Project 3
Goonie Blast

For questions about this project, first consult your TA.
If your TA can't help, ask Professor Chang.

Time due:

Part 1: 9 PM, Sunday, July 26
Part 2: 9 PM, Monday, Aug 03

WHEN IN DOUBT ABOUT A REQUIREMENT, YOU WILL NEVER LOSE CREDIT IF YOUR SOLUTION WORKS THE SAME AS OUR POSTED SOLUTION.
SO PLEASE DO NOT ASK ABOUT ITEMS WHERE YOU CAN DETERMINE THE PROPER BEHAVIOR ON YOUR OWN FROM OUR SOLUTION!

PLEASE THROTTLE THE RATE YOU ASK QUESTIONS TO 1 EMAIL PER DAY! IF YOU'RE SOMEONE WITH LOTS OF QUESTIONS, SAVE THEM UP AND ASK ONCE.

Table of Contents

Introduction

ChangSoft has learned that NachenGames corporate and SmallSoft released a very popular game called “Boulder Blast” and both companies are working on a new release called “Goonie Blast”. In order to beat these 2 companies, ChangSoft wanted you to program the exact same copy as “Goonie Blast”. ChangSoft spies have managed to steal a prototype Goonie Blast executable file and several source files from the NachenGames headquarters, so you can see exactly how your version of the game must work (see posted executable file) and even get a head start on the programming. Of course, such behavior would never be appropriate in real life, but for this project, you’ll be a programming villain.

In Goonie Blast, the player has to navigate through a series of robot-infested mazes in order to gather valuable jewels and rescue hostages. After the player has rescued all the hostages and gathered each of the jewels within the main maze and all of its sub-mazes, an exit will be revealed, and the player may then use the exit to advance to the next maze. The player wins by completing all of the mazes.

Here is an example of what the Goonie Blast game looks like:

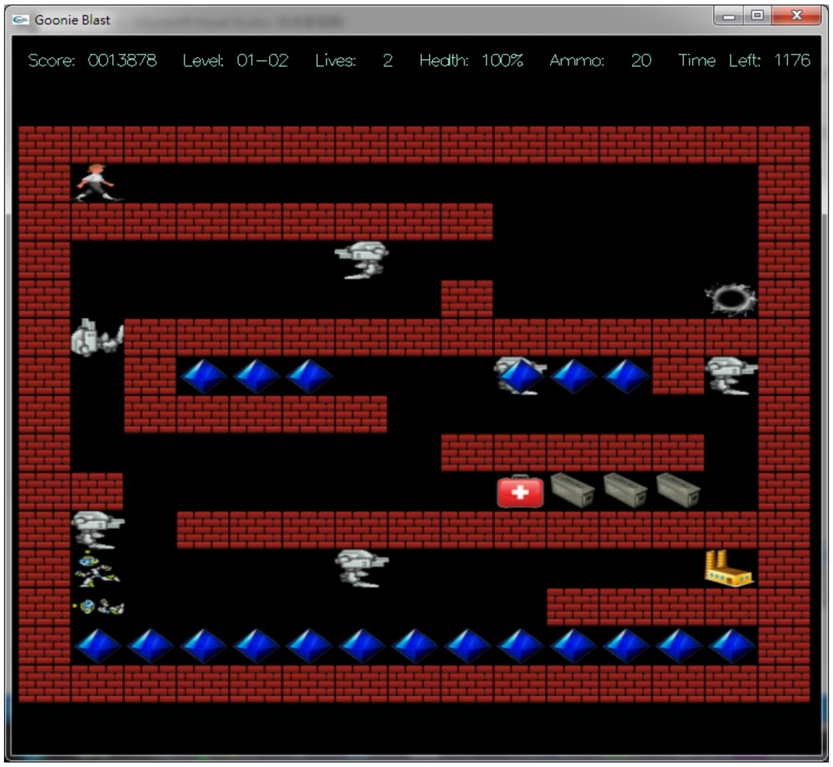
Goonie Blast
Score: 0013878 Level: 01–02 Lives: 2 Health: 100% Ammo: 20 Time Left: 1176

Figure #1: A screenshot of the Goonie Blast game. You can see the player (top-left), two different types of robots (SnarlBots and KleptoBots), a transport gate, KleptoBot factories, blue Jewels, a healing kit, and 3 ammunition kits.

Game Details

In Goonie Blast, the Player starts out a new game with three lives and continues to play until all of his/her lives have been exhausted. There are multiple main levels and their sub levels in Goonie Blast, beginning with level 0, and each level has its own sub-level starting also at 0. For example, 00-00 means 0-th main level and no sub-level and 01-02 means 01-th main level and second sub-level. During each main level, the Player must rescue all the hostages and gather all of the blue Jewels within the current maze and its sub-mazes before the Exit is revealed and they may use it to move on to the next main level.

Upon starting each main level, the Player’s avatar is placed in a main maze filled with one or more blue Jewels, transport gates, robots, robot Factories and other Goodies. The player may use the arrow keys to move their avatar (the Indiana Jones-looking character) left, right, up and down through the maze. They may walk on any square so long as it doesn’t have a Wall, a Factory, or a robot on it. In addition to walking around the maze, the Player may also shoot their laser cannon – but beware, it has only limited ammunition. Laser bullets (is that an oxymoron? ☺) can destroy robots, but it takes more than one shot to do so.

There are three different types of Goodies distributed throughout the maze that the Player can collect in addition to blue Jewels. If the Player’s avatar steps upon the same square as an Extra Life Goodie, it instantly gives the Player an extra life. If the avatar steps onto the same square as a Restore Health Goodie, it will restore the Player to full health (in case of having been injured by shots from the robots). Finally, if the avatar steps onto the same square as an Ammo Goodie, it will give the Player 25 additional rounds of ammunition.

There are three major types of robots in Goonie Blast: Horizontal SnarlBots, Vertical SnarlBots, and KleptoBots.

As mentioned, SnarlBots fall into two categories – Horizontal SnarlBots and Vertical SnarlBots. Horizonal SnarlBots simply move back and forth on a row of the screen (only reversing course when they run into an obstacle), shooting at the Player’s avatar if he/she ever walks in their line of sight. Vertical SnarlBots are identical to their Horizontal cousins, but move up and down within a single column of the maze. SnarlBots can start out anywhere in the maze (depending on where the designer of the maze choses to put them).

In contrast to the SnarlBot, the KleptoBots are a bit nastier. These robots wander around the maze looking for Goodies to steal. If they happen to step onto a Goodie (an Extra Life Goodie, a Restore Health Goodie, or an Ammo Goodie) and they don’t already hold one, they may pick it up for themselves. KleptoBots simply wander aimlessly around the maze looking for, and picking up, Goodies. They are otherwise harmless and will not fire upon the Player’s avatar. So beware! When KleptoBot dies, if it previously picked up a Goodie, it will drop this object upon its square in the maze. KleptoBots are created by

KleptoBot Factories. KleptoBots never start out in the maze; they are added only by Factories.

Once the Player has collected all of the blue Jewels and rescued all of the hostages from the main maze and all the sub-mazes, an Exit will appear. The Exit is invisible and unusable until all of the Jewels have been collected and until all of the hostages have been rescued from the main level and all its sub-levels. Once the Exit has been revealed, the Player must direct their avatar to the Exit in order to advance to the next main level. The Player will be granted 1000 points for exiting a level. The Player will also be given a bonus for completing the level quickly by adding the remaining time to the score. The game is complete once the Player has used the Exit on the last main level.

If the Player's health reaches zero (the Player loses health when shot), their avatar dies and loses one "life." If, after losing a life, the Player has one or more remaining lives left, they are placed back on the current main level and they must again solve the entire main level and its sub-levels from scratch (with the main level starting as it was at the beginning of the first time it was attempted). The Player will restart the main level with full health points, as well as zero rounds of ammunition (regardless of how many they had when they died). If the avatar dies and has no lives left, then the game is over.

The Goonie Blast maze is exactly 15 squares wide by 15 squares high, and both the Player's avatar and robots may move to any adjacent square that doesn't contain a Wall, a robot, the Player or a Factory. The one exception is that KleptoBots are born in Factories, so they may start out on the same square as a Factory, but are not allowed to move back onto their birth Factory once they've left it. The bottom-leftmost square has coordinates x=0,y=0, while the upper-rightmost square has coordinate x=14,y=14, where x increases to the right and y increases upward toward the top of the screen. You can look in our provided file, *GameConstants.h*, for constants that represent the maze's width and height.

In Goonie Blast, each main level's maze is stored in a different data file. For example, the first main level's maze is stored in a file called level00.dat. The first sub-level for main level 0 is stored at level00_1.dat. The second sub-level for main level 0 is stored at level00_2.dat. The second main level's maze is stored in a file called level01.dat, and so on. Each time the Player is about to start a new main level, your code must load the data from the appropriate data file (using a class called *Level* that we provide) and then use this data to determine the layout of the current level.

Each level data file contains a specification for the layout of the maze, the initial locations of all the SnarlBots and the Player's avatar, as well as the initial locations of all Gates, Factories, Jewels, Goodies and the Exit. For more information on the level data files, please see the Level Data File section below. You may define your own level data files to customize your game (and more importantly, to test it). When a main level data file is loaded, the sub-level data files for that main level shall be loaded all at once. There are only up to 5 sub-levels for the main level, and there are no sub-sub-levels for a sub-level.

When a Player's avatar moves to the same location as the Gate, the player will be transported to a sub-level (indicated by the Gate's number). However, all the Gates in Goonie Blast are all one-time gate. If the player fails to collect all the jewels and rescue all the hostages in a sub-level before returning back to the main-level, then the player can no longer return back to the sub-level and shall die when the time limit is zero.

Once a new maze has been prepared and the Player's avatar and all of the robots and items in the main maze and all of its sub-mazes have been properly situated, the game play begins. Game play is divided into *ticks*, and there are twenty ticks per second (to provide smooth animation and game play). During each tick, the following occurs:

1. The Player has an opportunity to move their avatar exactly one square horizontally or vertically, fire their laser cannon (if they have ammunition), or give up (some levels are unsolvable if the player makes a mistake, so if the player realizes this, they can press the Escape key to lose a life and restart the level from scratch).
2. Every other object in the maze (e.g., SnarlBots, KleptoBots, Factories, Goodies, etc.) is given an opportunity to do something. For example, when given the opportunity to do something, a SnarlBot can move one square (left, right, up or down) according to its built-in movement algorithm (the SnarlBot movement algorithms are described in detail in the various SnarlBot sections below).

The player controls the direction of their avatar with the arrow keys, or for lefties and others for whom the arrow key placement is awkward, WASD or the numeric keypad: up is *w* or *8*, left is *a* or *4*, down is *s* or *2*, right is *d* or *6*. The Player can sacrifice one life and restart the current main level by pressing the Escape key at any time.

The Player's avatar may fire their laser cannon by pressing the spacebar. A laser Bullet that hits a SnarlBot or a KleptoBot does 3 points of damage to it. If, after repeated hits, the robot dies, the Player earns points:

For destroying a SnarlBot of any type:	200 points
For destroying a KleptoBot:	20 points

The Player also earns points (and special benefits) by picking up (i.e., moving onto the same square as) various items:

Blue Jewel: 100 points (all Jewels must be collected to advance to the next level)
Extra Life Goodie: 500 points (the user gets an extra life)
Restore Health Goodie: 1000 points (the user's health is restored to 100%)
Ammo Goodie: 200 points (the user receives 25 additional rounds of ammunition)

Players also earn bonus points for completing a level quickly. Each main level's maze starts with a time limit of 1500 ticks (20 ticks per second). During each tick of the game, the time limit is reduced by one until the time limit reaches zero (and the player dies when the time limit hits zero). If and when the player completes the current main level, whatever time limit remains is added onto their score. This incentivizes the Player to complete each level as quickly as possible since the Player wants to maximize their score in the game.

The Player starts with three lives. The Player loses a life if their health reaches zero (from being shot by robots) or when the time limit reaches zero.

When a Player dies, the Player's number of remaining lives is decremented by 1. If the Player still has at least one life left, then the user is prompted to continue and given another chance by restarting the current main maze level from scratch. All of the SnarlBots that were initially on the level will again be alive and returned to their starting positions, the Player's avatar will be returned to their starting position, and the maze will revert back to its original state (all Jewels, Goodies, Gates and Hostages in their initial positions). In addition, if the Exit was exposed in the maze prior to the avatar's death, then this too will be hidden from the maze until such time that the Player collects all Jewels on the main level and all its sub-levels. Then game play restarts. If the player is killed and has no lives left, then the game is over. Pressing the *q* key lets you quit the game prematurely.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects; in Goonie Blast, those objects include the Player's avatar, SnarlBots (Horizontal and Vertical), KleptoBots, Goodies (e.g., Extra Life Goodies), Jewels, Hostages, Gates, Walls, Factories, and the Exit. Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own x,y location in the maze, its own internal state (e.g., a SnarlBot knows its location, what direction it's moving, etc.) and its own special algorithms that control its actions in the game based on its own state and the state of the other objects in the world. In the case of the Player's avatar, the algorithm that controls the avatar actor object is the user's own brain and hand, and the keyboard! In the case of other actors (e.g., SnarlBots), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

Once a game begins, gameplay is divided into *ticks*. A tick is a unit of time, for example, 50 milliseconds (that's 20 ticks per second).

During a given tick, the game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object 1 square to the left), or change other objects' states (e.g., when a SnarlBot's algorithm is called by the game, it may determine that the Player's avatar has moved into its line of fire, and it may fire its laser cannon). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a SnarlBot will move just one square left/right/up/down, rather than moving two or more squares; a SnarlBot moving, say, 5 squares in a single tick would confuse the user, because humans are used to seeing smooth movement in video games, not jerky shifts.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actors' states), our game framework (that we provide) animates the actors onto the screen in their new configuration. So if a SnarlBot changed its location from 10,5 to 11,5 (moved one square right), then our game framework would erase the graphic of the SnarlBot from location 10,5 on the screen and draw the SnarlBot's graphic at 11,5 instead. Since this process (asking actors to do something, then animating them to the screen) happens 20 times per second, the user will see somewhat smooth animation.

Then, the next tick occurs, and each object's algorithm is again allowed to do something, our framework displays the updated actors on-screen, etc.

Assuming the ticks are quick enough (a fraction of a second), and the actions performed by the objects are subtle enough (i.e., a SnarlBot doesn't move 3 inches away from where it was during the last tick, but instead moves 1 millimeter away), when you display each of the objects on the screen after each tick, it looks as if each object is performing a continuous series of fluid motions.

A video game can be broken into three different phases:

Initialization: The game World is initialized and prepared for play. This involves allocating one or more actors (which are C++ objects) and placing them in the game world so that they will appear in the maze.

Game play: Game play is broken down into a bunch of ticks. During each tick, all of the actors in the game have a chance to do something, and perhaps die. During a tick, new actors may be added to the game and actors who die must be removed from the game world and deleted.

Cleanup: The Player has lost a life (but has more lives left), the Player has completed the current level, or the Player has lost all of their lives and the game is over. This phase frees all of the objects in the World (e.g., SnarlBots, Walls, Hostages, Goodies, the Player's avatar, etc.) since the level has ended. If the game is not over (i.e., the Player has more lives), then the game proceeds back to the *Initialization* step, where the maze is repopulated with new occupants and game play restarts at the current level.

Here is what the main logic of a video game looks like, in pseudocode (we provide some similar code for you in our provided GameController.cpp):

```
while (The Player has lives left)
{
   Prompt_the_user_to_start_playing(); // "press a key to start"
   Initialize_the_game_world();        // you're going to write this

   while (the Player is still alive)
   {
```

```
            // each pass through this loop is a tick (1/20th of a sec)

            // you're going to write code to do the following
          Ask_all_actors_to_do_something();
          Delete_any_dead_actors_from_the_world();

            // we write this code to handle the animation for you
          Animate_all_of_the_actors_to_the_screen();
          Sleep_for_50ms_to_give_the_user_time_to_react();
    }
      // the Player died - you're going to write this code
    Cleanup_all_game_world_objects();   // you're going to write this
    if (the Player is still alive)
       Prompt_the_Player_to_continue();
 }

 Tell_the_user_the_game_is_over();      // we provide this
```

And here is what the `Ask_all_actors_to_do_something()` function might look like:

```
void Ask_all_actors_to_do_something()
{
   for each actor on the current level:
       if (the actor is still alive)
           tell the actor to doSomething();
}
```

You will typically use a container (an array, vector, or list) to hold pointers to each of your live actors. Each actor (a C++ object) has a *doSomething( )* member function in which the actor decides what to do. For example, here is some pseudocode showing what a (simplified) SnarlBot might decide to do each time it gets asked to do something:

```
class SnarlBot: public SomeOtherClass
{
   public:
       void doSomething()
       {
           If the player is in my line of sight, then
                  Fire my laser cannon in the direction of the player
           Else if I can move in my current direction w/o hitting an obstacle, then
                  Move one square in my current direction
           Else if I'm about to run into an obstacle, then
                  Reverse my direction, but don't move during this tick
       }
       ...
};
```

And here's what the Player's *doSomething( )* member function might look like:

```
class Player: public …
{
   public:
        void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the UP key and that square is open then
                   Increase my y location by one
```

```
                If the user pressed the DOWN key and that square is open then
                    Decrease my y location by one
                ...
                If the user pressed the space bar to fire and the player has
                   ammunition, then
                   Introduce a new laser bullet object into the game
                ...
        }
        ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Goonie Blast game. Your classes must work properly with our provided classes, and **you must not modify our classes or our source files in any way to get your classes to work properly (doing so will result in a score of zero on the entire project!).** Here are the specific classes that you must create:

1. You must create a class called *StudentWorld* which is responsible for keeping track of your game world (including the maze) and all of the actors/objects (SnarlBots, KleptoBots, Bullets, Jewels, Goodies, Gates, Hostages, the Player's avatar, etc.) that are inside the main maze and all of its sub-mazes.
2. You must create a class to represent the Player in the game.
3. You must create classes for Horizontal/Vertical SnarlBots, KleptoBots, Factories, Hostages, Gates, Walls, Jewels, Extra Life Goodies, Restore Health Goodies, Ammo Goodies, Walls, and the Exit, as well as any additional base classes (e.g., a Robot base class if you find it convenient) that help you implement the game.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all game play – it keeps track of the whole game world (the main maze / sub-mazes and all of their inhabitants such as SnarlBots, KleptoBots, the Player, Hostages, Walls, the Exit, Goodies, etc.). It is responsible for initializing the game world at the start of the game, asking all the actors to do something during each tick of the game, destroying an actor when it disappears (e.g., a SnarlBot dies), and destroying all of the actors in the game world when the user loses a life.

Your StudentWorld class **must** be derived from our GameWorld class (found in *GameWorld.h*) and **must** implement at least these three methods (which are defined as pure virtual in our GameWorld class):

```
        virtual int init() = 0;
        virtual int move() = 0;
        virtual void cleanUp() = 0;
```

The code that you write must *never* call any of these three functions. Instead, our provided game framework will call these functions for you. So you have to implement them correctly, but you won't ever call them yourself in your code.

When a new main level starts (e.g., at the start of a game, or when the player completes the current main level and advances to the next main level), our game framework will call the *init()* method that you defined in your *StudentWorld* class. You don't call this function; instead, our provided framework code calls it for you.

The *init()* method is responsible for loading the current main level's maze from a data file (we'll show you how below), constructing a representation of the current main level in your StudentWorld object, loading all of the main level's sub-levels, and constructing a representation of all the sub-levels in your StudentWorld object, using one or more data structures that you come up with.

The *init()* method is automatically called by our provided code either (a) when the game first starts, (b) when the player completes the current main level and advances to a new main level (that needs to be loaded/initialized), or (c) when the user loses a life (but has more lives left) and the game is ready to restart at the current main level.

When the player has finished the main level loaded from level00.dat, the next main level data file to load is level01.dat; after level01.dat, level02.dat; etc. If there is no main level data file with the next number, or if the main level just completed is level 99, the *init()* method must return `GWSTATUS_PLAYER_WON`. If the next main level file exists but is not in the proper format for a level data file, the *init()* method must return `GWSTATUS_LEVEL_ERROR`.

When the player has finished loading the current main level, the player needs to determine if there is a need to load sub-levels. The *init()* method must return `GWSTATUS_LEVEL_ERROR` if the sub-level file exists but is not in the proper format for a level data file. If there are no problems loading the main level and all of its sub-levels, the *init()* method must return `GWSTATUS_CONTINUE_GAME`.

Once a new main level and all of its sub-levels have been loaded/initialized with a call to the *init()* method, our game framework will repeatedly call the *StudentWorld's move()* method, at a rate of roughly 20 times per second. Each time the *move()* method is called, it must run a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., the Player's avatar, each SnarlBot, Goodie, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. Finally, this method is responsible for disposing of (i.e., deleting) actors (e.g., a Bullet, a dead SnarlBot, etc.) that need to disappear during a given tick. For example, if a SnarlBot is shot by the player and its "hit points" (life force) drains to zero, then its state should be set to dead, and then after all of the actors in the game get a chance to do something during the tick, the *move()* method should remove that SnarlBot from the game world (by deleting its object and removing any reference to the object from the *StudentWorld's* data structures). The *move()* method will automatically be called once during each tick of the game by our provided game framework. You will never call the *move()* method yourself.

The *cleanup()* method is called by our framework when the Player completes the current main level or loses a life (i.e., her/his hit points reach zero due to being shot by the robots or when the time limit is zero). The *cleanup()* method is responsible for freeing all actors (e.g., all SnarlBot objects, all KleptoBot objects, all Wall objects, the Player object, the Exit object, all Goodie objects, Jewel objects, all Bullet objects, etc.) that are currently in the game. This includes all actors created during either the *init()* method or introduced during subsequent game play by the actors in the game (e.g., a KleptoBot that was added to the maze by a KleptoBot Factory) that have not yet been removed from the game.

In addition to the above 3 pure abstract methods, there is one more pure abstract method that you need to implement in StudentWorld class.

```
virtual int getCurrentSubLevel() = 0;
```

The *getCurrentSubLevel()* method is used to locate the current sub-level. For example, if we store the main level as the first item in our container and the first sub-level as the second item in our container, then this method should return 0 for the main level and 1 for the first sub-level. After locating the correct sub-level, you can then manipulate all the objects correctly for that sub-level. If this method were not implemented correctly, you might be manipulating the objects from the main-level using the maze from the second sub-level. Implementing this method correctly is the key to complete this project.

You may add as many other public/private member functions or private data members to your *StudentWorld* class as you like (in addition to the above three member functions, which you *must* implement).

Your *StudentWorld* class must be derived from our *GameWorld* class. Our *GameWorld* class provides the following methods for your use:

```
unsigned int getLevel() const;
unsigned int getLives() const;
void decLives();
void incLives();
unsigned int getScore() const;
void increaseScore(unsigned int howMuch);
void setGameStatText(string text);
string assetDirectory() const;
bool getKey(int& value);
void playSound(int soundID);
```

getLevel() can be used to determine the current main level number.

getLives() can be used to determine how many lives the Player has left.

decLives() reduces the number of Player lives by one.

incLives() increases the number of Player lives by one.

getScore() can be used to determine the Player's current score.

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase the user's score upon successfully destroying a robot, picking up a Goodie of some sort, or completing a level (to give the Player their remaining level bonus). When your code calls this method, you must specify how many points the user gets (e.g., 200 points for destroying a SnarlBot). This means that the game score is controlled by our *GameWorld* object – you must *not* maintain your own score data member in your own classes.

The *setGameStatText()* method is used to specify what text is displayed at the top of the game screen, e.g.:

```
Score: 0321000  Level: 05-01  Lives:  3  Health:  70%  Ammo:  25  TimeLimit: 42
```

assetDirectory() returns the name of the directory that contains the game assets (image, sound, and level data files).

getKey() can be used to determine if the user has hit a key on the keyboard to move the Player, to fire, or to sacrifice one life and restart the main level. This method returns true if the user hit a key during the current tick, and false otherwise (if the user did not hit any key during this tick). The only argument to this method is a variable that will be set to the key that was pressed by the user (if any key was pressed). If the function returns true, the argument will be set to one of the following values (defined in *GameConstants.h*):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
KEY_PRESS_ESCAPE
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a robot dies or the Player picks up a Jewel). You can find constants (e.g., SOUND_ROBOT_DIE) that describe what noise to make in the *GameConstants.h* file. Here's how this method might be used:

```
  // if a SnarlBot reaches zero hit points and dies, make a dying sound

if (theRobotHasZeroHitPoints())
     studentWorldObject->playSound(SOUND_ROBOT_DIE);
```

init() Details

Your StudentWorld's *init()* member function must:

1. Initialize the data structures used to keep track of your game's world.
2. Load the current main maze's and all its sub-mazes' details from level data file(s).
3. Allocate and insert a valid Player object into the game world.
4. Allocate and insert any SnarlBots objects, Wall objects, Hostage objects, Factory objects, Jewel objects, Goodie objects, or Exit objects into the game world, as required by the specification in the level's data file.

To load the details of the current level from a level data file, you can use the Level class (described later) that we wrote for you, which can be found in the provided *Level.h* header file. Here's a brief example that uses the Level class to load a level data file:

```
#include "Level.h"   // you must include this file to use our Level class

int StudentWorld::someFunctionYouWriteToLoadALevel()
{
       string curLevel = "level03.dat";
       Level lev(assetDirectory());
       Level::LoadResult result = lev.loadLevel(curLevel);

       if (result == Level::load_fail_file_not_found ||
           result == Level:: load_fail_bad_format)
           return -1;              // something bad happened!

       // otherwise the load was successful and you can access the
       // contents of the level – here's an example

       int x = 0;
       int y = 5;
       Level::MazeEntry item = lev.getContentsOf(x, y);
       if (item == Level::player)
              cout << "The player should be placed at 0,5 in the maze\n";
       x = 10;
       y = 7;
       item = lev.getContentsOf(x, y);
       if (item == Level::wall)
              cout << "There should be a wall at 10,7 in the maze\n":

       … // etc
}
```

Notice that the *getContentsOf()* method takes the column parameter (x) first, then the row parameter (y) second. This is different than the order one normally uses when indexing a 2-dimensional array, which would be array[row][col]. Be careful!

You can examine the *Level.h* file for a full list of functions that you can use to access each level.

Once you load **the main level**'s layout and details using our Level class, your *init()* method must then construct a representation of your world and store this in a *StudentWorld* object. It is **required** that you keep track of all of the actors (e.g., SnarlBots, Walls, Extra Life Goodies, the Exit, Jewels, Hostages, etc.) in a **single** STL collection like a *list* or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, your *StudentWorld* object may keep a separate pointer to the Player object rather than keeping a pointer to that object in the container with the other actor pointers; the Player is the **only** actor allowed to not be stored in the single actor container.

The data structures get a bit more complex when loading all of the main level's sub-levels. Since the sub-levels have the same type of objects as the main-levels, it is possible to use an array of STL collections to store all the objects in the main level and its sub-levels. For example, we can declare a container of a container,

```
vector<  vector<Jewel*> > vec;
```

to store all the Jewel objects stored in the main level as vec[0] while vec[1] indicates all the Jewel objects in sub-level 1 (or in level00_1.dat).

You must not call the *init()* method yourself. Instead, this method will be called by our framework code when it's time for a new game to start (or when the player completes a level or needs to restart a level).

move() Details

The *move()* method must perform the following activities:

1. It must ask all of the actors that are currently active in the game world to do something (e.g., ask a SnarlBot to move itself, ask a Factory to potentially produce a new KleptoBot, give the Player a chance to move up, down, left or right, etc.).
    a. If an actor does something that causes the Player to die, then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.
    b. If the Player steps onto the same square as an Exit (after collecting all of the Jewels on the main level and all of its sub-levels and rescuing all the hostages), completing the current main level, then the *move()* method should immediately:
        i. Increase the player's score appropriately (by 1000 points for using the Exit, and by the remaining time limit for the level).
        ii. Return a value of `GWSTATUS_FINISHED_LEVEL`.
2. It must then delete any actors that have died during this tick (e.g., a SnarlBot that was killed by a Bullet and so should be removed from the game world, or a Goodie that disappeared because the Player picked it up).
3. It must reduce the level's time limit by one during each tick (The main level and all of its sub-levels share this time limit). Each level starts with a time limit of 1500, then goes down during each tick. So after the first tick, the time limit would drop to 999, after the second tick to 998, etc. This declining time limit value incentivizes the player to complete the level as quickly as possible to get the biggest score. The time limit may not go below a value of zero.

4. It should expose/activate the Exit on the current main level (no Exits are allowed in sub-levels) once the Player has collected all of the Jewels and has rescued all the hostages on the main level and all of its sub-levels (alternatively, the Exit object can do this instead of the *StudentWorld* object). This enables the Player to later move over to the Exit and complete the main level, advancing to the next main level.
5. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the time limit, etc.).

The *move()* method must return one of three different values when it returns at the end of each tick (all are defined in *GameConstants.h*):

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

The first return value indicates that the Player died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if the user has more lives left. If your *move()* method returns this value and the Player has more lives left, then our framework will prompt the Player to continue the game, call your *cleanup()* method to destroy the main level and all of its sub-levels, call your *init()* method to re-initialize the main level and all of its sub-levels from scratch, and then begin calling your *move()* method over and over, once per tick, to let the user play the main level and all of its sub-levels again.

The second return value indicates that the tick completed without the Player dying BUT the Player has not yet completed the current main level (Exit only appears in the main level). Therefore the game play should continue normally for the time being. In this case, the framework will advance to the next tick and call your *move()* method again.

The final return value indicates that the Player has completed the current main level (that is, gathered all of the Jewels on the level, rescued all the hostages, and stepped onto the same square as the Exit). If your *move()* method returns this value, then the current main level is over, and our framework will call your *cleanup()* method to destroy the main level and all of its sub-levels, advance to the next main level, then call your *init()* method to prepare that main level and all of its sub-levels for play, etc…

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a `GWSTATUS_PLAYER_DIED` status value from our dummy version of the *move()* method. Unless you implement something that returns `GWSTATUS_CONTINUE_GAME` your game will not display anything on the screen! So if your screen just shows up black once the user starts playing, you'll know why!

Here's pseudocode for how the move() method might be implemented:

```
int StudentWorld::move()
{
      // Update the Game Status Line
```

```
    updateDisplayText();   // update the score/lives/level text at screen top

     // The term "actors" refers to all robots, the Player, Goodies,
     // Hostages, Jewels, Gates, Bullets, the Exit, etc.

      // Give each actor a chance to do something
    for each of the actors in the game world
    {
        // You need to make sure the current_level has the correct level info
        // to indicate which level the actor resides in.
        if (actor[current_level][i] is still active/alive)
        {
            // ask each actor to do something (e.g. move)
          actor[current_level][i]->doSomething();

          if (thePlayerDiedDuringThisTick())
                return GWSTATUS_PLAYER_DIED;

          if (thePlayerCompletedTheCurrentLevel())
          {
                increaseScoreAppropriately();
                return GWSTATUS_FINISHED_LEVEL;
          }
        }
    }

      // Remove newly-dead actors after each tick for the current level
    removeDeadGameObjects(); // delete dead game objects for current level

      // Reduce the Time Limit by one
    reduceTimeLimitByOne();

      // If the player has collected all of the Jewels on the main level and
     // all of its sub-levels, then we must expose the Exit so the player
     // can advance to the next main level
    if (thePlayerHasCollectedAllOfTheJewelsOnTheMainLevel())
          exposeTheExitInTheMaze();  // make the exit Active in main level

      // return the proper result
    if (thePlayerDiedDuringThisTick())
          return GWSTATUS_PLAYER_DIED;

    if (thePlayerCompletedTheCurrentMainLevel())
    {
          increaseScoreAppropriately();
          return GWSTATUS_FINISHED_LEVEL;
    }

      // the player hasn't completed the current main level and hasn't died,
     // so continue playing the current level
    return GWSTATUS_CONTINUE_GAME;
}
```

Give Each Actor a Chance to Do Something

During each tick of the game each active actor must have an opportunity to do something (e.g., move around, shoot, etc.). Actors include the Player's avatar, SnarlBots, KleptoBots, Walls, Jewels, Hostages, Factories, Goodies, Gates, and the Exit.

Your *move()* method must enumerate each active actor in the maze (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g., a SnarlBot might move one step forward, the Player might shoot a Bullet, a Gate may transport the Player to a sub-level, etc.

It is possible that one actor (e.g., Bullet) may destroy another actor (e.g., a SnarlBot) during the current tick. If an actor has died earlier in the current tick, then the dead actor must not have a chance to do something during the current tick (since it's dead).

To help you with testing, if you press the `f` key during the course of the game, our game controller will stop calling move() every tick; it will call move() only when you hit a key (except the `r` key). Freezing the activity this way gives you time to examine the screen, and stepping one move at a time when you're ready helps you see if your actors are moving properly. To resume regular game play, press the `r` key.

Remove Dead Actors after Each Tick

At the end of each tick, your *move()* method must determine which of your actors are no longer alive, remove them from your container of active actors, and delete their objects (so you don't have a memory leak) in the current level (main level or its sub-level). So if, for example, a SnarlBot's hit points go to zero (due to it being shot) and it dies, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the SnarlBot object should be deleted (using the C++ delete expression) to free up memory for future actors that will be introduced later in the game. (Hint: Each of your actors could have a data member indicating whether or not it is still alive!)

Updating the Display Text

Your *move()* method must update the game statistics at the top of the screen during every tick by calling the *setGameStatText()* method that we provide in our *GameWorld* class. You could do this by calling a function like the one below from the *move()* method:

```
        void setDisplayText()
        {
           int score = getCurrentScore();
           int level = getCurrentGameLevel(); // (e.g. 03)
          int subLevel = getCurrentSubLevel(); // (e.g. 01. 00 is for main level)
           unsigned int timeLimit = getCurrentTimeLimit();
           int livesLeft = getNumberOfLivesThePlayerHasLeft();

// Next, create a string from your statistics, of the form:
// Score: 0000100  Level: 03-01  Lives:  3  Health:  70%  Ammo: 216  TimeLimit:   34

           string s = someFunctionToFormatThingsNicely(score, level, sublevel,
                                                        lives, health, ammo, bonus);
           // Finally, update the display text at the top of the screen with your
           // newly created stats
           setGameStatText(s);   // calls our provided GameWorld::setGameStatText
        }
```

Your status line must meet the following requirements:

1. Each field's label is followed by a colon and one space.
2. Each field's value after the colon and space must be exactly as wide as shown in the example above:
    a. The Score field must be 7 digits long, with leading zeros.
    b. The Level field must be 2 digits long, with leading zeroes.
    c. The subLevel field must be 2 digits long, with leading zeroes.
    d. The Lives field must be 2 digits long, with leading spaces (e.g., "_ 2", where _ in this sentence represents a space).
    e. The Health field must be 3 digits long and display the Player's health percentage (not hit points!), with leading spaces, and be followed by a percent sign (e.g., "_70%").
    f. The Ammo field should be 3 digits long, with leading spaces (e.g., "_ 24").
    g. The Time Limit field must be 4 digits long, with leading spaces (e.g., "_924").
3. Each statistic must be separated from the previous statistic by two spaces. For example, between the "0000100" of the score and the "L" in "Level" there must be exactly two spaces.

You may find the Stringstreams writeup on the class web site to be helpful.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the Player lost a life (e.g., their hit points reached zero due to being shot) or has completed the current level. In this case, every actor in the entire maze (the Player and every SnarlBot, Goodie, Jewel, Hostage, Wall, the Exit, etc.) must be deleted and removed from the StudentWorld's container of active objects, resulting in an empty maze. If the user has more lives left, our provided code will subsequently call your *init()* method to reload and repopulate the maze and the level will then continue from scratch with a brand new set of actors.

You must not call the *cleanUp()* method yourself when the Player dies. Instead, this method will be called by our code.

The Level Class and Level Data File

As mentioned, every level of Goonie Blast has a different maze. The maze layout for each level is stored in a data file, with the file *level00.dat* holding the details for the first level's maze, *level01.dat* holding the details for the second level's maze, etc.

Here's an example maze data file (you can modify our maze data files to create wacky new levels, or add your own new maze data files to add new levels, if you like):

level00.dat:

```
###############
#      @      #
#             #
# #    #    ###
#r#        h#2#
#*#h        #*#
#a#        h#e#
#*#h   #    #*#
#1#        h#a#
###h   #    # #
#             #
#             #
####### #######
#faraaaxaaara*#
###############
```

As you can see, the data file contains a 15x15 grid of different characters that represent the different actors in the main level. Valid characters for your maze data file are:

The @ character specifies the location of the Player's avatar when starting a level. The Player's avatar should also restart at this location if the player dies and must replay the current level.

The **#** character represents a Wall. The perimeter of each maze MUST be surrounded completely by Walls.

The **h** character represents a Horizontal SnarlBot, specifiying that a SnarlBot that moves only horizontally starts at this location in the maze when the Player starts or replays the current level.

The **v** character represents a Vertical SnarlBot, specifying that a SnarlBot that moves only vertically starts at this location in the maze when the Player starts or replays the current level.

The **1** character represents a transport Gate that transports the Player to the sub-level represented by level00_1.dat. The **2** character represents a transport Gate that transports the Player to the sub-level represented by level00_2.dat.

The ***** character represents a Jewel that the Player needs to pick up to complete the main level.

The **e** character represents an Extra Life Goodie that grants the Player an extra life (but does NOT restore the Player's current health!).

The **r** character represents a Restore Health Goodie that restores the Player's hit points to 100%.

The **a** character represents an Ammo Goodie that gives the Player 25 additional rounds of ammunition.

The **x** character represents the level's exit. The level's exit will be visible/active only after the Player has gathered all of the Jewels and rescued all the hostages on the main level and all of its sub-levels.

The **t** character represents a Hostage (not shown above).

All **space** characters represent locations where the Player's avatar and robots may walk within the maze.

The Level Class

We have graciously ☺ decided to provide you with a class that can load level data files for you. The class is called *Level* and may be found in our provided *Level.h* file. Here's how you might use this class:

```
#include "Level.h"    // required to use our provided class

void StudentWorld::someFunc()
{
        Level lev(assetDirectory());

        Level::LoadResult result = lev.loadLevel("level00.dat");
        if (result == Level::load_fail_file_not_found)
                cerr << "Could not find level00.dat data file\n";
        else if (result == Level::load_fail_bad_format)
                cerr << "Your level was improperly formatted\n";
        else if (result == Level::load_success)
        {
                cerr << "Successfully loaded level\n";

                Level::MazeEntry ge = lev.getContentsOf(5,10);       // x=5, y=10
                switch (ge)
                {
                        case Level::empty:
                                cout << "Location 5,10 is empty\n";
                                break;
                        case Level::exit:
                                cout << "Location 5,10 is where the exit is\n";
                                break;
                        case Level::player:
                                cout << "Location 5,10 is where the player starts\n";
                                break;
                        case Level::horiz_snarlbot:
                                cout << "Location 5,10 starts with a horiz. SnarlBot\n";
                                break;
                        case Level::vert_snarlbot:
                                cout << "Location 5,10 starts with a vertical SnarlBot\n";
                                break;
                        case Level::kleptobot_factory:
                                cout << "Location 5,10 holds a KleptoBot Factory\n";
                                break;
                        case Level::hostage:
                                cout << "Location 5,10 holds a hostage\n";
                                break;
                        case Level::wall:
                                cout << "Location 5,10 holds a Wall\n";
                                break;
                        default:
                                if( value >=0 && value <= 5) { // gate
                                        cout << "Location 5,10 must be a Gate\n";
                                }
                                break;
                }
        }
}
```

Hint: You will presumably want to use our *Level* class when loading the current main level (and all of its sub-levels) specification in your *StudentWorld*'s *init()* method.

You Have to Create the Classes for All Actors

The Goonie Blast game has a number of different game objects, including:

- The Player's avatar
- SnarlBots (Horizontal and Vertical varieties)
- KleptoBots
- Factories (for KleptoBots)
- Bullets (that can be shot by both the Player and robots)
- The Exit for the level
- Walls
- Hostages
- Gates
- Jewels
- Extra Life Goodies
- Restore Health Goodies
- Ammo Goodies

Each of these game objects can occupy the maze and interact with other game objects within the maze.

Now of course, many of your game objects will share things in common – for instance, every one of the objects in the game (SnarlBots, the Player, Walls, Hostages, etc.) has x,y coordinates. Many game objects have the ability to perform an action (e.g., move or shoot) during each tick of the game. Many of them can potentially be attacked (e.g., the

Player, and robots can be attacked by Bullets) and could “die” during a tick. All of them need some attribute that indicates whether or not they are still alive or they died during the current tick, etc.

It is therefore your job to determine the **commonalities** between your different game objects and make sure to factor out common behaviors and traits and move these into appropriate base classes, rather than duplicate these items across your derived classes – this is in fact one of the tenets of object oriented programming.

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. Your classes must never duplicate code or data member – if you find yourself writing the same (or largely similar) code across multiple classes, then this is an indication that you should define a common base class and migrate this common functionality/data to the base class. Duplication of code is a so-called *code smell*, a weakness in a design that often leads to bugs, inconsistencies, code bloat, etc.

Hint: When you notice this specification repeating the same text nearly identically in the following sections (e.g., in the Extra Life Goodie section and the Restore Health Goodie section, or in the SnarlBot and KleptoBots sections) you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate behaviors across classes that can be moved into a base class!

You MUST derive all of your game objects directly or indirectly from a base class that we provide called *GraphObject*, e.g.:

```
class Actor: public GraphObject
{
public:
        …
};

class KleptoBot: public Actor
{
public:
        …
};

class SnarlBot: public Actor
{
public:
        …
};
```

GraphObject is a class that we have defined that helps hide the ugly logic required to graphically display your actors on the screen. If you don’t derive your classes from our *GraphObject* base class, then you won’t see anything displayed on the screen! ☺

The *GraphObject* class provides the following methods that you may use:

```
GraphObject(int imageID, int startX, int startY,
                                          Direction startDirection = none);
void setVisible(bool shouldIDisplay);
void getX() const;
void getY() const;
void moveTo(int x, int y);
Direction getDirection() const;  // Directions: none, up, down, left, right
void setDirection(Direction d);  // Directions: none, up, down, left, right
unsigned int getID() const;
```

You may use any of these member functions in your derived classes, but you **must not** use any other member functions found inside of *GraphObject* in your other classes (even if they are public in our class). You must not redefine any of these methods in your derived classes since they are not defined as virtual in our base class.

GraphObject(int imageID, int startX, int startY, Direction startDirection, int sub_level) is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID that indicates how the *GraphObject* should be displayed on screen (e.g., as a SnarlBot, a Player, a Wall, etc.). You must also specify the initial x,y location of the object. The x value may range from 0 to VIEW_WIDTH-1 inclusive, and the y value may range from 0 to VIEW_HEIGHT-1 inclusive. Notice that you pass the coordinates as x,y (i.e., column, row starting from bottom left, and *not* row, column). For those objects for which the concept of a direction doesn't apply (e.g., Walls or Jewels), you may leave off the final Direction argument or may specify *none*; For those objects for which a direction applies (i.e., the Player, robots, Bullets), you must specify the initial direction the object is facing, (i.e., *left*, *right*, *up*, or *down* – these constants are defined in the *GraphObject.h* file). The sub_level should range from 0 to 5 indicating whether the object should be created at which level (e.g. 0 means creating this object at the main level and 1 means creating this object at the first sub-level).

One of the following IDs, found in *GameConstants.h*, must be passed in for the imageID value:

```
IID_PLAYER
IID_SNARLBOT
IID_KLEPTOBOT
IID_ROBOT_FACTORY
IID_BULLET
IID_EXIT
IID_WALL
IID_HOSTAGE
IID_GATE
IID_JEWEL
IID_RESTORE_HEALTH
IID_EXTRA_LIFE
IID_AMMO
```

New *GraphObjects* start out invisible and are **NOT** displayed on the screen until the programmer calls the *setVisible()* method with a value of true for the parameter.

setVisible(bool shouldIDisplay) is used to tell our graphical system whether or not to display a particular *GraphObject* on the screen. If you call *setVisible(true)* on a *GraphObject*, then your object will be displayed on screen automatically by our framework (e.g., a SnarlBot image will be drawn to the screen at a sub-level at the *GraphObject's* specified x,y coordinates if the object's Image ID is IID_SNARLBOT). If you call *setVisible(false)* then your GraphObject will not be displayed on the screen at that sub-level. When you create a new game object, always remember to call the *setVisible()* method with a value of *true* or the actor won't display on screen!

getX() and *getY()* are used to determine a GraphObject's current location in the maze. Since each GraphObject maintains its x,y location, this means that **your derived classes MUST NOT also have x,y member variables**, but instead use these fucntions and *moveTo()* from the GraphObject base class.

moveTo(int x, int y) is used to update the location of a GraphObject within the maze. For example, if a SnarlBot's movement logic dictates that it should move to the right, you could do the following:

```
	moveTo(getX()+1, y);       // move one square to the right
```

You must use the *moveTo()* method to adjust the location of a game object if you want that object to be properly animated. As with the GraphObject constructor, note that the order of the parameters to moveTo is x,y (col,row) and NOT y,x (row,col).

getDirection() is used to determine the direction a GraphObject is facing. For example, a bullet fired by a robot must travel in the direction the robot is facing, so you can use this method to learn that direction.

setDirection(Direction d) is used to change the direction a GraphObject is facing. For example, when the user presses the up arrow, you can use this method to cause the player's avatar to be displayed facing up, as well as causing any bullets the Player fires to travel upward.

The Player

Here are the requirements you must meet when implementing the Player class.

<u>What the Player Must Do When It Is Created</u>

When it is first created:

1. The Player object must have an image ID of IID_PLAYER.
2. The Player must always start at the proper location as specified by the current level's data file. Hint: Since your *StudentWorld*'s *init()* function loads the level, it knows this x,y location to pass when constructing the Player object.
3. The Player, in its initial state:

   a. Has 20 hit points.
   b. Has 0 rounds of ammunition.
   c. Faces right.
4. The sub-level that the Player is located (e.g. 0 means the main level, and 1 means the first sub-level).

In addition to any other initialization that you decide to do in your Player class, a Player object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<ins>What the Player Must Do During a Tick</ins>

The Player must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something, the Player must do the following:

1. The Player must check to see if it is currently alive. If not, then the Player's *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, the *doSomething()* method must check to see if the user pressed a key (the section below shows how to check this). If the user pressed a key:
   a. If the user pressed the Escape key, the user is asking to abort the current level. In this case, the Player object should set itself to dead. The code in the *StudentWorld* class should detect that the player has died and address this appropriately (e.g., replay the level from scratch, or end the game).
   b. If the user pressed the space bar, then if the Player has any ammunition, the Player will fire a Bullet, which reduces their ammunition count by 1. To fire a Bullet, a new Bullet object must be added at the square immediately in front of the Player's avatar, facing the same direction as the avatar. For example, if the Player is at x=10,y=7 facing upward, then the Bullet would be created at location x=10, y=8 facing upward. When the Player fires a bullet, it must play the SOUND_PLAYER_FIRE sound effect (see the *StudentWorld* section of this document for details on how to play a sound).

      Hint: When you create a new Bullet object in the proper location and facing the proper direction, give it to the StudentWorld to manage (e.g., animate) along with the other game objects.

   c. If the user asks to move up, down, left or right by pressing a directional key, then the Player's direction should be adjusted to the indicated direction (e.g., if the Player were facing upward, and the user hit the right arrow key, the Player's direction should be adjusted to *right*). Then, the Player will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: a Wall, a robot, or a KleptoBot Factory. If the Player can move, it must update its location with the *GraphObject* class's *moveTo()* method.

What the Player Must Do When It Is Attacked

When the Player is attacked (i.e., a Bullet collides with him/her), the Player's hit points must be decremented by 3 points. If the Player still has hit points after this, then the game must play an impact sound effect: SOUND_PLAYER_IMPACT; otherwise, the game must set the Player's state to dead and play a death sound effect: SOUND_PLAYER_DIE.

Getting Input From the User

Since Goonie Blast is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within the Player's *doSomething()* method— that would stop your program and wait for the user to type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a directional key, then hit Enter, etc. Instead of this approach, you will use a function called *getKey()* that we provide in our *GameWorld* class (from which your StudentWorld class is derived) to get input from the user[1]. This function rapidly checks to see if the user has hit a key. If so, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void Player::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
          // user hit a key this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move player to the left ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move player to the right ...;
            break;
          case KEY_PRESS_SPACE:
           ... add a Bullet in the square in front of the Player...;
           break;

         // etc…
        }
    }
    ...
}
```

[1] Hint: Since your Player class will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your Player class (or more likely, one of its base classes) will need a way to obtain a pointer to the *StudentWorld* object it's playing in. If you look at our code example, you'll see how the Player's *doSomething()* method first gets a pointer to its world via a call to *getWorld()* (a method in one of its base classes that returns a pointer to a *StudentWorld*), and then uses this pointer to call the *getKey()* method.

Wall

Walls don't really do much. They just sit still in place. Here are the requirements you must meet when implementing the Wall class.

What a Wall Must Do When It Is Created

When it is first created:

1. A Wall object must have an image ID of IID_WALL.
2. A Wall must always start at the proper location as specified by the current level's data file.
3. A Wall has no direction (Hint: Its ancestor *GraphObject* base object always has the direction *none*).
4. A Wall object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Wall class, a Wall object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Wall Must Do During a Tick

A Wall must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something, the Wall must do nothing. After all, it's just a Wall! (What would you think it would do?)

What a Wall Must Do When It Is Attacked

When a Wall is attacked (i.e., a Bullet collides with it), nothing happens to the Wall.

Bullet

You must create a class to represent a Bullet. Here are the requirements you must meet when implementing the Bullet class.

What a Bullet Must Do When It Is Created

When it is first created:

1. A Bullet object must have an image ID of IID_BULLET.
2. A Bullet must have its x,y location specified for it – the Player or robot that fires the Bullet must pass in this x,y location when constructing a Bullet object.

3. A Bullet must have its direction specified for it – the Player or robot that fires the Bullet must pass in this direction when constructing the Bullet object.
4. A Bullet object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Bullet class, a Bullet object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Bullet Must Do During a Tick

Each time a Bullet object is asked to do something (during a tick):

1. The Bullet must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, the Bullet must check to see if any other objects are on its current square:
   a. If the Bullet is on the same square as a robot, or the Player, then the Bullet must:
      i. Damage the object appropriately (by causing the object to lose 2 hit points) – each damaged object can then deal with this damage in its own unique way (this may result in the damaged object dying/disappearing, a sound effect being played, the user possibly getting points, etc.) Hint: The Bullet can tell the object that it has been damaged by calling a method that object has (presumably named *damage* or something similar).
      ii. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
      iii. Do nothing else during the current tick.
   b. If the Bullet is on the same square as a Wall or a KleptoBot Factory, it will simply hit the Wall or Factory and do no damage. In this case, the Bullet must set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick). The Bullet must then do nothing more during the current tick. Note: If a Bullet finds itself on a square with both a robot *and* a Factory, then the Bullet must damage the robot.
   c. If the Bullet is on the same square as any other game object (e.g., another Bullet, the Exit, a Gate, a Goodie, etc.) or on an empty square by itself, it does not interact with the other object in any way. Continue with step #3.
3. The Bullet must move itself one square in its initially-specified direction.
4. Finally, after the Bullet has moved itself, the Bullet must check to see if there are objects are on its *new* square where it just moved, using the same algorithm described in step #2 above. If so, it must perform the same behavior as described in step #2 (e.g., damage the object, etc.), but does not move any further during this tick.

What a Bullet Must Do When It Is Attacked

Bullets can't be attacked, silly. Bullets will pass right through other Bullets.

Gate

You must create a class to represent a one-time transport Gate. When a Player is moved onto the same square as an one-time transport Gate, the Player will be transported to the corresponding sub-level and this one-time transport will disappear forever. Here are the requirements you must meet when implementing the Gate class.

What a Gate Must Do When It Is Created

When it is first created:

1. A Gate object must have an image ID of IID_GATE.
2. A Gate must always start at the proper location as specified by the current level's data file.
3. A Gate has a gate number indicating which sub-level should the player be transported to.
4. A Gate has no direction.
5. A Gate object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Gate class, a Gate object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Gate Must Do During a Tick

Each time a Gate object is asked to do something (during a tick):

1. The Gate must check to see if it is alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if a Player is on the same square as the Gate, then the Gate must:
   a. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   b. Inform the *StudentWorld* object which sub_level should the Player be transported to (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick). Your StudentWorld object will need to respond to this input and set the sub_level appropriately so that the maze displayed on the screen is correct.

   Overall, this results in the Gate transporting the Player to a sub_level maze.

What a Gate Must Do When It Is Attacked

Gates can’t be attacked. Bullets will pass right over them.

Jewel

You must create a class to represent a Jewel. When the Player picks up a Jewel (by moving onto the same square as it), he/she gets 100 points. Here are the requirements you must meet when implementing the Jewel class.

What a Jewel Must Do When It Is Created

When it is first created:

1. A Jewel object must have an image ID of IID_JEWEL.
2. A Jewel must always start at the proper location as specified by the current level’s data file.
3. A Jewel has no direction.
4. A Jewel object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Jewel class, a Jewel object must make itself visible using the *GraphObject* class’s *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Jewel Must Do During a Tick

Each time a Jewel is asked to do something (during a tick):

1. The Jewel must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the Player is on the same square as the Jewel, then the Jewel must:
    a. Inform the *StudentWorld* object that the user is to receive 100 more points. It can do this by using *StudentWorld’s increaseScore()* method (inherited from *GameWorld*).
    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound effect to indicate that the Player picked up the Jewel: SOUND_GOT_GOODIE.

What a Jewel Must Do When It Is Attacked

Jewels can’t be attacked. Bullets will pass right over them.

Hostage

You must create a class to represent a Hostage. When the Player rescues a Hostage (by moving onto the same square as it), he/she gets 0 points. Here are the requirements you must meet when implementing the Hostage class.

What a Hostage Must Do When It Is Created

When it is first created:

1. A Hostage object must have an image ID of IID_ HOSTAGE.
2. A Hostage must always start at the proper location as specified by the current level's data file.
3. A Hostage has no direction.
4. A Hostage object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Hostage class, a Hostage object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Hostage Must Do During a Tick

Each time a Hostage is asked to do something (during a tick):

1. The Hostage must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the Player is on the same square as the Hostage, then the Hostage must:
   i. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   ii. Play a sound effect to indicate that the Player rescued the Hostage: SOUND_GOT_GOODIE.
3. The Hostage must do nothing. (If a Hostage can move around freely, it's not a Hostage anymore!)

What a Hostage Must Do When It Is Attacked

Hostages can't be attacked. Bullets will pass right over them.

The Exit

You must create a class to represent the Exit, the doorway the Player must step on to complete the current main level, but only after collecting all the Jewels on the main level and rescuing all the hostages on all of the sub-levels. Here are the requirements you must meet when implementing the Exit class.

What the Exit Must Do When It Is Created

When it is first created:

1. The Exit object must have an image ID of IID_EXIT.
2. The Exit must always start at the proper location as specified by the current level's data file.
3. The Exit has no direction.
4. The Exit must start out *invisible* when it is created, since it is revealed to the Player only after the Player has collected all of the Jewels on the level. At that point, the Exit object must be made visible, and then the Player can complete the level by stepping on it.
5. The Exit object should be placed at the correct sub_level.

What the Exit Must Do During a Tick

Each time the Exit object is asked to do something (during a tick):

1. If the Player is currently on the same square as the Exit AND the Exit is visible (because the player has collected all of the Jewels on the level), then the Exit must:
    a. Play a sound indicating that the Player finished the level: SOUND_FINISHED_LEVEL.
    b. Inform the *StudentWorld* object that the user is to receive 1000 more points for using the Exit.
    c. Inform the *StudentWorld* object somehow that the Player has completed the current level.
    d. If there are any bonus points left, the Player must receive the bonus points for completing the level quickly. It's you choice whether the *StudentWorld* object or the Exit object grants the points.

What the Exit Must Do When It Is Attacked

The Exit can't be attacked. Bullets will pass right over it.

What the Exit Must Do When It Is Revealed

If the Player collects all of the Jewels and rescued all the hostages on the main level and all of its sub-levels of the game, then the game must make the Exit on the **main** level visible, allowing it to be used by the Player to exit the current main level and advance to the next main level. We restrict that the Exit can only appear on the main level (sub_level =0) instead of any of its sub-levels (those sub-levels that the player entered into by going through a Gate).

You should have your StudentWorld's *move()* method detect whether all the Jewels have been collected and all the hostages have been rescued from the main level and all of its sub-levels, and if so, make the Exit object visible in the main level.

When the Exit is revealed, it must transition from being invisible (it starts in this state) to visible, and must play a reveal sound effect: SOUND_REVEAL_EXIT.

Note: The transition of the Exit on a main level from invisible to visible must happen only once, when the last remaining Jewel on the current level (assuming all hostages have been rescued) has been picked up by the Player. On the subsequent ticks until the Player uses the exit, your program must not play the reveal sound effect again.

Extra Life Goodie

You must create a class to represent an Extra Life Goodie. When the Player picks up this Goodie (by moving onto the same square as it), it gives the Player an extra life! Here are the requirements you must meet when implementing the Extra Life Goodie class.

What an Extra Life Goodie Must Do When It Is Created

When it is first created:

1. The Extra Life Goodie object must have an image ID of IID_EXTRA_LIFE.
2. An Extra Life Goodie must always start at the proper location as specified by the current level's data file.
3. Extra Life Goodies have no direction.
4. Extra Life Goodies object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Extra Life Goodie class, an Extra Life Goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What an Extra Life Goodie Must Do During a Tick

Each time an Extra Life Goodie is asked to do something (during a tick):

1. The Extra Life Goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the Player is on the same square as the Extra Life Goodie, then the Extra Life Goodie must:
    a. Inform the *StudentWorld* object that the user is to receive 500 more points.
    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound effect to indicate that the Player picked up the Goodie: SOUND_GOT_GOODIE.
    d. Inform the *StudentWorld* object that the Player is to gain one extra life.

<u>What an Extra Life Goodie Must Do When It Is Attacked</u>

Extra Life Goodies can't be attacked. Bullets will pass right over them.

Restore Health Goodie

You must create a class to represent a Restore Health Goodie. When the Player picks up this Goodie (by moving onto the same square as it), it restores the Player's health! Here are the requirements you must meet when implementing the Restore Health Goodie class.

<u>What a Restore Health Goodie Must Do When It Is Created</u>

When it is first created:

1. The Restore Health Goodie object must have an image ID of IID_RESTORE_HEALTH.
2. A Restore Health Goodie must always start at the proper location as specified by the current level's data file.
3. Restore Health Goodies have no direction.
4. Restore Health Goodies object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Restore Health Goodie class, a Restore Health Goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What a Restore Health Goodie Must Do During a Tick</u>

Each time a Restore Health Goodie is asked to do something (during a tick):

1. The Restore Health Goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.

2. Otherwise, if the Player is on the same square as the Restore Health Goodie, then the Restore Health Goodie must:
    a. Inform the *StudentWorld* object that the user is to receive 1000 more points.
    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound effect to indicate that the Player picked up the Goodie: SOUND_GOT_GOODIE.
    d. Inform the *Player* object that the Player is to restore itself to full health (i.e., 20 hit points, the initial value).

What a Restore Health Goodie Must Do When It Is Attacked

Restore Health Goodies can't be attacked. Bullets will pass right over them.

Ammo Goodie

You must create a class to represent an Ammo Goodie. When the Player picks up this Goodie (by moving onto the same square as it), it adds 25 rounds of ammunition to the Player ammunition supply. Here are the requirements you must meet when implementing the Ammo Goodie class.

What an Ammo Goodie Must Do When It Is Created

When it is first created:

1. The Ammo Goodie object must have an image ID of IID_AMMO.
2. An Ammo Goodie must always start at the proper location as specified by the current level's data file.
3. Ammo Goodies have no direction.
4. Ammo Goodies object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your Ammo Goodie class, an Ammo Goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What an Ammo Goodie Must Do During a Tick

Each time an Ammo Goodie is asked to do something (during a tick):

1. The Ammo Goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, if the Player is on the same square as the Ammo Goodie, then the Ammo Goodie must:

   a. Inform the *StudentWorld* object that the user is to receive 200 more points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that the Player picked up the Goodie: SOUND_GOT_GOODIE.
   d. Inform the Player that the Player is to add 25 rounds of ammunition.

<u>What an Ammo Goodie Must Do When It Is Attacked</u>

Ammo Goodies can't be attacked. Bullets will pass right over them.

SnarlBot

You must create a class to represent a SnarlBot. Here are the requirements you must meet when implementing the SnarlBot class.

<u>What a SnarlBot Must Do When It Is Created</u>

When it is first created:

1. The SnarlBot object must have an image ID of IID_SNARLBOT.
2. A SnarlBot must always start at the proper location as specified by the current level's data file.
3. A SnarlBot must start out facing either right or down, depending on whether it's a horizontal or vertical SnarlBot.
4. A SnarlBot must start out with 15 hit points.
5. A SnarlBot object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your SnarlBot class, an SnarlBot object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

A SnarlBot, unlike the Player, doesn't necessarily get to take an action during every tick of the game. (This is to make the game easier to play, since if a SnarlBot moved once every tick, it would move much faster than the typical user can think and hit the keys on the keyboard.) A SnarlBot must therefore compute a value indicating how frequently it's allowed to take an action. This value is to be computed as follows:

```
int ticks = (28 – levelNumber) / 4;  // levelNumber is the current
                                     // level number (0, 1, 2, etc.)
if (ticks < 3)
   ticks = 3;  // no SnarlBot moves more frequently than this
```

If the value of *ticks* is 5, for example, then the SnarlBot must "rest" for 4 ticks, perform its normal behavior on the next tick, rest for 4 ticks, perform its behavior on the next tick, etc., performing its behavior every 5th tick.

What a SnarlBot Must Do During a Tick

Each time a SnarlBot is asked to do something (during a tick):

1. The SnarlBot must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the SnarlBot is supposed to "rest" during the current tick, it must do nothing during the current tick other than to update its tick count.
3. Otherwise, the SnarlBot must determine whether it should fire its laser cannon: If the Player is in the same row or column as the SnarlBot AND the SnarlBot is currently facing the Player AND there are no obstacles (specifically, Walls, robots, or robot factories) in the way, then the SnarlBot will fire a bullet toward the Player and then do nothing more during the current tick.

   To fire a Bullet, a new Bullet object must be added at the square immediately in front of the SnarlBot, facing the same direction as the SnarlBot. For example, if the SnarlBot is at x=10,y=7 facing upward, then the Bullet would be created at location x=10, y=8 facing upward. Every time a SnarlBot fires a bullet, it must play the SOUND_ENEMY_FIRE sound effect.

   Hint: When you create a new Bullet object in the proper location and facing the proper direction, give it to your StudentWorld to manage (e.g., animate) along with the other game objects.

4. Otherwise, the SnarlBot will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: the Player, a Wall, a robot, or a KleptoBot factory.
   a. If the square does not contain an obstruction, the SnarlBot will move to it.
   b. Otherwise, the SnarlBot will not move, but instead reverse the direction it's facing (left ←→ right or up←→down).

What a SnarlBot Must Do When It Is Attacked

A SnarlBot should have a method named *damage* or something similar that a Bullet object can call to inform the SnarlBot that it has been damaged. When a SnarlBot is damaged:

1. If its hit points have not reached zero, the game must play the SOUND_ROBOT_IMPACT sound effect.
2. Otherwise, the SnarlBot has been killed, so it must:
   a. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   b. Play a sound indicating that it has died: SOUND_ROBOT_DIE.
   c. Inform the *StudentWorld* object that the user is to receive 200 more points.

KleptoBot

You must create a class to represent a KleptoBot. Here are the requirements you must meet when implementing the KleptoBot class.

What a KleptoBot Must Do When It Is Created

When it is first created:

1. The KleptoBot object must have an image ID of IID_KLEPTOBOT.
2. A KleptoBot must always start at the proper location as specified by the current level's data file.
3. A KleptoBot must start out facing right.
4. A KleptoBot must start out with 9 hit points.
5. A KleptoBot selects a random integer from 1 to 3 inclusive, which we'll call *distanceBeforeTurning*. This is how far the KleptoBot will move in a straight line (if it can) before turning.
6. A KleptoBot object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your KleptoBot class, a KleptoBot object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

A KleptoBot, unlike the Player, doesn't necessarily get to take an action during every tick of the game. (This is to make the game easier to play, since if a KleptoBot moved once every tick, it would move much faster than the typical user can think and hit the keys on the keyboard.) A KleptoBot must therefore compute a value indicating how frequently it's allowed to take an action. This value is to be computed as follows:

```
int ticks = (28 – levelNumber) / 4;  // levelNumber is the current
                                     // level number (0, 1, 2, etc.)
if (ticks < 3)
   ticks = 3;  // no KleptoBot moves more frequently than this
```

If the value of *ticks* is 5, for example, then the KleptoBot must "rest" for 4 ticks, perform its normal behavior on the next tick, rest for 4 ticks, perform its behavior on the next tick, etc., performing its behavior every 5th tick.

What a KleptoBot Must Do During a Tick

Each time a KleptoBot is asked to do something (during a tick):

1. The KleptoBot must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the KleptoBot is supposed to "rest" during the current tick, it must do nothing during the current tick other than to update its tick count.

3. Otherwise, if the KleptoBot is on the same square as a Goodie, and has not yet ever picked up a Goodie, then there is a 1 in 5 chance that it will pick up that Goodie. If it does:
    a. That Goodie must no longer appear on the screen and must not be able to be picked up by the Player until the KleptoBot is killed.
    b. The KleptoBot must play a "munching" sound: SOUND_ROBOT_MUNCH.
    c. The KleptoBot must do nothing more during the current tick.
    d. The KleptoBot can only pick up only one Goodie (one of Ammo, Extra Life Goodies, and Restore Health Goodies). Once the KleptoBot has already picked up one Goodie, it can no longer pick up any other Goodies.

    Hint: You can either (1) make the Goodie invisible and unable to be picked up while the KleptoBot is carrying it, undoing that when the KleptoBot is killed, or (2) have the KleptoBot remember the kind of Goodie it picked up, kill the Goodie object, and when the KleptoBot is killed, create a new Goodie object of that type. When the KleptoBot is killed, the Goodie must appear at the KleptoBot's location.

4. Otherwise, if the KleptoBot has not yet moved *distanceBeforeTurning* squares in its current direction, then it will try to move to the adjacent square in the direction it is facing if that square does not contain an obstruction: the Player, a Wall, a robot, or a KleptoBot factory. If the KleptoBot can move, it must update its location to that square and do nothing more during the curent tick.
5. Otherwise, the KleptoBot has either moved *distanceBeforeTurning* squares in a straight line or encountered an obstruction. In this case, the KleptoBot must:
    a. Select a random integer from 1 to 3 inclusive to be the new value of *distanceBeforeTurning*.
    b. Select a random direction, which we'll call *d*. It's OK if this direction happens to be the same as the KleptoBot's current direction.
    c. Starting by considering the direction *d*, repeat the following until it has either successfully moved or cannot move because of obstructions in all four directions:
        i. Determine whether there is an obstruction in the adjacent square in the direction under consideration. If there is none, the KleptoBot must set its direction to the direction under consideration, move to that square, and then do nothing more during the current tick.
        ii. If there is an obstruction, the KleptoBot must consider a direction it has not already considered during this loop, and repeat step i.

        If there is an obstruction in all four directions, then the KleptoBot must set its current direction to *d* and do nothing more during the current tick.

What a KleptoBot Must Do When It Is Attacked

A KleptoBot should have a method named *damage* or something similar that a Bullet object can call to inform the KleptoBot that it has been damaged. When a KleptoBot is damaged:

1. If its hit points have not reached zero, the game must play the SOUND_ROBOT_IMPACT sound effect.
2. Otherwise, the KleptoBot has been killed, so it must:
    a. If it had picked up a Goodie, make that Goodie appear on the screen and be able to be picked up by the Player.
    b. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound indicating that it has died: SOUND_ROBOT_DIE.
    d. Inform the *StudentWorld* object that the user is to receive 20 more points.

KleptoBot Factory

You must create a class to represent a KleptoBot Factory that manufactures KleptoBots. A KleptoBot Factory churns out new KleptoBots and adds them to the current level according to rules described below. Here are the requirements you must meet when implementing the KleptoBot Factory class.

What a KleptoBot Factory Must Do When It Is Created

When it is first created:

1. The KleptoBot Factory object must have an image ID of IID_ROBOT_FACTORY.
2. A KleptoBot Factory must always start at the proper location as specified by the current level's data file.
3. A KleptoBot Factory has no direction.
4. A KleptoBot Factory is told to produce KleptoBots.
5. A KleptoBot Factory object should be placed at the correct sub_level.

In addition to any other initialization that you decide to do in your KleptoBot Factory class, a KleptoBot Factory object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true).*

What a KleptoBot Factory Must Do During a Tick

Each time a KleptoBot Factory is asked to do something (during a tick):

1. The KleptoBot Factory must count how many KleptoBots are present in the square region that extends from itself 3 squares up, 3 squares left through 3 squares down, 3 squares right. For example, a Factory at x=10,y=8 would count all the KleptoBots in the region bounded by x=7,y=11; x=13,y=11; x=13,y=5; and x=7,y=5, inclusive. A Factory at x=2,y=13 would count the KleptoBots in the region bounded by x=0,y=14; x=5,y=14; x=5,y=10; and x=0,y=10, inclusive, properly accounting for the boundaries of the maze.

2. If the count is less than 3 AND there is no KleptoBot on the same square as the Factory, then there is a 1 in 50 chance that during the current tick, the Factory will create a new KleptoBot that Factory manufactures and add it to the maze on the same square as the Factory. If it creates a new KleptoBot, it must play the SOUND_ROBOT_BORN sound effect.

<u>What a KleptoBot Factory Must Do When It Is Attacked</u>

KleptoBot Factories can't be attacked. Bullets that smack into them do no damage to them. However, if a KleptoBot is on the same square as a Factory (because it was just created by the Factory), then a Bullet that moves onto that square damages the KleptoBot as usual.

How to Tell Who's Who

Depending on how you design your classes, you may find that you need to determine what type of object one of your pointers points to. For example, suppose the Player is directed to move right and needs to decide whether that's prevented by a Wall:

```
Actor* ap = getAnActorAtTheProposedLocation(…);
if (ap != nullptr)
{
    Determine if ap points to a Wall
    ...
}
```

In the code above, the Player calls a function *getAnActorAtTheProposedLocation* that you might write, which returns a pointer to an actor, if any, that occupies the destination square. The Player is allowed to occupy the same square as a Goodie or a Jewel, for example, but not a Wall. But how can we determine whether *ap* points to a Wall? Here's a possible way:

```
Actor* ap = getAnActorAtTheProposedLocation(…);
if (ap != nullptr)
{
    if(ap->isAlive()) // make sure the object is alive
    {
        if(ap->getID() == IID_WALL)// then check its ID
            cerr << "wp points to a Wall" << endl;
        ...
      }
    }
}
```

Although it is not recommended, you can also do this:

```
Actor* ap = getAnActorAtTheProposedLocation(…);
if (ap != nullptr)
{
    Wall* wp = dynamic_cast<Wall*>(ap);
    if (wp != nullptr)
    {
        cerr << "wp points to a Wall" << endl;
        ...
    }
}
```

A C++ *dynamic_cast* expression can be used to determine whether the object pointed to by a pointer of a general type may also be pointed to by an pointer of a more specific type (e.g., whether a Wall pointed to by an Actor pointer can in fact be pointed to by a Wall pointer, or whether a SnarlBot pointed to by an Actor pointer can in fact be pointed to by a Robot pointer, assuming Robot is a base class of SnarlBot). In the example above, if *ap* pointed to a Wall object, then dynamic casting *ap* to a Wall pointer (the target type in the angle brackets) succeeds, and the returned pointer that is used to initialize *wp* will point to that Wall object. If *ap* pointed to a Jewel object, then since a Wall pointer could not point to such an object, the dynamic cast yields a null pointer.

Here's another example in a different problem domain:

```
class Person { ... };
class Faculty : public Person { ... };
class Student : public Person { ... };
class GradStudent : public Student { ... };
class Undergrad : public Student { ... };
class ExchangeStudent : public Undergrad { ... };

Person* p = new Undergrad(...);         // The object is an Undergrad
...
Faculty* f = dynamic_cast<Faculty*>(p); // f is nullptr; an Undergrad is
                                        // not a kind of Faculty
Student* s = dynamic_cast<Student*>(p); // s is not nullptr; an Undergrad
                                        // is a kind of Student
Undergrad* u = dynamic_cast<Undergrad*>(p);  // u is not nullptr; an
                                        // Undergrad is an Undergrad
ExchangeStudent* e = dynamic_cast<ExchangeStudent*>(p);
                                        // e is nullptr; an Undergrad is
                                        // not a kind of ExchangeStudent
```

Note: dynamic_cast works only for classes with at least one virtual function. A base class should always have a virtual destructor, so that's an easy requirement to meet.

Stylistic note: Uses of dynamic_cast should be **rare**. In most cases, when you have a base pointer and you want to do different things depending on which kind of derived object it points to, you'd just call a virtual function declared in the base class and implemented in different ways in the derived classes. For example, we don't have to ask whether an object at the location the Player is directed to move to is a Wall, or a Kleptobot, etc., to determine whether the Player is blocked; instead, the Actor class could have a *blocksPlayer()* method that derived classes can implement in their own way (e.g., Wall's version returns true, Jewel's version returns false, etc.)

Don't know how or where to start? Read this!

When working on your first large object oriented program, you're likely to feel overwhelmed and have no idea where to start; in fact, it's likely that many students won't be able to finish their entire program. Therefore, it's important to attack your program piece by piece rather than trying to program everything at once.

Students who try to program everything at once rather than program incrementally almost always fail to solve CS32's project 3, so don't do it!

Instead, try to get one thing working at a time. Here are some hints:

1. When you define a new class, try to figure out what public member functions it should have. Then write dummy "stub" code for each of the functions that you'll fix later:

   ```
   class foo
   {
     public:
        int chooseACourseOfAction() { return 0; }      // dummy version
   };
   ```

   Try to get your project compiling with these dummy functions first, then you can worry about filling in the real code later.
2. Once you've got your program compiling with dummy functions, then start by replacing one dummy function at a time. Update the function, rebuild your program, test your new function, and once you've got it working, proceed to the next function.
3. **Make backups of your working code frequently. Any time you get a new feature working, make a backup of all your .cpp and .h files just in case you screw something up later.**

BACK UP YOUR .CPP AND .H FILES TO A REMOVABLE DEVICE OR TO ONLINE STORAGE JUST IN CASE YOUR COMPUTER CRASHES!

If you use this approach, you'll always have something working that you can test and improve upon. If you write everything at once, you'll end up with hundreds of errors and just get frustrated! So don't do it.

Building the Game

The game assets (i.e., image, sound, and level data files) are in a folder named *Assets*. The way we've written the main routine, your program will look for this folder in a standard place (described below for Windows and Mac OS X). A few students may find that their environment is set up in a way that prevents the program from finding the folder. If that happens to you, change the string literal `"Assets"` in main.cpp to the full path name of wherever you choose to put the folder (e.g., `"Z:/BoulderBlast/Assets"` or `"/Users/fred/BoulderBlast/Assets"`).

To build the game, follow these steps:

For Windows

Unzip the GoonieBlast-skeleton-windows.zip archive into a folder on your hard drive. Double-click on GoonieBlast.sln to start Visual Studio.

If you build and run your program from within Visual Studio, the Assets folder should be in the same folder as your *.cpp* and *.h* files. On the other hand, if you launch the program by double-clicking on the executable file, the Assets folder should be in the same folder as the executable.

For Mac OS X

Unzip the GoonieBlast-skeleton-mac.zip archive into a folder on your hard drive. Double-click on our provided GoonieBlast.xcodeproj to start Xcode.

If you build and run your program from within Xcode, the Assets directory should be in the directory yourProjectDir*/DerivedData/*yourProjectName*/BuildProducts/Debug* (e.g., */Users/fred/BugBlast/DerivedData/BugBlast/Build/Products/Debug*). On the other hand, if you launch the program by double-clicking on the executable file, the Assets directory should be in your home directory (e.g., */Users/fred*).

What to Turn In

Part #1 (20%)

Ok, so we know you're scared to death about this project and don't know where to start. So, we're going to incentivize you to work incrementally rather than try to do everything all at once. For the first part of Project 3, your job is to build a really simple version of the Goonie Blast game that implements maybe 15% of the overall project. Everything discussed below does not consider sub-levels. In other words, just assume that there is only a main level without sub-levels (by going through the Gate). Anyway, you must program:

1. A class that can serve as the base class for all of your game's actors (e.g., the Player, SnarlBots, KleptoBots, Goodies, Walls, Gates, Jewels, etc.):
   i. It must have a simple constructor.
   ii. It must be derived from our GraphObject class.
   iii. It must have a member function named *doSomething()* that can be called to cause the actor to do something.
   iv. You may add other public/private member functions and private data members to this base class, as you see fit.
2. A Wall class, derived in some way from the base class described in 1 above:
   i. It must have a simple constructor.
   ii. It must have an Image ID of IID_WALL.
   iii. It (or its base class) must make itself visible via a call to *setVisible(true);*
   iv. You may add other public/private member functions and private data members to your Wall class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes).
3. A limited version of your Player class, derived in some way from the base class described in 1 just above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):
   i. It must have a constructor that initializes the Player – see the Player section for more details on where to initialize the Player.
   ii. It must have an Image ID of IID_PLAYER.
   iii. It (or its base class) must make itself visible via a call to *setVisible(true);*
   iv. It must have a limited version of a *doSomething()* method that lets the user pick a direction by hitting a directional key. If the Player hits a directional key during the current tick and the target square does not contain a Wall, it updates the Player's location to the target square and the Player's direction. All this *doSomething()* method has to do is properly adjust the Player's x,y coordinates and direction, and our graphics system will automatically animate its movement it around the maze!
   v. You may add other public/private member functions and private data members to your Player class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes).
4. A limited version of the *StudentWorld* class.
   i. Add any private data members to this class required to keep track of Walls as well as the Player object. You may ignore all other items in the maze such as SnarlBots, the Exit, etc. for part #1.

ii. Implement a constructor for this class that initializes your data members.
iii. Implement a destructor for this class that frees any remaining dynamically allocated data that has not yet been freed at the time the *StudentWorld* object is destroyed.
iv. Implement the *init()* method in this class. It must create the Player and insert it into the maze at the right starting location (see the Level class section of this document for details on the starting location). It must also create all of the Walls and add them to the maze as specified in the current Level's data file.
v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask your Player and Walls to do something. Your *move()* method need not check to see if the Player has died or not; you may assume at this point that the Player cannot die. Your *move()* method does not have to deal with any actors other than the Player and the Walls.
vi. Implement a *cleanup()* method that frees any dynamically allocated data that was allocated during calls to the *init()* method or the *move()* method (e.g., it should delete all your allocated Walls and the Player). Note: Your *StudentWorld* class must have both a destructor and the *cleanUp()* method even though they likely do the same thing (in which case the destructor could just call *cleanup()*).

As you implement these classes, repeatedly build your program – you'll probably start out with lots of errors… Relax and try to remove them and get your program to run. (Historical note: A UCLA student taking CS131 once got 1,800 compilation errors when compiling a 900-line class project written in the Ada programming language. His name was Carey Nachenberg.)

You'll know you're done with part 1 when your program builds and does the following: When it runs and the user hits Enter to begin playing, it displays a maze with the Player in its proper starting position. If your classes work properly, you should be able to move the Player around the maze using the directional keys, without the Player walking through any walls.

Your Part #1 solution may actually do more than what is specified above; for example, if you are further along in the project, and what you have built and has at least as much functionality as what's described above, then you may turn that in instead.

Note, the Part #1 specification above doesn't require you to implement any SnarlBots, KleptoBots, Hostages, Gates, Jewels, any Goodies, or the Exit (unless you want to). You may do these unmentioned items if you like but they're not required for Part 1. **However, if you add additonal functionality, make sure that your Player, Wall, and StudentWorld classes still work properly and that your program still builds and meets the requirements stated above for Part #1!**

If you can get this simple version working, you'll have done a bunch of the hard design work. You'll probably still have to change your classes a lot to implement the full project, but you'll have done most of the hard thinking.

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **must build without errors** under either Visual Studio or Xcode. You do **not** have to get it to run under more than one compiler. You will turn in a zip file containing nothing more than these four files:

Actor.h	// contains base, Player, and Wall class declarations
	// as well as constants required by these classes
Actor.cpp	// contains the implementation of these classes
StudentWorld.h	// contains your StudentWorld class declaration
StudentWorld.cpp	// contains your StudentWorld class implementation

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. Therefore, your solution must NOT modify any of our files or you will receive zero credit! (Exception: You may modify the string literal `"Assets"` in main.cpp.) You will not turn in a report for Part #1; we will not be evaluating Part #1 for program comments, documentation, or test cases; all that matters for Part #1 is correct behavior for the specified subset of the requirements.

Part #2 (80%)

After you have turned in your work for Part #1 of Project 3, we will discuss one possible design for this assignment. For the rest of this project, you are welcome to continue to improve the design that you came up with for Part #1, **or you can use the design we provide**.

In Part #2, your goal is to implement a fully working version of the Goonie Blast game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in your source code for your game, which **must build without errors** under either Visual Studio or Xcode. You do **not** have to get it to run under more than one compiler. You will turn in a zip file containing nothing more than these five files:

Actor.h	// contains declarations of your actor classes
	// as well as constants required by these classes
Actor.cpp	// contains the implementation of these classes

StudentWorld.h // contains your StudentWorld class declaration
StudentWorld.cpp // contains your StudentWorld class implementation

report.doc, report.docx, or report.txt // your report (10% of your grade)

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. Therefore, your solution must NOT modify any of our files or you will receive zero credit! (Exception: You may modify the string literal `"Assets"` in main.cpp.)

You must turn in a report that contains the following:

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Goonie Blast are able to sneeze, and each type of actor sneezes in a different way."
2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I didn't implement the Exit class." or "My KleptoBot doesn't work correctly yet so I treat it like a SnarlBot right now."
3. A list of other design decisions and assumptions you made; e.g., "It was not specified what to do in situation X, so this is what I decided to do."
4. A description of how you tested each of your classes (1-2 paragraphs per class).

FAQ

Q: The specification is silent about what to do in a certain situation. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If neither the specification nor our program makes it clear what to do, do whatever seems reasonable and document it in your report. **If the specification is unclear, but your program behaves like our demonstration program, YOU WILL NOT LOSE POINTS!**

Q: What should I do if I can't finish the project?!
A: Do as much as you can, and whatever you do, make sure your code builds! If we can sort of play your game, but it's not complete or perfect, that's better than it not even building!

Q: Can I work with my classmates on this?
A: You can discuss general ideas about the project, but don't share source code with your classmates. Also don't help them write their source code.

GOOD LUCK!