Brian Palmer
Final Project
palmebri@oregonstate.edu

Final Project: “Escape the Beverage Factory”

Introduction:

“Escape the Beverage Factory” is a text-based game that lets users move around and interact with objects with the 6-spaced building. The objective of the game is to find the truck and the Loading Bay and drive it away. The catch is that the user must first find the ‘Key’ to the truck and open the garage door. There is a time requirement; the user must complete the objective before the step counter reaches 0 (user gets a total of 10 steps), and the step counter decrements by one each time the user moves.

Classes:

CREATURES:

Parent Class Creature

```
class Creature
{
protected:
    string type; - The type of creature
    Space * curr_space; - creatures current space

public:
    string get_type(); - type accesor
    Space * get_curr_space(); - curr_space accessor
    void set_curr_space(Space *); - curr_space mutator

    Creature(string n, Space *); - constructor
    ~Creature(); - destructor

    virtual void turn() = 0; - creature subclasses need defined steps for their turns
};
```

Creature Subclass Player

```
class Player : public Creature
{
private:
```

```
    int death_counter; - counts how many steps player has left
    bool won; - whether player won or lost
    bool moved; - tracks if player changed spaces
    bool game_over; - tracks if game is over
    friend class World;
    Stack * backpack; - players backpack
public:
    Player(string n = "Player", Space * c = NULL); - constructor
    ~Player(); - destructor

    bool get_won(); - won accessor
    void move(); - provides menu of available spaces to move to. If user moves, the
curr_space member inherited from creature will be updated
    virtual void turn(); - Steps involved in players turn are defined here
    void open_pack(); - accesses backpack member and calls backpacks open member
};
```

~~**Creature Subclass Boss**~~

```
~~class Creature~~
~~{~~
~~protected:~~
~~    string type; - The type of creature~~
~~    Space * curr_space; - creatures current space~~

~~public:~~
~~    string get_type(); - type accesor~~
~~    Space * get_curr_space(); - curr_space accessor~~
~~    void set_curr_space(Space *); - curr_space mutator~~

~~    Creature(string n, Space *); - constructor~~
~~    ~Creature(); - destructor~~

~~    virtual void turn() = 0; - creature subclasses need defined steps for their turns~~
~~};~~
```

The ‘boss’ was going to be a computer player that moved around the map whenever the player moved. If the boss wound up in the same space as the player, then some interaction would happen. I thought about having them fight or having the player lose unless they hid somewhere in the room. I did not end up coding this portion of the game due to a lack of time.

ITEMS:

Parent Class Item:

class Item
{
private:
    string type - *the name of the item*
    float weight - *the weight of the item*
public:
    string get_type() - *returns the type member*
    float get_weight() - *returns the weight member*

    Item(string n, float w, **bool b**) - *constructor initializing type and weight.* ***Bool added to initialize new Special member. This member is checked before an item can be removed. Prevents user from throwing out Key.***
    ~Item() - *destructor*
};

Subclass Class Water : Public Item
class Water : public Item
{
public:
    Water(string n = "Water", float = 1.50, **bool b**); - *constructor initializes type and weight.* ***Added bool, see parent class for details.***
    ~Water(); - *destructor*
};

Subclass Class Soda : Public Item
class Soda : public Item
{
public:
    Soda(string t = "Soda", float w = 1, **bool b**); - *constructor initializes type and weight*
    ~Soda(); - *destructor*
};

Subclass Class Key : Public Item
class Key : public Item
{
public:
    Key(string n = "Key", float f = 0.00, **bool b = true**); - constructor initializes
    ~Key();
};

SPACES:

Parent Class Space

```
class Space
{
protected:
        friend class World;
        string type; - type of space
        bool power_switch; - each space has a power switch

        Space * up; - space above
        Space * down; - space below
        Space * left; - space to the left
        Space * right; - space to the right

public:
        virtual bool inspect(Stack *) = 0;

        Space(string name); - constructor initializes space name
        virtual ~Space(); - destructor
        void flip_switch(); - change power switch bool assignment
        string get_type(); - returns type

        void inspect_item(Item *); - let's the user know what item they just found

           // Setters and Getters for up, down, left, right members
        Space * get_up();
        Space * get_down();
        Space * get_left();
        Space * get_right();

        void set_up(Space *);
        void set_down(Space *);
        void set_right(Space *);
        void set_left(Space *);

        void set_orthogs(Space ***, int, int);
};
```

Space Subclass Loading_Bay

```
class Loading_Bay : public Space
{
private:
        bool door; - Garage door is either up or down
```

void check_door(); - *User can open door or close door through here*
bool check_truck (Stack *); - *User inspects truck. Returns true if user has key and door is open.*

public:
Loading_Bay(string n = "Loading Bay"); - *Constructor initializes type*
~Loading_Bay(); - *Destructor*
bool is_door_open(); - *Checks if door is open*

virtual bool inspect(Stack *); - *menu that lets users choose what to check in the space. Returns true if users interaction causes them to win the game.*

};

Space Subclass Mess_Hall

class Mess_Hall : public Space
{
private:
Item * table; - *points to item hidden here. If NULL, item has already been removed*
Item * ceiling_fan; - *points to item hidden here. If NULL, item has already been removed*
Item * refridgerator;- *points to item hidden here. If NULL, item has already been removed*
Machine * soda_machine; - *points to machine*

public:
Mess_Hall(string n = "Mess Hall"); - *Constructor initializes type*
virtual ~Mess_Hall(); - *Destructor*

Item * check_table(); - *returns pointer to Item that table points to*
Item * check_ceiling_fan(); - *returns pointer to Item that ceiling fan points to*
Item * check_refridgerator(); - *returns pointer to Item that refridgerator points to*
void check_soda_machine(Stack *); - *User interacts with soda machine. Soda machine has three sodas. Each time users interacts with the machine, the user can choose to add soda to backpack or throw it away.*

Item * get_table(); - *returns the item that table points to and sets table to NULL*
Item * get_ceiling_fan(); - *returns the item that ceiling fan points to and sets ceiling fan to NULL*
Item * get_refridgerator(); - *returns the item that refridgerator points to and sets to NULL*

virtual bool inspect(Stack *); - *provides list of objects in the space for user to interact with. Also let's user interact with their backpack. Always returns false because user cannot win game by interacting with objects here.*

};

Space Subclass Office

class Office : public Space
{
private:
	Item * desk; - Points to item hidden under desk
	Item * check_desk(); - returns pointer to item that desk is hiding
	Item * get_desk(); - returns pointer to item that desk is hiding and sets desk to NULL
public:
	virtual bool inspect(Stack *); - *provides list of objects that user can interact with. Also lets user interact with their backpack. Always returns false because user cannot win game by interacting with items.*

	Office(string n = "Office"); - *constructor initializes type*
	virtual ~Office(); - destructor
};

Space Subclass Bathroom

class Bathroom : public Space
{
private:
	bool stall; - records if user already checked the stall
	Item * locker; - points to the Key Item
public:
	virtual bool inspect(Stack *); - Provides user with a menu of operations that are specific to the Bathroom

	Bathroom(string n = "Bathroom"); - constructor
	virtual ~Bathroom(); - destructor

	void check_right_stall(); - changes stall member
	Item * check_locker(); - returns pointer to item if an item exists, otherwise null
	void check_left_stall(); - displays random message

	Item * get_locker(); - passes back pointer to item and nulls locker member
};

Space Subclass Floor

class Floor : public Space
{
<u>private:</u>
	Conveyor_Belt * conveyor_belt; - Points to conveyer belt item

public:
Floor(string n = "Floor"); - constructor
~Floor(); - destructor

void check_conveyor_belt(Stack *); - facilitates users interaction with conveyor belt
virtual bool inspect(Stack *); - provides list of operations that user can execute that are specific to this area
};

MACHINES

Parent Class Machine
class Machine
{
protected:
string type; - type of machine
bool state; - whether machine is on or off
public:
bool get_state(); - state accessor
void change_state(); - changes state member's value
string get_type(); - type accessor
virtual void start(Stack *) = 0;

Machine(string n); - Machine constructor
virtual ~Machine(); - machine destructor
};

Machine Subclass Conveyor Belt
class Conveyor_Belt : public Machine
{
private:
Node * front; - points to the first node
Node * rear; - points to the last node
bool is_empty(); - returns true if empty (not used)

void rotate(); - rotates the circular queue; the front node is sent to the rear
public:
Conveyor_Belt(string type = "Conveyor Belt"); - Conveyor belt constructor
~Conveyor_Belt(); - conveyor belt destructor

void push(Item *); - creates node and adds to the rear of the queue. Used to create the conveyor belt
Item * check(); - returns pointer to item in front node

Item * get_item(); - returns pointer to item in front node and sets nodes item pointer to NULL

virtual void start(Stack *); - facilitates users interaction with the conveyor belt

};

Machine Subclass Soda Machine

class Soda_Machine : public Machine

{

private:

Queue * contents; - Queue containing nodes pointing to Soda items.

public:

Soda_Machine(string t = "Soda Machine"); - Soda Machine constructor

virtual ~Soda_Machine(); - Soda Machine destructor

virtual void start(Stack *); - Facilitates users interaction with the soda machine

};

STACK (Backpack)

class Stack

{

private:

int num_types; - keeps track of the number of item categories

~~//int num_items;~~ unnecessary

float weight; - running weight total

Node * top; - points to the first node of a doubly linked list.

Node * find(Item *); - searches for item in stack and returns. If not found, returns NULL.

bool is_empty(); - returns true if stack is empty

Node * get_node(int); - returns pointer to node at position

public:

void add(Item *); - adds item to the stack. First checks to see if item of same type already exists. If so, the item is added to the front of the nodes branch. If no item already exists, the item is added to the front of the list.

~Stack(); - stack destructor

Stack(); - stack constructor

bool find(string); - returns true if item type exists in bag

bool check_weight(Item *); - checks whether adding an item will make the bag too heavy. Returns true if adding item is okay.

bool maybe_add(Item *); - facilitates users decision to add item or not. Determines if adding item is possible.

void display(); - displays contents of bag by type. Also displays number of items and weights by type. At the bottom, the total weight of the bag displays.

Item * remove(int pos); - removes and returns pointer to item at a certain position in the bag. The user provides the position of the item that they want to remove.

void open(); - provides user list of possible operations that they can execute.

void debug(); - Not needed. Provided way to troubleshoot issues when testing adding and removing items.

};

NODE

class Node

{

private:

friend class Conveyor_Belt;

friend class Stack;

friend class Queue;

Item * item; - Points to an item pointer, otherwise NULL

Node * next; - points to the next node in the structure.

Node * branch; - Only the Stack/Backpack class uses this. The branch points to the front node of another doubly linked list. In the Stack/Backpack class, if an item was added that already existed in the bag, the item was added to the branch. Hence, the bag was categorized by item and existing items were added to their corresponding type.

*Node * prev;* - points to the node whose next pointer points to this node

int quantity; - generally zero. In the Stack/Backpack class, this member is incremented and decremented when items are added and removed from the branch linked list. It is also updated accordingly when the first item of it's type is created.

float type_weight; - generally zero. In the Stack/Backpack class, this member is incremented and decremented when items are added and removed from the branch class. It is also updated accordingly when the first item of it's type is created.

public:

Node(Item * v = NULL, Node * n = NULL); - constructor, initializes the item pointer and next pointer

~Node(); - destructor

};

RESULTS

Test	Expected	Observed	Passed	Comment
Make sure world is created and user can move around map				
Users is constrained to menu choices and there is no error. Menu updates as user successfully moves to new space.	- Menu only displays valid nearby spaces - User can only enter provided input - Menu updates with new valid options when user changes spaces	1) Segmentation fault 2) Passed	1) Failed - Caused seg fault because valid characters for user to choose did not change with valid spaces 2) Passed - Valid characters are determined when program finds valid spaces	Changed validation function to accept string of valid characters in argument. Originally included valid character string in body of function.
User can add and remove items from their backpack				
- User can pick up an item from the environment and add to bag - User can throw away item in bag	- When user displays bag contents, added items appear - When user throws away item, item disappears from bag	- Passed - although there was some troubleshooting (see comments)	Passed	- Debug function was created to display addresses of each node and their pointers. Some pointers were not being freed/not set to null. (Spec. example is not setting prev pointer to NULL when item became front node)
Make sure game ends appropriately				
- If user has key, opens garage door, and starts the truck, the	Game ends at the right moment and displays	- Passed	-Passed	

game should end and the user should win - If the user uses all of their steps, the game should end and the user should lose - In both cases, user is brought back to main menu game outcome is visible	outcome to user			
Objective is communicated to the user effectively				
User can find information about game	Main menu has option to display information about the game, and is visible to user	Passed	Passed	

DISCUSSION:

I tried to keep things simple with this game so that I could be sure to meet the basic requirements before the deadline. In my original design, I thought about moving a boss character around the map that the user may or may not have to fight. In the end, time constraints caused me to forgo that addition.

While coding, the biggest issue I had was removing items from my bag. As mentioned in the design above, the Stack (i.e bag) class was a doubly-linked list that made use of the Nodes branch member (see node class above for details). If an item was added to the bag that already existed there, the item was added to that nodes branch. As repeated items were added, the branch would extend. The branch is also a doubly linked list. Removing items caused problems depending on where the first item lay in the doubly linked list. Specifically, the front and second items seemed to cause the most issues. I realized after awhile that I was not setting the previous pointer to NULL when the second node became the first. The previous pointer was used when determining how to remove the first node, which caused problems when I tried removing items that were associated directly or indirectly with the first node. Another issue was not setting all the pointers to NULL in the node that I was deleting, causing allocation errors.

While the bag caused the most bugs for me, it was also one of my favorite parts of the project. I decided early on to store identical objects in a branch member and keep track of the number of members in the branch and their added weight in the first node. This decision was largely to make displaying the contents of the bag easier to write and to make the contents easier to read for the end user. My add and remove functions also update the quantity and weight members as the bag was mutated so that details of the contents were always up to date. The machines were also fun to create and were designed off of some of the structures we created in past labs. The Soda Machine class is not very different from a dynamic queue class, and the conveyor belt was based on a circular queue.

If I were to make improvements, I would try to make my structures more efficient. Instead of a dynamic queue, I would make it the Soda Machine a static queue and just use Item pointers in the array. My decision to create dynamic version was largely because of my pre-existing node class that I used to create the Bag. Since the length of the list was predetermined, a static array would probably be more efficient here. I would also use sentinels instead of deleting nodes from the linked lists in my bag. Currently, anytime an item is added or removed, a node is either added or deleted.

Another change I would consider in the future is using exceptions to track when time ran out or if the player won. I had my move and and inspect functions pass back and mutate boolean values to flag when the game should end. In 4 out of the 5 spaces (which housed the inpsect function) false was always returned; only the Loading Bay space could return true since that is where the user finds the truck. Looking back, this solution would probably be a lot cleaner using exceptions.