C Introduction

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 7 due Monday on Gradescope
- Midterm 2 is on April 3
  - Similar format to midterm 1
  - Schedule with SDAC early if needed

Structures

`struct` - Structures in C

- Act like Java classes, but no methods and all public fields
- Stores fields adjacently in memory (but may have padding)
- Compiler determines padding, use `sizeof()` to get size
- Name of the resulting type includes word `struct`

```
struct foo {
    long a; 8
    int b; 4
    short c; 2
    char d; 1
};

struct foo x;
x.b = 123;
x.c = 4;
```

When you create a struct in C, the compiler decides where to place each field in memory. It tries to align each field on an address that's a multiple of it's natural size, to make the CPU access faster.

the order of this list is important. It tells the compiler how to build this thing.

4+2+1=7
may be one byte of padding? I don't know.

↳ declare a variable x of type struct foo.

I can set directly the fields.

A struct in a struct?
Yes, just set a struct as one field of another struct.

Structure Literals

We can use struct literals just like we saw with the array literals.

```
struct a {
    int b;
    double c;
};

/* Both of the following initialize b to 0 and c to 1.0 */
struct a x = { 0, 1.0 };
struct a y = { .b = 0, .c = 1.0 };
```

"a" is a part of the type, not a part of the name.

→ must give values to fields in order.

→ you can use equal notation to give them values in other orders.

struct a z; ⇒ If I don't give any values, just allocate the memory and whatever is in memory is what's going to be there. I don't know what they are.

typedef

`typedef` - give new names to any type!

typedef unsigned long size_t;
typedef unsigned long address_t;

- Fairly common to see several names for same data type to convey intent
- Ex: `unsigned long` may be `size_t` when used in sizes
- Examples:

```
typedef int Integer;
Integer x = 4;
typedef double ** dpp;
```

Integer becomes another name for int.

dpp ptr; (Using typedef can make complex pointer types look simpler)

- Used with *anonymous structs*:

```
typedef struct { int x; double y; } foo;
foo z = { 42, 17.4 };
```

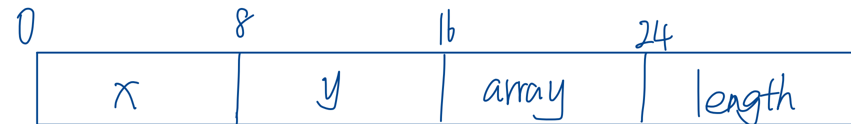
struct { ... } defines an anonymous struct (no struct name given).
The typedef immediately gives it the alias foo.
You can now create variables like foo z; instead of writing struct something z;.

Struct Example

Assuming I have no padding because all of these are 64 bits

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

remember we need this ";" at the end!



foo a;

a.array ⇒ "a+16" to get me to the array.

Struct Example

Sum2 gets a pointer, it must dereference to access fields.

arg -> x is same as (*arg).x

```
long sum2(foo *arg) {
    long ans = arg->x;
    for(long i = 0; i < arg->length; i += 1)
        ans += arg->y * arg->array[i];
    return ans;
}
```

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

the first parameter (arg) is passed in register %rdi

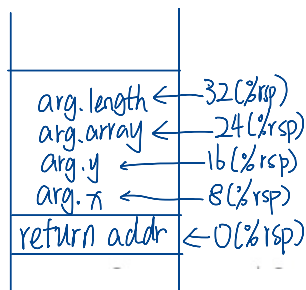
```
sum2:
    movq    (%rdi), %rax        ans = arg->x
    movq    24(%rdi), %r8       r8 = arg->length
    testq   %r8, %r8            performs bitwise AND.
    jle     .LBB1_3             if negative, jump to .LBB1_3
    movq    8(%rdi), %rdx       rdx = arg->y
    movq    16(%rdi), %rsi      rsi = arg->array
    xorl    %edi, %edi          i=0
.LBB1_2:
    movq    (%rsi,%rdi,8), %rcx rcx = array[i]
    imulq   %rdx, %rcx          rcx = y * array[i]   (integer multiply (signed))
    addq    %rcx, %rax          ans += .....
    incq    %rdi                i++                  (increment)
    cmpq    %rdi, %r8           compare i<length
    jne     .LBB1_2
.LBB1_3:
    retq
```

Struct Example

sum1 gets a copy of the whole struct (foo arg), it accesses fields directly on the stack.

```
long sum1(foo arg) {
    long ans = arg.x;
    for(long i = 0; i < arg.length; i += 1)
        ans += arg.y * arg.array[i];
    return ans;
}
```

Why does the compiler access the struct fields using (%rsp)+offset, instead of something like (%rdx-something)?
Modern compilers often omit it to save a register.
when you don't see %rbp, the compiler just use %rsp+offset instead.




```
sum1:
    movq   8(%rsp), %rax        ans = arg.x
    movq   32(%rsp), %r8        r8 = arg.length
    testq  %r8, %r8
    jle    LBB0_3
    movq   16(%rsp), %rdx       rdx = arg.y
    movq   24(%rsp), %rsi       rsi = arg.array
    xorl   %edi, %edi           i=0
.LBB0_2:
    movq   (%rsi,%rdi,8), %rcx  (array+i*8)  rcx = array[i]
    imulq  %rdx, %rcx           rcx = y*array[i]
    addq   %rcx, %rax           ans += .....
    incq   %rdi                 i++
    cmpq   %rdi, %r8            compare i<length
    jne    .LBB0_2
.LBB0_3:
    retq
```

imulq: integer multiply (signed)
incq: increment

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

C Reference Guide