The Assignment
Your job is to program a new Mars Rover, the Vanguard. The Vanguard is being deployed on a remote location of the planet Mars on a rescue mission to find missing pieces of the lost British rover Beagle 2. Satellite reconnaissance has revealed that several key components of the Beagle 2 are scattered about the Mars surface. Among these parts are the Apex Transmogrifier Unit (ATU), the Wavelet Transponder Dish (WTD), and a rare Earth mineral, Praesodymium (Crystal). Vanguard is being deployed in a random area of Mars and has been ordered to retrieve which ever missing piece of the Beagle 2 is found in that area.

Your task is to program a simple algorithm that will scan the Vanguard's current area to determine the location of one of the missing components of Beagle 2. After locating the missing component you must plan a simple path for the Vanguard to take in order to reach the item.

It is strongly recommended that you complete this homework problem, as it will make you very familiar with the Project 1 environment, and will save you a lot of time that you will need to finish Project 1.

Development Environment
Since this project utilizes modern video game technology, DirectX 9.0, you must develop in a Windows environment using Microsoft Visual Studio .NET 2002 or newer. The DirectX 9.0 SDK must also be installed in order to compile. Nearly all (95%) of the CAEN labs have this software installed ready for you to use. Your code should be ANSI-C compliant as it will be graded in a Unix environment under gcc.

You can download the environment with some basic maps from the course website. Details on what files you need to modify can be found below. The following are the provided maps you should be able to handle:

- ATU.txt
- Beagle2.txt
- Crystals.txt
- WTD.txt

Your Tasks
You are to develop a simple algorithm to help the rover move from its current location to the tile that the missing component is on. You should first iterate over the Vanguard's current area looking for a missing component. After you locate the item you should simply create a path that moves (+/-) X spots in the East/West direction and (+/-) Y spots in the North/South direction. That is if the Vanguard's current position is (3,3) and you find a missing item on (9,6) you should move 6 spots in the East direction and 3 spots in the South direction.

Map Representation
The game world is represented through several C++ classes: IVECTOR2, C_Map, C_Tile, C_Item, C_Player, and C_AI. The IVECTOR2 class is a simple data structure that is used to represent a grid location. The C_Map class contains a 2-D array of C_Tiles. The C_Tile class contains basic tile information such as (x, y) position, tile passability, etc. The C_Item class contains basic item type and (x, y) position. The C_Player class contains data relevant to the Vanguard, such as its current location, current map, and AI data. The AI for the Vanguard is what you will be working on, and it has its own class, C_AI. Do not be overwhelmed by the number of classes listed here, as the only class you must work in is C_AI, and all the functionality you will need from the other classes is listed in this document. In fact, there are many more classes in the project not listed here. You should not touch or use these classes, as it could cause your code to not function properly on the auto-grader.

As stated, the Mars domain consists of a 2D array of tile objects. Each area will be no bigger than 50 x 50 and no smaller than 4*4. In order for the Vanguard to move from one tile to another your algorithm must create a list of adjacent tiles that the rover must visit in order to reach its goal. There will be two possible paths from the Vanguard's starting point to the missing Beagle 2 component; however, your algorithm must produce only one of them. You may produce either path that reaches the goal.

After you find the goal and create the list of tiles you should begin returning C_Tile pointers to each tile on the path. See the section on the C_AI::RunOneStep() function for more information on how to do this.

What to Modify - AI Class (C_AI)
Two files (ai.cpp and ai.h) represent the C_AI class. This class represents the “brain” of the Vanguard. This class is where your algorithm will reside. C_AI consists of a couple public functions that you must fill out. They are:

```cpp
C_AI(C_Player* player); // Constructor
-C_AI(); // Destructor
C_Tile* RunOneStep(); // Run one "step" of ai
void Reset(); // Reset AI Search State
```

In the constructor you should handle searching the map for any missing Beagle 2 component. You should also use this section to compute a path to any found items. If your search ever fails, (you cannot find any items on the map), you must exit(1).

In the destructor you should clean up any dynamically allocated memory.

The RunOneStep() function should return a pointer to the next tile on the path from the goal to the Vanguard (note the order here). This function's return type is C_Tile* because after your algorithm finds a path, it will return it tile by tile until the entire solution path is returned. After the entire solution path is returned, it should return NULL. The first C_Tile pointer returned should be a pointer to the goal tile itself, then a pointer to the tile before that (which led to the goal tile), then tile before that, etc, all the way back to the start tile (Thus, the last tile you return will be the tile on which the Vanguard currently is)

After your algorithm returns a pointer to the start tile, the Vanguard will automatically start moving along this path (It will continue to call RunOneStep, and will move one square each time you return NULL. The path tiles you returned will be highlighted in a greenish color, to help you see what is going on. When the Vanguard reaches its destination, it will stop and call Reset(), and then start calling RunOneStep() again if you do not exit appropriately. **This may cause your code to not be properly graded as your executable will never exit and the autograder will assume your code is in an infinite loop.**

Reset() should test to see if the Vanguard has reached the goal location. If the Vanguard has reached the goal location you should exit(0). Reset will be called whenever the Vanguard physically reaches a goal, whenever user input is detected, and whenever a search fails.

### Important Class Descriptions

The following are a brief description of each of the important classes that are necessary to complete this assignment. If you would like more information on these or other classes (such as a complete list of member functions), check the appropriate header files.

#### IVECTOR2

Position is represented through the use of the IVECTOR2 class. This is a simple class with X and Y int members. Some examples on how to use the class:

```cpp
IVECTOR2 variableName; // declares an IVECTOR2
IVECTOR2 variableName(startingX, startingY); // declares an IVECTOR2 with
// startingX, startingY as initial
// values

variableName.x; // returns the x value (int)
variableName.y; // returns the y value (int)

IVECTOR2(someX, someY); // returns an IVECTOR2 with
// someX, someY values
```

#### Maps

Each game world is made up of multiple maps and each map is defined by the C_Map class. The C_Map class encapsulates a multi-dimensional array of C_Tiles. The C_Map class also contains methods to retrieve a C_Tile pointer at a coordinate location, as well as the ability to query whether or not a C_Item exists at a IVECTOR2 location. In order to represent the entire game world, an array of C_Map pointers called gp_GameWorldMaps is defined globally. You may access this array to determine information about any map in the entire game world (although for this project there will only be one map per world). Therefore, if you wish to know the size of map 1 (the array is indexed by zero) you would call the following function:

```cpp
gp_GameWorldMaps[0]->GetSize().
```

Some of the important functions and variables that you may need are:
Tiles
Each map that the Vanguard can travel on is a set of M x N tiles, where M and N may not necessarily be equal; however, they will both always be greater than or equal to four. Tiles are aligned in a Euclidean grid system. In the game world NORTH is in the upper right direction (see above image for a compass). The origin of each map is always the most NORTH WEST tile and each tile is always in the first quadrant. That is a tile’s (x,y) coordinates begin at (0,0) and increase in x values to the right of the origin and increase in y values to the left of the origin. Figure 2 shows a picture of one map and its coordinate system.

Each tile is defined by the C_Tile class. Some of the important functions you will need are:

- IVECTOR2 GetLocation(); // Returns an (x,y) location of the tile
- ITEM_TYPE GetItemType(); // Returns what type the item is
- C_Tile* GetCurrentTile(); // Returns a pointer to the tile

Items
In order to complete its mission, the Vanguard must reach one of the missing items {ATU, WTD, Crystals, Beagle 2}. The Vanguard can move in four directions to any adjacent, tile. The items are indicated visually on the map, but the Vanguard will not know where they are located before the mission. However, the Vanguard can query a tile to see what, if anything, is on it.

A list of all the different types of items that the rover could encounter on Mars is: {ATU, WTD, Crystal, and the Beagle 2}. Each one of these items is identified by the following respective enum types defined in item.h:

```c
enum ITEM_TYPE {ITEM_NONE, ITEM_ATU, ITEM_WTD, ITEM_CRYSTAL, ITEM_BEAGLE2};
```

Each item is defined by the C_Item class. Some of the important C_Item member functions that you may need are:

- ITEM_TYPE GetItemType() const; // Returns what type the item is
- IVECTOR2 GetLocation() const; // Returns a (x,y) location of the item

C_Player
The AI routine in which you must implement your algorithm will be given a pointer to the current player object, C_Player *p_Player. Through this pointer you will be able to query the current IVECTOR2 location of the player which is required to get your search started. The Vanguard is defined by the C_Player class. Some of the C_Player member important functions that you may need are:

- bool s_SearchFailure; // controls running of C_AI::RunOneStep();
- C_Tile* GetCurrentTile(); // Returns a pointer to the tile
- IVECTOR2 GetGoalLoc(); // Returns the current goal location as set by the user.

User Input
In order to move around Mars the Vanguard has been equipped with remote input. To move the Vanguard you may use the numpad keys. The following keys are mapped to diagonal movement:

- Numpad 1: SOUTH
- Numpad 3: EAST
- Numpad 7: WEST
- Numpad 9: NORTH

In addition to manual input for direct movement, the Vanguard has two methods of controlling its pathfinding movement:
F2: Sets AI to RUN_ONE_STEP, then switches back to DONT_RUN
F3: Sets AI to RUN_FULL_SPEED
F4: Toggles AI debugging information on/off.

The “F2” key is mapped to your pathfinding algorithm. By pressing the “F2” key the rover will call C_AI::RunOneStep() thereby calling your algorithm one time. This will certainly be helpful in debugging.

Also the key “F3” is set to call your RunOneStep() repeatedly. This is the equivalent of your code running in a while loop. RUN_FULL_SPEED stops on two conditions. First, it will stop on user input, such as a mouse click or numpad input. Second, it will stop when you set C_Player::s_SearchFailure to true.

Finally, the “F4” key will toggle visual debugging information for your pathfinding algorithm. If the pathfinding information is turned on, the solution path will be colored once you begin moving with the same search information.

The Vanguard is also set up to define a user-defined goal by left-clicking the mouse on a tile. While the auto-grader will not check to make sure your algorithm correctly handles pathfinding to user-defined target, you may find it helpful to define this behavior first for debugging. During searches the user-defined goal, if defined, is available by the C_Player member functions:

```cpp
int C_Player::GetGoalMap()
IVECT2 C_Player::GetGoalLoc()
```

So, instead of checking a tile to see that it is a desired item, you would check the tile’s location and map with those returned by GetGoalMap() and GetGoalLoc().

Grade Composition
10 of 50 points for homework 1, as determined by the autograder.

What to Turn in
Do all of your work (with all the needed files, as well as test cases) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code be sure that:

- The total size of your program and test cases does not exceed 10MB.
- You don’t have any unneeded files or other junk in your submit directory and your submit directory has no subdirectories.
- The files you must submit are: ai.cpp, ai.h, and any other files you use.

The Auto-grader
Your code will be graded using the auto-grader. Instructions on how to submit will be posted when the autograder is available.

Once the auto-grader is available, you can submit your code to it three times a day and receive correctness and timing feedback by email at a later time. **You should NOT use the auto-grader as a debugger!** Make sure you have tested your code thoroughly before even contemplating submission to the auto-grader. The auto-grader only has enough functionality to ensure that you are getting the basics right (compiles, runs, gets some trivial cases correct, etc.) but you still need to test your code carefully and thoroughly on your own.

If your code takes too long to run, the autograder will simply terminate it. How long is too long? It will be a time loosely based on the average run time of the top k best performing projects, including that of the staff. Every time you submit your code to the autograder, it will log your run time. We will periodically post the top 10 best performing numbers on the course web site or forum. Despite its name, the auto-grader does not assign grades. Your grade will be assigned after all final submissions are received and graded.

Every time you submit your code to the auto-grader, you **overwrite** your old copy. We **do not** keep backup copy of your code. It is **your** responsibility to back up your code. If you have working code, make sure you make a backup copy of it before making any more “improvements” to it that may break it. (The adventurous ones among you may try to learn how to use RCS or CVS for this purpose, but not if you don't have time.) Your last submission to the autograder is considered your final submission and is the one we will use to assign you a grade.

Have fun coding!