Programming Assignment 3 (PA3) - Revision 2
COVID Outbreak: Escape the Fall Semester
User-Defined Types, Encapsulation, Operator Overloading, Interacting Objects
Out: October 20, 2021, Monday -- DUE: November 24, 2021, Wednesday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2021

Total: 200 points

- *You may use any development environment you wish, as long as it is ANSI C++ 11 compatible. Please make sure your code compiles and runs properly under the Linux/Unix environment on the PHO 305/307 (or eng-grid) machines before submitting.*
- *PAs may be submitted up to a week late at the cost of a **30% fixed penalty** (e.g., submitting a day late and a week late is equivalent). It is in your best interest to complete as many checkpoints as possible before the deadline. If you have missing implementations in your original submission, you may complete and submit the missing solutions during the following week. Any submissions after the deadline will be subject to the 30% penalty. No credit will be given to solutions submitted after the 1-week late submission period following the deadline.*
- *Follow the assignment submission guidelines in this document or you will lose points.*

Submission Format (Must Read)

- Accept the PA3 GitHub Classroom assignment using this link**:** https://classroom.github.com/a/H14NOBgO
- Use the **exact** file names specified in each problem for your solutions.
- Push your changes to the repository generated by GitHub Classroom.
- Complete submissions should have **22 files.**
- Please do **NOT** submit *.exe and *.o or any other files that are not required by the problem.
- **Code must compile in order to be graded. Otherwise, this is an automatic zero.**
- Comment your code (good practice!). We **may** use your comments when grading.

Coding Style (reminder from PA2)

As you become an experienced programmer, you will start to realize the importance of good programming style. There are many coding “guidelines” out there and it is important that you become to adopt your own. Naming conventions will help you recognize variable names, functions, constants, classes etc. Similarly, there are many ways to elegantly format your code so that it is easy to read. All of these issues do not affect compilation and hence are easy to overlook at this stage in your development as a programmer. However, your ability (or inability) to create clean, readable code could be the difference in your career when you leave college.

Good reference: https://google.github.io/styleguide/cppguide.html
If you find any other good references, feel free to post the link on Piazza!

COVID Outbreak: Escape the Fall Semester

In this programming assignment, you will be implementing a “simulation” that involves COVID 19 consisting of objects located in a two-dimensional world that move around and behave in various ways. The user enters commands to tell the objects what to do, and they behave in simulated time. Simulated time advances one “tick” or unit at a time. Time is “frozen” while the user enters commands. When the user commands the program to “go”, time will advance one tick of time. When the user commands the program to “run”, time will advance several units of time until some significant event happens (to be described later).

How to Play:

You are a BU senior engineering student. You are trying to graduate but you need to stay heathy so you can finish your classes on time. However, the BU campus in-person classes are a potential hot spot to catch COVID. You must go to classes and earn credits by doing assignments. However, you also need to periodically get vaccine booster shots if you want to make it through the semester. You “win” the game by getting all the credits you need without being hospitalized. You can compare games with other students by seeing who completes all their classes the fastest.

In this assignment, you will be implementing these 9 classes:

- Point2D – represents a point on a Cartesian coordinate system.
- Vector2D – represents a vector in the real plane.
- GameObject – base class for all objects in the game.
- Building – base class for all building objects in the game.
- DoctorsOffice – DoctorsOffices can be used to recover antibodies but they have a limit on usage. Inherits various member variables from Building.
- ClassRoom – location for students to take classes. “Credits” are gained through completing “assignments”. Inherits various member variables from Building.
- Student – a simulated Student which has three abilities:
    - Move to a specified location
    - Recover antibodies from a specified DoctorsOffice
    - Gain Credits from a specified ClassRoom
- View - Displays game objects. More details to come (*to be described later*).
- Model - Holds references to game objects. Model details to come (*to be described later*).

- You will also provide a set of separate, related functions combined in one .cpp and one .h file:
- GameCommand.cpp/.h - Handles commands from the user input. (2 files)
- You will also be turning in main.cpp and a Makefile.

Class Specifications

Each class and its members are described below by listing its name, the prototypes for the member functions, and the names and types of the member variables. You **MUST** use the prototypes, types, and names in your program as specified here, in the same upper/lower case. Function argument names are allowed to be different. **Failure to follow these specifications will result in lost points.**

Point2D (10 points)

This class contains two double values, which will be used to represent a set of (x, y) Cartesian coordinates. This class and Point2D (described below) will be used to simplify keeping track of the coordinates of each object in the game, and updating their locations as they move. All data members and functions for this class should be **public**.

Public Members

- double x
  - The x value of the point.
- double y
  - The y value of the point.

Constructors

- The default constructor initializes x and y to 0.0
- Point2D(double in_x, double in_y)
  - Sets x and y to in_x and in_y, respectively.

Non-member Functions

- double GetDistanceBetween(Point2D p1, Point2D p2)
  - Returns the Cartesian (ordinary) distance between p1 and p2.

Non-member Overloaded Operators (assume p1 and p2 represent two Point2D objects, and v1 represents a Vector2D object)

- Stream output operator (<<): produces output formatted as (x, y)
  - Example: If p1 has x = 3.14, y = 7.07 then cout << p1 will print (3.14, 7.07)

- Addition operator (+): p1 + v1 returns a **Point2D** object with x = p1.x + v1.x and y = p1.y + v1.y
  - Example: If p1 has x = 3, y = 7 and v1 has x=5, y=-2 then this function should make a new **Point2D** with x = 8 and y = 5;

- Subtraction operator (-): p1 - p2 returns a **Vector2D** object with x = p1.x - p2.x and y = p1.y - p2.y
  - Example: If p1 has x = 3, y = 7 and p2 has x=5, y=-2 then this function should make a new **Vector2D** with x = -2 and y = 9;

Vector2D (10 points)

This class also contains two double values, but it is used to represent a vector in the real plane (a set of x and y displacements). The overloaded operators allow one to do simple linear-algebra operations to compute where an object's new location should be as it moves around. Some of the overloaded operators and other functions can be member functions, others cannot. All data members and functions for this class should be public.

Public Members

- double x
  - The x displacement value of the vector.
- double y
  - The y displacement value of the vector.

Constructors

- The default constructor that initializes x and y to 0.0
- Vector2D(double in_x, double in_y)
  - sets x and y to in_x and in_y, respectively.

Non-member Overloaded Operators (assume v1 represents a **Vector2D** object and d represents a non-zero double value)

- Multiplication operator (*): v1 * d returns a **Vector2D** object with x = v1.x *d and y = v1.y * d
  - Example: If v1 has x=10 and y=20 and d=5 then this function should make a new **Vector2D** with x=50 and y =100.

- Division operator (/): v1 / d returns a **Vector2D** object with x = v1.x / d and y = v1.y / d
  - Example - If v1 has x=10 and y=20 and d=5 then this function should make a new **Vector2D** with x=2 and y =4.
  - Dividing by zero should just create v1.

- Stream output operator (<<): produce output formatted as <x, y>
  - Example) If v1 has x = 5.3, y = 2.4 then cout << v1 will print <5.3, 2.4>
  - Notice that this is < and > NOT (and) (like for **Point2D**).

GameObject (20 points)

This class is the base class for all the objects in the game. It is responsible for the member variables and functions that they **all** have in common. It has the following members:

Protected members:

- Point2D location;
  - The location of the object
- int id_num
  - This object's ID
- char display_code;
  - How the object is represented in the View.
- char state;
  - State of the object; more information provided in each derived class.

Public members:

- GameObject(char in_code);
  - Initializes the display_code to in_code, id_num to 1, and state to 0. It outputs the message: **"GameObject constructed"**.
- GameObject(Point2D in_loc, int in_id, char in_code,);
  - Initializes the display_ code, id_num, and location. The state should be 0. It outputs the message: **"GameObject constructed"**.
- Point2D GetLocation();
  - Returns the location for this object.
- int GetId();
  - Returns the id for this object
- char GetState();
  - Returns the state for this object.
- void ShowStatus();
  - Outputs the information contained in this class: display_code, id_num, location. i.e. **"(display_code)(id_num) at (location)"**. See sample output for exact formatting.

(Notice that later you will be adding a few other functions including **pure virtual functions** called "Update() and ShouldBeVisible()" This will come when you learn about the "Model").

CHECKPOINT I

Start by writing the Point2D and Vector2D classes and a couple of their functions. Add the additional functions one at a time and test each one. Write a TestCheckpoint1.cpp file with a main function to test them. There, create multiple Point2D and Vector2D objects in order to test their constructors and overloaded operators (<<, +, -, *, /). Getting the overloaded output operator to work soon will make testing the remaining functions more fun. Create GameObject and instantiate a few cases in your main function and run the ShowStatus() function. Convince yourself that the objects have been created correctly and that their member variables have the right values.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT If you do not make sure that these two objects work fully, the rest of your game will NOT work. Use this time to also start to modify the provided makefile to also work for testing Checkpoint 1.

Building (10 points)

This class is the base class for all building type objects in the game. It inherits from GameObject and is responsible for tracking the total number of Students that enter or leave a building.

Private Members

- unsigned int student_count
  - The number of Students currently within this Building
  - Initial value should be set to 0.

Constructors

- The default constructor that initializes the member variables to their initial values:
  - display_code should be 'B'
  - Should print out the message **"Building default constructed"**.
- Building(char in_code, int in_Id, Point2D in_loc)
  - Initializes the id number to in_id, and the location to in_loc, display_code to in_code, and remainder of the member variables to their default initial values.
  - Prints out the message **"Building constructed"**.

Public Member Functions

- void AddOneStudent();
  - Increments student_count by one.
- void RemoveOneStudent();
  - Decrements student_count by one.
- void ShowStatus();
  - Prints **"(display_code)(id) located at (location)"**
  - Prints "**(student_count) students is/are in this building"**
  - Note: Instead of is/are try to conditionally select which of the verbs to use when printing
- bool ShouldBeVisible();
  - Returns true because buildings are always visible

DoctorsOffice (20 points)

This class has a location and a set amount of vaccines. It also has a display_code letter and an id number that are used to help identify the object in the output. DoctorsOffice inherits from Building.

State Machine Enums:

The following enumerated type should be declared in the DoctorsOffice.h

```
enum DoctorsOfficeStates {
        VACCINE_AVAILABLE       = 0,
        NO_VACCINE_AVAILABLE = 1
};
```

Private Members

- unsigned int vaccine_capacity
    - The maximum number of vaccines this DoctorOffice can hold.
- unsigned int num_vaccine_remaining
    - The amount of vaccine currently in this DoctorsOffice
    - Initial value should be set to vaccine_capacity.
- double dollar_cost_per_vaccine
    - The per vaccine cost in DoctorsOffice

Constructors

- The default constructor that initializes the member variables to their initial values:
    - display_code should be 'D'
    - vaccine_capacity should be 100 **(Piazza updates may change this as we tweak the game)**
    - num_vaccine_remaining should be set to vaccine_capacity
    - dollar_cost_per_vaccine should be set to 5 **(Piazza updates may change this as we tweak the game)**
    - state should be VACCINE_AVAILABLE.
    - Should print out the message **"DoctorsOffice default constructed"**.

- DoctorsOffice (int in_id, double vaccine_cost, unsigned int vaccine _cap, Point2D in_loc);
    - Initializes the id number to in_id, and the location to in_loc, dollar_cost_per_vaccine to vaccine_cost and vaccine_capacity to vaccine_cap. The rest of the variables are assigned default values.
    - Prints out the message **"DoctorsOfffice constructed"**.
    - State should be VACCINE_AVAILABLE.

Public Member Functions

- bool HasVaccine()
  - Returns true if this DoctorsOffice contains at least one vaccine.
  - Returns false otherwise.
- unsigned int GetNumVaccineRemaining()
  - Returns the number of vaccines remaining in this DoctorsOffice.
- bool CanAffordVaccine(unsigned int vaccine, double budget)
  - Returns true if this Student can afford to purchase vaccine with the given budget.
- double GetDollarCost(unsigned int vaccine)
  - Returns the dollar cost for the specified number of vaccines
- unsigned int DistributeVaccine(unsigned int vaccine_needed)
  - If the amount num_vaccine_remaining in the DoctorsOffice is greater than or equal to vaccine_needed, it subtracts vaccine_needed from DoctorsOffice amount and returns vaccine_needed. If the amount of vaccine in the DoctorsOffice is less, it returns the DoctorsOffince current amount, and the DoctorsOffice vaccine amount is set to 0.
- bool Update()
  - If the DoctorsOffice has no vaccine remaining
    - Its state is set to NO_VACCINE_AVAILABLE
    - display_code is changed to 'd'
    - Prints the message **"DoctorsOffice (id number) has ran out of vaccine."**
    - Returns true if vaccine is depleted; returns false if it is not depleted.
  - This function shouldn't keep returning true if the DoctorsOffice has no vaccine remaining. It should return true ONLY at the time when the DoctorsOffice runs out of vaccine, and return false for later Update() function calls.
- void ShowStatus()
  - Prints out the status of the object:
    - **"DoctorsOffice Status: "**
    - Calls Building::ShowStatus()
    - **"Dollars per vaccine: (cost_per_vaccine)"**
    - **"has (vaccine_remaining) vaccine(s) remaining. "**

ClassRoom (20 points)

A ClassRoom object has a location and an amount of assignments. You can educate your Student in a ClassRoom and earn credits. It also has a display_code letter and id number that are used to help identify the object in the output. It has the following members. For clarity, the private members are described first, although normally they are declared after the public members in the code. ClassRoom objects should inherit from Building.

State Machine Enums:

The following enumerated type should be declared in the ClassRoom.h

```
enum ClassRoomStates {
        NOT_PASSED  = 0,
        PASSED          = 1
};
```

Private Members

- unsigned int num_assignments_remaining
  - The amount of assignments remaining in the ClassRoom
- unsigned int max_number_of_assignments
  - Assignment capacity for this ClassRoom
- unsigned int antibody_cost_per_assignment
  - Antibody cost for a single assignment (it tires you out when you work)
- double dollar_cost_per_assignment
  - Dollar cost for single assignment
- unsigned int credits_per_assignment
  - Amount of credits gained for each assignment

Constructors

- The default constructor that initializes the member variables to their initial values:
  - Display Code: 'C'
  - State: NOT_PASSED .
  - max_number_of_assignments should be 10 **(Piazza updates may change this as we tweak the game)**
  - num_assignments_remaining should be set to max_number_of_assignments
  - antibody_cost_per_assignment should be 1 **(Piazza updates may change this as we tweak the game)**
  - dollar_cost_per_assignment should be 1.0 **(Piazza updates may change this as we tweak the game)**
  - credits_per_assignment should be 2 **(Piazza updates may change this as we tweak the game)**
  - Prints out the message **"ClassRoom default constructed"**.

- ClassRoom(unsigned int max_assign, unsigned int antibody_cost, double dollar_cost, unsigned int crd_per_assign int in_id, Point2D in_loc)
  - Initializes the id number to in_id
  - max_number_of_assignments to max_assign,
  - antibody_cost_per_gpa to antibody_cost,
  - credits_per_assignment to crd_per_assign,
  - dollar_cost_per_assignment to dollar_cost and location to in_loc
  - Initializes the rest of the member variables to default values
  - Prints out the message **"ClassRoom constructed"**.

Public Member Functions

- double GetDollarCost(unsigned int unit_qty)
  - Returns the cost of purchasing “unit_qty” assignments
- unsigned int GetAntibodyCost(unsigned int unit_qty)
  - Returns the amount of antibody required for “unit_qty” assignment
- unsigned int GetNumAssignmentsRemaining()
  - Returns the number of assignments remaining in this ClassRoom.
- bool IsAbleToLearn(unsigned int unit_qty, double budget, unsigned int antibodies)
  - Returns true if a ClassRoom with a given budget and antibodies can request to take unit_qty assignment
  - Returns false otherwise
- unsigned int TrainStudents(unsigned int assignment_units)
  - Subtracts assignments from num_assigments_remaining if this ClassRoom has enough units. If the amount of units requested is greater than the amount available at this ClassRoom, then num_assignments_remaining will be used instead of assignments when calculating credits.
  - Returns the number of credits gained by completing the assignments
  - credits points can be calculated using (number of assignments) * credits _per_assignment
- bool Update()
  - If the ClassRoom has zero assignments remaining, set the state to PASSED and display_code to ‘c’. Then print the message “**(display_code)(id) has been passed”**.
  - Returns false if assignments are still remain within the ClassRoom.
  - This function shouldn’t keep returning true if the ClassRoom is passed. It should return true ONLY at the time when the ClassRoom is passed, and return false for later “Update()” function calls.
- bool passed ();
  - Returns true if assignments remaining is 0
- void ShowStatus()
  - Prints out the status of the object by calling GameObject’s show status and then the values of its member variables:
    - **“ClassRoomStatus: “**
    - Calls Building::ShowStatus()
    - **“Max number of assignments: (max_number_of_assignments)”**
    - **“Antibody cost per assignment: (antibody_cost_per_assignment)”**
    - **“Dollar per assignment: (dollar_cost_per_assignment)”**
    - **“Credits per assignment: (credits_per_assignment)”**
    - **“(num_assignments_remaining) assignment(s) are remaining for this ClassRoom”**

How the Building, DoctorsOffice and ClassRoom Objects Behave

These objects also change their state, but they do so simply. The DoctorsOffice Update() function simply checks to see if there are any remaining vaccines for distribution. If there are no more vaccines remaining then its state changes from VACCINE_AVAILABLE to NO_VACCINE_AVAILABLE and the display code changes from 'D' to 'd'; true is returned to signify this event occurred. Similarly, the ClassRoom Update() function checks to see if the amount of assignments remaining is equal to zero; if it is then the ClassRoom's state changes from NOT_PASSED to PASSED and the display code is changed from 'C' to 'c'. They both inherit from GameObject. When a Student enters a building, the Building's student_count increases by one. Likewise, when a Student leaves a building, its student_count decreases by one.

The constructors should output the appropriate messages such as "Building constructed", "DoctorsOffice default constructed" and "ClassRoom default constructed" as before. The ShowStatus() functions for each building should output "**(class name) status:"**, then call the shadowed Building::ShowStatus() function, and then output the information specific to that building. Later, when the Student objects are created, you should see the proper sequence of messages from the constructors.

CHECKPOINT II

Write the Building, ClassRoom and DoctorsOffice classes. To test these classes, also write a TestCheckpoint2.cpp. Instantiate multiple objects of these classes in the main function and test out their functions (e.g. Update(), DistributeVaccine(), AddOneStudent(), etc.) in order to ensure their proper behavior. For example, call each object's ShowStatus() method after calling their Update() method.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT
If you do not make sure that Building, DoctorsOffice and ClassRoom work fully, the rest of your game will NOT work. Use this time to start to modify the provided makefile to also work for testing Checkpoint 2. DON'T RUSH THROUGH THIS.

Student (30 Points)

This class inherits from GameObject. It will represent objects that move around and can be commanded to do things. It is responsible for the data and functions for moving around, recovering antibodies and studying in ClassRooms.

State Machine Enums:

The following enumerated type should be declared in the Student.h

```
enum StudentStates {
        STOPPED                 = 0,
        MOVING                  = 1,
```

```
        INFECTED                = 2,
        AT_DOCTORS              = 3,
        IN_CLASS                = 4,
        MOVING_TO_DOCTOR        = 5,
        MOVING_TO_CLASS         = 6,
        STUDYING_IN_CLASS       = 7,
        RECOVERING_ANTIBODIES   = 8
};
```

Public members:

- Student();
  - It initializes the speed to 5 and outputs a message: **"Student default constructed."**
- Student (char in_code);
  - It initializes the speed to 5 and outputs a message: **"Student constructed."**
  - Also sets initial values as follows:
    - State: STOPPED
    - Display_code: in_code
- Student(string in_name, int in_id, char in_code, unsigned int in_speed, Point2D in_loc);
  - Initializes the speed to in_speed and sets name to in_name.
  - It outputs a message: **"Student constructed"**.
- void StartMoving(Point2D dest);
  - Tells the Student to start moving.
  - Calls the setup_destination() function.
  - Sets the state to MOVING
  - If this Student is already at the destination print "**(display_code)(id): I'm already there. See?"**
  - If this Student is infected print **"(display_code)(id): I am infected. I may move but you cannot see me."**
  - Otherwise prints **"(display_code)(id): On my way."**
- void StartMovingToClass(ClassRoom* classr);
  - Tells the Student to start moving to a ClassRoom.
  - Calls the SetupDestination() function with ClassRoom's location as the destination.
  - Sets the state to MOVING_TO_CLASS"
  - If this Student is infected print **"(display_code)(id): I am infected so I can't move to class…"**
  - Prints the message **"(display_code)(id): on my way to class (class id)."**
  - If this Student is already there print **"(display_code)(id): I am already at the Classroom!"**
- void StartMovingToDoctor(DoctorsOffice* office);
  - Tells the Student to start moving to a DoctorsOffice.
  - Calls the SetupDestination() function with DoctorsOffice's location as the destination.

- Sets the state to MOVING_TO_DOCTOR
- If this Student is infected print **"(display_code)(id): I am infected so I should have gone to the doctor's.."**
- Prints the message **"(display_code)(id): On my way to doctor's (office id)"**
- If Student is already there print **"(display_code)(id): I am already at the the Doctor's!"**

- void StartLearning(unsigned int num_assignments);
  - Tells the Student to start learning (num_assignments) in a ClassRoom.
  - Sets the state to STUDYING_IN_CLASS and prints the message **"(display_code): Started to learn at the ClassRoom (class id) with (number of assginments) assignments"** unless the following conditions are true:
    - This Student is too tired print **"(display_code)(id): I am infected so no more school for me..."**
    - This Student is not in a ClassRoom print **"(display_code)(id): I can only learn in a ClassRoom!"**
    - This Student does not have enough dollars print **"(display_code)(id): Not enough money for school"**
    - The current_class is done print **"(display_code)(id): Cannot lean! This Class has no more assignments!"**
  - If this Student can start training, set its credits_to_buy to the minimum of the requested assignments and update the remaining assignments in the class. This will be used when its Update() function is called.
- void StartRecoveringAntibodies(unsigned int num_vaccines);
  - Tells the Students to start recovering at a DoctorsOffice.
  - Sets the state to RECOVERING_ANTIBODIES and prints the message **"(display_code)(id): Started recovering (num_vaccines) vaccines at Doctor's Office (current_office_id)"** unless the following conditions are true:
    - This Student does not have enough Dollars print **"(display_code)(id): Not enough money to recover antibodies."**
    - The Doctor's Office does not have at least one vaccine remaining otherwise print **"(display_code)(id): Cannot recover! No vaccine remaining in this Doctor's Office"**
    - This Student is not in a Doctor's Office otherwise print **"(display_code)(id): I can only recover antibodies at a Doctor's Office!"**
  - If this Student can start recovering antibodies, set its vaccines_to_buy to the minimum of the requested vaccines and update the remaining vaccines in the office. This will be used when its Update() function is called.
  - Five antibodies are gained for each vaccine purchased. **(Piazza updates may change this as we tweak the game)**
- void Stop();
  - Tells this Student to stop doing whatever it was doing.
  - Sets the state to STOPPED.
  - Prints **"(display_code)(id): Stopping.."**

- bool IsInfected();
  - Returns true if antibodies is 0
- bool ShouldBeVisible();
  - Returns true if this Student is **Not** infected
- void ShowStatus();
  - Prints **"(name) status: "**
  - Call GameObject::ShowStatus()
  - Print state specific status information. Refer to **How Student Objects Behave** for complete details
- bool Update()
  - Check "How Student objects behave part"

Protected members:

- bool UpdateLocation();
  - Updates the object's location while it is moving (See "How the Student Moves" for details).
  - Prints **"(display_code)(id): I'm there!"** if a Student has arrived at its destination.
  - **Prints "(display_code)(id): step…"** otherwise.
- void SetupDestination(Point2D dest);
  - Sets up the object to start moving to dest. (See "How the Student Moves")

Private members:

- double speed;
  - The speed this object travels, expressed as distance per update time unit.
- bool is_at_doctor;
  - Set to true of the Student is in a DoctorsOffice
  - Initial value should be false.
- bool is_in_class;
  - Set to true if this Student is in a ClassRoom.
  - Initial value should be false.
- unsigned int antibodies;
  - Amount of antibodies a Student has
  - Initial value should be 20. **(Piazza updates may change this as we tweak the game)**
- unsigned int credits;
  - The amount of credits points this Student has.
  - Initial value should be 0.
- double dollars;
  - The total amount of dollars this Student holds.
  - Initial value should be 0.
- unsigned int assignments_to_buy;

    - Stores the number of assignments to buy when in a ClassRoom
    - Initial value should be 0.
- unsigned int vaccines_to_buy;
    - Stores the number of vaccines to buy when in a DoctorsOffice
    - Initial value should be 0.
- string name;
    - The name of this Student.
- DoctorsOffice* currrent_office
    - A pointer to the current DoctorsOffice.
    - Initial value should be 0 (NULL).
- ClassRoom* current_class
    - A pointer to the current ClassRoom.
    - Initial value should be 0 (NULL).
- Point2D destination;
    - This object's current destination coordinates in the real plane.
    - Point2D's default constructor will initialize this to (0.0, 0.0).
- Vector2D delta;
    - Contains the x and y amounts that the object will move on each time unit.
    - See "How Student Moves" for more information.

Non- members (static):

- double GetRandomAmountOfDollars();
    - Returns a random number between 0.0 and 2.0 inclusive. **(Piazza updates may change this as we tweak the game)**

How Student Objects Move

The main function of the program accepts commands from the user. Simulated time is stopped while the user enters commands. The user can command individual objects to move to specified destination coordinates. When the user tells the program to "go", one step of simulated time then happens, and the program calls the update function on every object. The program then pauses to let the user enter more commands, and when the user commands "go" again, another step of simulated time happens, and every object is updated again. So each "go" command corresponds to one "tick" of the clock, one step of the simulated time. The "run" command conveniently makes the program run until an important event happens (to be defined).

An object is commanded to move by calling its StartMoving() function (inside the Student class) and supplying the destination. The StartMoving() function does the following: Call the SetupDestination function to save the destination and calculate the delta value. Then set the object in the moving state. The delta value contains the amount that the object's x and y coordinates will change on each update step. We calculate it once and then apply it to each step. This is the purpose of the overloaded operators for Point2D and Vector2D. To calculate the value of delta, use:

delta = (destination – location) * (speed / GetDistanceBetween(destination, location))

In other words, the object will move in a straight line, moving a distance equal to its speed on each step. The change in the x and y values of the location on each step are thus proportional to the ratio of the speed to the distance. So the SetupDistance() function calculates the delta value to be used in the updating steps.

On each step of simulated time, the main routine will call each object's Update() function. The update function for Student does a variety of things, but if the object is moving, it calls the UpdateLocation() function. This function first checks to see if the object is within one step of its destination (see below). If it is, UpdateLocation() sets the object's location to the destination, prints an "arrived" message, and then returns true to indicate that the object arrived. If the object is not within a step of destination, UpdateLocation() adds the delta to the location, prints a "moved" messaged, and returns false to indicate that the object has not yet arrived. Thus the object will take a "speed-sized" step on each update "tick" until it gets within one step of the destination, and then on the last step, goes exactly to the destination.

Finally, notice that the user can command all of the Student objects to move to a destination, and then tell the program to "go", and all of the objects will start moving, and each will stop when it arrives at its destination. The objects are responsible for themselves!

***An object is within a step of the destination if the absolute value of both the x and y components of fabs(destination - location) are less than or equal to the x and y components of delta (use the fabs() function in math.h library). By checking our distance with very simple computations using the delta value, we don't have to calculate the remaining cart_distance and compare it to the speed using a slow square root function on every update step.*

How Student Objects Behave

The behavior of the Student class is programmed using an approach called a "state machine". A state machine is a system that can be in one of several states and behaves depending on what state it is in. It can either stay in the same state or change to a different state and behave differently. The neat feature of this approach is that you can easily specify a complicated behavior pattern by simply listing the possible states, and then with each state, describe the input/output behavior of the machine and whether it will change state (See http://en.wikipedia.org/wiki/Finite-state_machine for detail).

In the Student class, the Update() function will do something different depending on the state of the Student object. The state is represented with an enumerated type that contains a number code for the particular state. A good way to program this is to use a switch statement that switches on the state variable and has a case for each possible state. In each case, perform the appropriate action for the state, and if needed, change the state by setting the state variable to a different value. Then the next time the Update() function is called, the Student will do the appropriate thing for the current state. Thus, the Update() function contains nothing but a big switch statement.

Here are the states of Student, and what the Update() and ShowStatus() functions do for each state. Generally, the Update() function should return true whenever the state is changed and return false if it stays in the same state:

- STOPPED
  - The Student does nothing and stays in this state.
  - ShowStatus() prints **" stopped"**
  - Update() should return false
- MOVING
  - ShowStatus() prints **" moving at a speed of (speed) to destination <X, Y> at each step of (delta)."**
  - Update() should
    - Call UpdateLocation() to take a step;
    - if the object has arrived, set the state to STOPPED and return true
    - Otherwise, stay in the MOVING state.
- MOVING_TO_CLASS
  - ShowStatus() prints **" heading to Classroom (current_class id) at a speed of (speed) at each step of (delta)"**
  - Update() should
    - Call UpdateLocation().
    - If it has arrived, set the state to IN_CLASS, and return true
    - Otherwise stay in the MOVING state.
- MOVING_TO_DOCTOR
  - ShowStatus() prints **" heading to Doctor's Office (current_office id) at a speed of (speed) at each step of (delta)"**
  - Update() should
    - Call UpdateLocation(). If it has arrived, set the state to AT_DOCTORS, and return true
    - Otherwise stay in the current state.
- IN_CLASS
  - ShowStatus() prints **" inside Classroom (current_class id)"**
  - Update() should return false
- AT_DOCTORS
  - ShowStatus() prints **" inside Doctor's Office (current_office id)"**
  - Update() should return false
- STUDYING_IN_CLASS
  - ShowStatus() prints **" studying in Classroom (current_class id)."**
  - Update() should:
    - reduce Student antibodies based on total antibodies cost for the current class request
    - reduce the amount of dollars based on the dollar cost for the current class request

        - increase student credits based on credit gain for the current class request; this should be calculated using TrainStudent()
        - print **“** (name) completed (assignements_to_buy) assignment(s)! **”**
        - print **“** (name) gained (credits gained) credit(s)! **”**
        - Set state to IN_CLASS and return true
- RECOVERING_ANTIBODIES
    - ShowStatus() prints **“ recovering antibodies in Doctor’s Office (current_office id)”**
    - Update() should
        - Increase Antibodies; antibodies should be calculated by StartRecoveringAntibodies()
        - Reduce dollars by the total cost of vaccines for the current DoctorsOffice
        - Prints **“** (name) recovered (antibodies recovered) antibodies! **”**
        - Prints **"** (name) bought (vaccines_recieved) vaccine(s)! **"**
        - Set state to AT_DOCTORS and return true

In all states print the following:

- **“Antibodies: (antibodies)”**
- **“Dollars: (dollars)”**
- **“Credits: (credits)”**

Thus, if the Student is commanded by StartMoving to a destination, it goes into the moving state, starts moving, and then stops when it arrives and does nothing until commanded again. **For each “speed-sized” step the Student should decrease their antibodies by 1 and increase their dollar count by a random amount. This is true for any time the Student moves.**

If the Student is supposed to learn the following happens: the StartMovingToClass() function sets the current_class pointer member variable and the calls SetupDestination() to target the current_class and sets the state to MOVING_TO_CLASS. Once at a ClassRoom, Student’s state becomes IN_CLASS and they can now study to gain credits. Use the StartLearning() function to command a Student to complete a specified number of assignments. If the Student does not have enough antibodies or dollars, their state should remain the same. If, however, a Student can afford the assignments set the state to STUDYING_IN_CLASS. After one update tick, credits, antibodies and dollars are updated to reflect completing a training session. Finally, its state is then set to IN_CLASS.

Eventually, your Student will need to recover lost antibodies. Use StartMovingToDoctor() to command your Student to move to a DoctorsOfffice. StartMovingToDoctor() sets the state to MOVING_TO_DOCTOR and sets the current_doctor member pointer variable to the requested office. Once a Student reaches a doctor, its state is set to AT_DOCTORS. Antibodies can now be recovered through the StartRecoveringAntibodies() function which sets the Student's state to RECOVERING_ANTIBODIES. If the Student can afford the total amount of vaccine requested,

after one update tick, the Student's antibodies are updated and its state is set to AT_DOCTORS. Otherwise, the Student remains in its current state.

Note

- If a Student runs out of antibodies it should not be able to move!
    - When a Student runs out of antibodies Update() should print **"(name) is out of antibodies and can't move"**
    - The state should then be set to INFECTED
- When a Students leaves or enters a building it should call the building's RemoveOneStudent () and AddOneStudent() function respectively
- a Student can only recover antibodies when its state is AT_DOCTORS. Likewise, a Student can only study if its state is IN_CLASS.
- If a Student requests more vaccines or assignments then are available, they should get the available amount

Checkpoint III

Iteratively develop the Student class. Start implementing the class by only writing the constructors and a partial form of the ShowStatus() function; leave everything else out. To test the correct behavior, write another simple TestCheckpoint3.cpp file. In the main function, create a Student object, and call its ShowStatus() function. It should display the correct initial state of the objects, so you can verify if both the constructors and the ShowStatus() function work properly. Only then start implementing the StartMoving(), SetupDestination(), and UpdateLocation() functions, and STOPPED and MOVING parts of the update and ShowStatus() functions.

Change your trivial main function to call the StartMoving() function on your one Student object, showing its status, calling its Update() function, and showing its status again - it should be in a different location! Check that amount moved on the step is correct. Put in a few more calls of update and see if the Student stops like it should. With the help of DoctorsOffice and ClassRoom objects, you are able to train your Student and recover its antibodies after it becomes tired.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT
If you do not make sure that the Student class works fully, the rest of your game will NOT work. Use this time to also start to modify the provided makefile to also work for testing Checkpoint 3. DON'T RUSH THROUGH THIS.

The Model-View-Controller (MVC) Pattern

Model (15 points)

The Model is a central component in the MVC pattern and stores all game objects in memory. Hence, it contains various arrays of pointers to the instances of the Game Object class. Also it offers multiple methods to interact with the Controller and View components. Here, it has the following structure:

Private members:

- int time;

- the simulation time.

We have a set of arrays of pointers and a variable for the number in each array:

- GameObject * object_ptrs[10];
- int num_objects;
- Student * student_ptrs[10];
- int num_students;
- DoctorsOfffice * office_ptrs[10];
- int num_offices;
- ClassRoom * class_ptrs[10];
- int num_classes;

Each object will have a pointer in the object_ptrs array and also in the appropriate other array. For example, a Student object will have a pointer in the object_ptrs array and in the student_ptrs array.

Public members:

- Model();
  - It initializes the time to 0 and then creates the objects using **new**, and stores the pointers to them in the arrays as follows:
  - The list shows the object type, its id number, initial location, and subscript in object_ptrs, and the subscript in the other array.
    - Student 1 (5, 1), object _ptrs[0], student_ptrs[0]
    - Student 2 (10, 1), object _ptrs[1], student_ptrs[1]
    - DoctorsOffice 1 (1, 20), object_ptrs[2], office_ptrs[0]
    - DoctorsOffice 2 (10, 20), object_ptrs [3], office_ptrs [1]
    - ClassRoom 1 (0, 0), object_ptrs[4], class_ptrs[0]
    - ClassRoom 2 (5, 5), object_ptrs[5], class_ptrs[1]
  - Here is a description of the objects that should exist.
  - D1 is a DoctorsOffice
    - ID number is 1
    - initial location is (1, 20)
    - vaccine cost is 1
    - vaccine capacity is 100
  - D2 is a DoctorsOffice
    - ID number is 2
    - initial location is (10, 20)
    - vaccine cost is 2

        - vaccine capacity is 200
    - S1 is a Student
        - name is Homer
        - ID number is 1
        - initial location is (5, 1)
        - speed is 2
    - S2 is a Student
        - name is Marge
        - ID number is 2
        - initial location is (10, 1)
        - speed is 1
    - C1 is a ClassRoom
        - ID number is 1
        - location is (0, 0)
        - antibody cost is 1
        - dollar cost is 2
        - credit per assignment is 3
        - assignments is 10
    - G2 is a ClassRoom
        - ID number is 2
        - location is (5, 5)
        - antibody cost is 5
        - dollar cost is 7.5
        - credit per assignment is 4
        - assignments is 20
    - Set num_objects to 6, num_students to 2, num_offices to 2, num_class to 2;
    - Finally, output a message: **"Model default constructed"**

- ~Model();
    - the destructor deletes each object, and outputs a message: **"Model destructed."**

Note: For purposes of demonstration, add to GameObject a destructor function that does nothing except output a message: **"GameObject destructed."** Define similar ones for Student, DoctorsOfffice, and ClassRom. This is so that you can see the order of destructor calls for an object.

There are three functions that provide a lookup and validation services to the main program (Controller). They return a pointer to the object identified by an id number, depending on the type of object we are interested in. The functions search the appropriate array for an object matching the supplied id, and either return the pointer if found, or 0 if not.

- Student * GetStudentPtr(int id);
- DoctorsOffice * GetDoctorsOfficePtr(int id);

- ClassRoom * GetClassRoomPtr(int id);
- bool Update()
    - It provides a service to the main program. It increments the time, and iterates through the object_ptrs array and calls Update() for each object. Since GameObject::Update() will be made virtual, this will update each object.
    - returns true if any one of the GameObject::Update() calls returned true.
    - If the player finishes all the ClassRooms the game should print **"GAME OVER: You win! All assignments done!"**. The game should then exit. **Try using the exit function to achieve this.**
    - If **all** the Students are infected and can't move, print **"GAME OVER: You lose! All of your Students are infected!"**. The game should then exit. **Try using the exit function to achieve this.**
- void Display(View& view);
    - Provides a service to the main program. It outputs the time, and generates the view display for all of the GameObjects.
    - This will be created later, comment out for now.
- void ShowStatus();
    - It outputs the time and outputs the status of all of the GameObjects by calling their ShowStatus() function.

Implement the polymorphism in the class hierarchy as follows:

- Declare bool Update() to be a pure virtual function in GameObject. This makes GameObject an abstract base class and ensures that each of the derived classes will have defined an update function, or we get a linker error to tell us we have left it out.
- Declare ShowStatus() to be virtual in GameObject.
- Declare ShouldBeVisible() to be pure virtual function in GameObject.
- Important: Make the destructors in GameObject and Student virtual.

Your main function and its sub functions can now be radically simplified because they are no longer responsible for keeping track of all the objects or how many of them or what kinds there are. Remove all the declarations of Students, DoctorsOffices, and ClassRooms from main, and replace them with declaring a single Model object.

GameCommand (15 points)

The GameCommand represents the Controller of the MVC pattern and provides multiple functions that interpret user input in order to perform the appropriate actions.

You should create a set of functions that can be called from main to handle the processing of user provided commands. The command functions should use the Model member functions like

GetDoctorsOfficePtr() to check that an input id number is valid and get a pointer for the object if it is. The DoGoCommand() and DoRunCommand() functions can just call Model::Update() to update all the objects, and Model::Display() can be called to display the current view of the game. Note that some commands will cause messages to be printed. Each command function should be given the Model object (by reference). The function prototypes are listed below along with messages they print:

- void DoMoveCommand(Model & model, int student_id, Point2D p1);
  - If the command arguments are valid prints **"Moving (Student name) to (p1)"**
  - otherwise prints **"Error: Please enter a valid command!"**
- void DoMoveToDoctorCommand(Model & model, int student_id, int office_id);
  - If the command arguments are valid prints **"Moving (Student name) to doctors office (office_id)"**
  - otherwise prints **"Error: Please enter a valid command!"**
- void DoMoveToClassCommand(Model & model, int student_id, int class_id);
  - If the command arguments are valid prints **"Moving (Student name) to class (class_id)"**
  - otherwise prints **"Error: Please enter a valid command!"**
- void DoStopCommand(Model & model, int student_id);
  - If the command arguments are valid prints **"Stopping (Student name)"**
  - otherwise prints **"Error: Please enter a valid command!"**
- void DoLearningCommand(Model & model, int student_id, unsigned int assignments);
  - If the command arguments are valid prints **"Teaching (Student name)"**
  - otherwise prints **"Error: Please enter a valid command!"**
- void DoRecoverInOfficeCommand(Model& model, int student_id, unsigned int vaccine_needs);
  - If the command arguments are valid prints **"Recovering (Students name)'s antibodies"**
  - otherwise prints **"Error: Please enter a valid command!"**
- void DoGoCommand(Model& model, View& view);
  - prints **"Advancing one tick."**
- void DoRunCommand(Model& model, View& view);
  - prints **"Advancing to next event."**

Note that a command is valid if all arguments are valid. For example, a Student ID argument is valid if there exists a Student in the game with that ID.

Move all these Do**Command functions into a separate .h and .cpp file.

Your program should work as before. You should be able to command the Students to move around, learn at the ClassRoom, recover antibodies at the DoctorsOffice and list the status of all the objects. When the program terminates, you should see the correct sequence of destructor messages.

Here is a description of the commands and their input values:

- **m** ID x y
  - "move": command Student ID to move to location (x, y)
- **d** ID1 ID2

    - "move towards a DoctorsOffice": command Student ID1 to start heading to DoctorsOffice ID2.
- **c** ID1 ID2
    - **“**move towards a Classroom”: command Student ID1 to start heading towards ClassRoom ID2.
- **s** ID
    - "stop": command Student ID to stop doing whatever it is doing
- **v** ID vaccine_amount
    - **“**buy vaccines”: command Student ID to buy vaccine_amount of vaccine at a DoctorsOffice.
- **a** ID assignment_amount
    - **“**complete assignment_amount assignments at a ClassRoom”: command Student ID to complete assignment_amount of training units at a ClassRoom.
- **g**
    - "go": advance one-time step by updating each object once.
- **r**
    - "run": advance one-time step and update each object, and repeat until either the update function returns true for at least one of the objects, or 5 time steps have been done.
- **q**
    - "quit": terminate the program

You do NOT need to do error checking. This will be done in PA4.
You must have a separate command-handling function for each command that collects the input required for the command and calls the appropriate object member functions. We recommend using the switch statement to pick out the function for each command; this is the simplest and cleanest way to do this sort of program branching. For example, to handle the "move" command, the case would look like this:

```
case 'm':
        DoMoveCommand(m1, m2);
        break;
```

All input for this program must be done with the stream input operator >>.

Important*: Your program must "echo" each command to confirm it and to provide a record in the output of what the input command was. For example, if the user enters "m 1 10 15" the program should output something like "moving 1 to (10, 15)". This way, if output redirection is used to record the behavior of the program, the output file will contain a record of the input. If there is an error in the input, there should be an informative error message, but your program is not responsible for trying to output an exact copy of the erroneous input. So the echo does not have to be present or complete if there is an error in the input.*

CHECKPOINT IV

Implement the Model and the GameCommand and add them to your program. You now should be able to actively have a user manually enter commands and have them play your game. Write a TestCheckpoint4.cpp file and in the main function command the Studdents to move to classroom, buy vaccines, etc., and list the status of all game objects. Make sure you comment out parts that reference View as that does not exist yet. When the program terminates, you should see the correct sequence of destructor messages.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT
If you do not make sure that these Model and GameCommands work fully, the rest of your game will NOT work. Use this time to also start to modify the provided makefile to also work for testing Checkpoint 4. DON'T RUSH THROUGH THIS.

View (15 points)

Thanks to inheritance, we can easily add a better display using simple "ASCII graphics." This display will be like a game board – a grid of squares. Each object will be plotted in the grid corresponding to its position in the plane. As the object moves around its position on the grid will be changed. The object will be identified on the board by its display_code letter and its id_num value. **We will be assuming that the id_nums are all one digit in size at this time.** We will provide sample output to show what the display should look like.
The code for this display will be encapsulated in a class of an object called a "View" whose function is to provide a view of some objects. The View object has a member function Plot() that puts each object into display grid; it will ask GameObject to provide the two characters to identify itself. Member function Draw() will output the completed display grid, and Clear() will reset the grid to empty in preparation for a new round of plotting.

In class GameObject, add the following public member function:

- void DrawSelf(char * ptr);.
    - The function puts the display_code at the character pointed to by ptr, and then the ASCII character for the id_num in the next character.

Write the View class with its own .h and .cpp file to have the following members.

Constant defined in the header file

- const int view_maxsize = 20;
    - the maximum size of the grid array

Private members:

- int size;
    - the current size of the grid to be displayed; not all of the grid array will be displayed in this project.
- double scale;

- the distance represented by each cell of the grid

- Point2D origin;
  - the coordinates of the lower left-hand corner of the grid
- char grid[view_maxsize][view_maxsize][2];
  - an array to hold the characters that make up the display grid.
- bool GetSubscripts(int &out_x, int &out_y, Point2D location);
  - This function calculates the column and row subscripts of the grid array that correspond to the supplied location. Note that the x and y values corresponding to the subscripts can be calculated by (location - origin) / scale. Assign these to integers to truncate the fractional part to get the integer subscripts, which are returned in the two reference parameters. The function returns true if the subscripts are valid, that is within the size of the grid being displayed. If not, the function prints a message: **"An object is outside the display"** and returns false.

Public members:

- View()
  - It sets the size to 11, the scale to 2, and lets the origin default to (0, 0). No constructor output message is needed.
- void Clear();
  - It sets all the cells of the grid to the background pattern shown in the sample output.
- void Plot(GameObject * ptr);
  - It plots the pointed-to object in the proper cell of the grid. It calls get_subscripts and if the subscripts are valid, it calls DrawSelf() for the object to insert the appropriate characters in the cell of the grid. However, if there is already an object plotted in that cell of the grid, the characters are replaced with '*' and ' ' to indicate that there is more than one object in that cell of the grid.

Tip about base class pointers. C++ normally refuses to convert one pointer type to another. But it will allow a conversion from a Derived class pointer to a Base class pointer (upcasting). This allows you to plot all kinds of GameObjects with a single function.

Tip about C/C++ arrays: If a is a three-dimensional array, then a[i][j][k] is the i, j, k element in the array, and a[i][j] is a pointer to a one-dimensional array starting a[i][j][0].

- void Draw();
  - outputs the grid array to produce a display like that shown in the sample output The size, scale, and origin are printed first, then each row and column, for the current size of the display. Note that the grid is plotted like a normal graph: larger x values are to the right, and larger y values are at the top. The x and y axes are labeled with values for every alternate column and row. Use the output stream formatting facilities to save the format settings, set them for neat output of the

axis labels on the grid, and then restore them to their original settings. Specifications: Allow two characters for each numeric value of the axis labels, with no decimal points. The axis labels will be out of alignment and rounded off if their values cannot be represented well in two characters. This distortion is acceptable in the name of simplicity for this project

Now modify your main program to declare a view object. Where you previously did a ShowStatus() call for each object, replace that code with a first call to the View object's Clear() function, then call the plot function using each object, and then the Draw() function. Now you should have the graphical display. Thanks to the inheritance from GameObject, the display works for all three types of specific game objects.

Overall Structure of the Program (main.cpp)

(35 points for behavioral tests; an update will be posted on Blackboard regarding this requirement and sample output)

The main program should include a loop that reads a command, executes it, and then asks for another command. **You do NOT have to detect errors or bad input. PA4 will add these features.**

The program will declare two Student objects, two DoctorsOffice objects, and two ClassRoom objects as outlined in the Model class description. These objects will persist throughout the execution of the program.

The program starts by displaying the current time and the status of each object using the show status function. The main function then asks for a command and the user inputs a single character for the command, and main calls a function for the appropriate command. The function for the command inputs requests any required additional information from the user, and then carries out the command. If the user entered the "go" or "run" commands, the program repeats the main loop by displaying the time and current status and prompting for a new command. Otherwise, it continues to prompt for new commands. This enables the user to command more than one object to move before starting the simulation running again.

Additional Specifications

You MUST:

- Make use of the classes and their members; you may not write non-object oriented code to do things that the classes can do.
- Declare function prototypes for the functions that are not part of a class, such as those called by main(), and list the function definitions after main. This will improve the readability of your program.
- Make .h and .cpp files for each of your class with data fields and member functions as specified above.
- Make sure your code compiles!

Programming Guideline

Unless you have so much experience that you shouldn't be in this course, trying to write this program all at once is the **hard** way to do it! Object oriented programming is easy to do in chunks! That's the idea! The individual classes can be written and tested pretty much one at a time, and a piece at a time. Here's how to write this program a chunk at a time:

1. Start by writing the Point2D and Vector2D classes and a couple of their functions. Write a trivial version of main to test them by creating one of each and doing things with them. Add the additional functions one at a time, and test each one. Getting the overloaded output operator to work soon will make testing the remaining functions more fun. Also, write the GameObject class and test with your trivial main.

2. Write the DoctorsOffice and ClassRooms classes. Add to your testing main function a declaration of an DoctorsOffice and ClassRoom object, and call TrainStudets() and DistributeVaccine(), then Update() and ShowStatus() on the objects multiple times

3. Start on writing the Student class, but only write the constructors and a partial form of the ShowStatus function; leave everything else out. Write another trivial testing main; create a Student object and a ShowStatus() function. It should display the correct initial state of the object to verify both the constructors and the ShowStatus function. Then, add the StartMoving(), stop, SetupDestination() function, and UpdateLocation() functions. Write the ShowStatus() and Update() functions. After testing this class, follow the same procedure to make the Student class. When working on ShowStatus() and Update(), add the STOPPED and MOVING parts first. Change your trivial main function to call the StartMoving() function on your one Student object. This should show its status, call its Update() function, and show its status again - it should be in a different location! Check that the amount moved on the step is correct. Put in a few more calls of update and see if the Student stops as it should.

4. Write the command loop for main(), and put the whole program together, and test it by making the objects all move around. See if you can have the two Students recover in two DoctorsOffices in various combinations at the same time.

Makefile

Makefiles automate and facilitate the compilation process of software projects that consists of a large number of source code (.cpp) files (such as PA3). Instead of specifying each .cpp file in the compilation process (g++) of PA3, we will create makefiles to automate the compilation process and the achievement of the checkpoints.

In general, a makefile consists of multiple named targets (“rules”) that can depend on each other. Every target has an associated action that is usually a UNIX command, such as g++. The structure of a named target looks as follows:

target_name : dependencies
	command_to_execute

IMPORTANT!
The command_to_execute line MUST be indented using a Tab space. This can be achieved by using the Tab key on your keyboard.

The UNIX make command interprets makefiles by executing the targets and their associated actions in the order of the targets' dependencies. Per default, the make command executes a makefile that is literally called Makefile (case-sensitive!). Incase of naming your Makefile differently (e.g. Makefile_Checkpoint1), then you must use the –f option for the make command. Also check out the manual page of the make command (man make).
To demonstrate makefiles, we provide a makefile for Checkpoint I. Let's name the makefile Makefile_Checkpoint1.

```
# in EC327, we use the g++ compiler
# therefore, we define the GCC variable
GCC = g++

# a target to compile the Checkpoint1 which depends on all object-files
# and which links all object-files into an executable
Checkpoint1: TestCheckpoint1.o Point2D.o Vector2D.o
	$(GCC) TestCheckpoint1.o Point2D.o Vector2D.o -o Checkpoint1

# a target to compile the TestCheckpoint1.cpp into an object-file
TestCheckpoint1.o: TestCheckpoint1.cpp
	$(GCC) -c TestCheckpoint1.cpp

# a target to compile the Point2D.cpp into an object-file
Point2D.o: Point2D.cpp
	$(GCC) -c Point2D.cpp

# a target to compile the Vector2D.cpp into an object-file
Vector2D.o: Vector2D.cpp
	$(GCC) -c Vector2D.cpp

# a target to delete all object-files and executables
clean:
	rm TestCheckpoint1.o Point2D.o Vector2D.o Checkpoint1
```

As demonstrated, makefiles only support single-line comments that start with the '#' character. Furthermore, makefiles support the definition of variables, such as GCC, which get a specific value assigned. In our case, the GCC variable holds the value g++. You can access the value of the variables using the $() notation. For example, $(GCC) returns the value of the GCC variable, which is g++.

In the file Makefile_Checkpoint1 we specify five targets (denoted in bold text). The Checkpoint1 target depends on three targets, namely TestCheckpoint1.o, Point2D.o, and Vector2D.o. Each *.o target depends on the .cpp file, which should be compiled to an object file (using the -c option of the g++ compiler). We define those dependencies in order to avoid re-compilation of .cpp files that have not been changed during the last compilation process. We also define one target clean, which deletes all object-files and the executable Checkpoint1 (using the rm UNIX command).

Per default, the make command executes the first target, which is in our case Checkpoint1. You can, however, instruct what target to execute by passing the name of the target to the make command. For example, the following command will execute the clean target of the Makefile_Checkpoint1 makefile.

make –f Makefile_Checkpoint1 clean

The EC327 staff will provide a final makefile for PA3. However, it is your task to formulate the makefiles for each checkpoint.

For further information on makefiles, please consult https://www.gnu.org/software/make/

Submission

Clone the repository created by GitHub Classroom. Write all of your code in the cloned directory. Please use the file names **Point2D.h, Point2D.cpp, GameObject.h, GameObject.cpp, Vector2D.h, Vector2D.cpp, Building.h, Building.cpp, DoctorsOffice.h, DoctorsOffice.cpp, ClassRoom.h, ClassRoom.cpp, Student.h, Student.cpp, Model.h, Model.cpp, View.h, View.cpp, GameCommand.h, GameCommand.cpp, and main.cpp**. After each checkpoint or major change, make commits to your local repository. When you are ready to submit, push your changes to the repository created by GitHub Classrooms. Do **NOT** commit or submit your executable files (a.out or others) or any other files in the folder. Make sure to **comment** your code.