CSE 333
Lecture 2 - gentle re-introduction to C

Hal Perkins
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 a laptop I can bring to class / section</td>
<td>88%</td>
<td>12%</td>
</tr>
<tr>
<td>languages I have used:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C (80%) C++ (28%) x86 (64%) ARM (0%) Java (100%) Python (56%) Perl (8%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ruby (24%) JavaScript (64%) Go (0%) Haskell (4%) Romulan (4%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>languages I’m awesome at:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C (4%) C++ (0%) x86 (4%)        ARM (0%) Java (96%) Python (16%) Perl (0%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ruby (12%) JavaScript (20%) Go (0%) Haskell (0%) Romulan (4%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Most code I have written as part of a product is:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-100 lines (4%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100-1000 lines (36%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000-10000 lines (52%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000+ (8%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## HW0 results

<table>
<thead>
<tr>
<th>question</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have debugged pointer errors in my code</td>
<td>60%</td>
<td>40%</td>
</tr>
<tr>
<td>I have debugged memory leaks in my code</td>
<td>20%</td>
<td>80%</td>
</tr>
<tr>
<td>I have written network code</td>
<td>44%</td>
<td>56%</td>
</tr>
<tr>
<td>I have used the file system from my code</td>
<td>64%</td>
<td>36%</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 system call is; I’ve used one</td>
<td>20%</td>
<td>80%</td>
</tr>
<tr>
<td>I can write a Makefile</td>
<td>36%</td>
<td>64%</td>
</tr>
<tr>
<td>I’ve used a revision control system</td>
<td>76%</td>
<td>24%</td>
</tr>
</tbody>
</table>

pick a number between 0 and 9
HW 0 results
HW 0 results

Factor a ridiculously large number:
HW 0 results

Factor a ridiculously large number:

“I know that 2 is one…”
HW 0 results

Factor a ridiculously large number:

“I know that 2 is one…”

“2, I don’t have the time to figure out the rest”
HW 0 results

**Factor a ridiculously large number:**

“I know that 2 is one…”

“2, I don’t have the time to figure out the rest”

“2 and some other numbers”
HW 0 results

Factor a ridiculously large number:

“I know that 2 is one…”

“2, I don’t have the time to figure out the rest”

“2 and some other numbers”

“Looks like a problem for a quantum computer.”
HW 0 results

**Factor a ridiculously large number:**

“I know that 2 is one…”

“2, I don’t have the time to figure out the rest”

“2 and some other numbers”

“Looks like a problem for a quantum computer.”

“Not worth solving, unless Bill Gates’ bank’s PGP key”
HW 0 results

Factor a ridiculously large number:

“I know that 2 is one…”

“2, I don’t have the time to figure out the rest”

“2 and some other numbers”

“Looks like a problem for a quantum computer.”

“Not worth solving, unless Bill Gates’ bank’s PGP key”

“2, 311, 2004800585918905910644911527, why would you ask such a question?”
Administrivia

Bring a laptop to section tomorrow if you can

Yes, there will be a midterm and final

- Summer calendar: “final” is last day of class; we’ll treat it as a 2nd midterm; weigh equally in final grade
  ‣ Could have a 2-part final last two class periods if huge demand

Grading - see course overview (5% “other” is wiggle room)

Office hours: when and where?

- Best times? Worst times? CSE lab? TA room? Big Time?
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:

- Steve Gribble’s CSE 333 + some material of my own
- slides from other UW CSE courses (CSE303, CSE351, CSE374; thanks Magda Balazinska, Marty Stepp, John Zahorjan, Gaetano Borriello 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 2011
  ‣ But 1999 version (C99) is still most common

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 2011
- 1999 version (C99) is still most common

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

This book was typeset \(\text{pic|tbl|eqn|troff -ms}\) 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

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

Source files (.c, .h)

- foo.c
- foo.h
- bar.c
C workflow

Editor (emacs, vi) or IDE (eclipse)

source files (.c, .h)

foo.c
foo.h
bar.c

libZ.a
libc.so

statically linked libraries
shared libraries
C workflow

Editor (emacs, vi) or IDE (eclipse)

- foo.h
- foo.c
- bar.c

compiles to

- foo.o
- bar.o

linked to

- libZ.a
- libc.so

Source files (.c, .h)
Object files (.o)
Statically linked libraries
Shared libraries
C workflow

Editor (emacs, vi) or IDE (eclipse)

source files (.c, .h)

foo.h

foo.c

bar.c

object files (.o)

foo.o

bar.o

compile

link

link

bar

executable

link

bar

statically linked libraries

libZ.a

shared libraries

libc.so
C workflow

Source files (.c, .h)

foo.h

foo.c

Editor
(emacs, vi)

or IDE
(eclipse)

Compile

foo.o

Object files (.o)

bar.c

link

bar.o

link

libZ.a

Executable

bar

link

libc.so

Process

Object files (.o)

link

statically linked libraries

link

shared libraries

CSE333 lec 2 C.1 // 06-20-12 // perkins
C workflow

Editor (emacs, vi) or IDE (eclipse)

Source files (.c, .h)

Object files (.o)

Compile

Edit

Link

Executable

Process

Link

Load

Execute, debug, profile, ...

Link

Static libraries

Object files (.o)

Link

Object files (.o)

Executable

Load

Process

Static libraries

Shared libraries

CSE333 lec 2 C.1 // 06-20-12 // perkins
From C to machine code

C source file
(sumof.c)

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

C source file
(sumof.c)

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

assembly source file
(sumof.s)

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

C source file (sumof.c)

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

C compiler (gcc -S)

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

assembly source file (sumof.s)

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

machine code (sumof.o)
Skipping assembly language

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

sumof.c → gcc -S → sumof.s → as → sumof.o → gcc -c
Multi-file C programs

C source file
(sumof.c)

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

C source file
(sumnum.c)

```c
#include <stdio.h>

int sumof(int i, int j);

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

C source file
(sumof.c)

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

C source file
(sumnum.c)

```c
#include <stdio.h>

int sumof(int i, int j);

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

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

C source file (sumof.c)

```c
#include <stdio.h>

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

C source file (sumnum.c)

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

This “prototype” of `sumof()` tells gcc about the types of `sumof`’s arguments and its return value.

`sumof()` is implemented in `sumof.c`.
Multi-file C programs

C source file (sumof.c)

```c
#include <stdio.h>
int sumof(int i, int j) {
    return i+j;
}
```

C source file (sumnum.c)

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

C source file (sumof.c)

```c
#include <stdio.h>
int sumof(int i, int j) {
    return i+j;
}
```

C source file (sumnum.c)

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

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

C source file (sumof.c)

```c
#include <stdio.h>
int sumof(int i, int j) {
    return i+j;
}
```

why do we need this `#include`?

C source file (sumnum.c)

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

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
Object files

sumnum.o, sumof.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 sumof(), but these are defined in libc.a and sumof.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>

see sizeofs.c
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 a;  // exactly 8 bits, unsigned
    ...etc.
}
```
Similar to Java...

- variables
  - C99: don’t have to declare at start of a function or block
  - need not be initialized before use  (gcc -Wall will warn)

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int x, y = 5;   // note x is uninitializated!
  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;
}
```

varscope.c
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;
}
```

consty.c
Similar to Java...

for loops
- C99: can declare variables in the loop header

if/else, while, and do/while loops
- C99: `bool` type supported, with `#include <stdbool.h>`
- 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 $10 \times 4$ 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

```c
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; struct is closest feature (set of fields)
Very different than Java

memory management

- **you** must worry about this; there is no garbage collector
- local variables are allocated on 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!