CS251, Spring 2013

Homework 3: Merge Sort for Generic Data

Due: Monday, Feb 11 at the beginning of class.

Revision: 12:30pm Feb 4: removed constraint on modifying sort.h

---

In Lab-2, you learned how to write a comparison function in C which could then be passed as a ***parameter*** to the standard library `qsort` routine. Whenever the `qsort` routine needed to compare two elements in the given array, it would just call the comparison function that you passed to it. So, no matter what kind of data you want to sort and no matter how you want to specify the ordering of that data, all you need to do is write a suitable comparison function to pass to `qsort --` instead of writing a completely new sort routine just for a particular type of data.

In Lab-2, you were the ***client or user*** of the generic library qsort function. In this homework, you will play the role of a library function ***author***. In short, you will implement a version of Merge Sort which is generic in the same way that `qsort` is generic-- in fact, your routine will have the same parameter list as `qsort`.

Let's review the parameter list for `qsort` (and for your `msort`):

> **`void *base`**: the base address of the array being sorted; or equivalently, a pointer to the first element of the array.
>
> **`size_t n`**: the number of elements in the array.
>
> **`size_t size`**: the size of an individual entry in the array where the unit of measure is the size of a `char`. On almost all systems, this is a byte but the C standard only states that a `char` is the basic memory unit (conceivably, a system could use 2 bytes for a `char).`
>
> **`int (*cmp)(const void *, const void *))`**: pointer to the caller-specified comparison function.

Some possible questions:

> **Q:** what is this `size_t` business?
> **A:** it is an unsigned integer type. `size_t` is not a basic type in C (like `int` or `char`). It is defined in the standard header file `stddef.h` (which is included by `stdlib.h`). On any given system it could be any of the `unsigned int` basic types of C.
> Wikipedia page on C basic types.

Q: how do I call/invoke another function passed to me as a parameter (a “function pointer”)?
A: in this case, the parameter is called `cmp`. It is called by “dereferencing” the function pointer; for example if I wanted to call the function and store the returned value in an integer variable `result,` I would do something like this (parameters left blank since we discuss them below):

```
          result = (*cmp)(__________, _________);
```

Q: why do I need to know the size of the individual array elements?
A: the size of an array element depends on the type of data being stored -- it might be just a single `char` or a large `struct` with 10 integer fields (probably 40 bytes). (But we ***can*** count on all elements in an array being the same size.) But the array passed to you is completely generic, specified simply as `void *`. So, your code needs to do some *address arithmetic.* Remember that the base address of the array is also the address of element 0 in the array.
Remember that `size` tells us how big (“wide”) each array element is. So, where would the element at index 1 be in memory? Start from base and skip over the `size` units occupied by element 0:

```
     void *p1;
     // …
     p1 = base + size;
     // now p1 could be passed to (*cmp)
```

Or in general, to get a pointer to the element at index `i`:

```
     void p_i;
     // …
     p_i = base + i*size; // note we get right thing when i=0
```

Q: So I can’t use the handy array indexing operator `[]`?
A: Not as far as I know if you really claim to support arbitrary array element sizes. Here’s the deal: when the compiler generates code for an expression involving the `[]` operator, it produces code which does pointer arithmetic as above. ***But***, the element size must be known at ***compile-time*** (a constant) to generate the correct pointer arithmetic expression. Consider this code to dynamically allocate an array of 100 `doubles` and set one of the values in the array:

```
     double *a = (double *)malloc(100 * sizeof(double));
     a[20] = 3.1415926;
```

On my machine `sizeof(double)` evaluates to 8. The compiler knows this at *compile time* and so can generate code equivalent to correctly compute the address of `a[20]`.

Q: What do you mean "at compile time"? When calling `malloc`, we need to call `sizeof(double)` and doesn't that happen at runtime?
A: Ahh, although `sizeof(double)` *looks* like a function call, it is actually an operator and is evaluated to a constant at *compile-time*.[1]

Q: When I look at the merge sort code for arrays of integers, there is lots of code that moves array elements around using the `[]` operator. How can I do the equivalent when the element size is not known until my function gets called?
A: More fun with pointers. Remember that `size` tells you the number of `char`s an array entry occupies. So suppose you want to copy element i from array `a` into index `j` of array `b`. I'm not going to give you code here, but I'll outline the procedure:

```
1.  Initialize a char pointer (say pa) to point to the
i'th element in a[] as previously discussed.

2.  Do the same with a char pointer (say pb) for the j'th
element of b[].

3.  Then copy the appropriate number (given by size) of
chars starting from pa into the block of chars starting at
pb -- i.e., by looping.

    IMPORTANT:  in case you've never knew, if you have a
    char pointer and increment it  (like pa++), it will
    then point to the next char in memory.  (Similarly,
    if it is a pointer to an int, incrementing it will
    advance to the next int in memory -- probably 4-bytes
    away).

    Optimization:  To speed up this kind of copying, you
    can check if the element size is a multiple of the
    size of another type usually larger than char --
    e.g., long int.  Then you can copy larger chunks at a
    time by treating the pointer as a pointer to the
    larger type.  Not necessary for full credit.
```

[1] There is one situation in the C99 standard where it is evaluated at runtime, but that is for retrieving the size of an entire "Variable Length Array"; but the element type still needs to be known at compile time.

What to Submit

Along with this handout, you have been given a Code::Blocks project containing the sorting library code we used in Lab1 (the profiling lab). It contains an implementation of merge sort on integer arrays and a stub (function with empty body) for the generic msort you are to complete.

There are three source files in the project:

> `sort.h`: this includes the prototypes for the functions exported by the library. **You may not alter any of the function prototypes that are already in the file.** However, you may add other function prototypes (e.g., for testing/sanity checking).
>
> `sort.c`: this is where you will write your code -- i.e., completing the stub for the `msort` function. You may add additional utility functions. **This is the only file you need to submit.** (Even if you added other functions to *your* sort.h file, we won't need to call them to test your submission).
>
> `test.c`: just a toy program that exercised the non-generic sorts. You can modify this file for your test driver (but you don't need to turn it in).

Suggestions

> *"Debugging is an art that needs much further study .... The most effective debugging techniques seem to be those which are designed and built into the program itself -many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other half; the first half... will eventually be thrown away, but the net result is a surprising gain in productivity."*
>
> *- D. Knuth (The Art of Computer Programming, Volume 1)*

How does the above quote apply to this assignment? Since the fundamental problem being solved is sorting, we can ask what it means for a particular run to work correctly. There are two properties for a sort of an array with *n* elements to be correct:

1. After the sort the array elements are in non-decreasing order according to the comparison criteria used.
2. The collections of elements in the is the same after the sort as it was before the sort (nothing was lost and no extraneous elements were added).

So sanity checker functions that verify (or refute) these properties might be useful during your testing and debugging process. Of course, a sanity checker itself could be buggy, but usually, sanity checkers are pretty easy to get correct for an experienced programmer; also, since you don't have to worry about efficiency of the checkers, they are that much easier to get correct.

Also you should definitely try out your sort for the `Point` structs used in Lab-2. Maybe throw in a couple of other types.

For the Ambitious

We know that Merge Sort is an $O(n \log n)$ worst case runtime algorithm. But you could try to optimize it to reduce the runtime constant. You could even evaluate your implementation versus `qsort`(which is pretty darn optimized). Here are some ideas.

1. We've already discussed one optimization: when copying array entries, if it is possible to copy more than one `char` at a time, take advantage of this by copying something like an `int` at a time.
2. Big-Oh analysis tells us that Merge Sort is faster in the worst case than Insertion Sort or Selection Sort (quadratic time algorithms) -- *at least when n gets large enough.* **For small values of *n,* Insertion Sort in particular might actually run faster (as determined experimentally)**. Can we exploit this somehow? In the conventional presentation of Merge Sort, the base cases is when we have just a single element. We could instead just call an appropriate implementation of Insertion Sort once the sub-problem size gets small enough (this threshold determined experimentally). This way you get the best of both worlds.
3. We know there is quite a bit of copying that goes on in a conventional implementation of Merge Sort. **With a little cleverness, this can be reduced by merging back and forth between the two arrays instead of merging from one to the other and then copying back.** Of course in the end, your answer has to be in the correct array, but that can be done as a final copy if necessary. Yes, I'm waving my hands..... but give it some thought. Suggestion: focus on what happens on the way "back up" from the recursion (in fact, you can implement merge sort non-recursively).
4. There are lots of other possibilities...

I'm not offering extra credit for this kind of extra code tuning, but we will profile your submissions and announce the submissions that were fastest in practice. So there are bragging rights!

Comments

As it turns out with a little searching, you can find implementations of this homework online (I didn't realize this when I started writing this handout!) Any such code we can find will be included in the MOSS submission set. So **do your own work**! If you look at code online, make sure you completely understand what is going on and then use what you've learned to complete your own implementation (without referring to the code from the internet).

Despite the length of this handout, this isn't a hugely complex assignment and working your way through it should be a useful exercise.