule and Angel of Death executing Death if !2 neighbors rule
3. Test world sketch to elucidate operations of agents and a rough sketch of world class interface.

Game Off Life D013< L3

8 neighbors

< Interface >

Generation()

Live()
Die()
?

World

Map < Cord, int Neighbors >
Neigbors

Cord <x,y>
⇕
Pair <int, int>

+

Set <cords>
alive

Alive
ThisGen* ⇄ Next gen*

GOD

Neighbors.Clear()

∀ alive .+ Neigbors

~~[illegible] Swap()~~

LiveCell

Cell
+ Alive
+ Cordsnates

Angel Of Life

∀ Neibors ≠ ! Alive ∧ 3

Birth → Next Gen

Angel Of Death

∀ this gen Alive ∧ Neibors ! 2

DIE → Next Gen

↓

Swap Gens()

neighbors < Live

$\boxed{\text{Always}}$

Life $\boxed{\text{new Set ADLife only}}$

```
∀ neighbors
    ~~if 3~~ [illegible] alive() ∧ living()
    if [alive bool]  neighbor [int]
    if (alive)
        if < 2 die
        else live
    if !(alive)
        if 3 alive
```

4

iter neighbors.size
search alive ∧ neighbors.size()

$\boxed{\text{Dupeset}}$

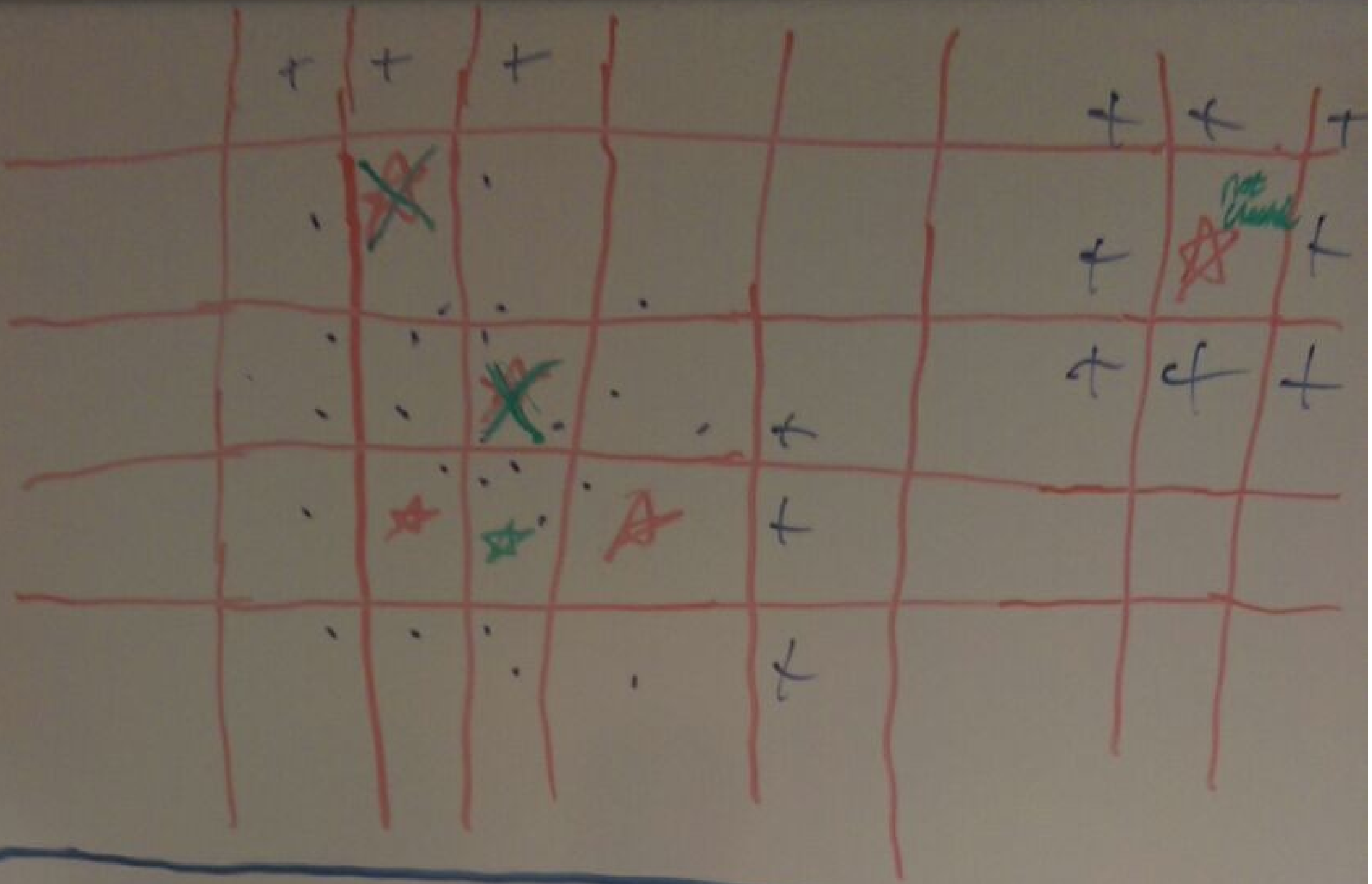
$\boxed{\text{Life}}$

```
∀ neighbors
if 3 ∧ !Live
    Born()
```

$\boxed{\text{Death}}$

```
∀ Live  if neighbor ! 2
    Die()
```

access all neighbors all alive
Search alive ∧ neighbors
Search ~~∧~~ neighbors ∧ alive



World   Set + Map wrapper

```
struct Coord {
  Coord x
  int y
}
struct Cell {
  Coord
  int Neighbors
  bool Alive
}

int CellsWithNeighbors() → sizeof() Neighbors
Cell NextCell() → it++

bool is alive (Coord) → alive.count()

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

int num Living() → sizeof() Living
Coord NextLiving() → it++
```

Test Patterns on a 8x8 grid

Glider 1)

						#	
				#		#	
					#	#	

2)

					#		
						#	#
					#	#	

3)

						#	
							#
					#	#	#

4)

					#		#
						#	#
						#	

5)

							#
					#		#
						#	#

6)

							#
#	#						
#							#

7)

#							
	#						
#	#						#

8)

	#						#
#	#						
#							

9)

	#						
	#						#
#	#						

Stable 1)

#	#						
#	#			#	#	#	
			#	#			
		#			#		
			#	#			

2)

#	#				#		
#	#				#		
					#		
			#	#			
		#			#		
			#	#			

3)

#	#						
#	#			#	#	#	
			#	#			
		#			#		
			#	#			

Scatter 1)

		#	#				
	#						
		#					
		#	#				
			#				

2)

		#					
	#	#					
			#				

3)

	#						
		#					

Template for observing output.

1)	**2)**	**3)**
4)	**5)**	**6)**
7)	**8)**	**9)**