Computer Science 432
Spring 2022
Lab 1: A Compute Intensive Program, due Monday before lab.

Objective. To develop idioms for managing dynamic data collections.

Discussion. An important resource that an operating system is responsible for managing is *memory*. One obvious way that this precious resource is handed out is through the management of the *heap*. On Unix machines, the heap is accessed through the C library routines `malloc(3)`, `realloc(3)`, and `free(3)`. These routines, along with others, allow us to access, resize, and return dedicated chunks carved from the heap.

While in object-oriented languages (Java, Swift, *et al.*) integrate heap management into object allocation and garbage collection, the C language requires that programmers manage memory on their own. When you want to create a new object, you will typically need to call `malloc(3)`. When you're done with an object, you need explicitly return the memory to the store, with `free(3)`. The `realloc(3)` allows you to extend (or reallocate) a previously allocated memory chunk.

It is important to realize that the heap is not managed by the operating system, but occasionally the operating system is called on to expand the amount of memory dedicated to a heap when it is practically exhausted. We'll not (immediately) be concerned about this, but when we're writing different parts of an O/S kernel, we'll have to perform similar memory management operations.

Suppose we are interested in maintaining an extensible array, `pts`, of (hypothetical) directly containing *point* types:

```
typedef   struct point   point;  // we will declare "struct point" below
struct point {                   // structure definition only
  int x,y;
};
```

We typically initialize `pts` with a small `allocation` and keep track of the actual `length`:

```
int allocation = 6; // physical extent of the vector, pts
point *pts = (point*)malloc(allocation * sizeof(point));
int length = 0;     // logical extent of the vector, pts
```

Then, provided there is sufficient room, adding a new point to this vector involves a statement like:

```
pts[length].x = ...; // some x value
pts[length].y = ...; // some y value
length++;            // we now have one more in the array
```

Everything is fine until, of course, the length of the vector first exceeds the underlying allocation. When this happens, we typically want to extend the array with code that looks something like this:

```
// do we have room?
if (length >= allocation) {
   allocation = ...; // compute how much more you need (cs136)
   /*
    * allocate a new, larger chunk, and copy the existing data over:
    *  - realloc returns a "void *" (a pointer to anything); cast it
    *  - if realloc gives you a new chunk, it frees the old
    *  - it returns 0 (NULL) if it fails; ideally check this (I didn't)
    */
   pts = (point*)realloc(pts, allocation * sizeof(point));
}
// assertion: pts[length] is valid memory
```

Everywhere where we mention the "`point`" type above, it could be any other type. If you use `int`, it will be an array of integers. If you use `int*`, it will be an array of pointers to integers, etc.

Assignment. We'll develop another small C program:

> Write a method, `mex`, that returns the minimum excluded non-negative integer in a list. That is, if `i = mex(n,v)`, the value `i` is the first non-negative integer not found in a list `v` of `n` integers. Make no unreasonable assumptions about the size of `n`.
>
> As part of testing for correctness place this routine in a program, `mex`, that reads in all integers available from the input and prints the first non-negative integer that was not mentioned. Again, make no unreasonable assumptions; try to be space and time efficient. See `malloc(3)`, `free(3)`, `realloc(3)`, but *not* `qsort(3)`.
>
> Hints: The file `~cs432/mex.dat` contains about 100 million non-negative integers.
>
> **Do not copy this to your directory.**
>
> How long does it take your driver program to report the smallest missing non-negative integer? You can use the `time(1)` command to measure how much elapsed time is required to run any command.

1. You can clone this lab with

   ```
   $ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab1.git
   ```

   where 22xyz is replaced with your CS username.

2. Spend some time *thinking* about how this code is organized. Clearly, there are two main things to think about: (1) how to read in an arbitrary number, n, of integers, and (2) how you organize the internals of `mex(n,v)`. If you are hasty and make bad decisions now, we'll see you back here, rethinking your design, later.

3. Always keep in mind that an array is simply a pointer to its first element. You'll want to use arrays, everywhere, but they'll be dynamic so they'll be declared as pointers to dynamically allocated memory, treated as an array. See me if you do not understand any aspect of this.

4. Add an entry into the `Makefile` that will compile your code. Always compile with the `-Wall` switch, which is responsible for printing all warnings. If your code compiles with a warning, you **must** address it.

5. Thoroughly test your program. How will you do this? What are trivial tests? What are assumption-breaking tests with no more than, say, 3 inputs? Does `mex` need to consider *every* value of `v`?

6. You may find it useful to keep track of how long it takes (see `time(1)`) to run your program on `mex.dat`, and the answer printed. Compare this time with others in class. Are you fast? slow? why?

7. When you think you are finished, add, commit, and push your code for review and grading.

Thought questions.

1. How do you increase your allocation? Why? Are there other choices?

2. Given your allocation strategy, how much memory do you expect to use in processing `mex.dat`? What is the maximum overhead (extra memory) that you may have allocated? If it is high, is there a good reason?

3. Does your program process input data at a constant speed?

4. Suppose you stored your data in a linked list of your own design. Would there be any advantage? Would it be faster? Slower? Use more space?

5. What resources are consumed by your program?

6. I said: No sorting. Is that true?

Why we care. Nim is a game played with a pile of n stones. You and an opponent are allowed to alternately remove up to m stones from the pile. Let's assume: The winner removes the last stone. Obviously, for piles $n \leq m$, it is a winning move to take all n stones. For $n > m$, the winning move is the `mex` of the winning moves for piles of size $n-1, n-2, ..., n-m$. If that value is zero, you are in a losing position. *The analysis of all impartial games between two players can be mapped to the analysis of games of Nim.*

⋆