CS 33

Introduction to Computer Systems
Welcome!

- Prof: Tom Doeppner
- HTAs: William Flotte, Alex Fratila, Jonathan Lister, Emily Magavern
- UTAs: Nicholas Anthony, Andrea Bennett, Jackson Chaiken, Tim Chang, Jared Cohen, Russell Dodd, Michael Gillett, Christopher Harvie, Arohi Kapoor, Caleb Kim, Stephen Leung, Natalie Lindsay, Yuyang Lu, Manav Kohli, Jacob Meltzer, Beatriz Mora, Benjamin Navetta, Lan Nguyen, Silei Ren, Sophie Saskin, Ankita Sharma, Emma Sloan, Misha Sohan, Katie Ta, Charles Tan, Brendan Walsh, Anna White, Amy Winkler, Michael Xu
What You’ll Learn

• Programming in C
• Data representation
• Programming in x86 assembler language
• High-level computer architecture
• Optimizing programs
• Linking and libraries
• Basic OS functionality
• Memory management
• Network programming (Sockets)
• Multithreaded programming (POSIX threads)
Prerequisites:
What You Need to Know

• Ability to program in an object-oriented or procedural language (e.g., Java)
  – CS15 or CS18
What You’ll Do

• Twelve 2-hour labs
• Nine one- to two-week programming assignments
  – most will be doable on OSX as well as on SunLab machines
• No exams!
• Clickers used in class
  – not anonymous: a small portion of your grade
  – full credit (A) for each correct answer
  – partial credit (B) for each wrong answer
  – NC for not answering
  – one to three or so questions per class
Grad Students

• You’re welcome to take the class!
• Weekly homeworks, just for you
  – 10% of your grade
  – ugrads encouraged to try them, but we’ll grade only those of grad students
Gear-Up Sessions

• Optional weekly sessions
  – handle questions about the week’s assignment and course material
  – generally Thursdays, 7pm – 8:30pm
    » the first will be Monday, September 11
  – generally BH 166
Collaboration Policy

• Learn by doing
• You may:
  – discuss the requirements with others
  – discuss the high-level approach with others
• Write your own code
• Debug your own code
• Get stuck
  – others may help you find bugs
  – may not give you solutions or test cases
• Acknowledge (in README) those who assist you
• We run MOSS on all relevant assignments
  – your MOSS score will be supplied with your grade
Textbook

If Programming Languages Were Cars …

• Java would be an SUV
  – automatic transmission
  – stay-in-lane technology
  – GPS navigation
  – traction control
  – gets you where you want to go
    » safe
    » boring

• Racket would be a Tesla
  – you drive it like an SUV
    » definitely cooler
    » but limited range
If Programming Languages Were Cars …

• C would be a sports car
  – manual everything
  – dangerous
  – fun
  – you really need to know what you’re doing!
U-Turn Algorithm
(Java and Racket Version)

1. Switch on turn signal
2. Slow down to less than 3 mph
3. Check for oncoming traffic
4. Press the accelerator lightly while turning the steering wheel pretty far in the direction you want to turn
5. Lift your foot off the accelerator and coast through the turn; press accelerator lightly as needed
6. Enter your new lane and begin driving
U-Turn Algorithm
(C Version)

1. Enter turn at 30 mph in second gear
2. Position left hand on steering wheel so you can quickly turn it one full circle
3. Ease off accelerator; fully depress clutch
4. Quickly turn steering wheel either left or right as far as possible
5. A split second after starting turn, pull hard on handbrake, locking rear wheels
6. As car (rapidly) rotates, restore steering wheel to straight-ahead position
7. When car has completed 180° turn, release handbrake and clutch, fully depress accelerator
History of C

• Early 1960s: CPL (Combined Programming Language)
  – developed at Cambridge University and University of London

• 1966: BCPL (Basic CPL): simplified CPL
  – intended for systems programming

• 1969: B: simplified BCPL (stripped down so its compiler would run on minicomputer)
  – used to implement earliest Unix

• Early 1970s: C: expanded from B
  – motivation: they wanted to play “Space Travel” on minicomputer
  – used to implement all subsequent Unix OSes
More History of C

• 1978: Textbook by Brian Kernighan and Dennis Ritchie (K&R), 1st edition, published
  – de facto standard for the language

• 1989: ANSI C specification (ANSI C)

• 1990: ISO C specification (C90)
  – essentially ANSI C

• 1999: Revised ISO C specification (C99)

• 2011: Further revised ISO C specification (C11)
  – not widely used
CS 33

Introduction to C
A C Program

```c
int main( ) {
    printf("Hello world!\n");
    return 0;
}
```
Compiling and Running It

$ ls
hello.c
$ gcc hello.c
$ ls
a.out hello.c
$ ./a.out
Hello world!
$ gcc -o hello hello.c
$ ls
a.out hello hello.c
$ ./hello
Hello world!
$
What’s gcc?

- gnu C compiler
  - it’s actually a two-part script
  - part one compiles files containing programs written in C (and certain other languages) into binary machine code (known as object code)
  - part two takes the just-compiled object code and combines it with other object code from libraries to create an executable
  - the executable can be loaded into memory and run by the computer
gcc Flags

- gcc [-Wall] [-g] [-std=gnu99]
  - -Wall
    » provide warnings about pretty much everything that might conceivably be objectionable
  - -g
    » provide extra information in the object code, so that gdb (gnu debugger) can provide more informative debugging info
      • discussed in lab
  - -std=gnu99
    » use the 1999 version of C syntax, rather than the 1990 version
Declarations in C

```c
int main() {
    int i;
    float f;
    char c;
    return 0;
}
```

Types are promises
- promises can be broken

Types specify memory sizes
- cannot be broken
Declarations in C

```c
int main() {
    int i;
    float f;
    char c;
    return 0;
}
```

Declarations reserve memory space
- where?

Local variables can be uninitialized
- junk
- whatever was there before
Declarations in C

```c
int main() {
    int i;
    float f;
    char c;
    return 0;
}
```

```
i 1435097815
f 6.1734e-23
c b
```
Using Variables

```c
int main() {
    int i;
    float f;
    char c;
    i = 34;
    c = 'a';
}
```
printf Again

```c
int main() {
    int i;
    float f;
    char c;
    i = 34;
    c = 'a';
    printf("%d\n", i);
    printf("%d%c\n", i, c);
}
```

$ ./a.out
34
34 a
printf Again

```c
int main() {
    ...
    printf("%d\t%c\n",i,c);
}
```

Two parts
- formatting instructions
- arguments
printf Again

```c
int main() {
    ...
    printf("%d\t%c\n", i, c);
}
```

$ ./a.out
34 a

Formatting instructions

- Special characters
  - \n : newline
  - \t : tab
  - \b : backspace
  - \" : double quote
  - \\ : backslash
printf Again

```c
int main() {
    ...
    printf("%d\t%c", i, c);
}
```

$ ./a.out
34   a

Formatting instructions

• Types of arguments
  – %d: integers
  – %f: floating-point numbers
  – %c: characters
printf Again

```c
int main() {
    ...
    printf("%6d%3c", i, c);
}
```

$ ./a.out
34 a

Formatting instructions
- `%6d`: decimal integer at least 6 characters wide
- `%6f`: floating point at least 6 characters wide
- `%6.2f`: floating point at least 6 wide, 2 after the decimal point
printf Again

```c
int main() {
    int i;
    float celsius;
    for (i=30; i<34; i++) {
        celsius = (5.0/9.0)*(i-32.0);
        printf("%3d %6.1f\n", i, celsius);
    }
}
```

```
$ ./a.out
 30  -1.1
 31  -0.6
 32   0.0
 33   0.6
```
For Loops

```c
int main() {
    int i;
    float celsius;
    for (i=30 ; i<34 ; i=i+1) {
        celsius = (5.0/9.0)*(i-32.0);
        printf("%3d %6.1f\n", i, celsius);
    }
}
```
Some Primitive Data Types

char
   – a single byte: interpreted as either an 8-bit integer or a character

short
   – integer: 16 bits

int
   – integer: 16 bits or 32 bits (implementation dependent)

long
   – integer: either 32 bits or 64 bits, depending on the architecture

long long
   – integer: 64 bits

float
   – single-precision floating point

double
   – double-precision floating point
What is the size of my int?

```c
int main() {
    int i;
    printf("%d\n", sizeof(i));
}
```

`sizeof` – returns the size of a variable in bytes

– very very very very very very very very important function in C
int main() {
    int a[100];
    int i;
}

Arrays
Arrays

```c
int main() {
    int a[100];
    int i;
    for (i=0; i<100; i++)
        a[i] = i;
}
```
Array Bounds

```c
int main() {
    int a[100];
    int i;
    for (i=0; i<=100; i++)
        a[i] = i;
}
```
Arrays in C

C Arrays = Storage + Indexing
- no bounds checking
- no initialization

Welcome to the jungle
Welcome to the Jungle

```
int main() {
    int j = 8;
    int a[100];
    int i;
    for (i = 0; i <= 100; i++)
        a[i] = i;
    printf("%d\n", j);
}
```

```
$ ./a.out
8
```
Quiz 1

• What is printed for the value of j when the program is run?
  a) 0
  b) 8
  c) 100
  d) indeterminate
Welcome to the Jungle

```c
int main() {
    int j=8;
    int a[100];
    int i;
    for(i=0;i<=100;i++)
        a[i] = i;
    printf("%d\n", j);
}
```

```bash
$ ./a.out
100
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[2]</td>
<td>2</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>a[99]</td>
<td>99</td>
</tr>
<tr>
<td>j</td>
<td>100</td>
</tr>
</tbody>
</table>
```
Welcome to the Jungle

```c
int main() {
    int j;
    int a[100];
    int i;
    for (i=0; i<100; i++)
        a[i] = i;
    printf("%d\n", j);
}
```

```bash
$ ./a.out
???
```
Quiz 2

• What is printed for the value of j when the program is run?
  a) 0
  b) 8
  c) 100
  d) indeterminate
int main() {
    int j;
    int a[100];
    int i;
    for (i=0; i<100; i++)
        a[i] = i;
    printf("%d\n", j);
}

$ ./a.out
-1880816380
Welcome to the Jungle

```c
int main() {
    int a[100];
    int i;
    a[-3] = 25;
    printf("%d\n", a[-3]);
}
```

```
$ ./a.out
25
```
Welcome to the Jungle

```c
int main() {
    int a[100];
    int i;
    a[-3] = 25;
    a[11111111] = 6;
    printf("%d\n", a[-3]);
}
```

$ ./a.out
Segmentation fault

What is a segmentation fault?
- attempted access to an invalid memory location
Function Definitions

```c
int fact(int i) {
    int k;
    int res;
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}

int main() {
    printf("%f\n", fact(5));
    return 0;
}
```

- **main**
  - is just another function
  - starts the program

- All functions
  - have a return type
Compiling It

```
$ gcc -o fact fact.c
$ ./fact
120
```
Function Definitions

```c
int main() {
    printf("%f\n", fact(5));
    return 0;
}

float fact(int i) {
    int k;
    float res;
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}
```
Function Definitions

$ gcc -o fact fact.c
main.c:27: warning: type mismatch with previous implicit declaration
main.c:23: warning: previous implicit declaration of 'fact'
main.c:27: warning: 'fact' was previously implicitly declared to return 'int'

$ ./fact
1079902208
Function Declarations

float fact(int i);

int main() {
    printf("%f\n", fact(5));
    return 0;
}

float fact(int i) {
    int k;
    float res;
    for(res=0,k=1; k<=i; k++)
        res = res * k;
    return res;
}

$ ./fact
120.000000
Methods

- C has functions
- Java has methods
  - methods implicitly refer to objects
  - C doesn’t have objects
- Don’t use the “M” word
  - TAs will laugh at you
Function Declarations

```c
#include "fact.h"

int main() {
    printf("%f\n", fact(5));
    return 0;
}

float fact(int i) {
    int k; float res;
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}

float fact(int i);
```

```
#include "fact.h"

int main() {
    printf("%f\n", fact(5));
    return 0;
}

float fact(int i) {
    int k; float res;
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}
```
The Preprocessor

#include

• calls the preprocessor to include a file

What do you include?

• your own header file:
  #include "fact.h"
  – look in the current directory

• standard header file:
  #include <assert.h>
  #include <stdio.h>
  – look in a standard place

Contains declaration of printf (and other things)
#define

```c
#define SIZE 100
int main() {
    int i;
    int a[SIZE];
}
```

- defines a substitution
- applied to the program by the preprocessor
#define

```
#define forever for(;;)
int main() {
  int i;
  forever {
    printf("hello world\n");
  }
}
```
assert

- verify that the assertion holds
- abort if not

```c
#include <assert.h>
float fact(int i) {
    int k; float res;
    assert(i >= 0);
    for(res=1,k=1; k<=i; k++)
        res = res * k;
    return res;
}
int main() {
    printf("%f\n", fact(-1));
}
```

$ ./fact
main.c:4: failed assertion 'i >= 0'
Abort
Parameter passing

Passing arrays to a function

```c
int average(int a[], int size) {
    int i; int sum;
    for (i=0, sum=0; i<size; i++)
        sum += a[i];
    return sum/s;
}
int main() {
    int a[100];
    ...
    printf("%d\n", average(a,100));
}
```

- Note that I need to pass the size of the array
- This array has no idea how big it is
Swapping

Write a function to swap two entries of an array

```c
void swap(int a[], int i, int j) {
    int tmp;
    tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```
Selection Sort

```c
void selectsort(int array[], int length)
{
    int i, j, min;
    for (i = 0; i < length; ++i)
    { /* find the index of the smallest item from i onward */
        min = i;
        for (j = i; j < length; ++j)
        { /* swap the smallest item with the i-th item */
            if (array[j] < array[min])
                min = j;
        }
    }
    /* at the end of each iteration, the first i slots have the i smallest items */
}
```
Swapping

Write a function to swap two ints

```c
void swap(int i, int j) {
}

int main() {
    int a = 4;
    int b = 8;
    swap(a, b);
    printf("a:%d  b:%d", a, b);
}
```

Parameters are passed by value
Swapping

Write a function to swap two ints

```c
void swap(int i, int j) {
    int tmp;
    tmp = j; j = i; i = tmp;
}

int main() {
    int a = 4;
    int b = 8;
    swap(a, b);
    printf("a:%d  b:%d", a, b);
}
```

```
$ ./a.out
a:4  b:8
```