COS 217: Introduction to Programming Systems

Fall 2007 (TTh 10:00-10:50 in CS 104)

Prof. David I. August

Preceptors: Bob Dondero (lead),
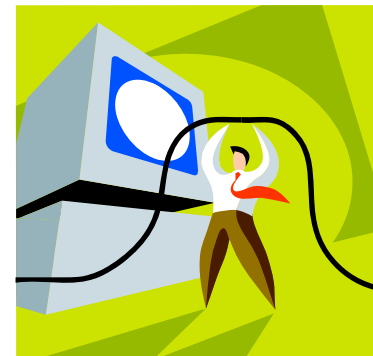Anirudh Badam, Lindsey Poole, Arun Raman

http://www.cs.princeton.edu/courses/archive/fall07/cos217/

1

Goals for Today's Class

- COS 217 overview
  - Goals of the course
  - Introductions
  - Learning the material
  - Course grading
  - Academic policies

- Getting started
  - Modularity/Interfaces/Abstraction
  - C Programming: How C differs from Java
  - Getting input and providing output

http://www.cs.princeton.edu/courses/archive/fall07/cos217/

2

Goals of COS 217

- Understand boundary between code and computer
  - Machine architecture
  - Operating systems
  - Compilers

- Learn C and the Unix development tools
  - C is widely used for programming low-level systems
  - Unix has a rich development environment
  - Unix is open and well-specified, good for study & research

- Improve your programming skills
  - More experience in programming
  - Challenging and interesting programming assignments
  - Emphasis on modularity and debugging

3

Introductions

- David August (professor)
  - Room 209 in Computer Science Building
  - august@cs.princeton.edu

- Bob Dondero (lead preceptor)
  - Room 206 in Computer Science Building
  - rdondero@cs.princeton.edu

- Anirudh Badam, Lindsey Poole, and Arun Raman
  - See web site for contact info

- Donna O'Leary (administrator)
  - Room 410 in Computer Science Building
  - doleary@cs.princeton.edu

4

Learning the Material: Tuning In

- Lecture
  - Goal: Introduce concepts and work through examples
  - When: TTh 10:00-10:50 in CS 104
  - Slides available online at course Web site

- Precept
  - Goal: Demonstrate tools and work through programming examples

- Website - get there from: http://www.cs.princeton.edu

- Mailing List at cos217@lists.cs.princeton.edu

Learning the Material: Books

- Required textbooks
  - *C Programming: A Modern Approach,* King, 1996.
  - *The Practice of Programming,* Kernighan and Pike, 1999.
  - *ONE OF:*
    - *Online -- Programming from the Ground Up*, Bartlett, 2004.
    - *Preferred -- Computer Systems: A Programmer's Perspective*, Randal E. Bryant and David R. O'Hallaron, Prentice-Hall 2003.

- Highly recommended
  - *Programming with GNU Software*, Loukides and Oram, 1997.

- Optional (available online)
  - *IA32 Intel Architecture Software Developer's Manual, Volumes 1-3*
  - *Tool Interface Standard & Executable and Linking Format*
  - *Using as, the GNU Assembler*

- Other textbooks (on reserve in the Engineering Library)
  - *The C Programming Language (2nd edition),* Kernighan and Ritchie, 1988.
  - *C: A Reference Manual*, Harbison and Steele, 2002.
  - *C Interfaces and Implementations*, Hanson, 1996.

Learning the Material: Doing

1. A "de-comment" program

2. A string module

3. A symbol table abstract data type (ADT)

4. A heap manager

5. UNIX commands in AI-32 assembly language

6. A buffer overrun attack

7. ???

Facilities for Programming

- Recommended options: OIT "hats" LINUX cluster
  - Friend Center 016 or 017 computer, secure shell to "hats", or
  - Your own PC, secure shell to "hats.princeton.edu" (Linux)
  - Why: common environment, and access to lab TAs

- Other option: on your own PC (not recommended; reasonable only for some parts of some assignments):
  - Running GNU tools on Linux, or
  - Running GNU tools on Windows, or
  - Running a standard C development environment

- Assignments are due Sundays (typically) at 9:00PM

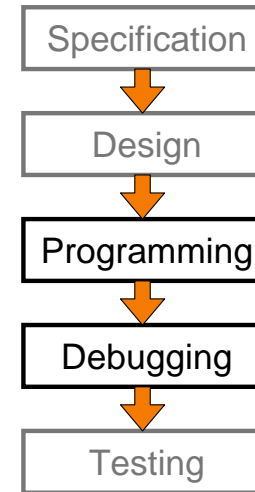- Advice: start early, to allow time for debugging (especially in the background while you are doing other things!)…

Why Debugging is Necessary…

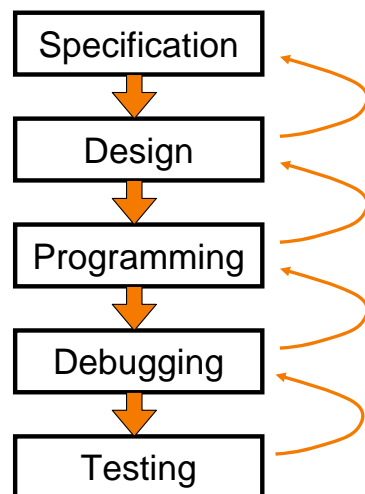

Copyright 2003 Randy Glasbergen.   www.glasbergen.com

Software in COS126

Specification

↓

Design

↓

Programming

↓

Debugging

↓

Testing

1 Person
10^2 Lines of Code
1 Type of Machine
0 Spec Modifications
1 Week

Software in the Real World

Specification

↓

Design

↓

Programming

↓

Debugging

↓

Testing

Lots of People
10^6 Lines of Code
Lots of Machines
Lots of Spec Modifications
1 Decade or more

Grading

- Seven programming assignments (60%)
  – Working code
  – Clean, readable, maintainable code
  – On time (penalties for late submission)

- Exams (30%)
  – Midterm
  – Final

- Class participation (10%)
  – Precept attendance is *mandatory*

Policies

Programming in an individual creative process much like composition. You must reach your own understanding of the problem and discover a path to its solution. During this time, discussions with friends are encouraged. However, when the time comes to write code that solves the problem, such discussions are no longer appropriate - the program must be your own work. If you have a question about how to use some feature of C, UNIX, etc., you can certainly ask your friends or the teaching assistants, but **do not, under any circumstances, copy another person's program.** Letting someone copy your program or using someone else's code in any form is a **violation of academic regulations**. "Using someone else's code" includes using solutions or partial solutions to assignments provided by commercial web sites, instructors, preceptors, teaching assistants, friends, or students from any previous offering of this course or any other course.

13

Any questions before we start?

14

Intuition: Modularity/Abstraction/Interfaces

Client

Interface
- universal remote
- volume
- change channel
- adjust picture
- decode NTSC, PAL signals

Implementation
- cathode ray tube
- electron gun
- Sony Wega 36XBR250
- 241 pounds, $2,699

15

Intuition: Modularity/Abstraction/Interfaces

Client

Interface
- universal remote
- volume
- change channel
- adjust picture
- decode NTSC, PAL signals

Implementation
- gas plasma monitor
- Pioneer PDP-502MX
- wall mountable
- 4 inches deep
- $19,995

Can substitute better implementation without changing client!

Good Software is Modularized

- Understandable
  - Well-designed
  - Consistent
  - Documented

  ← Write code in modules with well-defined interfaces

- Robust
  - Works for any input
  - Tested

  ← Write code in modules and test them separately

- Reusable
  - Components

  ← Write code in modules that can be used elsewhere

- Efficient
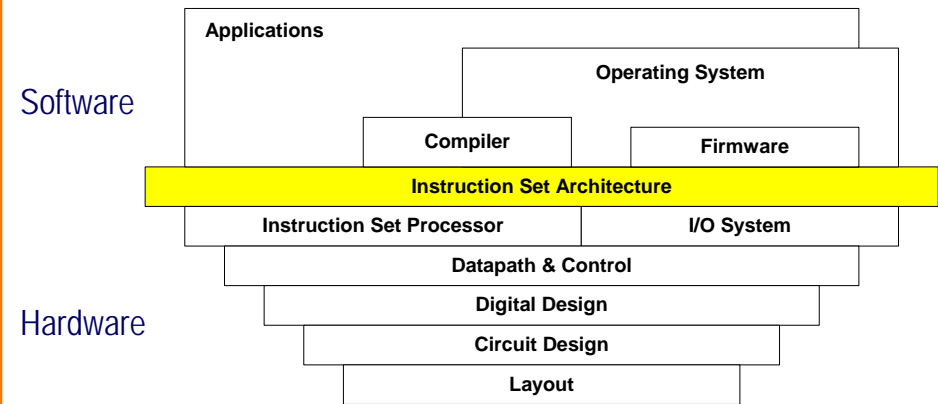 - Only matters for 1%

  ← Write code in modules and optimize the slow ones

17

System Interfaces/Abstraction

- Applications
- Operating System
- Compiler
- Firmware
- **Instruction Set Architecture**
- Instruction Set Processor
- I/O System
- Datapath & Control
- Digital Design
- Circuit Design
- Layout

Software

Hardware

More than 99.5% of Linux OS code goes through a compiler…
Almost 100% of application code…

Figure Source H&P

The C Programming Language

- "C has always been a language that never attempts to tie a programmer down."
- "C has always appealed to systems programmers who like the terse, concise manner in which powerful expressions can be coded."
- "C allowed programmers to (while sacrificing portability) have direct access to many machine-level features that would otherwise require the use of Assembly Language."
- "C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments." – Dennis Ritchie

19

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

- Pros and cons
  - Can do whatever you want: flexible and efficient
  - Can do whatever you want: can shoot yourself in the foot

- Notable features
  - All functions are call-by-value
  - Pointer (address) arithmetic
  - Simple scope structure
  - I/O and memory management facilities provided by libraries

- History
  - BCPL → B → C → K&R C → ANSI C
    1960   1970   1972   1978   1988
  - LISP → Smalltalk → C++ → Java

20

Java vs. C

- Abstraction
  - C exposes the raw machine
  - Java hides a lot of it

- Bad things you **can** do in C that you **can't** do in Java
  - Shoot yourself in the foot (safety)
  - Others shoot you in the foot (security)
  - Ignoring wounds (error handling)

- Dangerous things you **must** do in C that you **don't** in Java
  - Memory management (i.e., malloc and free)

- Good things that you **can** do in C, but Java **makes** you
  - Objected-oriented methodology

- Good things that you **can't** do in C but you **can** in Java
  - Portability

21

Java vs. C

	Java	C

Program	`hello.java:`
	
	`public class hello {`
	`    public static void`
	`      main(String[] args) {`
	`        System.out.println(`
	`          "Hello, world");`
	`    }`
	`}`
		`hello.c:`
		
		`#include <stdio.h>`
		
		`int main(void) {`
		`  printf("Hello, world\n");`
		`  return 0;`
		`}`

Compile	`% javac hello.java`
	`% ls`
	`hello.java   hello.class`
	`%`
		`% gcc hello.c`
		`% ls`
		`a.out       hello.c`
		`%`

Run	`% java hello`
	`Hello, world`
	`%`
		`% a.out`
		`Hello, world`
		`%`

22

Java vs. C, cont'd

	Java	C
Boolean	`boolean`	`int`
Char type	`char  // 16-bit unicode`	`char /* 8 bits */`
Void type	`// no equivalent`	`void`

Integer types	`byte    // 8 bits`
	`short   // 16 bits`
	`int     // 32 bits`
	`long    // 64 bits`
		`char`
		`short`
		`int`
		`long`

Floating point types	`float   // 32 bits` `double  // 64 bits`	`float` `double`
Constant	`final int MAX = 1000;`	`#define MAX 1000` `(enumerations, "const")`

Arrays	`int [] A = new int [10];`
	`float [][] B =`
	`    new float [5][20];`
		`int A[10];`
		`float B[5][20];`

Bound check	`// run-time checking`	`/* no run-time check */`

23

Java vs. C, cont'd

	Java	C
Pointer type	`// pointer implicit in` `// class variables`	`int *p;`

Record type	`class r {`
	`    int x;`
	`    float y;`
	`}`
		`struct r {`
		`    int x;`
		`    float y;`
		`}`

String type	`String s1 = "Hello";`
	`String s2 = new`
	`    String( "hello" );`
		`char *s1 = "Hello";`
		`char s2[6];`
		`strcpy( s2, "hello" );`

String concatenate	`s1 + s2`	`#include <string.h>` `strcat( s1, s2 );`
Logical	`&&, ||, !`	`&&, ||, !`
Compare	`=, !=, >, <, >=, <=`	`=, !=, >, <, >=, <=`
Arithmetic	`+, -, *, /, %, unary -`	`+, -, *, /, %, unary -`
Bit-wise ops	`>>, <<, >>>, &, |, ^`	`>>, <<, &, |, ^`

24

Java vs. C, cont'd

	Java	C
Comments	`/* comments */` `// another kind`	`/* comments */`

Block	`{`
	`    statement1;`
	`    statement2;`
	`}`
		`{`
		`    statement1;`
		`    statement2;`
		`}`

Assignments	`=, *=, /=, +=, -=, <<=,` `>>=, >>>=, =, ^=, |=, %=`	`=, *=, /=, +=, -=, <<=,` `>>=, =, ^=, |=, %=`
Function / procedure call	`foo( x, y, z );`	`foo( x, y, z );`
Function return	`return 5;`	`return 5;`
Procedure return	`return;`	`return;`

Java vs. C, cont'd

	Java	C

Conditional	`if (expression)`
	`    statement1`
	`else`
	`    statement2;`
		`if (expression)`
		`    statement1`
		`else`
		`    statement2;`

Switch	`switch (n) {`
	`  case 1:`
	`    ...`
	`    break;`
	`  case 2:`
	`    ...`
	`    break;`
	`  default:`
	`    ...`
	`}`
		`switch (n) {`
		`  case 1:`
		`    ...`
		`    break;`
		`  case 2:`
		`    ...`
		`    break;`
		`  default:`
		`    ...`
		`}`

"goto"	`// no equivalent`	`goto L;`
Exception	`throw, try-catch-finally`	`/* no equivalent */`

Java vs. C, cont'd

	Java	C

"for" loop	`for (int i=0;i<10;i++)`
	`    statement;`
		`int i;`
		`for (i=0; i<10; i++)`
		`    statement;`

"while" loop	`while (expression)` `    statement;`	`while (expression)` `    statement;`

"do- while" loop	`do {`
	`    statement;`
	`    …`
	`} while (expression)`
		`do {`
		`    statement;`
		`    …`
		`} while (expression)`

Terminate a loop body	`continue;`	`continue;`
Terminate a loop	`break;`	`break;`

Standard Input/Output
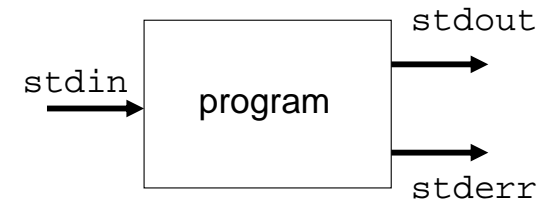
- Three standard I/O streams
  - In: stdin
  - Out (normal): stdout
  - Out (errors): stderr

- Binding
  - Flexible/dynamic binding of streams to actual devices or files
  - Default binding
    - stdin bound to keyboard
    - stdout and stderr bound to the terminal screen

Standard I/O in C

- Three standard I/O streams
  - **stdin**
  - **stdout**
  - **stderr**

- Basic calls for standard I/O
  - **int getchar(void);**
  - **int putchar(int c);**
  - **int puts(const char *s);**
  - **char *gets(char *s);**

- Use "man" pages
  ```
  % man getchar
  ```

copyfile.c:

```
#include <stdio.h>
int main(void) {
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```
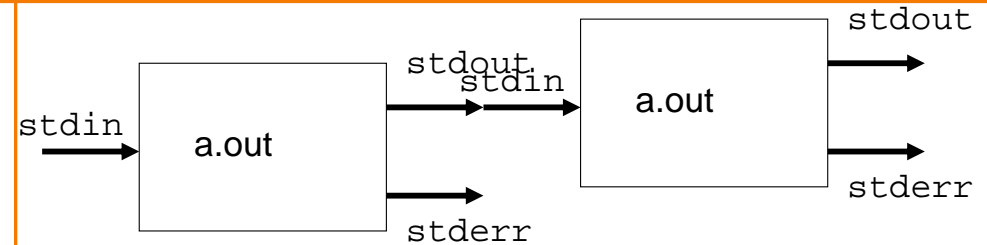
```
% a.out < file1 > file2

% a.out < file1 | a.out > file2

% a.out < file1 | a.out | a.out > file2
```

Pipes Connect Output to Input


```
% a.out < file1 | a.out > file2
```
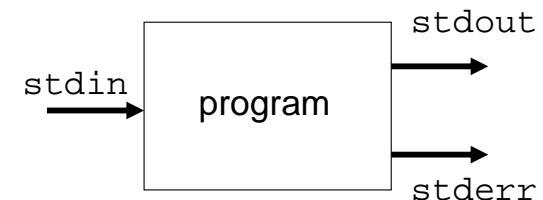

What's all this good for?

- In the old days…
  - Programmers hard-coded input/output devices into programs
  - Hard to program, and hard to port to different I/O devices

- Along came OS-360 (1964)
  - Separate I/O device driver (in OS) from data (in program)
  - A good early example of modularity and data abstraction
  - However, still clumsy to connect output of one program to input of another

---

What's all this good for?

- Unix (early 1970s)
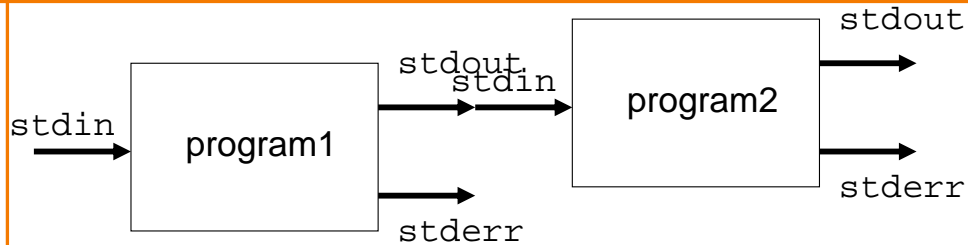 - First OS to have standard I/O redirection and pipes

- Standard I/O redirection
  - Write program once
  - Same program can be made to work for different input/output devices at run time

- Good practice of modularity

What's all this good for?

- Pipes
  - Write small programs that specialize in very simple tasks
  - Connect lots of smaller programs to make bigger programs
  - Makes bigger programs easier to write
  - Earliest and best success story of programming with components

- Standard I/O redirection & pipes: big part of Unix success

- Good practice of modularity is a learned art

Formatted Output: printf

- `int printf(char *format, ...);`
  - **Translate arguments into characters according to "format"**
  - **Output the formatted string to stdout**

- **Conversions (read "man printf" for more)**
  - **%d – integer**
  - **%f – float or double**
  - **%3f – float or double with 3 decimal places**
  - **%% – percent**

- **Examples**
  - `int i = 217;`
    `printf("Course number is: %d", i );`

Formatted Input: scanf

- `int scanf(const char *format, ...);`
  - **Read characters from stdin**
  - **Interpret them according to "format" and put them into the arguments**

- **Conversions (read "man scanf" for more)**
  - **%d – integer**
  - **%f – float**
  - **%lf – double**
  - **%% – literal %**

- **Example**
  - `double v;`
    `scanf( "%lf", &v );`
  - `int day, month, year;`
    `scanf( "%d/%d/%d", &month, &day, &year);`

Standard Error Handing: stderr

- `stderr` is the second output stream for output errors

- Some functions to use `stderr`
  - `int fprintf(FILE *stream, const char *format, ...);`
    - **Same as `printf` except the file stream**
  - `int fputc(int c, FILE *stream);`
    - `putc()` **is the same as** `fputc()`
  - `int fgetc(FILE *stream);`
    - `getc()` **is the same as** `fgetc()`

- **Example**
  - `fprintf( stderr, "This is an error.\n" );`
  - `fprintf( stdout, "This is correct.\n" );`
  - `printf( "This is correct.\n" );`

Example

```
#include <stdio.h>
#include <stdlib.h>

const double KMETERS_PER_MILE = 1.609;

int main(void) {
    int miles;
    double kmeters;
    printf("miles: ");
    if ( scanf("%d", &miles) != 1 ) {
        fprintf( stderr, "Error: Expect a number.\n");
        exit(EXIT_FAILURE);
    }
    kmeters = miles * KMETERS_PER_MILE;
    printf("= %f kilometers.\n", kmeters );
    return 0;
}
```

Summary

- The goal of this course:
  - Master the art of programming
  - Learn C and assembly languages for systems programming
  - Introduction to computer systems

- It is easy to learn C if you already know Java
  - C is not object oriented, but many structures are similar
  - Standard I/O functions are quite different from Java's input and output

- Next lecture
  - Character input and output

http://www.cs.princeton.edu/courses/archive/fall07/cos217/