EECS 281 – Winter 2014 **v2.00** (2/17/14)[1]

Project 2

Revenge of the Living Deadline![2]

Due Tuesday, March 11th 11:55pm

Overview

Zombies are invading!

After coding for 7 years, a warlock has taken over the land. Waking up, you find yourself surrounded by zombies in a dark town. You're surrounded, but as the Hero of EECS 281, you have your trusty bow and quiver after quiver of arrows. You need to hold off the zombie horde - and hope that they don't kill you while you're grabbing your next quiver.

You will find a summary of the project functionality in the form of a 'Linux manual page' in Appendix A.
For changes,

Project Goals

- Understand and implement several kinds of priority queues
- Be able to independently read, learn about, and implement an unfamiliar data structure
- Be able to develop efficient data structures and algorithms
- Implement an interface that uses templated "generic" code
- Implement an interface that uses inheritance and basic dynamic polymorphism
- Become more proficient at testing and debugging your code

Gameplay - Zombies & Player

A zombie is defined by five attributes:

- A name – Names will always be unique in our test cases (and you do not need to check for this).
- An integer speed (>0) at which the zombie moves toward you
- An integer distance between the zombie and the player
- An integer describing the zombie's health and damage – as the zombie weakens, it does less damage
- The number of rounds the zombie has been active. This is measured as number of rounds, from and including the round it was created, to and including the round it was shot or when the game ends. If the zombie was created in round 2 and shot or the player was eaten in round 5, it was active for 4 rounds (rounds 2, 3, 4, and 5).

The zombies emit high-pitched screams, freezing you in your tracks. Therefore the player is stationary throughout the game; only zombies can move. The player must prioritize, using a priority queue, how to shoot the zombies in order to survive for as long as possible; in particular, you should approximate their *integer* Estimated Time of Arrival (ETA) using:

[1] See Appendix E: Patch Notes
[2] Plot credits: Spencer Kim

```
ETA = distance/speed;
```

While there may be more accurate formulas for determining when zombies arrive, you should not use them, as this would cause your output to differ from that of the autograder. In the event of ties in ETA, zombies with lexicographically smaller names should be shot first (use `std::string`'s operator< for this). For example, if FoxMcCloud and FalcoLombardi both have a priority of 3, FalcoLombardi would be shot first because `"FalcoLombardi"` < `"FoxMcCloud"`.

The player operates in 'rounds', beginning with round 1. Each round the player can shoot as many zombies as they have arrows. At the start of every round, including the first round, the player has a full quiver. Between rounds, when the player is reloading (fetching the next quiver), the zombies move in closer, with each zombie's new distance being calculated as follows:

```
random_offset = (rand()%speed)/2;
new_distance = distance – speed - random_offset;
if(new_distance < 0) new_distance = 0;
```

(If you worry about subtracting feet/sec from feet, imagine that speed is multiplied by time_per_round measured in seconds; since this value is 1 for our project, we removed it from the formula).

Note: You should perform this update on the zombies in the order in which they were initially created (see input section below). You should not update/generate random numbers for zombies that are inactive (not entered the scene or dead). The order is extremely important because pseudo-random number generation is taking place. This will require that you keep a data structure other than your priority queue, referred to in this document as 'the master list' to have all of the zombies available in the order of their creation. Your priority queue should then refer to elements inside this list (see Appendix B). One of the possibilities for this master list is to use `std::list`. This is convenient because if you get a pointer to an element inside a std::list, it is still valid (the element doesn't move around in the memory) when you modify the list.

Should a zombie's distance ever be 0, the zombie attacks the player, dealing damage equal to its health: The player's health is reduced by the current health of the zombie. If the player takes `<player_health>` or more damage, the player dies, and the game is lost. On the other hand, if the player manages to kill all of the zombies, the player wins. Note that a zombie does not die until its health reaches 0.

Gameplay - Arrows

The player has different types of arrows: Normal and Light, and for each round, the player has <quiver_capacity> arrows. Which of these arrows is used is designated by command line input.

- Normal arrows deal 1 damage.
- Light arrows instantly kill zombies.

Game behavior is also dependent on the arrow mode, which is given on the command line.

- If the `NORMAL` option is specified on the command line, the player's quiver consists of Normal arrows.
- If `LIGHT` is specified, the player shoots only Light arrows. However, `LIGHT` mode - and `LIGHT` mode only - zombies have a death touch: If a zombie's distance becomes 0 or less in `LIGHT` mode, you die instantly.

See 'Round Breakdown' in Appendix C for further clarifications.

Input/Output

Game Settings (Input):

Game Settings will be given from an input file, 'GAMEFILE'. Note that unlike project 1, this is **not** from standard input (cin), but rather, from a file specified on the command line. You should use an ifstream to handle input.

The file format is as follows:

You will first be presented with the following in precisely this order:

- '`Quiver_Capacity`' - Integer number arrows the player can shoot (number of zombies the player can shoot) in a round. Arrows always hit their target. Arrows do not accumulate between rounds.
- '`Random_Seed`' - Integer random seed to be used with `std::srand(unsigned int)` before you generate any pseudo-random numbers with `rand()`. `std::srand` should only be called once in your program. (See 'Random Generation of Zombies')
- '`Max_Rand_Distance`' - Integer maximum starting distance for randomly generated zombies (though explicitly specified zombies may start farther away).
- '`Max_Rand_Speed`' - Integer maximum zombie speed for randomly generated zombies (though explicitly specified zombies may be faster).
- '`Max_Rand_Health`' - Integer maximum zombie health points for randomly generated zombies (though explicitly specified zombies may have more health).
- '`Player_Health`' - Integer number designating the health points the player has. The amount of damage the player can take and still live is <`Player_Health`>-1.

After which, the file will contain:

- A delimiting line, which will always be '`---`', and divides the input for different rounds
- '`Round`' - Integer number, dictating what round the following zombies should be created in.
    - Round numbers will only increase as you go down in the file (we will not give you round 3's information then round 1's), this allows certain input optimizations to be made.
    - Zombies that are still active at the end of previous rounds should remain active at the start of the next.
- '`Num_Zombies`' - Integer number of zombies to randomly generate (this does not include the 'explicitly specified' zombies).
- An optional list of explicitly specified (non-random) zombies which follow the following format, each line of this format being a new zombie:
    - `START_DISTANCE<SPACE>SPEED<SPACE>HEALTH<SPACE>NAME`
    - Where <SPACE> will be an actual space character in the file.
    - Where `NAME` is a string that may only contain alphanumeric characters and '_'. Names will always be unique in our test cases (and you do not need to check for this).
    - Explicitly specified zombies are always created **after** the random ones.

Example:

```
Quiver_Capacity: 10
Random_Seed: 2049231
Max_Rand_Distance: 50
```

```
Max_Rand_Speed: 60
Max_Rand_Health: 1
Player_Health: 10
---
Round: 1
Num_Zombies: 25
150 300 15 FoxMcCloud
2 3 6 FalcoLombardi
100 1 100 SlippyToad
---
Round: 3
Num_Zombies: 50
20 10 20 DarkLink
```

You may assume that the Game Settings input is **correctly formatted** (and do not need to error check it).

Random Generation of Zombies[3]:

Before generating any random numbers with rand(), call `std::srand()` with the `Random_Seed`. Call this function only once, at the beginning of your program.

You should generate random zombies before the player starts shooting zombies but after existing zombies have been updated. When randomly generating zombies, you should generate them as follows:

```
start_distance = rand() % Max_Rand_Distance + 1;
speed = rand() % Max_Rand_Speed + 1;
health = rand() % Max_Rand_Health + 1;
name = "AI";
name += to_string(counter++);
```

Where `counter` is a number that starts at 0 and is only incremented (it is never reset).

Note: You should always determine the zombie's `start_distance`, `speed`, and `health`, in that order, as this will affect the result of the pseudo-random number generation.

Also: Explicitly specified zombies are always created **after** the random ones.

Command Line (Input):

Your program `rotld` (Revenge of the Living Deadline) should take the following case-sensitive command-line options:

- `-h, --help`
  Print the manual page information provided in Appendix A and exit(0).
- `-c, --container`
  Required option. Changes the type of priority queue that your implementation uses at runtime. Must be

[3] We have seen unsupported systems (Mac/Windows/etc) generate random numbers differently than CAEN Linux, which has led to incorrect output for local testing.

one of `BINARY`, `POOR_MAN`, `SORTED`, or `PAIRING`.
NOTE: It is a violation of the honor code to misrepresent your code or submit code that uses a priority queue implementation different from what is asked. (i.e., if you don't finish a particular implementation, your code should immediately terminate when we invoke it with that container).

- `-a, --arrow`
  Required option. Indicates that your program should use the designated arrow set - one of `NORMAL` or `LIGHT`. You can assume this command line flag will appear exactly **once**.
- `-v, --verbose N`
  An optional flag that indicates the program should print additional output statistics (see the Output section). This option takes a required argument, N, which is an integer value greater than 0.
- `-d, --debug`
  An optional flag that indicates the program should print additional output statistics (see the Testing and Debugging section).

`rotld` also takes a mandatory file argument (whenever the help option is not specified), 'GAMEFILE', which will always be given as the very last argument on the command line. If 'GAMEFILE' is not specified on the command line, you should print an error message to cerr and either return 1 from main or exit(1). See the "Print any remaining command-line arguments" section of this example for how to get the name of the GAMEFILE.

Examples of legal command lines:

- ./rotld --container BINARY -a LIGHT infile.txt
- ./rotld --verbose 15 -c PAIRING --arrow NORMAL infile.txt > outfile.txt

Examples of illegal command lines:

- ./rotld --container BINARY --arrow LIGHT < infile.txt
  - No input file was given on command line. We are **not** reading input from standard input with input redirection in this project.
- ./rotld --arrow NORMAL infile.txt
  - No container type was specified. Container type is a required option.
- ./rotld -c SORTED infile.txt
  - No arrow type was specified. Arrow type is a required option.

We will not be error checking your command-line handling.

Output:

The output of `rotld` should be as follows:

```
VICTORY IN ROUND <ROUND_NUMBER>! <NAME_OF_LAST_ZOMBIE_KILLED> was the last
zombie. You survived with <PLAYER_HEALTH> health left.
```

If the player lives after all zombies have been killed and no new zombies will be generated in a later round. And:

```
DEFEAT IN ROUND <ROUND_NUMBER>! <NAME_OF_ZOMBIE_THAT_ATE_PLAYER'S_BRAINS> ate
your brains!
```

If the player is killed. (Note: this output should be printed all on one line [without the newline character in the middle]). If multiple zombies would be able to eat your brains in the same round, you should print the name of the one who was updated first.

Example:

```
DEFEAT IN ROUND 2! FoxMcCloud ate your brains!
```

If and only if the '`--verbose N`' option is specified on the command line, the following additional output should be printed after the 'DEFEAT' or 'VICTORY' line in the following order without any blank lines separating them:

- The number of zombies still active at the end

This should be printed in the format:
`Zombies still active: NUMBER_OF_ZOMBIES`

Where `NUMBER_OF_ZOMBIES` is the number of zombies still active.

- The names of the first `N` zombies that were killed, followed by a space, and then the number (`1,2,3,...,N`) corresponding to the relative order they were killed in. These zombies should be printed in order, such that the very first one killed is printed first, and the `N`th first one is printed `N`th.
- The names of the last `N` zombies that were killed, followed by a space, and then the number (`M,M-1,...,1`) corresponding to the relative order they were killed in, where `M = min(N, total_number_of_zombies_killed)`. These zombies should be printed in order, such that the very last one killed is printed first, and the `N`th-to-last zombie is printed `N`th.

- The names of the `N` zombies who were active for the most number of rounds a space and then the number of rounds. These zombies should be printed in order such that the zombie who was in the most rounds appears first. In the event that there is more than one zombie who survived for the same amount of time, you should print the one who has a lexicographically smaller name first (use `std::string`'s < operator for this).
- The names of the `N` zombies who were active for the least number of rounds, but at least one round, followed by a space and then the number of rounds. These zombies should be printed in order such that the zombie who was in the least rounds appears first. Break ties using the same lexicographic rule as before.

Hint: The last two statistics (that deal with how many rounds a zombie has been active) can be calculated with a linear (in terms of the total number of zombies) or better average runtime complexity. We expect you to be able to produce these statistics in better than O($total\_number\_of\_zombies^2$) time.

If for any of the above statistics you do not have `N` zombies to print, you should print the data for as many as you can.

Example: If the program is run with '`--verbose 10`', there were only three zombies in the game, and you killed FoxMcCloud before FalcoLombardi and they were both active for 3 rounds and then you killed DarkLink in round 5, his fourth round of being active, you should print:

```
First zombies killed:
FoxMcCloud 1
FalcoLombardi 2
DarkLink 3
Last zombies killed:
DarkLink 3
FalcoLombardi 2
FoxMcCloud 1
Most active zombies:
DarkLink 4
FalcoLombardi 3
FoxMcCloud 3
Least active zombies:
FalcoLombardi 3
FoxMcCloud 3
DarkLink 4
```

Note that '`FalcoLombardi`' < '`FoxMcCloud`' lexicographically and that some zombies may be included in several lists.

Implementations of Priority Queues

We have provided a header file, `eecs281priority_queue.h` on ctools. For this project, you are required to implement and use your own priority queue containers. You will implement a *binary heap*, a '*poor man's priority queue*', a '*sorted array priority queue*', and a *pairing heap* that compile with the interface given in `eecs281priority_queue.h`. To implement these priority queue variants, you will need to fill in separate header files, `binary_heap.h`, `poorman_priority_queue.h`, `sorted_priority_queue.h`, and `pairing_heap.h`, containing all the definitions for the functions declared in `eecs281priority_queue.h`. We have provided these files with empty function definitions for you to fill in. These files will also specify more information about each priority queue type, including runtime requirements and a general description of the container.

You are **not** allowed to modify `eecs281priority_queue.h` in any way. Nor are you allowed to change the interface (names, parameters, return types) of the functions that we give you in any of the provided headers. You are allowed to add your own private helper functions and variables as needed, so long as you still follow the requirements outlined in both the spec and the comments in the provided files. You should not construct your program such that one priority queue implementation's header file is dependent on another priority queue implementation's header file.

The *poor man's priority queue* implements the priority queue interface using an unordered array-based container. This implementation is simple and supports supporting adding new items in amortized O(1) time, finding the minimum in O(n) time via linear search and update the priority of any element inside in O(1) time (given its location).

The *sorted array priority queue* implements the priority queue interface while keeping all elements in sorted

order at all times.

For binary heaps, we suggest reviewing Chapter 6 of the CLRS book. They will also be covered in lecture.

You are expected to do your own reading to figure out how to implement pairing heaps. For pairing heaps you will find recommended (but not required) reading in the:
EECS 281 001 W14 Resources/Projects/Project 2
folder on ctools. We have provided an extract of a textbook by Sartaj Sahni as well as the original paper on pairing heaps, by Fredman et al.

Note: We may compile your priority queue implementations with our own code to ensure that you have correctly and fully implemented them. To ensure that this is possible (and that you do not lose credit for these test cases), do not define your main function in one of the priority queue headers.

Libraries and Restrictions

You **are** allowed to use std::vector, std::list and std::deque.

You **are** allowed to use the <algorithm> library, but with **exceptions**:

- You are **NOT** allowed to use std::partition, std::partition_copy, std::stable_partition, std::make_heap, std::push_heap, std::pop_heap, std::sort_heap, std::lower_bound, std::upper_bound, std::equal_range, or std::binary_search.
- You **are** allowed to use std::sort.

You are **NOT** allowed to use std::qsort.
You are **NOT** allowed to use other STL containers. Specifically, this means that use of std::stack, std::queue, std::priority_queue, std::forward_list, std::set, std::map, std::unordered_set, std::unordered_map, and the 'multi' variants of the aforementioned containers are forbidden.
You are **NOT** allowed to use the C++11 regular expressions library (it is not fully implemented on gcc) or the thread/atomics libraries (it spoils runtime measurements).
You are **NOT** allowed to use other libraries (eg: boost, pthread, etc).

Furthermore, you may **not** use any STL component that trivializes the implementation of your priority queues (if you are not sure about a specific function, ask us).

Testing and Debugging

Part of this project is to prepare several test cases that will expose defects in the program. **We strongly recommend** that you **first** try to catch a few of our buggy solutions with your own test cases, before beginning your solutions. This will be extremely helpful for debugging. The autograder will also now tell you if one of your own test cases exposes bugs in your solution.

Each test case should consist of a `GAMEFILE` file. We will run your test cases on several buggy project solutions. If your test case causes a correct program and the incorrect program to produce different output, the test case is said to expose that bug.

Test cases should be named `test-n-CONTAINER-ARROW.txt` where 0 < n <= 15. The autograder's buggy solutions will run your test cases in the specified `CONTAINER` (ie: test-1-BINARY-LIGHT will run your test on the binary heap with Light arrows).

Your test cases may have no more than 30 lines in any one file and may generate no more than 50 zombies (counting both random and explicitly specified zombies). You may submit up to 15 test cases (though it is possible to get full credit with fewer test cases). Note that the tests the autograder runs on your solution are **NOT** limited to 30 lines in a file; your solution should not impose any size limits (as long as sufficient system memory is available).

Testing `pairing_heap.h`: We will be testing your pairing heap implementation in isolation from the rest of your code (in addition to in the context of the zombie game). Specifically, we will be testing the functionality and runtime of your 'updateElt' implementation. We strongly recommend that you create test cases for your local use to evaluate whether or not your code is correct and performs well. For these tests, we recommend doing tests where update operations are much more frequent than removal operations.

When debugging, we highly recommend setting up your own system for quick, automated, regression testing. In other words, check your solution against the old output from your solution to see if it has changed. This will save you from wasting submits. You may find the Linux utility 'diff' useful as part of this.

In addition, we acknowledge that random input can be difficult to debug at times. If you are stumped on how to debug your code, you could start by implementing the --debug command line flag: If --debug is specified, at every round, print the following:

"Round <round#>".
For each zombie updated, print "Moved: " and its name and new distance from the player.
After a zombie is created, print "Created: " and its name and distance from the player.

For example, your output with the --debug flag may look like:

```
Round 1
Created: FoxMcCloud 10
Created: FalcoLombardi 2
Round 2
Moved: FoxMcClound 0
Moved: FalcoLombardi 0
DEFEAT IN ROUND 2! FoxMcCloud ate your brains!
```

Submitting to the Autograder

Do all of your work (with all needed files, as well as test cases) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- You have deleted all .o files and your executable(s). Typing '`make clean`' shall accomplish this.
- Your makefile is called Makefile. Typing '`make -R -r`' builds your code without errors and generates an executable file called `rotld`. (Note that the command-line options -R and -r disable automatic build rules, which will not work on the autograder).
- Your Makefile specifies that you are compiling with the gcc optimization option -O3. This is extremely important for getting all of the performance points, as -O3 can often speed up code by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that

compiles with -g, as this will slow your code down considerably. Note: If your code “works” when you don’t compile with -O3 and breaks when you do, it means you have a bug in your code!
- Your test case files are named `test-n-CONTAINER-ARROW.txt` and no other project file names begin with test. Up to 15 test cases may be submitted.
- The total size of your program and test cases does not exceed 2MB.
- You don't have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (i.e. the .git folder used by git source code management).
- Your code compiles and runs correctly using version 4.7.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 4.7.0 running on Linux (students using other compilers and OS did observe incompatibilities). Note****: In order to compile with g++ version 4.7.0 on CAEN you must put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-4.7.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-4.7.0/lib64
LD_RUN_PATH := /usr/um/gcc-4.7.0/lib64
```

Turn in all of the following files:
- All your .h and .cpp files for the project
- Your Makefile
- Your test case files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run this command:

```
dos2unix *; tar -czf submit.tar.gz *.cpp *.h Makefile test-*.txt
```

This will prepare a suitable file in your working directory.

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. You can safely ignore and override any warnings about an invalid security certificate. **Note that when the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to three times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). We will count only your last submission for your grade. Part of programming is knowing when you are done (when you have achieved your task and have no bugs); this is reflected in this grading policy. We realize that it is possible for you to score higher with earlier submissions to the autograder; however this will have no bearing on your grade. We strongly recommend that you use some form of revision control (ie: SVN, GIT, etc) and that you ‘commit’ your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and CTools regarding the use of version control.

Please make sure that you read all messages shown at the top section of your autograder results! These messages will help explain some of the issues you are having (such as losing points for having a bad Makefile).
Also be sure to note if the autograder shows that one of your own test cases exposes a bug in your solution (at the bottom).

Grading

90 points -- Your grade will be derived from correctness and performance (runtime). This will be determined by the autograder.
10 points -- Test case coverage (effectiveness at exposing buggy solutions).

We also reserve the right to deduct up to 5 points for bad programming style, code that is unnecessarily duplicated, etc.

Refer to the Project 1 spec for details about what constitutes good/bad style, and remember:

It is **extremely helpful** to compile your code with the gcc options: -Wall -Wextra -pedantic. This will help you catch bugs in your code early by having the compiler point out when you write code that is either of poor style or might result in unintended behavior.

Appendix A: Linux Manual Page for rotld

Note: This is intended as a *SUMMARY* of the basic functionality of each command. More detail on how they should be implemented is given earlier in the spec. ***We are not without fault, if something here does not precisely match what is in an earlier section, the description given in the earlier section should be used.

```
NAME
      rotld - a command line ‘Revenge of the Living Deadline’ game simulation program.
SYNOPSIS
      rotld (-c | --container) CONTAINER (-a | --arrow ) ARROW [-v | --verbose N] GAMEFILE
      rotld (-h | --help)
DESCRIPTION
      ROTLD (Revenge of the Living Deadline) game simulator with both zombie and player
      AI. This program acts as the player in a ROTLD game and shoots zombies in the
      optimal ordering to achieve the best possible result (stay alive for as long as
      possible).
OPTIONS
-h, --help
      Print this help screen and exit.
-c, --container
      Required option that states the type of priority queue to use. Must be one of
      BINARY, POOR_MAN, SORTED, or PAIRING.
-a, --arrow
      Required option that states the type of arrows to use. Must be one of NORMAL, LIGHT.
-v, --verbose N
      An optional option that indicates the program should print additional output
      statistics.
-d, --debug
      An optional option that indicates the program should print some debugging output.
```

Appendix B: Using a container of pointers

You might recall from EECS 280 that storing large values in a container can sometimes lead to slow copies. For example, in the following code, a list insertion results in at least one copy.

```
class Gorilla {
  // OVERVIEW: a big, expensive class ...
};
int main() {
  List<Gorilla> zoo;
  zoo.insertFront(Gorilla());
}
```

An alternative is to use a container of pointers that point to objects on the heap. This way, inserting and removing items from the container is done with pointers and is much faster for some programs.

```
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla);
```

Now, the code that creates the objects on the heap is also responsible for removing them. This is because containers do not manage memory outside of themselves. The container couldn't possibly know what you wanted to do!

```
Gorilla *g = zoo.removeFront();
delete g;
```

Containers of pointers are subject to two broad classes of bugs: using an object after it has been deleted, and leaving an object orphaned by never deleting it. You can avoid these problems with a pattern of use that follows these three rules:

Existence: Allocate a Gorilla before inserting it into any container.

```
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla);
```

Ownership: Once it is inserted, do not modify it until it is removed.

```
List<Gorilla*> zoo;
Gorilla *colo = new Gorilla;
zoo.insertFront(colo);
colo->set_name("Bill"); //bad!
```

Breaking the ownership rule: you can modify a pointer inside a container if you tell the container to update!

Conservation: Once it is removed, it must either be deallocated or inserted into some container. Don't forget to delete at the end!

```
// bad example
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla);
zoo.removeFront(); //bad! memory leak!

// good example
List<Gorilla*> zoo;
```

```
zoo.insertFront(new Gorilla);
```
Gorilla *g = zoo.removeFront();
delete g; **//fixed**

Example: Using a poor man's heap of pointers. This assumes that you have implemented the poorman's priority queue correctly.

```
#include <iostream>
#include <vector>
#include "poorman_priority_queue.h"
using namespace std;

// Comparison functor for integer pointers
struct intptr_comp {
  bool operator() (const int *a, const int *b) const {
    return *a < *b;
  }
};

int main() {
  //Pointer to a container of pointers-to-int in priority order, using
  //polymorphism.  The int objects live in dynamic memory (THE heap)
  eecs281priority_queue<int *, intptr_comp> *pq;
  pq = new poorman_priority_queue<int *, intptr_comp>;

  pq->push(new int(10));
  pq->push(new int(5));
  pq->push(new int(20));
  pq->push(new int(7));

  //Container of pointers to odd elements found while processing the pq
  vector<int*> odds;

  //Process each number in priority order, removing evens and remembering odds
  while (!pq->empty()){

    //Pop one int pointer off the priority queue
    //At this point, the integer itself is still alive in dynamic memory
    int *z = pq->top();
    pq->pop();
    cout << *z << ' ';

    //Check even or odd
    if (*z % 2)
      odds.push_back(z); //Remember odds
    else
```

```
      delete z;           //Delete evens
    }

    cout << "\nstatistics: found " << odds.size() << " odds\n";

    //remember to delete the odds or they will be leaked!
    for (auto p : odds)
      delete p;

    //delete the priority queue itself
    delete pq;

    return 0;
}
```

Appendix C: Round Breakdown

The following is a summary of the actions that need to occur in a round and the order in which they should happen:

1. Player refills quiver
2. Active zombies advance towards the player, updated in the order that they were created.
    a. Update one Zombie
    b. If the --debug flag is enabled, print the zombie name and location, along with "Moved:" For example, `Moved: AI1 5`
    c. If the zombie has arrived at the player (distance=0), it deals damage to the player.
        i. In NORMAL mode, the damage is equal to the zombie's current health.
        ii. In LIGHT mode, the damage is equal to the player's current health. It is fatal.
    d. If the player dies (health <= 0), the zombie that "dealt the killing blow" is the one that eats the player. Note: you still need to update the other zombies; don't exit the game yet!
3. New zombies appear
    a. Random zombies are created.
    b. Explicitly specified zombies are created.
    c. If the --debug flag is enabled, print new active zombie names and distances in the order they were created, along with "Created." For example, `Created: FoxMcCloud 150`
4. At this point, if the Player was killed in Step 2, the game ends.
    a. Print any required messages and statistics.
5. Player shoots zombies.
    a. Shoot arrows until your quiver is empty.
        i. Shoot at the Zombie with the lowest ETA.
        ii. If he remains active, keep shooting at the same zombie.
        iii. Repeat
6. If there are no more zombies and none will be generated in a future round, the player has won the game.
    a. Print any required messages and statistics. The game ends.

Appendix D: Full Output with --verbose and --debug

Here is a sample input and output of `rotld` with the --debug and --verbose flags:

sample.txt:

```
Quiver_Capacity: 10
Random_Seed: 2049231
Max_Rand_Distance: 50
Max_Rand_Speed: 60
Max_Rand_Health: 1
Player_Health: 10
---
Round: 1
Num_Zombies: 25
150 300 15 FoxMcCloud
2 3 6 FalcoLombardi
100 1 100 SlippyToad
---
Round: 3
Num_Zombies: 50
20 10 20 DarkLink
```

Full Output, when ran with ./rotld -c POOR_MAN -a NORMAL -v 10 -d sample.txt :

```
Round: 1
Created: AI0 21
Created: AI1 35
Created: AI2 25
Created: AI3 17
Created: AI4 2
Created: AI5 2
Created: AI6 14
Created: AI7 17
Created: AI8 16
Created: AI9 4
Created: AI10 31
Created: AI11 45
Created: AI12 18
Created: AI13 27
Created: AI14 18
Created: AI15 9
Created: AI16 2
Created: AI17 22
Created: AI18 39
Created: AI19 13
Created: AI20 27
```

```
Created: AI21 22
Created: AI22 20
Created: AI23 8
Created: AI24 7
Created: FoxMcCloud 150
Created: FalcoLombardi 2
Created: SlippyToad 100
Round: 2
Moved: AI1 5
Moved: AI3 15
Moved: AI4 0
Moved: AI5 0
Moved: AI6 0
Moved: AI7 0
Moved: AI8 0
Moved: AI9 0
Moved: AI11 38
Moved: AI17 0
Moved: AI18 16
Moved: AI21 0
Moved: AI22 15
Moved: AI23 0
Moved: AI24 6
Moved: FoxMcCloud 0
Moved: FalcoLombardi 0
Moved: SlippyToad 99
DEFEAT IN ROUND 2! FoxMcCloud ate your brains!
Zombies still active: 18
First zombies killed:
AI0 1
AI10 2
AI12 3
AI13 4
AI14 5
AI15 6
AI16 7
AI19 8
AI2 9
AI20 10
Last zombies killed:
AI20 10
AI2 9
AI19 8
AI16 7
AI15 6
AI14 5
AI13 4
```

AI12 3
AI10 2
AI0 1
Most active zombies:
AI1 2
AI11 2
AI17 2
AI18 2
AI21 2
AI22 2
AI23 2
AI24 2
AI3 2
AI4 2
Least active zombies:
AI0 1
AI10 1
AI12 1
AI13 1
AI14 1
AI15 1
AI16 1
AI19 1
AI2 1
AI20 1

Appendix E: Patch Notes

v2.00 (2/17/14):

- Added clarifications such as another line specifying where to use srand().
- **Removed** FIRE, ICE and RANDOM arrows/modes.
- Arrow type is a **required** option. You can assume it'll appear on the command line exactly once.
- Distance does **NOT** go below 0. If distance < 0, distance = 0.
- Sample Output is now **Full Output**, which has the same test case but a **new** command line input as well as more useful output. We hope the new Full Output can help you verify your random number generation. Note: the old Sample Output was incorrect.
- You **are** allowed to use <algorithm>, but with **exceptions** (see the Libraries and Restrictions section).
- **--debug** mode is now required (but easy!), has additional output, and more specific instructions. This will help you make sure your use of random number generation matches ours.
- Clarification in Appendix B: Using a container of pointers: modifying objects without removing their pointers from container. It's a OK, just be sure to update the container.
- Clarifications in Appendix C

v1.12 (2/14/14):
Removed the LIGHT case for Round Breakdown; see Gameplay - Arrows.

v1.11 (2/13/14):
Fixed the --arrows typo: The command line flag is --arrow.

v1.10 (2/12/14):
Modified Appendix C to show the proper round breakdown.
Removed the burning/frozen "stacks" and replaced it with the burning/frozen status.

v1.01 (2/11/14):
Fixed some typos.

v1.00 (2/11/14):
Released.