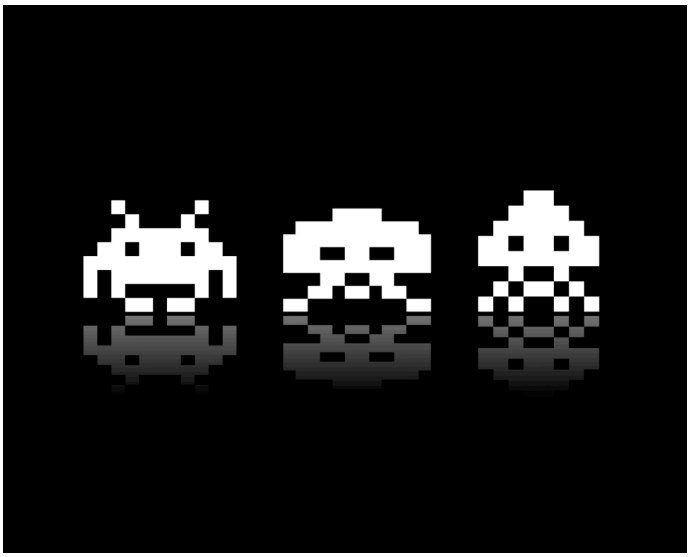
Project 3
Space Inflators

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



Time due:

Part 1: 9 PM, Saturday, February 23
Part 2: 9 PM, Thursday, February 28

[Pages 24 through 38 are undergoing minor editing; you can ignore them for Part 1 anyway.]

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!

Table of Contents

Introduction

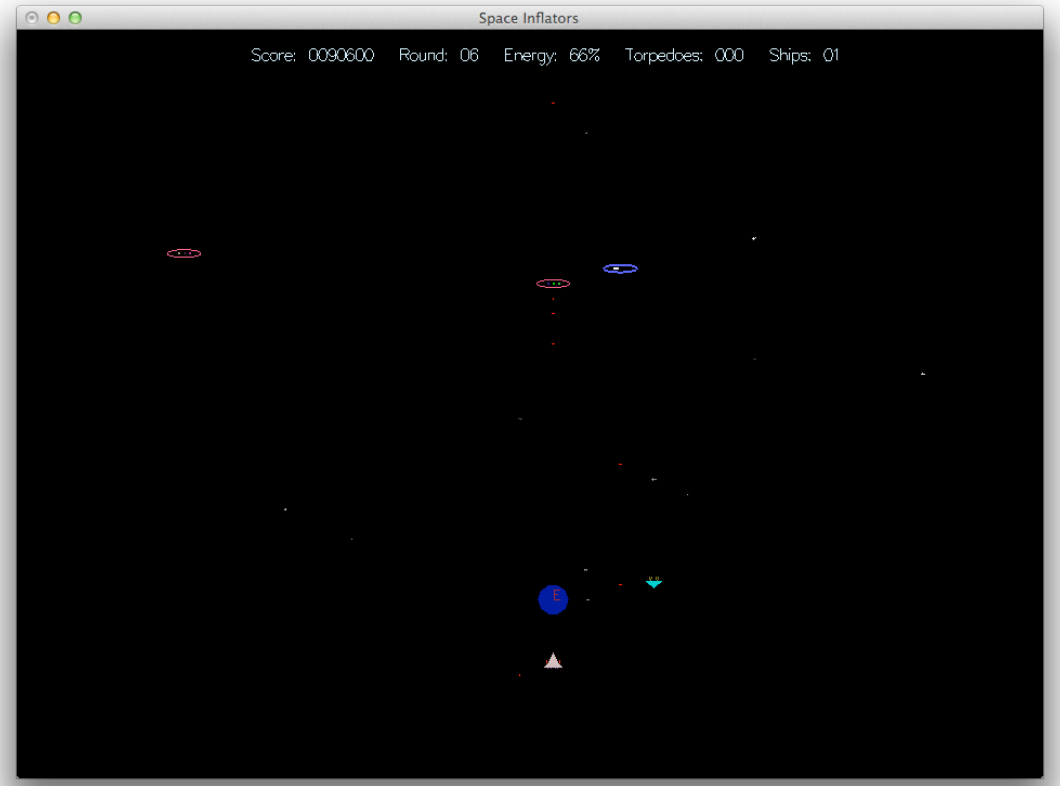
NachenGames corporate spies have learned that SmallSoft is planning to release a new game, called Space Inflators, and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, NachenGames corporate spies have managed to steal a prototype Space Inflators executable file and several source files from the SmallSoft headquarters, so you can see exactly how your version of the game must work (see attached 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.)

Space Inflators is a simplified version of the original Bally Corporation Space Invaders game. In Space Inflators, the Player has to fly through space, attacking enemy Alien space ships ad nauseam, until he or she dies. The goal of Space Inflators is to destroy as many Alien ships as possible.

Upon starting a new game, the Player ship is placed in the lower middle section of the screen, at the bottom of the space field. The Player can use the arrow keys to move their ship left, right, up or down on the screen in order to best position it to battle the invading Aliens. The Player can press the space key to fire Septic Bullets, or if the Player has one or more Flatulence Torpedoes they can fire one by hitting the tab key. To kill an Alien ship requires a varying number of shots depending on the life energy of the Alien. Similarly, the Player has a limited amount of life energy which is depleted as the Player gets hit by enemy Bullets or Torpedoes, or collides with enemies.

There are three different types of Aliens in Space Inflators: Nachlings, Wealthy Nachlings and Smallbots. None of the Aliens are particularly nice. Nachlings are a poor, bitter race of Alpha Centauri Aliens who mostly like to fling Septic Bullets at the Player. Wealthy Nachlings are an offshoot race that have similar behavior to their poorer cousins, but also occasionally drop expensive Goodies that the Player might find useful. Finally, The Smallbots are an aggressive cyborg race (half organic, half robot) based on the planet Uranus that are notoriously trigger happy. Smallbots have discovered the secret to everlasting life (i.e., an extra-life bonus) and occasionally drop such a Goodie when they die. It would do well for the Player to pick these up or face certain death.

Here is an example of what the Space Inflators game looks like:



Game Details

The Player starts out a new game with 3 lives and continues to play until all of his/her lives have been exhausted. There are multiple *rounds* or levels in Space Inflators – during each round, the Player must destroy a specific number of Alien ships in order to move on to the next round.

The Space Inflators space field is exactly 30 squares wide by 40 squares high. The bottom-leftmost square has coordinates x=0,y=0, while the upper-rightmost square has x=29,y=39. Notice that x increases to the right and y increases upward. You can look in our provided file, GameConstants.h, for constants like the space field's width and height.

When a new game begins (or when the Player continues playing their game after losing a ship), the space field is initialized as follows:

1. The space field starts out empty.
2. The Player ship is placed in the space field at location x=15, y=1.
3. At the start of the game or when continuing a game after losing a life, NO Aliens or Stars initially appear on the screen.

Once a new space field has been prepared and the Player ship is in its proper position, the game play begins. Game play is divided into *ticks*, and there are dozens of ticks per second (to provide smooth animation and game play). During each tick, the following occurs:

1. One new Alien *may possibly* be added to the space field (see sections below for details on this).
2. One new Star *may possibly* be added to the space field (see sections below for details on this)
3. The Player has an opportunity to move their ship exactly one square horizontally, move it one square vertically, or fire a Septic Bullet or a Flatulence Torpedo.
4. Every other actor in the space field, if any, has an opportunity to do something. For example, each Alien ship can move one square (left, right or possibly diagonally) according to their built-in movement algorithms (described in the various Alien sections below). Each Alien may also perform some other action; for example, a Smallbot might fire a Flatulence Torpedo at the Player.

During game-play, the user controls the direction of their Player ship 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 ship may move anywhere in the space field.

The Player may shoot a Septic Bullet by pressing the space key, or, it they have any, a Flatulence Torpedo by pressing the Tab key or the *t* key. Once fired, a Bullet or Torpedo shoots up from the square just above the Player and continues until it either hits an Alien or until it reaches the top of the space field (y=39). An Alien destroyed by a Bullet or Torpedo *fired by the Player* earns the Player points:

For destroying a Nachling:	1000 points
For destroying a Wealthy Nachling:	1200 points
For destroying a Smallbot:	1500 points

The Player also earns 100 points (and special benefits) by picking up a Goodie that a Wealthy Nachling or a Smallbot might drop when it dies.

During each tick, each of the Aliens and the Player ship decides how to move. Once each character in the game has updated its coordinates based on its movement algorithm, our provided code will animate all of the game's actors onto the computer screen for the user to see. (You don't need to worry how our code works, unless, of course, you are curious.) Then the next tick begins, each of the characters decides how to move and update its coordinates, and then they are again animated to the screen, etc.

The Player starts with 50 life force points (i.e., hit points). These life points are reduced when the Player is hit by a Bullet or a Torpedo or runs into an Alien ship (or an enemy ship runs into the Player). When a Player ship loses all of its life force points, 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 to finish up the current round. Once the round restarts, all of the old actors (e.g., Aliens, Stars and Goodies) disappear from the space field. The Player ship is returned to its home position in the space field and given full life points (and *no* torpedoes, if the Player had any before dying), and game play continues. If the Player ship is killed and has no lives left, then the game is over. Pressing the *q* key lets you quit the game prematurely.

The game is divided into rounds, starting with round 1 (not zero). In order for the Player to advance to the next round, the Player must destroy exactly N=4*RoundNumber Alien ships. For example, the Player must destroy 4 Alien ships before advancing from round 1 to round 2. To advance from round 2 to round 3, the Player must destroy 8 Alien ships, and so on. To ensure that the game does not become too difficult too quickly, there is a cap on the number of Alien ships that may be in the space field simultaneously at any particular time. This number may be computed as follows: S = int(2+.5*RoundNumber). For example, during round 1, no more than 2 Alien ships may be on the screen at any one time. If there are two Aliens on the space field during round 1, a new Alien may not be added until the player destroys at least one of these existing Aliens. During round 2, there must be no more than 3 active Aliens on the screen at a time, and so on.

Each of the Aliens moves according to a different algorithm. A full listing of all Alien behaviors is provided in the sections below.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects; in Space Inflators, those objects include Nachlings, Wealthy Nachlings, Smallbots, Free Ship Goodies, Energy Goodies, Torpedo Goodies, Stars, Bullets, Torpedoes, and the Player ship. Each object has its own x,y location in the space field, its own internal state (e.g., the location of the Alien, what direction the Alien is moving, its life energy level, etc.) and its own special algorithm to 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 ship, the algorithm that controls the Player ship object is the user's own brain and hand, and the keyboard!

Once a game begins, it is divided into *ticks*. A tick is a unit of time, for example, 50 milliseconds. During a given tick, every actor (e.g., Nachlings, Stars, Septic Bullets, and the Player ship) in the game has an opportunity to do something (e.g., move to an adjacent square in the space field, shoot a Bullet, die, disappear off the edge of the screen, etc.). During each tick, the user may hit a key to control the Player ship in the game (e.g., to move left, right, up, or down, or to fire). As each actor runs its algorithm during each tick, in addition to altering the actor's own state (e.g., its location), the actor may also affect other actors in the game. For example, a Nachling might fire a Bullet, adding a new actor (a Bullet) to the space field. Or, a Smallbot might run into the Player ship and damage it. So an actor can affect not only its own state, but the state of the World and of other actors within it.

After the current tick is over and all actors have had a chance to adjust their state, they are then animated onto the screen in their new configuration.

Then, the next tick occurs, and each object is again allowed to do something, 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 Nachling 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 like 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 the actors (which are C++ objects) and placing them in the game world so that they will appear in the space field.

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 or fly off-screen will be removed from the game world and deleted.

Cleanup: The Player has lost a life or the game is over. This phase frees all of the objects in the World (e.g., Nachlings, Bullets, etc.) since the level has ended. If game-play is not over (i.e., the user has more lives), then the game proceeds back to the *Initialization* step, where the space field is repopulated with new occupants and game play continues where it left off.

Here is what the main logic of a video game looks like, in pseudocode (we provide some similar code for you in 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 func

   while (The Player's life energy is greater than zero)
   {
         // each pass through this loop is a tick (1/20th of a sec)

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

         // we write this code to handle the animation for you
         Animate_all_of_the_alive_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
}

Tell_the_user_the_game_is_over(); // "game over dude"; we wrote this
```

And here is what the `Ask_all_actors_to_do_something()` function might look like (which is one of the functions you'll have to write):

```
void Ask_all_actors_to_do_something()
{
   for (int i = 0; i < m_actors.size(); i++)
       if (m_actors[i]->isStillAlive())
           m_actors[i]->doSomething();
}
```

As you can see, you have a container (an array or vector in this example, but a list would also be fine) of actor pointers. Each actor has a *doSomething* method. In this method, each actor (e.g., an Alien like a Nachling) can decide what to do. For example, here is some pseudo code showing what a (simplified) Smallbot might decide to do each time it gets asked to do something:

```
class Smallbot: public SomeOtherClass
{
   public:
       void doSomething()
       {
           If I was just hit by a player projectile (e.g., a
           bullet or a torpedo), then
                Move randomly either one space left or one space right

           Always advance one space downward (toward the player)

           If the player is in my line of sight, then
                Fire a bullet or torpedo
       }
       ...
};
```

And here's what the Player's doSomething method 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 I am not at the screen top
                  Adjust my y location UP one square
              ...
              If the user pressed the space bar to fire, then
                  Add a new Septic Bullet just above the Player ship

              If the Player moves into the same square as an Alien, then
                  Damage the Player ship appropriately
                  Damage the Alien ship appropriately
        }
        ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Space Inflators game. Your classes must work properly with our provided classes, and **you must not modify our classes or our sources 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 space field) and all of the objects (Aliens, Stars, Bullets, Torpedoes, Goodies, and the Player ship) that are inside the space field.
2. You must create a class to represent the Player ship in the game.
3. You must create classes for Nachlings, Wealthy Nachlings, Smallbots, Stars, Bullets, Torpedos, Free Ship Goodies, Torpedo Goodies, and Energy Goodies as well as any additional base classes (e.g., an Alien base class if you need one) that are required to implement the game.

You Have to Create the StudentWorld Class

The StudentWorld class is responsible for orchestrating virtually all game play – it keeps track of the whole game world (the space field and all of its inhabitants such as Aliens, the Player, Stars, Bullets, etc.). It is responsible for initializing the game world at the start of the game, asking all of the actors to do something during each tick of the game, and destroying all of the actors in the game world when the user loses a life or when actors disappear (e.g., an Alien ship flies off the screen or dies due to being fired upon).

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

The init() method is responsible for initializing the space field (creating the Player and placing it in its proper location in the space field). This method is automatically called by our provided code either when the game first starts or after the user loses a life (but has more lives left) and the game play is ready to resume.

Each time the move() method is called, it runs a single tick of the game; as such, the move method is called over and over during the game, dozens of times per second. This means that it is responsible for asking each of the game actors (e.g., the Player, Aliens, Goodies, etc) to try to do something: e.g., move themselves and/or perform their specified behavior. This method also occasionally introduces new Aliens or Stars into the space field – for instance, a Smallbot might be introduced into the game at a random tick

during game play to cause trouble for the Player. Finally, this method is responsible for disposing of game objects (e.g., Stars, Aliens, etc) that happen to die or fly off the screen during a given tick. For example, if the Player shoots and destroys a Wealthy Nachling, then the after all of the actors in the game get a chance to do something during the tick, the move() method should remove that Wealthy Nachling from the game world (by deleting its object and removing any reference to the object from the StudentWorld's data structures). Your move() method will automatically be called once during each tick of the game by our provided game framework.

The cleanup() method is called when the Player loses a life (i.e., loses all of its life energy). It is responsible for freeing all game objects (e.g., Aliens, Stars, Torpedoes, the Player object, Goodies, etc.) that are currently active in the game at the time when the Player lost the life. This includes all actors created during either the init() method or introduced during subsequent game play in one of the many calls to the move() method that have not yet been removed from the game.

You may add as many other public/private methods and private member variables to your StudentWorld class as you like.

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

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

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 user's current score.

increaseScore() is used by your StudentWorld class (or you other classes) to increase the user's score upon successfully destroying an Alien or picking up a Goodie of some sort. When your code calls this method, you must specify how many points the user gets (e.g., 1000 points for destroying a Nachling). This means is that the game score is controlled by our GameWorld object – you must *not* maintain your own score member variable 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 Round: 05 Energy: 100% Torpedoes: 000 Ships: 03

getKey() can be used to determine if the user has hit a key on the keyboard to move the Player ship or to fire. 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 filled in with the key that was pressed by the user (if any was pressed). If the Player does hit a key, 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_TAB
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., the Player ship fires a Bullet or an Alien ship dies). You can find constants (e.g., SOUND_ENEMY_HIT) that describe what noise to make inside of the GameConstants.h file. Here's how this method might be used:

```
// if the bullet's coords are the same as the Alien,
// then make a "boom" sound
if (bullet_x == alien_x && bullet_y == alien_y)
      studentWorldObject->playSound(SOUND_ENEMY_HIT);
```

getTestParam() can be used to get the value of a test parameter provided by the CS32 grader. Our CS32 graders don't have time to play each student's game for hours on end to check for bugs. Therefore, we are providing you with special settings that your code must use in order to help facilitate testing. For example, the grader may want to test your project with just a single Nachling flying down the screen at a time. If so, they will specify a test parameter that tells your StudentWorld class to do this. Your code can use the *getTestParam()* method to obtain these grader-provided parameters. For example, here's how your StudentWorld class's *move()* method could determine if the grader wanted you just to insert a single actor in the game for testing:

```
int whichActor = getTestParam(TEST_PARAM_ACTOR_INDEX);
if (whichActor == TEST_PARAM_NACHLING)
      add_a_new_nachling_to_the_game_if_the_space_field_is_empty();
```

Constants like `TEST_PARAM_ACTOR_INDEX` and `TEST_PARAM_NACHLING` are defined in GameConstants.h.

Don't worry - the rest of the spec will specify, in detail, what grader parameters you are responsible for in your code.

The *testParamsProvided()* method can be used to determine if the grader provided any testing parameters at all. This method returns true or false. If this method returns false, then you can assume that normal game play is taking place (i.e., a regular user is playing your game). If the method returns true, then the grader has specified a special setting that must be honored by your code (meaning that your program is being tested and must exhibit slightly different behaviors to facilitate testing).

init() Details

Your *init()* method must initialize the data structures used to keep track of your game world and then allocate and insert a valid Player ship object into the game world.

It is *required* that you keep track of all of your actors (e.g., Nachlings, Stars, TorpedoGoodies, etc.) in a single STL collection like a list or vector (e.g., a list of pointers to actors). If you like, a StudentWorld may keep a separate pointer to the Player ship rather than keeping the Player ship in this collection along with the other actors.

You should 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 a game continues after the user has lost a life, but has more lives left).

Adding the Player ship to the Game World

Your *init()* method must place the Player ship in the space field. The Player ship must start in location x=15, y=1 at the bottom-middle of the space field.

The Player ship is the only actor that may start in the space field once the init() function completes. Other actors (e.g., Stars, SmallBots, etc) will be introduced to the game by the *move()* method (or a method of your choosing called by your *move()* method).

move() Details

The move() method must perform three different activities:

1. It must determine if it's time to add new actors into the game (e.g., introduce a new Smallbot, Nachling, Wealthy Nachling or Star), and if so, add them to the game world.
2. It must ask each of the actors that is currently in the game world to do something (e.g., ask a Nachling to move itself).
3. It must then delete any actors that have died during this tick (e.g., a Nachling that was destroyed by a Septic Bullet and so should be removed from the game world, or a Star that flew off the bottom of the space field into oblivion).
4. The method must then determine if the Player died during the current tick (e.g., because an Alien ran into it or shot it, or because it ran into one of them).

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

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
```

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. The second return value indicates that the tick completed without the Player dying, and therefore that game play should continue normally for the time being (i.e., in another tick, your *move()* method will be called again).

Here's pseudocode for your move() method:

```
int StudentWorld::move()
{
      // Add new Aliens or Stars
    addAliensOrStars();    // add any Aliens/Stars to the space field as
                           // required (see below for details)

      // Update the Game Status Line
    updateDisplayText();   // update the score/lives/round at screen top

      // The term "actors" refers to all Aliens, the Player's ship, Stars,
      // and any projectiles that are still active

      // Give each actor a chance to do something
    for each of the actors in the game world
    {
        if (actor[i] is still active/alive)
        {
              // ask each actor to do something (e.g. move)
            actor[i]->doSomething();
        }
    }

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

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

    return GWSTATUS_CONTINUE_GAME;
}
```

Adding New Aliens and Stars

You must use the following algorithm to add new actors to the game world in your *move()* method:

1. If the grader provided a test parameter (you can check to see if the grader specified testing parameters using the *testParametersProvided()* method

described above), then the grader wants to test your actors one at a time. You must do the following:

   a. If there is already at least one non-Player object in the space field (e.g., a Smallbot or an Energy Goodie), then you must add no new actors during this tick.
   b. Otherwise, get the value of the `TEST_PARAM_ACTOR_INDEX` parameter by calling `getTestParam(TEST_PARAM_ACTOR_INDEX).`
   c. If the value of this parameter is `TEST_PARAM_NACHLING` then you must add a new Nachling to the space field (details on where the Nachling should start in the space field are below).
   d. If the value of this parameter is `TEST_PARAM_WEALTHY_NACHLING` then you must add a new Wealthy Nachling to the space field (details on where the Wealthy Nachling should start are below).
   e. If the value of this parameter is `TEST_PARAM_SMALLBOT` then you must add a new Smallbot to the space field (details on where the Smallbot should start are below).
   f. If the value of this parameter is `TEST_PARAM_GOODIE_ENERGY` then you must add a new Energy Goodie to the space field (at location x=15,y=39).
   g. If the value of this parameter is `TEST_PARAM_GOODIE_TORPEDO` then you must add a new Torpedo Goodie to the space field (at location x=15,y=39).
   h. If the value of this parameter is `TEST_PARAM_GOODIE_FREE_SHIP` then you must add a new Free Ship Goodie to the space field (at location x=15,y=39).
   i. At this point, your method must not add any additional actors at all during this tick (*even* if no actors were added in steps a-h above).

2. If the grader did **not** provide test parameters, then you must add a new actors as follows:

   a. Count the number of active Aliens currently in the space field:
   ```
   numberOfActiveAliens =
       somethingThatDeterminesTheNumberOfActiveAliens()
   ```
   b. If the number of active Aliens in the space field is equal or greater to the maximum allowed number of simultaneous Aliens in the space field for the current round, then do not any new Aliens during this tick (skip to step f).
   c. Otherwise, compute the following value v, which measures how many more Aliens need to be killed before the current round is complete:
   ```
   v = totaAliensOnScreen - numKilledAliensThisRound
   ```
   d. If v is less than or equal to numberOfActiveAliens, then do not add any new Aliens during this tick and skip to step f (since the space field already contains enough Aliens for the Player to kill to complete the current round).
   e. Otherwise, there is a 100% chance that you will add some type of new Alien to the space field:
      i. Compute a random probability p1 between 0% and 100% and use this to decide whether to add either some type of Nachling or a Smallbot:

ii. If the probability p1 is less than 70%, you must add some type of Nachling to the space field.
    1. Compute another random probability p2
    2. If this new probability p2 is less than 20%, then you must add a Wealthy Nachling to the space field. Each new Wealthy Nachling should have a starting energy level of 8 times the current round number. To determine where to add your Wealthy Nachling, see the section on Wealthy Nachlings below.
    3. Otherwise you must add a regular Nachling to the space field. Each new regular Nachling should have a starting energy level of 5 times the current round number. To determine where to add your Nachling, see the section on Nachlings below.
iii. Otherwise, you must add a Smallbot. Each new Smallbot should have a starting energy level of 12 times the current round number. To determine where to add your Smallbot, see the section on Smallbots below.

f. Compute a random probability p3 between 0% and 100%
g. If the probability p3 is less than 33% (a 1 in 3 chance), then add a new Star to the space field. To determine where to add the Star, see the section on Stars below.

Give Each Actor a Chance to Do Something

Actors include the Player ship, Nachlings, Wealthy Nachlings, Smallbots, Bullets, Torpedos, three kinds of Goodies, and Stars. During each tick of the game, after first possibly adding new actors into the space field, each active actor must have an opportunity to do something.

Your move() method visit each active actor in the space field (i.e., held by your StudentWorld object) and ask it to do something by calling a method named doSomething(). In each actor's doSomething() method, the object will have a chance to perform some activity: e.g., move, shoot, disappear from the space field, etc.

It is possible that one actor (e.g., the Player) may destroy another actor (e.g., a Smallbot) during the current tick. If an actor has died earlier in the current tick by another actor, 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 any 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 them. So if, for example, the Player shoots and destroys a Nachling, then the Nachling's object *pointer* should be removed from your StudentWorld's container of active objects and the object it points to should be deleted to free up room in memory for future actors that will be introduced later in the game. Similarly, if any actor flies off the bottom of the screen (e.g., hits Y<=0), then it should similarly be removed from the space field. (Hint: Each of your actors could have a member variable indicating whether or not is 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 your StudentWorld's move() method:

```
void setDisplayText()
{
   int score = getCurrentScore();
   int round = getCurrentGameRound();
   double energyPercent = getPercentageOfEnergyThePlayerHasLeft();
   int torpedoes = getNumOfTorpedoesPlayerHasLeft();
   int shipsLeft = getNumOfShipsPlayerHasLeft();

   // Next, create a string from your statistics, of the form:
   //  "Score: 0000123  Round: 01  Energy: 98%  Torpedoes 003  Ships: 03"
   // where all numeric values except the Energy value must have leading
   // zeros (e.g., 003 vs. 3), and each
   // statistic should be separated from the last statistic by two
   // spaces.  E.g., between the "0000123" of the
   // score and the "R" in "Round" there must be two spaces. Each field
   // must be exactly as wide as shown,
   // e.g., the score must be exactly 7 digits long, the torpedo field
   // must be 3 digits long, except for the
   // Energy field, which could be between 1 and 3 digits (e.g., 5%, 89%
   // or 100%)

   string s = someFunctionYouUseToFormatThingsNicely(score, round,
                                  energyPercent, torpedoes, shipsLeft);

   // Finally, update the display text at the top of the screen with your
   // newly created stats
   setGameStatText(s);     // calls GameWorld::setGameStatText
}
```

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the Player's life energy went to or below zero and the Player lost a life. In this case, every actor in the entire space field (the Player and every Alien, Goodie, Star, etc.) must be deleted and removed from your StudentWorld's container of active objects, resulting in an empty space field. If the user has more lives left, our provided code will subsequently

call your *init()* method to repopulate the space field and the game will then continue 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.

You Have to Create the Classes for All Actors

The Space Inflators game has a number of different actors, including:

- The Player ship
- Nachlings
- Wealthy Nachlings
- Smallbots
- Septic Bullets
- Flatulence Torpedos
- Torpedo Goodies
- Energy Goodies
- Free Ship Goodies
- Stars

Each of these actors can occupy the space field and interact with other actors within the space field.

Now of course, many of your actors will share things in common – for instance, every one of the objects in the game (Nachlings, the Player, Stars, etc) has an x,y coordinate. They all have the ability to perform an action during each tick of the game. Many of them can be potentially attacked (e.g., by a bullet, or by colliding with the Player) and could die during a tick. All of them need some attribute that indicates whether or not they are still alive (or died during the current tick), etc.

It is therefore your job to determine the commonalities between your different actors and make sure to factor out common behavior 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 largely depend upon your ability to create an intelligent set of classes that follow good object-oriented design principles. Your classes must never duplicate functions or member variables – 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.

HINT: When you notice this spec repeating the same text nearly exactly in the following sections (e.g., in the Torpedo Goodie section and the Free Life Goodie section, or in the

Bullet and Torpedo 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 actors directly or indirectly from a base class that we provide called GraphObject, e.g.:

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

class Alien: public Actor
{
public:
	…
};

class Nachling: public Alien
{
public:
	…
};
```

GraphObject is a class that we have defined that helps hide a lot of 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 in your classes:

```
GraphObject(int imageID, int startX, int startY); // the constructor
void setVisible(bool shouldIDisplay);
void setBrightness(double brightness);
void getX() const;
void getY() const;
void moveTo(int x, int y);
```

You may use any of these methods in your derived classes, but you must not use any other methods 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) is the constructor for a new GraphObject. When you construct a new GraphObject, you must specify an image ID which indicates how the GraphObject should be displayed on screen (e.g., as a Nachling, a Player ship, a Star, etc.). You must also apecify 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 and row starting from

bottom left, not row and column.) One of the following IDs, found in GameConstants.h, must be passed in for the imageID value:

```
IID_PLAYER_SHIP
IID_NACHLING
IID_WEALTHY_NACHLING
IID_SMALLBOT
IID_BULLET
IID_TORPEDO
IID_FREE_SHIP_GOODIE
IID_ENERGY_GOODIE
IID_TORPEDO_GOODIE
IID_STAR
```

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. New GraphObjects start out with a brightness level of 1.0 (normal brightness).

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 Nachling image will be drawn to the screen at the GraphObject's specified x,y coordinates if the object's Image ID is IID_NACHLING). If you call *setVisible(false)* then your GraphObject will not be displayed on the screen. When you create a new actor, always remember to call the *setVisible()* method with a value of true or the actor won't display on screen!

setBrightness(double brightness) is used to specify the relative brightness of a GraphObject on the screen. A passed-in value of 1.0 indicates that the object is at full brightness, while a passed-in value of 0.0 indicates that the object is totally blackened out. A value of 0.5 would therefore indicate that the object is half its usual brightness. You should use this method to adjust the brightness of your Goodie objects as they fall downward on the screen; they get darker over time after they're dropped by an Alien until they completely disappear.

getX() and *getX()* are used to determine a GraphObject's current location in the space field. Since each GraphObject maintains an x,y location, this means that your derived classes MUST NOT also have x,y member variables, but instead use those in the GraphObject base class.

moveTo(int x, int y) is used to update the location of a GraphObject within the space field. For example, if a Nachling'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 GameObject in the game 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).

Stars

Here are the requirements you must meet when implementing the Star class:

Basic Star Requirements

1. All Stars have an image ID of IID_STAR.
2. Each Star starts out at the top of the screen in a random column, e.g., x = a random column between 0 and VIEW_WIDTH-1, inclusive, and at y = VIEW_HEIGHT-1.
3. Stars cannot be attacked (shot) by the Player or the Aliens, and cannot collide with other actors. In other words, they do not interact with any other actor in the game. They just move downward until they reach the bottom edge of the space field; at this point they disappear from the game.

What a Star Must Do During a Tick

During each tick of the game (e.g., in the Star's doSomething() method), each Star simply moves down one row on the screen (toward y=0).

When does a Star Die?

A Star must be removed from the space field when its y is less than zero (when it has fallen off the bottom of the screen); in other words, it should die at this point and be removed from any data structures used by your StudentWorld that track the object.

The Player Ship

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

Basic Player ship Requirements

1. The Player ship must have an image ID of IID_PLAYER_SHIP.
2. The Player ship must always start at location x=VIEW_WIDTH/2, y=1
3. The Player ship has life energy (also know as hit points). This life energy must start out at 50 energy points when a new Player ship is created.

What the Player ship Must Do During a Tick

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

1. The Player ship must check to see if it's collided (e.g., is at the same x,y coordinate) with an Alien ship of any kind.

2. If the Player ship has collided with one or more Aliens, then for each Alien it has collided with:
   // ** *Collision Code is in RED below*
   a. The Player ship must have its life energy drained by 15 points - it can use its own Damage method (see below) to inflict this damage and play the appropriate sound effects.
   b. The Alien ship that collided with the Player ship must die immediately (e.g., have its life energy set to zero) regardless of the Alien's current life energy. The Player does NOT get any points for destroying an Alien ship in this way. This will NEVER cause an Alien ship to drop a Goodie. An Alien ship destroyed in this way does not count towards the number of destroyed Aliens required to complete the current round.
3. If the Player ship has reached zero or lower life energy, then the *doSomething()* method must return immediately and do no further activity; the Player is DEAD!
4. Otherwise, the *doSomething()* method must check to see if the user pressed a key. If the user pressed a key:
   a. If the user asked to move down and the Player's y location is > 0, then they should move down one square using the GraphObject moveTo() method.
   b. If the user asked to move up and the Player's y location is < 39, then they should move up one square using the GraphObject moveTo() method.
   c. If the user asked to move left and he Player's x location is > 0, then they should move left one square using the GraphObject moveTo() method.
   d. If the user asked to move right and the Player's x location is < 29, then they should move right one square using the GraphObject moveTo() method.
   e. If the user pressed the space bar and did NOT fire a projectile (e.g., Bullet or Torpedo) during the previous tick of the game (the user may never fire during two successive ticks; they must wait at least one tick after firing before they may fire again) then you must:
      i. Add a new Septic Bullet to the space field at playerx, playery+1 (just above the Player ship).
      ii. Use the GameWorld's playSound method to play a SOUND_PLAYER_FIRE sound effect.
      iii. Remember that the Player fired during the current tick so you can prevent them from firing during the next tick.
   f. If the user pressed the TAB key (or the t key) and did NOT fire a projectile during the last tick of the game (the user may never fire during two successive ticks; they must wait at least one tick after firing before they may fire again) then you must:
      i. Add a new Torpedo to the space field at playerx, playery+1 (just above the Player ship).
      ii. Use the GameWorld's playSound method to play a SOUND_PLAYER_TORPEDO sound effect.
      iii. Remember that the Player fired during the current tick so you can prevent them from firing during the next tick.

5. The Player must then check AGAIN (after it has adjusted its x,y location) to see if it collided with an Alien ship. You must follow the ** Collision Code algorithm above again.

When does a Player ship Die?

A Player ship dies when its life energy is <= 0.

The Player Class Must Have a damage() Method that will be Called when the Player is Hit by a Bullet/Torpedo or Collides with an Alien

Any time a Bullet or Torpedo fired by an Alien hits the Player ship or an Alien collides with the Player ship, the Player's object must be informed of this collision (and the resulting damage from it). You should add a *damage()* method to your Player class to deal with this case. In this method:

1. The method should deduct the appropriate number of life points from the Player ship, and, if necessary, update the status of the Player ship to dead (if the Player's life energy is <= 0).
2. The Player ship must use the GameWorld's *playSound()* method to play a SOUND_ENEMY_PLAYER_COLLISION sound effect if the Player collided with an Alien ship.
3. The Player ship must use the GameWorld's *playSound()* method to play a SOUND_PLAYER_HIT sound effect if the Player was hit by a Bullet or Torpedo.
4. The Player ship must also use the GameWorld's *playSound()* method to play a SOUND_PLAYER_DIE sound effect if the Player ship has died due to this impact (e.g., its life energy has gone <= 0 points)

Getting Input From the User

Since Space Inflators is a *real-time* game, you can't use the typical *getline* and *cin* approach to get a user's key press within the Player ship's *move()* method. This would stop your program and wait until the user types in the proper data and hits the enter key. This would make for a really boring Space Inflators game (requiring the user to hit a directional key then hit Enter, then hit a direction key, then hit Enter, etc). Instead, you will need to use a special function that we provide in our GameWorld class (which your StudentWorld class is derived from) called `getKey()` to get input from the user[1]. This function rapidly checks to see if the user hit a key. If the user hit a key, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the

[1] Hint: Since your Player ship class will need to access the getKey() method in the GameWorld class (which is the base class for your StudentWorld class), your Player ship class (or more correctly, one of its base classes) will need to have a pointer to the StudentWorld class. If you look at the code example below, 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.

function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void Player::doSomething()
{
    ...
    char 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 left ...;
            break;
               // etc…
        }
    }
    ...
}
```

The Septic Bullet

Here are the requirements you must meet when implementing the Septic Bullet (likely inside a Bullet class, derived in some way from our GraphObject class):

Septic Bullets are fired by the Player ship to kill Aliens, or by the Aliens to kill the Player ship.

Basic Bullet Requirements

When the Player ship fires a Bullet, the Player's code must insert a Septic Bullet into the space field one square above their ship. Once the Bullet has been fired, it will proceed to move upward until it hits an Alien or flies off the screen (and dies). When an Alien ship fires a Bullet, their code must insert a Septic Bullet into the space field in the square below the Alien ship. Once the Bullet has been fired, it will proceed to move downward until it hits the Player ship or flies off the screen. You must not have multiple different Bullet classes (one for Player-fired Bullets and one for Alien-fired Bullets) – you should be able to implement both types of Bullets in a single class.

Each Septic Bullet object must have an image ID of IID_BULLET.

What a Bullet Must Do During a Tick

Each time the Septic Bullet is asked to do something (during a tick), it should:

1. Determine if it has collided with a target. Player-fired Bullets may only collide with (and injure) Alien ships. Similarly, Alien-fired Bullets may only collide with (and injure) the Player ship. Alien-fired Bullets will pass harmlessly by other Alien ships. All Bullets will pass harmlessly by Stars and Goodies.

2. If the Bullet collided with one or more VALID targets (e.g., a Player-fired Bullet landed on the same square as an Alien):
    a. The Bullet must damage the target by reducing its life energy by 2 points (by calling the target's Damage method)
    b. If the Bullet simultaneously impacts multiple targets, it must damage all targets by 2 points (e.g., what if multiple Aliens are on the same square and a Player-fired Bullet arrives on that square)
    c. The Bullet must then set its status to "dead" so it will be removed from the space field at the end of the current tick.
3. The Bullet, if it is still alive (i.e., it didn't collide in step #1), must update its location by one square (moving either up or down one square, depending on who fired it) using the GraphObject's *moveTo()* method.
4. If the Bullet moved and is still alive, it must then AGAIN check to see if it collided with a target and take the appropriate action, if so. (See step #2)

When does a Bullet Die?

A Bullet dies when it either impacts a valid target (e.g., an Alien or the Player), or it flies off the top or bottom of the space field.

The Flatulence Torpedo

Here are the requirements you must meet when implementing the Flatulence Torpedo (derived in some way from our GraphObject class). You'll notice that Torpedo's are awfully similar to Bullets (HINT!)

Flatulence Torpedoes are fired by the Player ship to kill Aliens, or by certain Aliens to kill the Player ship.

Basic Flatulence Torpedo Requirements

When the Player ship fires a Torpedo, the Player's code must insert a Torpedo into the space field one square above their ship. Once the Torpedo has been fired, it will proceed to move upward until it hits an Alien or flies off the screen. When an Alien ship fires a Torpedo, their code must insert a Torpedo into the space field one square below the Alien ship. Once the Torpedo has been fired, it will proceed to move downward until it hits the Player ship or flies off the screen.

Each Torpedo object must have an image ID of IID_TORPEDO.

What a Torpedo Must Do During a Tick

Each time the Torpedo is asked to do something (during a tick), it should:

1. Determine if it has collided with a target. Player-fired Torpedoes may only collide with (and injure) Alien ships. Similarly, Alien-fired Torpedoes may only collide with (and injure) the Player ship. Alien-fired Torpedoes will pass harmlessly by other Alien ships. All Torpedoes will pass harmlessly by Stars and Goodies.
2. If the Torpedo collided with one or more VALID targets (e.g., a Player-fired Torpedo landed on the same square as an Alien):
    a. The Torpedo must damage the target by reducing its life energy by 8 points (by calling the target's Damage method)
    b. If the Torpedo simultaneously impacts multiple targets, it must damage all targets by 8 points (e.g., what if multiple Aliens are on the same square when a Player-fired Torpedo hits)
    c. The Torpedo must then set its status to "dead" so it will be removed from the space field at the end of the current tick.
3. The Torpedo, if it is still alive (i.e., it didn't collide in step #1), must update it's location by one square (moving either up or down one square, depending on who fired it) using the GraphObject's *moveTo()* method.
4. If the Torpedo moved and is still alive, it must then AGAIN check to see if it collided with a target and take the appropriate action, if so. (See step #2)

When does a Torpedo Die?

A Torpedo dies when it either impacts a target (e.g., an Alien or the Player), or it flies off the top or bottom of the space field.

Nachlings

Here are the requirements you must meet when implementing the Nachling (likely inside a Nachling class, derived in some way from our GraphObject class):

Basic Nachling Requirements

All Nachlings must have an image ID of IID_NACHLING.

When a new Nachling is created, it must start out at a random X coordinate (between 0 and 29) and with a fixed Y coordinate of 39 (at the top of the space field). All Nachlings must start out with a "state" of 0 (zero) - more on this later. All Nachlings start out with a certain amount of life energy (aka hit points), which is described in the Adding New Aliens section above.

What a Nachling Must Do During a Tick

Each Nachling must be given an opportunity to do something during every tick. When given an opportunity to do something, the Nachling must do the following:

1. Nachlings may only do something every other tick of the game; therefore, their doSomething method should do nothing (immediately return) every other time it's called.
2. Assuming a Nachling has decided to do something during the current tick, it must use the following algorithm:
3. The Nachling will check its current "state" and vary its behavior based on this state.
4. If its state is 0 (zero):
    a. If the X coordinate of the Nachling is the same as the Player ship's X coordinate AND the Nachling is not in the far-left or far-right columns (e.g., X==0 or X==29), then:
        i. The Naching will change its state from 0 to 1.
        ii. The Nachling will compute a minimum distance to a border (MDB) value. This value is equal to the minimum distance to both the far-left and far-right columns of the space field. So, for instance, if the Nachling were at X=2, the distance to the leftmost column would be 2 and the distance to the rightmost column would be 27 (29 – 2). The MDB would be the smaller of 2 and 27; in this case, it would be 2. If the Nachling were at X=22, the distance to the leftmost column would be 22, and the distance to the rightmost column would be 7 (29– 22). Therefore the MDB would be 7.
        iii. If the MDB is larger (but not equal to) than 3, then set the Nachling's horizontal movement distance (HMD) to a random value between 1 and 3. Otherwise set the Nachling's HMD equal to the computed MDB.
        iv. Choose a random horizontal movement direction (left or right) for your Nachling.
        v. Set a horizontal movement remaining (HMR) member variable equal to the HMD value.
        vi. The Nachling will then continue with step c (below).
    b. Otherwise, there is a 1 in 3 chance that the Nachling will do the following:
        i. If the Nachling's X coordinate is less than the Player's X coordinate, then the Nachling will use the *moveTo()* method to adjust it's location right one square AND down one square. The Nachling must exhibit no other behaviors during the current tick.
        ii. If the Nachling's X coordinate is greater than the Player's X coordinate, then the Nachling will use the *moveTo()* method to adjust it's location left one square AND down one square. The Nachling must exhibit no other behaviors during the current tick.
    c. Otherwise, the Nachling will use the *moveTo()* method to adjust it's location down one square closer to the bottom of the space field.
    d. If a Nachling goes past the bottom of the screen (its Y coordinate is < 0 then it should die and be removed from the space field, but the Player should get no credit for killing the Nachling, nor should it contribute to completing the current round of the game)
    e. The Nachling must exhibit no other behaviors during the current tick.

5. If its state is 1:
    a. If the Player ship's Y coordinate is less than the Nachling's Y coordinate, then the Nachling will transition to state 2. The Nachling must exhibit no other behaviors during the current tick.
    b. If the Nachling's HMR value is equal to zero:
        i. Flip the Nachling's horizontal movement direction (from left to right, or from right to left)
        ii. Set the HMR value equal to double the HMD you computed in state 0 earlier.
        iii. Skip to step d (below).
    c. If the Nachling's HMR value was not equal to zero:
        i. Decrement the Nachling's HMR value by one.
    d. Use the *moveTo()* method to adjust the Nachling's x coordinate – it must move one square horizontally in its current horizontal movement direction.
    e. The Nachling must then compute the "chance" that it will fire at the Player ship:
        i. chancesOfFiring = (BASE_CHANCE_OF_FIRING / game_round) + 1 (where BASE_CHANCE_OF_FIRING is 10).
        ii. Generate a random number between 0 and chancesOfFiring-1 inclusive.
        iii. If the value of this random number is zero, then the Nachling must count the current number of enemy projectiles that are active in the space field (e.g., bullets that were fired by Nachlings, Wealthy Nachlings or SmallBots that are currently active).
        iv. If the count of active enemy projectiles is less than 2 times the current game round number, then the Nachling must fire a Septic Bullet. The Septic bullet must appear just below the Nachling on the screen and fly downward. Such a bullet can only harm the Player ship, not other Aliens.
    f. There is then a 1 in 20 chance that the Nachling will transition to state 2. You must therefore generate a random number and use this to decide if the Nachling should transition to state 2. It will then end the current tick.
6. If its state is 2:
    a. If the Nachling is at the top of the space field (Y=39) then set its state to 0 and do nothing more during this tick.
    b. Otherwise, if the Nachling is at the far left side of the space field (its X coordinate is 0), set it's horizontal movement direction to right and then skip to step e.
    c. Otherwise, if the Nachling is at the far right side of the space field (its X coordinate is 29), set it's horizontal movement direction to left and then skip to step e.
    d. Otherwise, if the Nachling is somewhere in the horizontal middle of the space field. In this case, it will randomly pick a new horizontal movement direction (either left or right) and then continue with step e.
    e. The Nachling will then use the *moveTo()* method to adjust its location both upward (toward Y=39) and in its current horizontal movement direction.

In other words, if its horizontal movement direction were left, and the Nachling was at X,Y, then it's new location would be X-1,Y+1.

f. The Nachling will do nothing else during this tick.

What a Nachling Must Do When Its Attacked or Crashed Into

Each Nachling may be attacked by the Player ship with projectiles like Septic bullets and Flatulence Torpedos. A Nachling might also crash with the Player ship. However, Alien ships do not crash into each other and may freely occupy the same location in the space field. Nor do Alien ships crash into Goodies or visa versa. Since Nachlings can be collided into, they (and likely all actors) should have a method called *damage()* that can be called when another actor happens to collide with them; this method can be used to tell the Nachling (or other actor) that it was just hit, how much damage was done to it, and what caused this damage (e.g., a bullet/torpedo or a crash, since the player wouldn't get points for killing a Nachling by crashing into it, but it would get points by shooting and killing the Nachling).

1. If a projectile (e.g., a Bullet or Torpedo) or the Player ship crash into a Nachling, then do the following:
    a. Decrement the Nachling's energy by the appropriate amount (see Bullet and Torpedo sections for details on damage for projectiles, colliding with a Player ship will cause the Nachling to immediately lose all its energy and die).
    b. If the Nachling's energy has reached zero (or gone below zero), then:
        i. Use the *playSound()* method in the GameWorld class to play the SOUND_ENEMY_DIE effect.
        ii. If the Nachling was damaged by a collision with a Bullet or Torpedo (but NOT by crashing into the Player ship), then use the *increaseScore()* method in the GameWorld class to increase the Player's score by 1000 points.
    c. Otherwise, if the Nachling is still alive, then use the *playSound()* method in your World class to play the SOUND_ENEMY_HIT effect.

Wealthy Nachlings

Here are the requirements you must meet when implementing the Wealthy Nachling (likely inside a WealthyNachling class, derived in some way from our GraphObject class):

Basic Wealthy Nachling Requirements

All Wealthy Nachlings must have an image ID of IID_WEALTHY_NACHLING.

When a new Wealthy Nachling is created, it must start out at a random X coordinate (between 0 and 29) and with a fixed Y coordinate of 39. All Wealthy Nachlings must

start out with a "state" of 0 (zero) - more on this later. All Wealthy Nachlings start out with a certain amount of energy (aka hit points), which is described in the Adding New Aliens section above.

What a Wealthy Nachling Must Do During a Tick

Each Wealthy Nachling must be given an opportunity to do something during every tick. When given an opportunity to do something, the Wealthy Nachling must do the following:

1. The programming logic of Wealthy Nachlings was long ago infected by an insidious computer virus. Therefore, when its *doSomething()* method is called during every tick, it must first:
    a. Check to see if it is currently malfunctioning. If so, it must do nothing during this tick (the *doSomething()* method must immediately return).
    b. If it is not currently malfunctioning, there is a 1 in 200 chance that it will begin to malfunction. Generate a random number to decide if the Wealthy Nachling should begin malfunctioning.
    c. If your random number indicates that the Wealthy Nachling has just started malfunctioning, then it:
        i. Must transition into a malfunctioning state for the current tick AND the following 30 ticks (and do nothing else during any of these ticks).
    g. Otherwise, the Wealthy Nachling's *doSomething()* method should behave just like a regular Nachling's *doSomething()* method.

What a Wealthy Nachling Must Do When Its Attacked or Crashed Into

Each Wealthy Nachling may be attacked by the Player ship with projectiles like Septic bullets and Flatulence Torpedos. A Wealthy Nachling might also crash with the Player ship. However, Alien ships do not crash into each other and may freely occupy the same location in the space field. Nor do Alien ships crash into Goodies or visa versa. Since Wealthy Nachlings can be collided into, they (and likely all actors) should have a method called *damage()* that can be called when another actor happens to collide with them; this method can be used to tell the Wealthy Nachling (or other actor) that it was just hit, how much damage was done to it, and what caused this damage (since the player wouldn't get points for killing a Wealthy Nachling by crashing into it, but it would get points by shooting and killing the Wealthy Nachling).

1. If a projectile (e.g., a Bullet or Torpedo) or the Player ship crash into a Wealthy Nachling, then do the following:
    a. Decrement the Wealthy Nachling's energy by the appropriate amount (see Bullet and Torpedo sections for details on damage for projectiles, colliding with a Player ship will cause the Wealthy Nachling to immediately lose all its life energy and die).

b. If the Wealthy Nachling's energy has reached zero (or gone below zero), then:
    i. Use the *playSound()* method in the GameWorld class to play the SOUND_ENEMY_DIE effect.
    ii. If the Wealthy Nachling was collided into by a Bullet or Torpedo (but NOT by crashing into the Player ship), then use the *increaseScore()* method in your StudentWorld class to increase the Player's score by 1200 points.
    iii. There is a 1 in 3 chance that the Wealthy Nachling will drop a Goodie into the space field at their current location. If they do decide to drop a Goodie of some sort:
        1. There is a 50% chance they will drop an Energy Goodie.
        2. There is a 50% chance they will drop a Torpedo Goodie.
c. Otherwise, if the Wealthy Nachling is still alive, then use the *playSound()* method in your World class to play the SOUND_ENEMY_HIT effect.

SmallBots

Here are the requirements you must meet when implementing the SmallBot (likely inside a SmallBot class, derived in some way from our GraphObject class):

Basic SmallBot Requirements

All SmallBots must have an image ID of IID_SMALLBOT.

When a new SmallBot is created, it must start out at a random X coordinate (between 0 and 29) and with a fixed Y coordinate of 39. All SmallBots start out with a certain amount of life energy (aka hit points), which is described in the Adding New Aliens section above.

What a SmallBot Must Do During a Tick

Each SmallBot must be given an opportunity to do something during every tick. When given an opportunity to do something, the SmallBot must do the following:

1. SmallBots may only do something every other tick of the game; therefore, their *doSomething()* method should do nothing every other time it's called (it must just return).
2. Assuming a SmallBot has decided to do something during the current tick, it must follow the following algorithm:
3. If the SmallBot was just attacked (e.g., during the current/last tick by a Bullet or Torpedo prior to its *doSomething()* method being called) then it will adjust it's X coordinate as it moves down:

a. If the SmallBot is currently in the far left (X=0) column of the space field, it will adjust its X coordinate to X=1.
b. Otherwise, if the SmallBot is currently in the far right (X=29) column of the space field, it will adjust its X coordinate to X=28)
c. Otherwise, the SmallBot will randomly adjust its X coordinate by either +1 or -1.
d. The SmallBot must use the *moveTo()* method to adjust it's location down one and in the specified horizontal direction as determined by steps a-c above.

4. Otherwise, if the SmallBot was NOT just attacked, then it must use the *moveTo()* method to adjust it's location down one (maintaining the its same X coordinate).
5. If the SmallBot is in the same X column as the Player ship and the SmallBot is above the Player ship in the space field, then the SmallBot will try to fire on the player:
    a. The SmallBot computes a value q, where q = (100/game_round).
    b. There is a 1 in q chance that the SmallBot will try fire a Torpedo projectile.
    c. The remaining times, the SmallBot will try to fire a Bullet projectile.
    d. To decide whether to fire:
        i. The SmallBot must count the current number of enemy projectiles that are active in the space field (e.g., bullets that were fired by Nachlings, Wealthy Nachlings or Smallbots that are currently active and flying *toward* the player).
        ii. If the count of active enemy projectiles is less than 2 times the current game round number, then the SmallBot must fire the decided-upon projectile at the Player. The projectile must appear in the square below the SmallBot on the screen and fly downward at the Player ship. Such a bullet can only harm the Player ship, not other Aliens.
6. If a SmallBot ever goes off the bottom of the screen (i.e., it's Y coordinate is < 0), then it should be removed from the space field and die. It must not contribute to the player's score or toward advancing the current round.

<u>What a SmallBot Must Do When Its Attacked or Crashed Into</u>

Each SmallBot may be attacked by the Player ship with projectiles like Septic bullets and Flatulence Torpedos. A SmallBot might also crash with the Player ship. However, Alien ships do not crash into each other and may freely occupy the same location in the space field. Nor do Alien ships crash into Goodies or visa versa. Since SmallBots can be collided into, they (and likely all actors) should have a method called *damage()* that can be called when another actor happens to collide with them; this object can be used to tell the SmallBot (or other actor) that it was just hit, how much damage was done to it, and what caused this damage (since the player wouldn't get points for killing a SmallBot by crashing into it, but it would get points by shooting and killing the SmallBot).

1. If a projectile (e.g., a Bullet or Torpedo) or the Player ship crash into a SmallBot, then do the following:
    a. Decrement the SmallBot's energy by the appropriate level (see Bullet and Torpedo sections for details on damage for projectiles, colliding with a Player ship will cause the Nachling to immediately lose all its energy and die).
    b. If the SmallBot's energy has reached zero (or gone below zero), then:
        i. Use the playSound method in your World class to play the SOUND_ENEMY_DIE effect.
        ii. If the SmallBot was collided into by a Bullet or Torpedo (but NOT by crashing into the Player ship), then use the *increaseScore()* method in your StudentWorld class to increase the Player's score by 1500 points.
        iii. If the SmallBot was collided into by a Bullet or Torpedo (but NOT by crashing into the Player ship), and they died, then there is a 1 in 3 chance that they will drop a FreeShipGoodie into the space field at their current location. They must generate a random number, and if the odds work out, drop a FreeShipGoodie at their current location.
    c. Otherwise, if the SmallBot is still alive, then use the *playSound()* method in our GameWorld class to play the SOUND_ENEMY_HIT effect.

The Free Ship Goodie

Here are the requirements you must meet when implementing the Free Ship Goodie (likely inside a FreeShipGoodie class, derived in some way from our GraphObject class):

Free Ship Goodies are dropped by Aliens under certain circumstances when the Aliens die.

<u>Basic Free Ship Goodie Requirements</u>

Once the Free Ship Goodie has been added to the space field, it will proceed to move downward until it collides with the Player ship, flies off the screen, or exhausts its Free Ship Goodie tick lifetime and simply disappears from the space field. A Free Ship Goodie's tick lifetime is determined when it is created, as follows:

goodieTickLifetime = (100 / current game round) + 30

If the Free Ship Goodie runs into the Player ship, it will increase their number of lives by 1 (see below).

Each Free Ship Goodie object must have an image ID of IID_FREE_SHIP_GOODIE.

What a Free Ship Goodie Must Do During a Tick

Each time the Free Ship Goodie is asked to do something (during a tick), it should:

1. Determine if it has collided with the Player ship.
2. If the Free Ship Goodie collided with the Player ship, then go to step 8
3. Otherwise, the Free Ship Goodie must adjust its brightness using the GraphObject's *setBrightness()* method to: brightness = ((ticksLeftToLive / goodieTickLifetime) + .2); Free Ship Goodies get darker every tick they exist in the space field until they eventually disappear.
4. If the Free Ship Goodie has existed for its entire goodieTickLifetime (it's been in the space field for that many ticks), then it should die and be removed from the space field immediately and do nothing else this tick.
5. Otherwise, once every 3 ticks (starting with the Free Ship Goodie's 3rd tick in existence), the Free Ship Goodie must use the *moveTo()* method to move one square down toward the bottom of the space field. The other 2 out of 3 ticks, the Free Ship Goodie will not move in the space field. This ensures the Free Ship Goodie moves slowly and can more easily be picked up by the Player. If the Free Ship Goodie moves off the bottom of the space field (i.e., its Y coordinate is less than zero), then it should die and subsequently be removed from the space field at the end of the current tick.
6. After moving using *moveTo()*, the Free Ship Goodie must AGAIN check to see if this latest move has caused it to collide with the Player ship. If it did collide with the Player ship, then go to step 8.
7. The Free Ship Goodie is now done and it should do nothing more during the current tick.
8. If you get here, it means the Free Ship Goodie collided with the Player (which is a good thing for the Player). The following must occur:
    d. The Free Ship Goodie must increase the Player's score by 5000 points.
    e. The Free Ship Goodie must play a sound of SOUND_GOT_GOODIE using the *playSound()* method in our GameWorld class.
    f. The Free Ship Goodie must then set its status to "dead" so it will be removed from the space field at the end of the current tick.
    g. The Free Ship Goodie must increase the Player's number of lives by 1 (but do NOTHING to the Player's current life energy level).

When does a Free Ship Goodie Die?

A Free Ship Goodie dies when it either collides with the Player, it has existed exactly goodieTickLifetime ticks, or it flies off the bottom of the space field.

The Energy Goodie

Here are the requirements you must meet when implementing the Energy Goodie (likely inside a EnergyGoodie class, derived in some way from our GraphObject class):

Energy Goodies are dropped by Aliens under certain circumstances when the Aliens die.

Basic Energy Goodie Requirements

Once the Energy Goodie has been added to the space field, it will proceed to move downward until it hits the Player ship, flies off the screen, or exhausts its Energy Goodie tick lifetime and simply disappears from the space field. An Energy Goodie's tick lifetime is determined when it is created, as follows:

goodieTickLifetime = (100 / current game round) + 30

If the Energy Goodie runs into the Player ship, it will restore the Player ship's life energy level to 100% (see below).

Each Energy Goodie object must have an image ID of IID_ENERGY_GOODIE.

What a Energy Goodie Must Do During a Tick

Each time the Energy Goodie is asked to do something (during a tick), it should:

1. Determine if it has collided with the Player ship.
2. If the Energy Goodie collided with the Player ship, then go to step 8
3. Otherwise, the Energy Goodie must adjust its brightness using the GraphObject's *setBrightness()* method to: brightness = ((ticksLeftToLive / goodieTickLifetime) + .2); Energy Goodies get darker every tick they exist in the space field until they eventually disappear.
4. If the Energy Goodie has existed for its entire goodieTickLifetime (it's been in the space field for that many ticks), then it should die and be removed from the space field immediately and do nothing else this tick.
5. Otherwise, once every 3 ticks (starting with the Energy Goodie's 3rd tick in existence), the Energy Goodie must use the *moveTo()* method to move one square down toward the bottom of the space field. The other 2 out of 3 ticks, the Energy Goodie will not move in the space field. This ensures the Energy Goodie moves slowly and can be picked up by the player. If the Energy Goodie moves off the bottom of the space field (i.e., its Y coordinate is less than zero), then it should die and subsequently removed from the space field at the end of the current tick.
6. After moving, the Energy Goodie must AGAIN check to see if this latest move has caused it to collide with the Player ship. If so, then go to step 8.
7. The Energy Goodie is now done and it should do nothing more during the current tick.
8. If you get here, it means the Energy Goodie collided with the Player (which is a good thing for the Player). The following must occur:
    a. The Energy Goodie must increase the Player's score by 5000 points.
    b. The Energy Goodie must play a sound of SOUND_GOT_GOODIE using the *playSound()* method in our GameWorld class.

c. The Energy Goodie must then set its status to "dead" so it will be removed from the space field at the end of the current tick.
d. The Energy Goodie must restore the Player's energy level to its maximum of 50 hit points.

When does an Energy Goodie Die?

An Energy Goodie dies when it either collides with the Player, it has existed exactly goodieTickLifetime ticks, or it flies off the bottom of the space field.

The Torpedo Goodie

Here are the requirements you must meet when implementing the Torpedo Goodie (likely inside a TorpedoGoodie class, derived in some way from our GraphObject class):

Torpedo Goodies are dropped by Aliens under certain circumstances when the Aliens die.

Basic Torpedo Goodie Requirements

Once the Torpedo Goodie has been added to the space field, it will proceed to move downward until it hits the Player ship, flies off the screen, or exhausts its Torpedo Goodie tick lifetime and simply disappears from the space field. A Torpedo Goodie's tick lifetime is determined when it is created, as follows:

goodieTickLifetime = (100 / current game round) + 30

If the Torpedo Goodie runs into the Player ship, it grants the Player ship 5 new Flatulence Torpedos (see below).

Each Torpedo Goodie object must have an image ID of IID_TORPEDO_GOODIE.

What a Torpedo Goodie Must Do During a Tick

Each time the Torpedo Goodie is asked to do something (during a tick), it should:

1. Determine if it has collided with the Player ship.
2. If the Torpedo Goodie collided with the Player ship, then go to step 8
3. Otherwise, the Torpedo Goodie must adjust its brightness using the GraphObject's *setBrightness()* method to: brightness = ((ticksLeftToLive / goodieTickLifetime) + .2); Torpedo Goodies get darker every tick they exist in the space field until they eventually disappear.
4. If the Torpedo Goodie has existed for its entire goodieTickLifetime (it's been in the space field for that many ticks), then it should die and be removed from the space field immediately and do nothing else this tick.

5. Otherwise, once every 3 ticks (starting with the Torpedo Goodie's 3rd tick in existence), the Torpedo Goodie must use the *moveTo()* method to move one square down toward the bottom of the space field. The other 2 out of 3 ticks, the Torpedo Goodie will not move in the space field. This ensures the Torpedo Goodie moves slowly and can be picked up by the player. If the Torpedo Goodie moves off the bottom of the space field (i.e., its Y coordinate is less than zero), then it should die and subsequently removed from the space field at the end of the current tick.
6. After moving, the Torpedo Goodie must AGAIN check to see if this latest move has caused it to collide with the Player ship. If so, then go to step 8.
7. The Torpedo Goodie is now done and it should do nothing more during the current tick.
8. If you get here, it means the Torpedo Goodie collided with the Player (which is a good thing for the Player). The following must occur:
    a. The Torpedo Goodie must increase the Player's score by 5000 points.
    b. The Torpedo Goodie must play a sound of SOUND_GOT_GOODIE using the *playSound()* method in our GameWorld class.
    c. The Torpedo Goodie must then set its status to "dead" so it will be removed from the space field at the end of the current tick.
    d. The Torpedo Goodie must add 5 new Flatulence Torpedoes to the Player ship's arsenal.

When does a Torpedo Goodie Die?

A Torpedo Goodie dies when it either collides with the Player, it has existed exactly goodieTickLifetime ticks, or it flies off the bottom of the space field.

How to Tell Who's Who?

When writing your various classes, you will often need to determine what type of object one of your pointers points to. For example:

```
class Nachling: public Alien
{
public:
  virtual void doSomething()
  {
      Actor* ap = didICollideWithAnotherActor();
      if (ap != NULL)
      {
          // do something to address the collision
      }
  }
};
```

In the code above, the Nachling wants to determine if it's collided with another actor, so it calls a function called *didICollideWithAnotherActor* to determine this (this is a function you might write). Let's assume that this function returns a pointer to another actor if a

collision occurred between the current Nachling object and some other actor in the game. So assuming a collision occurred, how can you determine what type of object "so" points to? Does it point to a Star object? A Nachling object? The Player ship object? A Goodie object? Why do we care? Well, if a Nachling happens to fly into the same X,Y coordinate in the space field as another Alien or a Goodie, well then that's OK and nothing needs to happen (no collision will occur). But if a Nachling flies into the same square as the Player ship, well then that should result in a collision with damage to the Player ship and death for the Nachling. So how can you determine exactly what type of object the "so" variable above points to? Here's the preferred way:

```
if (ap != NULL) // ap points to an actor… What kind is it?
{
     Player *p = dynamic_cast<Player*>(ap);
     if (p != NULL)
    {
        cerr << "so variable points to a Player object.\n";
        p->damagePlayer(10);       // do 10 points of damage to the player
        // etc…
    }
    Nachling *n = dynamic_cast<Nachling*>(ap);
    if (n != NULL)
        cerr << "so points to a Nachling object.";
    Alien *a = dynamic_cast<Alien*>(ap);
    if (a != NULL)
        cerr << "so points to some type of Alien object";
    // etc…
}
```

C++'s *dynamic_cast* keyword can be used to determine whether a given pointer to an object (e.g., "so") points to a particular type of object (e.g., a Player or a Nachling or more generally, any type of Alien). If the conversion type in angled brackets (e.g., "Nachling *" or "Alien *") is either the same pointer type or a superclass pointer type of the pointer in parentheses (e.g., "so"), then this command will return a non-NULL pointer of the type specified in the angled brackets. You can use this pointer to access the referred to object in its native form (e.g., as a Nachling rather than just a generic actor).

Here's another example. For this example, let's assume that the variable ap is a pointer that points to a Nachling object and that Nachlings are derived from the Alien class, and that both Aliens and Players are derived from an Actor base class. Finally the WealthyNachling class is derived from the Nachling class:

```
Actor* ap = new Nachling(…);      // ap points to a Nachling object

      …

// this returns a non-NULL value of type Nachling *,
// since ap points to a Nachling object
Nachling* n = dynamic_cast<Nachling*>(ap);

// this returns NULL, since the object ap points to
// is not a Player nor is it a subclass of Player
Player* p = dynamic_cast<Player*>(ap);

// this returns a non-NULL value of type Alien*, since
// the object ap points to is a Nachling
```

```
// and Nachling is a subclass of the Alien class in this example
Alien* a = dynamic_cast<Alien*>(ap);

// this returns a NULL value, since the object ap points
// to is not a WealthyNachling; nor is it a subclass of WealthyNachling
WealthyNachling* w = dynamic_cast<WealthyNachling*>(ap);
```

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, re-compile 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.**

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 or thousands of errors and just get frustrated! So don't do it.

Compiling the Game

To compile the game, follow these steps:

For Windows

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

For OS X

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

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 Space Inflators game that implements maybe 15% of the overall project. You must program:

1. A class that can serve as the base class for all of your game's objects (e.g., Stars, the Player ship, SmallBots, Nachlings, Goodies, Bullets, etc):
    i. It must have a simple constructor and destructor.
    ii. It must be derived from our GraphObject class.
    iii. It must have a single virtual method called *doSomething()* that can be called by the world to get one of the game's actors to do something.
    iv. You may add other public/private methods and private member variables to this base class, as you see fit.
2. A limited version of your Player ship 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 simple constructor and destructor that initializes the Player – see the Player section for more details on where to initialize the Player ship.
    ii. It must have an Image ID of IID_PLAYER_SHIP.
    iii. 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 is within the limits of where the Player is allowed to move (see the Player section), it updates the Player's location to the target square.

All your *doSomething()* method has to do is properly adjust the Player ship's X,Y coordinates and our graphics system will automatically animate its movement it around the space field!

iv. You may add any public/private methods and private member variables to your Player ship 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. Create a limited version of your Star class, derived in some way from the base class described in 1 above:
    i. It must have a simple constructor and destructor that initialize a new Star (see the Star section for information on where in the space field new Stars should start out).
    ii. It must have an Image ID of IID_STAR.
    iii. It must have a limited version of a *doSomething()* method. All the doSomething method should do is move the Star down one row in the space field until the Star reaches the bottom of the space field and disappears (at which point it should be removed/deleted from the game by your StudentWorld class's *move()* method).
    iv. You may add any set of public/private methods and private member variables to your Star 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. Create a limited version of the StudentWorld class.
    i. Add any private member variables to this class required to keep track of a variable number of Stars as well as the Player ship object.
    ii. Implement a constructor for this class that initializes your member variables.
    iii. Implement a destructor for this class that frees any remaining dynamically allocated data that has not yet been freed at the time the class is destroyed.
    iv. Implement the *init()* method in this class. It must create the Player ship and insert it into the space field at the right starting location (see the Player section for details on the starting location).
    v. Implement the *move()* method in your StudentWorld class. Each time your *move()* method is called, there is a 1 in 3 chance that you will add a single Star to the space field (see the details in the *Adding New Aliens and Stars* section above). You must then program enough of the *move()* method so that it can ask your Player and all of the Stars to do something during each tick and then return. Your *move()* method need not check to see if the Player ship has died or not; you may assume at this point that the Player ship cannot die. Nor need your *move()* method deal with any Aliens or other actors (e.g., Goodies or Bullets) at this point – just Stars and the Player ship.

vi. After your *move()* method has given every Star and the Player a chance to move during a tick, make sure that it then identifies any dead Stars (a star is dead if it has flown off the bottom of the space field - i.e., its Y coordinates are less than zero) and removes them from your data structure and deletes their objects.
vii. 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 Stars and the Player). Note: You must have both a destructor and the *cleanUp()* mehod even though they likely do the same thing.

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 student taking CS131 once got 1,800 compilation errors when compiling a 900-line program 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 an empty space field with the Player ship in its proper starting position and randomly-distributed Stars flying down the screen. If your base class(es) and Player class work properly, you should be able to move the Player around the space field using the directional keys.

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 builds 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 Aliens or Goodies (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, Star, 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 **builds 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, Star and Player 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!

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 Space Inflators game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in the following files, and ONLY the following files. If you name your source files with other names, you will be docked points, so be careful!

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 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 blah() function in my base class because all actors in Space Inflators must have a blah function, and each type of actor defines their own special version of it."
2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I wasn't able to implement shooting of Septic Bullets." or

"My Wealthy Nachling doesn't work correctly yet so I just treat it like a Nachling right now."

3. A list of other design decisions and assumptions you made, e.g.:
    i. It was ambiguous what to do in situation X, and 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 ambiguous. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If the specification is ambiguous and our program is ambiguous, do whatever seems reasonable and document it in your report. **If the specification is ambiguous, 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 perfect, that's better than it not even building!

Q: Where can I go for help?
A: Try TBP/HKN/UPE – they provide free tutoring and can help your with your project!

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!