CSC209 Summer 2015 — Software Tools and Systems Programming

www.cdf.toronto.edu/~csc209h/summer/

Week 2 — May 21, 2015

Peter McCormick
pdm@cs.toronto.edu

Some materials courtesy of Karen Reid
# Labs

<table>
<thead>
<tr>
<th>Last Name</th>
<th>Room</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-H</td>
<td>BA2270</td>
<td>Daniel Kats</td>
</tr>
<tr>
<td>I-M</td>
<td>BA2240</td>
<td>Alexey Khrabrov</td>
</tr>
<tr>
<td>N-Z</td>
<td>BA2220</td>
<td>Michael Chiu Pan Zhang</td>
</tr>
</tbody>
</table>
Asking for help

“It doesn’t work”

“How do I do XYZ?”

“I get an error on line 10”
Asking for help

“I tried X
and expected Y
but got Z instead.
Please help!”
Asking for help

I tried the following code but instead of printing 3.1415 which I expected, it instead prints 3.000000. What am I doing wrong? Please help!

```c
#include <stdio.h>
int main() {
    int pi = 3.1415;
    printf("%f\n", (double) pi);
    return 0;
}
```
Agenda

- Introduction to the C language
- The memory model of the machine
The C Problem Language

- C is a high-level language — structured
  - Supports functions, records and some forms of code modularity
  - Not as high-level as Python or Java

- C is a low-level language — machine level access

- C is a small language — relatively simple syntax, with libraries for extensibility

- C does not hold your hand — it assumes that you know what you’re and how you want to do it
The C Problem Language

- **Good:**
  - Efficient
  - Powerful
  - Portable
  - Flexible

- **Bad:**
  - Easy to make errors
  - Obfuscation
  - Weak support for modularization
From Java to C
Common Syntax between Java and C (1)

• Distinction between statements and expressions
• Semicolon denotes end of statement
• Whitespace is generally ignored!
• Braces to denote scope:
  
  { statement1;
    statement2; ... }


Common Syntax between Java and C (2)

• Binary Expressions:
  • Comparison: ==, !=, <, <=, >, >=
  • Arithmetic: +, -, /, *, %
  • Boolean logical: && (and), || (or)
  • Bitwise logical: & (and), | (or), ^ (xor)
  • Bitwise shift: << (left), >> (right)
Common Syntax between Java and C (3)

• Unary expressions:
  • Minus: -
  • Logical negation: ! (not)
  • Bitwise negation/flip: ~ (not)
  • Pre- and post- increment and decrement (with side effect): ++, --
  • Ternary conditional: ?: 
Common Syntax between Java and C (4)

• More expressions:
  
  • Assignment: =

  • Operator assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

• Statement

  • return expression (or just return if void return type)

• Declarations:

  • int variable;

  • short var1, var2;

  • double array[10];
Common Syntax between Java and C (5)

• Loops
  • for, while, do-while
  • break and continue statements
• if and if-else
• switch (with case and default)
Compiling C

$ gcc -Wall -g -o hello hello.c

-Wall
Include all warnings. Helps you prevent errors.

-g
Include debugging symbols. Allows you to debug with gdb

-o hello
Produce an executable called hello

hello.c
The list of source files to compile.
main function

- Entry point for all programs
- Each shell argument is passed in as a string
- Standard signature: (not quite true)

```c
int main(int argc, char *argv[])
```

- Returns an exit status: non-0 indicates an error occurred, otherwise 0 for success
C data types

• Basic types and literals (King: Ch 7)

```c
int i = 38;  long el = 38L;
int hex = 0x2a;  int oct = 033;
printf("i = %d, el = %ld, hex = %d, oct = %d\n", 
i, el, hex, oct);

i = 38, el = 38, hex = 42, oct = 27

double d1 = 0.3;  double d2 = 3.0;
double d3 = 6.02e23;
printf("d1 = %f, d2 = %f, d3 = %e\n", d1, d2, d3)

d1 = 0.300000, d2 = 3.000000, d3 = 6.020000e+23
```
C literals and types

<table>
<thead>
<tr>
<th>Literal</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>38</td>
<td>int</td>
</tr>
<tr>
<td>38L</td>
<td>38</td>
<td>long int</td>
</tr>
<tr>
<td>0x2a (hex)</td>
<td>42</td>
<td>int</td>
</tr>
<tr>
<td>033 (octal)</td>
<td>27</td>
<td>int</td>
</tr>
<tr>
<td>38.0</td>
<td>38.0</td>
<td>double</td>
</tr>
<tr>
<td>38.0f</td>
<td>38.0</td>
<td>float</td>
</tr>
</tbody>
</table>
C data types

- Most things in C are ints:
  - Boolean values are ints
    - 0 means false, nonzero means true
  - characters are ints (ASCII code)
    - 'a' == 97, '\n' == 10, '\033' == 033 == 27
  - enumerations are really ints
- signed vs. unsigned types
- char, int, long, … are just different sizes of integers.
Mixed Mode Arithmetic

double m = 5/6; /* int / int = int */
printf("Result of 5/6 is %f\n", m);
Result of 5/6 is 0.000000

double n = (double)5/6; /* double / int = double */
printf("Result of (double)5/6 is %f\n", n);
Result of (double)5/6 is 0.833333

double o = 5.0/6; /* double / int = double */
printf("Result of 5.0/6 is %f\n", o);
Result of 5.0/6 is 0.833333

int p = 5.0/6; /* double / int = double but then converted to int */
printf("Result of 5.0/6 is %d\n", p);
Result of 5.0/6 is 0
Data Type Conversion

• The expression on the right side is converted to the type of the variable on the left.

```c
char c;
int i = c;    /* c is converted to int */
double d = i; /* i is converted to double */
```

• This is no problem as long as the variable’s type is at least as “wide” as the expression.

```c
char c = 500; /* compiler warning */
int k = d;
printf("c = %c, k = %d\n", c, k);
```

\[c = \phantom{-}, k = 0\]
printf and format strings

- `printf(a_string)` will print the given string
- *Variadic*: `printf` can take a variable number of arguments
- Whether it actually does will depend on special *format strings*:
  - `%d` for signed integers: `printf(“%d + %d = %d\n”, -3, 5, 2)`
  - `%s` for strings: `printf(“Hello %s!\n”, “CSC209”)`
  - `%f` for floating point: `printf(“pi ~ %f\n”, 3.14f)`
  - `%c` for ASCII character: `printf(“C%cC209”, ’S’);`
  - `%%` to print an actual %: `printf(“100%%!\n”)`
- Other modifiers available: look them up with `man 3 printf`
Boolean values in C

• No built-in `bool` type, nor `true` and `false` values!

• 0 is considered to be `false`, anything else is `true`

• if (0) { printf(“Never run\n”); } 

• if (-1) { printf(“Always run\n”); }
Data Type Capacity

• What happens when the following code is executed?

```c
char c = 127;
int d;

printf("c = %d\n", c);
c++;

d = 512 / c;
printf("c = %d, d = %d\n", c, d);
```
Memory Model
Memory Model

• System memory is can be viewed as a sequence of bytes (8 bit values)

• Each location in that sequence (and thus its associated value) is assigned a unique address

• Each address is just a number:

<table>
<thead>
<tr>
<th>1st address</th>
<th>→</th>
<th>1st byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd address</td>
<td>→</td>
<td>2nd byte</td>
</tr>
<tr>
<td>3rd address</td>
<td>→</td>
<td>3rd byte</td>
</tr>
</tbody>
</table>

...
Memory Model

A 32 bit address can give a unique address number to ~4 billion ($2^{32}$) different bytes

4294967296 bytes

~4294967 thousand bytes

~4295 million bytes

~4 billion bytes

aka ~4.29 gigabytes

== 4 gibibytes ($4 \times 2^{30}$)
Memory Model

• A 32-bit system can address, and thus is limited to, a maximum of 4GB of system memory (RAM)

• A 64-bit system has a much higher limit (~16 billion GB worth of unique addresses, less usable in practise)

  • The CDF server *Wolf* is a 64-bit machine (with 64GB of physical RAM)

  • This is indicated by the string “x86_64 “in the output of *uname –m*
Memory Model

• Java and Python hide (shield?) all of this from you

• C does not
Logical Memory M

- Memory is just a sequence of bytes
- A memory location is identified by an address

<table>
<thead>
<tr>
<th>Stack ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused Logical Address Space</td>
</tr>
<tr>
<td>Dynamic Data ↑</td>
</tr>
<tr>
<td>Static Data</td>
</tr>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>

Logical address 0

$2^{64} - 1$
Example

```c
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {
    int i = x;
    y = f(i, i);
    return 0;
}
```
Arrays

• Arrays in C are a contiguous chunk of memory that contain a list of items of the same type.
• If an array of ints contains 10 ints, then the array is 40 bytes. There is nothing extra.
• In particular, the size of the array is not stored with the array. There is *no* runtime checking.
Arrays

int x[5];
for (i = 0; i <= 5; i++) {
   x[i] = i*i;
}

- No runtime checking of array bounds
- Behaviour of exceeding array bounds is “undefined”
  - program might appear to work
  - program might crash
  - program might do something apparently random
Next Week

• Assignment 1 will be posted within the next few days

• Lecture: More on C pointers and memory
<table>
<thead>
<tr>
<th>Last Name</th>
<th>Room</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-H</td>
<td>BA2270</td>
<td>Daniel Kats</td>
</tr>
<tr>
<td>I-M</td>
<td>BA2240</td>
<td>Alexey Khrabrov</td>
</tr>
<tr>
<td>N-Z</td>
<td>BA2220</td>
<td>Michael Chiu</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pan Zhang</td>
</tr>
</tbody>
</table>