Final Exam: Friday, March 15th

Name: ISHAN MUKHERJEE **NetID:** sdd7853

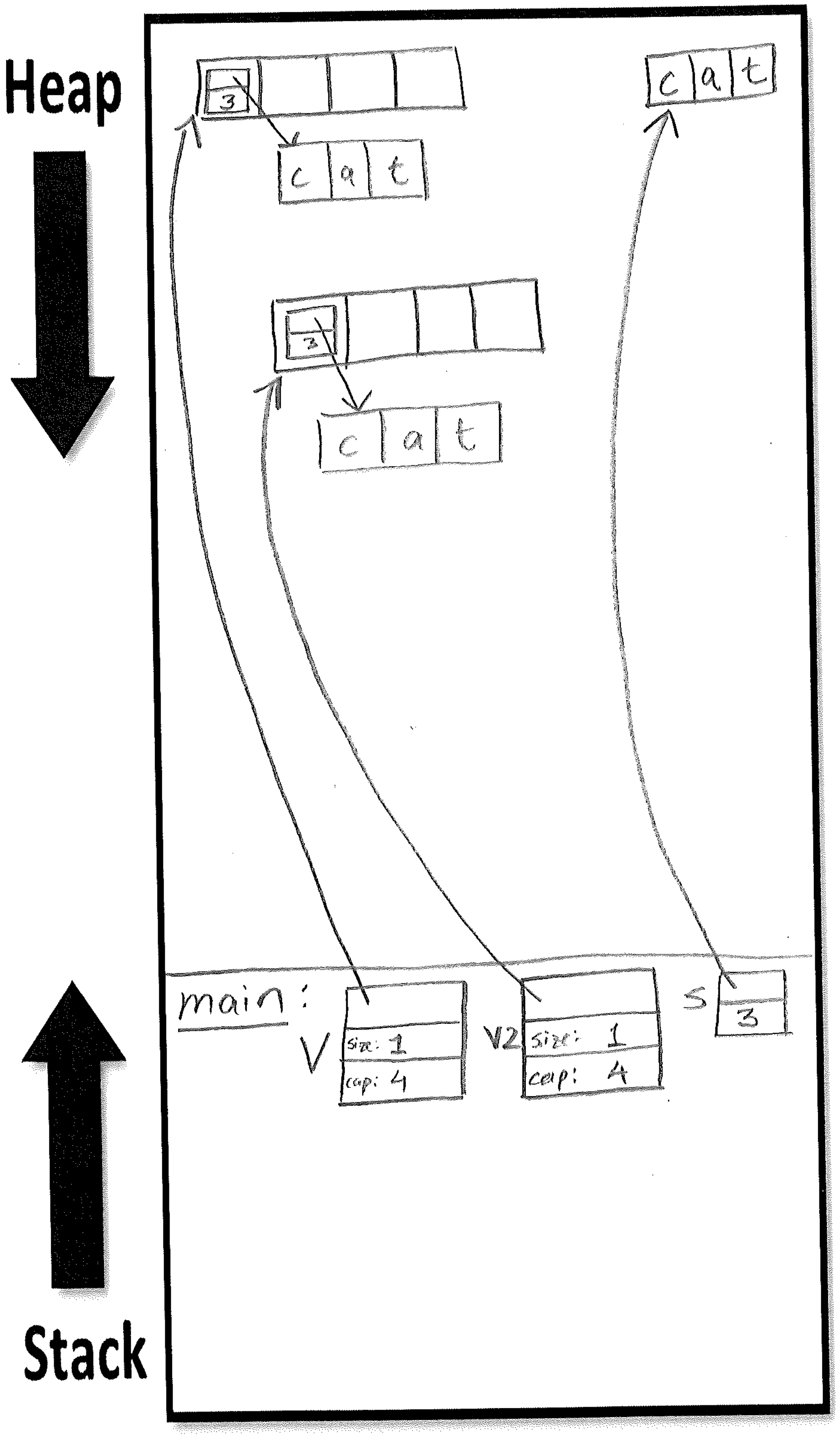
NOTE #1: no questions during the exam. Answer the question as given; if you think the question is ambiguous, explain your reasoning and answer based on that reasoning. You will be asked to write C++, do your best to recall actual C++ syntax and functions. Do not ask for help / clarifications.

NOTE #2: do not write on the back pages of this exam. We scan only the front pages for grading purposes. If you need more room for an answer, make a note and continue here on the front page.

Diagramming [10 points]

1) Consider the following C++ program; assume necessary C++ boilerplate (#include, etc.) is present. Stop and draw the state of memory where it says "stop here". Your answer must follow the conventions used in class lectures; label stack frame(s) with the function name.

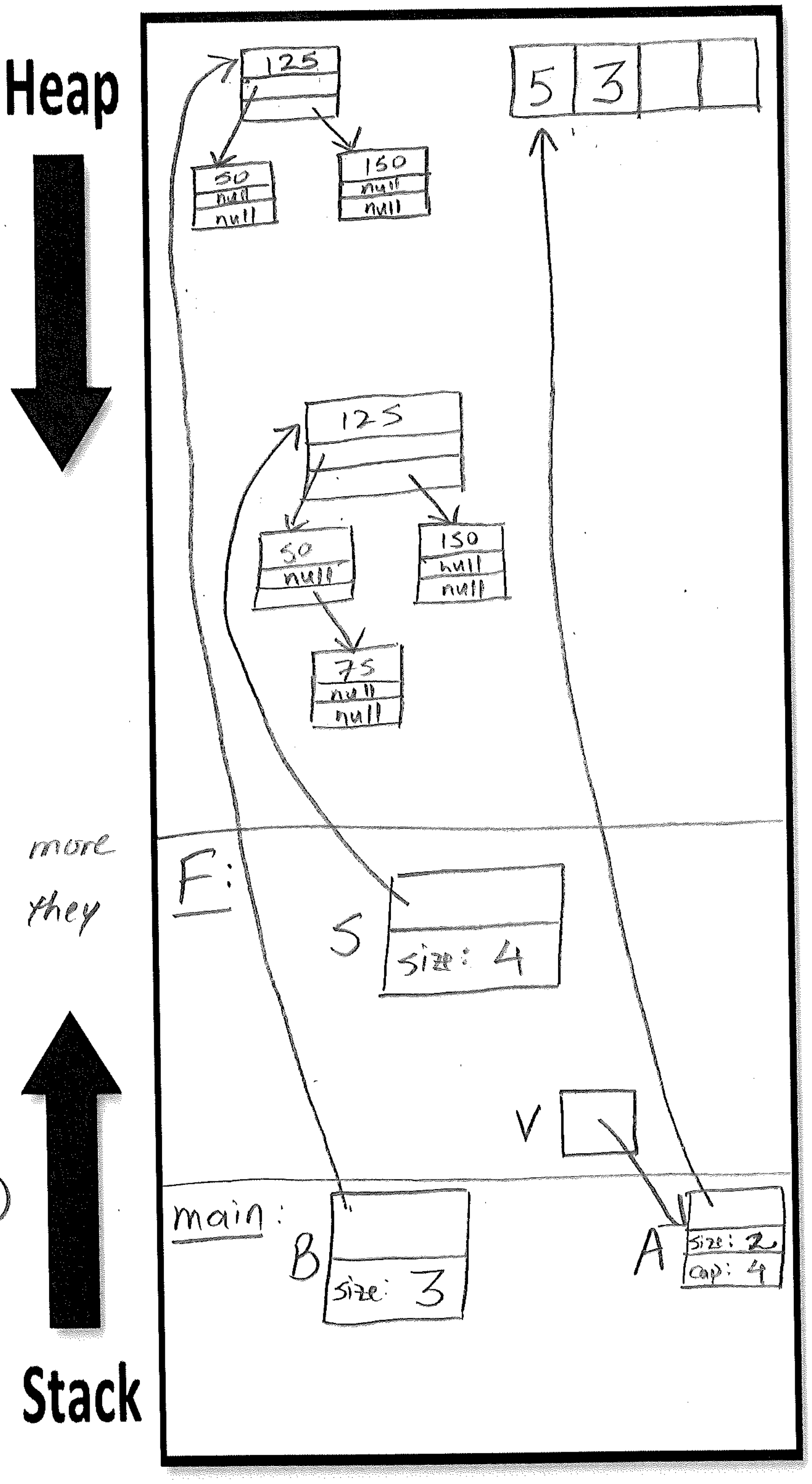
```
int main()
{
   vector<string>  V;
   string          S;

   S = "cat";

   V.push_back(S);

   vector<string> V2 = V;

   // stop here
```



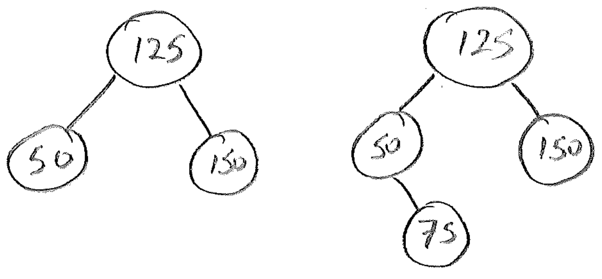
Diagramming [10 points]

2) Consider the following C++ program; assume necessary C++ boilerplate (#include, etc.) is present. Stop and draw the state of memory where it says “stop here”. Your answer must follow the conventions used in class lectures; label stack frame(s) with the function name. Assume the initial capacity of a vector is 4.

```
void F(set<int> S, vector<int>& V)
{
   S.insert(75);
   V.push_back(3);

   // stop here
}


int main()
{
   vector<int>  A;
   set<int>     B;

   A.push_back(5);

   B.insert(125);
   B.insert(50);
   B.insert(150);
   B.insert(50);

   F(B, A);
```



Note: If BSTs were drawn more to look like class examples, they would be:



C++ programming [80 points]

You're going to write a C++ program working with products for sale by the Anabru company. This will require a series of steps. Commenting is not required, and you do not need to #include files. But you are expected to write correct C++ code (small syntax errors are fine).

class **Product**
{

};

Assume the existence of a **Product** class that denotes one product for sale. The Product class stores information about one product: a unique ID, the product's name, the cost to purchase one instance of this product, and the number of instances currently in stock and ready to be shipped. The Product class provides *getters* for retrieving a product's information, and one *setter* for changing the number of instances in stock:

```
class Product
{
private:
  int    ID;      // product's unique ID
  string Name;    // product's name
  double Cost;    // cost of one instance of this product
  int    Instock; // quantity of this product in stock

public:
  Product(int id, string name, double cost, int instock);

  int    get_ID();
  string get_Name();
  double get_Cost();
  int    get_Instock();

  void set_Instock(int new_value);
};
```

<< continued on the next page >>

Your job is to develop a **Products** class that keeps track of the Product objects. The constructor for the Products class takes a filename, inputting the products from this file and storing the resulting Product objects into a *map* using the product ID as the key. The Products class also provides three methods for interacting with the product objects. These methods are described in the comments below:

class **Products**
{

};

```
class Products
{
private:
  map<int, Product> products;  // <product id, Product>

public:
  //
  // Opens the given file, inputs the products, and stores as
  // Product objects in the map.
  //
  Products(string filename);

  //
  // Returns the total value of all the products based on product
  // cost and quantity instock. If there are no products, the total
  // inventory is 0.0.
  //
  double totalInventory();

  //
  // Searches for the product with the matching pid, returning true if
  // found and false if not. If true is returned, the product's name,
  // cost, and quantity instock are returned via the reference parameters.
  // If false is returned, no change is made to the map or the parameters.
  //
  bool getProduct(int pid, string& name, double& cost, int& instock);

  //
  // Searches for the product with the matching pid, and if found updates
  // the product's quantity instock to the given value. If the product is
  // not found, this function makes no changes to the map.
  //
  void updateProductInstock(int pid, int new_instock);
```

Your job is to implement the constructor and three additional methods on the following pages.

3) Implement the constructor for the **Products** class, which takes a filename, inputs the products from this file, and stores the resulting Product objects in the private *products* map using the product ID as the key. The file contains one product per line, with commas separating the values in this order: id, cost, quantity instock, and the name. Example input file:

```
123,10.00,3,coffee mug
456,20.00,5,baseball hat
789,150.00,2,noise-canceling headphones
987,999.00,0,laptop
```

Assume the file exists and contains no errors; minimize object copying if possible, but not required.

```
Products::Products(string filename)
{
    ifstream file(filename);
    if (!file) {   //redundant since file exists but doesn't hurt
        cerr << "file not found" << endl;
        return 1;
    }
    string line;
    While (getline(file, line)) {
        stringstream parser(line);
        string pid, name, cost, instock;
        getline(parser, pid, ',');
        getline(parser, cost, ',');
        getline(parser, instock, ',');
        getline(parser, name);
        this->products[stoi(pid)] = Product(stoi(pid), name,
            stod(cost), stoi(instock)); // assuming stod converts
            string to double
    }
}
```

4) Implement the **totalInventory( )** method in the **Products** class. You cannot change the design of the Product class and you cannot change the design of the Products class --- work with what you are given.
Example: suppose these were the products input from "products.txt":

```
123, 10.00, 3, coffee mug
456, 20.00, 5, baseball hat
789, 150.00, 2, noise-canceling headphones
987, 999.00, 0, laptop
```

The total inventory is 430.00.

```
//
// Returns the total value of all the products based on product
// cost and quantity instock. If there are no products, the total
// inventory is 0.0.
//
double Products::totalInventory()
{
    double inventory = 0.0;
    for (auto product : this->products) {
        inventory += product.second.get_Cost()*product.second.get_Instock();
    }
    return inventory;
}
```

) Implement the **getProduct( ) method in the **Products** class. You cannot change the design of the Product class and you cannot change the design of the Products class --- work with what you are given. The "pid" is the product id.

NOTE: you must perform an efficient O(lgN) search (or call an efficient map search method) --- answers performing linear search will not be graded.

```
//
// Searches for the product with the matching pid, returning true if
// found and false if not. If true is returned, the product's name,
// cost, and quantity instock are returned via the reference parameters.
// If false is returned, no change is made to the map or the parameters.
//
bool Products::getProduct(int pid, string& name, double& cost, int& instock)
{
    if (! this->products.find(pid)) {
        return false;
    } else {
        Product p = this->products[pid];
        name = p.get_Name();
        cost = p.get_Cost();
        instock = p.get_Instock();
        return true;
    }
}
```

J) Implement the **updateProductInstock( )** method in the **Products** class. You cannot change the design of the Product class and you cannot change the design of the Products class --- work with what you are given. The "pid" is the product id.

NOTE: you must perform an efficient O(lgN) search (or call an efficient map search method) --- answers performing linear search will not be graded.

```
//
// Searches for the product with the matching pid, and if found updates
// the product's quantity instock to the given value. If the product is
// not found, this function makes no changes to the map.
//
void Products::updateProductInstock(int pid, int new_instock)
{
    if (!this->products.find(pid)) {
        return;
    } else {
        this->products[pid].set_Instock(new_instock);

    }
}
```

,) Now it's time to write the main program, using the **Product** and **Products** classes. The program starts by inputting the products from the file "products.txt", outputting the total inventory. Then main( ) reads sales orders from the file "orders.txt", and computes the total sales based on product cost and quantity in stock. The program also computes the total lost sales, i.e. sales that didn't occur because of insufficient stock. For example, suppose these were the products input from "products.txt":

```
123, 10.00, 3, coffee mug
456, 20.00, 5, baseball hat
789, 150.00, 2, noise-canceling headphones
987, 999.00, 0, laptop
```

Now suppose we have the following sales orders in "orders.txt". The first column is the quantity ordered, and the second column is the product id --- the text shown on the right is not part of the file, but explains what happens when this sales order is processed by the program:

2 123	<---- sold 2, 1 now instock
10 456	<---- sold 5, 0 now instock, 5 in lost sales
1 999	<---- no such product, ignored
1 987	<---- sold 0, 0 still instock, 1 in lost sales
1 789	<---- sold 1, 1 now instock
2 123	<---- sold 1, 0 now instock, 1 in lost sales
0 789	<---- sold 0, 1 still instock

When the program runs, it outputs the total inventory, the total sales, and the total lost sales. For the products and sales orders shown above, the program outputs the following:

```
Total inventory value: $430.00
Total sales: $280.00
Lost sales: $1109.00
```

Write main( ), continuing your work onto the next page --- DO NOT WRITE ON THE BACK OF THIS PAGE (if you need more space, continue onto page 01 of the exam). main() will need to open and input the sales orders from "orders.txt"; assume "orders.txt" exists and contains two columns as shown. You must use the *Product* & *Products* classes, as defined, to interact with the products.

```
int main()
{
   cout << setprecision(2) << fixed;  // for outputting monetary values:
   Products products = Products("products.txt");
   ifstream file("orders.txt");  // no error checking throughout
   double total_sales, lost_sales;
   total_sales = lost_sales = 0.0;
   string line;
   while (getline (file, line)) {
       stringstream parser(line);
       string sorder, spid; // create strings to store order & pid
       getline (line, sorder, ' '); // look for a space char
       getline (line, spid);
```

→ next pg

```
        int order, pid;
        order = stoi (sorder);
        pid = stoi (spid);
        string pname;
        double pcost;
        int pinstock;
        if (!products.getProduct (pid, pname, pcost, pinstock) {
            continue;
        } else {
            if (order > pinstock) {
                total_sales += pinstock * pcost;
                lost_sales += (pinstock - order) * pcost;
                products.updateProductInstock (pid, 0);
            } else {
                total_sales += order * pcost;
                products.updateProductInstock (pid, pinstock - order);
            }
        }
    } // while loop to read orders
    cout << "Total inventory value: $" << products.totalInventory <<
        endl;
    cout << "Total sales: $" << total_sales << endl;
    cout << "Lost sales: $" << lost_sales << endl;
} // main
```