

CSC2/452 Computer Organization

More Arrays, Pointers

Sreepathi Pai

URCS

October 2, 2019

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Assignments and Homeworks

- ▶ Assignment #2 landing today (currently undergoing testing)
 - ▶ Due Saturday Oct 12, 7PM
- ▶ Homework #4 is out
 - ▶ Due next Monday, Oct 7 at *beginning* of class
- ▶ Midterms, Wednesday Oct 9 in class
 - ▶ Will send instructions on Blackboard
 - ▶ One 8.5x11in double-sided handwritten sheet of notes allowed
 - ▶ Content inclusive of Oct 4 class on midterm
- ▶ Oct 7, review class
 - ▶ Please collect questions on Piazza

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Pointers

- ▶ C's abstraction of addresses
- ▶ Declared as `sometype *var`
 - ▶ Indicates `var` is a pointer to `sometype`
 - ▶ Tracks size of data
- ▶ You modify pointers (i.e. `var`) by treating them as ordinary variables
 - ▶ Assignment, `var = &varofsometype`
 - ▶ Arithmetic, `var = var + 1` (but 1 is internally multiplied by size of `sometype`)
- ▶ You read/write the data pointed to by a pointer using indirection operator (`*`)
 - ▶ `var2ofsometype = *var`, which assigns the value `varofsometype` to LHS
 - ▶ Also called dereferencing
- ▶ Primarily used in C for
 - ▶ Passing arguments for modification by function
 - ▶ Arrays, which are just pointers under the hood

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Today on Arrays

- ▶ Stack-allocated arrays
 - ▶ local variables that are arrays
 - ▶ limited in size
- ▶ One-dimensional arrays
 - ▶ What about storing matrices?
- ▶ Today:
 - ▶ heap-allocated arrays (essentially unlimited)
 - ▶ multidimensional arrays

Memory Allocators

- ▶ Memory on a system is a (finite) resource
- ▶ Must be distributed among programs
- ▶ Need something that tracks who “owns” a particular piece of memory
- ▶ This something is called a *memory allocator*
- ▶ There are many memory allocators in a system
 - ▶ Kernel (or System Allocator)
 - ▶ Runtime library allocator
 - ▶ Custom application allocators
- ▶ You will build a memory allocator later in the course...

Memory Allocator

Usually, supports 3 functions:

- ▶ allocation: to obtain a region of memory
- ▶ free: to release the region of memory
- ▶ reallocation: usually to resize a region of memory

Stack Allocation

We have already seen stack frame creation in previous lectures:

- ▶ Creating a stack frame: allocating memory on the stack
 - ▶ `sub from %esp`
- ▶ Destroying a stack frame: freeing memory on the stack
 - ▶ `add to %esp`
- ▶ Note, both operations really only supported at top of stack

Stack allocators are a special case of *bump* allocators.

- ▶ Only allocate at one end of a memory region
- ▶ Use `add/sub` to allocate/free memory
- ▶ Not very sophisticated, but very fast

C Runtime Allocator

The C standard library supports the following *heap* allocators, all require including `stdlib.h`:

- ▶ `malloc`: allocate memory of a certain size
- ▶ `calloc`: allocate memory and zero it
- ▶ `realloc`: change size of memory (may move data)
- ▶ `free`: free memory allocated by the above functions

- ▶ All allocators return a pointer to the newly allocated region of memory or `NULL` if they fail.

Heap allocation

- ▶ The heap is a region of memory like the stack
 - ▶ But grows towards higher addresses (unlike the stack)
- ▶ Data stored in the heap is accessible to the entire program
 - ▶ Anybody who has the pointer to the data can access it
- ▶ Heap data is only freed automatically when your program exits
 - ▶ You must manually free heap data you no longer want
 - ▶ Otherwise your program could run out of memory

Garbage collection

- ▶ Keeping track of memory that is no longer needed and freeing it is very tedious
 - ▶ Mistakes lead to difficult to track down bugs
- ▶ Most languages now use a technique called “garbage collection” (GC)
 - ▶ Automatically detects which regions of memory are no longer being used
 - ▶ Java, Go, JavaScript, Python, etc.
 - ▶ Tends to be unpredictable, and can cause performance issues
- ▶ C does not support GC by default

Back to C memory allocation

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);

int *p;

// allocate 1000 ints, note size is size in bytes
p = (int *) malloc(1000 * sizeof(int));

// --- OR --- //

// or allocate 1000 ints initialized to zero
p = (int *) calloc(1000, sizeof(int));

// free allocated memory
free(p);
```

- ▶ The `sizeof` C **operator** returns the size of the type

Allocating an array on the heap

```
int *alloc_and_init_array(int N) {
    int *x;

    x = (int *) malloc(N * sizeof(int))

    // initialize it
    for(int i = 0; i < N; i++) {
        x[i] = i*i;
    }

    return x;
}
```

- ▶ Note we're returning the value of x
 - ▶ I.e. the address *in* x , not the address *of* x
- ▶ Since the address is on the heap, it is independent of the function call
- ▶ You can treat pointers to the heap as any other pointer

Summary of Memory Allocation

- ▶ Call `malloc` (or `calloc`) when you need memory
 - ▶ Obtain a pointer to the memory region
- ▶ Use the pointer to read/write to the memory region
- ▶ Call `free` on the pointer when you no longer need it

Common Bugs using Dynamic Memory

- ▶ Memory leaks
 - ▶ When you lose the pointer to an allocated region of memory
 - ▶ Can't be freed until program exits
 - ▶ “harmless”, program just consumes more and more memory as it runs
- ▶ Reading uninitialized memory
 - ▶ Memory from `malloc` should be initialized before reading it
- ▶ Out-of-bound accesses
 - ▶ Pointer points outside the allocated region
 - ▶ Undefined behaviour!
- ▶ Use-after-free
 - ▶ Attempt to access memory region after `free`
 - ▶ Essentially a dangling pointer pointing to the heap
 - ▶ Dangerous!
- ▶ Double-frees
 - ▶ Trying to call `free` on the same pointer twice
 - ▶ Undefined behaviour!

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Strings in C

- ▶ Strings in C are just character arrays
 - ▶ `char *c` is a string
 - ▶ as are `char c[10]`, `unsigned char`, etc.
 - ▶ also types `uint8_t` and `int8_t`
- ▶ C does not track array length
 - ▶ So by convention, strings end with a NUL character (note the single L)
 - ▶ NUL is a unprintable (and untypeable character)

Specifying unprintable characters

- ▶ ASCII characters 0 to 31 are non-printable characters
 - ▶ 0x9 is TAB
 - ▶ 0xa is LINEFEED/LF (move to next line)
 - ▶ 0xd is CARRIAGE RETURN/CR (return to head of line in a printer)
 - ▶ 0xc is FORM FEED (eject page out of printer)
- ▶ Among these you will encounter NUL (0x0), TAB (0x9), LF (0xa), CR (0xd)
 - ▶ You type these as '\0' (NUL), '\t' (TAB), '\n' (LF), '\r' (CR)
 - ▶ These are called escape sequences

Strings in C

```
int main(void) {  
    // C automatically puts a \0 at the end  
    // length of array is also automatically calculated  
  
    char greeting[] = "hello World\n";  
  
    greeting[0] = 'H';    // note character literals use apostrophes  
}
```

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Memory vs Arrays

- ▶ Memory is (logically) one-dimensional
 - ▶ addresses go from 0 to $2^n - 1$, for some n
- ▶ 1-D arrays map naturally to memory
- ▶ What about higher-dimensional arrays?
 - ▶ 2-D, i.e. matrices
 - ▶ higher-dimensions, “tensors”

The problem

- ▶ For 1-D
 - ▶ $a[i]$ translates to memory address $a + i$

How do we translate an 2-D array index (i, j) , where i indicates row, and j indicates column?

- ▶ Assume number of rows is r and columns is c

Row-major ordering

```
| a b c d |  
| e f g h | => [ a b c d e f g h i j k l ]  
| i j k l |
```

- ▶ Index (i, j) is translated to $j + i * c$
 - ▶ Here $r = 3$ and $c = 4$
- ▶ All elements of the same row are laid out next to each other in memory

Row-major ordering in C

```
float *m;
int R = 3, C = 4;

m = (float *) malloc(R * C * sizeof(float))

for(int i = 0; i < R; i++) {
    for(int j = 0; j < C; j++) {
        m[j + i * C] = 1.0;
    }
}
```

Column-major ordering

```
| a b c d |  
| e f g h | => [ a e i b f j c g k d h l ]  
| i j k l |
```

- ▶ Index (i, j) is translated to $i + j * r$
 - ▶ Here $r = 3$ and $c = 4$
- ▶ All elements of the same column are laid out next to each other in memory

Column-major ordering in C

```
float *m;  
int R = 3, C = 4;  
  
m = (float *) malloc(R * C * sizeof(float))  
  
for(int i = 0; i < R; i++) {  
    for(int j = 0; j < C; j++) {  
        m[i + j * R] = 1.0;  
    }  
}
```

Generalizing to higher dimensions

- ▶ For index (i, j, k) in space with dimensions (x, y, z)
 - ▶ Linear address: $k + j * z + i * y * z$
- ▶ Note: knowledge of the size of dimension of index i is not necessary

Statically Multidimensional Arrays

- ▶ C supports syntax like `A[i][j]` for statically allocated arrays
 - ▶ Sizes must be specified at compile-time
 - ▶ C99 makes this more flexible, see the textbook for details
- ▶ Such arrays are always in row-major order
- ▶ Since the compiler translates addresses for you
 - ▶ And C doesn't track dimensions of arrays
 - ▶ You must always specify all dimensions except last when passing these arrays as parameters, if you want to use them

```
int a[10][2];
```

```
void sum(int m[][2]) {
```

```
}
```

Arrays of Pointers

```
int *a[10];

for(i = 0; i < 10; i++) {
    a[i] = (int *) malloc(2);
}

for(i = 0; i < 10; i++) {
    for(j = 0; j < 2; j++) {
        a[i][j] = i + j;
    }
}
```

- ▶ Don't confuse this with `int a[10][2]`
 - ▶ This is commonly written as `int **a`
- ▶ This is an array of pointers
 - ▶ The previous was an array of ints
- ▶ But this is another, C-specific idiom to emulate multi-dimensional arrays

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Addresses of Data vs Addresses of Code

- ▶ In assembly, labels are addresses
- ▶ Labels can indicate both data and code
 - ▶ Naturally, since both data and code are in memory
- ▶ What about in C?

Function Pointers

- ▶ C allows function pointers
- ▶ But must store more information than just address of function
 - ▶ Parameters and their types
 - ▶ Return type of function

Syntax

```
int sum(int a, int b);
```

- ▶ `sum` takes two `int` parameters
- ▶ `sum` is a function returning an `int`
- ▶ How shall we declare a variable to points to `sum`?

Try #1

```
int *v;
```

- ▶ We're trying to make `v` a pointer to `sum`
- ▶ This is a pointer to an `int`
- ▶ Also lacks information about parameters

Try #2: Adding parameter information

```
int *v(int a, int b);
```

- ▶ This now contains parameter info
 - ▶ Can clearly see this is not a pointer to `int`
- ▶ But, can this be confused for a function returning a *int* pointer?

Try #3: Disambiguating function pointers from function declarations

```
int (*v)(int a, int b);
```

- ▶ This contains parameter info
 - ▶ Can clearly see this is not a pointer to `int`
- ▶ The `(*v)` indicates that `v` is a pointer to function
- ▶ And the function that `v` points to returns an `int`

Using function pointers

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int (*v)(int a, int b);

    v = sum;
    printf("address of sum: %p\n", v);

    v(1, 2);
}
```

Output:

```
address of sum: 0x556bfef3964a
```

- ▶ No & required (though it can be used)
- ▶ No * required (though it can be used)
 - ▶ (*v)(1, 2), watch precedence

Outline

Administrivia

Recap

More on Arrays

Strings

Multidimensional Arrays

Function Pointers

Structures and Unions

Record Types

```
struct point {  
    float x;  
    float y;  
};  
  
struct point pt;  
struct point pts[10];  
struct point *polygon;
```

- ▶ struct holds multiple values
- ▶ Each *field* of struct has a name
- ▶ struct can be nested

Recursive structures

```
struct node {  
    int value;  
    struct node *left;  
    struct node *right;  
};
```

- ▶ This is a *recursive* structure
 - ▶ Possibly from a binary tree
- ▶ The “self-references” must be pointers

struct interactions with pointers

```
struct node {  
    int value;  
    struct node *left;  
    struct node *right;  
};  
  
struct node *head;
```

Which of these accesses head's right node?

- ▶ `*head.right`
- ▶ `(*head).right`

The -> operator

```
struct node {
    int value;
    struct node *left;
    struct node *right;
};

struct node *head;
```

Since field access (.) has higher precedence than dereference, you must use (*head).right.

- ▶ Alternative: head->right, which has same precedence as .

Structure Layout

```
#include <stdio.h>

struct node {
    int value;
    struct node *left;
    struct node *right;
};

int main(void) {
    printf("%d\n", sizeof(struct node));
}
```

Output (on a LP64 system)?

- ▶ 20
- ▶ 24
- ▶ 32

Structure Memory Layout

Tight packing:

```
value left      right
[0123] [01234567] [01234567]
```

Packing with “holes” (also called padding)

```
value      left      right
[0123] [0123] [01234567] [01234567]
```

- ▶ Structure layout in memory is implementation defined
- ▶ Usually:
 - ▶ Struct size is a multiple of largest field
 - ▶ Each individual field is naturally aligned

Unions

```
union intvar {  
    char c;  
    short s;  
    int i;  
    long l;  
};
```

- ▶ Like struct, union contains fields
- ▶ However, all fields in a union *overlap* in memory
 - ▶ At most one contains valid data

Union Memory Layout

```
c|  
s-|  
i---|  
l-----|  
[01234567]
```

- ▶ This union occupies 8 bytes of memory
 - ▶ The size of its largest field
- ▶ But all fields overlap
 - ▶ Writing to one changes the others as well
- ▶ Writing to one field, and reading that data through another field is allowed
 - ▶ But it is implementation-defined

Summary

- ▶ Seen how C is implemented at machine level
 - ▶ Variables, Expressions
 - ▶ Conditionals, Loops
 - ▶ Functions
 - ▶ Pointers, Structures and Unions
- ▶ Every program executes a stream of assembly instructions
 - ▶ If you know the assembly instructions you know what is happening