HW1b DArray and Hashtable Feedback

Feedback written by Julie Zelenski.

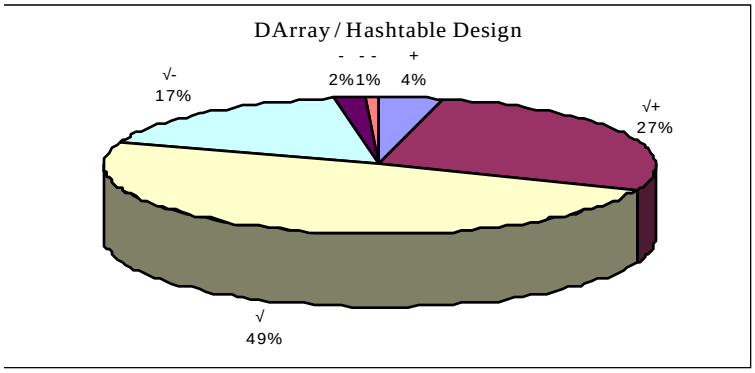
Actual comment from Miler posted yesterday:

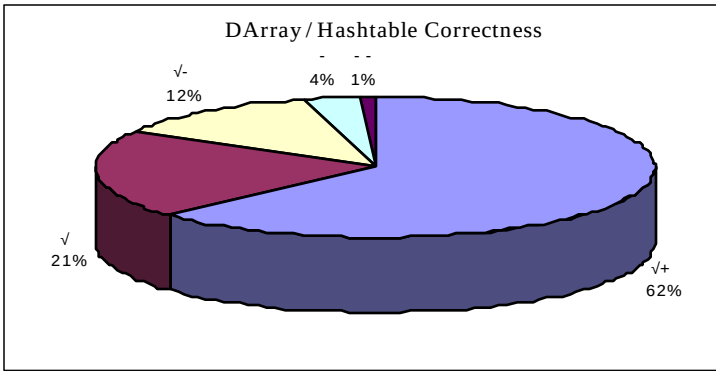
> I'm pretty impressed -- I didn't think I'd be giving out that many check-pluses.

No, you shouldn't assume that we think you're all programming meons until you prove otherwise. His comment came in response to having been through the CS107 program before (all of the TAs have, actually), and knowing how miserable the DArray experience can be if students try to fake it like they did in CS106B. You should all be very happy with yourselves here: I haven't had a chance to look at too many programs myself, but I did look at all the `DArrays` and `Hashtables` for one particular grader (no, not telling) and saw a lot of really great things. Code was pretty, functions were lean and tight and bullet-proof, implementations looked professional. You all come in CS107 knowing very well how hostile this assignment can be, and you've done more than survived it. All but a few pretty much nailed everything about it. Rage.

As mentioned in the `Scanner` feedback sheet, we decided to split the single grade into two grades: one for correctness, and one for clarity and design. For this and all future assignments, you'll be getting two grades, and each of them will be weighted equally. We've even introduced a + grade in the design/clarity category for those submissions which were publishable as answer keys. We **might** consider introducing a + grade for correctness as well, but we haven't yet, since we have yet to define what really very correct means.

Here are the distributions for the two categroies across four of the five TAs. (The fifth TA's scores weren't available at press time.) Remember that √+'s map to 95's, √'s to 88's.





On memory and void *

- It is not legal to perform pointer arithmetic using pointers of type `void*`. When you add a number to a pointer, the compiler uses the size of the type being pointed to in order to figure out what to multiply the offset by. The `void` type has no size, so it makes no sense to add to a `void*` pointer. If you wish to do byte-wise arithmetic, you need to cast your pointers to `char*`, since the size of a `char` is defined to be one and then explicitly multiply the offset by the element size. Some compilers will allow math on `void*`s and assume you meant to treat the `void*` like a `char*`. This is not a portable coding practice and you should not write code which depends on it.

- `memcpy` is more usually more efficient than `memmove` and thus should be used when the source and destination do not overlap.

- It is very important that you pay attention to the result from a `realloc` call. `realloc` cannot always just stretch the size of the block you are already using, it may have to find a block elsewhere and copy your contents over and return the base address of the new block. If you ignore the return value and continue using the address of the previous block, you are accessing memory that no longer belongs to you (it's been freed) and isn't even the right size. The return value is also important for error-checking purposes, you should assert that the value returned was not `NULL`, as would happen when `realloc` couldn't satisfy your request.

- As encouraged in the assignment handout, you should be making liberal use of the `assert` macro to fortify your code against exceptional conditions and incorrect uses by the client. For starters, assert that all allocations and reallocations succeeded. Although in a virtual memory system, pretty much any allocation request can be satisfied, a `NULL` result can be returned when the internal `malloc` data structures have been damaged, which is certainly something you would like to notice earlier rather than later. You also should protect against out-of-range or unacceptable values for array positions, element sizes, number of buckets, etc.

On designing ADTs

- One of the issues in implementing an ADT is whether to call other public functions while implementing a public function. Should `ArrayAppend` call `ArrayInsertAt` or repeat its code? Should all the `DArray` functions call `ArrayNth` to obtain the address of any element rather than inlining the nasty pointer math? Going through the public API may involve some efficiency loss, but is typically easier to write, easier to debug, results in more readable functions, avoids duplication of code, and makes the ADT more malleable in the future. Those are pretty compelling benefits, and in most cases, I encourage using the public functions. However, one situation to avoid is where the reuse introduces a significant performance cost. For example, a direct coding of `ArrayReplaceAt` is a constant-time operation whereas calling `ArrayDeleteAt` and `ArrayInsertAt` will slow it down to linear-time. You also should watch for the case

where the public functions introduce a lot of redundancy (e.g. a sorted insert that searches once for duplicate by using a public lookup and then again to find the position to insert). Use your judgment to decide if the projected performance gain overshadows the implementation hassle and the lessened future flexibility.

- In this assignment, you had to switch hats between `DArray` implementor (when writing the `DArray`) and `DArray` client (when writing the `HashTable`) and then finally a client of all three ADTs. It was important to keep your two roles straight. The `HashTable` should not make any assumptions about behavior of the `DArray` that is not specified in its interface file and or rely on any details of the `DArray` implementation. Most of you know better than to do any obvious breaking of the wall, but there will still some subtle dependencies present. For example, why did so many of you pass 0 as the parameter for `numElemsToAllocate` when calling `ArrayNew` from your `HashTable` implementation? As a protection against client misuse, your DArray handled this and substituted a reasonable default, but why rely on it bailing you out? You know the bucket arrays will be small and you want to keep allocations tight, so you should be supplying the allocation value you want, not letting the `DArray` pick on your behalf. Another example would be asserting that the element size was positive in the `HashTable`. Since you know the `DArray` is supposed to `assert` on that, many of you didn't bother to have the `HashTable` check on its own. It's a better idea to make the `HashTable` take responsibility for its own part in defensive programming.

DArray

- There was some particularly odd bits of math and logic centered around how some programs decided that a `DArray` needed to allocate and how much to allocate. Often, programs would name the `DArray` fields things like `allocNum`, `allocLength`, `logicalLength`, etc., but it is essential to keep in mind what units these are expressed in: number of bytes or number of elements? We saw several programs with nasty memory crashers directly resulting from misinterpretation of the units. In times like these, a good naming convention can be a lifesaver— how about names like `numElemsUsed` or `numBytesAllocated` that makes it really clear what units you are taking about? Also, avoid that weird div/mod stuff when it isn't necessary. It is much clearer to test for an obvious condition like the number of elements used being equal to the number of elements allocated than to check if the number of elements used is a multiple of `numElemsToAllocate` or some such silliness.

- There was clearly a lot of commonality to exploit between the functions. `ArrayAppend` and `ArrayInsertAt`. In fact, one of them is just a special case of the other. Note that it is `Append` that is a special case of `InsertAt`, the case of inserting into the last slot, not the other way around. `Append` just calls `InsertAt` to put the new element in the last position.

- memcpy/memmove are used to copy a chunk of any size from one location to another. When you need to shuffle the elements in an array over, do not use a loop to memcpy each element one by one, you move the entire chunk of data with just one call to memmove.

- It is important to realize the "sortedness" of the DArray was completely determined by the client's use. If the client has been adding the elements in order or has recently sorted the array, they can indicate so when calling ArraySearch and that will be your clue to use the faster binary search. As the implementor, you cannot use binary search if the array is not sorted and you are not at liberty to sort the array for your own convenience before searching.

HashTable

- Although the assignment handout was very explicit about this, a few programs didn't use the DArray in their implementation of the HashTable. Not only did this show you didn't read the handout carefully, it made the HashTable harder to write, since you could have leveraged a lot of the DArray functionality to do the job. TableEnter is mostly just an ArrayAppend, TableLookup is passed off to ArraySearch, TableMap is assisted by ArrayMap. Let's hear it for modular code re-use!

- The point of a hashtable is to avoid the costs of linear lookup by dividing things into small buckets. Searching all the buckets for an element rather than just the one bucket it hashes to really defeats all the benefits of hashing. If you don't quite understand the data structure or algorithm we're looking for, you should definitely ask one of the course staff to give you a hand.

- In the HashTable struct, you have a field which contains the address of an array of buckets, where each bucket is a DArray. The proper type for this field was DArray*, not the more general type void*. As a rule, you should type things as precisely as you can. Only use a void* if you have to accommodate different pointer types and cannot commit to a specific base-type at compile-time.

- You need to be sure to allocate a DArray (using ArrayNew) for each bucket in the table. You could either do that pre-emptively in TableNew, or defer creating each bucket until needed during a TableEnter call. Either way, when you call ArrayNew, you specify the allocation number (the amount of elements to grow by) for the DArray. Since you expect that the buckets will only contain a few elements, it is wise to choose a small allocation number (say 4) than a potentially wasteful larger one (or even sloppier, let the DArray pick for you).

- If you have a choice between using a constant-time function (ArrayAppend) and a linear-time function (ArrayInsertAt) and you don't care about the differences between the functions (in this case you don't care about the order of your array

elements within a bucket), go ahead and use the faster one.

- Some of you choose to sort the buckets `DArrays` in order to use the faster bsearch lookup. However, it's important to keep in mind that the whole point of hashing was to keep the number of elements in each bucket small, hopefully just 1 or 2 items. When N is that small, binary search doesn't have any appreciable performance gain over a simple linear search and you're paying the cost of sorting for no benefit.

- One case programs often forget to assert was that the value returned by the client's hash function was in range for the number of buckets in the hash table. Thus a malfunctioning client hash function causes your code to reference into la-la land. Code defensively and report the problem rather than allow the client's mistake to cause subtle damage without warning.

Home page lookup client

- The best data structure is the most straightforward: the lookup table is a `HashTable` of structures, where each `struct` contained a `char *` (the word) & a `DArray` of `char *`'s (usernames that referenced that word).

- It is best to avoid extra levels of indirection when you can. For example, in the hashtable, it is much preferable to store structs themselves as elements, rather than pointers to structs. Using pointers introduces an extra dereference on every access, and requires that you dynamically allocate many small structures, fragmenting the heap, and means you have to be careful to free them all when done.

- In a similar vein, if you only need a variable temporarily (i.e. for the lifetime of a particular function) and its size is known at compile-time, you should allocate space for it on the stack, rather than dynamically allocating it at the beginning of the function and freeing it at the end. Using dynamic allocation is less efficient and more prone to problems (mallocing the wrong size, failing to free, etc.) Also, you should realize that both the `DArray` and the `HashTable` copy the element you pass it, so you do not need to keep a copy around (i.e. it is okay for element to be on the stack and go away).

- Keeping in mind that space for a temporary structure can be allocated on the stack, some of you wrote a helper function that would create a structure on the stack, initialize its values, and return that structure back to the calling function. In C, assignment between structs requires that all structure fields are copied and returning a struct is often an expensive operation since it usually necessitates special-case handling (as opposed to primitive types which are returned in a register). In general, you want to avoid unnecessarily passing and returning structures by value. In this case, the structure declared on the stack could be passed by reference to the initialization function and avoid the expensive and superfluous copying.

- Remember that the goal of hashing is to use a number of buckets roughly equal to the number of items, so that the number of collisions is kept small. Since you're expecting a lot of words to be entered in the table (about 3,000), be generous when choosing a bucket size. Ten buckets, fifty buckets, even a hundred buckets will result in a large number of collisions. Something like a few thousand is much more appropriate.

- Although most of you are onto to using the `Scanner` for both parsing the home pages and for extracting the usernames, this time there was yet another opportunity to use the `Scanner`: for reading from `stdin` when getting a word from the user. The `Scanner` is much nicer to use that mucking with `fgets` and the like. And the best code is code that is already written, debugged, and tested. Be diligent about code reuse!

- To print out the array of matching usernames, you can leverage the `ArrayMap` interface which allows you to iterate over all the members in a DArray using a callback function. The only part that requires a little creativity is figuring out hot to number the elements as you go. One way to accomplish this is passing the address of the counter as the `clientData` parameter and incrementing as you go.

- By far the most common error in the client programs was incorrect use of the `void*` parameter when calling `ArrayAppend` and misinterpretation of the `void*` return from `ArrayNth` or the `void*` passed to the client callback functions. You tell `ArrayNew` the size of the data element you intend to store and pass to `ArrayAppend` a pointer to an element and `ArrayNth` will return a pointer to an element and a callback function receives a pointer to an element.. This is true even if the element type is already a pointer (such as `char *` or `DArray`). Understanding this was key. If you are still unclear on this issue, I recommend that you trace through the operations very carefully until you fully understand how they operate.

- This pointer problem that pretty much everyone encountered is a good illustration of the limitations of C: you have to sacrifice compile-time type-checking to support generic data types, and you never realize how much you appreciated the type-checker until it's gone.

Readability

- Use pointer math only when you have to. When you don't know the base type of the array, you will have to result to pointer math to access elements by index, but for any array which has a known base type (such as the array of buckets in the `HashTable`), it is much more readable to use array subscripting!

- By now, you should know exactly what a typecast does and doesn't do, and you should only use them when absolutely necessary. In particular, don't introduce redundant or nonsensical casts. For example, it is never appropriate to cast an l-value, like this: `(char *)nthElemPtr = (char *)darray->base + offset;`

- You also never need to cast anything back to a `void*`, since it is the "universal recipient" and is compatible with any pointer type.

- Typecasts should not be thrown about indiscriminately. Any time you introduce a typecast you must assume all responsibility for type-checking. You should feel very wary about typecasts, and use them only when you must. In this assignment, pretty much the only casts needed were one in the `DArray` to cast the `void*` pointers when doing pointer arithmetic and in the client who had to cast the `void*` into the correct element type.

- In the `DArray`, you have an oft-repeated sequence that calculates the address of the nth element given the base and the element size. As we repeatedly encouraged, do yourself a favor and make a helper to call everywhere. It's easy to slip up a little and using the helper will greatly improve the readability where it's used. The `DArray` operations have short but tricky sequences (such as the calls to memmove to shuffle things around) that you want it be as clear and readable as possible so you more easily verify that it is correct. You don't want to have stare through that pointer math expression until you're sure it's okay each time.

- The function `ArrayNth` is almost, but not quite, what you need. It computes the address, but the function also includes an assert. In a few places you needed to calculate the position one beyond the last, which would have failed that `assert`. Some people compromised by using `ArrayNth` in most places while resorting to the raw math on those few occasions, but this really isn't ideal. Instead, create an `ArrayNthHelper` that does the math without the `assert`. The helper can freely be used in all situations, and even `ArrayNth` itself calls it, after it checks the assert.

A few random C thoughts

- Our class position on using library functions will be "You can always use anything from the standard ANSI libraries (i.e. if it's in K & R, it's fine by us)." So all the string functions, `qsort`, `bsearch`, etc. are at your disposal. However, other random library stuff is not okay unless we specifically state otherwise. For example, the non-

standard `lfind` and `lsearch` functions are not something you should have used (and were clearly listed as off-limits in the handout, too).

- Private internal functions should be declared to be `static`. Otherwise, they default to be globally visible, which may cause "multiple define" problems for the client at link-time.

- It is not a good idea to put code in a assert, it is best to only include a test expression with no side effects. For example, avoid using assert like this:

```
assert ((s = malloc(100) != NULL);
```

  There is a preprocessor flag that allows all asserts to be turned off (and thus completely removed), often used when a project is moving from development to production. In such a case, blocking asserts would be remove not only the test, but also the call to `malloc`, which would be unfortunate, indeed.