CS 162 Final Project Design Document
(*All designs reflect preliminary design)*

Item

private:
string name;
int id;

pulbic
Item(string name, int id);
Void setName(string name);
Void setId(int id);
string getName();
int getId();

Backpack

private:
vector <Item *> bag;
int const capacity = 3;
int mergeNum;

public:
Backpack();
bool hasSpace();
void add(Item *);
Item * removeItem();
string toString();
void merge();

Space

protected:
Space *up;
Space *down;
Space *left;
Space *right;
Vector<Item *> items
string id;
bool win;

public:

Space(string id);
void virtual special(backpack *);
string virtual toString();
string getId();
Item* pickup()
Void drop(Item *);
bool getWin();
Space* getRight();
Space* setRight();
Space *getLeft();
Space *setLeft();
Space *getDown();
Space *setDown();
Space *getUp();
Space *setUp();

(Park, Street, and Home are subclasses of Space class)

Park : public Space

private:
bool hasKey;

public:
Park();
special(backpack *);
string toString();

Street : public Space

public:
Street();
special(backpack *);
string toString();

Home: public Space

public:
Home();
special(backpack *)
string toString

Jose Bigio
CS162 Final Project

Person

private:
string name;
Space * tracker;
int moveCount;
bool winner;
Backpack *bag;

public:
Person(string name);
void getMoveCount();
Space* getTracker();
Void setTracker(Space *);
Void setWin(bool win);
bool void getWin();
backpack* getBag();

Game

private:
Person * player;
string help;

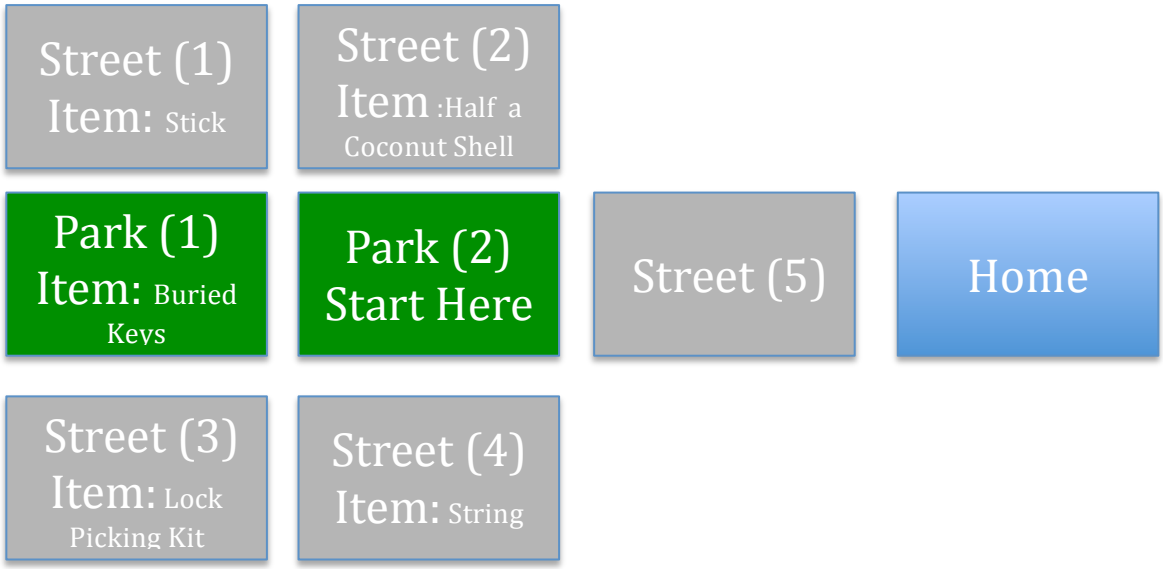
public:
Game();
void addItem();
void dropItem();
void move(int x);
void action();
void playGame();

Game Description

This design reflects the design that will be utilized to create a game. The objective of the game is to get back home before your parents realize you have snuck out. Unfortunately, you can't seem to find the keys to your house, and you aren't sure if you took the keys with you.

Jose Bigio
CS162 Final Project

The player only has a set number of moves (tentatively 30) before the player's parents find out that the player has snuck out. Below is a map of the game. Along with this map is an indication of where the player will start the game, and the items that can be found in each location.



Each square represents a space in the game. The mechanism of the game will allow the player to walk to adjacent squares. In cases where there are no adjacent squares the player will be notified that they can't go in that direction.

As the map indicates some spaces have items that the player can pick up. The player has a backpack that allows them to store up to 3 total items. If the backpack is full the player has the option to drop an item. When an item is dropped it is dropped in the space they are in. Items never disappear, so if a player later wants to pick the item up again they can return to the space where they have dropped the item.

Every time a player enters a new space the number of moves the player made will increase by one. The player only has a set number of moves before they lose the game. The only moves that count are moving from space to space. Picking up or dropping items do not count as a move. When a player enters a space they will be told whether they are in the park, the street, or their home.

Although the player's house is locked the player can be in the home space, they just won't be able to enter their home. If the player wants more details they will have the option to have more details about the space they are in. At any time the player has the option to pick up an item, drop an item or use an item. These actions are subject to limitations. To elaborate, if there are no items to pick up nothing will happen if a player is attempting to pick up an item. Similarly if a player's bag is full they must first empty their bag before adding an item. Additionally, if a player has no items in their bag they can't use the item.

The way in which an item is used will depend upon the space the player is in. The only way to win the game is for the player to get back home fast enough before their parents find out they have left. Although, there is a lock picking kit the player is not skilled enough to use it, and if they attempt to this will alert the player's parents, and the game will be lost. Thus to win the game the player must find his keys, which happen to be buried in the sandbox of one of the parks.

In order to dig up the keys the player first needs a shovel. Since there is no shovel item the player will have to create a makeshift shovel item by picking up the string, half a coconut shell, and the stick. Together these items will all jumble up in the players bag and make a shovel. The shovel can then be used to dig up the keys, which can then be used to open the house. Below there will be more detail regarding the implementation of the game.

Implementation Details

Every item will be in the Item class. There is no need for an inheritance relationship between the items, because each item only needs an id, and a name. The name will be used to describe to the player what the item is, and the id will be utilized to determine the actions of the item.

The Backpack class represents a vector of items. The backpack class will have a maximum capacity of 3 items. This class will allow items to be added using the add(Item*) function. This function will take in an item pointer, check if the pointer is null, and if the pointer is not null and the backpack is not full the item will be added. If the item was added to the bag the user will be notified, if the bag is full the user will be notified that the item couldn't be added because the bag is full. If the bag is full the user can use the removeItem() function to remove an item. This function will first check if the back has items. In the case that there are no items the user will be notified that the backpack is empty. If the backpack has only one item that item will be removed. If there is more than one item in the bag the function will use the toString method to allow the user to determine what item they want to remove.

Each time an item is added or removed from the bag the merge() function will be called. This function will keep a running sum of the ids of the Items in the bag. The ids of each item will be as follows stick 1, rope 2, half a coconut shell 3, key 100, lock picking kit 1000, and shovel 6. Notice that the sum of the stick, the rope, and the half coconut shell equals the id of the shovel. This is intentional. When the sum of the items in the bag equals 6 the items will become a shovel, and subsequently only take up one space in the bag since it is now only one item. After this point there will no longer be a string, coconut shell, or stick in the game.

The space class is the base class for a set of three derived class that will represent the map of the game. Each space class will have 4 pointers to other spaces. These pointers will not be utilized in every instance. In the space class these pointers have been called up, down, left, and right because the map for this game exists on a 2d plane where these directions make sense. Not every pointer to a space class will be utilized. In fact, as the map indicates there is only one space that has 4 pointers to other spaces.

The space class will have a special(backpack *) function. This function is what determines how items will be used. The function is a virtual function because items will interact depending on what space they are in. A space will have a string, which represents its id, and a vector that holds item pointers. The id serves to indicate to the user what space they are in when they move to a space. The vector of item pointers serves to hold items in that space. A vector is used, because this allows the user to drop an item in any space. Since vectors have dynamic size the implementation doesn't have to be concerned with how the player moves around the items they are adding, and then possibly dropping.

For a more detailed description of the space there is a virtual toString() function. This function will use the id of the class, as well as the vector of items in the class to indicate to the user what items there are in the area. To set the directions of the neighboring spaces, and to get the directions of the neighboring spaces there are corresponding get, and set methods for all the directions. The set methods will be used for creating the initial map of the game. The get methods will then be used as the player navigates the map. The Space constructor will set the take in the id of the space. This will set the space id, and then all the space pointers will be set to null. To help determine when the user has won the game the space class will have a variable called win This variable is only significant in the Home subclass, but the variable will be set to true in the special function of the House subclass if the player opens the door to the house.

The space class will also have a function to pickup items, and a function to take the dropped items from the user called drop(Item *). The pickup function will first check if there are any items to be picked up. If there are items to be picked up the function will print the list of items to be picked up, and allow the user to select which item they would like to pick up. If there is only one item to pick up the list containing one item is shown in order to allow the user the chance to not pick the item up. To accomplish this the function will always have an additional option to allow the user to not pick up any item. If the user does want to pick up an item the item is removed from the space, and a pointer for that item is returned by the function. If the user does not want to pick up an item or there are no items to be picked up a null pointer will be returned. The drop(Item *) function takes in an item pointer representing an item that the user has dropped. This item is added to the vector of items of the corresponding space where the user is located.

The three subclasses of the Space class are Park, Home, and Street. All these subclasses will have a constructor, a special(backpack *) function, and a toString function which describes the space in further detail. The Park subclass will have an additional member variable called hasKey. Since there will be two park spaces in the game, but the key will only be in one sandbox of a park space the boolean member variable will be used to determine what space has the key. The special function of each class will determine how items are used in that space. In order to determine what items can be used the special function will take in a pointer to a Backpack object.

If the backpack is empty the special function will indicate that there are no items to use. If the backpack only has one item then that item will be used. If the

backpack has more than one item, then the user is prompted to select what item they will like to use. The ids of each item are used to determine the effect of the item in a particular space. Since the items will interact differently depending on the space, a virtual method is being used to allow different implementations of the item depending on the space.

In the game there are really only a limited number of item interactions. In the Street space no item interactions exist. In the Park space subclass only the shovel provides a valid interaction, which allows the user to dig. The hasKey boolean variable is also utilized in these classes to determine if there is actually a key in the sandbox. If there is a key the user has the option to add the key to their backpack. If the user's backpack is full then the key is added to the vector of the corresponding space. For the Home class there are two item interactions. The key opens the door and wins the player the game. The lock picking kit makes a lot of noise, and loses the player the game.

To represent the player in the game their will be a Person class. The Person class has Space pointer which keeps track of where the player currently is on the map. The Person(string name) constructor will initialize this space pointer to Park(2). The constructor will also take in the players name and set it. There is also an int member variable called moveCount, which will be utilized to keep track of the number of moves that the player has made. The player has a bool variable to indicate whether they have won the game yet. As has been mentioned throughout the game one of the other member variables of the person is a backpack, which holds the items the player might pick up along the way. This class has corresponding get, and set methods for the member variables. There is a getBag() method which returns a pointer to the backpack. There is setTracker(Space *) function which takes in a Space *, and updates the tracker to that pointer. There is a setMoveCount(int moves), and a getMoveCount() function which set, and get the number of moves the player has made. There is also a setWin(), and getWin() function to keep track of whether the player has won the game.

The whole game will come together in the game class. The game constructor will set the map of the game up, and initialize the player variable prompting the user to enter the name they want to use for their player. The game class will have a move class which kind of implements the movement mechanism. The game class will take in an int to indicate what direction the player wants to move in. The function will then determine if the direction the player wants to move into is valid. If the direction is valid the player's tracker variable will be updated to the new space, and the id of the space will be printed to indicate to the user that they have moved. The player's move count will also increase by 1. If the move is not valid a message will print to the screen indicating that they can't move in that direction.

The game class will also have addItem(), and dropItem() functions. These functions allow a user to add an item to their backpack or drop an item. The add item function will check if the backpack of the user has space, and if there are items to pick up in the space the player is currently in (using the tracker variable of the player). If both of these conditions are satisfied the player can add an item using backpack->add(tracker->pickup()). To clarify once the conditions are met to add an

item you can call the add method, and as an argument pass the pickup() method from the corresponding space class.

To drop an item the function will first check if the backpack has any items to drop. If there are no items to drop the user will be notified. If the bag has items to drop the list of items in the bag will be shown and the user will have the choice of what item to remove from bag. Before that item is removed from the bag it will be added to the corresponding space using the drop(Item *) function. The action() function in the game class will take user input to determine what action the user wants to take. The action options will be to move, drop an item, pick up an item, observe the space, ask for help or use an item. The keyboard input for these items will be
h for help (this will list all the controls)
w to move up
a to move left,
s to move down
d to move right
p to pick up an item
x to drop an item
z to use an item
o to observe space

The action() function will be implemented with a series of ifs or a case statement. If w, a, s or d are inputted the move function will be called. If p is input the addItem() function will be called, x the dropItem() will be called, o the toString function of the corresponding space will be called, z the special() function of the corresponding space will be called, and h a string representing the controls will be printed.

The action() function will be used in the playGame() function. This function will be a loop that continues as long as Home1->getWin = false, and the move count is less than the limit. This function will also indicate to the user if they have won the game.

Design Modifications

Since the design was relatively well thought out there weren't many design changes that needed to be made. The majority of changes were made in creating more get and set methods for particular member variables of classes. For example, initially there wasn't a getBag() function in the Person class, but this get method was essential because the bag was a private member, but it serves a great functionality in the game. A design change of a similar nature was that the constructor for the Park class was change to take in a boolean variable to indicate whether a key is hidden in the sandbox or not. Before this variable would have been initialized with a set method, but it seemed easier to set this variable in the constructor.

The design itself also doesn't go into much detail in how a win will be determined. A win is defined, but the design doesn't describe the implementation details. In order to determine a win the Space subclass has a boolean variable called win. This variable is initialized to false for all spaces, and is only really significant in the Home class. In the special function of the special function if the key is used the win variable is set to true. This variable is used in a while loop to determine whether the game should continue.

Although not a very large design change, but kind of a large game play change was that a new way to win was added. Now the player can win the game by either finding the key and opening the door, or by trying to lock pick the door. As before the player can still lose the game by lock picking the door, but there is now an element of chance which provides the player the opportunity to possibly win with this fashion. The success rate of pick the lock is approximately 25%.

Testing

Test	**Expected Result**	**Outcome**
Move around the map, and move 30 spaces	Spaces should connect as map indicates. Player should be moving in the direction that they are indicating. If the player tries to move to a location that doesn't exist they should be prompted. Once the player has moved more than 30 times the game should end.	Initially the movement was working relatively well. The results were printing the correct location as the map indicates in the design document. However, in one area the connections didn't seem right. It turns out that the switch statement was missing a few break statements once this was corrected the movement reflected what the map in the design document indicated. Once the error was fixed after moving more than 30 spaces the game did end.
Test help function	When the user presses h the controls of the game appear, and they reflect the accurate controls of the game.	When compiling there was a few problems with how the help string was being declared. Once this was fixed everything worked fine.

Ensure that Observe function works (really toString function of Space class)	When the user presses o the more detailed information about the space they are should be printed to the screen. The text should be printing with the correct format, and it should reflect the starting configuration of the game.	In testing the observe function I realized that it was printing a statement relating to the items at the space independent of whether the space had items. I included a design modification to make this clearer. During the testing I also realized that I had to utilize sstream objects in order to concatenate ints and strings since this wasn't working with the string class. Once these problems were fixed the toString function worked correctly it corresponded to the space the player was in, and it confirmed that the items were placed in their correct locations as the design document indicates.
addItem()	If there is an item allow user to pick the item up, if there isn't an item let user know that an item can't be picked up.	As mentioned previously to allow for concatenation of ints, and strings I had to use sstream objects whereas before I was attempting to do this with strings. Also I had forgotten to accept user input in the function so it wasn't initially working. Once the problems were sorted out the function worked correctly. This means that items could be

		added if there were items, and the user would be notified if there weren't items. When an item was added it reflected the choice of the user
dropItem()	If the user has an empty backpack don't allow any items to be dropped. If the user has items allow the user to select what item to drop, and drop the correct item.	The dropItem() function had an initial problem in allowing a user to drop the item they indicated because their was a logical statement that was a strict equality, but should have been an inequality. As with the other tests once this was fixed the dropItem() function was working, meaning that no items could be dropped if a user had no items to drop. If a user had more than one item to drop the item the user selected would be dropped, and it could be found in the space where the user had dropped the item.
Use addItem() when bag is full	Message tells the user that the bag is full	Message tells the user that the bag is full
Add the string, coconut, and stick to test the merge function.	Expect the coconut, string, and stick to merge into a shovel. Once the merge has occurred there should only be a shovel in the bag.	The merge worked successfully, but there was still an additional item in the bag. This arose from a misunderstanding of the parameters of the erase function for a vector. This problem was fixed subsequently.
drop all the Items in one space	Expect all the items to be in the place after they have been dropped	All the items were in the place after they were dropped, and they could be picked up again.
drop an item when there	User is told there are no	User is told there are no

are no items to drop	items to drop	items to drop
Test special function in street space	The shovel can't dig. The items that make up the shovel don't do anything. There are no doors to use the keys on.	Initially the special function wasn't working correctly for any spaces it appeared. The issue was that an index was off by one. This was fixed for all functions, and the function in the street space worked as described in the expected results.
Test special function in park spaces	Only the shovel should do anything. In park 1 a key should be found, and it should be added to the items of the space. In park2 the user should be able to dig with a shovel, but not find anything.	The shovel allows the user to dig in the sandboxes. In park1 a key is found, and can be picked up from the items in the park subsequently. In park2 the user can dig, but finds nothing.
Test special function in Home space	Lockpicking should not work consistently. Having the key and using it should win the game.	Lockpicking was tested 3 times and worked only once. Somewhat as expected. The key was tested twice, and it won the game both times.