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
  - Undergrads 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
Some of this lecture is based on material prepared by Pascal Van Hentenryck.
Following K&R, this is everyone’s first C program. Note that C programs start in a procedure called `main`, which is a function returning an integer. This integer is interpreted as an error code, where 0 means no errors and anything else is some sort of indication of a problem. We’ll see later how we can pass arguments to main.

```c
int main( ) {
    printf("Hello world!\n");
    return 0;
}
```
gcc (the Gnu C compiler), as do other C compilers, calls its output “a.out” by default. (This is supposed to mean the output of the assembler, since the original C compilers compiled into assembly language, which then had to be sent to the assembler.) To give the output of the C compiler, i.e., the executable, a more reasonable name, use the “-o” option.

```
$ 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 gnu? It’s a project of the Free Software Foundation and stands for “gnu’s not Unix.” That it’s not Unix was pretty important when the gnu work was started in the 80s. At the time, AT&T was the owner of the Unix trademark and was very touchy about it. Today the trademark is owned by The Open Group, who is less touchy about it.
The use of the –Wall flag will probably produce lots of warning messages about things you had no idea might possibly be considered objectionable. Unless you’re really concerned about getting the last ounce of performance from your program, it’s a good idea always to use the –g flag.

Most of what we will be doing is according to the C90 specification. The C99 specification cleaned a few things up and added a few features. There’s also a C11 (2011) specification that is not yet widely used.
Declarations in 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
int main() {
    int i;
    float f;
    char c;
    return 0;
}
Using Variables

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

```
$ ./a.out
34 a
```
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
```c
int main() {
    ...
    printf("%6d%3c", i, c);
}
```

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

$ ./a.out
   34 a
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
```
Note that the “should loop continue” test is done at the beginning of each execution of the loop. Thus, if in the slide the test were “i<30”, there would be no executions of the body of the loop and nothing would be printed.
The sizes of integers depend on the underlying architecture. In the earliest versions of C, the `int` type had a size equal to that of pointers on the machine. However, the current definitions of C apply this rule to the `long` type. The `int` type has a size of 32 bits on pretty much all of today’s computers.

For the sunlab computers, a `long` is 64 bits.
What is the size of my int?

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

```
$ ./a.out
4
```

`sizeof`
- returns the size of a variable in bytes
- very very very very very very important function in C

Note that the argument to `sizeof` need not be a variable, but could be the name of a type. For example, “`sizeof(int)`” is legal and returns 4 on most machines.
The array $a$ and the variable $i$ really are arranged in memory as shown, assuming that “higher” memory addresses are at the bottom of the diagram and “lower” memory addresses are at the top. We draw it this way because this is how one normally draws pictures of memory. However, later on in the course we will reverse this and arrange our memory diagrams so that higher addresses are at the top and lower addresses are at the bottom.
After executing the program, memory should contain what’s shown in the diagram.

```c
int main() {
    int a[100];
    int i;
    for(i=0;i<100;i++)
        a[i] = i;
}
```
Here the for loop goes one element too far, storing 100 into a[100], despite the fact that we didn’t declare the array to be that large.
Arrays in C

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

Welcome to the jungle
Note how `j` is both declared and initialized in the same statement.
Quiz 1

- What is printed for the value of j when the program is run?
  
a) 0
b) 8
c) 100
d) indeterminate
Note that \( j \) occupies memory where \( a[100] \) would be if \( a \) were declared to be that large. Thus \( j \)'s location is overwritten when the program goes beyond the bounds for \( a \).
Note that \( j \) no longer has an initial value. Note also that the for loop ends with after setting \( a[99] \) to 99.
Quiz 2

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

```
$ ./a.out
-1880816380
```

<table>
<thead>
<tr>
<th>i</th>
<th>100</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>-1880816380</td>
</tr>
</tbody>
</table>

CS33 Intro to Computer Systems  1-44  Copyright © 2017 Thomas W. Doeppner. All rights reserved.
Welcome to the Jungle

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

```
$ ./a.out
25
```

Note that this code is not guaranteed to work!
Sometimes the error message is “bus error.” Both terms (segmentation fault and bus error) come from the original C/Unix implementation on the PDP-11. A segmentation fault resulted from accessing memory that might exist, but for which the accessor has no permission. A bus error results from trying to use an address that makes no sense.

```
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
Note the use of the comma in the initialization part of the for loop: the initialization part may have multiple parts separated by commas, each executed in turn.
Compiling It

```
$ gcc -o fact fact.c
$ ./fact
120
```
Note that not only has the definition of `main` been placed before the definition of `fact`, but also that `fact` has been changed so that it now returns a `float` rather than an `int`. 

```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;
}
```
If a function, such as `fact`, is encountered by the compiler before it has encountered a declaration or definition for it, the compiler assumes that the function returns an `int`. This rather arbitrary decision is part of the language for “backwards-compatibility” reasons — so that programs written in older versions of C still compile on newer (post-1988) compilers.
Here we have a declaration of `fact` before its definition. (If the two are different, 
gcc will complain.)
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

```c
for (;;) printf("C does not have methods!\n");
```
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;
}
```
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)

The rules for the distinction between using double quotes and angle brackets are a bit vague. What's shown here is common practice, which should be the rule.

Note that the preprocessor directives (such as `#include`) must start in the first column.

Note that one must include `stdio.h` if using `printf` (and some other routines) in a program.

On most Unix systems (including Linux, but not OS X), the standard place for header files is the directory `/usr/include`. 
#define

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

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

#define forever for(;;)
int main() {
    int i;
    forever {
        printf("hello world\n");
    }
}
The assert statement is actually implemented as a macro (using #define). One can “turn off” asserts by defining (using #define) NDEBUG. For example,

```
#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));
```

In this case, the assert will not be executed, since NDEBUG is defined. Note that one also can define items such as NDEBUG on the command line for gcc using the –D flag. For example,

```
gcc –o prog prog.c –DNDEBUG
```

Has the same effect as having “#define NDEBUG” as the first line of prog.c.
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;
}
```
Note that C uses the same syntax as Java does for conditional (if) statements. In addition to relational operators such as "==", "!=", "<", ">", "<=", and ">=", there are the conditional operators "&&" and "||" ("logical and" and "logical or", respectively).
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