C for Java Programmers

- lecture notes credits:

  - *Advanced Programming* (cs3995, Spring 2002, Prof Schulzrinne)

  - *Software Construction* (J. Shepherd)

  - *Operating Systems* at Cornell (Indranil Gupta)

- today:

  - Why learn C after Java?

  - A brief background on C

  - C preprocessor

  - Modular C programs

Why learn C after Java (1)?

- Both high-level and low-level language

  – OS: user interface to kernel to device driver

- Better control of low-level mechanisms

  – memory allocation, specific memory locations

- Performance sometimes better than Java (Unix, NT!)

  – usually more predictable (also: C vs. C++)

- Java hides many details needed for writing code, but in C you need to be careful because:

  – memory management responsibility left to you

  – explicit initialization and error detection left to you

  – generally, more lines of (your) code for the same functionality

  – more room for you to make mistakes

Why learn C after Java (2)?

- Most older code is written in C (or C++)

  – Linux, Unix/BSD

  – Windows

  – Most Java implementations

  – Most embedded systems

- Philosophical considerations:

  – Being multi-lingual is good!

  – Should be able to trace program from UI to assembly

C pre-history (1)

- 1960s: many new languages

  - COBOL for commercial programming (databases)

  - FORTRAN for numerical and scientific programs

  - PL/I as second-generation unified language

  - LISP, Simula for CS research, early AI

  - Assembler for operating systems and timing-critical code

- Operating systems:

  - OS/360

  - MIT/GE/Bell Labs Multics (PL/I)

C pre-history (2)

- Bell Labs (research arm of Bell System $\rightarrow$ AT&T $\rightarrow$ Lucent) needed own OS

- BCPL as Multics language

- Ken Thompson: B

- Unix = Multics - bits

- Dennis Ritchie: new language = B + types

- Development on DEC PDP-7 with 8K 16-bit words

C history

- C

  – Dennis Ritchie in late 1960s and early 1970s

  – systems programming language

  – make OS portable across hardware platforms

  – not necessarily for real applications — could be written in Fortran or PL/I

- C++

  – Bjarne Stroustrup (Bell Labs), 1980s

  – object-oriented features

- Java

  – James Gosling in 1990s, originally for embedded systems

  – object-oriented, like C++

  – ideas and some syntax from C

C for Java programmers

- Java is mid-90s, high-level *Object-Oriented (OO)* language

- C is early-70s, *procedural* language

- C advantages:

  - direct access to OS primitives (system calls)

  - more control over memory

  - fewer library issues — just execute

- C disadvantages:

  - language is portable, but APIs are not

  - no easy graphics interface

  - more control over memory (memory leaks)

  - preprocessor can lead to obscure errors

C vs. C++

- We'll cover both, but C++ should be largely familiar

- Very common in Windows

- Possible to do OO-style programming in C

- C++ can be rather opaque: encourages "clever" programming

C vs. Java (1)

Java	C
object-oriented	function-oriented
strongly-typed	can be overridden
polymorphism (+,==)	very limited (integer/float)
classes for name space	(mostly) single name space, file-oriented
macros are external, rarely used	macros common (preprocessor)
layered I/O model	byte-stream I/O
automatic memory management	function calls (C++ has some support)
no pointers	pointers (memory addresses) common
by-reference, by-value	by-value parameters
exceptions, exception handling	signals, signal handling
concurrency (threads)	library functions (system calls)
length of array	on your own
string as a type	on your own (byte[] or char[] with \0 end)
dozens of common libraries	OS-defined

C vs. Java (2)

- Java program

  - collection of classes

  - class containing main method is starting class

  - running `java StartClass` invokes `StartClass.main` method

  - JVM loads other classes as required

- C program

  - collection of functions

  - one function – `main()` – is starting function

  - running executable (default name `a.out`) starts main function

  - typically, single program with all user code linked in — but can be dynamic libraries (.dll, .so)

C vs. Java: simple example.

Java

```
public class hello {
    public static void main( String[] args ) {
        System.out.println( "hello world! " );
    }
}
```

C

```
#include <stdio.h>
int main( int argc, char *argv[] ) {
  puts( "hello world!" );
  return 0;
}
```

Dissecting the example

- `#include <stdio.h>` to include header file stdio.h

- `#` lines processed by pre-processor

- No semicolon at end of pre-processor lines

- Lower-case letters only — C is case-sensitive

- `void main(void){ ... }` is the only code executed

- `puts( " /* message you want printed */ ");`

- `\n` = newline, `\t = tab`

- `\` in front of other special characters within printf.

- `printf( "Have you heard of \"The Matrix\" ? \n" );`

Executing C programs (1)

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

- `argc` is the argument count
- `argv` is the argument vector

  – array of strings with command-line arguments

- the int value is the return value

  – convention: return value of 0 means success, > 0 means there was some kind of error

  – can also declare as `void` (no return value)

Executing C programs (2)

- Name of executable followed by space-separated arguments

- `$ a.out 1 23 "third arg"`

- this is stored like this:

argc

4

argv

a.out	1	23	"third arg"

Executing C programs (3)

- If no arguments, simplify:

```
int main() {
  puts( "hello world" );
  exit( 0 );
}
```

- Uses `exit()` instead of `return()` — almost the same thing.

Executing C programs (4)

- Java programs are compiled and interpreted:
  - `javac` converts `foo.java` into `foo.class`
  - class file is not machine-specific
  - *byte codes* are then interpreted by JVM
- C programs are compiled into object code and then linked into executables (to allow for multiple object files to work together):
  - `gcc` compiles `foo.c` into `foo.o` and then links `foo.o` into `a.out`
  - you can skip writing `foo.o` if there is only one object file used to create your executable
  - `a.out` is executed by OS and hardware

Executing C programs (5)

x.java → javac → java x

data → java x

args → java x

x.c → gcc → a.out

data → a.out

args → a.out
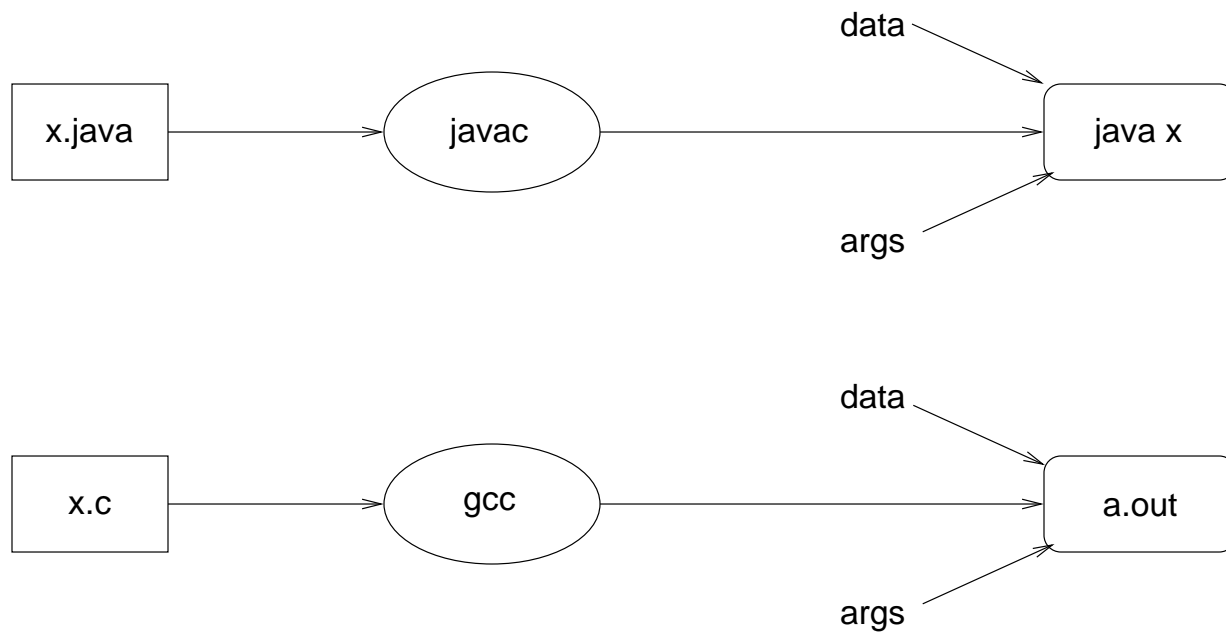
Compiling C programs (1)

- gcc is the C compiler we'll use in this class

- it's a free compiler from Gnu (i.e., Gnu C Compiler)

- gcc translates C program into executable for some target

- default file name `a.out`

```
$ gcc hello.c
$ a.out
hello world!
```

The C compiler gcc (2)

- Behavior controlled by command-line switches:

-o *filename*	output file for object or executable
-Wall	display all warnings
-c	compiles but doesn't link
-g	insert code for debugger (gdb)
-p	insert code for profiler
-I	specify path for include files
-L	specify path for library files
-l	specify library
-E	preprocessor output only

Using gcc

- Two-stage compilation

  1. pre-process and compile: `gcc -c hello.c`

  2. link: `gcc -o hello hello.o`

- Linking several modules:
  ```
  gcc -c a.c → a.o
  gcc -c b.c → b.o
  gcc -o hello a.o b.o
  ```

- Using math library:
  ```
  gcc -o calc calc.c -lm
  ```

Error reporting in gcc

- Multiple sources

- preprocessor: missing include files

- parser: syntax errors

- assembler: rare

- linker: missing libraries