CSE 333
Lecture 2 - gentle re-introduction to C

Steve Gribble
Department of Computer Science & Engineering
University of Washington
**HW0 results**

<table>
<thead>
<tr>
<th>question</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have programmed in C before</td>
<td>89%</td>
<td>11%</td>
</tr>
<tr>
<td>I have programmed in C++ before</td>
<td>26%</td>
<td>74%</td>
</tr>
<tr>
<td><strong>languages:</strong> <a href="100%25">Java</a>, <a href="9%25">Python</a>, <a href="4%25">x86</a>, <a href="5%25">C#</a>, <a href="0%25">Ruby</a>, <a href="10%25">JavaScript</a>, <a href="15%25">PHP</a>, <a href="0%25">Pascal</a>, <a href="0%25">Haskell</a>, <a href="2%25">visual basic</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I am taking 332 right now</td>
<td>95%</td>
<td>5%</td>
</tr>
<tr>
<td>I know what a hash table is</td>
<td>100%</td>
<td>0%</td>
</tr>
</tbody>
</table>
## HW0 results

<table>
<thead>
<tr>
<th>question</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>I know what a hash table is</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>I have implemented a hash table</td>
<td>78%</td>
<td>22%</td>
</tr>
<tr>
<td>I know what a C pointer is</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>I have debugged pointer bugs</td>
<td>65%</td>
<td>35%</td>
</tr>
<tr>
<td>I know what ((*(x+5))[5] = &amp;y;) means</td>
<td>54%</td>
<td>46%</td>
</tr>
</tbody>
</table>
unsigned char *mystery_function(unsigned short bufsize) {
    unsigned char *tmp_buf;

    if (bufsize == 0)
        return NULL;
    tmp_buf = malloc(bufsize);
    if (tmp_buf == NULL)
        return NULL;
    if (verify_something() == 0) // something bad happened
        return NULL;
    return tmp_buf;
}
# HW0 results

<table>
<thead>
<tr>
<th>question</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>spot the bug</strong>: I don’t know 50%, 25% memory leak, 10% type error, 5% multiple of 4 issue, 10% other</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Linux</strong>: 0% never, 85% &lt; 1 year, 13% years, 2% expert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know what an inode is</td>
<td>7%</td>
<td>93%</td>
</tr>
<tr>
<td>I know what a socket is</td>
<td>17%</td>
<td>83%</td>
</tr>
<tr>
<td>I’ve written multithreaded code</td>
<td>89%</td>
<td>11%</td>
</tr>
</tbody>
</table>
## HW0 results

**what is the air-speed velocity of an unladen swallow?**

<table>
<thead>
<tr>
<th>Option</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>African or European?</td>
<td>48%</td>
</tr>
<tr>
<td>24 miles an hour</td>
<td>16%</td>
</tr>
<tr>
<td>8-11 m/s</td>
<td>30%</td>
</tr>
<tr>
<td><strong>it depends</strong></td>
<td>8%</td>
</tr>
<tr>
<td>500</td>
<td>1%</td>
</tr>
<tr>
<td>Red.  No, blue.  AHHHHH....</td>
<td>1%</td>
</tr>
</tbody>
</table>
Today’s goals:

- overview of the C material you learned from cse351

Next two weeks’ goals:

- dive in deep into more advanced C topics
- start writing some C code
- introduce you to interacting with the OS
Attribution

The slides I’ll be using are a mixture of:

- my own material

- slides from other UW CSE courses (CSE303, CSE351; thanks Magda Balazinska, Marty Stepp, John Zahorjan, Hal Perkins, and others!!)

- material from other universities’ courses (particularly CMU’s 15-213 and some Harvard courses; thanks Randy Bryant, Dave O’Hallaron, Matt Welsh, and others!!)

All mistakes are mine. (No, really.)
C

Created in 1972 by Dennis Ritchie

- designed for creating system software
- portable across machine architectures
- most recently updated in 1999 (C99)

Characteristics

- low-level, smaller standard library than Java
- procedural (not object-oriented)
- typed but unsafe; incorrect programs can fail spectacularly
C

Created in 1972 by Dennis Ritchie

- designed for creating system software
- portable across machine architectures
- most recently updated in 1999 (C99)
- low-level, smaller standard library than Java
- procedural (not object-oriented)
- typed but unsafe; incorrect programs can fail spectacularly

This book was typeset \texttt{(pic|tbl|eqn|troff -ms)} using an Autologic APS-5 phototypesetter and a DEC VAX 8550 running the 9th Edition of the UNIX operating system.
Mindset of C

“The PDP-11/45 on which our UNIX installation is implemented is a:

- 16-bit word (8-bit byte) computer with
  ▶ 144K bytes of core memory; UNIX occupies 42K bytes
  ▶ a 1M byte fixed-head disk
  ▶ a moving-head disk with 40M byte disk packs
- The greater part of UNIX software is written in C.”

Dennis M. Ritchie and Ken Thompson
Bell Laboratories
1974
C workflow

Editor
(emacs, vi)

or IDE
(eclipse)
C workflow

source files (.c, .h)

Editor (emacs, vi) or IDE (eclipse)

edit

foo.c

foo.h

bar.c
C workflow

- Editor (emacs, vi)
- IDE (eclipse)

- source files (.c, .h)
- foo.c
- foo.h
- bar.c

- libZ.a statically linked libraries
- libc.so shared libraries
C workflow

Editor (emacs, vi) or IDE (eclipse)

- source files (.c, .h)
  - foo.h
  - foo.c
  - bar.c

compile

- object files (.o)
  - foo.o
  - bar.o

- library files
  - libZ.a (statically linked)
  - libc.so (shared)
C workflow

- Source files (.c, .h)
- Object files (.o)
- Executable
- Static libraries
  - libZ.a
- Shared libraries
  - libc.so

Edit (editor: emacs, vi) or IDE (eclipse)

- foo.c
- foo.h
- bar.c
- bar.h
- bar.o
- Compile
- Link
C workflow

Editor (emacs, vi) or IDE (eclipse)

source files (.c, .h)

foo.h

bar.c

edit

compile

foo.c

foo.o

link

bar

link

link

link

load

process

bar

bar

bar

executable

object files (.o)

libZ.a

statically linked libraries

libc.so

shared libraries
C workflow

source files (.c, .h)

object files (.o)

edit

compile

link

load

execute, debug, profile, ...

process

foo.h

foo.c

bar.c

bar.o

libZ.a

libc.so

statically linked libraries

shared libraries

Editor (emacs, vi)

or IDE (eclipse)
From C to machine code

C source file
(dosum.c)

```c
int dosum(int i, int j) {
    return i+j;
}
```
From C to machine code

C source file
(dosum.c)

int dosum(int i, int j) {
    return i+j;
}

assembly source file
(dosum.s)

dosum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
addl 12(%ebp), %eax
popl %ebp
ret
From C to machine code

C source file
(dosum.c)

C compiler (gcc -S)

assembly source file
(dosum.s)

machine code
(dosum.o)

int dosum(int i, int j) {
    return i+j;
}

dosum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    popl %ebp
    ret

80483b0: 55
    89 e5 8b 45
    0c 03 45 08
    5d c3
Skipping assembly language

Most C compilers generate .o files (machine code) directly
- i.e., without actually saving the readable .s assembly file
Multi-file C programs

C source file (dosum.c)

```c
int dosum(int i, int j) {
    return i+j;
}
```

C source file (sumnum.c)

```c
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
    printf("%d\n", dosum(1,2));
    return 0;
}
```
Multi-file C programs

C source file (dosum.c)

```c
#include <stdio.h>

int dosum(int i, int j) {
    return i+j;
}
```

C source file (sumnum.c)

```c
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
    printf("%d\n", dosum(1,2));
    return 0;
}
```

dosum() is implemented in sumnum.c
Multi-file C programs

C source file (dosum.c)

```c
#include <stdio.h>
int dosum(int i, int j);
int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

C source file (sumnum.c)

```c
#include <stdio.h>
int dosum(int i, int j); 
int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

dosum() is implemented in sumnum.c

this “prototype” of dosum() tells gcc about the types of dosum’s arguments and its return value
Multi-file C programs

C source file (dosum.c)

```c
#include <stdio.h>
int dosum(int i, int j); 
int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

C source file (sumnum.c)

```c
int dosum(int i, int j) {
  return i+j;
}
```
Multi-file C programs

C source file
(dosum.c)

```c
#include <stdio.h>
int dosum(int i, int j);

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

C source file
(sumnum.c)

```c
int dosum(int i, int j) {
  return i+j;
}
```

where is the implementation of printf?
Multi-file C programs

C source file (dosum.c)

```c
#include <stdio.h>

int dosum(int i, int j) {
  return i+j;
}
```

C source file (sumnum.c)

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

why do we need this `#include`?

where is the implementation of `printf`?
Compiling multi-file programs

Multiple object files are **linked** to produce an executable

- standard libraries (libc, crt1, ...) are usually also linked in
- a library is just a pre-assembled collection of .o files

```
dosum.c
  `------` gcc -c `------`
  |                        |
  v                        v
dosum.o

sumnum.c
  `------` gcc -c `------`
  |                        |
  v                        v
sumnum.o
```

```
ld (or gcc)
```

```
libraries (e.g., libc)
```

```
sumnum
```
Object files

sumnums, dosum.o are **object files**

- each contains machine code produced by the compiler
- each might contain references to external symbols
  - variables and functions not defined in the associated .c file
  - e.g., sumnum.o contains code that relies on printf() and dosum(), but these are defined in libc.a and dosum.o, respectively
- linking resolves these external symbols while smooshing together object files and libraries
Let’s dive into C itself

Things that are the same as Java

- syntax for statements, control structures, function calls
- types: int, double, char, long, float
- type-casting syntax: float x = (float) 5 / 3;
- expressions, operators, precedence
  + - * / % ++ -- = += -= *= /= %= < <= == != > >= && || !
- scope (local scope is within a set of { } braces)
- comments: /* comment */ // comment
Primitive types in C

integer types
- char, int

floating point
- float, double

modifiers
- short [int]
- long [int, double]
- signed [char, int]
- unsigned [char, int]

<table>
<thead>
<tr>
<th>type</th>
<th>bytes (32 bit)</th>
<th>bytes (64 bit)</th>
<th>32 bit range</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>[0, 255]</td>
<td>%c</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
<td>[-32768, 32767]</td>
<td>%hd</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>2</td>
<td>[0, 65535]</td>
<td>%hu</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>[-2147483648, 2147483647]</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>4</td>
<td>[0, 4294967295]</td>
<td>%u</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
<td>[-2147483648, 2147483647]</td>
<td>%ld</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
<td>[-9223372036854775808, 9223372036854775807]</td>
<td>%lld</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>approx [10^{-38}, 10^{38}]</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>approx [10^{-308}, 10^{308}]</td>
<td>%lf</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
<td>approx [10^{-4932}, 10^{4932}]</td>
<td>%Lf</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>[0, 4294967295]</td>
<td>%p</td>
</tr>
</tbody>
</table>
C99 extended integer types

Solve the conundrum of “how big is a long int?”

```c
#include <stdint.h>

void foo(void) {
    int8_t w;    // exactly 8 bits, signed
    int16_t x;   // exactly 16 bits, signed
    int32_t y;   // exactly 32 bits, signed
    int64_t z;   // exactly 64 bits, signed
    uint8_t w;   // exactly 8 bits, unsigned
    ...etc.
}
```
Similar to Java...

- variables
  - must declare at the start of a function or block  \textit{(changed in C99)}
  - need not be initialized before use \textit{(gcc -Wall will warn)}

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int x, y = 5;  // note x is uninitialized!
  long z = x+y;

  printf("z is '%ld'\n", z);  // what's printed?
  {
    int y = 10;
    printf("y is '%d'\n", y);
  }
  int w = 20;  // ok in c99
  printf("y is '%d', w is '%d'\n", y, w);
  return 0;
}
```
Similar to Java...

**const**

- a qualifier that indicates the variable’s value cannot change
- compiler will issue an **error** if you try to violate this
- why is this qualifier useful?

```c
#include <stdio.h>

int main(int argc, char **argv) {
  const double MAX_GPA = 4.0;
  printf("MAX_GPA: %g\n", MAX_GPA);
  MAX_GPA = 5.0; // illegal!
  return 0;
}
```
Similar to Java...

for loops
- can’t declare variables in the loop header  
(changed in c99)

if/else, while, and do/while loops
- no boolean type  
(changed in c99)
- any type can be used; 0 means false, everything else true

```c
int i;

for (i=0; i<100; i++) {
    if (i % 10 == 0) {
        printf("i: %d\n", i);
    }
}
```
Similar to Java...

parameters / return value

- C always passes arguments by value
- “pointers”
  - lets you pass by reference
  - more on these soon
  - least intuitive part of C
  - very dangerous part of C

```c
void add_pbv(int c) {
    c += 10;
    printf("pbv c: %d\n", c);
}

void add_pbr(int *c) {
    *c += 10;
    printf("pbr *c: %d\n", *c);
}

int main(int argc, char **argv) {
    int x = 1;
    printf("x: %d\n", x);
    add_pbv(x);
    printf("x: %d\n", x);
    add_pbr(&x);
    printf("x: %d\n", x);
    return 0;
}
```
Very different than Java

arrays

- just a bare, contiguous block of memory of the correct size
- an array of 10 ints requires 10 x 4 bytes = 40 bytes of memory
- arrays have no methods, do not know their own length
- C doesn’t stop you from overstepping the end of an array!!
  - many, many security bugs come from this
Very different than Java

strings
  - array of char
  - terminated by the NULL character ‘\0’
  - are not objects, have no methods; string.h has helpful utilities

```
char *x = "hello\n";
```
Very different than Java

errors and exceptions

- C has no exceptions (no try / catch)
- errors are returned as integer error codes from functions
- makes error handling ugly and inelegant

crashes

- if you do something bad, you’ll end up spraying bytes around memory, hopefully causing a “segmentation fault” and crash

objects

- there aren’t any; \texttt{struct} is closest feature (set of fields)
Very different than Java

memory management

- **you** must to worry about this; there is no garbage collector

- local variables are allocated off of the stack
  - freed when you return from the function

- global and static variables are allocated in a data segment
  - are freed when your program exits

- you can allocate memory in the heap segment using `malloc()`
  - you must free malloc’ed memory with `free()`
  - failing to free is a leak, double-freeing is an error (hopefully crash)
Very different than Java

console I/O
- C standard library has portable routines for reading/writing
  ‣ scanf, printf

file I/O
- C standard library has portable routines for reading/writing
  ‣ fopen, fread, fwrite, fclose, etc.
  ‣ does **buffering** by default, is **blocking** by default
- OS provides (less portable) routines
  ‣ we’ll be using these: more control over buffering, blocking
Very different than Java

network I/O

- C standard library has no notion of network I/O
- OS provides (somewhat portable) routines
- lots of complexity lies here
  - errors: network can fail
  - performance: network can be slow
  - concurrency: servers speak to thousands of clients simultaneously
Very different than Java

Libraries you can count on

- C has very few compared to most other languages
- no built-in trees, hash tables, linked lists, sort, etc.
- you have to write many things on your own
  ‣ particularly data structures
  ‣ error prone, tedious, hard to build efficiently and portably
- this is one of the main reasons C is a much less productive language than Java, C++, python, or others
See you on Friday!