# C Introduction

## CS 2130: Computer Systems and Organization 1

**Xinyao Yi** Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

## Announcements

- Homework 7 due tonight at 11:59pm on Gradescope
- Exam 2 Friday

# Struct Example

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

```
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;
```

```
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]
    addq    %rcx, %rax          ans += .....
    incq    %rdi                i++
    cmpq    %rdi, %r8           compare i<length
    jne     .LBB1_2
.LBB1_3:
    retq
```

integer multiply (signed)

increment

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

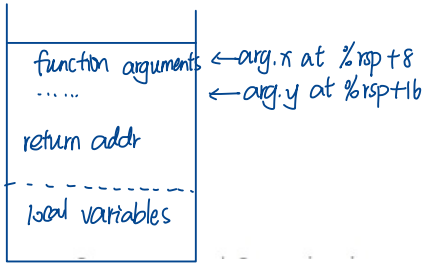
# Struct Example

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;
}
```

```
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
    imulq   %rdx, %rcx
    addq    %rcx, %rax
    incq    %rdi
    cmpq    %rdi, %r8
    jne     .LBB0_2
.LBB0_3:
    retq
```

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.



# C Reference Guide

Over the past couple of weeks, we've been talking the reference guide in reading "C reference", from floating points to pointers.

Union

A union is like a struct, except that all of the fields are stored in the same memory address. In practice, this means only one of them has a meaningful value at a time.

# Calling Functions

The C code

```
long a = f(23, "yes", 34uL);
```

follow the calling conventions if I don't know anything else about function f.

compiles to

```
movl $23, %edi
leaq label_of_yes_string, %rsi
movq $34, %rdx
callq f
# %rax is "long a" here
```

put the address of "yes" to rsi.

without respect to how `f` was defined. It is the calling convention, not the type declaration of `f`, that controls this.

# Calling Functions

But, if the C code has access to the type declaration of `f`, then it might perform some implicit casting first; for example, if we declared

```
long f(double a, const char *b, double c);
```

```
long a = f(23, "yes", 34uL);
```

We need to make sure that we have the declaration of the function before we call it. So the type checker knows how the function supposed to look.

then the call would be interpreted by C as having implicit casts in it:

```
long a = f((double)23, "yes", (double)34uL);
```

# Calling Functions

When I call a function with floating pointer numbers, there's a whole set of registers that just yield floating point numbers.

and the arguments would be passed in floating-point registers, like so:

```
movl $23, %eax
cvtsi2sd %eax, %xmm0          # first floating-point argument
leaq label_of_yes_string, %rdi # first integer/pointer argument

movl $34, %eax
cvtsi2sd %eax, %xmm1          # second floating-point argument

callq f
# %rax is "long a" here
```

Not edi anymore. It's a general-purpose integer, not an argument.

the first floating point number register. (follow the calling conventions for floating point numbers).

cvtsi2sd convert signed integer to signed double.

For different types of the parameters, use different kinds of registers in order.

# Functions Declaration

We want to know what f is so that we can call it appropriately.

```
int f(int x);
```

→ just a function header with a semicolon, no body.

just different names

- Declaration of the function — We leave it's implementation somewhere else.
- Function header
- Function signature
- Function prototype

We want this in every file that invokes `f()`

We want to know what f looks like in every single one of our C files.

## Functions Definition

We only want the function definition to appear in one place.

```
int f(int x) {
    return 2130 * x;
}
```

- Definition of the function

We only want this in **one** `.c` file

- Do not want 2 definitions
- Which one should the linker choose?

# Header Files

C header files: `.h` files

- Written in C, so look like C
- Only put header information in them
  - Function headers
  - Macros
  - `typedef`s
  - `struct` definitions
- Essentially: information for the **type checker** that does not produce any actual binary
- `#include` the header files in our `.c` files (take the entire file and paste it at the top of my code).

## Big Picture

Header files

- Things that tell the type checker how to work
- Do not generate any actual binary

C files

- Function definitions and implementation
- Include the header files (one header file can be used by multiple C files).

# Including Headers

`#include "myfile.h"` (I want to include a file that I wrote).

- Quotes: look for a file where I'm writing code
- Our header files

`#include <string.h>`

- Angle brackets: look in the standard place for includes (a set of includes that came with the compiler).
- Code that came with the compiler
- Likely in `/usr/include`