C Introduction
Functions, Header Files

CS 2130: Computer Systems and Organization 1
April 6, 2026

Announcements

- Homework 8 releasing soon, due next Monday on Gradescope

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
  - `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

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`

Macros

`#define NAME something else`

- Object-like macro
- Replaces `NAME` in source with `something else`

`#define NAME(a,b) something b and a`

- Function-like macro
- Replaces `NAME(X,Y)` with `something Y and X`

Lexical replacement, *not* semantic

Interesting Example

```
#define TIMES2(x)   x * 2          /* bad practice */
#define TIMES2b(x) ((x) * 2)       /* good practice */

int w = ! TIMES2(2 + 3);
```

`int w = ! 2 + 3 * 2;` ⟹ `!2 + (3*2)` ⟹ `0+6 = 6`

```
int y = ! TIMES2b(2 + 3);
```

`int y = ! ((2+3) * 2 );` ⟹ `!(5*2)` ⟹ `!10` ⟹ `0`

Examples and More

- header example
- `string.h`
- variadic functions

Memory

The Heap

The heap: unorganized memory for our data

- Most code we write will use the heap
- *Not a heap data structure...*

The Heap: Requesting Memory

```
void *malloc(size_t size);
```

- Ask for `size` bytes of memory
- Returns a `(void *)` pointer to the first byte
- It does not know what we will use the space for!
- Does not erase (or zero) the memory it returns

The Heap: Freeing Memory

Freeing memory: `free`

```
void free(void *ptr);
```

- Accepts a pointer returned by `malloc`
- Marks that memory as no longer in use, available to use later
- You should `free()` memory to avoid *memory leaks*