

Data Structures

Announcements

- **The midterm is next Monday during class.** In order to allow for alternate seating, **we are in a different room: 200-34.**
- Midterm review session will be this Friday in b08 (12:50 - 1:40).
- The practice midterm solutions will be given out in section on Friday.
- Apparently, I put a different due date on the schedule (5/10) than on the Quicksort handout (5/12)—oops. The date on the schedule (5/10) is the correct one. In order to avoid confusion, I will allow you to turn in Quicksort on the 12th with no penalty, but doing so means you will be rushed on Lab #3 (you will only have a week). I urge you not to do this, but feel that it is the only fair thing.
- A few of the `#defines` in the Subroutines handout aren't quite right. `frameFoo` (in 2 places) should be `0x08` (not `0x0A`). When talking about the frame size, the prose on that page should be changed from 12 to 8. Oh yeah, and the parameters to `LINK` are flipped in a later example.

Last time

- Finished up stacks
- Subroutines
 - Calling and returning
 - RA in registers/global space/stack
 - JSR/RTS (does stack method)
 - Passing parameters
 - Registers/global space
 - Stack based (C calling convention)
 - Stack based (Pascal calling convention)

This time

- Subroutines (cont)
 - Passing parameters (cont)
 - Other calling conventions
 - Returning results
 - Local variables
 - Register conventions
- Where does the stack (A7) come from?

- Data structures
 - Compound data types
 - Arrays (1-D, 2D, nD)
 - structs
 - Structured data
 - Stacks (already covered)
 - Queues

Random Thought of the Day: “what do foo and bar mean?”

<http://kb.indiana.edu/data/aetq.html>

In the world of computer programming, "foo" and "bar" are commonly used as generic examples of the names of files, users, programs, classes, hosts, etc. Thus, you will frequently encounter them in manual (man) pages, syntax descriptions and other computer documentation.

The etymologies of these terms are unclear, but they are most commonly considered derivatives of the word "foobar". Foobar, in turn is usually defined as a deliberate misspelling of FUBAR, the acronym for a less polite version of "Fouled Up Beyond All Recognition" (or "Repair").

<http://student-www.uchicago.edu/users/tprater/fooandbar.html>

Here are some links to pages that discuss the origins of foo and bar as symbols in coding (like for class names; procedure, method, function names, variable names, and so on). Basically, it's that during WWII, the phrase "fubar" came into existence as an acronym for "F*cked Up Beyond All Recognition/Repair", although how this one word got split up into 2 is subject to debate.

Data structures

In one of the first lectures of CS 110, we learned how we could represent simple data types in a computer's memory. Soon thereafter, we learned how we could access these structures using assembly language. Today, we will talk about how to represent and use more complex data types. In other words, we will learn about data structures.

What exactly are we going to cover? Where going to tackle the topic from two directions. First we'll talk about techniques for representing generic data structures in assembly. We'll learn how to translate the C notions of structures, 1-D arrays, and multi-D arrays. Second, we'll talk about how two specific data structures translate into assembly: stacks (which we've already covered) and queues.

Arrays

1-D Arrays

One-dimensional arrays shouldn't be a difficult concept at this point in the class. Hopefully, the representation in memory (just putting values sequentially in memory and referring to the array as the base address) are familiar from CS 106. We've used some of the concepts of arrays already when we talked about case tables and also when we talked about stacks. You've also used arrays (in essence) in the Toys assignment. You can think of the screen as a big array of bytes. However, a quick example won't hurt. Because we know about stack storage and local variables, in my example the array will be on the stack (this also gives a good chance to get more experience with stack storage). The C code we want to translate is:

```
#define kArraySize 10

void foo (int x) {
    Char cArray[kArraySize];
    Word wArray[kArraySize];

    cArray[x] = 65;
    wArray[x] = -2;
}
```

Granted, the code is a little silly, but let's translate it anyway. First, let's just write the base code to start out with (that doesn't do any accesses). Basically, we just want to set up the stack frame correctly, allocating enough space for local variables. Let's also make x a little more accessible by moving it to D0:

```
#define kArraySize 10
#define kFooFrame kArraySize+2*kArraySize
#define cArrayOff -kArraySize
#define wArrayOff -(kArraySize+2*kArraySize)
#define xOff 8

asm foo() {
    LINK    A6, #-(kFooFrame)
    MOVE.W  xOff(A6), D0

    ...

    UNLK   A6
    RTS
}
```

Great...we have a stack frame and we have space for our 2 arrays (we allocated 30 bytes on the stack). Now, how do we make the two assignments? The first one is pretty straightforward, right? We can just do:

```
MOVE.B    #65, cArrayOff(A6, D0.W)
```

That will work here, but will it always work? What if we changed `kArraySize` to something else—say 300? There’s a really subtle problem that pops up. Look at the addressing mode we’re using. It is $d_8(A_n, X_n)$. The “8” subscript for the displacement is there for a reason. We really only have 8 bits for our displacement. If our array offset is too big, we won’t be able to use this addressing mode. We can fix our problem by using two steps:

```
LEA      cArrayOff(A6), A0
MOVE.B   #65, 0(A0, D0.W)
```

This fixes things because in our `LEA` we are now using a 16-bit offset. That allows us to have 64K on the stack, which should be plenty. Bigger offsets can be accomplished using addition.

What about our second instruction? That translates mostly the same way. The big difference is that now we need to multiply our index (we already saw this when accessing tables):

```
LEA      wArrayOff(A6), A0
LSL.W    #1, D0                // Multiply by 2...
MOVE.B   #-2, 0(A0, D0.W)
```

That concludes our discussion of 1-D arrays. You could imagine applying the same techniques to arrays where each individual item was larger.

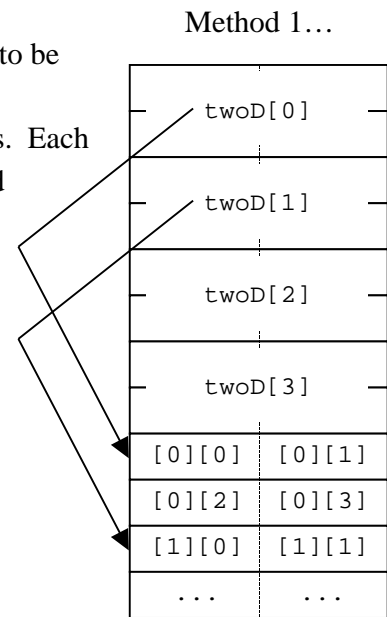
2-D Arrays, Method 1

2-D arrays should also be familiar from CS 106, but there’s something to be said for looking at them again with your new knowledge of memory. Remember that one way to do 2-D arrays is to have an array of pointers. Each of these pointers points to an array of the actual elements. That method (method 1) is illustrated on the right. It corresponds to the method in C of declaring a (**). The array on the right might be created in C like:

```
{
    char **array;
    int i;

    array = malloc (4 * sizeof (char*));
    for (i = 0; i < 4; i++) {
        array[i] = malloc (4 * sizeof (char));
    }

    ...
}
```



Note that this translation doesn’t correspond perfectly to the picture. In the picture, I have placed the “subarrays” directly after the array itself. In C, the `malloc` call is allowed to allocate

memory wherever it feels like. The C code illustrates the more general concept. However if we were to use this technique for representing a 2-D array in assembly language, we would likely set aside a single chunk of memory (as my diagram shows).

Writing code to access this array is fairly straightforward. We can use the first index (aka the row) to access into the first array to get the address of the subarray. Then, we can use the second index (aka the column) to access into the subarray. Basically, we're just combining two 1-D array operations.

2-D Arrays, Method 2

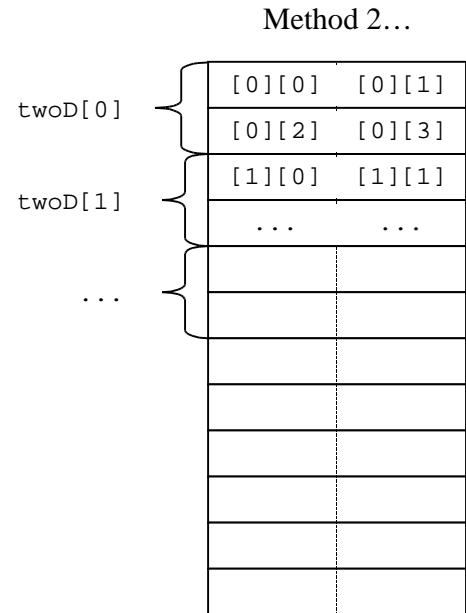
What's the second way to do 2-D arrays? Instead of having an array of pointers, we can just have an array of arrays. That is, the "outer" array is an array where each element of the array is an inner array. That's diagramed on the right. Method 2 is equivalent in C to doing:

```

{
    char array[4][4];

    ...
}

```



There are some interesting properties of method 2:

- We can find the address of an element in our array using:
$$\text{elementAddress} = \text{base} + \text{elementSize} * (\text{row} * \text{numCols} + \text{col})$$
...where "row" is the first index and "col" is the second. Interestingly enough, this is just like accessing a 1-D array whose index is $(\text{row} * \text{numCols} + \text{col})$.
- In order to find any element in our array, we must know the number of columns that the array has. That's because in order to access an array, we need to know how big its elements are. To access the "outer" array, we must know how big the inner elements (the subarrays) are. We can see this from the previous formula. Note that this is why when you pass arrays in C, you need to specify the widths of all but the first array.

Comparing Method 1 and 2

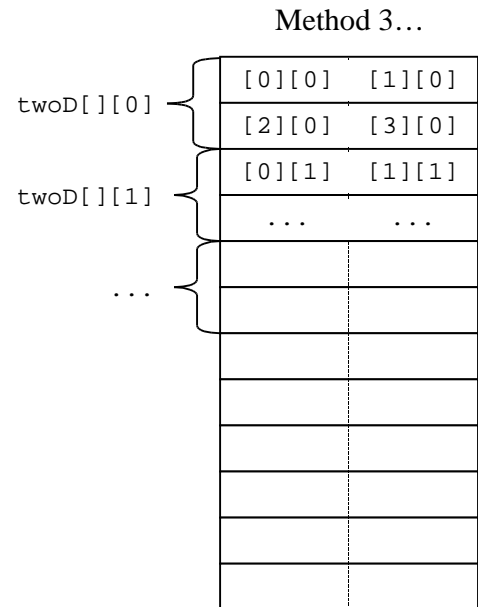
How can we compare the two methods? Well, one thing that wasn't really illustrated by my diagrams (but mentioned) is that in method 1, the subarrays can be located all over memory. They don't have to be close to the main array in any way. However, in method 2 the entire array has to be in one big chunk of memory.

Another difference is in memory usage. In method 1, the array took up $(4 * 32) + (4 * 4 * 8) = 256$ bytes. However, in method 2, the array took up $(4 * 4 * 8) = 128$ bytes. We've reduced our space requirements by half just because we didn't need to store the pointers. Granted that this example is a bit extreme (after all, usually you use subarrays that are larger than 4 bytes).

As we've already seen, access method is a third difference. For the first method, we have to make two memory accesses to get to an element (we need the base address of the subarray and then the element itself). For the second method, we access an element by calculating its offset and then making one memory access. Which of these is faster? As I said in class last time (in response to a question), the speed of an operation is largely related to the amount of data transferred to/from memory. That would make method #2 faster. However, an exception to this rule is multiplication. Multiplication in general is a slow operation. Is it slower than memory access? It really depends. However, we can claim victory for method #2 if we make sure that the size of our subarrays is a power of 2. If this is the case, we can use `LSL` to do the multiplication!

2-D Arrays, Method 3 (?)

There's one other method of storing 2-D arrays that you should know about. Well, it's *technically* another method at least. In method 2, remember that our "outer" or *major* array was an array of rows. Inside there we had the column elements. Method 2 is called "row major" order. You can imagine doing the opposite: making the major array an array of columns and then storing the rows. This would be "column major" order. The distinction is a little weak (after all, one could argue that you could just put the column in the first element and the row in the second element and forget about it). However, you will hear the term from time-to-time.



Multi-D arrays (and conclusion)

So you should be able to see how to do multi-dimensional arrays by now. You still have a choice of methods, but this time you are going one step further (an array of pointers to arrays of pointers to elements or arrays of arrays of arrays)...

Ok, enough about arrays. I'm going to leave you without any sample code for doing multi-D arrays (because I think that both methods should be straightforward given what you know). However, if you are interested in more samples, feel free to stop by office hours.

Structs

So the next thing that we really need to go over is how to do structures in assembly language. Like arrays, this too is fairly straightforward. In fact, there aren't really even any interesting special cases or tricks to talk about. You simply allocate enough space for all the elements in the structure and then access them using the proper offsets. One example should illustrate the point:

```
typedef struct {
    Word foo;
    Char *bar;
    DWord cool;
} fubar;
```

How would we allocate memory for this? Well, we would simply allocate 10 bytes of memory. If we had a pointer to this structure in A0, how could we access it?

```
MOVE.W    (A0), D0           // Access foo component
MOVEA.L   2(A0), A1         // Access bar component
MOVE.L    6(A0), D1         // Access cool component
```

...of course, we probably should `#define` the 0, 2, and 6, but otherwise that's just fine.

Stacks

Technically, this is where stacks really fit in. As I said in the beginning of the handout, learning about stacks isn't quite the same as learning about arrays or structures. Whereas those are language concepts, stacks generally aren't (at least not in C). That's because arrays and structures are ways of building types, whereas stacks are types in themselves. In fact, stacks are generally built using arrays or structures as primitives (as we saw in class).

As we talked about in class, there are different ways to represent stacks (the two most common are using linked structures and using a simple array-type structure). Using our knowledge of arrays and structures, we could imagine building either one. However, in the context of subroutines ("the" stack), we always use the simple concept.

Queues

Typically, when you talk about stacks, you also talk about queues. That's because the two structures have a lot in common. Stacks are LIFO structures. That is, the last item you entered is the first one out. Queues, on the other hand, are FIFO: first in first out. Queues also have two standard implementations that correspond to the two implementations for stacks. We will talk about queues at the beginning of the next handout.