EECS 281 – Fall 2013
Programming Assignment 2 (version 1.4)
Mine Mine Mine[1]

Due Sunday, October 27th 11:55pm

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. Will you be able to find your way out before you run out of dynamite?

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

Breaking Out of the Mine

The mine you are trapped in is represented by a 2-dimensional grid, where each spot in the grid either has rubble or is clear of any rubble. You start at a spot in the grid which is clear of rubble. At every turn, you will attempt to blast away the pile of rubble which requires the fewest sticks of dynamite to destroy. If multiple tiles require the same amount of dynamite to destroy, break ties using the following rules:

1. Destroy the tile with the lower x coordinate first.
2. If the tiles have the same x coordinate, destroy the tile with the lower y coordinate (closer to the bottom of the grid) first.

You can *reach* a rubble tile if and only if there exists a path from the starting position to the rubble tile which doesn't pass through rubble tiles. As in project 1, diagonal movement is not permitted. You keep blasting the weakest tile of rubble you can reach until one or both of the following

[1] Credits: David Paoletti, Don Winsor, Ian Lai, Qi Yang, Rajeev Vadhavkar

occurs:

1. You do not have enough dynamite to destroy any reachable piles of rubble.
2. You have blasted a path so that you can reach the edge of the grid, in which case you have successfully escaped!

Note that the movement of the miner is actually not modelled, you can assume that at each turn the miner will move to the reachable tile to be blasted and destroys it. Similarly, the miner will escape as soon as a clear path from starting position to the edge of the map is available.

Example

In the following example, you start at position `[2,2]` (marked by `'SL'`) with `100` sticks of dynamite. The tiles that the miner can reach are shown in **bold**. Each numbered tile is a pile of rubble. The number is the number of sticks of dynamite required to destroy the pile of rubble. Each tile marked with `'..'` is clear of rubble.

Please note: This example mine is for illustrative purposes only and **does not conform to the input file format described below**. Specifically, the italicized numbers refer to row and column number, and are not a part of the actual grid. Also, the `'..'` markings will be numeric zeros in the actual data.

```
      Dynamite: 100

4   39  30  39  28  17
3   32  12  28  29  30
2   10  29  SL  12  73
1   70  ..  13  ..  42
0   07  31  28  48  32

    0   1   2   3   4
```

At the first round, the miner will choose to break tile `[3,2]` because it requires the fewest sticks of dynamite (12) to destroy. Removing that tile has allowed us to reach more tiles!

```
       Dynamite: 88

4   39  30  39  28  17
3   32  12  28  29  30
2   10  29  SL  ..  73
1   70  ..  13  ..  42
0   07  31  28  48  32

    0   1   2   3   4
```

The miner will then choose to break tile `[2,1]` because out of the tiles we can reach, it requires the fewest sticks of dynamite (13) to destroy.

```
       Dynamite: 75

4   39  30  39  28  17
3   32  12  28  29  30
2   10  29  SL  ..  73
1   70  ..  ..  ..  42
0   07  31  28  48  32

    0   1   2   3   4
```

At this point, we have a problem: We can reach two different tiles which both require 28 sticks to destroy! However, following the earlier rules for tie breaking, our miner will choose to blow up `[2,0]` over `[2,3]`.

```
     Dynamite: 47

4   39  30  39  28  17
3   32  12  28  29  30
2   10  29  SL  ..  60
1   70  ..  ..  ..  29
0   07  31  ..  35  32

    0   1   2   3   4
```

After blowing up `[2,0]`, we have cleared a path to the edge of the grid and we have escaped!

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 (PR). Map input is for your personal debugging, but pseudorandom allows us to better analyze your runtime.

Map input mode (M)

Input will be formatted as follows:

- '`M`' - A single character indicating that this file is in map format.
- '`Dynamite`' - Positive integer that specifies how much dynamite you start with
- '`Grid_Size`' - Positive integer number that specifies the size of the square grid (20 means a 20 X 20 grid)

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, which is signified by an integer between 0 and 99 (0 means the tile is clear). Note that unlike the example at the beginning of this project description, empty locations will be designated with a numeric `0`, not with '`..`' characters.
- The miner's starting location, which is signified by 'SL'
- Tiles are indexed as follows:
    - The tile in the bottom left corner is at location [0,0]
    - The tile in the top right corner is at location [Grid_Size-1,Grid_Size-1]

Example of M input (SL is at location [1,2]):

```
M
Dynamite: 100
Grid_Size: 5
10  3 28  9 19
 0 74 32 29 20
38 SL  0 48 19
49 90 70 27  8
19 48 23 59 57
```

Pseudorandom Mode (PR)

Input will be formatted as follows:

- '`PR`' - Two characters indicating that this file is in pseudorandom format.
- '`Dynamite`' - Positive integer that specifies how much dynamite you start with
- '`Grid_Size`' - Positive integer number that specifies the size of the square grid
- '`Random_Seed`' - Number used to seed the random number generator
- '`Max_Rubble`' - Positive integer that specifies the maximum strength (amount of dynamite needed to clear) of a single tile of rubble.
- '`Start_x`' - x coordinate of starting location. You may assume this will always be within `0` and `Grid_Size - 1`, inclusive
- '`Start_y`' - y coordinate of starting location. You may assume this will always be within `0` and `Grid_size - 1`, inclusive

Example of PR input:

```
PR
Dynamite: 100
Grid_Size: 7
Random_Seed: 2934
Max_Rubble: 35
Start_x: 3
Start_y: 3
```

Generating your grid in PR mode:
The procedure for generating your input in pseudorandom mode is as follows:

1. Include `<random>` in your header
2. Seed the the random number generator with `std::mt19937 gen(Random_seed);`
3. Initialize a uniform integer distribution with the desired range:
   `std::uniform_int_distribution<> dis(0, Max_Rubble); //init first`
4. For each tile in the grid, starting from `[0,0]`, generate the rubble value by calling `dis(gen)`
   You should fill up row 0 before moving on to row 1. So, in order it should be:
   `[0,0] [1,0]` … `[Grid_Size-1,0] [0,1] [1,1]` ...
5. Set the rubble at the miner's starting location to 0.

For more details see http://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution. Do *not* use `rand()`.
Please make sure that you generate the grid exactly as described, or else the grid you generate will differ from our sample solutions.[2]

Command Line (Input):
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).**

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

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.

- `-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.

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 --verbose 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 match the default behavior of getopt.

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 we only test with properly formed input.

Output

The output of `MineEscape` should be as follows:

The output will always begin with a single line that summarizes the success or failure of the escape attempt.

If you are successful in your escape, print out the coordinates for the last destroyed tile (if you

did not have to destroy any tiles, print the starting location tile):

`You escaped through tile <X_COORD>,<Y_COORD>! You have <DYNAMITE_LEFT> sticks of dynamite left.`

If you were unable to destroy any tiles, print the following and **skip verbose output**:

`You couldn't escape!`

If were able to blow up a non-zero number of tiles but still could not escape, print the following:

`You couldn't escape! You used your last dynamite sticks on tile <X_COORD>, <Y_COORD>.`

Examples:

- `You escaped through tile 3,0! You have 10 sticks of dynamite left.`
- `You couldn't escape! You used your last dynamite sticks on tile 4,2.`

If a `--verbose` or `-v` option is not specified, or if you are unable to destroy any tiles, this one line is the only output of the program.

Otherwise, if the `'--verbose N'` option is specified on the command line and at least one tile was destroyed, the following additional output should be printed in the following order:

A. The line `'First tiles blown up:'` followed by:
The coordinates of the first N tiles that you removed or detonated. Please keep in mind that you do not add cleared tiles to this list, **only tiles that you blew up**.
<NUM> (<X>,<Y>): <WEIGHT>
Where **<NUM>** corresponds to the order in which the tile was removed, listing earliest ones first.

B. The line 'Last tiles blown up:' followed by:
The last N tiles that you detonated, in the same format as part **[A]**, listing the latest ones first. **<NUM>** will again correspond to the order in which the tile was removed, starting with the total number of tiles removed and counting down.

C. The line 'Hardest tiles to blow up:' followed by:
The N hardest tiles you blew up in the same format as part **[A]**.
<NUM> just counts up starting from 1 for this section.

D. The line 'Easiest tiles to blow up:' followed by:
The N easiest tiles you blew up in the same format part **[A]**.
<NUM> just counts up starting from 1 for this section.

The last two statistics **[C]** and **[D]** should be calculated as fast as possible, regardless of the

specified container type. You may wish to use additional heaps for this purpose. These additional heaps may be of any type you wish. In other words, if your program was invoked with `--container POOR_MAN`, you must use the `POOR_MAN` heap for planning which tiles you will blast to escape from the mine, but you could use faster heaps such as `BINARY` or `PAIRING` for recordkeeping for your statistics.

If for any of the above statistics you do not have N tiles to print, you should print the data for as many as you can. For example, if the program is run with `'--verbose 10'`, and you only had to interact with 3 tiles, you might print these additional lines:

```
First tiles blown up:
1 (2,3): 19
2 (3,3): 5
3 (4,3): 19
Last tiles blown up:
3 (4,3): 19
2 (3,3): 5
1 (2,3): 19
Hardest tiles to blow up:
1 (4,3): 19
2 (2,3): 19
3 (3,3): 5
Easiest tiles to blow up:
1 (3,3): 5
2 (2,3): 19
3 (4,3): 19
```

The numbering of the verbose mode output lines is important. For the first, hardest, and easiest tiles, just number them consecutively starting with 1. For the last tiles, the numbers count down, starting from the total number of tiles blown up. For example, if you blew up a total of 800 tiles, and you were running with --verbose 4, the “Last tiles” lines would be numbered 800, 799, 798, 797.

Implementations of Priority Queues

We have provided a header file, `eecs281heap.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 heap*’, a ‘*sorted array heap*’, and a *pairing heap* that compile with the interface given in `eecs281heap.h`. To implement these priority queue variants, you will need to fill in separate `header files, binary_heap.h, poor_heap.h, sorted_heap.h,` and `pairing_heap.h`, containing all the definitions for the functions declared in `eecs281heap.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 `eecs281heap.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 heap* implements the priority queue interface using an unordered array-based container. The expected complexities for this container can be found in it's specific header file.

The sorted heap 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 F13 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 heap headers.

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::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.

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` (ie: `test-1-BINARY` will run your test on the *binary* heap).

Your test cases **must be in map input mode** and cannot have a `Grid_Size` larger than 30. 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 `Grid_Size` of 30; your solution should not impose any size limits (as long as sufficient system memory is available). Be sure to designate empty locations with numeric `0` characters and not with `'..'` as shown in the examples (which are not valid input or test cases).

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 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 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.

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`' shall 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 did observe incompatibilities).
8. 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
```

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 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 four times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). 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. Although current EECS 281 course policy is to give you credit for your best submission, we will assume that your last submission is your best unless we detect a substantial decrease in your score on the last day of the project (so if something goes wrong earlier, we expect you to correct the problem and resubmit).

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

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

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

10 points -- Specific tests on the performance and correctness of your pairing heap 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: Using a heap of pointers

You might recall from EECS 280 that one should consider certain rules (at-most-once, existence, ownership and conservation) when designing/using containers that hold pointers-to-objects. We will not discuss these rules in detail here, but you should probably go back and review your EECS 280 notes on this subject to refresh your mind.

These rules were introduced in EECS 280 to make it easier to understand the basics about containers. However, in this project, the use of STL means that some of these rules will either be maintained for you (via STL) or will be ignored due to project specs. The purpose of this note is to reconcile what you learned in EECS 280 with the objectives of this project, and show simple examples of how we expect you to use the priority queue interface.

Rule 1 (at-most-once): *any particular object lives in *at most one* container at a time*.

There is (almost) never a good reason to maintain duplicate information so this is a rule you need to stick to. You're probably aware by now that you will be maintaining at least two separate containers in this project. Both containers will have the same data (or some portion of it), which means that you must decide which of the containers will be the "**owner**" of the objects (i.e. the container will hold the objects themselves). The other container(s) will hold pointers to the objects already inserted into the owner container. For example, you may declare two such containers as:

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

Rule 2 (existence): *allocate an object before inserting it into any container*.

STL will take care of allocating the dynamic memory for you. Thus, the use of the **new** keyword to create objects that will be inserted into the owner container *is not necessary and we highly discourage it for this project.*

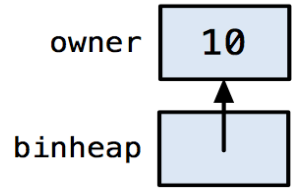
```
int x = 10;
owner.push_back(x);
```

This will insert 10 into "owner" where it will live and you can refer to it via a pointer until it is either popped out of the `list` or the `list` ceases to exist. If the owner container was a vector, you would not be able to refer to it inside the container with a pointer because when a vector resizes it moves its elements in memory and thus some pointers might "lose" the object they were pointing to.

To insert a pointer to the same object into the "**non-owner**" container(s), you can do

```
binheap.push(&owner.back());
```

This creates the following relationship:



Observe that you will be the one implementing the `push()` method in each of the priority queue implementations (`binary_heap, sorted_heap, pairing_heap, poorman_heap`), so it is up to you to make sure that it works as the above example shows while also making sure that heap invariants are maintained.

Rule 3 (ownership): *once an object is inserted, that object becomes the property of the container. No one else may use or modify it in any way*.

This project violates this rule/uses an exception to the rule since non-owner containers will need write access to the objects in the owner container to update priorities between rounds. This is why, in the example for Rule 1, the binheap container was declared to hold non-const pointers.

```
int* min = binheap.top();
*min = 5;                  // sets owner.back() to 5
```

Rule 4 (conservation): once an object is removed from a container, it must either be deallocated or inserted into some container.

This is another rule that STL will manage for you since the objects themselves will live in an STL container (as the above examples show). Thus, the objects will be deallocated when the object is popped out of the owner STL container or when the container ceases to exist (e.g., the scope in which the container was declared terminates). In other words, you should not use the **`delete`** keyword when you `pop()` a pointer out of a non-owner container.

Example: using a heap of pointers. This assumes that you have implemented the poorman's heap correctly.

```
#include <iostream>
#include <list>
#include "poorman_heap.h"

using namespace std;

// Comparison functor for integer pointers
struct intptr_comp {
   bool operator() (const int *a, const int  *b) const {
       // *a < *b means that *a has a lower priority than *b
```

```
        return *a < *b;
    }
};

int main() {
    poorman_heap<int *, intptr_comp> pmheap;
    list<int> owner; //Pointers to the elements inside a list are
    //not invalided when you modify the list.
    owner.push_back(20);
    pmheap.push(&owner.back());

    owner.push_back(5);
    pmheap.push(&owner.back());

    owner.push_back(10);
    pmheap.push(&owner.back());

    owner.back() = 30;

    while(!pmheap.empty())    {          // prints “30 20 5”
        int *z = pmheap.top();           // note: if you comment out
        cout << *z << ' ';                // the line “owner.back() = 30;”
        pmheap.pop();                     // it prints “20 10 5”
    }
    return 0;
}
```