Introduction to Programming Systems
CS 217

Goals

• Master the art of programming
  – exploit abstraction, modularity, interfaces
  – write efficient programs
  – write robust programs
• Learn C and the Unix development tools
  – C is the systems language of choice
  – Unix has a rich development environment
• Introduction to computer systems
  – operating systems and networks
  – compilers
  – machine architecture

The C Programming Language

• Systems programming language
  – originally used to write Unix and Unix tools
  – data types and control structures close to most machines
  – now also a popular application programming language
• Notable features
  – pointer (address) arithmetic
  – all functions are call-by-value
  – simple 2-level scope structure
  – no I/O or memory mgmt facilities (provided by libraries)
• History
  – BCPL  →  B  →  C  →  K&R C  →  ANSI C
Difficult Lessons

• Program specifications are ambiguous (buggy)
  – code you write (your assignments)
  – code you use
• Programming is mostly about writing robust
  code; the algorithms are often simple
• Systems programming cannot be rushed

Course Details

• Lectures
  – www.cs.princeton.edu/courses/cs217/
• Precepts
  – work through programming examples
  – demonstrate tools (gdb, makefiles, emacs, …)
• Assignments
  – six total (yes, you implement a shell)
  – 2/3rds of your grade
Course Details (cont)

• Textbooks
  – SPARC Architecture, Assembly Language Programming, and C. Paul.
  – C Interfaces and Implementations. Hanson.
  – Programming with GNU Software. Loukides & Oram.

Course Details (cont)

• Facilities
  – CIT’s arizona cluster
  – SPARC lab in Friend 016
  – Your own laptop
    ssh access to arizona
    run GNU tools on Windows
    run GNU tools on Linux
Programming Style (cont)

• Names
  – use descriptive names for globals and functions
    e.g., `elementCount`
  – use concise names for local variables
    e.g., `i` (not `arrayindex`) for loop variable
  – use case judiciously
    e.g., `PI`, `MAXLINE` (reserve for constants)
  – use consistent style for compound names
    e.g., `printword`, `PrintWord`, `print_word`
  – use module prefixes to distinguish names
    e.g., `StrSet_T`, `StrSet_add`

• Layout and indentation
  – use white space judiciously
    e.g., to separate code into paragraphs
  – use indentation to emphasize structure
    use editor’s autoindent facility
  – break long lines at logical places
    e.g., by operator precedence
  – line up parallel structures
    `alpha = angle(p1, p2, p3);`
    `beta = angle(p1, p2, p3);`
    `gamma = angle(p1, p2, p3);`

• Control structures
  `if (x < v[mid])`
  `  high = mid - 1;`
  `else if (x < v[mid])`
  `  low = mid + 1;`
  `else`
  `  return mid;`

Implement multiway branches with `if ... else if ... else`
emphasize that only one action is performed
avoid empty `then` and `else` actions
handle default action, even if can’t happen (use `assert(0)`)
avoid `continue`, minimize use of `break` and `return`
avoid complicated nested structures
Programming Style (cont)

- Documentation
  - comments should add new information
    
```c
    i = i + 1;    /* add one to i */
```
  - comments must agree with the code
  - comment procedural interfaces liberally
  - comment algorithms, not coding idiosyncracies
  - master the language and its idioms; let the code speak for itself

Process Memory

```
0
  text
  data
  bss
  heap

heap

stack

0xffffffff
```

Process Memory (cont)

```c
int i;
int j = 74;
main()
{
    char *p;
    p = malloc(8);
    ... 
}```
“What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD. From now on I shall have significantly greater respect for every successful software tool that I encounter. During the past decade I was surprised to learn that the writing of programs for TeX and Metafont proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks.”

Donald Knuth, 1989