**Design decisions**

These are the final design decisions, updated after the results of the test plan showed incorrect output or non-optimal results which are elaborated in a separate section.

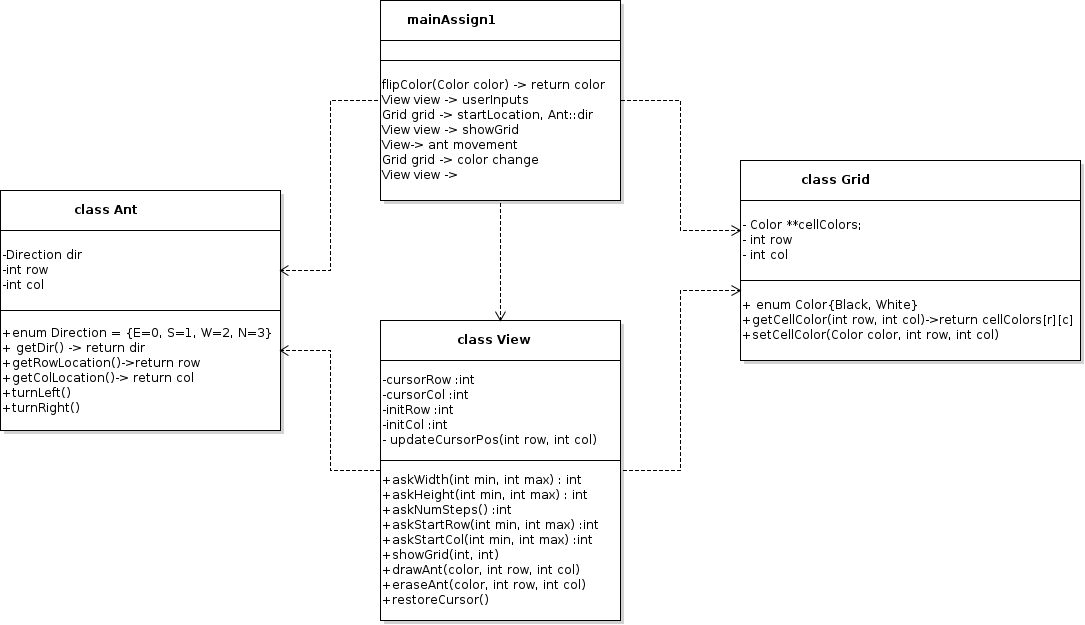
**Design overview**

To meet the program requirements, I decided to create separate classes for Ant and Grid and designed them such that they are encapsulated from each other, while another class, the class View, was used for interactions between Ant and Grid. As shown in the diagram below, I also used mainAssign1, to interact with all three classes and to drive the program. Input validation of user input is done via functions in the “utils” namespace that is declared in the header file, “utils.hpp”. I created this utilities function, as a generic functions that could be used for many programs. Since these “utils” functions are generic and not intrinsic to this program, I have not shown the utils function in my model design.

**utils Function:** was used to create generic functions that could be used elsewhere.

1. int readPositiveInteger (string prompt, int min, int max) → allowed me make it a very generic function which can be used to get input from user and validate it based on criteria that can be passed in. In this program assignment, I used it to validate input from user for grid dimensions, number of steps, and location of the ant at the start of the simulation.
2. string replicates(char val, int numTimes) → for use when you want a particular character repeated. In the present program, I used it to great the vertical grid borders for the simulation

**Model design**



The general purpose of each class is as follows:

**class Ant:** represents the Ant, the starting location and direction it is facing at the start of the simulation and the direction to move, whether left or right.

**class Grid**: is a dynamically allocated 2D array with two colors

**class View**: depends on both the Ant and Grid class, to display the grid, controls the drawing and erasing the ant (as a means of showing movement forward) and also controls the location of the cursor with each movement of the Ant

**mainAssign1**: interacts with all the classes, uses the utils functions and with the creation of the class objects controls the entire program. The detailed description follows.

**Details of each class:**

**class Grid**

* private member variables include:
  + Since at compile time, there would be no information of size of grid, I used a dynamically defined 2D array. Additionally, I used an enumerated data type, called Color
  + integer type variable, row
  + integer type variable, column
* public member functions include:
  + Color variables, Black and White
  + accessors – getCellColor with parameters of integers for row and column, returning a cell colors for the rows and columns of the grid
  + mutators – setCellColor with parameters of enum Color, integers for row and column
* Constructor was used to populate the 2D array with nested loop to create the grid
* Destructor was used to free the dynamically allocated memory, to prevent memory leak. Though, in this particular case, since the program would exit at the end of the simulation, there would not be a need to free the dynamic allocation, I thought it would be a good practice to acquire.

**class Ant**

* private member variables include:
  + enumerated data type called Direction with variable, dir
  + integer type variable, row
  + integer type variable, colum
* public member functions include:
  + Direction variable dir with values, E, S, W & N. I chose to assign default values to these, to be able to use the modulo operation to control movement of the Ant.
  + accessors: getRow, getColumn and getDir to get grid location for the Ant and the direction the Ant would be facing when the simulation began.
  + turnLeft and turnRight functions to control Ant movement.
  + mutators: to set row and column
* Constructor with parameters for row, column and direction

**class View**

* private member variables & function
  + cursorRow, cursorCol, initRow and initCol defined as integers to initialize row and column and also to control the cursor. After much, trial and error, I chose to use cursor control, rather than redrawing the grid for each forward movement of the Ant. This way, I was able to speed up the simulation significantly.
  + I used updateCursorPos with parameters of row and column of type integer to ensure the cursor move according to movement of the ant. This function uses the UNIX cursor control command to move the cursor forward (i.e. right), backward (i.e.left), up and down for every change in row value and column value, which I defined as rDelta and cDelta. With the use of if-else statements, I controlled the cursor for each movement forward of the Ant. I kept this function private, since it was unnecessary to give access to this function to others.

I also used the updateCursorPos function in the drawAnt function, each time, the Ant (represented by ‘ \* ’) was drawn on the grid.

* public member functions include:
  + user-input to get dimensions of the grid, number of steps for the simulation and starting location of the ant
  + showGrid function to display the grid
  + drawAnt function to draw the ant
  + eraseAnt function to erase the ant, as a proxy for movement forward
  + restoreCursor function to ensure the cursor ends up at the correct location when the simulation ends

**mainAssign1**

* object of class View:
  + to get user-input for grid dimensions, number of steps and starting location of Ant
* object of class Grid
  + for number of rows and columns of the grid
* object of class Ant
  + for starting location of Ant and also to set the direction the ant faces at the start of the simulation
* a flipColor function of enumerated type Color, with a conditional operator to change color from Black to White
* accessor function of class View for initial positions and to drawAnt
* accessor function of class Grid for color of grid
* mutator functions of class Ant and class Grid to show ant movement forward.
* Use of a while-loop to count down the number of steps of the simulation. Also, within this while-loop, use of mutator function to setCellColor and change color of grid, with each forward movement of the ant. Additionally, with the use of if-else statement, calling the turnLeft and turnRight functions of the class Ant, to follow the rules of Langton’s Ant.
* With the use of switch statement and accessor function getDir of class Ant, I used the modulo operator to wrap around when the ant encounters the borders of the grid.
* Use of usleep function from unistd.h library to slow down the movement of Ant to display each step of the ant.
* Use of restorCursor function to ensure the cursor is at the end of grid on a new line

**Changes to initial design**:

My initial design had only one Ant class to cover all the requirements, but as I started writing the test plan for input validation, I found that I needed to encapsulate better, so began to create separate classes, which then evolved into 3 classes, separate from the main and utils functions.

The second change in my initial design, was the method to address the case where the ant hits the boundary of the grid. After several discussions about this functionality on the discussion thread on Canvas, I decided to go with the **wrap-around option**, so that the grid ‘behaves’ like it has no edges, rather than expanding the size of the array. Since the user was asked the size of the array, I believed that **changing the array size halfway through the program to meet a condition, was incorrect,** **so I decided to use a wrap-around**, so that the ant would move either up/down or side-to-side, if need be **to execute the number of steps specified by the user.**

The third change was to the grid printing to the console with each forward movement of the ant. I started by re-rendering the grid each time to show ant movement. In reading the discussion forum on Canvas, based on a hint given by one of the TAs to an unrelated question, I discovered ANSI escape codes and on reading further about it, found that I could optimize my program to run faster by not re-rendering the grid, instead controlling where the cursor was positioned with respect to the rules of the movement for the ant. After several tries, I was able to create two functions, updateCursorPos & restoreCursor, to ensure that the cursor on the console could be useful for both the wrap-around and obviate the need for re-rendering the grid each time.

**The fourth change resulted from observed outcomes in my test plan**. While doing the input validation, I saw that my observed outcomes were not correct, if the user input characters instead of integers or used a floating point number instead of an integer. The other change was to ensure that the user had the option of making the starting location of the ant at (0,0). So I created two functions, the readPositiveInteger & startLocation which controlled for invalid inputs and allowed the user to randomly select the starting location of the ant.

**The fifth change also resulted from observed outcomes in my test plan**. During input validation, if the starting location of ant was outside the boundaries of the grid, it caused a segmentation fault with a core dump error. In trying to amend the startLocation function, I realized that I had duplicated functions with readPositiveInteger and startLocation, but could not figure out how to combine the two. Once again, hints from the discussion forum on Canvas with the TAs helped me to combine them to one function, which I updated by passing the parameters for minimum and maximum values in the readPositiveInteger function. I had to research the documentation for maximum limits, in which I found the limits.h library function, with which I could use the INT\_MAX library function.

**The sixth change also resulted from observed outcomes in my test plan**. When the starting location ant equaled the grid size, I got a segmentation fault with a core dump error. I had forgotten that since I was using a generic utils functions, the parameters that I passed in had to account for zero-indexing, since I was permitting a start location of (0,0), so I made the change whereby the start location had to one short of grid size.

**Test Plan & Results**

With the aim of meeting all the requirements laid out in the grading criteria segment of the assignment document, I used an incremental development strategy with testing each segment before moving on to the next. At first, I thought of only creating one class, called Ant in which all the member variables and functions would be defined, with a main function controlling the program.

So, my test plan follows:



**Changes made to program to solve problems highlighted by test plan:**

1. I created two functions, the readPositiveInteger & startLocation which controlled for invalid inputs and allowed the user to randomly select the starting location of the ant.
2. I amended the readPositiveInteger function to take in min and max parameters and deleted the startLocation function as it was redundant.