Project 3 Warmup

We anticipate that many people working on Project 3 will spend a lot of time debugging something that arises from some common novice misunderstandings. To save you that time later, we'll give you a chance to make those mistakes in a simpler context, so you can work out the issues and how they manifest themselves. (It may turn out that you don't have the misunderstandings, so you won't have any problems during this warmup. Still, keep this warmup in mind, because you may still make the mistake in Project 3.)

You will not turn in any of the code you write because of this warmup. Be sure, though, to run them under g32 to be sure there are no memory leaks.

Material about vectors, lists, and iterators are in the STL slides (to be covered in class Monday) or lecture9-updated.pptx in Carey Nachenberg's slides.

Implement the `removeOdds` function:

```
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
#include <cassert>
using namespace std;

  // Remove the odd integers from li.
  // It is acceptable if the order of the remaining even integers is
not
  // the same as in the original list.
void removeOdds(list<int>& li)
{
}

void test()
{
    int a[8] = { 2, 8, 5, 6, 7, 3, 4, 1 };
    list<int> x(a, a+8);  // construct x from the array
    assert(x.size() == 8 && x.front() == 2 && x.back() == 1);
    removeOdds(x);
    assert(x.size() == 4);
    vector<int> v(x.begin(), x.end());  // construct v from x
    sort(v.begin(), v.end());
    int expect[4] = { 2, 4, 6, 8 };
    for (int k = 0; k < 4; k++)
        assert(v[k] == expect[k]);
}

int main()
```

```
{
    test();
    cout << "Passed" << endl;
}
```

Implement the `removeOdds` function:

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <cassert>
using namespace std;

  // Remove the odd integers from v.
  // It is acceptable if the order of the remaining even integers is
not
  // the same as in the original vector.
void removeOdds(vector<int>& v)
{
}

void test()
{
    int a[8] = { 2, 8, 5, 6, 7, 3, 4, 1 };
    vector<int> x(a, a+8);  // construct x from the array
    assert(x.size() == 8 && x.front() == 2 && x.back() == 1);
    removeOdds(x);
    assert(x.size() == 4);
    sort(x.begin(), x.end());
    int expect[4] = { 2, 4, 6, 8 };
    for (int k = 0; k < 4; k++)
        assert(x[k] == expect[k]);
}

int main()
{
    test();
    cout << "Passed" << endl;
}
```

Implement the `removeBad` function:

```
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
#include <cassert>
using namespace std;

vector<int> destroyedOnes;

class Movie
{
  public:
    Movie(int r) : m_rating(r) {}
```

```
        ~Movie() { destroyedOnes.push_back(m_rating); }
        int rating() const { return m_rating; }
      private:
        int m_rating;
    };

      // Remove the movies in li with a rating below 50 and destroy them.
      // It is acceptable if the order of the remaining movies is not
      // the same as in the original list.
    void removeBad(list<Movie*>& li)
    {
    }

    void test()
    {
        int a[8] = { 85, 80, 30, 70, 20, 15, 90, 10 };
        list<Movie*> x;
        for (int k = 0; k < 8; k++)
            x.push_back(new Movie(a[k]));
        assert(x.size() == 8 && x.front()->rating() == 85 && x.back()-
>rating() == 10);
        removeBad(x);
        assert(x.size() == 4 && destroyedOnes.size() == 4);
        vector<int> v;
        for (list<Movie*>::iterator p = x.begin(); p != x.end(); p++)
        {
            Movie* mp = *p;
            v.push_back(mp->rating());
        }
          // Aside:  In C++11, the above loop could be
          //     for (auto p = x.begin(); p != x.end(); p++)
          //     {
          //         Movie* mp = *p;
          //         v.push_back(mp->rating());
          //     }
          // or
          //     for (auto p = x.begin(); p != x.end(); p++)
          //     {
          //         auto mp = *p;
          //         v.push_back(mp->rating());
          //     }
          // or
          //     for (Movie* mp : x)
          //         v.push_back(mp->rating());
          // or
          //     for (auto mp : x)
          //         v.push_back(mp->rating());
        sort(v.begin(), v.end());
        int expect[4] = { 70, 80, 85, 90 };
        for (int k = 0; k < 4; k++)
            assert(v[k] == expect[k]);
        sort(destroyedOnes.begin(), destroyedOnes.end());
        int expectGone[4] = { 10, 15, 20, 30 };
        for (int k = 0; k < 4; k++)
            assert(destroyedOnes[k] == expectGone[k]);
        for (list<Movie*>::iterator p = x.begin(); p != x.end(); p++)
            delete *p;
```

```
}

int main()
{
    test();
    cout << "Passed" << endl;
}
```

Implement the `removeBad` function:

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <cassert>
using namespace std;

vector<int> destroyedOnes;

class Movie
{
  public:
    Movie(int r) : m_rating(r) {}
    ~Movie() { destroyedOnes.push_back(m_rating); }
    int rating() const { return m_rating; }
  private:
    int m_rating;
};

  // Remove the movies in v with a rating below 50 and destroy them.
  // It is acceptable if the order of the remaining movies is not
  // the same as in the original vector.
void removeBad(vector<Movie*>& v)
{
}

void test()
{
    int a[8] = { 85, 80, 30, 70, 20, 15, 90, 10 };
    vector<Movie*> x;
    for (int k = 0; k < 8; k++)
        x.push_back(new Movie(a[k]));
    assert(x.size() == 8 && x.front()->rating() == 85 && x.back()-
>rating() == 10);
    removeBad(x);
    assert(x.size() == 4 && destroyedOnes.size() == 4);
    vector<int> v;
    for (int k = 0; k < 4; k++)
        v.push_back(x[k]->rating());
    sort(v.begin(), v.end());
    int expect[4] = { 70, 80, 85, 90 };
    for (int k = 0; k < 4; k++)
        assert(v[k] == expect[k]);
    sort(destroyedOnes.begin(), destroyedOnes.end());
    int expectGone[4] = { 10, 15, 20, 30 };
    for (int k = 0; k < 4; k++)
        assert(destroyedOnes[k] == expectGone[k]);
```

```
        for (vector<Movie*>::iterator p = x.begin(); p != x.end(); p++)
            delete *p;
    }

    int main()
    {
        test();
        cout << "Passed" << endl;
    }
```

Make sure you understand why the code below passes the first two tests but fails the third. Draw pictures if necessary. Don't forget the lesson you learn from this problem when working on Project 3.

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

const int MAGIC = 11223344;

void test()
{
    bool allValid = true;

    vector<int> v1(5, MAGIC);
    int k = 0;
    for ( ; k != v1.size(); k++)
    {
        if (v1[k] != MAGIC)
        {
            cout << "v1[" << k << "] is " << v1[k] << ", not " << MAGIC
<<"!" << endl;
            allValid = false;
        }
        if (k == 2)
        {
            for (int i = 0; i < 5; i++)
                v1.push_back(MAGIC);
        }
    }
    if (allValid  &&  k == 10)
        cout << "Passed test 1" << endl;
    else
        cout << "Failed test 1" << endl;

    allValid = true;
    list<int> l1(5, MAGIC);
    k = 0;
    for (list<int>::iterator p = l1.begin(); p != l1.end(); p++, k++)
    {
        if (*p != MAGIC)
        {
            cout << "Item# " << k << " is " << *p << ", not " << MAGIC
<<"!" << endl;
```

```
            allValid = false;
        }
        if (k == 2)
        {
            for (int i = 0; i < 5; i++)
                l1.push_back(MAGIC);
        }
    }
    if (allValid  &&  k == 10)
        cout << "Passed test 2" << endl;
    else
        cout << "Failed test 2" << endl;

    allValid = true;
    vector<int> v2(5, MAGIC);
    k = 0;
    for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++, k++)
    {
        if (k >= 20)  // prevent infinite loop
            break;
        if (*p != MAGIC)
        {
            cout << "Item# " << k << " is " << *p << ", not " << MAGIC
<<"!" << endl;
            allValid = false;
        }
        if (k == 2)
        {
            for (int i = 0; i < 5; i++)
                v2.push_back(MAGIC);
        }
    }
    if (allValid  &&  k == 10)
        cout << "Passed test 3" << endl;
    else
        cout << "Failed test 3" << endl;
}

int main()
{
    test();
      }
```

Project 3
MiniRogue

Time due: 11:00 PM Friday, May 22

Introduction

For this project, you will be building a simple computer game called MiniRogue. MiniRogue is an action-adventure game that is modeled after the original Unix Rogue game. In MiniRogue, the player navigates through a multi-level dungeon of mazes, in search of a golden idol. Upon locating the idol, the player is instantly transported out of the dungeon and wins the game.

Upon starting a new game, you the player are placed on the top-level maze of the dungeon. From here, you must work your way through the maze, battling monsters, finding treasures, and using staircases to descend ever deeper into the dungeon's depths. When you reach the bottom level of the dungeon, you must scavenge the maze looking for the golden idol in order to win the game. Once you have taken the idol, you instantly win and the game ends.

As you move through the maze, you will encounter a number of different types of monsters that you will have to battle, including:

- Bogeymen (shown on screen as a `B`)
- Dragons (shown on screen as a `D`)
- Goblins (shown on screen as a `G`)
- Snakewomen (shown on screen as an `S`)

Each of these monsters has a different behavior and reacts differently to the player (who is shown on screen as an `@`). More details about the monsters are below.

In addition to monsters, you'll find many different objects during your adventure:

- Impenetrable walls (shown on screen as `#`)
- Stairways down to the next level (shown on screen as `>`)
- The golden idol (shown on screen as `&`)
- Weapons (all weapons are shown on screen as a `)` character)
    - Maces
    - Short swords
    - Long swords

    - Magic fangs of sleep (that put an opponent to sleep)
    - Magic axes (that hit an opponent more often than regular weapons)
- Scrolls (all scrolls are shown on screen as `?`)
    - A scroll of teleportation (when read, randomly moves the player)
    - A scroll of improve armor (when read, makes it harder for monsters to hit the player)
    - A scroll of raise strength (when read, makes a player's blows more effective)
    - A scroll of enhance health (when read, raises maximum hit points)
    - A scroll of enhance dexterity (when read, makes it more likely the player will hit an opponent)

Here is an example of what the game display might look like:

```
######################################################################
######################          ##            ######################
######################      B   ##            ##              ######
######################          ##     G                  )   ######
######################          ##            ##       S       ######
#####              ####          ?D        #######    G          ######
##### >            ####      B       ###### ########             ######
#####         D    ####              ##          ########### ############
#####               S           )    ##          ########### ############
#####     G         ###################          ########### ############
#####               ################### D@       ########### ############
#####################################       S ########### ############
#####################################         ########          ######
#######################################################   ?      ######
#########################################################         ######
######################################################################
######################################################################
######################################################################
Dungeon Level: 3, Hit points: 10, Armor: 3, Strength: 2, Dexterity: 2

Player slashes short sword at the Dragon and misses.
the Dragon swings long sword at Player and misses.
```

To control the player in the game, you issue commands from the keyboard. Use the getCharacter function we will supply you to read characters from the keyboard in a way that does not require the user to hit Enter after each command. (Also, getCharacter does not echo the character onto the screen.) MiniRogue is not a "real-time" game; the game progresses only when the player issues a command. After each player command, the monsters each make a move and then the screen is updated to reflect the current state of the game. The player commands are:

- Move one space using the arrow keys or the classic Rogue movement letters:
    - `h` to move left
    - `l` to move right

    - `k` to move up
    - `j` to move down

  Our getCharacter function translates movement commands into character constants named `ARROW_LEFT`, etc.

- Attack a monster next to you by moving in its direction.
- Pick up an object by standing on it and typing `g`.
- Wield a weapon by typing `w` and then selecting a weapon from your inventory.
- Read a scroll by typing `r` and then selecting a scroll from your inventory.
- See an inventory of your items by typing `i`.
- When standing on a stairway, descend deeper into the dungeon by typing `>`.
- Quit the game by typing `q`.
- Cheat by typing `c`. This command sets the player's characteristics to make defeating monsters easy, and exists solely to let us test some aspects of your program without our having to slog through a regular game.

If you type something not on this list, you do nothing for this turn, but the monsters take their turn.

This sample Windows executable, this sample Mac executable, or this sample Linux executable lets you get a feel for the game.

High-level Game Details

This section describes high-level details of the MiniRogue game. These are mandatory project requirements.

The Dungeon

- The MiniRogue dungeon is 5 levels deep. When the game starts, the player is placed on level 0; the deepest level is level 4.
- At the start of a new game and every time the player takes a stairway down, a new level should be generated (not precomputed). Each level of the dungeon is a maze 70 columns wide and 18 rows high. The level should appear as rooms connected by corridors. Each level must have its outer edges be surrounded by walls, so the only exit for the player is the (randomly placed) stairway down. Only level 4 has the golden idol and no stairway down, since picking up the golden idol ends the game.
- At the start of the game and each time the player descends to a new level L of the dungeon, the player should be randomly placed, along with M randomly

placed monsters, where M = randInt(2, 5*(L+1)+1) (So, for example, level 3 will have between 2 and 21 monsters.)
- Either 2 or 3 random objects (scrolls or weapons) should be randomly placed on each level. However, magic fangs of sleep, magic axes, and teleportation scrolls are never placed on the level; you have to kill a monster to get one.
- Nothing may be initially placed on or may move onto a wall.
- Only one actor (i.e., the player or a monster) at a time may occupy a given place on a level.
- Only one non-actor (i.e., a scroll, a weapon, the idol, or a stairway) at a time may occupy a given place on a level.
- It is permissible for an actor to be initially placed on or move to a space occupied by a non-actor. In other words, an actor can start off standing on a scroll, a weapon, the idol, the stairway, or an empty space, and may move onto those items.

Playing the Game

- When the game starts, the initial level is displayed. When the player descends the stairway, the next level down is displayed. Otherwise, after each player command is entered, the command is performed, the monsters all make their moves, and the level is redisplayed.
- Below the display of the level, display the player's statistics: the dungeon level (0 to 4), hit points (0 to 99), armor level (0 to 99), strength (0 to 99), and dexterity (0 to 99).
- Below the statistics line, display one empty line and then one or more result messages from the last move (e.g., *Player slashes short sword at the Snakewoman and misses.*).
- When the player dies (i.e., the player's hit points are 0 or less), the game is over. If the player wins by picking up the golden idol, the game is over. In either case, an appropriate message should be written at the bottom of the display, and the game should exit only after the user types `q`.
- The easiest way to change the display is to clear the screen using the clearScreen function we provide you and rewrite everything.
- Develop your project starting from this skeleton project. In the skeleton, the main routine's code `Game g(15); g.play();` is what sets up the game and plays it. Clearly, you'll need a Game class with a constructor that takes an integer and a play function. See below in the Goblin section for the meaning of the Game constructor argument.

  The skeleton project contains some utilities you may use:

- getCharacter, to get a character from the keyboard without waiting for a newline and without echoing the character to the screen. The directional keys h, j, k, and l, as well as the arrow keys, are mapped to constants ARROW_LEFT, ARROW_RIGHT, etc.
- clearScreen, to clear the screen in preparation for redrawing the screen.
- randInt, to pick a random integer in a certain range.
- trueWithProbability, which takes an argument that represents a probability p. The function picks a random bool value to return: With probability p, it picks true, and with probability 1-p it picks false.

You must **not** make any changes to the utilities.h and the utilities.cpp files, and you may change main.cpp only to reduce the value of the argument passed to the Game constructor.

Actors

Since the player and the monsters share so many characteristics, let's classify them as *actors*. Each actor has these attributes:

- The actor's current position in the level.
- Hit points, an integer from 0 to 99 indicating how healthy the actor is. Hit points decrease when an attacker hits the actor. The actor dies when the hit points drop to 0 or less.
- The weapon the actor is currently wielding.
- Armor points, an integer from 0 to 99 specifying how well protected from attacks the actor is. The higher the number, the better the protection.
- Strength points, an integer from 0 to 99 specifying how strong the actor is when attacking.
- Dexterity points, an integer from 0 to 99 specifying how skilled the actor is at using a weapon or dodging an attack. The higher this number is, the more likely an actor will hit an opponent during battle or avoid attack.
- Sleep time, an integer from 0 to 9 indicating how long the actor will sleep. The actor starts out awake (sleep time of 0). Getting hit by the magic fangs of sleep may put the actor to sleep, in which case the sleep time is set to some positive value. A sleeping actor does not move and if it's the player, does not respond to any command. Each time a sleeping actor is given an opportunity to move, it does nothing, but the sleep time is decreased by 1.

Whenever an actor does any of the following, you must ensure a message is displayed below the statistics line the next time the level is displayed:

- The result of an attack of one actor on another must be displayed. Examples include
    - Player swings mace at the Bogeyman and hits.
    - the Goblin slashes short sword at Player dealing a final blow.
    - the Snakewoman strikes magic fangs at Player and hits, putting Player to sleep.
    - the Snakewoman strikes magic fangs at Player and misses.
- The result of picking up an object must be displayed (e.g., "You pick up long sword." or "You pick up a scroll of teleportation.") (Only the player can pick up objects.)
- The result of reading a scroll must be displayed (e.g., "You read a scroll of enhance armor. Your armor glows blue.") (Only the player can read scrolls.)

Player

- The player starts with 20 hit points, which is the maximum hit point value possible for the player initially. Before each command from the user, there is a 1 in 10 chance the player will regain 1 hit point, up to the player's maximum hit point value. The only way the player can increase the maximum hit point value is to read a scroll of enhance health.
- The player starts with a short sword as the current weapon. The player can change the current weapon by wielding (the `w` command) any weapon in the player's inventory.
- The player starts with 2 armor points. The only way to increase the armor points is to read a scroll of enhance armor.
- The player starts with 2 strength points. The only way to increase strength is to read a scroll of strength.
- The player starts with 2 dexterity points. The only way to increase dexterity is to read a scroll of enhance dexterity.
- The player maintains an inventory consisting of the current weapon and the items the player has picked up. When a scroll is read, it magically self-destructs (i.e., it is removed from the inventory). The player may pick up an object only if the inventory has 25 or fewer items (so when the inventory is listed, items can be labeled by letters a through z). In this game, an item picked up can not be dropped to make room for another item.

The user can issue the following commands, each of which counts as the player's turn:

- Move one space left, right, up, or down (`h`, `l`, `k`, `j`, or the arrow keys).
- Attack a monster next to you by moving in its direction.
- Pick up an object by standing on it and typing `g`. If the player's inventory is not full, the item will be removed from the level's floor and added to the player's

inventory. If the player picks up the golden idol, the game is over and the player wins.
- Wield a weapon. When the `w` command is typed, the screen should be cleared, the inventory should be displayed, and a character read from the keyboard. If that character is a letter labeling a weapon, then that weapon becomes the player's current weapon (and the previous weapon remains in the inventory); otherwise, the current weapon is not changed.
- Read a scroll. When the `r` command is typed, the screen should be cleared, the inventory should be displayed, and a character read from the keyboard. If that character is a letter labeling a scroll, then that scroll performs its magic and magically self-destructs; otherwise, nothing happens.
- See an inventory of your items by typing `i`.
- When standing on a stairway, descend deeper into the dungeon by typing `>`.
- Quit the game by typing `q`.
- Cheat by typing `c`. This command sets the player's strength to 9 and maximum hit points to 50. (This command exists solely to make it easier for us to test some aspects of your program.)

The `w`, `r`, and `i` commands should display the inventory in a manner such as this:

```
Inventory:
 a. short sword
 b. long sword
 c. a scroll of strength
 d. long sword
```

Monsters

Monsters are limited in their actions; when it's a monster's turn, that monster will either:

- Do nothing.
- Move one space left, right, up, or down.
- Attack the player.

If a monster is next to the player, it will attack the player. When a monster dies, it might drop an item onto the position where it dies if there is no item already at that position.

Bogeymen

Bogeymen appear only at dungeon level 2 or deeper. All bogeymen are created with the following values:

- Hit points: a random integer from 5 to 10
- Weapon: a short sword
- Strength: a random integer from 2 to 3
- Dexterity: a random integer from 2 to 3
- Armor: 2

Bogeymen can smell when the player is near, but are not too bright. If the bogeyman could take one through five steps, ignoring walls or other monsters, and reach the player, then it smells the player. (Of course, like other actors, bogeymen can't actually walk through walls or monsters.) If the bogeyman smells the player on its turn, it will move one step closer to the row or column the player is on, if possible, but never one step *farther* from the player's row or column. If the bogeyman is too far away to smell the player, it will not move. Consider the following example:

```
..B.DB..B.
B#####.##.
..#.@..DB.
.B........
```

The bottommost bogeyman would move one step up or right, closer to the player's row or column; it's too stupid to consider whether moving right is better than moving up in this case. The leftmost bogeyman would move one step down. The top left one would move one step right, closer to the player's column. The top middle one will not move; the wall and the dragon are blocking it from moving closer to the player's row or column, and it's too stupid to move *away* from the player's column in order to go around the wall. The top right bogeyman is too far away to smell the player, so will not move. The bottom right bogeyman will not move; since it's on the same row as the player, it will not move off it, and it's blocked by the dragon from moving closer to the player's column.

When a bogeyman dies, there is a 1 in 10 chance it will drop a magic axe if there is no item at the position where it dies.

Snakewomen

Snakewomen may appear on any dungeon level. All snakewomen are created with the following values:

- Hit points: a random integer from 3 to 6
- Weapon: magic fangs of sleep
- Strength: 2
- Dexterity: 3
- Armor: 3

Snakewomen are as dumb as bogeymen, and don't smell as well. They move according to the same criteria as bogeyman, except that they can smell the player only from a distance of three or less, not five or less. When a snakewoman dies, there is a 1 in 3 chance she will drop her magic fangs of sleep if there is no item at the position where she dies.

Dragons

Dragons appear only at dungeon level 3 or deeper. All dragons are created with the following values:

- Hit points: a random integer from 20 to 25
- Weapon: a long sword
- Strength: 4
- Dexterity: 4
- Armor: 4

Dragons do not move, since they want to protect their treasure, but like any monster, will attack the player if the player is next to it. Dragons are the only monster that, like the player, occasionally regains hit points. Before each turn a dragon takes, there is a 1 in 10 chance the dragon will regain 1 hit point, up to its initial maximum number of hit points. When a dragon dies, there is a 100% chance it will drop a scroll of some kind if there is no item at the position where it dies.

Goblins

Goblins may appear on any dungeon level. All goblins are created with the following values:

- Hit points: a random integer from 15 to 20
- Weapon: a short sword
- Strength: 3
- Dexterity: 1
- Armor: 1

Goblins are very intelligent creatures, and can smell the player if the player is reachable in 15 steps or less. (Notice this is different from bogeymen and snakewomen, who can smell through walls and other monsters; for a goblin to be able to smell the player, there must be a path it could move along to reach the player in 15 steps or less.) If the goblin smells the player on its turn, it will make an optimal move (there may be more than one) to get one step closer to the player along a traversable

path. If the goblin is too far away to smell the player, it will not move. Consider the following example:

```
##################
#.....G.#G......G
#D#.###.#####.###
#.#..@..#........
#.GD..#...####..G
############G..##
##################
```

The top left goblin will move one step right, since that puts it closer to the player along a path than if it moved one step left. The bottom left goblin will not move, because the player is not reachable (the goblin is blocked by walls and dragons). The bottom middle goblin will move one step right, and the top right goblin will move one step left, because both can just barely smell the player (they're exactly 15 steps away). The bottom right goblin will move either up or left, since either one is an optimal move. The top middle goblin will not move, because it could not reach the player in 15 steps (the shortest traversable path is 16 steps).

The algorithm you write for determining an optimal goblin move must use recursion in a significant way. ("Significant" means, for example, that you can't just slap a trivial recursive call onto a fundamentally non-recursive algorithm, such as `void f(int n) { if (n != 0) f(0); else solveNonRecursively(); }`.) Your game should not slow to a crawl every time the algorithm is run, although it may run a bit slower.

When a goblin dies, there is a 1 in 3 chance it will drop either a magic axe or magic fangs of sleep if there is no item at the position where it dies.

This spec defines the distance a goblin can smell as 15. In fact, your program must use the argument passed to the Game constructor as the distance. (In the main routine we provided, it is 15.) The reason for this is that if you carelessly code your recursive goblin move algorithm, it might take an uncomfortably long time to search up to 15 steps away. Until you improve your algorithm, to be able to comfortably test the game as you work on other parts of the code, you can reduce the value of argument main passed to the Game constructor, so that the search distance is smaller and thus faster. When we build your program for our testing, we will use 15 as the argument as per this spec, but if your program is uncomfortably slow, we will reduce that argument (for a small loss of points) as the only way we will try to speed things up; we will not try to find other places in your code where you hard-coded a value like 15.

Game Objects

Since weapons, scrolls, and the golden idol share some characteristics (e.g., they can be picked up, they can occupy the same space as an actor, etc.), let's classify them as *game objects*. Each game object has a name (e.g. "long sword").

Weapons

All weapons have:

- An action string, which is used to help form the message displayed when one actor attacks another. This string is part of a message like "Player swings long sword at ..." or "the Goblin slashes short sword at ...".
- A dexterity bonus, an integer used in determining whether an attacker using that weapon hits the defender, using the following formula:
  ```
  attackerPoints = attackerDexterity + weaponDexterityBonus;
  defenderPoints = defenderDexterity + defenderArmorPoints;
  if (randInt(1, attackerPoints) >= randInt(1, defenderPoints))
      attacker has hit defender
  ```
- A damage amount, an integer used in determining how much damage is done to a defender who is hit (i.e., how much the defender's hit points decrease), using the following formula:
  ```
  damagePoints = randInt(0, attackerStrength + weaponDamageAmount - 1);
  ```

The weapons are:

- Maces, with a dexterity bonus of 0 and a damage amount of 2.
- Short swords, with a dexterity bonus of 0 and a damage amount of 2.
- Long swords, with a dexterity bonus of 2 and a damage amount of 4.
- Magic axes, with a dexterity bonus of 5 and a damage amount of 5.
- Magic fangs of sleep, with a dexterity bonus of 3 and a damage amount of 2. In addition to the regular hit point damage magic fangs of sleep do when the attacker hits the defender, there is a 1 in 5 chance that the magic fangs of sleep will put the defender to sleep. The number of moves the sleep will last (the sleep time) is a random integer from 2 to 6, call it X. If the defender is already asleep, with sleep time Y, then the defender's sleep time becomes the maximum of X and Y (and not, say, Y+X).

Scrolls

Scrolls are magical parchments that can be read in order to cast spells. Unlike the Rogue game, in MiniRogue, a scroll always has a positive effect. When a scroll is read, it performs its effect and then magically self-destructs (i.e., is removed from the player's inventory). These are the types of scrolls and their effect:

- A scroll of teleportation: The player is randomly moved to another place in the level that is not occupied by a wall or a monster.
- A scroll of improve armor: The player's armor points are increased by a random integer from 1 to 3.
- A scroll of raise strength: The player's strength points are increased by a random integer from 1 to 3.
- A scroll of enhance health: The player's maximum hit point value is increased by a random integer from 3 to 8. This scroll does *not* affect the player's current number of hit points.
- A scroll of enhance dexterity: The player's dexterity is increased by 1.

Implementation suggestions

We don't expect that everyone will finish all aspects of the game. Make sure you turn in something that is minimally playable (e.g., the player can move and be blocked by walls). The more of the feel of the game you implement, the better. Start with something really simple (e.g., no monsters or scrolls, one big room, etc.), get that working, and add functionality a little at a time.

If you're having trouble wielding a weapon from your inventory or reading a scroll, you probably want to learn about dynamic_cast.

While you can build your program under Xcode, if you launch the resulting executable from Xcode, you'll find that non-arrow key characters you type are echoed to the screen, and the screen won't clear; this is because Xcode does not provide a true terminal window. Running from the command line in a Terminal window should work fine.

If you want to build your program from the command line under Linux or macOS:

1. Put all the .h and .cpp files in some directory — perhaps ~/Minirogue
2. Open a shell window from the Terminal utility.
3. Execute these commands:

```
cd ~/Minirogue
g++ -o minirogue *.cpp
```

   This should produce an executable file named `minirogue`. On cs32.seas.ucla.edu, using g32 instead of g++ will check for runtime issues like bad pointer accesses or memory leaks.

6. Execute the command `./minirogue` to run the program.

Turn it in

By Thursday, May 21, there will be a link on the class webpage that will enable you to turn in your source files and report. You will turn in a zip file containing:

- All of your source files (the .h and .cpp files that make up your program). Do not turn in the other files that Visual Studio or Xcode produces for your project (e.g., files ending in .sln, .vcproj, .ncb, .pbxproj etc.) Class and function implementations should be appropriately commented to guide a reader of the code.
- `report.docx` or `report.doc` (a Word document), or `report.txt` (a text file), a report containing
    - The name of your recursive goblin movement function and the name of the file it's implemented in, or else a statement that your goblin movement function is not recursive. (Make this the first thing, to help the grader find the function.)
    - a description of the design of your program. For each of your classes, indicate its purpose, what behaviors it implements, and how it relates to other classes.
    - documentation of non-trivial algorithms. Use pseudocode where it helps clarify the presentation.
    - a list of any known bugs, features not implemented, or serious inefficiencies.

    Notice that for this project, your report does *not* have to include a list of test cases.

When you turn in your program, you should also fill out the 10-second survey about whether you'd rather have us test your program under Windows, macOS, or Linux.

Here's how we will evaluate what you turn in:

- Correctness (60 points total):
    - (10 points) Do you display the dungeon? Is there a player that you can move around? Do walls block you? Is there at least one kind of interesting thing you can do (e.g., encounter and interact with a monster, or find and pick up an object)?
    - (10 points) Can you fight monsters? Can you kill and be killed?
    - (10 points) Are you approached by bogeymen, snakewomen, and goblins when you come within range?
    - (10 points) Can you wield different weapons? Read scrolls? Check your inventory?
    - (5 points) Do you display player statistics (hit points remaining, etc.)?
    - (5 points) Do the levels have the feel of rooms connected by corridors?

    - (10 points) Overall completeness. While maybe not perfect in execution, does it have the overall feel of the game we required?
- Style/design/required elements (40 points total):
    - (10 points) Do goblins use a working recursive search algorithm to help decide how to move?
    - (10 points) Programming style. Is the code clear, simple, and straightforward? Are there useful comments that help clarify the code?
    - (10 points) Class design quality. Did you assign the project responsibilities to appropriate classes? Is the relationship of classes to each other sensible?
    - (10 points) Design documentation: Do you tell the purpose of and major responsibilities of each class and how it relates to other classes? Do you explain the non-trivial algorithms (how you generate a level (if it's not just one big room), how goblins move (if they use a clever algorithm), etc.)

Project 3 FAQ

1. **The spec says a level should appear as rooms connected by corridors. How do I do that?**

   The layout of levels is independent of everything else in this project, so you can save it for last if you want to, just having each level be one large room until you're ready to start working on having rooms and corridors. When you are ready to, you could generate a few random nonoverlapping rectangles, pick spots on their edges, and see if looking from there in the direction away from the rectangle hits another rectangle (and if not, maybe does with a right angle turn somewhere, if you want to get fancy). Other approaches are possible: experiment. Your goal is to have it look something like the posted games, with a reasonable number of rooms of reasonable sizes, connected by corridors, but while experimenting, make simplifying assumptions to help you work out ideas: Try having just two rooms, or don't worry initially about the rectangles being nonoverlapping.

2. **What are the various strings the sample program prints?**

   Here are some:

```
"swings mace at"
"slashes short sword at"
"swings long sword at"
"strikes magic fangs at"
"chops magic axe at"

" dealing a final blow."
" and hits, putting "    " to sleep."
" and hits."
" and misses."

"scroll of teleportation"  "You feel your body wrenched in space and
time."
"scroll of strength"  "Your muscles bulge."
"scroll of enhance armor"  "Your armor glows blue."
"scroll of enhance health"  "You feel your heart beating stronger."
"scroll of enhance dexterity"  "You feel like less of a klutz."

"the golden idol"

"You are wielding "
"You can't wield "
"You read the scroll called "
"You can't read a "
```

```
"Your knapsack is full; you can't pick that up."
"You pick up "
"You pick up a scroll called "

"Congratulations, you won!"
```