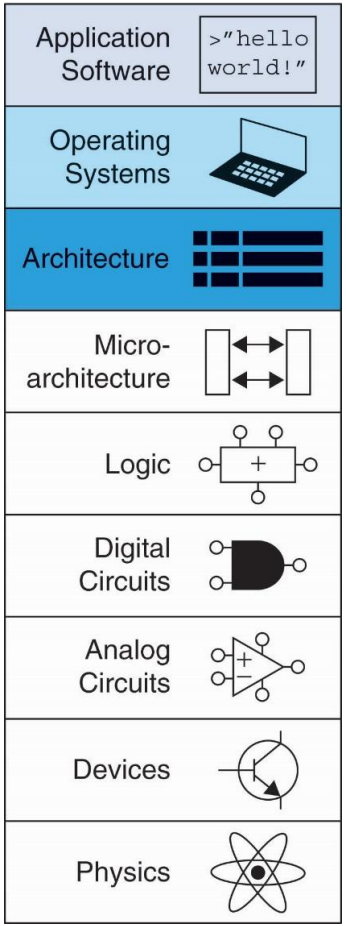
COSC175 (Systems I): Computer Organization & Design

Professor Lillian Pentecost
Fall 2024

Warm-Up November 7

- Where we were
  - Picking back up with memory allocation and function calls, from a C perspective and a RISC-V assembly perspective
- Where we are going
  - How are high-level languages translated into assembly instructions?
  - Demo w/ command line, GDB, and Makefile
- Logistics, Reminders
  - TA help 7-9PM on Sundays, Tuesdays, Thursdays in C107
  - LP Office hours M 9-10:30AM, Th 2:30-4PM
  - Weekly Exercises due Friday 5PM
  - Lab 6 due **Tuesday 10PM** (since I'm posting Part 2/3 a little later than expected)

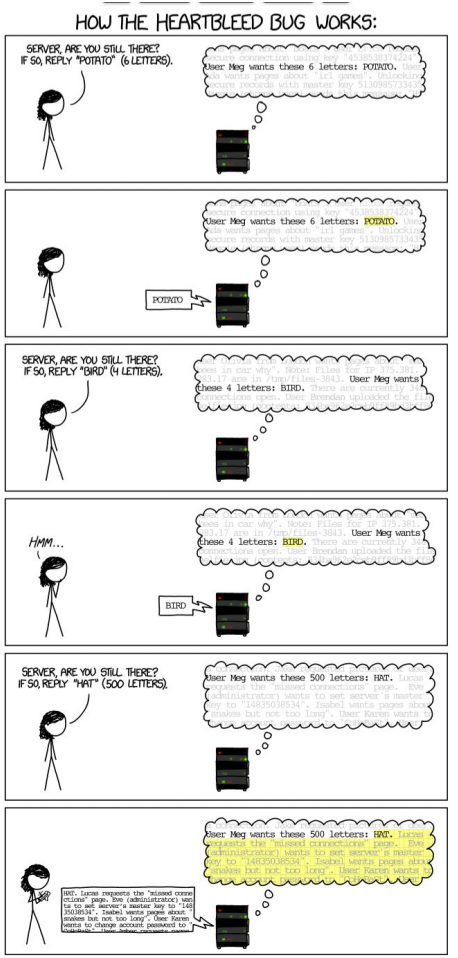


Dynamic memory allocation on the heap

- `void * calloc (size_t num, size_t size)`
  - Allocate a block of `num` elements, each of `size` bytes, for a total block of `num*size`
  - Initializes memory to zero values
  - Returns a `NULL` pointer on a failure
  - `int* ptrA = calloc(10, sizeof(int)); // allocates 1 pointer on the stack, with value referencing block of 10 int on heap`
- `void * malloc (size_t size)`
  - Allocate a block of `size` bytes
  - (often, called as `malloc(elements*sizeof(<datatype of element>)`)
  - Does NOT initialize memory
  - Returns a `NULL` pointer on a failure
  - `int* ptrB = malloc(10*sizeof(int)); // allocates 1 pointer on the stack, with value referencing block of 10 int on heap`

Dynamic memory allocation

- `void * calloc (size_t num, size_t size)`
  - Initializes memory to zero values
- `void * malloc (size_t size)`
  - Does NOT initialize memory (or `know` size per element)
- How to choose?
  - `calloc` is generally safer, but slower
  - If you make an error with `malloc`, you can leak existing contents, or addresses you shouldn't be able to access
    - Take security with Prof. Alfeld if you want to know how badly this can go wrong



https://xkcd.com/1354/

Clean up after yourself!

- `void * calloc (size_t num, size_t size)`
- `void * malloc (size_t size)`
- `void free(void * ptr)`
  - Frees a block of heap memory previously allocated by `malloc` or `calloc`, value of `ptr` is starting address
  - Call 1 `free` per `malloc`, `calloc` (or other heap allocation)
- Number `malloc`s should equal number of `free`s
  - If # `malloc`s > # `free`s, you've leaked memory
  - If # `malloc`s < # `free`s, this can also cause issues

Some basic guard rails

- Don’t return a pointer to a callee stack-allocated variable
  - It may be deallocated when we leave the active frame’s scope
- Remember that malloc/calloc is the way to point to a block in the heap
- Don’t dereference a freed block
- Don’t dereference uninitialized values
- On your own: introduce yourself to `valgrind` for debugging memory leaks
  - Example Tutorial 0, Tutorial 1, Tutorial 2 (no suggested order)
- `gdb` will become a steadfast friend
  - Here is an example tutorial, but project instructions also give more details and links

gdb, makefile demo

Compilation Details: Anatomy of a Makefile

- Declaration of shortcuts, compiler options, *making your life easier!*
- How to generate executables with a mix of sources (e.g., `*.c`, `*.o`)?

- Definition of ***targets***
  - Compiling
  - Linking
- Additional targets to make life easier
  - Group them with something like `all`
  - Scrape old executables, keep it `clean`

```
CC = gcc $(CFLAGS)
CFLAGS = -g -O2 -Wall -Wextra -std=gnu99

all: first checkinA part1 part3 test_mm

first_mm:
	$(CC) -o test_mm test_mm.c

first:
	$(CC) -o first first_program.c

checkinA:
	$(CC) -o checkinA checkin_partA.c

part1:
	$(CC) -o part1 part1.c

part3:
	$(CC) -o part3 part3.c

clean:
	rm first checkinA part1 part3 test_mm
```

Casting Rules

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
    uint16_t value = 0xFFFF;
    printf("Original Value: %u\n", value);
    printf("Unsigned 16->Signed 16: %hd\n", (int16_t)value);
    printf("Unsinged 16->Unsigned 32: %u\n", (unsigned int)value);
    printf("Unsigned 16->Unsigned 8: %hu\n", (uint8_t)value);
    printf("Unsigned 16->Signed 32: %d\n", (int)value);
    int16_t svalue = -1;
    printf("Original Value: %d\n", svalue);
    printf("Signed 16->Unsigned 16: %u\n", (uint16_t)svalue);
    printf("Signed 16->Signed 32: %d\n", svalue);
    printf("Signed 16->Signed 8: %d\n", (uint8_t)svalue);
    printf("Signed 16->Unsigned 32: %u\n", (unsigned int)svalue);
}
```

- ***Yes***, you can cast between types, but there are rules:
- Signed <-> unsigned integers: the bits don't change, they are reinterpreted
- Integer → double (or larger type): preserves value (with padding)
- Casting down (e.g., int → char) will ***truncate the data***
- ***Always be explicit***
  - `int x = (int) y;`
- ***Pointers have a type, and can be cast too***
  - Highly common bug is dereferencing a pointer as an **incorrect type** – this will cause an undefined memory access, possibly a *segmentation fault* – more on that later

Other C variable declarations

- Global variables
  - Declare a variable outside a function in one file
  - That variable can now be seen/accessed by other files via `extern`

```
// in exampleA.c

int a = 7; //global variable
void incrementA()
{
  a++;
  printf("%d", a);
}
```

```
//Some other file
#include "exampleA.c";
main()
{
  extern int a;
  a++;
  incrementA();
}
```

Other C variable declarations

- Global variables
  - Declare a variable outside a function in one file
  - That variable can now be seen/accessed by other files via `extern`
- VS `#define` outside of functions, for constants
  - A first example of something to separate into a *header* file, not the source files

```
// in exampleA.h
#define NAME “MyVariable”
```

```
//Some other file
#include “exampleA.h”;
main()
{
  printf(“About to change %s”,
NAME);
}
```

Other C variable declarations

- `static` variables *(distinct from a static function)*
  - Local variables ***retaining*** value between invocations to a given function

```
// in exampleB.c

int incrementA()
{
  static int a = 0;
  a++;
  return a;
}
```

```
// still in exampleB.c

int main()
{
   printf("%d ", incrementA());
   printf("%d ", incrementA());
   return 0;
}
```

What is the printed output of this program?

1 2

Other C variable declarations

- Strings
  - An array of characters, terminated by the null character (`\0`, as in Project 1 details)
  - All standard C library functions on strings assume null-terminated character arrays
- Arrays
  - Can allocate a block of memory on the `stack` or on the `heap`

```
1.  int A[10]; // allocates 10 int on stack
2.  int* A = calloc(10, sizeof(int)); // allocates 1 pointer on the stack,
    with value referencing block of 10 int on heap
```

C is a ***call-by-value*** language

- ***Call-by-value:*** Changes made to arguments passed to a function aren't reflected in the calling function
- ***Call-by-reference:*** Changes made to arguments passed to a function are reflected in the calling function

- To cause changes to values outside the function, pass *pointers* as args
  - The value of the pointer should not change (that won't be reflected!)
  - **The pointer can/should be *dereferenced* and assign a value to the resulting address**

Wrap-Up November 7

- Coming up next!
  - MORE Practice with the stack and the heap, allocating memory and building up to “real” programs in C syntax, studying the corresponding assembly
- Logistics, Reminders
  - TA help 7-9PM on Sundays, Tuesdays, Thursdays in C107
  - LP Office hours M 9-10:30AM, Th 2:30-4PM
  - Weekly Exercises due Friday 5PM
  - Lab 6 due **Tuesday 10PM** (because I’m posting Part 2/3 a little later than expected, right after class today)
- FEEDBACK
  - https://forms.gle/5Aafcm3iJthX78jx6