

# C Introduction

---

CS 2130: Computer Systems and Organization 1

April 3, 2023

# Announcements

- Homework 7 (C functions) due tonight at 11pm
- Bring questions to our review session on Wednesday
- Exam 2 on Friday

# Other Types and Values

- Literal values - integer literals are implicitly cast
  - `unsigned long very_big = 9223372036854775808uL`
  - `u` for unsigned, `L` for long
- **enum** - named integer constants (in ascending order)
  - `enum { a, b, c, d=100, e };`
  - `int foo = e;`
- **void** - a byte with no meaning or "nothing"
  - Pointers: `void *p`
  - Return values: `void myfunction();`
- Casting - changing type, converting
  - Integer: zero- or sign-extend or truncate to space
  - Int to float: convert to nearby representable value
  - Float to int: truncate remainder (no rounding)

# 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;  
    int b;  
    short c;  
    char d;  
};
```

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

# Structure 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 };
```

# typedef

`typedef` - give new names to any type!

- 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;
```

- Used with *anonymous structs*:

```
typedef struct { int x; double y; } foo;
```

```
foo z = { 42, 17.4 };
```

# Struct Example

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

# Struct Example

```
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;
}
```

```
sum2:
    movq    (%rdi), %rax
    movq    24(%rdi), %r8
    testq   %r8, %r8
    jle    .LBB1_3
    movq    8(%rdi), %rdx
    movq    16(%rdi), %rsi
    xorl    %edi, %edi
.LBB1_2:
    movq    (%rsi,%rdi,8), %rcx
    imulq   %rdx, %rcx
    addq    %rcx, %rax
    incq    %rdi
    cmpq    %rdi, %r8
    jne    .LBB1_2
.LBB1_3:
    retq
```



# Struct Example

```
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  
    movq    32(%rsp), %r8  
    testq   %r8, %r8  
    jle    LBB0_3  
    movq    16(%rsp), %rdx  
    movq    24(%rsp), %rsi  
    xorl    %edi, %edi  
.LBB0_2:  
    movq    (%rsi,%rdi,8), %rcx  
    imulq   %rdx, %rcx  
    addq    %rcx, %rax  
    incq    %rdi  
    cmpq    %rdi, %r8  
    jne    .LBB0_2  
.LBB0_3:  
    retq
```



# C Reference Guide

# Calling Functions

The C code

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

compiles to

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

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);
```

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

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

# Calling Functions

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
```

# Function Declaration

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

- Declaration of the function
- Function header
- Function signature
- Function prototype

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

# Function Definition

```
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
  - `typedefs`
  - `struct` definitions
- Essentially: information for the **type checker** that does not produce any actual binary
- `#include` the header files in our `.c` files

# 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

# Including Headers

```
#include "myfile.h"
```

- 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
- Code that came with the compiler
- Likely in `/usr/include`

`string.h`