Computer Systems Principles

C Programming

UMASS CS
SCHOOL OF COMPUTER SCIENCE
Learning Objectives

• C Programming
  – Understand where C comes from
  – Understand why we use C
  – Understand C compilation
  – Understand the difference between Java and C
  – Learn the structure of C programs
  – Learn and apply C basic input/output
  – Understand basic C example programs
What is C?

C was created for **systems programming**.

C was created to **write Unix**.

Dennis Ritchie
Evolution of C – K&R, ANSI C

• Based on B, a simplified version of BCPL.
  — B lacked byte-addressability.
• K&R (Brian Kernighan & Dennis Richie)
  — Bell Labs
• ANSI C (American National Standards Institute)
  — Departed from Bell Labs C
Evolution of C – ISO C90, ISO C99

- **ISO C90 (International Standards Organization)**
  - Mostly the same as ANSI C in 1990

- **ISO C99 (International Standards Organization)**
  - New data types
  - Support for text strings with characters not found in the English language
Why C in this course?

• Did you ever see the wizard of Oz?
What was going on behind the curtains?
More than what you would think!
The mystery revealed!
So, why C in this course?

• **Closed Curtains (e.g., Java)**
  – A safe place for programmers!
  – Java hides certain aspects of reality
  – This is good!

• **Behind The Curtains Programming (e.g., C)**
  – The Operating System (OS)
  – Access to memory and memory management
  – Dangerous! ... but necessary
  – Important to understand how the real system works
  – Makes you a **better** programmer!
So, what are some differences between C and Java?
Java versus C: Paradigms

- Java and C follow different programming paradigms.
- Java is Object-Oriented while C is Procedural.

Most differences between the features of the two languages arise due to the use of different programming paradigms. C breaks down to Functions while Java breaks down to Objects. C is more procedure-oriented while Java is object-oriented.
Java versus C: Native

• Java is an “interpreted” language while C is a “compiled” language
• Java is compiled to bytecode and that bytecode is “interpreted” by the Java Virtual Machine (JVM).
• C is compiled to machine code that is “executed” by the underlying processor.
Java versus C: High/Low

• Java is a high-level language
• C is a low-level language
  – C is closer to the machine. It gives you access to aspects of the machine not accessible in Java.
  – Types in C are dependent on the machine on which you are running. Java types are the same across all machines.
  – Both arrangements are intentional!
Java versus C: Memory

• **Java does not allow direct access to memory.**
  – Instead you have references to objects in memory. You can pass these references to methods and even re-assign what that reference refers to.

• **C does allow direct access to memory.**
  – In C, you have pointers to data in memory. These allow you to manipulate memory in arbitrary ways, sometimes helpful, sometimes dangerous.
  – You can interpret that data as you wish.
  – You can (and will) shoot yourself in the foot.
Java versus C: Memory Management

• **Java has automatic memory management.**
  – In Java, you can dynamically allocate objects. You use the `new` operator. When you are finished with an object you simply forget about it.

• **C provides manual memory management.**
  – In C, you can dynamically allocate “objects”. You use the `malloc` library function. When you are finished with that “object” you must use the `free` library function. If you do not, you will have a memory leak.*
Java versus C: Exceptions

• **Java provides “exceptions”**.
  – If your program has an error at run-time it will throw an exception. You get a nice “stack” trace.

• **C does not.**
  – If you are lucky (and smart) your program will check “error conditions” and fail gracefully.
  – If you are somewhat lucky your program will crash and simply tell you: *segfault*.
  – Otherwise it will just produce incorrect output, or loop forever, or some other nasty behavior.
So, the point(er) is....?

• Programming Languages Are Tools
  – Java is one language and it does its job well
  – C is another language and it does its job well

• Pick The Right Tool for the Job
  – C is a good language to explore how the system works under-the-hood.
  – It is the right tool for the job we need to accomplish in this course!
GNU Compiler Collection

unix> gcc -std=c99 prog.c
GNU Compiler Collection

unix> gcc -std=c99 prog.c

Can compile programs according to conventions of several different versions of the C programming language.
GNU Compiler Collection

```
unix> gcc -std=c99 prog.c
```

Can compile programs according to conventions of several different versions of the C programming language.

This command line option specifies the specific version of C you wish to compile.
GNU Compiler Collection

unix> gcc -std=c99 prog.c

Linux is built using the GNU compiler collection.

GNU = GNU’s Not Unix

**GOAL**: Develop a complete Unix-like system that is *open source*.

GCC  GDB  EMACS

And many, many other useful tools and utilities for working with and manipulating binary programs
Java Compilation

Foo.java → javac

Source program (text)
Java Compilation

```
Foo.java
javac
Foo.class
```

Binary program
(Java bytecodes)
Java Compilation

Foo.java → javac → Foo.class → java (JVM)

Binary program
(Java bytecodes)
Java Compilation

Dynamically loads and links libraries at run time.
The C pre-processor modifies the original C program according to directives that begin with the ‘#’ character.

```
unix> gcc -E foo.c > foo.i
```
C Compilation

The `cc1` compiler translates the modified C program into “human” readable assembly representing machine instructions.

```bash
unix> gcc -S foo.i
```
C Compilation

The *as* assembler translates the assembly program into machine readable *binary* object files containing relocatable machine instructions.

```
unix> gcc -c foo.s  
unix> objdump -d foo.o
```
The **ld** linker composes the object files into a machine readable *binary* binary file executable for the machine it is compiled for.

```
unix> gcc foo.o -o foo
```
## C Data Types and Sizes

<table>
<thead>
<tr>
<th>C declaration</th>
<th>32-bit (# bytes)</th>
<th>64-bit (# bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Basic Declarations

/* basic declarations & sizes on 64-bit machines */

int x;          /* 4 bytes */
short int x;    /* 2 bytes */
long int x;     /* 8 bytes */
char x;         /* 1 byte */

unsigned int x; /* 4 bytes */
Basic Constants

int x = 32;

short int x = 32;

long int x = 32L;

char x = 'c';

/* Can also specify hexadecimal
   (base-16, octal, binary formats *)
   x = 0x20 /* hex */
   x = 040; /* octal */
   x = 0b1010; /* binary: a gcc thing */
Basic Constants

int x = 32;

short int x = 32;

long int x = 32L;

char x = 'c';

/* Can also specify hexadecimal (base-16, octal, binary formats */
x = 0x20 /* hex */
x = 040; /* octal */
x = 0b1010; /* binary: a gcc thing */

There are also string constants “hello, world”
We will hold off on this for now.
### #define

- **Processed by the preprocessor**
  - Remember: C source to C source

- **Simple way to define “constants”**
  - Advanced usage involves “macros”

```c
#define LETTERCOUNT 26
#define LETTER ‘a’

char foo = LETTER;
int alphabet_length = LETTERCOUNT;
```
Arithmetic Operators

/* binary arithmetic: +,-,*,/,%, */
int x;
x = 32 + 2;
x = 32 - 2;
x = 32 * 2;
x = 32 / 3;  /* truncates fraction part */
x = 32 % 3;  /* produces remainder */

/* unary: +,- */
x = +3;
x = -3;
Relational

/* relational operators:
   >,>=,<,<=,!=,== */

int x = 4;
int x = 4;
x >  3; /* => 1 */
x <  3; /* => 0 */
x >= 3; /* => 1 */
x <= 3; /* => 0 */
x != 3; /* => 1 */
x == 3; /* => 0 */
Logical Connectives

/* relational operators: 
   &&, || */

int x = 4;
x > 3 && x < 3; /* => 0 */
x > 3 || x < 3; /* => 1 */

Evaluation is from left to right.

The evaluation stops as soon as the truth or falsehood of the result is known.
(short-circuit evaluation)
Increment/Decrement

/* increment operators: x++, ++x
decrement operators: x--, --x */

int x = 4;
x++;       /* x == 5 */
++x;       /* x == 5 */
x--;       /* x == 3 */
--x;       /* x == 3 */

int y = x++; /* y == ??, x == ?? */
int y = ++x;

iClicker question:
A: y == 5, x == 4
B: y == 4, x == 4
C: y == 5, x == 5
D: y == 4, x == 5
E: none of the above
Increment/Decrement

/* increment operators: x++, ++x
decrement operators: x--, --x */

```c
int x = 4;
x++;    /* x == 5 */
++x;    /* x == 5 */
x--;    /* x == 3 */
--x;    /* x == 3 */
```

```c
int y = x++;  
int y = ++x;  /* y == ??, x == ?? */
```
Increment/Decrement

/* increment operators: x++, ++x
decrement operators: x--, --x */

int x = 4;
x++;       /* x == 5 */
++x;       /* x == 5 */
x--;       /* x == 3 */
--x;       /* x == 3 */

int y = x++; /* y == 4, x == 5 */
int y = ++x; /* y == 5, x == 5 */
Conditionals

/* if (c) { stmts } else { stmts }
   c ? e1 : e2;
*/

int x = 4;
if (x >= 4) {
    printf("x >= 4\n");
}
else {
    printf("x < 4\n");
}

int y = x >= 4 ? 1 : 0;
Conditionals

```c
/* if (c) { stmts } else { stmts }
    c ? e1 : e2;
 */

int x = 4;
if (x >= 4) {
    printf("x >= 4\n");
}
else {
    printf("x < 4\n");
}

int y = (x >= 4 ? 1 : 0);
```
Switch

/* switch (c) {
   cases
} */

char c;
switch (0) {
   case 0:
   case 1:
      c = 'a';
   case 2:
      c = 'b'; break;
   default:
      c = 'c';
}

What is the value of c after executing this switch statement?

c must be a character or integer value, and the cases must be character or integer constants
Switch

/* switch (c) {
   cases
} */

char c;

switch (0) {
   case 0:
   case 1:
      c = 'a'; break;
   case 2:
      c = 'b'; break;
   default:
      c = 'c';
}

How about now?
Loops

/*
   while (c) { stmts }

   do { stmts } while (c);

   for (e1; e2; e3) { stmts }
*/
break and continue

• Sometimes it is useful to control loop exits other than by testing and the top or bottom.

• **break:** provides an early exit from a loop just as from a switch statement.

• **continue:** causes a *next iteration* of the enclosing loop – “jump to start of loop”
Comments

/*
   Multi-line comments!
*/

// Single line comments!
C does not have a *boolean* type

- **If, while, etc., all work like this:**
  - 0 means ‘false’
  - Anything else means ‘true’

- **The boolean negation operator is !**
  - !0 gives 1
  - !(anything else) gives 0

- **Not to be confused with ~, ones complement**
- **Not to be confused with -, twos complement**
- **Different from Unix success code convention!**
A simple program

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("Hello, world.");
    return 0;
}
```
A simple program

#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("Hello, world.");
    return 0;
}
A simple program

main is the program entry point

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("Hello, world.");
    return 0;
}
```
A simple program

The first argument to main is the number of arguments passed to your program on the command line

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("Hello, world.");
    return 0;
}
```
A simple program

The second argument to main are the program arguments passed to your program on the command line

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("Hello, world.");
    return 0;
}
```
A simple program

The return value from main is an integer

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  puts("Hello, world.");
  return 0;
}
```

A return value of 0 indicates success.
Other values indicate failure and they are programmer defined.
Let’s look at some C programs!

• simple.c
• simple2.c
• simple3.c
• echo.c
• lines.c
• wc.c