Final Project

Nathan Mize

Filenames:

SurvivalGame.cpp

hunter.cpp

world.cpp

creature.cpp

resources.h

makefile

1. This program is a very basic survival game that asks the player to collect food fr the upcoming winter to avoid starvation. They can explore the small 5x5 map, hunt animals and forage for food and weapons. Some animals fight back. I picked this for two reasons, I thought about making an RPG because they're one of my hobbies (playing, not coding) but it seemed little involved and a basic survival game would help focus and simplify what I needed to make it work as a game while still being able to meet the requirements.

Requirements:

1) Simple output

The first example of simple out put presented to the user is in SurvivalGame.cpp 'void game::intro()'

```
/*****************************requirement 1*****************************************/
    cout << "You have become lost in a crater with no hope of" << endl
         << "rescue until next spring. It is up to you to " << endl
         << "collect enough food to survive until then." << endl << endl
         << "Good Luck...    you'll need it." << endl << endl;
/***********************************************************************************/
```

2) Simple input:

The function int game::get_integer(int min, int max) in SurvivalGame.cpp gets an integer using cin

```
876  // gets integer from user, with verification
877  /*****************************requirement 2*****************************/
878  int game::get_integer(int min, int max)
879  {
880      int integer;
881      do
882      {
883          cin >> integer;
884          if ((!is_integer()) || (!in_range(integer,min,max)))
885              cout << "not an option, try again: ";
886      } while ((!is_integer()) || (!in_range(integer,min,max)));
887      return integer;
888  }
889  /*************************************************************************/
```

3) Explicit type casting

I used explicit type casting to put a ratio of two integers into a double. Without it the result would have been rounded down to the nearest integer, 0 in this case given the possible values. This is in hunter.cpp, void hunter::print_inventory()

```
146          int food_gathered = 0;
147          int food_consumed;
148          double efficiency;
```

```
165          // explicit type cast lets us get the ratio precisely from two integers
166          // otherwise it would be zero
167          efficiency = double ( food_consumed ) / food_gathered;
```

4) Conditional

Lots of conditionals here, I like this one in SurvivalGame.cpp It goes on for a while but I didn't want to put it in here, ending in else allows it to catch unexpected integer input and still return a valid pick.

```
506  /*****************************requirement 4*****************************/
507      // creatures passed by pointer to avoid slicing problem
508      if (type > 95)  // wolf
509      {
510          wolf beast;
511          hostile *quarry;
512          quarry = &beast;
513          fight_hostile(quarry);
514      }
515
516      else if (type > 90) // bear
517      {
518          bear beast;
519          hostile *quarry;
520          quarry = &beast;
521          fight_hostile(quarry);
522      }
```

5) Logical or bitwise operator

The function bool hunter::dead() in hunter.cpp has a good use of these:

```
263     // returns true if the player has died (0 health or food)
264     bool hunter::dead()
265     {
266         if ((health <= 0) || (food <= 0))
267         {
268             return true;
269         }
270         else
271         {
272             return false;
273         }
274     }
```

6) Loop

The function void game::gameplay() in SurvivalGame.cpp has a good do loop

```
119 // starts and controls the game structure
120 void game::gameplay()
121 {
122     intro();
123     // do loop for turns, no time limit loop ends when player has 0 food, 0 health, or enough
124     do
125     {
126         display();
127         selection();
128     }while ( (!player.dead()) || (player.get_food() >= food_target ) );
129     end_game();
130 }
```

7) Random number

The random number comes along frequently here is a good one in world.cpp world::world() pick gets a random number from 0-3 and then te switch picks a type of location to put in that segment of the map.

```
60  /*****************************requirement 7*************************************/
61                      pick = rand() % 4;
62                      switch (pick)
63                      {
64                          // ***Important note:
65                              // the names of locations currently need to be 6 characters
66                              // long to work with the print function.
67                          case 0:
```

8) Error catagories

I made the function names as descriptive as possible in order to make sure I could avoid syntax errors in using the classes I created. Lots of testing and extra print statement to find and ID run-time errors. I've since removed the print statements though because they tend to be an eyesore in the code.

9) Debugging tricks

The print statements to find runtime errors. Sementation fault drove me nuts until I tried that to ID the problem. Only requested inputs are integer or [y/n] to limit errors in interpreting input. there is input verification for the command line argument in SurvivalGame.cpp function bool is_valid(char **argv). It makes sure that the command

line argument is a positive integer.

```
95  // input verification for argv being a positive integer
96  bool is_valid(char **argv)
97  {
98      for (unsigned int i = 0; i < strlen(argv[1]) ; i++)
99      {
100         if (!isdigit(argv[1][i]))
101             return false;
102     }
103     return true;
104 }
```

10) Function

Take your pick, functions, functions everywhere, the above is the only function outside of main not a member of a class.

11) Functional decomposition

I tried very hard to keep the functions basic as possible and avoid repeated code. One example is the basic function clear_screen() It only saves a one line each time but makes it nice and clear what is going on whenever I need to clear the screen:

```
189 /************************REQUIREMENT 11*******************************/
190 // clears screen
191 void game::clear_screen()
192 {
193     for (int i = 0; i < 50; i++)
194         cout << endl;
195 }
```

12) Scope

Scope allows me to reuse simple variable names like health in class hunter and health in class creature and its subclasses. And int lat and int lon when working with the map.

```
121     // alter name of location at coordinates lat,lon
122     void world::set_name(int lat, int lon, string new_name)
123     {
127
128     // alter food_cost of location at coordinates lat,lon
129     void world::set_cost(int lat, int lon, int new_cost)
130     {
134
135     // alter visited of location at coordinates lat,lon
136     void world::set_visited(int lat, int lon, bool been_here)
137     {
141
```

13) Passing mechanisms

Using member variables so frequently made it so that I didn't have to pass variables by reference explicitly most of the time. In world.cpp the copy constructor syntax required pass by reference:

```
96  /************************REQUIREMENT 13*****************************************/
97      world::world(const world& old_world)
98      {
99          map = new location*[7];
100         for (int i = 0; i < 7; i++)
101         {
102             map[i] = new location[7];
103         }
104         for (int lon = 0; lon < 7; lon++)
105         {
106             for (int lat = 0; lat < 7; lat++)
107             {
108                 map[lat][lon] = old_world.map[lat][lon];
109             }
110         }
111     }
```

14) Function overloading

Most function overloading in my program comes in the constructors for my classes. For example in creature.cpp :

```
33  /*********************************REQUIREMENT 14*****************************/
34      creature::creature()
35      {
36          health = 1;
37          evasion = 1;
38          food_value = 1;
39          name = "placeholder";
40      }
41
42      creature::creature(int new_health, int new_evasion, int new_food, string new_name)
43      {
44          health = new_health;
45          evasion = new_evasion;
46          food_value = new_food;
47          name = new_name;
48      }
```

15) String Variable

Lots of strings for example in resources.h class hunter has a string variable for the name of the weapon the player has.

```
85  /*********************************REQUIREMENT 15*****************************************************
86              string weapon;          // weapon the player is holding
```

16) Recursion

The classic intro to programming example of recursion, the crazy old hermit demands to be told the nth Fibonacci number, in SurvivalGame.cpp:

```
824  /*********************************REQUIREMENT 16*************************************/
825  // recursively determine nth Fibonacci number
826  int game::fibonacci(int n)
827  {
828      if (n <= 0)
829      {
830          return 0;
831      }
832      else if (n == 1)
833      {
834          return 1;
835      }
836      else
837      {
838          return (fibonacci(n-1) + fibonacci(n-2));
839      }
840  }
```

17) Multi-dimensional Array:

The map for the game is stored in a multi-dimensional array in world.cpp:

```
33  /*********************************REQUIREMENT 17*************************************/
34  /*********************************REQUIREMENT 18*************************************/
35          // dynamically declare the array
36          map = new location*[7];
37          for (int i = 0; i < 7; i++)
38          {
39              map[i] = new location[7];
40          }
41
42          int pick;
43          srand(time(NULL));
44          for (int lon = 0; lon < 7; lon++)
45          {
46              for (int lat = 0; lat < 7; lat++)
47              {
48                  // the edges of the array are "impassable" this creates a boundary for
49                  // class game to recognize
50                  if ((lat == 0) || (lon == 0) || (lat == 6) || (lon == 6))
51                  {
52                      map[lat][lon].name = "impassable";
53                      map[lat][lon].food_cost = 0;
54                      map[lat][lon].visited = false;
55                  }
```

18) Dynamically declared array

See above, the multi-dimensional array is declared dynamically. It is deleted in the deconstructor ~world().

19) Command line argument

The program accepts a single positive inteer as a command line argument it is the amount of food required to win the game, so ./game 400 sets th victory condition to 400 food.

```
75  // takes single command line argument of an integer
76  int main(int argc, char **argv)
77  {
78      srand(time(NULL));
79      int victory = 500;
80
81      if (argc == 2)
82      {
83          if (is_valid(argv))
84          {
85              victory = atoi(argv[1]);
86          }
87      }
88
89      game game1(victory);
90      game1.gameplay();
91
92  }
```

20) Struct

The inventory of how much food came from what source is stored in struct. Resources.

```
29  /**********************************REQUIREMENT 16**************************************/
30  // This struct is used to store enteries in the vector inventory
31      struct pack
32      {
33          int food_gained; // the amount of food player added
34          string source;   // were the food came from
35
36  /**********************************REQUIREMENT 30**************************************/
37          // this overloaded operator allows the structs to be sorted by
38          // the size of food_gained when they are in a vector
39          bool operator < (const pack& object) const
40          {
41              return (food_gained < object.food_gained);
42          }
43      };
```

21) Class and object

Nearly everything is done in classes, not really a single one to label.

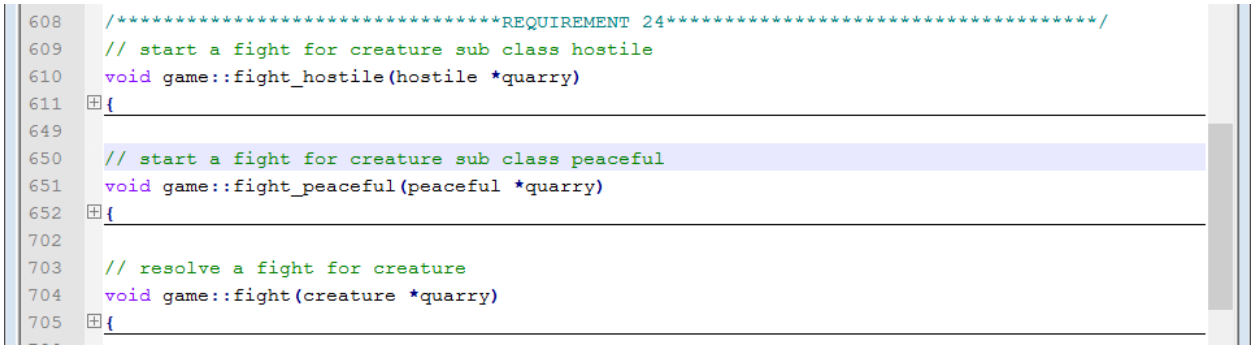
22) pointer to an array

23) pointer to a struct

The map in class world is a dynamic array of pointers to arrays of structs. world.cpp

24) pointer to an object

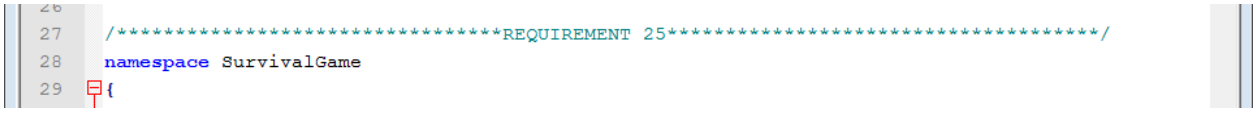
Pointers to objects avoid the slicing problem as I use the subclasses of creature in the following functions:



```
608  /*********************************REQUIREMENT 24************************************/
609  // start a fight for creature sub class hostile
610  void game::fight_hostile(hostile *quarry)
611  {
649
650  // start a fight for creature sub class peaceful
651  void game::fight_peaceful(peaceful *quarry)
652  {
702
703  // resolve a fight for creature
704  void game::fight(creature *quarry)
705  {
```

25) Custom Namespace

My classes creature, world and hunter are in the custom namespace SurvivalGame for example in resources.h:



```
27  /*********************************REQUIREMENT 25************************************/
28  namespace SurvivalGame
29  {
```

26) header file

resources.h

27) Makefile

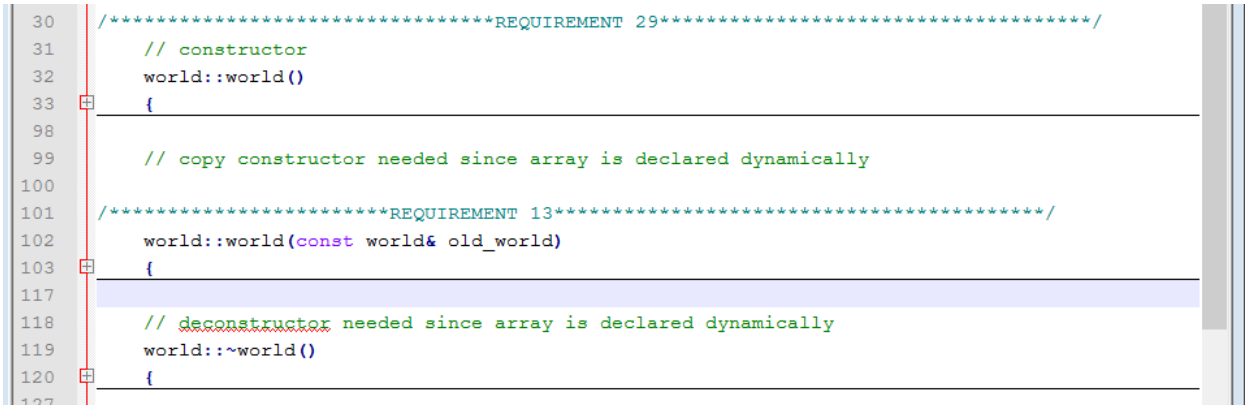
Included

28) vector

I use an vector of structs to store the inventory in class hunter. See resources.h

29) constructors

used in the class world. See world.cpp



```
30  /*********************************REQUIREMENT 29************************************/
31      // constructor
32      world::world()
33      {
98
99      // copy constructor needed since array is declared dynamically
100
101 /************************REQUIREMENT 13*****************************************/
102     world::world(const world& old_world)
103     {
117
118     // deconstructor needed since array is declared dynamically
119     world::~world()
120     {
```

30) overloaded operator

The overloaded operator < in the struct pack allows the vector<pack> to be sorted by the STL algorithm function sort(). Wihtout it it wouldn't know how to compare them and sort them.

```
32      struct pack
33      {
34          int food_gained; // the amount of food player added
35          string source;   // were the food came from
36
37  /*********************************REQUIREMENT 30*************************************/
38          // this overloaded operator allows the structs to be sorted by
39          // the size of food_gained when they are in a vector
40          bool operator < (const pack& object) const
41          {
42              return (food_gained < object.food_gained);
43          }
44      };
```

31) File IO

The function void hunter::file_inventory(string output) in hunter.cpp shows using file IO to output to a file of the users choosing:

```
184     void hunter::file_inventory(string output)
185     {
186         int food_gathered = 0; // food gathered so far, totaled as inventory is output
187         int food_consumed;     // food player has used
188         double efficiency;     // how much of the food gathered has been consumed
189
190         // sorts inventory so that smallest food values are printed first
191         sort(inventory.begin(), inventory.end());
192
193         // set output stream and open file
194         std::ofstream ofs;
195         ofs.open(output.c_str());
196
197         // print player's weapon
198         ofs << "Your current weapon is " << weapon << "." << endl << endl;
199
```

32) STL

I used the sort() function from <algorithm> to sort a vector of structs It required overloading the < operator for the struct otherwise it would have been unable to make the comparison in hunter.cpp

```
21  /*********************************REQUIREMENT 32*************************************/
22  #include <algorithm>
```

```
150 /*********************************REQUIREMENT 32*************************************/
151         // sorts inventory so that smallest food values are printed first
152         sort(inventory.begin(), inventory.end());
153
```

33) inheritance

This is illustrated in the creature class and it's attendant sub classes see creature.cpp

34) Polymorphism

This is illustrated in the get damage function of creature and its subclasses. By making the function virtual it has the program determine at run time which to call so when a creature attacks the commentary can be very specific without resorting to long conditional statements.

35) Exceptions

The funtion void world::check_coord(int lat, int lon) illustrates exceptions. It is used in the other member functions of world to avoid junk data or segmentation faults from providing the wrong paramaters for latitude and longitude.

```
235 ******************************REQUIREMENT 35************************************/
236 // function throws an exception exiting the program if lat,lon is outside the array
237 void world::check_coord(int lat, int lon)
238 {
239     try
240     {
241         if ((lat < 0) || (lat > 7) || (lon < 0) || (lon > 7))
242         {
243             throw 1;
244         }
245     }
246     catch (int a)
247     {
248         cout << "This location is beyond the edge of the map! " << lat << "," << lon << endl
249              << "Terminating program..." << endl << endl;
250         exit(1);
251     }
252 }
```

36) Something Awesome

The game came out really well, especially considering I haven't learned to use a GUI library.

37) something else

I think the map display and navigation in my game is another something awesome.

Reflection:

Funnily enough I gained quite the appreciaton for working under a deadline as a programmer. In the past everything I did in programming has been on my own time. I have a lot more sympathy for game studios that have to delay releases. I've had to tun in several programs when I know I could make them so much better with just a few more days. It has taught me to focus on good design work ahead of coding and focusing on end user requirements.