Winter 2022 Study Guide: ECS 36B with Gygi

1. All the ways that a parameter can be passed to a function in C++
    a. Pointer | Int a = 7; funct(&a); —> void funct(int* p) —> *p = 5;
    b. Reference | int a = 7; funct(a) → void funct(int& r) → r = 6;
    c. Pass by Value | int a = 7; a = funct(a) → int funct(int b) → b = 6;
    d. Const keyword
        i. Int getMonth() const; | function is "read-only" function, doesn't modify object / parameter for which is it called
        ii. Physical const-ness: all data (bits in memory) representing the object is unchanged
        Logical const-ness: bits in memory (data) may change, but no change is visible
        Use "mutable" to promise logical const-ness while breaking physical
2. The ways in which const variables and references can be initialized in a class ctor
    a. Initialization lists can be implemented within ctors to initialize const data members and references, which are done so before the body of the ctor. Variables are initialized in the order of the var declarations within the class, not the order in which they are listed within the initializer list.
    Syntax: Base (int i) : val(i) {} and/or Base (int& a) : val(a) {}
3. Base & Derived Class Data Access based on public/protected/private data members
    a. Public Inheritance (class Derived : public Base)
        i. All public inherited members stay public, protected members inherited stay protected, and the private members are inaccessible by derived class.
    b. Protected Inheritance (class Derived : protected Base)
        i. All public and protected inherited members from the base class become protected in the derived class, which priv members stay inaccessible
    c. Private Inheritance (class Derived : private Base)
        i. All public and protected members from the base class become private in the derived class.
    d. Outside Access (ex. main): can only access public members, not prot or priv
4. Definitions of Diff Constructors

a. Default Ctor - ctor that takes no arguments. If no ctor is defined by the user, the compiler will auto generate a default ctor & copy ctor?.
b. Ctor with Parameter - pass values as args to ctor. Ex: Value v(5); -> Value(int v). If ctor w/ parameter is defined by the user, then u need to declare default ctor?
c. Copy Ctor - Ctor that takes an object as an arg, and is used to copy values of the data members of one object into the other object. Ex: Value w(v); → Value(const Value& v);

5. Destructor and Virtual Destructor
a. Member function called when an instance of a class (mouthful for object) goes out of scope [past '}']. Used to clean up any memory space or other resources allocated during the object's lifetime.

~ClassName (void) {}

b. Virtual Destructor: Necessary to put virtual destructors in the base class, so that you can correctly delete a derived object using a pointer of base type. Trying to do so without a virtual destructor results in undefined behavior.

virtual ~ClassName(void) {}

6. Default Assignment Operator
a. Function of '=' predefined by the compiler.

7. This pointer
a. Pointer with address of the current instance of a class data member (not function parameter)

this->varName;

8. What is Const member function, how is it declared
a. Refer to 1d ?

9. Allocate array of doubles of size 10 using new, and initializes a pointer. Then free the memory allocated above.

```
double* p;
p = new double[10];
delete [] p;
```

10. Try/catch blocks logic & syntax
a. Logic kinda common sense ig.

```
#include <stdexcept>
   // Within set(x)
if( x < 5) { throw_argument ("value must be greater than 5"); }

   // Within main
try {set(-4); }
catch (invalid_argument &e) {cerr << "Exception: " << e.what() <<
endl;
```

11. Redefining/Overloading vs. Overriding a Member function

a. **Overriding** a Member Function: Choice of which function to call is achieved at **runtime**. A function in the base class gets redefined within the derived class. Usage of virtual functions required. Function maintains same signature (return type, parameters, etc)

b. **Overloading** a Member function: Choice of which function to call is achieved at **compile** time. Basically you provide multiple definitions of the same function (# of parameters. Type of parameters, etc)
EX: void area(int i); void area (double b);

12. Role/Syntax of functions declared as **friend** of a class

a. Friend functions can access private and protected members of that class. Friendship is not symmetric nor transitive

13. Overloaded operator+ for class with 2d points

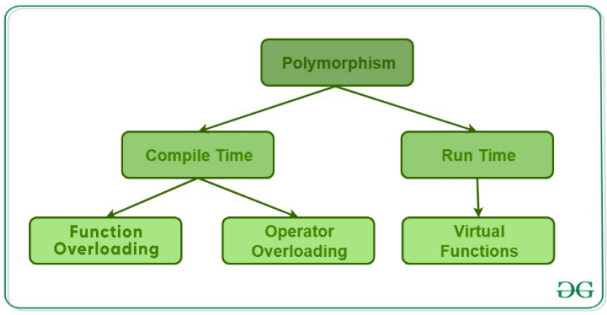
```
Class Point {
   Public:
      friend Point operator+(const Point& first, const Point& second)
        { Point sum;
          Sum.x = first.x + second.x;
          Sum.y = first.y + Second.y;
          return sum; }
   Private:
      Int x, y;
};
```

14. Why the class Penguin shouldn't be derived from the Bird class.

a. Penguins can't fly lmfao. Base classes should have properties that are universal to all of its derived classes.

15. Polymorphism

a. .



b.

16. Run-Time Type Identification (RTTI). Give an example using the dynamic_cast operator

a. RTTI is basically just converting an object's type to another type during runtime. If the case is unsuccessful (Dynamic_cast), then nullptr returned

- `static_cast<T>()`
  - to undo an implicit conversion (ok) e.g. int to float
- `dynamic_cast<T>()`
  - to identify a polymorphic object (ok)
- `const_cast<T>()`
  - to write on something declared `const` (dubious)
- `reinterpret_cast<T>()`
  - to change the meaning of a bit pattern (nastiest)

17. Factory member functions, and why they should be static functions

a. Factory member functions should be static because you want to be able to create derived classes, even if there isn't an object of the base class. Additionally, it's important to be able to call the function without an object of the base class if the base class is an abstract class (can't initialize instances of abstract classes).

18. Generic Programming

a. template <typename T>

19. STL Container Definitions

- **Sequential Containers**
  - **Vector**: resize dynamically at runtime, elements added to the end efficiently, provides random access to elements [O(1)]

    - **Array**: Vector but of fixed type
    - **Deque** (double ended queue): elements can be efficiently added or removed at both ends, provides random access to elements
    - **Lists** (doubly linked): Elements can be efficiently added or removed anywhere, but access to elements is sequential
- **Associative Containers** (refer to element by giving value)
    - **Set**: collection of objects. No duplicates allowed. Efficient lookup of elements
    - **Multiset**: set, but duplicates allowed. Efficient lookup of elements
    - **Map**: set of key-value pairs. No duplicates allowed. Access elements using key
    - **Multimap**: map but duplicated allowed.
- **Container Adapters** (a way of using containers)
    - **Stack**: LIFO sequence of objects
    - **Queue**: FIFO sequence of objects
    - **Priority Queue**: sequence of objects ordered by priority. Order preserved as elements are added/deleted

20. .

a.
```
vector<int> v {1,2,3,4,5};

// v.insert(position, val);
v.insert(v.begin() + 3, 6);
// ---> (1,2,3,6,4,5)
```

21. Function Object

a. type that implements operator() uses function objects primarily as sorting criteria for containers and in algorithms.
b. Predicate: function that returns true or false. It's a special kind of function object

22. Write a function object that can be used as predicate with partition algorithm

a.

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

bool is_even(int i) { return i % 2 == 0; }

int main() {
   vector<int> v;
   while (cin) {
       int a;
       cin >> a;
       v.push_back(a);
   }

   vector<int>::iterator iter1 = v.begin();
   vector<int>::iterator iter2 = v.end();
   vector<int>::iterator last_val;
   last_val = partition(iter1, iter2, is_even );

   for (int i = 0; i < v.size(); i++) {
       cout << v[i] << endl;
   }
```

23. Write a while loop that reads integers from standard input until the end of file is reached and prints them one per line on standard output.

a.

```cpp
int a;
vector<int> v;
while (cin >> a) {
    v.push_back(a);
}
vector<int>::iterator iter = v.begin();
while (iter != v.end() ) {
    cout << *iter << endl;
    ++iter;
}
```

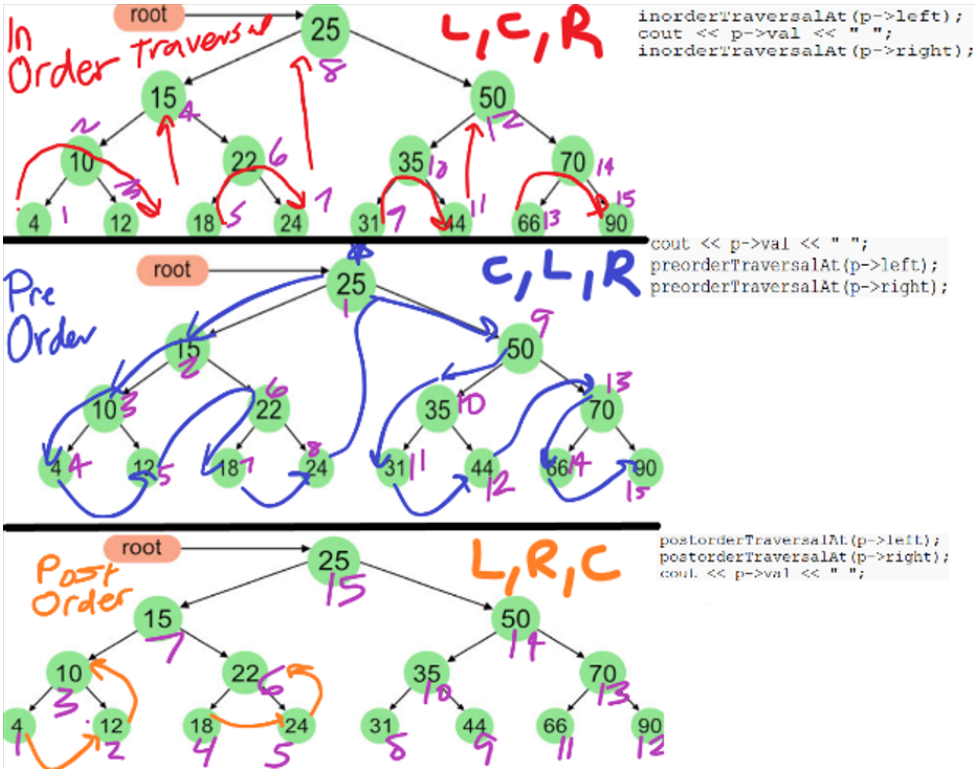
24. Write a program that reads strings from stdin, inserts them in a set, then prints them sorted on stdout

a.

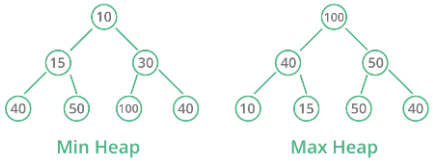
```
string str;
set<string> strSet;
while (cin >> str) {
    strSet.insert(str);
}

// option 1
set<string>::iterator iter = strSet.begin();
while (iter != strSet.end() ) {
    cout << *iter << endl;
    ++iter;
}

// option 2
// for (auto const& dummyVar : strSet) {
//     cout << dummyVar << endl;
// }
```

(remember: sets get auto sorted)

25. **Binary Tree:** A data structure where each node is connected to at most 2 nodes.
26. **Binary Search Tree:** A type of binary tree where the node values can be ordered. Values in the left subtree are smaller than the parent node. Implemented using the set container
    a. In-order Traversal : left, center, right
    b. Pre Order Traversal : center, left, right
    c. Post Order Traversal : left, right, center



27. Cost of finding an element in a balanced binary search tree: O(log N)
    Cost of insertion: O(log N)
    Cost of memory: O(N) | sizeof(node) = sizeof(type) + 2 * sizeof(Node*)
28. Binary Heap: type of binary tree that is fully filled except for potentially the last row. A binary heap can either be:
    a. max heap: every element is larger than its two children, with the root having the largest value
    b. min heap (every element is smaller than its two children, root = smallest value



29. Where to add an element in a given binary search tree
    a. Start from the root node, then compare the value with the current node's value. If the value < current node, then go to the left child. If the value > current node, then go to the right child. Repeat the process until you find a null ptr, which is where you can insert elements. If you find that the value and the current node's value is the same, then there's no need to insert (u found a duplicate).
30.

```
list<string> a {"abcd", "efhg", "poiut"};

for (auto const& dummyVar : a) {
    cout << dummyVar << endl;
}
```

Miscellaneous Notes:

```
 - used to manipulate containers: (find, search, copy, sort, remove)
 - algo's defined separately from containers
   - interacted with container only through iter
   - algo can copy elements, but not directly add/rm/resize
*/
#include <algorithm>

1. Find (element)
```

```
    find (first, last, value)
    returns iterator pointing to first instance
    //   if ( (iter = find(l.begin(),l.end(),14)) != l.end() )
    // cout << 14 << " was found" << endl;

2. for_each
    calls function for each element in a range
    for_each(l.begin(),l.end(),funct);  // funct is function pointer (store
address of funct)
    void funct (int i) { .. }

predicates : T/F statement abt elemment properties
    used by algorithms to select elements from container
    define predicate -> define bool f

bool gt_10 (int i) {return i>10;}
  // find first element greater than 10
  iter = find_if(l.begin(), l.end(), gt_10);
  cout << " the first element larger than 10 is " << *iter << endl;

// function object - clas/struct containing function. Passed as parameter
    // to algo's

// function object for predicate "greater than 10"
struct Gt_10
{ //   i think dis overloading function call
  bool operator() (int i) { return i > 10; }
};
  // find first element greater than 10
  it = find_if(l.begin(), l.end(), Gt_10() );
  cout << " the first element larger than 10 is " << *it << endl;

3. Sort Algo
    sort(first,last)
    sort(first,last,binary_predicate) -> predef or user-defined
        bool function | true if arg1 < arg2
    predef binary predicate ex: less<int> , greater<double>
          sort(v.begin(), v.end(), less<int>());
          sort(v.begin(), v.end(), greater<int>());
```

```
    predicate must define strict weak ordering
        basically less than rules. a < b and b < c, a < c
        '<' defines S.W.O if has following properties:
            1. Antisymmetry: if x<y true then y<x is false
            2. Transitivity: if x<y and y<z then x<z
            3. Irreflexive: x<x is false
            4. Transitivity of Equivalence: x & y == if x<y and y<x both F
            ("<=" doesnt define a S.W.O)

    if user defined binary predicate is j bool function:
        // smaller compares absolute values only
        bool smaller(int i, int j) {
        return i*i < j*j; }
        --->   sort(v.begin(), v.end(), smaller);
    if user def binary predicate is class:
        // smaller compares absolute values
        class smaller {
        public:
        bool operator()(int i, int j) {
            return i*i < j*j;
        } };
        ---> sort(v.begin(), v.end(), smaller());

3a. Stable-Sort Algo
    default sort by norm: (random order if 2 are equiv)
    // sort(pts.begin(), pts.end());
        (0,1)
        (1,0)
        (-1,0)
        (0,-1)
    pt a, sort by xless, sort(pts.begin(), pts.end(), xless);
    then:
    pt b, stable_sort by norm (in each group with same norm,
        x-values ordered in increasing order)
            //     stable_sort(pts.begin(), pts.end());
            //     where:
            //     int norm2(void) const { return x*x + y*y; }
            // bool operator<(const Point& p) const
            // { return norm2() < p.norm2(); }
```

```
        (-1,0)
        (0,1)
        (0,-1)
        (1,0)

4. Search Algorithm: find sequence in a container
    search(first1, last1, first2, last2)
        find 1st occur of [first2,last2) in [first1, last1)
        f1, l1 is like list1 (longer), f2 l2 like list2 (subseq)
            returns last1 if no occur found (rem: past l1)
            returns first2 is found occur
     list<int>::iterator first2 = l2.begin();
    list<int>::iterator last2 = l2.end();

    it = search(l1.begin(),l1.end(),first2,last2);
    if ( it != l1.end() ) {
        cout << "sequence l2 found" << endl;}

5. Random Stuff
    max(val1,val2) -> returns largest of 2
    min(val1,val2)
    max_element(first,last) -> returns iter pointing to 1st instance of largest
elment
        list<int>::iterator imax;
        imax = max_elements(l.begin(),l.end());
            *imax -> is largest value
    min_element(first,last)

    fill(first,last,value); -> fill a container with same value
    fill (myvector.begin(),myvector.begin()+4,5);
        // myvector: 5 5 5 5 0 0 0 0
        // remember: [first, last)

    partition algorithm: reorders elements to beg of container if predicate
satisfied
        1 2 3 5 7 11 13 17
        partition(l.begin(), l.end(), gt_10);
            -> 17 13 11 5 7 3 2 1
    stable partition: preserves original relative order in l when reordering
```

```
        stable_partition(l.begin(),l.end(),gt_10);
            -> 11 13 17 1 2 3 5 7

5. Copy algorithm
    copy(first,last,dest) -> returns iter pointing one past end of dest range
        ranges specified by iters
        copy to diff container: copy(a_first,a_last,b_dest)
        or same container: copy(a_first,a_last,a_dest)

    use copy_backwards is i/o ranges overlap in same container
      ex scenario:
        ----------------------------
            ^       ^       ^
            |       |       |
        a_first   a_dest   a_last
            ------->  -------->

6. Remove Algorithm
    remove (first, last, value);
    reorders elements in container. Stuff to keep is at beg
    does NOT resize container.
    Use only for sequential containers, not associative ones
    TIP: follow up remove with erase()
    ex:
        v: 1 2 3 4 1 2 3 4 1 2 3 4
    iter = remove (v.begin(), v.end(), 3);
        v: 1 2 4 1 2 4 1 2 4 2 3 4  // note: 3 rm'd, last 3 elements unchanged
from orig v
        iter points to first irrelevant element
    for ( vector<int>::iterator i = v.begin(); i != iter; i++ ) cout << *i;
        -> 1 2 4 1 2 4 1 2 4
    v.erase(iter,v.end());  --> to erase 2,3,4 (index 9-11)

    Set Operations;
    set_union : return sorted range formed by elements present in either one of
                sets or present in both ranges (remember set - no duplicates)
        set_union(first1,last1,first2,last2,result);

        ar1: 5, 10, 15, 20, 25
```

```
        ar2: 50, 20, 30, 40, 10   & vector v of size 10
        iter = set_union(ar1,ar1+5, ar2, ar2+5, v.begin());
            --> 5 10 15 20 25 30 40 50 0 0
                (union has 8 elements, v of size 10)
            ---< iter points to end of constructed range

    set_intersection : return sorted range with elements present in both sets
        set_intersection(first1,last1,first2,last2,result);
            iter = set_union(ar1,ar1+5, ar2, ar2+5, v.begin());
                --> 10 20 0 0 0 0 0 0 0 0
            v.resize(iter-v.begin());
                --> 10 20

    set_difference : return sorted range with unique elements only found in set1
        set_difference(first1,last1,first2,last2,result);
        iter = set_difference(a1,ar+5,ar2,ar2+5,v.begin());
            --> 5 15 25 0 0 0 0 0 0 0
            iter points to first 0.

    set_symmetric_difference : elements present in one set, but not other
    iter = set_symmetric_difference (first, first+5, second, second+5,
v.begin());
        --> 5 15 25 30 40 50 0 0 0 0
```

```
lambda expression: express predicate w/o defining additional bool f or funct
object
A lambda expression constructs a closure: an
unnamed function object capable of capturing
variables in scope

sort(v.begin(),v.end(),
      [](const Person& p1, const Person& p2)
      { return p1.first_name < p2.first_name; } );

sort(v.begin(), v.end(), [](int& a,int& b){return a*a < b*b;} );

[] part of the lambda expression
specifies how variables are captured
```

```
 capture specify how local vars accessible from within lambda

 []: no variables captured
•[=]: all local variables captured by value
•[&]: all local variables captured by reference
•[x]: local variable x captured by value
•[&x]: local variable x captured by reference

ex:
int threshold = 10;
  auto iter = stable_partition(v.begin(),v.end(),
              [threshold](int& i) { return i > threshold; } );

---
c++ streams are type-safe unlike printf
typesafe can be overridden

cin.good(): ok
cin.eof():  found end of file
cin.fail(): error has occurred in I/O
cin.bad():  unrecoverable error in I/O

// choose fixed point representation
  cout.setf(ios_base::fixed,ios_base::floatfield);
    --> fixed (decimal) or scientific notation
    --> floatfield set to fixed
// cout.precision() affects all following operations
  cout.precision(5);

  // cout.width() only affects the next operation. Must be repeated.
  cout.width(12);
  cout << a << endl;

  cout.width(12);
  cout << b << endl;
  cout.setf(ios_base::right, ios_base::adjustfield);  <-- default

#include <iomanip> //manipulators
 cout << setprecision(5) << setw(12) << fixed << a << endl;
```

```
persistent manipulators: setprecision, fixed, adjustfield
manipulators to be repeated: setw

use #include <cstdio> to use C-style I/O functions
    printf included in <iostream>

-------------------

#include<fstream>
ifstream myfile("not_there.conf");
  int i;
  myfile >> i;  // i = 0 if file not present
// or

ifstream myfile("not_there.conf");
  if (!myfile) {
    cout << "error opening file" << endl;
    return 1; }

//  create new file, write to it
 ofstream myfile("myofstream.txt");
  myfile << " hello " << endl;
  myfile.close();

------------------------------

Ostream iterators are output iterators that write
    sequentially to an output stream (such as cout)
    #include<iterator>

      for ( int i = 0; i < 5; i++ )
    v.push_back(i);

  ostream_iterator<int> iter(cout);
  copy(v.begin(), v.end(), iter); // this is printing the shit

  ostream_iterator<string> os_iter (cout, "\n");
  copy(s,s+6,os_iter)
```

```
  ---

    basic_ostream& put(char c)
    basic_ostream& write(const char *p, size_t n)
  char a[] = { 'a', 'b', 'c', 'd' };
  cout.put(a[0]);
  cout.put(a[1]);
  cout.put(a[2]);

  cout.write(a,4); (index: 0-3)

read(): read bytes, unformatted
istream& read(char *p, size_t n)

// write 4 unformatted integers
int buf[4];
char *p = reinterpret_cast<char*>(&buf[0]);
cin.read(p,4*sizeof(int));

--
 shape of binary tree depends on order of elements added

}
```

BINARY SEARCH TREE SYNTAX

```
class Node {
  public:
    Node(int i) : val(i), left(0), right(0) {}
    int val;
    Node *left, *right;
};

class Tree {
  public:
    Tree(void) : root(0) {}
    void insertNode(int i);
    void preorderTraversal(void) const;
    void inorderTraversal(void) const;
```

```
    void postorderTraversal(void) const;
  private:
    Node *root;
    void insertNodeAt(Node*& p, int i);
    void preorderTraversalAt(Node *p) const;
    void inorderTraversalAt(Node *p) const;
    void postorderTraversalAt(Node *p) const;
};

void Tree::insertNode(int i) {
  insertNodeAt(root,i); }

void Tree::insertNodeAt(Node*& p, int i) {
  if ( p == 0 )
    p = new Node(i);
  else {
    if ( i < p->val ) {
      // insert in the left subtree
      insertNodeAt(p->left, i);
    } else if ( i > p->val ) {
      // insert in the right subtree
      insertNodeAt(p->right, i);
    } else {
      // i == value of current node: i == p->val
      // do nothing: no duplicates
    } } }
void Tree::preorderTraversal(void) const {
  preorderTraversalAt(root); }

void Tree::preorderTraversalAt(Node *p) const {
  if ( p ) {
    cout << p->val << " ";
    preorderTraversalAt(p->left);
    preorderTraversalAt(p->right); } }
```

```
// Method 1:
int arr [] = {1,2,3,4,5};
vector<int> v(arr, arr+5);

// Method 2:
vector<int> v {1,2,3,4,5};
```

- When creating object of derived class. Base ctor called first, then derived ctor. For destructor, oppo sequence

- Bytes in memory represented either with:
  - Big Endian (MSB first - ex 2^{31}) -> IBM CPUs
  - Little Endian (LSB first - ex 2^{0}) -> intel x86 (most common)
- Pros/Cons of Lower Level Representation of Data
  - +: unformatted data is most compact
  - +: unformatted read/write has fastest I/O (binary)
  - -: may not be portable across platforms (data file written on x86 can't be read on ibm cpu, etc)
- UNIX File Descriptors: Connection made between I/O streams and OS system
  - 0 : stdin
  - 1 : stdout
    2 : stderr
  - 3 - 9 : other shtuff
- Unix misc
  - ./a.out > output.txt | ">" redirects output (not stderr) to file
  - ./a.out > output.txt 2> error.log | redirect output & stderr to 2 diff files
  - ./a.out &> output.txt | redirect both to same file
    - Can also do : ./a.out > output_and_error.txt 2>&1
    - 2>&1 → take file descr 2 and reroute everything into file descr 1
- Here Documents: include input in next command line
  - ./a.out << EOF | (EOF is dummy str) , basically type it to mark end of input
    - +: for writing shell scripts
- Here strings → include single line of input on command line
  - ./a.out <<< data1

31. Wed morning:

a. Look through past 2 mts

b. Look through shape progam

c. Print out shit

d. Attempt to leave conceptual Hash tables (or j take L)