# EECS 281 – Fall 2014
# Programming Assignment 2 (version 1.1)
# Mine All Mine[1]

Due Thursday, October 23rd 11:55pm

## 1. Overview

You are an adventurous gold miner. However, in your avarice you've ignored several safe-tunneling practices, and a recent earthquake has left you trapped! Luckily, out of paranoia, you always carry ridiculous quantities of dynamite sticks with you. You need to blast your way out and make the most of each dynamite stick by blasting the piles of rubble which take the fewest sticks to destroy.

### 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.

## 2. Breaking Out of the Mine

The mine you are trapped in can be represented with a 2-dimensional grid. There are 2 types of tiles:

- Tiles containing rubble.
    - Think of cleared tiles as containing 0 rubble.
- Tiles containing TNT.

You (the miner) start on a specified tile. At every iteration, you will attempt to blast away the "easiest" tile you can "access", until you escape!

**Definition of Visibility**

By default, your starting tile is always visible. Once on a tile, no matter where you are in the map, there are only four tiles that are visible from your current tile: up, down, left and right (except for edge-of-map conditions). Another way to think of it is this: the mine is dark, and you cannot see very far.

[1] Credits: David Paoletti, Marcus Darden, Ian Lai, Spencer Kim, Brian Wang

**Definition of Movement**
You must always add any visible tiles to a priority queue (more on this below). As in Project 1, make sure you don't add tiles twice. The priority queue will always tell you what your "next" tile will be. Movement is taking the "next" tile from the priority queue and making it your "current" location. After moving, you can then see any of the four tiles visible from your new location, add them to the priority queue, etc. You use the priority queue to remember tiles that you have seen, but have not yet moved to.

**Definition of Easiest Tile**
The easiest tile you can reach is defined as follows, in the stated order:
1. Any TNT tile you can reach.
2. The lowest rubble-value tile you can reach.

**Tie-breaking**
In the event of ties (two TNT tiles or two rubble tiles with the same rubble-value):
1. Clear the tile with the lower column coordinate first.
2. If the tiles have the same column coordinate, clear the tile with the lower row coordinate.

**Clearing Tiles**
When clearing away rubble tiles, the tile simply turns into a cleared tile.
When clearing away TNT tiles, the following happens:
- The TNT tile becomes a cleared tile.
- All tiles touching the TNT tile are also "cleared"
    - If a TNT tiles is touching another TNT tile, this will cause a chain explosion.
    - Again, diagonals are not considered for TNT explosions.

**Definition of Escape**
The miner escapes when his or her current tile has been cleared, and is at the edge of the grid.

# 3. Example

In the following example, you start at position `[1,2]` (row 1, column 2). The tiles that the miner can reach are shown in **bold and red**. Positive integers signify rubble tiles (0 is a clear tile) and the negative integer -1 signifies a TNT tile.

**Please note:** This example mine is for illustrative purposes only and does not conform exactly to the input file format described in the later section**.** Specifically, there are **bold** numbers that refer to row and column number - they are not a part of the actual grid.

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100  10  20  15
2  20  15   5   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

At the first iteration, the only tile visible to the miner is the starting location, [1,2]

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100   0  20  15
2  20  15   5   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

Next, the miner will clear tile `[2,2]` because there are no TNT tiles in view, and it is has the lowest rubble-value. Clearing that tile has allowed us to reach more tiles!

```
    0   1   2   3   4
0  20  20  20  20  20
1  20  20   0  20  15
2  20  15   0   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

Here, our miner sees two TNT tiles! However, due to the tie-breaking rules, the miner will choose to blow up the TNT at [4,2] instead of [3,3] because tiles with lower column coordinates are favored. Blowing up the TNT tile at [4,2] clears all the tiles touching it, creating a chain reaction with the TNT tile at [4,1]. After all the TNT explosions have been resolved, the grid looks like the following:

```
    0   1   2   3   4
0  20  20  20  20  20
1  20  20   0  20  15
2  20  15   0   0  20
3  20   0   0  -1 100
4   0   0   0   0  20
```

Since the current location is [4,2], and this tile is cleared, we have escaped!

# 4. TNT Explosions

As you will see in **section 8** (output), you need to keep track of the order in which tiles are cleared. When a TNT tile detonates, **all tiles** that are cleared as a result of the TNT detonation (including chain reactions) are cleared in order from "easiest" to "hardest" (as defined in section 2, including tie-breaker rules). As stated in **section 2** (Breaking Out of the Mine), do **NOT** consider diagonals; even TNT only destroys piles up rubble up, down, left and right of it.

When a TNT detonation occurs, you should use the priority queue type specified on the command line to determine detonation order. Push all the detonated tiles into a new priority queue, and then pop them out in priority order. You need to use some type of priority queue, because TNT blows up other TNT first, followed by smaller piles of rubble, etc.

Notice that, as you progress through your TNT priority queue, you may blow up a tile that is already waiting in your primary priority queue. If this happens, you have to make sure not to "clear" that tile twice, and will also have to update the primary PQ; see **section 9** below for more details, specifically the `fix()` member function.

# 5. Command Line

Your program **`MineEscape`** should take the following case-sensitive command-line options:

- `-h, --help`
  - Print a description of your executable and `exit(0)`.
- `-c, --container`
  - Required option. Changes the type of priority queue that your implementation uses at runtime. Must be one of `BINARY, POOR_MAN, SORTED,` or `PAIRING`.
  - **NOTE: It is a violation of the honor code to misrepresent your code or submit code for a grade 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).** However, for testing purposes you may find it useful to submit preliminary versions that do not conform to this. This is allowed as long as you clearly place a comment near the start of your main function that indicates this is only a test version not intended for final grading.
- `-s, --stats N`
  - An optional flag that tells the program it should print extra summarization statistics upon termination. This optional flag has a required argument N. Details are covered in the output section.

MineEscape also takes a mandatory file argument (whenever the help option is not specified), `'MINEFILE'`, which will always be given as the very last argument on the command line. If `'MINEFILE'` 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 `MINEFILE`.

Examples of legal command lines:

- `./MineEscape --container BINARY infile.txt`
- `./MineEscape --stats 15 -c PAIRING infile.txt > outfile.txt`

Examples of illegal command lines:

- `./MineEscape --container BINARY < infile.txt`
  - No input file was given on command line. We are **not** reading input from standard input with input redirection in this project.
- `./MineEscape infile.txt`
  - No container type was specified. Container type is a required option.

**We will not be error checking your command-line handling, but we expect your program to accept any valid combination of input parameters.**

You are not required to check for any errors not specifically identified here in this project specification. You are not required to check for incorrect map input. We only test with properly formed input.

# 6. Input

In this project, you can assume the input file is **always correctly formatted**, so you can focus more on implementing your priority queues and your algorithm. However, you should still know how to correctly format input so you can self-test your project.

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

There will be two different types of input: map (M) and pseudo-random (R). Map input is for your personal debugging, but pseudorandom allows easier testing of large grids.

Map input mode (M)

Input will be formatted as follows:

- '`M`' - A single character indicating that this file is in map format.
- '`Size`' - Positive integer number that specifies the size of the square grid (20 means a 20 X 20 grid).
- '`Start`' - Coordinate indicating the starting location, row followed by column.

Map input consists of a map of all the tiles in the mine. Each tile will be separated from other tiles on the same line with whitespace (any number of spaces or tabs). There are 2 types of tiles:

- Rubble tiles which are signified by an integer between 0 and 999 (0 means the tile is already clear of rubble).
- TNT tiles, which are signified with the integer -1.

Tiles are indexed as follows:
- The tile in the top left corner is at location [0,0].
- The tile in the bottom right corner is at location [Size-1,Size-1].

Example of M input (starting location is at [2,2], or row 2 column 2):

```
M
Size: 5
Start: 2 2
   -1   11    9   19    2
   18   15   10    7    0
   18   -1   14    4   12
   17   13   17    4    2
   14    1   14    8   -1
```

Pseudorandom Mode (R)

Input will be formatted as follows:
- 'R' -character indicating that this file is in pseudorandom format.
- 'Size' - Positive integer number that specifies the size of the square grid.
- 'Start' - Coordinate indicating the starting location.
- 'Seed' - Number used to seed the random number generator.
- 'Max_Rubble'- The max rubble value a tile can have.
- 'TNT'- Chance that a generated tile will be a TNT tile (20 = 1 in 20 chance of a given tile being a TNT tile).

Example of R input:

```
R
Size: 5
Start: 2 2
Seed: 4
Max_Rubble: 20
TNT: 10
```

**Generating your grid in R mode:**
Included in CTools with the project spec is a file `P2random.h that` contains definitions for the

following functions:

```
void P2random::seed_mt(int seed);
// seeds the Mersenne Twister used for pseudorandom number generation.

int  P2random::generate_tile(int max_rubble, int tnt_chance);
// returns an integer that represents a tile of rubble
// -1 means this rubble tile is a TNT tile
```

In order to generate your grid in R mode, first call `P2random::seed_mt()` with the seed number from the input file. Then, you can generate tiles by calling `P2random::generate_tile()`. The first call to generate_tile should generate tile [0, 0]. The second call will generate tile [0,1] (row 0, column 1), etc. Fill each row before proceeding to the next.

The nth call to generate_tile should will generate tile
`[(n-1) / size, (n-1) % size]` where size is the size of the grid.

**Note that the example R and M input files given above are equivalent.** That is, *both* should generate the exact same map!

# 7. Output

### Default Output

After terminating, your program should **always** print:

`Cleared` **`<NUM>`** `tiles containing` **`<AMOUNT>`** `rubble and escaped.`

- **`<NUM>`** the number of tiles cleared. This number **does** include tiles cleared by TNT, but does not include the TNT tile itself.
- **`<AMOUNT>`** the total rubble cleared. This number **does** include rubble cleared by TNT, bu clearing (detonating) the TNT tile itself counts as 0 rubble cleared.

### Stats Output

After printing the default output, if `-s` or `--stats` option is specified print the following output: (Note: **N** is the argument given to the -s flag on the command line)

**A.** The line, **"**`First tiles cleared:`**"** (without quotes) followed by:
The first **N** tiles cleared in the following format:

**`<TILE_TYPE>`** `at [`**`<ROW>,<COL>`**`]`

`<TILE_TYPE>` the type of the tile being cleared. If it is a rubble tile, this is the rubble amount. If this is a TNT tile, print **"**`TNT`**"** without the quotes

`<ROW> <COL>` the coordinates of the tile being cleared.

**Remember**: when a TNT tile detonates or when there is a chain reaction, all the tiles are cleared from easiest to hardest. Refer back to **section 4** for more details.

**B.** The line, **"**`Last tiles cleared:`**"** (without quotes) followed by:
The last **N** tiles cleared in the same format as part **A**. The **last** tile cleared should be printed first, followed by the second last, etc.

**C.** The line, **"**`Easiest tiles cleared:`**"** (without quotes) followed by:
The **N** easiest tiles you blew up in the same format as part **A** in *descending order* (easiest tile followed by next easiest tile)

**D.** The line, **"**`Hardest tiles cleared:`**"** (without quotes)' followed by:
The **N** hardest tiles you blew up in the same format as part **A**. in *ascending order* (hardest tile followed by next hardest tile)

These statistics should be calculated as fast as possible, regardless of the specified container type. In other words, if your program was invoked with `--container POOR_MAN`, you must use the `POOR_MAN` priority queue for planning which tiles you will blast to escape from the mine, but you could use faster priority queues such as `BINARY` or `PAIRING` for tracking statistics.

If you have cleared less than N tiles, then simply print as many as you can.

# 8. Full I/O Example

## Input (mine.txt):

Equivalent Input files (the R input file will generate a grid that looks just like the M input file):

```
M
Size: 5
Start: 1 2
    9    0    9    3    3
    6    9    6    8    3
    9    4    1    9    0
    2    0   -1   -1    9
    8    3    9    7    5
```

```
R
Size: 5
Start: 1 2
Seed: 0
Max_Rubble: 10
TNT: 5
```

## Output:

After running this on the command line: **`./MineEscape -c BINARY -s 10 mine.txt`**

```
Cleared 6 tiles containing 41 rubble and escaped.
First tiles cleared:
6 at [1,2]
1 at [2,2]
TNT at [3,2]
TNT at [3,3]
7 at [4,3]
9 at [4,2]
9 at [2,3]
9 at [3,4]
Last tiles cleared:
9 at [3,4]
9 at [2,3]
9 at [4,2]
7 at [4,3]
TNT at [3,3]
TNT at [3,2]
1 at [2,2]
6 at [1,2]
Easiest tiles cleared:
TNT at [3,2]
TNT at [3,3]
1 at [2,2]
6 at [1,2]
7 at [4,3]
9 at [4,2]
9 at [2,3]
9 at [3,4]
Hardest tiles cleared:
9 at [3,4]
9 at [2,3]
9 at [4,2]
7 at [4,3]
6 at [1,2]
1 at [2,2]
TNT at [3,3]
TNT at [3,2]
```

# 9. eecs281_priority_queue

For this project, you are required to implement and use your own priority queue containers. You will implement a **"binary heap priority queue"**, a **"poor man's priority queue"**, a **"sorted array priority queue"**, and a **"pairing priority queue"** that compile with the interface given in **eecs281_priority_queue.h.**

To implement these priority queue variants, you will need to fill in separate header files, **heap_priority_queue.h**, **poorman_priority_queue.h**, **sorted_priority_queue.h**, and **pairing_priority_queue.h**, containing all the definitions for the functions declared in **eecs281_priority_queue.h**. We have provided these files with empty function definitions for you to fill in.

These files specify more information about each priority queue type, including runtime requirements and a general description of the container. Functionally, they are part of the project spec, so make sure you read it carefully.

You are **not** allowed to modify `eecs281_priority_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.

## eecs281_priority_queue interface

Functions:

| | |
|---|---|
| `push(TYPE val)` | `inserts a new element into the priority queue` |
| `top()` | `returns the highest priority element in the priority_queue` |
| `pop()` | `removes the highest priority element from the priority queue` |
| `size()` | `returns the size of the priority queue` |
| `empty()` | `returns whether or not this priority queue is empty` |
| `fix()` | `"fixes" the priority queue if the elements contained have changed priority so that top() and pop() work properly.` **`Note that TNT may result in tiles changing priority.`** |

## Poor Man's Priority Queue

The *poor man's priority queue* implements the priority queue interface using an **unordered** vector. Complexities and more details can be found in poorman_priority_queue.h

### Sorted Priority Queue

The *sorted priority queue* implements the priority queue interface by maintaining a **sorted** vector. Complexities and details are in sorted_priority_queue.h

### Heap Priority Queue

Heaps will be covered in lecture. We also suggest reviewing Chapter 6 of the CLRS book. Complexities and details are in heap_priority_queue.h

### Pairing Priority Queue

We have provided two papers about pairing heaps in the CTools resources in the project folder. Further details can be found in pairing_priority_queue.h

**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 PQ headers.

# 10. Libraries and Restrictions

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

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 `std::partition`, `std::partition_copy`, `std::partial_sort`, `std::stable_partition`, `std::make_heap`, `std::push_heap`, `std::pop_heap`, `std::sort_heap`, or `std::qsort`.
You are **not** allowed to use the C++11 regular expressions library (it is not fully implemented on gcc 4.7) 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.

Also near the beginning of your development, create a simple `main()` in a separate .cpp file (which is not submitted for grading) to test one or more of your priority queues, outside the setting of the mine. For instance, if you push 20 values into one of your priority queues and they come out in the wrong order, this will be much easier to debug than if you had to track down the same bug in a large mine. Also try a mixture of push/pop operations in different orders.

Each test case should consist of a `MINEFILE` 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>.txt` where 0 < n <= 10. The autograder's buggy solutions will run your test cases in the specified `CONTAINER` (for example, `test-1-BINARY.txt` will run your test on the *heap* priority queue, whereas a file named `test-2-PAIRING.txt` will run with a *pairing* priority queue).

Your test cases must be in map input mode and cannot have a `Size` larger than 15. You may submit up to 10 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 having a `Size` of 15; your solution should not impose any size limits (as long as sufficient system memory is available).

**Testing pairing_priority_queue.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 mining simulation). 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 testing 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.

# 11. 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:

1. You have deleted all .o files and your executable(s). Typing '`make clean`' should accomplish this.
2. Your makefile is called Makefile. Typing '`make -R -r`' builds your code without errors and generates an executable file called `MineEscape`. (Note that the command-line options -R and -r disable automatic build rules, which will not work on the autograder).
3. 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 the debug flag, -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!

4. Your test case files are named `test-n-CONTAINER.txt` and no other project file names begin with test. Up to 10 test cases may be submitted.
5. The total size of your program and test cases does not exceed 2MB.
6. 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).
7. 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 may observe incompatibilities).
8. Note: In order to compile with g++ version 4.7.0 on CAEN and the autograder 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
```

9. Turn in all of the following files:
    a. All your *.h and *.cpp files for the project.
    b. Your Makefile.
    c. 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 -U *; tar czvf ./submit.tar.gz *.cpp *.h Makefile test-*.txt`
OR use the sample makefile and make target "submit".

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. **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 between them. You may submit up to four times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time).

**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).**

# 12. Grading

60 points -- Your algorithm for escaping the mine will be evaluated for both correctness and performance (runtime). This will be determined by the autograder.

10 points -- Test case coverage (effectiveness at exposing buggy solutions).

20 points -- Specific tests on the performance and correctness of pairing heap implementation.

5 points -- Proper sorted priority queue implementation.

5 points -- Proper binary heap priority queue implementation.

**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 -Wvla -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: Printing out the Grid

While this is never required to do well in the project, you may find it helpful to be able to print out the state of the grid in a human readable format. Unfortunately, the values of rubble may have a different number of digits, which can make it hard to read. For example, consider the following grid:

```
10 5 100
200 400 200
1 1 -1
```

It is fairly hard to see which tiles touch which other tiles because they do not line up well. Luckily, the header <iomanip> contains the definition for the std::setw stream manipulator.

The following code snippet gives a quick example of how to use it:

```
#include <iomanip>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> ints = {0, 24, 100, 5};
  vector<int> ints2 = {100, 2, 40, 2};
  for(auto i : ints)
    cout << setw(4) << i;
  cout << endl;
  for(auto i : ints2)
    cout << setw(4) << i;
  cout << endl;
}
```

**Output:**

```
   0  24 100   5
 100   2  40   2
```

# Appendix B: Polymorphism

All of the priority queues (poormans, sorted, heap, and pairing) all inherit from the abstract class eecs281_priority_queue. This means that a pointer to eecs281_priority_queue can correctly refer to any of its derived types, and can call the correct underlying member functions. Example:

```
eecs281_priority_queue<int>* priority_queue;
poorman_priority_queue<int> poorman;
sorted_priority_queue<int> sorted;
heap_priority_queue<int> heap;
pairing_priority_queue<int> pairing;

//if using poorman
priority_queue = &poorman;
//if using sorted
priority_queue = &sorted;
//if using heap
priority_queue = &heap;
//if using pairing
priority_queue = &pairing;
```

```
//do stuff with priority_queue
priority_queue->push(5);
priority_queue->push(10);
cout << priority_queue->top() << endl; // prints out 10
```

# Appendix C: Using a priority queue of pointers

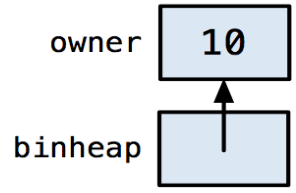
It is highly recommended that you use STL's vector<> or list<> for dynamic memory management in this project. In order to avoid duplicating information, other containers that want to access the data can hold pointers to members inside the "owner" container. For example:

```
list<int> owner;                          // the objects are ints
heap_priority_queue<int*> binheap;        // holds pointers to ints

int x = 10;
owner.push_back(x);

binheap.push(&owner.back());
// *NOT* binheap.push(&x);
```

This creates the following relationship:




The above example is slightly incomplete in that what will be prioritized is the pointers themselves. To output the values in numerical order, a functor is needed (see next example below).

**Warning about using vector<> as an owner container**
Recall from lecture that vector runs out of space, it "moves" all of it's data members into a new, larger space. This will invalidate any existing pointers at the time! If you do decide to use a vector, make sure that the vector never resizes while pointers to it's members are in use. You can do so by using `vector::reserve()` or `vector::resize()`

Because STL containers automatically deallocate the memory they use, you should **never** call **delete** on a pointer to an object that exists in a vector<> or list<>. If you do, you will get a "double free" runtime error. This also means that referencing pointers to elements that have been removed from an owner container will cause a segmentation fault.

The example below assumes you have implemented the poorman's heap correctly. If that is not the case, or for testing purposes, you can substitute any one of the other priority queues.

**Example:**

```
#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()  {
  poorman_priority_queue<int *, intptr_comp> pmheap;
  vector<int> owner = {10, 5, 20, 7};

  for(auto &i : owner) // reference needed for next line
    pmheap.push(&i);   // so that this is the address of the value in owner

  // Process each number in priority order
  while(!pmheap.empty()){

    // Pop one int pointer off the Poor Man's Heap
    // the integer itself is still “alive” in the vector owner

    cout << *pmheap.top() << ' ';
    pmheap.pop();
  }

  cout << endl;

  // pmheap is empty, but owner still has all 4 integers.
  // program should print 20 10 7 5

  return 0;
}
```