Chapter 7 Arcade Live Action Games

Live action arcade style games present new challenges, particularly in timing and in the handling of the keyboard. In these games, a turn represents a finite amount of real time. Thus, the turn must be based upon the computer's clock time. How much time is allowed for a turn requires fine tuning. If the turn time is too large, the game appears lethargic, while if the turn time is too short, the action goes faster than a player can respond.

Further, the keyboard must be checked once each turn to see if the player has entered a command. Thus, arcade games must in effect poll the keyboard. If no key press has been made, present the next action sequence. If a key has been pressed, obtain the command and, if valid, carry out the command before handling the remaining action sequence. Normal C++ iostreams and C functions such as **_getch()** cannot be used, since these functions wait until a valid key has been pressed. By valid key is meant a standard key stroke, such as a letter, number, or special key. Merely pressing the shift or alt key is not a standard key stroke. All these functions wait for the user to enter a key stroke, which completely stops the game action. Hence, we need to utilize more control over the keyboard which means we must use some Windows input functions which can test for a keyboard event. If one has occurred, we can then go ahead and retrieve it.

Let's examine the class KeyBoard first.

Creating a Real-time Keyboard Handler Class

Handling the keyboard directly requires the use of only a few Windows functions. First, we must obtain a handle to the keyboard, via **GetStdHandle()**. Once we are finished with the keyboard, we must release that handle back to Windows, using the **FreeConsole()** function. These two calls are optimally done in the constructor and destructor functions. In the ctor, we get the keyboard by coding the following.

```
hKeyBd = GetStdHandle (STD_INPUT_HANDLE);
 if (hKeyBd == INVALID_HANDLE_VALUE)// failed
```

In the dtor, we call

```
FreeConsole ();
```

To peek ahead and see if there is a keystroke waiting for us, we use the **PeekConsoleInput()** function. This function is passed the keyboard handle and an array of **INPUT_RECORD** structures, which will contain all of the console input requests since the last time that we checked. Windows states that an array of up to sixty-four elements might be needed. Hence, we also pass the function how many elements we have in our array. Finally, the function fills up one of our data members with just how many elements are actually in use this time and returns a BOOL, TRUE if it is successful.

Given these variables,

```
const int MAXBUF = 64;
INPUT_RECORD ir[MAX_BUF];
DWORD num;
```

the call to **PeekConsoleInput()** is done this way.

```
BOOL res = PeekConsoleInput (hKeyBd, ir, MAXBUF, &num);
```

The structure is defined as:

```
struct INPUT_RECORD {
  WORD EventType;
  union {
    KEY_EVENT_RECORD KeyEvent;
    MOUSE_EVENT_RECORD MouseEvent;
    WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
    MENU_EVENT_RECORD MenuEvent;
    FOCUS_EVENT_RECORD FocusEvent;
  } Event;
};
```

Exactly what the second member actually is depends upon the value in the **EventType** member. All input events are logged, including mouse, focus, and keyboard. Here we are only interested in keyboard events, all others are ignored. The value we need is **KEY_EVENT**. In this case the second member contains a **KEY_EVENT_RECORD** structure with information about a keyboard event.

This structure is defined:

```
struct KEY_EVENT_RECORD {
  BOOL bKeyDown;
  WORD wRepeatCount;
  WORD wVirtualKeyCode;
  WORD wVirtualScanCode;
  union {
    WCHAR UnicodeChar;
    CHAR AsciiChar;
  } uChar;
  DWORD dwControlKeyState;
};
```

The meaning of these members are as follows. **bKeyDown**: **TRUE**, if the key is pressed, **FALSE** if released. That is, an event occurs when the key is pressed down and then another event is signaled when the key is released. We are only interested in those events in which the key is pressed down and must ignore the released events.

wRepeatCount is the number of times this key has been pressed. This occurs when a key is pressed and held down for a length of time, triggering the auto-repeat feature, which may say that the letter 'a' was pressed five times, for example.

wVirtualKeyCode is the Windows identifier of which key is pressed. These codes are device-independent. The **wVirtualScanCode** is the code that represents the device-dependent value generated by the keyboard hardware. For this arcade game, we are not interested in anything but the virtual key codes. If you wanted the actual ASCII character, use the **AsciiChar** member. The **dwControlKeyState** indicates which other keys are also held down: CAPSLOCK_ON, ENHANCED_KEY, LEFT_ALT_PRESSED, LEFT_CTRL_PRESSED, NUMLOCK_ON, RIGHT_ALT_PRESSED, RIGHT_CTRL_PRESSED, SCROLLLOCK_ON, and SHIFT_PRESSED. These are self-explanatory.

Please note that the ALT key, when pressed and released without combining with another character, has special meaning to the system and is not passed through to our application. It is handled by Windows. For example, Alt+tab switches between running applications.

Our coding follows the function call by checking for the presence of a key event by examining each element of the event array, returning true, a key event has occurred.

```
for (int i=0; i<(int) num; i++) {
  if (ir[i].EventType == KEY_EVENT) return true;
}
return false;
```

The next function that we need is the ability to flush all remaining events from the system. For example, if the player is holding down the left arrow key asking to continually move to the left, and the character falls through a hole in the floor, we must cancel all subsequent move left keystrokes, since these are now invalid. The **FlushConsoleInputBuffer()** function does this.

```
FlushConsoleInputBuffer (hKeyBd);
```

Finally, the last function that we need is to actually retrieve the key code of the pressed key. This is done using the **ReadConsoleInput()** function, which is similar to the peek function.

```
BOOL res = ReadConsoleInput (hKeyBd, ir, MAXBUF, &num);
```

Once more, we iterate through the events looking for the keyboard key press event. Once found, let's translate that into a Command enum value that the game can readily use.

```
// the possible game commands
enum Command {Stop, GoLeft, GoRight, GoUp, GoDown, Jump,
              DoNothing, Abort};
```

Here are the KeyBoard class definition and implementation.

```
Class KeyBoard Definition

 1 #pragma once
 2 #include "Windows.h"
 3 #include "Object.h"
 4
 5 const int MAXBUF = 64;
 6
 7 /***************************************************************/
 8 /*                                                             */
 9 /* KeyBoard: encapsulates real time keyboard                   */
10 /*                                                             */
11 /***************************************************************/
12
13 class KeyBoard {
14 protected:
15 HANDLE       hKeyBd;     // the keyboard handle
16 INPUT_RECORD ir[MAXBUF]; // array of keyboard events
17 DWORD        num;        // the number of events
18
19 public:
20   KeyBoard ();
21  ~KeyBoard ();
22
23  bool Peak ();          // returns true if a key stroke is present
24  Command GetKey ();     // returns the key stroke as a command
25  void Flush ();         // flushes the keyboard buffer
26 };
```

```
Class KeyBoard Implementation

  1 #include "KeyBoard.h"
  2
  3 /***************************************************************/
  4 /*                                                             */
  5 /* KeyBoard: acquire handle to the key board                   */
  6 /*                                                             */
  7 /***************************************************************/
  8
  9 KeyBoard::KeyBoard () {
 10  hKeyBd = GetStdHandle (STD_INPUT_HANDLE);
 11  if (hKeyBd == INVALID_HANDLE_VALUE)
 12   throw "Error: Unable to get the std keyboard handle";
 13 }
 14
 15 /***************************************************************/
 16 /*                                                             */
 17 /* ~KeyBoard: free up the keyboard                             */
 18 /*                                                             */
 19 /***************************************************************/
 20
 21 KeyBoard::~KeyBoard () {
 22  FreeConsole ();
 23 }
 24
 25 /***************************************************************/
 26 /*                                                             */
 27 /* Peak: returns true if a key stroke is present               */
 28 /*                                                             */
 29 /***************************************************************/
 30
 31 bool KeyBoard::Peak () {
 32  BOOL res = PeekConsoleInput (hKeyBd, ir, MAXBUF, &num);
 33  if (!res) return false;
 34  for (int i=0; i<(int) num; i++) {
 35   if (ir[i].EventType == KEY_EVENT) {
 36    return true;
 37   }
 38  }
 39  return false;
 40 }
 41
 42 /***************************************************************/
 43 /*                                                             */
 44 /* Flush: empties keyboard buffer of all key strokes           */
 45 /*                                                             */
 46 /***************************************************************/
 47
 48 void KeyBoard::Flush () {
 49  FlushConsoleInputBuffer (hKeyBd);
```

```
50 }
51
52 /****************************************************************/
53 /*                                                              */
54 /* GetKey: returns the key stroke as a game command             */
55 /*                                                              */
56 /****************************************************************/
57
58 Command KeyBoard::GetKey () {
59  Command cmd = DoNothing;
60  BOOL res = ReadConsoleInput (hKeyBd, ir, MAXBUF, &num);
61  if (!res) return cmd;
62  for (int i=0; i<(int) num; i++) {
63   if (ir[i].EventType == KEY_EVENT &&
         ir[i].Event.KeyEvent.bKeyDown) {
64    if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_SPACE) {
65     cmd = Jump; break;
66    }
67    else if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_UP) {
68     cmd = GoUp; break;
69    }
70    else if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_DOWN) {
71     cmd = GoDown; break;
72    }
73    else if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_LEFT) {
74     cmd = GoLeft; break;
75    }
76    else if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_RIGHT) {
77     cmd = GoRight; break;
78    }
79    else if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_ESCAPE) {
80     cmd = Stop; break;
81    }
82    else if (ir[i].Event.KeyEvent.wVirtualKeyCode == VK_CANCEL) {
83      cmd = Abort; break;
84    }
85   }
86  }
87  Flush ();
88  return cmd;
89 }
```

Handling Time

The **time()** function returns the current time in many forms. By using zero as the parameter, the function returns the time in milliseconds. Thus, a game's **Run** function would begin a turn by getting and saving the current time, carry out the turn's actions, and then get the current time at the end of the turn. Assume that variable **gameSpeed** holds how long a turn should last in milliseconds. The loop then can calculate how much of the **gameSpeed** milliseconds remain, stored in **waitTime**. If the game needs to pause, the **Sleep()** function is called.

```
void Game::Run () {
 time_t startTime;             // clock time at start of turn
 time_t endTime;               // the time at the end of turn
 while (!done) {               // repeat until game is over
  startTime = time (0);        // get turn starting time
. . . take all necessary turn actions
  endTime = time (0);          // get the turn ending time
  // calculate how long we need to wait for the next turn
  long waitTime = gameSpeed- (long) (endTime - startTime);
  // wait as needed before starting next turn
  if (waitTime > 0) Sleep (waitTime);
 }
```

Hence, we have a simple, effective way to handle real-time game turns. Now we need an arcade style game to implement.

The Ladder Game

I well remember the first home computer I bought, a 9" green screen KayPro II. The Ladder game came with it. Figure 7.1 shows our version of the Ladder game in action. Th ‘p’ character represents the player whose task it is to climb to the very top ‘$’ location to win. The opponents are rolling, falling barrels, represented by ‘o’ characters. The ‘&’ characters represent money bags which the player might obtain as he or she moves along the route.

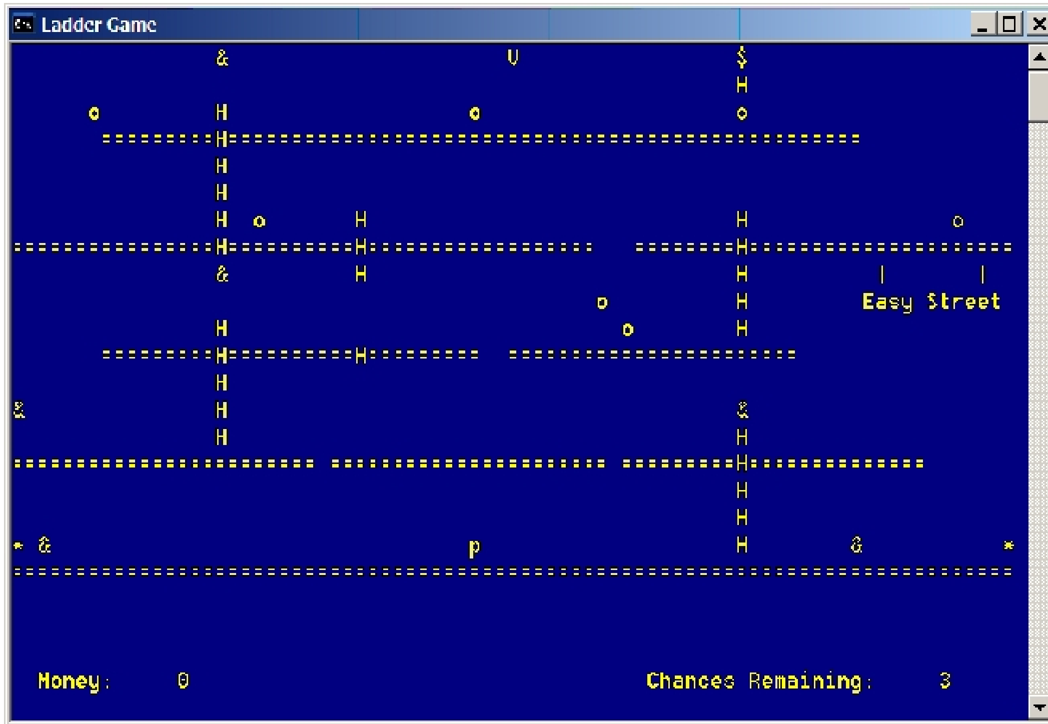

Figure 7.1 The Ladder Game in Operation

The ‘H’ represent ladders which may be climbed or descended. The gaps in the ‘=’ floors are holes which must be jumped over or else the player falls back down to the floor below. The ‘V’ character is the location from which new barrels appear, and the barrels are destroyed when they hit the ‘*’ characters on the bottom row.

The only commands that the player of the game needs are those to indicate the direction of movement (the four arrows), a command to jump, and a command to temporarily pause the game in progress. The jump command is the space bar, while the pause command is the ESC key. Pressing Ctrl-C will abort the game. Hence, the keyboard input is extremely simple indeed.

Theoretically, when the player reaches the exit point, the ‘$’ character, he or she moves into the next level, which should be more challenging or difficult.

If a barrel hits the player 'p,' the player is automatically repositioned back at the starting point of the level. Three chances are given for the player to succeed in reaching the exit point.

Along the way, the player may grab some money bags. Additionally, a larger amount is awarded upon successful completion of a level. If you wish, you can change "money" to "points."

Each level is stored as a text file for easy creation of levels. Figure 7.2 shows the text file for "Easy Street."

```
                &                      V                 $
                                                         H
                H                                        H
       =========H==================================================
                H
                H
                H          H                             H
================H==========H==================   ========H=====================
                &          H                             H          |       |
                                                         H         Easy Street
                H                                        H
       =========H==========H=========  =======================
                H
&               H                                        &
                H                                        H
======================== ====================== =========H==============
                                                         H
                                                         H
* &  p                                                   H        &           *
===============================================================================
```

Figure 7.2 The Text File: EasyStreet.lvl

Any level must have a starting location for the player given by the 'p.' Each level must have a barrel generator location, given by the 'V' code. Each level must have an exit point, given by the '$' code. Each level must have one or more locations where barrels are destroyed, given by the '*' codes. Other than these specifications, the level creation is up to the game designer. However, when making holes in the floor, none can be greater than three columns wide; the maximum a player may jump is over three columns.

What represents the difficulty or challenge level? Ah, there are several ways levels can be made more challenging. First, is the speed of a turn. If the speed is too fast, the player cannot react fast enough and gets clobbered by the barrels. Next, the total quantity of barrels in play at one moment in time is a factor. There is a huge difference between having only a single barrel somewhere in the level versus having fifty barrels scattered about the level. Finally, to a much lesser extent, how many game turns elapse before a new barrel appears, subject to the maximum number of allowed barrels, impacts the challenge. These three parameters can be adjusted to create an easy or exceedingly challenging game out of nearly any level.

Further, one could limit the total amount of time a player has to reach the exit point, but this is not done in this sample program.

Since this is a sample program to illustrate real-time action, the program will allow the player to set the game speed and maximum number of barrels before the game begins. Figure 7.3 shows the opening screen.

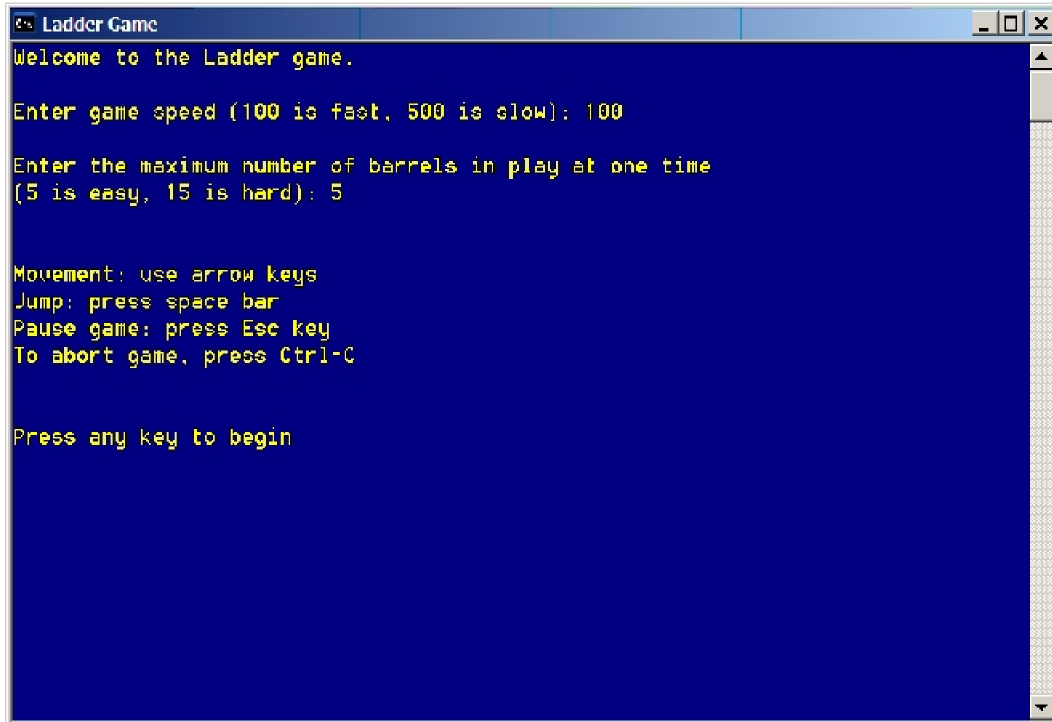

Figure 7.3 Opening Screen to Adjust Game Parameters

The Classes for the Ladder Game

Okay, design time. What classes do we need for this game? First, ask what objects are involved? We can reuse our Screen class from chapter 1 to handle the screen, along with our new KeyBoard class for the real-time keystrokes. We have the actual level as input from a file, so let's encapsulate that into a Level class. Next, we have both players and barrels. However, there are a lot of similarities between the player and the barrels, both are objects which move around the level. Hence, let's encapsulate the basic behavior of something that moves about the level in an Object class. Then, we derive class Player and Class Barrel from class Object, providing specific variations for each of these two objects. The Game class will then create all of these objects and actually run the game.

However, in this game, which is basically a two-dimensional playing field, every object has an x-y coordinate location. It makes sense to have a simple Point structure to encapsulate the x and y values.

The Point Structure

A structure can have member functions. However, unlike a class, all member functions are public access, just as all data members of a structure have public access. Thus, we can make convenient constructor functions as well as implement operator== for easy comparisons of two point structure instances. Here the three functions are implemented inline.

The Point Structure

```
 1 #pragma once
 2 
 3 /***************************************************************/
 4 /*                                                             */
 5 /* Point: encapsulates x,y location                            */
 6 /*                                                             */
 7 /***************************************************************/
 8 
 9 struct Point {
10  int row;
11  int col;
12 
13  Point (int r, int c) { row = r; col = c; }
14  Point () { row = 0; col = 0; }
15  bool operator== (const Point& p) {
16                   return p.row == row && p.col == col; }
17 };
18 
```

The Level Class

The Level class must load and store the level text file. Essentially, a level is nothing more than a two dimensional array of char. A maximum of 24 rows of up to 80 columns is allowed. However, we really do need a couple rows at the bottom to display messages, so perhaps the maximum number of rows should be reduced to 22. Since the actual number of rows in a level can vary, the class stores the number of rows in use in this level.

The Level Class Definition

```
 1 #pragma once
 2 #include "Screen.h"
 3 #include "Point.h"
 4
 5 /***************************************************************/
 6 /*                                                             */
 7 /* Level: encapsulates the playing field, loaded from a file   */
 8 /*                                                             */
 9 /***************************************************************/
10
11 const int MAXROWS = 24;
12 const int MAXCOLS = 80;
13
14 class Level {
15 protected:
16  char field[MAXROWS][MAXCOLS];
17  int numRows;
18
19 public:
20   Level ();
21  ~Level ();
22
23  void LoadLevel (const char* filename);
24
25  int GetNumRows () const { return numRows; }
26
27  const char* GetRow (int i) const { return field[i]; }
28
29  // removes the player's initial location
30  void RemoveInitialP (Point& at);
31
32  // display the entire level - this is only done once
33  void Render (Screen& s);
34
35  // returns the char at this location
36  char GetChar (int row, int col) const { return field[row][col];}
37
38  // replaces the char at this location
39  void ReplaceChar (int row, int col, char c) {
40                  field[row][col] = c; }
```

```
41 };
```

Most of the functions are obvious. Once loaded, the game needs to locate the initial player's location and save it, in case the player is hit by a barrel and must begin again. Thus, RemoveInitialP() will eliminate the 'p' character, once we know where the starting point is located. When a player grabs a money bag, that character must be permanently removed from play, in case the player is hit by a barrel and has to start over. That is, once grabbed, a money bag disappears permanently from play.

All of the function bodies are very straight-forward.

The Level Class Implementation

```
 1 #include <iostream>
 2 #include <fstream>
 3 using namespace std;
 4 #include "Level.h"
 5
 6 Level::Level () {}
 7
 8 Level::~Level (){}
 9
10 /***************************************************************/
11 /*                                                             */
12 /* LoadLevel: inputs the level from a text file               */
13 /*                                                             */
14 /***************************************************************/
15
16 void Level::LoadLevel (const char* filename) {
17  ifstream infile (filename);
18  if (!infile) {
19   char msg[500] = "Error: cannot load file: ";
20   strcat_s (msg, sizeof(msg), filename);
21   throw msg;
22  }
23  int i = 0;
24  while (i < MAXROWS &&
25         infile.getline (field[i], sizeof (field[i]))) {
26   i++;
27  }
28  char c;
29  if (i == MAXROWS && infile >> c)
30   throw "Error: level file contains too many rows";
31  infile.close();
32  numRows = i;
33 }
34
35 /***************************************************************/
36 /*                                                             */
```

```
 37 /* RemoveInitialP: remove the 'p' character - the player       */
 38 /*                                                             */
 39 /***************************************************************/
 40 
 41 void Level::RemoveInitialP (Point& at) {
 42  field[at.row][at.col] = ' ';
 43 }
 44 
 45 /***************************************************************/
 46 /*                                                             */
 47 /* Render: display the entire level                            */
 48 /*                                                             */
 49 /***************************************************************/
 50 
 51 void Level::Render (Screen& s) {
 52  for (int i=0; i<GetNumRows(); i++) {
 53   s.GoToXY (0, i);
 54   s << GetRow (i);
 55  }
 56 }
```

The Object Class

The Object class encapsulates the basic behavior of player and barrels. **enum**s are vital to making readable code. The **enum** Direction tracks which of the four directions an object has just moved, along with None. The **enum** Command tracks what the current command from the keyboard is. The enum JumpStatus handles the ending situation of each portion of a jump. Notice that a player can make a jump and end up catching a ladder. Equally, the end of a jump can land on a hole, so the player is now falling.

However, the purpose of the **enum** Location is not obvious. When we are going to attempt to implement a player's command or the next motion of a barrel, we must know what the level characters immediately around the object's current position actually are. For example, the player may press the up arrow, we need to know that the current character in the level at the player's location is an 'H' for ladder. The player may be moving left, we need to know whether or not there is a money bag there, whether there is even a space to move left into, and so on. The key spots for any possible move are nine in total: where the player is at, the up, down, left, and right spots, along with up and left, up and right, down and left, and down and right. The Location **enum** provided an easy way to keep track of these and will be used as subscripts into a single dimensioned array of these level characters.

The array grid, stores these nine character values of the object at hand. The Location **enum** values are used as subscripts into the gird array, making very convenient coding. Before the object can move, the Game class must fill up this grid based upon the object's current position. While the coding could continue to directly access the Level's master 2-d array, this approach greatly improves execution speed, replacing 2-d look-ups with 1-d's.

Object encapsulates the Point where the object is currently located. It stores the previous direction of motion. This is critical when jumping, since a jump may take several game turns to complete. At each portion of the jump, we need to know in which direction to continue the motion. Similarly, when an object falls through a hole, several turns may pass before it lands. Hence, isFalling is set while the object falls.

The current command is stored as well.

The member symbol represents the symbol to display for the object. It will be a 'p' for the player and an 'o' for a barrel. However, when an object moves into a new location, it may well temporarily hide something in the background. Specifically, the player or barrel may hide a ladder character, a money bag, etc. Hence, the member saveSymbol stores the character of the level behind the currently displayed object character. Once the object moves from this location, the saveSymbol is redisplayed once more, when appropriate.

```
The Object Class Definition
```

```
 1 #pragma once
 2 #include <iostream>
 3 using namespace std;
 4 #include "Point.h"
 5
 6 /**************************************************************/
 7 /*                                                            */
 8 /* Object: base class for player and barrels                  */
 9 /*                                                            */
10 /**************************************************************/
11
12 // the direction of the last movement
13 enum Direction {MovedLeft, MovedRight, MovedUp, MovedDown, None};
14
15 // the locations surrounding a given point
16 enum Location {At, Left, Right, Up, UpRight, UpLeft, Down,
17                DownRight, DownLeft};
18 const int MaxAround = 9; // number of locations
19
20 // the possible game commands
21 enum Command {Stop, GoLeft, GoRight, GoUp, GoDown, Jump,
22               DoNothing, Abort};
23
24 // the status of a jump action
25 enum JumpStatus {Okay, IllegalMove, CaughtLadder, Falling};
26
27
28 class Object {
29 protected:
30  Point     at;              // current location
31  Direction dir;             // previous direction of motion
32  bool      isFalling;       // true if is falling downwards
33  char      symbol;          // char to use for this display
34  char      saveSymbol;      // char that this one displays over
35  char      grid[MaxAround];// chars around this one,set each turn
36  Command   cmd;             // the current command
37
38 public:
39  Object () : at(Point (0,0)), dir(None), isFalling(false),
40   symbol(' '), saveSymbol(' '), cmd(Stop) {}
41
42  ~Object () {}
43
44  void  SetLocation (const Point& p) {at = p; }
45  Point GetLocation () const { return at; }
46
47  Direction GetDirection () const { return dir; }
48  void      SetDirection (Direction d) { dir = d; }
49
50  bool  GetIsFalling () const { return isFalling; }
51  void  SetIsFalling (bool fall) { isFalling = fall; }
52
```

```
53  char  GetSymbol () const { return symbol; }
54  void  SetSymbol (char sym) { symbol = sym; }
55
56  char  GetSavedSymbol () const { return saveSymbol; }
57  void  SetSavedSymbol (char s) { saveSymbol = s; }
58
59  void  SetGrid (const char g[]) { memcpy (grid, g,sizeof(grid));}
60  char  GetGridLoc (Location l) const { return grid[l]; }
61
62  Command GetCommand () const { return cmd; };
63  void    SetCommand (Command c) { cmd = c; }
64
65  virtual bool CanGoUp () const = 0;
66  virtual bool CanGoDown () const = 0;
67  virtual bool CanGoLeft () const = 0;
68  virtual bool CanGoRight () const = 0;
69 };
```

Finally, note that the four CanGoxxx functions are pure virtual functions, making Object an *abstract base class*. The rules for motion differ between a barrel and the player. For example, a barrel can never go up. All member functions have been implemented inline.

The Barrel Class

Next, let's examine the Barrel class, because it is not as complicated as the Player class. It has three static data members. Remember that static data are available whether or not there are any instances of the class in existence. Only one copy of static member data are ever stored, independent of how many instances of the class exist. All three static data members are points.

We need to store the generator of barrels' location, that is, where all new barrels appear. Likewise, we need to store the two locations at which a barrel is destroyed when it reaches them.

For convenience, references to the Screen and Level classes are stored as data members.

The only other data member is a toggle switch. When a barrel lands upon the floor, which direction does it begin to roll? If we always have barrels going, say right, then large sections of any level will always be devoid of the presence of barrels. Hence, each time a barrel lands after a fall, it changes this bool. Thus, barrels can go either direction, making the game more challenging.

The HitPlayer() function compares the barrel's location to that of the player's, returning true if the player is hit. The Game class then handles the collision.

The Barrel Class Definition

```
  1 #pragma once
  2 #include "Point.h"
  3 #include "Level.h"
  4 #include "Screen.h"
  5 #include "Object.h"
  6
  7 /***************************************************************/
  8 /*                                                             */
  9 /* Barrel: encapsulates the moving obsticles                   */
 10 /*                                                             */
 11 /***************************************************************/
 12
 13 class Barrel : public Object {
 14 protected:
 15 static Point generator;     // the location of barrel generator
 16
 17 // barrels are destroyed when they arrive at these locations
 18 static Point rightDestroy;
 19 static Point leftDestroy;
 20
 21 Screen& s;                 // the screen
 22 Level&  l;                 // the level matrix
 23
 24 bool    afterFallGoRight;  // when a falling barrel lands, go
```

```
25                            // right or left - alternates
26
27 public:
28   Barrel (Screen& ss, Level& ll);
29  ~Barrel ();
30
31  void SetAfterFallGoRight (bool s) { afterFallGoRight = s; }
32
33  // returns true if it hits the player
34  bool HitPlayer (const Point& p) {
35                 return at.row == p.row && at.col == p.col; }
36
37  Point MoveBarrel ();          // handles moving the barrel
38  void Render (const Point& p); // displays the barrel
39  void InitialRender ();        // first time display of barrel
40  void FinalRender ();          // last time display of barrel
41
42  bool CanGoUp () const;
43  bool CanGoDown () const;
44  bool CanGoLeft () const;
45  bool CanGoRight () const;
46
47  Point MoveDown ();
48  Point MoveLeft ();
49  Point MoveRight ();
50
51  void  SetIsFalling ();
52
53  // given the level, find where the barrel generator is located
54  static void FindGeneratorInitialLocation (const Level& l);
55 };
```

Most of the functions are self-evident. However, one time only the static member function FindGeneratorInitialLocation() must be called to find where in this level the barrels are to appear, along with the two destruction points. Perhaps this function is mis-named.

The Barrel Class Implementation

```
 1 #include <iostream>
 2 using namespace std;
 3 #include "Barrel.h"
 4
 5 Point Barrel::generator;
 6 Point Barrel::rightDestroy;
 7 Point Barrel::leftDestroy;
 8
 9 /***************************************************************/
10 /*                                                             */
11 /* Barrel: make new barrel just below the generator            */
12 /*                                                             */
```

```
13 /***********************************************************/
14
15 Barrel::Barrel (Screen& ss, Level& ll) : Object (), s(ss), l(ll){
16  at.col = generator.col;
17  at.row = generator.row + 1;
18  symbol = 'o';
19  dir = MovedDown;
20  isFalling = true;
21  saveSymbol = ' ';
22  afterFallGoRight = false;
23 }
24
25 Barrel::~Barrel () { }
26
27 /***********************************************************/
28 /*                                                         */
29 /* MoveBarrel: move barrel either down, left or right      */
30 /*                                                         */
31 /***********************************************************/
32
33 Point Barrel::MoveBarrel () {
34  Point oldAt = at;
35  // if the barrel is at either of the two destruction points,
36  // remove this barrel from play
37  if (at == leftDestroy || at == rightDestroy)
38   return Point (42, 42); // destroy this barrel
39
40  // handle a falling barrel
41  if (isFalling) {
42   char c = GetGridLoc (Down);
43   if (c == '&') c = ' '; // ignore money locations
44   if (c != ' ') {        // detect landing and stop falling
45    isFalling = false;    // set move direction, alternating
46    dir = afterFallGoRight ? MovedRight : MovedLeft;
47   }
48   else at.row++;         // here, continue to fall
49   return oldAt;          // return the original location
50  }
51  // barrels always move in one direction until they fall
52  // so set cmd to left or right
53  cmd = (dir == MovedLeft) ? GoLeft : GoRight;
54  // handle moving right - if at a barrier, reverse directions
55  if (cmd == GoRight) {
56   if (CanGoRight ())
57    oldAt = MoveRight ();
58   else {
59    dir = MovedLeft;
60    oldAt = MoveLeft ();
61   }
62  }
63  // handle moving left - if at a barrier, reverse directions
64  else if (cmd == GoLeft) {
```

```
| 65   if (CanGoLeft ())                                                |
| 66    oldAt == MoveLeft ();                                           |
| 67   else {                                                           |
| 68    dir = MovedRight;                                               |
| 69    oldAt = MoveRight ();                                           |
| 70   }                                                                |
| 71  }                                                                 |
| 72  return oldAt;                                                     |
| 73 }                                                                  |
| 74                                                                    |
| 75 /***************************************************************/|
| 76 /*                                                             */|
| 77 /* InitialRender: show the barrel for the first time           */|
| 78 /*                                                             */|
| 79 /***************************************************************/|
| 80                                                                    |
| 81 void Barrel::InitialRender () {                                    |
| 82  saveSymbol = l.GetChar (at.row, at.col);                          |
| 83  s.OutputUCharWith (symbol, at.row, at.col, Screen::Blue,          |
| 84                     Screen::BrightYellow);                         |
| 85 }                                                                  |
| 86                                                                    |
| 87 /***************************************************************/|
| 88 /*                                                             */|
| 89 /* FinalRender: remove the barrel from the screen permanently  */|
| 90 /*                                                             */|
| 91 /***************************************************************/|
| 92                                                                    |
| 93 void Barrel::FinalRender () {                                      |
| 94  s.OutputUCharWith (saveSymbol, at.row, at.col, Screen::Blue,      |
| 95                     Screen::BrightYellow);                         |
| 96 }                                                                  |
| 97                                                                    |
| 98 /***************************************************************/|
| 99 /*                                                             */|
|100 /* Render: clear old image and show new image at new location  */|
|101 /*                                                             */|
|102 /***************************************************************/|
|103                                                                    |
|104 void Barrel::Render (const Point& oldAt) {                         |
|105  // restore symbol the barrel may be hiding                        |
|106  s.OutputUCharWith (GetSavedSymbol(), oldAt.row, oldAt.col,        |
|107                     Screen::Blue, Screen::BrightYellow);           |
|108  // save the symbol the barrel is about to overlay, if any         |
|109  SetSavedSymbol (l.GetChar (at.row, at.col));                      |
|110  // show barrel at this new location                               |
|111  s.OutputUCharWith (symbol, at.row, at.col, Screen::Blue,          |
|112                     Screen::BrightYellow);                         |
|113 }                                                                  |
|114                                                                    |
|115 /***************************************************************/|
|116 /*                                                             */|
```

```
|117 /* CanGo functions: return true if the barrel can move this way*/|
|118 /*                                                              */|
|119 /****************************************************************/|
|120                                                                 |
|121 bool Barrel::CanGoUp () const { // barrel can never go up         |
|122  return false;                                                  |
|123 }                                                               |
|124                                                                 |
|125 // a barrel can move over a ladder or money object              |
|126                                                                 |
|127 bool Barrel::CanGoDown () const {                                |
|128  if (grid[Down] == '.') return false;                            |
|129  char c = GetGridLoc (Down);                                    |
|130  if (c == ' ' || c == 'H' || c == '&') return true;             |
|131  return false;                                                  |
|132 }                                                               |
|133                                                                 |
|134 bool Barrel::CanGoRight () const {                               |
|135  char c = GetGridLoc (Right);                                   |
|136  if (c == ' ' || c == 'H' || c == '&') return true;             |
|137  return false;                                                  |
|138 }                                                               |
|139                                                                 |
|140 bool Barrel::CanGoLeft () const {                                |
|141  char c = GetGridLoc (Left);                                    |
|142  if (c == ' ' || c == 'H' || c == '&') return true;             |
|143  return false;                                                  |
|144 }                                                               |
|145                                                                 |
|146 /****************************************************************/|
|147 /*                                                              */|
|148 /* SetIsFalling: set falling barrel with no direction or cmd   */|
|149 /*                                                              */|
|150 /****************************************************************/|
|151                                                                 |
|152 void  Barrel::SetIsFalling () {                                  |
|153  isFalling = true;                                              |
|154  SetDirection (None);                                           |
|155  cmd = DoNothing;                                               |
|156 }                                                               |
|157                                                                 |
|158 /****************************************************************/|
|159 /*                                                              */|
|160 /* MoveDown: move the barrel down one location                  */|
|161 /*                                                              */|
|162 /****************************************************************/|
|163                                                                 |
|164 Point Barrel::MoveDown () {                                      |
|165  Point oldAt = at;                                              |
|166  at.row++;                                                      |
|167  char c = GetGridLoc (Down);                                    |
|168  if (c == ' ')                                                  |
```

```
|169   SetIsFalling ();                                                 |
|170  else                                                             |
|171   SetDirection (MovedDown);                                        |
|172  return oldAt;                                                    |
|173 }                                                                 |
|174                                                                   |
|175 /***************************************************************/|
|176 /*                                                             */|
|177 /* MoveRight: move barrel right, can begin to fall              */|
|178 /*                                                             */|
|179 /***************************************************************/|
|180                                                                   |
|181 Point Barrel::MoveRight () {                                      |
|182  Point oldAt = at;                                                |
|183  at.col++;                                                        |
|184  char c = GetGridLoc (Right);                                     |
|185  c = GetGridLoc (DownRight);                                      |
|186  if (c == ' ')                                                    |
|187   SetIsFalling ();                                                 |
|188  else                                                             |
|189   SetDirection (MovedRight);                                       |
|190  return oldAt;                                                    |
|191 }                                                                 |
|192                                                                   |
|193 /***************************************************************/|
|194 /*                                                             */|
|195 /* MoveLeft: move barrel left, can begin to fall                */|
|196 /*                                                             */|
|197 /***************************************************************/|
|198                                                                   |
|199 Point Barrel::MoveLeft () {                                       |
|200  Point oldAt = at;                                                |
|201  at.col--;                                                        |
|202  char c = GetGridLoc (Left);                                      |
|203  c = GetGridLoc (DownLeft);                                       |
|204  if (c == ' ')                                                    |
|205   SetIsFalling();                                                  |
|206  else                                                             |
|207   SetDirection (MovedLeft);                                        |
|208  return oldAt;                                                    |
|209 }                                                                 |
|210                                                                   |
|211 /***************************************************************/|
|212 /*                                                             */|
|213 /* FindGeneratorInitialLocation: get point where new barrels    */|
|214 /*                                                  appear     */|
|215 /***************************************************************/|
|216                                                                   |
|217 void Barrel::FindGeneratorInitialLocation (const Level& l) {      |
|218  bool gotgenerator = false;                                       |
|219  for (int i=0; i<l.GetNumRows(); i++) {                           |
|220   const char* line = l.GetRow (i);                                 |
```

```
221    for (int j=0; j<(int)strlen (line); j++) {
222     if (line[j] == 'V') { // found it!
223      generator.col = j;   // save its location
224      generator.row = i;
225      gotgenerator = true;
226      break;
227     }
228    }
229   }
230   if (!gotgenerator)
231    throw
232      "Error: cannot find the barrel generator starting location";
233
234   // find the bottom destruction points for the barrels
235   int row = l.GetNumRows () - 2;
236   // last row has to be ======
237   const char* grid = l.GetRow (row);
238   int cols = (int) strlen (grid);
239   bool gotleft = false;
240   bool gotright = false;
241   for (int i=0; i<cols; i++) {
242    if (grid[i] == '*') { // found the left edge of the game
243     if (!gotleft) {
244      leftDestroy = Point (row, i + 1);
245      gotleft = true;
246     }
247     else if (!gotright) { // found the right edge of the game
248      rightDestroy = Point (row, i - 1);
249      gotright = true;
250     }
251    }
252   }
253   if (!gotleft || !gotright)
254    throw "Error: cannot find the barrel's two ending locations";
255 }
```

In the constructor, notice that all barrel's initial location is just below the generator. Further, all barrels begin by falling down.

MoveBarrel() saves the current location of the barrel, removes any barrel that has hit the destruction point, and handles the movement of the barrel, whether going left, right, or falling. It returns the original location of the barrel. Notice then, that when displaying this barrel, besides just displaying the 'o' at the new location, first, any character that the barrel was hiding at the original location must be re-displayed. Then, any character that will become hidden at the new location is saved and the 'o' displayed.

To indicate this barrel needs to be destroyed, MoveBarrel() returns a special Point instance whose coordinates are 42 by 42, a location that cannot be in any level. Recall from

Object, that the grid array has to already been filled up with the level's characters in the nine surrounding locations and that the Location enum values are used as subscripts into the grid array. MoveBarrel() next checks to see if the barrel is currently falling and continues its action.

Falling is handled by obtaining the Down character from the grid. However, if there happens to be a money bag over the hole, the bag is replaced by a blank, a hole. Next, if the down character is not a hole, falling is halted and the direction to roll is reset, toggling the direction flag for use after the next fall. If there is a hole, the barrel's position is incremented down a row.

```
  if (isFalling) {
   char c = GetGridLoc (Down);
   if (c == '&') c = ' ';
   if (c != ' ') {
    isFalling = false;
    dir = afterFallGoRight ? MovedRight : MovedLeft;
   }
   else at.row++;
   return oldAt;
```

If the barrel is not falling, then the barrels always move in one direction until they fall again. Thus, we set the barrel's command to move left or right.

```
      cmd = (dir == MovedLeft) ? GoLeft : GoRight;
```

If the barrel is supposed to go right, then if it actually can go right, MoveRight() is called. However, if it has hit the side of the level, its direction is reversed and it is moved left. Left is handled similarly.

```
      if (cmd == GoRight) {
       if (CanGoRight ())
        oldAt = MoveRight ();
       else {
        dir = MovedLeft;
        oldAt = MoveLeft ();
       }
```

To actually move right, the original location is saved to be returned. The column is incremented. Next, the character in the level at this new location is checked for a hole and is falling is set if so. If there is no hole, the current direction of movement is set.

```
      Point oldAt = at;
      at.col++;
      char c = GetGridLoc (Right);
      c = GetGridLoc (DownRight);
      if (c == ' ')
       SetIsFalling ();
      else
       SetDirection (MovedRight);
      return oldAt;
```

Thus, handling a barrel's action is fairly simple. The player's, on the other hand, is more involved because of the jumping, climbing of ladders, and the finding of money bags.

The Player Class

The Player class adds new members to track the money and the number of remaining chances to complete the level. Also, three members deal with the jumping action.

isJumping is set to **true** when the player jumps. A jump depends on the direction of previous movement, if any. If the player was not moving previously, a jump goes one row up, dropping down the next turn. However, if the player was moving left or right, the jump goes one up to the left or right, and then one down continuing to the left or right in the next turn. During the jump, if the player hits the jump key again or a direction arrow that is consistent with the jump, a jump boost is applied, but only once. In the boost, the player goes one more row up, either vertically or vertically plus horizontally, if the player was moving. This boost enable the player to jump over a three-wide hole, for example, or over a couple barrels. Jumping is, then, handled in a series of turns with jumpPhase keeping track of the successive turns.

The Player Class Definition

```
 1 #pragma once
 2 #include "Object.h"
 3 #include "Level.h"
 4 #include "Screen.h"
 5
 6 // point to display the total money value
 7 const int MoneyRow = 23;
 8 const int MoneyCol = 2;
 9
10 // point to display the remaining chances value
11 const int ChancesRow = 23;
12 const int ChancesCol = 50;
13
14 /***************************************************************/
15 /*                                                             */
16 /* Player: encapsulates the player                             */
17 /*                                                             */
18 /***************************************************************/
19
20 class Player : public Object {
21 protected:
22  Screen& s;
23  Level& l;
24
25  int  chances;     // the number of chances to succeed remaining
26  int  money;       // accumulated money
27
```

```
28  int  maxRows;     // maximum number of rows in this level
29
30  bool isJumping;   // true when player is jumping
31  int  jumpPhase;   // the phase the jump is in 0 = start, 1, 2
32
33  bool doneExtraBoost; // allow one extra boost, doubling jump
34
35 public:
36   Player (Screen& ss, Level& ll);
37  ~Player ();
38
39  // obtain the player's initial starting location in the level
40  void  FindInitialPlayerLocation ();
41  void  InitialRenderPlayer ();
42  void  SetMaxRows (int m) { maxRows = m; }
43
44  // display player at this new location, removing it from old loc
45  void  Render (const Point& oldAt);
46
47  Point PlayerMove ();    // handle player's request to move
48
49  bool  CanGoUp () const;
50  bool  CanGoDown () const;
51  bool  CanGoRight () const;
52  bool  CanGoLeft () const;
53
54  // implement specific player move commands
55  Point MoveUp ();
56  Point MoveDown ();
57  Point MoveLeft ();
58  Point MoveRight ();
59
60  void  SetIsFalling ();  // set player is falling downwards now
61  Point HandleFalling (); // handle the falling senario
62
63  bool       IsJumping () const { return isJumping; }
64  void       StartJumping ();          // begin a jump command
65  Point      HandleJumping ();         // handle ongoing jump
66  JumpStatus ImplementJumpGoingUp ();  // handle going upwards
67  JumpStatus ImplementJumpGoingDown ();// handle coming down
68
69
70  bool  DecrementChances (); // lower chances by one
71  void  DisplayChances ();   // display chances remaining
72
73  // handle player finding money sacks
74  void  HandleMoney (int row, int col, Location loc);
75  void  AddMoney (int amount) { money += amount; }
76  void  DisplayMoney ();
77 };
```

As far as moving the player, the Game class calls either the player's HandleJumping(), HandleFalling() or PlayerMove() functions, depending upon whether the player is in the middle of falling, jumping or just moving. Hence, PlayerMove() does not have to deal with the complex jumping operation, simply attempting to move in one of the four directions. Note that CanGoUp() must check for the presence of a ladder, the 'H' code.

In all movements, the functions must check for the presence of the money bag. If found, HandleMoney() is called, which increments the money counter and displays the value onscreen.

Only the jumping functions really need further explanation. Examine the coding for the Player class and then let's focus on how jumping is handled.

The Player Class Implementation

```
 1 #include <iostream>
 2 #include <strstream>
 3 #include <iomanip>
 4 using namespace std;
 5 #include "Player.h"
 6 
 7 /***************************************************************/
 8 /*                                                             */
 9 /* Player: initialize the player object                        */
10 /*                                                             */
11 /***************************************************************/
12 
13 Player::Player (Screen& ss, Level& ll) : Object (), s(ss), l(ll){
14  isJumping = false;
15  chances = 3;       // player gets three chances
16  money = 0;
17  symbol = 'p';      // char to represent the player
18  maxRows = 0;
19  jumpPhase = 0;
20  doneExtraBoost = false;
21 }
22 
23 Player::~Player () {
24 }
25 
26 /***************************************************************/
27 /*                                                             */
28 /* FindInitialPlayerLocation: find player's starting point     */
29 /*                                                             */
30 /***************************************************************/
31 
32 void Player::FindInitialPlayerLocation () {
33  for (int i=0; i<l.GetNumRows(); i++) {
34   const char* line = l.GetRow (i);
35   for (int j=0; j<(int)strlen (line); j++) {
36    if (line[j] == 'p') { // found it
```

```
| 37     at.col = j;                                                  |
| 38     at.row = i;                                                  |
| 39     return;                                                      |
| 40    }                                                             |
| 41   }                                                              |
| 42  }                                                               |
| 43  throw "Error: cannot find the player's starting location";      |
| 44 }                                                                |
| 45                                                                  |
| 46 /***************************************************************/|
| 47 /*                                                             */|
| 48 /* InitialRenderPlayer: display player at starting location    */|
| 49 /*                                                             */|
| 50 /***************************************************************/|
| 51                                                                  |
| 52 void  Player::InitialRenderPlayer () {                           |
| 53  s.OutputUCharWith ('p', at.row, at.col, Screen::Blue,           |
| 54                     Screen::BrightYellow);                       |
| 55 }                                                                |
| 56                                                                  |
| 57 /***************************************************************/|
| 58 /*                                                             */|
| 59 /* Render: remove player from old loc and show at new loc      */|
| 60 /*                                                             */|
| 61 /***************************************************************/|
| 62                                                                  |
| 63 void  Player::Render (const Point& oldAt) {                      |
| 64  if (GetSavedSymbol() != 'p') // was hiding something?           |
| 65   s.OutputUCharWith (GetSavedSymbol(), oldAt.row, oldAt.col,     |
| 66                      Screen::Blue, Screen::BrightYellow);        |
| 67  // save what the player will now be hiding                      |
| 68  SetSavedSymbol (l.GetChar (at.row, at.col));                    |
| 69  // show player char                                             |
| 70  s.OutputUCharWith (symbol, at.row, at.col, Screen::Blue,        |
| 71                     Screen::BrightYellow);                       |
| 72 }                                                                |
| 73                                                                  |
| 74 /***************************************************************/|
| 75 /*                                                             */|
| 76 /* PlayerMove: handle a player's move request                  */|
| 77 /*              Note: cannot get here if falling or jumping    */|
| 78 /*                                                             */|
| 79 /***************************************************************/|
| 80                                                                  |
| 81 Point Player::PlayerMove () {                                    |
| 82  Point oldAt = at;                                               |
| 83  if (cmd == GoUp && CanGoUp ())                                  |
| 84   oldAt = MoveUp ();                                             |
| 85  else if (cmd == GoDown && CanGoDown ())                         |
| 86   oldAt = MoveDown ();                                           |
| 87  else if (cmd == GoRight && CanGoRight ())                       |
| 88   oldAt = MoveRight ();                                          |
```

```
| 89  else if (cmd == GoLeft && CanGoLeft ())                          |
| 90   oldAt == MoveLeft ();                                           |
| 91  else if (cmd == Jump)                                            |
| 92   StartJumping ();                                                |
| 93  return oldAt;                                                    |
| 94 }                                                                 |
| 95                                                                   |
| 96 /**************************************************************/|
| 97 /*                                                            */|
| 98 /* CanGo functions: returns true if player can move this way   */|
| 99 /*                                                            */|
|100 /**************************************************************/|
|101                                                                   |
|102 bool Player::CanGoUp () const {                                  |
|103  if (grid[Up] == '.') return false;                              |
|104  char c = GetGridLoc (At);                                        |
|105  // a ladder is the only way upwards                              |
|106  return c == 'H' ? true: false;                                   |
|107 }                                                                 |
|108                                                                   |
|109 bool Player::CanGoDown () const {                                |
|110  if (grid[Down] == '.') return false;                            |
|111  char c = GetGridLoc (At);                                        |
|112  char cd = GetGridLoc (Down);                                     |
|113  // if on a ladder rung, it may be possible to go down            |
|114  if (c == 'H' && (cd == ' ' || cd == 'H' || cd == '&'))          |
|115   return true;                                                    |
|116  if (c != '=' && (cd == ' ' || cd == 'H'))                        |
|117   return true;                                                    |
|118  return false;                                                    |
|119 }                                                                 |
|120                                                                   |
|121 bool Player::CanGoRight () const {                               |
|122  if (grid[Right] == '.' || grid[Right] == '*')                   |
|123   return false;                                                   |
|124  char c = GetGridLoc (Right);                                     |
|125  if (c == ' ' || c == 'H' || c == '&')                            |
|126   return true;                                                    |
|127  return false;                                                    |
|128 }                                                                 |
|129                                                                   |
|130 bool Player::CanGoLeft () const {                                |
|131  if (grid[Left] == '.' || grid[Left] == '*') return false;       |
|132  char c = GetGridLoc (Left);                                      |
|133  if (c == ' ' || c == 'H' || c == '&') return true;               |
|134  return false;                                                    |
|135 }                                                                 |
|136                                                                   |
|137 /**************************************************************/|
|138 /*                                                            */|
|139 /* SetIsFalling: signal the player is now falling             */|
|140 /*                                                            */|
```

```
|141 /***************************************************************/|
|142                                                                  |
|143 void  Player::SetIsFalling (){                                   |
|144  isFalling = true;                                               |
|145  SetDirection (None); // clear direction of motion and cmds      |
|146  cmd = DoNothing;                                                |
|147 }                                                                |
|148                                                                  |
|149 /***************************************************************/|
|150 /*                                                             */|
|151 /* HandleFalling: handle an ongoing falling action             */|
|152 /*                                                             */|
|153 /***************************************************************/|
|154                                                                  |
|155 Point Player::HandleFalling () {                                 |
|156  Point oldAt = at;                                               |
|157  if (isFalling) {                                                |
|158   char c = GetGridLoc (Down);                                    |
|159   if (c == '&') { // catchs a money bag as he goes down          |
|160    HandleMoney (at.row-1, at.col, Down);                         |
|161    c = ' ';                                                      |
|162   }                                                              |
|163   if (c != ' ') // hit something, so stop falling                |
|164    isFalling = false;                                            |
|165   else at.row++;                                                 |
|166  }                                                               |
|167  return oldAt;                                                   |
|168 }                                                                |
|169                                                                  |
|170 /***************************************************************/|
|171 /*                                                             */|
|172 /* MoveUp: handle moving upwards                               */|
|173 /*                                                             */|
|174 /***************************************************************/|
|175                                                                  |
|176 Point Player::MoveUp () {                                        |
|177  Point oldAt = at;                                               |
|178  at.row--;                                                       |
|179  char c = GetGridLoc (Up);                                       |
|180  if (c == '&')                                                   |
|181   HandleMoney (at.row, at.col, Up);                              |
|182  SetDirection (MovedUp);                                         |
|183  return oldAt;                                                   |
|184 }                                                                |
|185                                                                  |
|186 /***************************************************************/|
|187 /*                                                             */|
|188 /* MoveDown: handle moving downward                            */|
|189 /*                                                             */|
|190 /***************************************************************/|
|191                                                                  |
|192 Point Player::MoveDown () {                                      |
```

```
|193  Point oldAt = at;                                              |
|194  at.row++;                                                      |
|195  char c = GetGridLoc (Down);                                    |
|196  if (c == '&')                                                  |
|197   HandleMoney (at.row, at.col, Down);                           |
|198  if (c == ' ') // fell through an opening, now falling          |
|199   SetIsFalling ();                                             |
|200  else SetDirection (MovedDown);                                 |
|201  return oldAt;                                                  |
|202 }                                                               |
|203                                                                 |
|204 /***************************************************************/|
|205 /*                                                             */|
|206 /* MoveRight: handle moving right                              */|
|207 /*                                                             */|
|208 /***************************************************************/|
|209                                                                 |
|210 Point Player::MoveRight () {                                    |
|211  Point oldAt = at;                                              |
|212  at.col++;                                                      |
|213  char c = GetGridLoc (Right);                                   |
|214  if (c == '&') // found a money bag                             |
|215   HandleMoney (at.row, at.col, Right);                          |
|216  c = GetGridLoc (DownRight);                                    |
|217  if (c == ' ') // fell through a hole, now is falling          |
|218   SetIsFalling ();                                             |
|219  else                                                           |
|220   SetDirection (MovedRight);                                   |
|221  return oldAt;                                                  |
|222 }                                                               |
|223                                                                 |
|224 /***************************************************************/|
|225 /*                                                             */|
|226 /* MoveLeft: handle moving left                                */|
|227 /*                                                             */|
|228 /***************************************************************/|
|229                                                                 |
|230 Point Player::MoveLeft () {                                     |
|231  Point oldAt = at;                                              |
|232  at.col--;                                                      |
|233  char c = GetGridLoc (Left);                                    |
|234  if (c == '&') // found a money bag                             |
|235   HandleMoney (at.row, at.col, Left);                           |
|236  c = GetGridLoc (DownLeft);                                     |
|237  if (c == ' ') // fell through a hole, so now is falling       |
|238   SetIsFalling();                                              |
|239  else                                                           |
|240   SetDirection (MovedLeft);                                    |
|241  return oldAt;                                                  |
|242 }                                                               |
|243                                                                 |
|244 /***************************************************************/|
```

```
|245 /*                                                          */|
|246 /* StartJumping: begin a jumping operation                  */|
|247 /*                                                          */|
|248 /************************************************************/|
|249                                                              |
|250 void Player::StartJumping () {                               |
|251  if (isJumping) return;                                      |
|252  if (at.row -1 < 0) return; // at top, cannot go up further  |
|253  jumpPhase = 1;             // set is in the 1st phase       |
|254  doneExtraBoost = false;    // allow one boost action        |
|255                                                              |
|256  // handle going up one location                             |
|257  JumpStatus j = ImplementJumpGoingUp ();                     |
|258  if (j == Okay) {                                            |
|259   isJumping = true;                                          |
|260   cmd = DoNothing;                                           |
|261  }                                                           |
|262  else if (j == IllegalMove || j == CaughtLadder)             |
|263   isJumping = false; // jumping is ended                     |
|264 }                                                            |
|265                                                              |
|266 /************************************************************/|
|267 /*                                                          */|
|268 /* HandleJumping: handle an ongoing jump action             */|
|269 /*                                                          */|
|270 /************************************************************/|
|271                                                              |
|272 Point Player::HandleJumping () {                             |
|273  Point oldAt = at;                                           |
|274  // two situations                                           |
|275  // 1: no additional commands, so finish the jump            |
|276  // 2: an additional command, so double all actions and permit no|
|277  //    additional commands til jump is done.                  |
|278  jumpPhase++;                                                |
|279  JumpStatus j;                                               |
|280  if (jumpPhase == 2) {                                       |
|281   if (cmd == Jump || (dir == MovedLeft && cmd == GoLeft) ||  |
|282       (dir == MovedRight && cmd == GoRight) ||               |
|283       (dir == MovedUp && cmd == GoUp)) {                     |
|284    // double jump parms                                      |
|285    doneExtraBoost = true;                                    |
|286    JumpStatus j = ImplementJumpGoingUp ();                   |
|287    if (j == CaughtLadder)                                    |
|288     isJumping = false;                                       |
|289    else if (j == IllegalMove) {                              |
|290     dir = None;                                              |
|291     j = ImplementJumpGoingDown ();                           |
|292     isJumping = false;                                       |
|293     if (j == Falling)                                        |
|294      isFalling = true;                                       |
|295    }                                                         |
|296   }                                                          |
```

```
|297   else {                                                            |
|298    j = ImplementJumpGoingDown ();                                   |
|299    isJumping = false;                                               |
|300    if (j == Falling)                                                |
|301     isFalling = true;                                               |
|302   }                                                                 |
|303  }                                                                  |
|304  else if (jumpPhase == 3) {                                         |
|305   j = ImplementJumpGoingDown ();                                    |
|306   if (j == CaughtLadder)                                            |
|307    isJumping = false;                                               |
|308  }                                                                  |
|309  else {                                                             |
|310   j = ImplementJumpGoingDown ();                                    |
|311   isJumping = false;                                                |
|312   if (j == Falling)                                                 |
|313    isFalling = true;                                                |
|314  }                                                                  |
|315  return oldAt;                                                      |
|316 }                                                                  |
|317                                                                    |
|318 /***************************************************************/|
|319 /*                                                             */|
|320 /* ImplementJumpGoingUp: handle jump while going upwards       */|
|321 /*                                                             */|
|322 /***************************************************************/|
|323                                                                    |
|324 JumpStatus Player::ImplementJumpGoingUp () {                       |
|325  char cUp = GetGridLoc (Up);                                       |
|326  if (cUp == '.') {                                                 |
|327   dir = None;                                                      |
|328   return IllegalMove;                                              |
|329  }                                                                 |
|330  if (cUp == 'H') {                                                 |
|331   // caught ladder instead - stop jump                             |
|332   at.row--;                                                        |
|333   return CaughtLadder;                                             |
|334  }                                                                 |
|335  if (cUp == '&')                                                   |
|336   HandleMoney (at.row - 1, at.col, Up);                            |
|337  if (dir == MovedLeft) {                                           |
|338   cUp = GetGridLoc (UpLeft);                                       |
|339   if (cUp == '.') {                                                |
|340    dir = None;                                                     |
|341    return IllegalMove;                                             |
|342   }                                                                |
|343   if (cUp == 'H') {                                                |
|344    // caught ladder - stop jump                                    |
|345    at.col--;                                                       |
|346    at.row--;                                                       |
|347    return CaughtLadder;                                            |
|348   }                                                                |
```

```
|349   if (cUp == '&')                                                |
|350    HandleMoney (at.row - 1, at.col - 1, UpLeft);                 |
|351   if (cUp != '*') {                                              |
|352    at = Point (at.row-1, at.col-1);                              |
|353    return Okay;                                                  |
|354   }                                                              |
|355  }                                                               |
|356  else if (dir == MovedRight) {                                   |
|357   cUp = GetGridLoc (UpRight);                                    |
|358   if (cUp == '.') {                                              |
|359    dir = None;                                                   |
|360    return IllegalMove;                                           |
|361   }                                                              |
|362   if (cUp == 'H') {                                              |
|363    // caught ladder - stop jump                                  |
|364    at.col++;                                                     |
|365    at.row--;                                                     |
|366    return CaughtLadder;                                          |
|367   }                                                              |
|368   if (cUp == '&')                                                |
|369    HandleMoney (at.row - 1, at.col + 1, UpRight);                |
|370   if (cUp != '*') {                                              |
|371    at = Point (at.row-1, at.col+1);                              |
|372    return Okay;                                                  |
|373   }                                                              |
|374  }                                                               |
|375  else { // going straight up                                     |
|376   cUp = GetGridLoc (Up);                                         |
|377   if (cUp == '.') {                                              |
|378    dir = None;                                                   |
|379    return IllegalMove;                                           |
|380   }                                                              |
|381   if (cUp == 'H') {                                              |
|382    // caught ladder - stop jump                                  |
|383    at.row--;                                                     |
|384    return CaughtLadder;                                          |
|385   }                                                              |
|386   at.row--;                                                      |
|387   if (cUp == '&')                                                |
|388    HandleMoney (at.row, at.col, Up);                             |
|389   return Okay;                                                   |
|390  }                                                               |
|391  dir = None;                                                     |
|392  return IllegalMove;                                             |
|393 }                                                                |
|394                                                                  |
|395 /***************************************************************/|
|396 /*                                                             */|
|397 /* ImplementJumpGoingDown: handle ongoing jump going downwards */|
|398 /*                                                             */|
|399 /***************************************************************/|
|400                                                                  |
```

```
|401 JumpStatus Player::ImplementJumpGoingDown () {                      |
|402  Location loc;                                                       |
|403  char c;                                                             |
|404  at.row++;                                                          |
|405  if (dir == MovedLeft) {                                            |
|406   at.col--;                                                         |
|407   c = GetGridLoc (DownLeft);                                        |
|408   loc = DownLeft;                                                   |
|409   if (at.col < 0) {                                                 |
|410    at.col = 0;                                                      |
|411    c = GetGridLoc (Down);                                           |
|412    loc = Down;                                                      |
|413   }                                                                 |
|414  }                                                                  |
|415  else if (dir == MovedRight) {                                      |
|416   at.col++;                                                         |
|417   c = GetGridLoc (DownRight);                                       |
|418   loc = DownRight;                                                  |
|419   if (at.col >= 80) {                                               |
|420    at.col--;                                                        |
|421    c = GetGridLoc (Down);                                           |
|422    loc = Down;                                                      |
|423   }                                                                 |
|424  }                                                                  |
|425  else {                                                             |
|426   c = GetGridLoc (Down);                                            |
|427   loc = Down;                                                       |
|428  }                                                                  |
|429  if (c == 'H')                                                      |
|430   return CaughtLadder;                                              |
|431  else if (c == ' ')                                                 |
|432   return Falling;                                                   |
|433  else if (c == '&')                                                 |
|434   HandleMoney (at.row, at.col, loc);                                |
|435  return Okay;                                                       |
|436 }                                                                   |
|437                                                                     |
|438 /**************************************************************/|
|439 /*                                                            */|
|440 /* HandleMoney: handle found a new money bag                  */|
|441 /*                                                            */|
|442 /**************************************************************/|
|443                                                                     |
|444 void Player::HandleMoney (int row, int col, Location loc) {        |
|445  money += 100;                                                      |
|446  l.ReplaceChar (row, col, ' '); // remove money bag from screen    |
|447  s.OutputUCharWith (' ', row, col, Screen::Blue,                    |
|448                     Screen::BrightYellow);                          |
|449  saveSymbol = ' ';                                                  |
|450  grid[loc] = ' ';                                                   |
|451  DisplayMoney ();                                                   |
|452 }                                                                   |
```

```
|453                                                                        |
|454 /**************************************************************/|
|455 /*                                                            */|
|456 /* DisplayMoney: display current total money                  */|
|457 /*                                                            */|
|458 /**************************************************************/|
|459                                                                        |
|460 void Player::DisplayMoney () {                                         |
|461  char msg[80];                                                         |
|462  ostrstream os (msg, sizeof (msg));                                    |
|463  os << "Money: " << setw (5) << money << ends;                         |
|464  s.OutputAt (msg, MoneyRow, MoneyCol);                                 |
|465 }                                                                      |
|466                                                                        |
|467 /**************************************************************/|
|468 /*                                                            */|
|469 /* DecrementChances: lower chances remaining by one           */|
|470 /*                                                            */|
|471 /**************************************************************/|
|472                                                                        |
|473 bool Player::DecrementChances () {                                     |
|474  chances--;                                                            |
|475  return chances == 0;                                                  |
|476 }                                                                      |
|477                                                                        |
|478 /**************************************************************/|
|479 /*                                                            */|
|480 /* DisplayChances: display chances remaining on screen        */|
|481 /*                                                            */|
|482 /**************************************************************/|
|483                                                                        |
|484 void Player::DisplayChances () {                                       |
|485  char msg[80];                                                         |
|486  ostrstream os (msg, sizeof (msg));                                    |
|487  os << "Chances Remaining: " << setw (5) << chances << ends;           |
|488  s.OutputAt (msg, ChancesRow, ChancesCol);                             |
|489 }                                                                      |
```

Okay, ready for jumping? Four functions are involved: StartJumping(), HandelJumping(), ImplementJumpGoingUp(), and ImplementJumpGoingDown (). In StartJumping(), if this is a valid jump, jumpPhase is set to 1 and doneExtraBoost is set to false. Then ImplementJumpGoingUp() is called, which returns an instance of the enum JumpStatus, indicating the results of the attempt. If the status is Okay, then isJumping is set to true and the command is set to DoNothing, that is a jump is in progress. If it was an illegal attempt, the jump is aborted.

```
void Player::StartJumping () {
 if (isJumping) return;
 if (at.row -1 < 0) return; // at top, cannot go up further
 jumpPhase = 1;             // set is in the 1st phase
```

```
   doneExtraBoost = false;    // allow one boost action
   JumpStatus j = ImplementJumpGoingUp ();
   if (j == Okay) {
    isJumping = true;
    cmd = DoNothing;
   }
   else if (j == IllegalMove || j == CaughtLadder)
    isJumping = false; // jumping is ended
  }
```

HandleJumping() handles an ongoing jump action. It must deal with two situations: a basic jump and a boost given during a jump, which you might call an extended jump. The member jumpPhase keeps track of how many turns the jump has lasted. StartJumping() took one turn, so jumpPhase is incremented right away. If this is to be a simple one turn jump, then the phase number will be 2. If during phase 2 the player has issued another jump or move command, the boost is implemented, subject to only one boost per jump.

```
Point Player::HandleJumping () {
 Point oldAt = at;
 jumpPhase++;
 JumpStatus j;
 if (jumpPhase == 2) {
  if (cmd == Jump || (dir == MovedLeft && cmd == GoLeft) ||
      (dir == MovedRight && cmd == GoRight) ||
      (dir == MovedUp && cmd == GoUp)) {
   // double jump parms
```

If a boost has been detected, then the bool is set to true to avoid multiple boosts during this jump. To implement the boost, ImplementJumpGoingUp() is called once more, which moves the player up another row, if possible. Doing so, the player might catch onto a ladder, and if so, the jump is ended. However, if the attempt to go up higher fails, then the boost fails and ImplementJumpGoingDown() is called which will end this jump.

```
   doneExtraBoost = true;
   JumpStatus j = ImplementJumpGoingUp ();
   if (j == CaughtLadder)
    isJumping = false;
   else if (j == IllegalMove) {
    dir = None;
    j = ImplementJumpGoingDown ();
    isJumping = false;
    if (j == Falling)
     isFalling = true;
   }
  }
```

If a boost was not detected here in phase 2, then the jump must be ended with a call to ImplementJumpGoingDown(), which returns the downward movement's status. If the player

lands on a hole, isFalling is turned on.

```
        else {
         j = ImplementJumpGoingDown ();
         isJumping = false;
         if (j == Falling)
          isFalling = true;
        }
       }
```

If this is now phase 3, then the player is on the down portion of the jump, which might last one or two turns, depending upon whether or not a boost was done. In all cases, ImplementJumpGoingDown() is called, which returns the status of the downward movement. It is possible to catch a ladder on the downward path, and if so, the jump ends with the player on the ladder.

```
       else if (jumpPhase == 3) {
        j = ImplementJumpGoingDown ();
        if (j == CaughtLadder)
         isJumping = false;
       }
```

In all other cases, the downward movement is continued and the jump is ended with the player possibly landing on a hole.

```
       else {
        j = ImplementJumpGoingDown ();
        isJumping = false;
        if (j == Falling)
         isFalling = true;
       }
       return oldAt;
      }
```

Two workhorse functions deal with the upward and downward motion. The basic idea here is to check all of the moved through locations for illegal moves, such as moving out of the level, finding a money bag, landing on a ladder. If a location is illegal to move into, the grid has a '.' code in it, placed there by the calling functions in the Game class. Thus, these two lengthy functions really do not contain any tricky coding.

The Game Class

The Game class ties all of the pieces together and runs the game until it is finished. It stores the player's current location so that the many barrels can be checked to see if they hit the player. It stores the barrels in a growable array of pointers to the barrel objects, since the barrels come and go as time moves along in the game.

It stores the two parameters, gameSpeed and maxBarrels, which are adjustable by the user at the start of the game. In a real game, these would be initialized by the level itself and not under the user's control. By making them user adjustable, you can experiment with these and see their impact upon game playability. The member newBarrelTime is actually also an adjustable parameter, but I chose not to make this one adjustable by the user.

The **bool**s done and win control the game. The main Run function continues with turn after turn until the game is either aborted or the player wins or loses.

The member array grid stores the level chars immediately surrounding the current position of both player and barrels, avoiding the overhead of 2-d array look ups.

The Game Class Definition

```
 1 #pragma once
 2 #include <iostream>
 3 #include <fstream>
 4 using namespace std;
 5
 6 #include "screen.h"
 7 #include "level.h"
 8 #include "Point.h"
 9 #include "Player.h"
10 #include "Barrel.h"
11 #include "Array.h"
12 #include "KeyBoard.h"
13
14 /***************************************************************/
15 /*                                                             */
16 /* Game: encapsulates the Ladder game                          */
17 /*                                                             */
18 /***************************************************************/
19
20 class Game {
21 protected:
22  Screen& s;         // the screen
23  KeyBoard kb;       // the realtime keyboard
24  Level  l;          // the level loaded from a file
25  Player p;          // the player
26  Point  playerAt;   // player's current location
27
```

```
28  int maxRows;       // maximum number of rows in this level
29
30  Array barrels;     // the array of barrels currently in existence
31
32  int gameSpeed;     // parm for game speed, in milliseconds
33  int maxBarrels;    // max number of barrels on screen at one time
34
35  bool done;         // when true, the level is done
36  bool win;          // when true, the player won
37
38  char grid[MaxAround];  // the level chars surrounding this loc
39
40  int  newBarrelTime;    // counts turns between new barrels
41  bool afterFallGoRight; // toggel - left - right move after fall
42
43  Point oldAt;        // the old location
44  Point pAt;          // the player's current location
45  Barrel* ptrb;       // working pointer to get at items in array
46
47 public:
48   Game (const char* filename, Screen& s);
49  ~Game ();
50
51  void GetInitialParamters (); // get user's speed and max barrels
52
53  void Run ();             // plays the game, until done
54  void HandlePlayer ();    // handles a player's actions
55  void HandleBarrels ();   // handles all barrel movements
56  void MakeNewBarrels (); // makes new barrels when needed
57
58  bool GetNextCommand (); // gets the next player command
59
60  // loads the level chars surrounding this location
61  void FillGrid (char grid[], const Point& p);
62
63  void Wait (); // pause game until player hits another command
64 };
```

The main function first calls GetInitialParameters() to get the game speed and the maximum number of barrels. As usual, main then calls the Run() function to play the game until it is done. Run() calls GetNextCommand() and then calls HandlePlayer(), HandleBarrels(), and MakeNewBarrels(). Before each of these, it calls FillGrid() based on the object's current location. Finally, should the ESC key be pressed to pause the game action, Wait() waits until the player presses another key.

Note FillGrid() stores a '.' character in any location that is illegal, such as a row or column being out of range.

```
The Game Class Implementation
```

```
 1 #include "Game.h"
 2 #include <ctime>
 3
 4 /***************************************************************/
 5 /*                                                             */
 6 /* Game: load the game level, preparing for first turn         */
 7 /*                                                             */
 8 /***************************************************************/
 9
10 Game::Game (const char* filename, Screen& ss) : s(ss), p(ss, l) {
11  try {
12   GetInitialParamters (); // get user's game speed & max barrels
13
14   l.LoadLevel (filename); // load level from a file
15
16   p.FindInitialPlayerLocation (); // find player's initial loc
17   playerAt = p.GetLocation();     // and set it
18   l.RemoveInitialP (playerAt);    // remove 'p' from level
19   maxRows = l.GetNumRows ();      // save max rows in level
20   p.SetMaxRows (maxRows);
21
22   l.Render (s);                   // display the entire level
23   p.InitialRenderPlayer ();       // show player at initial loc
24   p.DisplayMoney ();              // display initial money
25   p.DisplayChances ();            // display chances remaining
26
27   // find the point from which new barrels appears
28   Barrel::FindGeneratorInitialLocation (l);
29  }
30  catch (const char* msg) {
31   throw msg; // failed, so pass error msg on to main
32  }
33
34  done = false;              // set game in progress
35  win = false;               // set player has lost
36  newBarrelTime = 0;         // initialize new barrel counter
37  afterFallGoRight = false;  // init barrel goes switch
38 }
39
40 /***************************************************************/
41 /*                                                             */
42 /* ~Game: delete any remaining barrels in the array            */
43 /*                                                             */
44 /***************************************************************/
45
46 Game::~Game(void) {
47  Barrel* ptrb;
48  for (int i=barrels.GetSize() -1; i>=0; i--) {
49   ptrb = (Barrel*) barrels.GetAt (i);
50   if (ptrb) delete ptrb;
51  }
```

```
| 52  barrels.RemoveAll ();                                            |
| 53 }                                                                 |
| 54                                                                   |
| 55 /***************************************************************/|
| 56 /*                                                             */|
| 57 /* GetInitialParamters: get speed and max barrels, show hints  */|
| 58 /*                                                             */|
| 59 /***************************************************************/|
| 60                                                                   |
| 61 void Game::GetInitialParamters () {                               |
| 62  cout << "Welcome to the Ladder game.\n\n"                        |
| 63       << "Enter game speed (100 is fast, 500 is slow): ";         |
| 64  cin >> gameSpeed;                                                |
| 65  if (!cin) throw "Error bad data entered.\n";                     |
| 66  cout <<                                                          |
| 67     "\nEnter the maximum number of barrels in play at one time\n"|
| 68       << "(5 is easy, 15 is hard): ";                             |
| 69  cin >> maxBarrels;                                               |
| 70  if (!cin) throw "Error bad data entered.\n";                     |
| 71  cout << "\n\nMovement: use arrow keys\nJump: press space bar\n" |
| 72       << "Pause game: press Esc key\n"                            |
| 73       << "To abort game, press Ctrl-C\n"                          |
| 74       << "\n\nPress any key to begin";                            |
| 75  s.GetAnyKey ();                                                  |
| 76 }                                                                 |
| 77                                                                   |
| 78 /***************************************************************/|
| 79 /*                                                             */|
| 80 /* FillGrid: fill grid with level chars surrounding this loc   */|
| 81 /*                                                             */|
| 82 /***************************************************************/|
| 83                                                                   |
| 84 void Game::FillGrid (char grid[], const Point& p) {               |
| 85  grid[At] = l.GetChar (p.row, p.col);                             |
| 86  grid[Right] = p.col+1 >= 80 ? '.' : l.GetChar (p.row, p.col+1); |
| 87  grid[Left] = p.col-1 < 0 ? '.' : l.GetChar (p.row, p.col-1);    |
| 88  grid[Up] = p.row-1 < 0 ? '.' : l.GetChar (p.row-1, p.col);      |
| 89  grid[UpRight] = p.row-1 <0 ? '.' : (p.col+1 >= 80 ? '.' :       |
| 90                  l.GetChar (p.row-1, p.col+1));                   |
| 91  grid[UpLeft] = p.row-1 < 0 ? '.' : (p.col-1 < 0 ? '.' :         |
| 92                 l.GetChar (p.row-1, p.col-1));                    |
| 93  grid[Down] = p.row+1 >= maxRows ? '.' :                          |
| 94               l.GetChar (p.row+1, p.col);                         |
| 95  grid[DownRight] = p.row+1 >= maxRows ? '.' :                     |
| 96             (p.col+1 >= 80 ? '.' : l.GetChar (p.row+1, p.col+1));|
| 97  grid[DownLeft] = p.row+1 >= maxRows ? '.' :                      |
| 98               (p.col-1 < 0 ? '.' : l.GetChar (p.row+1, p.col-1));|
| 99 }                                                                 |
|100                                                                   |
|101 /***************************************************************/|
|102 /*                                                             */|
|103 /* Run: plays the game until player wins or loses              */|
```

```
|104 /*                                                          */|
|105 /************************************************************/|
|106                                                               |
|107 void Game::Run () {                                           |
|108  time_t startTime;            // clock time at start of turn   |
|109  time_t endTime;              // the time at the end of turn   |
|110  while (!done) {              // repeat until game is over     |
|111   startTime = time (0);       // get turn starting time        |
|112   HandlePlayer ();            // handle player's action        |
|113   HandleBarrels ();           // move all barrels              |
|114   MakeNewBarrels ();          // make new barrels as needed    |
|115   endTime = time (0);         // get the turn ending time      |
|116                                                               |
|117   // calculate how long we need to wait for the next turn      |
|118   long waitTime = gameSpeed- (long) (endTime - startTime);     |
|119                                                               |
|120   // wait as needed before starting next turn                  |
|121   if (waitTime > 0) Sleep (waitTime);                          |
|122  }                                                            |
|123  // here, display game over messages                           |
|124  const char* msg = win ?                                       |
|125   "You completed this level and can go on" :                   |
|126   "You have failed this level. Try again ";                    |
|127  s.OutputAt (msg, 22, 2);                                      |
|128  s.FlashArea (22, 2, (int) strlen (msg));                      |
|129 }                                                             |
|130                                                               |
|131 /************************************************************/|
|132 /*                                                          */|
|133 /* HandlePlayer: handle player's action                     */|
|134 /*                                                          */|
|135 /************************************************************/|
|136                                                               |
|137 void Game::HandlePlayer () {                                  |
|138  FillGrid (grid, p.GetLocation ()); // fill grid around player |
|139  p.SetGrid (grid);                  // install grid into player|
|140  oldAt = p.GetLocation ();          // save current location   |
|141  if (p.GetIsFalling ()) {           // handle falling player   |
|142   oldAt = p.HandleFalling ();                                  |
|143   p.SetCommand (DoNothing);                                    |
|144   p.Render (oldAt);                                            |
|145  }                                                            |
|146  else if (p.IsJumping ()) {         // handle jumping player   |
|147   if (!GetNextCommand ()) p.SetCommand (DoNothing);            |
|148   oldAt = p.HandleJumping ();                                  |
|149   p.Render (oldAt);                                            |
|150  }                                                            |
|151  else { // here, not jumping or falling, so get a command      |
|152   if (GetNextCommand ()) { // if key stroke entered, handle    |
|153    oldAt = p.PlayerMove ();                                    |
|154    p.Render (oldAt);                                           |
|155   }                                                            |
```

```
|156  }                                                              |
|157  pAt = p.GetLocation ();             // get new location         |
|158  // check to see if player is at the ending point                |
|159  char c = l.GetChar (pAt.row, pAt.col);                          |
|160  if (c == '$') {                                                  |
|161   p.AddMoney (3000);                                              |
|162   p.DisplayMoney ();                                              |
|163   done = true;                                                    |
|164   win = true;                                                     |
|165  }                                                                |
|166 }                                                                 |
|167                                                                   |
|168 /**************************************************************/|
|169 /*                                                            */|
|170 /* HandleBarrels: move each barrel in the array               */|
|171 /*                                                            */|
|172 /**************************************************************/|
|173                                                                   |
|174 void Game::HandleBarrels () {                                     |
|175  Point oldP_At;                                                   |
|176  int i = 0;                                                       |
|177  while (i<barrels.GetSize()) {                                    |
|178   ptrb = (Barrel*) barrels.GetAt (i);                             |
|179                                                                   |
|180   // fill grid surrounding this location                          |
|181   FillGrid (grid, ptrb->GetLocation ());                          |
|182   ptrb->SetGrid (grid);                                           |
|183                                                                   |
|184   // see if barrel hits the player                                |
|185   if (ptrb->HitPlayer (pAt)) {                                    |
|186    if (p.DecrementChances ())                                     |
|187     done = true;                                                  |
|188    p.DisplayChances ();                                           |
|189    oldP_At = pAt;                                                 |
|190    p.SetLocation (playerAt);                                      |
|191    p.Render (oldP_At);                                            |
|192   }                                                               |
|193                                                                   |
|194   // move barrel                                                  |
|195   oldAt = ptrb->MoveBarrel ();                                    |
|196                                                                   |
|197   // see if barrel hits player                                    |
|198   if (ptrb->HitPlayer (pAt)) {                                    |
|199    if (p.DecrementChances ())                                     |
|200     done = true;                                                  |
|201    p.DisplayChances ();                                           |
|202    oldP_At = pAt;                                                 |
|203    p.SetLocation (playerAt);                                      |
|204    p.Render (oldP_At);                                            |
|205   }                                                               |
|206                                                                   |
|207   // see if barrel should now be destroyed                        |
```

```
|208   if (oldAt == Point (42, 42)) {                               |
|209    ptrb->FinalRender ();                                       |
|210    delete ptrb;                                                |
|211    barrels.RemoveAt (i);                                       |
|212   }                                                            |
|213   else {                                                       |
|214    ptrb->Render (oldAt);                                       |
|215    i++;                                                        |
|216   }                                                            |
|217  }                                                             |
|218 }                                                              |
|219                                                                |
|220 /************************************************************/|
|221 /*                                                          */|
|222 /* MakeNewBarrels: make new barrel just below the generator */|
|223 /*                                                          */|
|224 /************************************************************/|
|225                                                                |
|226 void Game::MakeNewBarrels () {                                 |
|227  newBarrelTime++;                                              |
|228  if (newBarrelTime >= 15 && barrels.GetSize() < maxBarrels) {  |
|229   newBarrelTime = 0;                                           |
|230   ptrb = new Barrel (s, l);                                    |
|231   ptrb->SetAfterFallGoRight (afterFallGoRight);                |
|232   afterFallGoRight = afterFallGoRight ? false : true;          |
|233   barrels.Add (ptrb);                                          |
|234   ptrb->InitialRender ();                                      |
|235  }                                                             |
|236 }                                                              |
|237                                                                |
|238 /************************************************************/|
|239 /*                                                          */|
|240 /* GetNextCommand: gets the next player's command           */|
|241 /*                                                          */|
|242 /************************************************************/|
|243                                                                |
|244 bool Game::GetNextCommand () {                                 |
|245  // if no key is pressed, quit                                 |
|246  if (!kb.Peak ()) return false;                                |
|247                                                                |
|248  // get the key converted into a game command                  |
|249  Command cmd = kb.GetKey ();                                   |
|250  if (cmd == Stop) { // handle the "pause game" cmd             |
|251   Wait ();                                                     |
|252   return GetNextCommand ();                                    |
|253  }                                                             |
|254  else if (cmd == Abort) { // handle the abort game cmd         |
|255   done = true;                                                 |
|256   return false;                                                |
|257  }                                                             |
|258  else p.SetCommand (cmd); // here, set the cmd into the player |
|259  return true;                                                  |
```

```
|260 }                                                                |
|261                                                                  |
|262 /***************************************************************/|
|263 /*                                                             */|
|264 /* Wait: pause the game until the player hits a new command key*/|
|265 /*                                                             */|
|266 /***************************************************************/|
|267                                                                  |
|268 void Game::Wait () {                                             |
|269  Command cmd;                                                    |
|270  do {                                                            |
|271   cmd = kb.GetKey ();                                            |
|272  }                                                               |
|273  while (cmd == Stop);                                            |
|274                                                                  |
└──────────────────────────────────────────────────────────────────┘
```

As you look over the coding of the Game class functions, notice that none of it is particularly tricky or obtuse. It's all straightforward, indicating a good design.

The coding for main() is exceedingly simple. It allocates an instance of the Game class and calls Run(). A try-catch block handles the displaying of any errors in loading the game.

```
int main () {
 Screen s (Screen::Blue, Screen::BrightYellow);
 s.SetTitle ("Ladder Game");
 s.ClearScreen ();
 try {
  Game g ("EasyStreet.lvl", s);
  g.Run ();
 }
 catch (const char* msg) {
  cerr << msg << endl;
 }
 s.GetAnyKey ();
```

If this were a real game, then main would need to know whether or not the player actually won the level. If the player successfully navigated the level, then main would allocate a new game using the next level in the overall campaign. Additionally, some means of retaining the player's current score must be made so that when the player enters the second level, his money does not revert back to zero. Similarly, main could also handle saving a record of the highest scores to a small text file.

Problems

Problem 7-1 Modifications to Ladder

Modify the Ladder Game as follows.

1. Devise a way for main to retrieve the player's score and save the player's name and score to a "highest scores" type file. Allow for saving the last five highest scores.

2. Redesign the barrel generator mechanism to allow for the generator to be located on either side of the top row, thus allowing barrels to initially spew out from the right or left sides instead of always from the top falling down.

3. Redesign the Run loop to incorporate a finite amount of time for a player to get to the exit point. If time runs out, the player loses. Additionally, add in some bonus time for each money bag retrieved along with incrementing the player's money.

4. Redesign the parameter settings method to display menu choices for

Very Easy
Easy
Difficult
Challenging
Impossible

Thus, when the game begins, this menu is shown. You may choose values for game speed, maximum number of barrels, and the frequency of barrel production that corresponds to these categories.

5. Create a second .lvl file for the game. Once it is working, modify main to first use Easy Street. Then, when the player succeeds with this first level, load in the second level and allow the user to play the second level. This is the start of a campaign Ladder game.

Thoroughly test all of your modifications.