Ian Nelson

CMIS315

Final Project

Jeff H. Sanford

**Final Project Analysis**

For the final project, the amount of objects in play was larger than all assignments previously given in the semester. In order to create a modular and connected structure, I created the following files:

1. **Utility.cpp**: Acts as a base class providing a common method to convert integers to string. This was done to illustrate inheritance properties, as well as reuse of this function application-wide.
2. **Menu.cpp**: Acts as the master logic for interaction with the program. This object creates the base *Building* class, and processes all user input, validation, and interaction with the subordinate processes and logic.
3. **Building.cpp**: The main class encapsulating the building and floors. This class contains a vector managing the *Floor* objects.
4. **Floor.cpp**: The main class encapsulating the floors and rooms. This class contains a vector managing the *Room* objects.
5. **Room.cpp**: The main class encapsulating the rooms and access points. This class contains a vector managing the *AccessPoint* objects.
6. **AccessPoint.cpp**: The class which encapsulates functionality revolving around a network access point. This is the most subordinate class in the program hierarchy.
7. **Iannelson\_finalproject.cpp**: The file containing the main function. Creates and starts the *Menu* object and subsequent UI actions.
8. **Makefile**: Used to build the application with G++/GNU Make.

Each C++ class also has header (.h) file containing its definition, other than the main execution function.

The strategy for developing this program was to adhere to the hierarchal requirement for nested or managed objects. In this respect, a tree of functionality was represented as documented in the class breakdown above. By using vectors and pointers to represent searches of subordinate objects (such as *Building* -> *Floor*), I was able to create this relationship easily using pointer method accesses (->). This can be demonstrated in the Menu methods where nested access is required to access an object further down the tree. Additionally, each representative class has methods to add, remove, and find its subordinate object within the vector. This allowed for full management of all subordinate objects and easy access to this logic via the *Menu* methods.

The Menu class contains all user interface methods for accessing the system. This set of methods allows the user to :

1. See the representation of the tree structure for the building. This representation is given in a nested view with indentation to eliminate confusion, as seen below:

Building Information

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

Building -> 2 floors

Floor 1 Information

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

Floor -> 1 rooms

Room 1 Information

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

Room -> 1 access points

[Access Point 1] : On : On as of October

Floor 2 Information

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

Floor -> 2 rooms

Room 1 Information

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

Room -> 1 access points

[Access Point 1] : Off : On as of N/A

Room 2 Information

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

Room -> 1 access points

[Access Point 1] : Off : On as of N/A

This indentation at 2 spaces per subordinate allows the user to easily scan the building information and see what floors, rooms, and access points are available at a glance.

1. Add and remove floors from the building. Each check of a floor also validates the number of the floor from the vector and throws an exception if it does not exist, which is trapped and logged to the user.
2. Add and remove rooms from a floor. Each check of a room also validates the number of the floor from the vector and throws an exception if it does not exist, which is trapped and logged to the user. Additionally, the floor number is requested and validated to ensure the room is attached to the floor.
3. Add and remove access points from a room. Each check of an access point also validates the number of the room and floor from the vector and throws an exception if it does not exist, which is trapped and logged to the user. Additionally, the floor and room number is requested and validated to ensure the access point is attached to the room, which is in turn attached to the floor.
4. Toggle power status of an access point. This toggle simply sets a Boolean flag within an access point object, and represents it via ternary operator when it is printed to the console as “on” or “off”.
5. Date that the access point was set. This setting allows for a user to select a validated month from a menu, and that textual representation is stored in the access point object.

Of note regarding the date and power settings is the power status is defaulted to “off”, and the month is defaulted to “N/A” to signal a user that it is not active until adjusted.

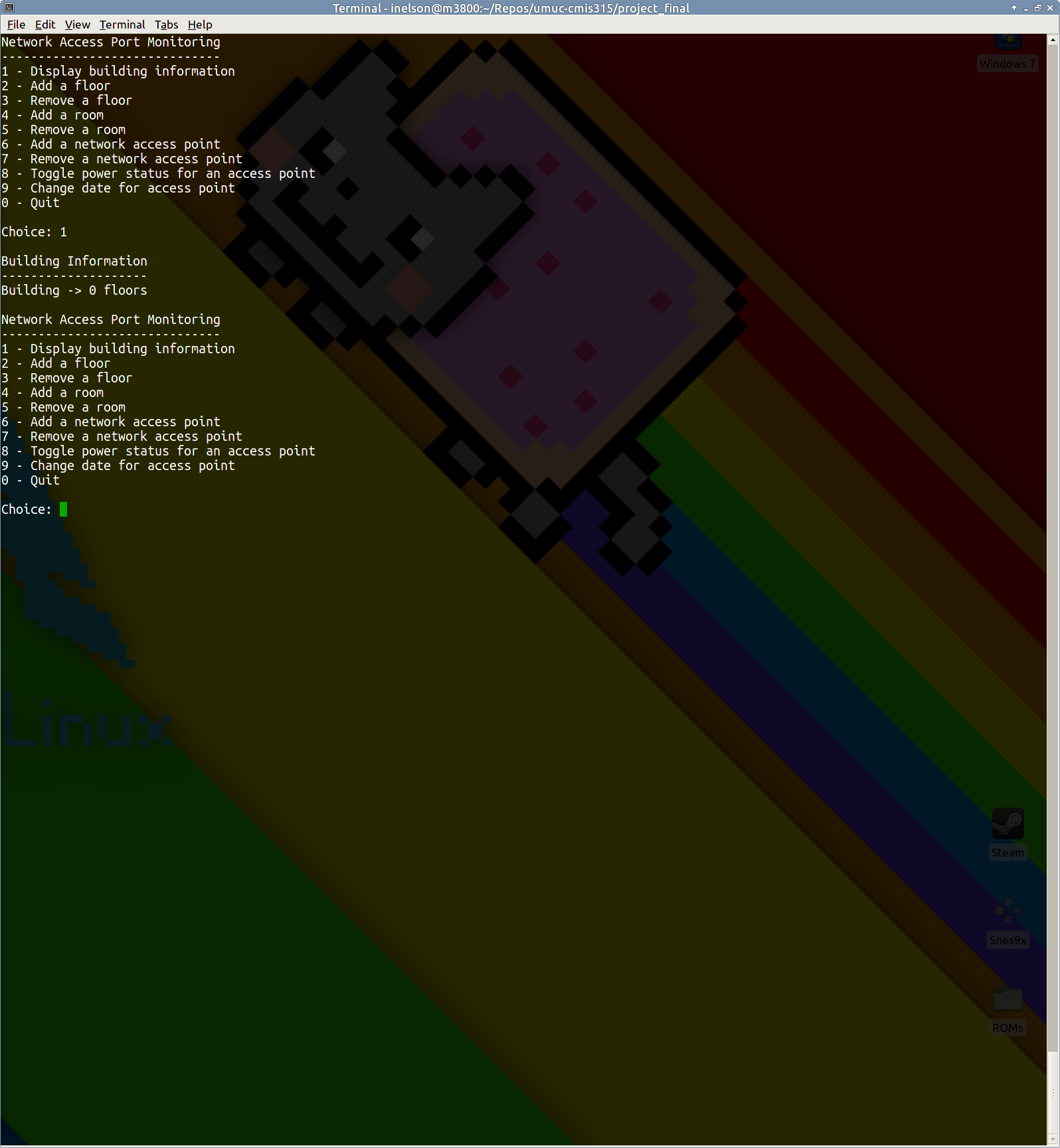
Finally, the program was build in GCC version 4.9.0 on ArchLinux 3.15.5-1-ARCH.



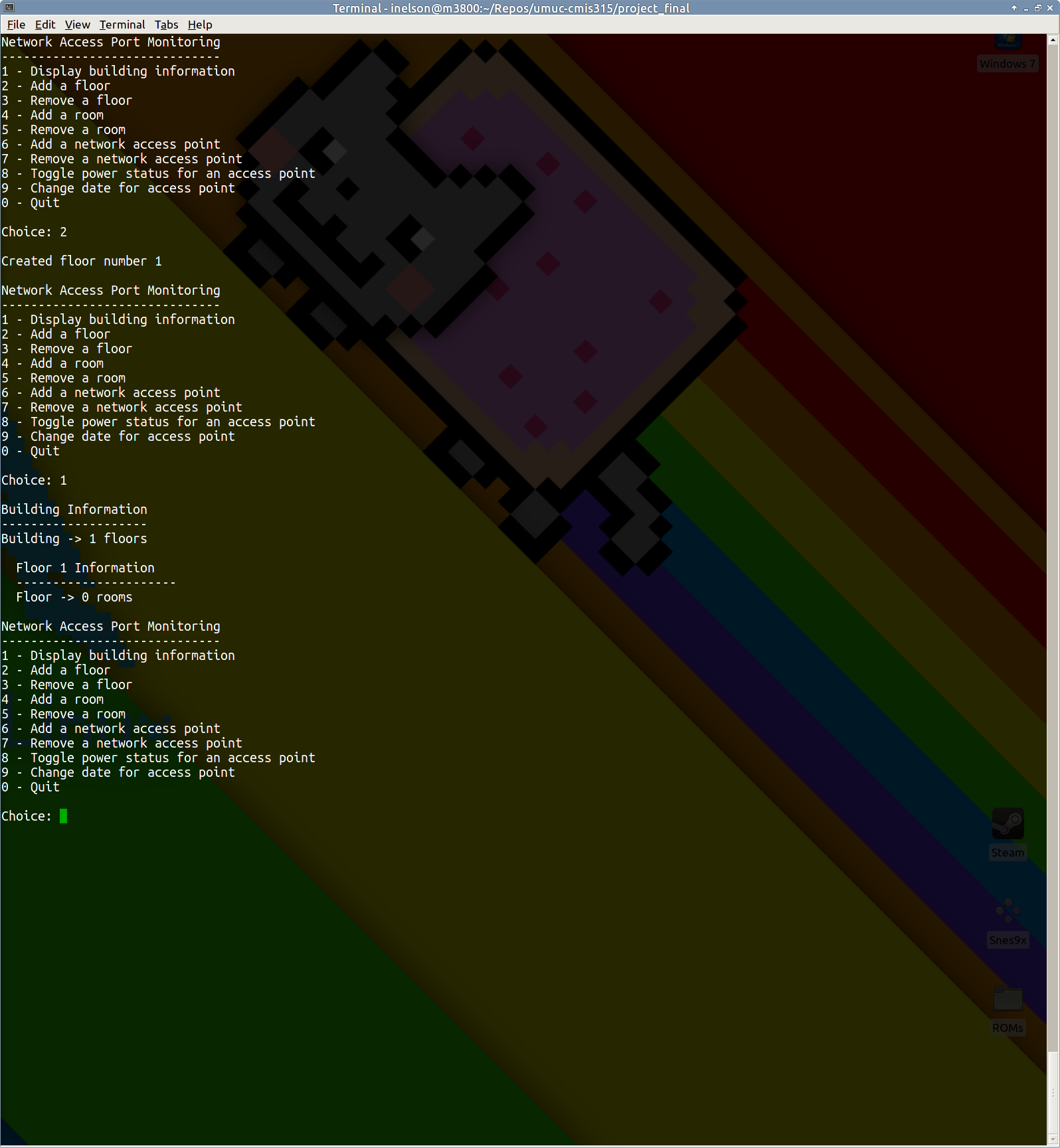
***Figure 1. Build Output from GCC/Make and the resulting binary***

******

***Figure 2. Exiting the program via the menu***

******

***Figure 3. An empty building at launch***

******

***Figure 4. Creating a floor***

******

***Figure 5. Creating a subordinate room***

******

***Figure 6. Creating a subordinate access point (with defaults)***

******

***Figure 7. Toggling the power status of an access point***

******

***Figure 8. Toggling the date of an access point***

******

***Figure 9. Removing an access point***

******

***Figure 10. Removing a room***

******

***Figure 11. Removing a floor***

******

***Figure 12. Input validation***

******

***Figure 13. Room access validation (via exceptions)***