CS 201
Computer Systems Programming

Prof. Harry Porter

www.cs.pdx.edu/~harry/cs201
Agenda for Today

Attendance
Course objectives
Syllabus
  Textbooks, policies, class mailing list, etc.
Assignment 1
Overview of the “C” programming language
Intel Core i7
- 4 Processors (CPUs) per chip
- 14 nm feature size
- 4 GHz clock speed
- 2 billion transistors
- 1100 pins
The Big Questions

What is “System Software” and how does it work?

• Compiling
• Assembly Language
• OS organization

How is a program actually executed?

• Skills and knowledge of “C” programming

How does the hardware get the job done?

• Computer Architecture
Syllabus

Course home page

cs.pdx.edu/~harry/cs201

Course Schedule
Homeworks
Lecture Slides
Information about instructor, TA, office hours
Email mailing list (mailman):

PorterClassList
Syllabus

Accounts

Instructions on course web page
Activate your account in person at CAT front desk.
linux.cs.pdx.edu
Linux systems in FAB 88-09 and FAB 88-10
(Homework assignments will be tested here.)

Login remotely or in person (Basement of EB)
ssh user@linux.cs.pdx.edu
- ssh is included in the “putty” package
  http://www.chiark.greenend.org.uk/~sgtatham/putty
- ssh is included in “cygwin”
  http://www.cygwin.com
Alternatively, you can develop your code via cygwin
Code will be graded on linuxlab so make sure it works on linuxlab systems
Textbooks


Syllabus

Getting help

- CS Tutors
- TA and instructor office hours
- On-line resources for gdb, make, etc.

Policies

- You are responsible for everything that takes place in class
- Reading assignments will be posted with each lecture
- Homework assignments due at start of class on due date
  - Follow submission instructions on home page carefully, especially for programming assignments.
  - Late policy: 50% off, if not submitted before class time.
Academic integrity

- Automatic failing grade given
- Departmental guidelines available in CS office

What is not cheating?

- Discussing the design for a program is OK.
- Helping each other orally (not in writing) is OK.
- Using anything out of the textbook or my slides is OK.
- Copying code “snippets”, templates for system calls, or declarations from a reference book or header files are OK

What is cheating?

- Copying code verbatim without attribution
  - Source-code plagiarism tools
- Copying someone’s answer or letting someone copy your answer.
- Mailing code to the class mailing list.
Grading

Reading assignments
Practice problems in the textbook – do them!

Homework:

- Programming assignments in C – email to grader
- Written homework – hand in hardcopy

Two exams
- Midterm exam
- Final comprehensive exam (covering entire term)
Supporting Video Material

- Accessible through website
- To augment lectures
- 100% Attendance is required

“Binary Numbers”
“Assembly Language and Processor Architecture”
The specification is on course web site
   cs.pdx.edu/~harry/cs201

The Task: Write a small C program

Will use several system calls
   rand, gettimeofday, printf, scanf, strlen, etc…

Goal: Learn C
   • Discover an algorithm
   • Write clean, well-formatted code
   • Write appropriate comments

Grader will run and read your program
Due in two weeks

If you are unable to complete this program on time, you should consider dropping the course.
Introduction to C Programming
Why Learn C?

Used prevalently

- Operating systems (e.g. Windows, Linux, OS X)
- Web servers
- Web browsers
- Mail servers
- DNS servers
- Video games
- Graphics card programming

Why?

- Performance
- Portability
- Flexibility / Ability to do things
Why Learn C?

Compared to other high-level languages

- Maps almost directly into hardware instructions
- Code efficiency!!!
  - C Provides a minimal set of abstractions
  - Other *High-Level Languages* make programming simpler at the expense of efficiency

Compared to assembly programming

- Abstracts out hardware (i.e. registers, memory addresses)
  - Possible to write portable code
  - A “portable assembly language”
- Provides variables, functions, arrays, complex arithmetic and boolean expressions
Why Learn Assembly?

Learn how programs map onto underlying hardware
- Allows programmers to write efficient code

Perform platform-specific tasks
- Access and manipulate hardware-specific registers
- Interface with hardware devices
- Utilize latest CPU instructions

Reverse-engineer unknown binary code
- Analyze security problems caused by CPU architecture
- Identify what viruses, spyware, rootkits, and other malware are doing
- Understand how cheating in on-line games work
The C Programming Language

One of many programming languages

**Imperative** (procedural) programming language
- Computation consisting of statements that change program state
- Language makes explicit references to state (i.e. variables)
- Computation broken into modular components (“procedures” or “functions”) that can be called from any point

**Declarative** programming languages
- Describes what something is like, rather than how to create it
- Implementation left to other components
- Examples: HTML, SQL
The C Programming Language

Simpler than C++, C#, Java

No support for…
- Objects
- Memory management
- Array bounds checking
- Non-scalar operations

Simple support for…
- Typing
- Structures

Extended Functionality? Just a collection of functions
- Functions are in “libraries” (libc, libpthread, libm)

Low-level, direct access to machine memory specifics

Easier to write bugs, harder to write programs, typically faster
The C Programming Language

Compilation is to machine code
- Same as C++
- Compiled, assembled, linked via gcc

Compared to interpreted languages...  
Perl / Python / Ruby ...  
- Commands interpreted by interpreter software  
- Interpreter runs natively  
Java  
- Compiles to virtual machine “byte codes”  
- Byte codes interpreted by virtual machine software  
- Virtual machine runs natively (and is written in C)  
  (Exception: “Just-In-Time” (JIT) compilation to machine code)
Our environment

All programs must run on the CS Linux Lab machines

ssh user@linuxlab.cs.pdx.edu

Architecture will be x86-64

IA-32 (32 bits)

i386

x86-64 (64 bits)

Intel-64, AMD64

IA-64 (64 bits)

“Itanium”
Our environment

GNU gcc compiler

gcc –m64 –Wall hello.c –o hello

–m64 = compile for 64-bit machines
–Wall = print warnings as well as errors

GNU gdb debugger

Must use “-g” flag when compiling and remove –O flags

gcc –g hello.c

(Will add debug symbols; will not reorder instructions for optimized performance)

- ddd is a graphical front end to gdb
- “gdb -tui” is a graphical curses interface to gdb
Variables

Identifiers use letters, numbers, some special characters

Examples:

\[ x \quad \text{mySizeVar} \quad x_43 \quad \text{_init} \]

Must be declared before use

- Contrast to typical scripting languages (Perl, Python, PHP, JavaScript)
- C is statically typed

Static Typing

Compiler checks for type errors

\[ x = 123 \quad \text{“hello”}; \]

Dynamic Typing

Less checking; Errors may occur at runtime
Data Types and Sizes

**char** – single byte integer
- 8-bit characters
- Strings implemented as arrays of char and referenced via a pointer to the first char of the array

**short** – short integer
- 16-bit (2 bytes), not used much

**int** – integer, *size varies by architecture*
- Normally 32-bits (4 bytes)

**float** – single precision floating point
- 32-bit (4 bytes)

**double** – double precision floating point
- 64-bit (8 bytes)
Integer Types

16 bit integers
  short int

32 bit integers
  int

32 or 64 bits
  long int

64 bit integers
  long long int
Integer Types - Synonyms

16 bit integers
- short int
- short

32 bit integers
- int

32 or 64 bits
- long int
- long

64 bit integers
- long long int
- long long
**Integer Types - Synonyms**

16 bit integers
- short int
- short

32 bit integers
- int

32 or 64 bits
- long int
- long

64 bit integers
- long long int
- long long

Preferred
Constants

Integer literals

1234
0xFE    0xab78

Character constants

' a ' is the numeric value of character ‘a’ in the ASCII code (decimal=97, hex=0x61)

char letterA = 'a';
int asciiA = 'a';

String Literals

"I am a string"
"

// This is the empty string.

What's the difference?
Constant pointers

Used for static arrays

- Symbol that points to a fixed location in memory
  ```c
  char amsg[ ] = "This is a string";
  ```
- Can change characters in string
  ```c
  amsg[3] = 'x';
  ```
- Can not reassign amsg to point elsewhere
  ```c
  char p[ ] = "This is a different string";
  amsg = p;
  ```
Declarations and Operators

Variable declaration

```c
int foo;
char *ptr;
float ff;
```

Can include initialization

```c
int foo = 34;
char *ptr = "fubar";
float ff = 34.99;
```

Arithmetic operators

- `+`, `-`, `*`, `/`, `%`
- Modulus operator (`%`)
- Arithmetic operators associate left to right
Expressions

In C, oddly, assignment is an expression

“x = 4” has the value 4

```c
if (x == 4)
    y = 3;  /* sets y to 3 if x is 4 */

if (x = 4)
    y = 3;  /* always sets y to 3 */

while ((c=getchar()) != EOF) ...
```
Example: Using == in an Exploit

But on Nov. 5, 2003, Larry McVoy noticed that there was a code change in the CVS copy that did not have a pointer to a record of approval. Investigation showed that the change had never been approved and, stranger yet, that this change did not appear in the primary BitKeeper repository at all. Further investigation determined that someone had apparently broken in (electronically) to the CVS server and inserted this change.

What did the change do? This is where it gets really interesting. The change modified the code of a Linux function called wait4, which a program could use to wait for something to happen. Specifically, it added these two lines of code:

```c
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
    retval = -EINVAL;
```

Increment and Decrement

Comes in prefix and postfix flavors

i++
++i
i--
--i

Makes a difference in evaluating complex statements

- A major source of bugs
- Prefix: increment happens before evaluation
- Postfix: increment happens after evaluation

**Important to understand:**

When the actual increment/decrement occurs

Is “i++*2” the same as “++i*2”?
Comparison to Java

Operators same as Java:

- Arithmetic
  
  
  ```
  +  -  *  /  \
  i = i+1;  i++;  i--;  i *= 2;
  ```

- Relational and Logical

  ```
  <  >  <=  >=  ==  !=
  &&  ||  &  |  !
  ```

Control flow syntax same as in Java:

```java
if ( ) { } else { }
while ( ) { }
do { } while ( );
for (i=1; i <= 100; i++) { }
switch ( ) {case 1: ... }
continue; break;
```
### Simple data types are the same

<table>
<thead>
<tr>
<th>datatype</th>
<th>size</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>-128 ... 127</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-32,768 ... 32,767</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2,147,483,648 ... 2,147,483,647</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>-10^{18} ... +10^{18}</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>3.4 \times 10^{\pm 38}  (7 digits)</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>1.7 \times 10^{\pm 308} (15 digits long)</td>
</tr>
</tbody>
</table>

*Exact details of size are “implementation dependent”!*
Java programmer gotchas (1)

You must declare variables ahead of time in C:

```c
{  
  int i  
  for (i = 0; i < 10; i++)  
    ...  
}
```

This is not okay in C:

```c
{  
  for (int i = 0; i < 10; i++)  
    ...  
}
Java programmer gotchas (2)

Uninitialized variables in C are allowed!!!
  ■ Best practice: Always use `-Wall` compiler option

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  int i;
  factorial(i);
  return 0;
}
```
Java programmer gotchas (3)

Error handling

There is no “throw/catch” mechanism

Ways to deal with an error:

- Return a special code to indicate an error.
- Set a global variable.
- Install a signal handler.

(We will learn about “signals” later)
Java programmer gotchas (4)

Dynamic memory

- Managed languages such as Java perform memory management (i.e., garbage collection) for programmers.
- C requires the programmer to *explicitly* allocate and deallocate memory.
- No “new” construct to create objects.

Memory can be allocated dynamically at run-time.

Allocate with `malloc()`
Deallocate with `free()`

You supply the number of bytes you want.
The “Hello, world” Program

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    /* print a message*/
    printf("Hello, world!\n");
    return 0;
}
```

```
$ gcc -Wall hello.c -o hello
$ ./hello
Hello, world!
$ 
```
#include <stdio.h>

- Include the contents of the file stdio.h
  - Case sensitive – lower case only
- No semicolon at the end of line

int main(...) 

- The OS calls this function when the program starts running.

printf(format_string, arg1, ...)

- Calls a function from libc library
- Prints out a string, specified by the format string and the arguments.
main has two arguments from the command line

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

**argc**
Number of arguments (including program name)

**argv**
Pointer to an array of string pointers

```
argv[0]: = program name
argv[1]: = first argument
argv[argc-1]: last argument
```

Example:  

```bash
$ find . -print
```

```
argc = 3
argv[0] = “find”
argv[1] = “.”
argv[2] = “-print”
```
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    printf("%d arguments\n", argc);
    for(i = 0; i < argc; i++)
        printf("  %d: %s\n", i, argv[i]);
    return 0;
}
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    printf("%d arguments\n", argc);
    for (i = 0; i < argc; i++)
        printf("  %d: %s\n", i, argv[i]);
    return 0;
}

$ ./cmdline CS-201 is for SERIOUS programmers
6 arguments
  0: ./cmdline
  1: CS-201
  2: is
  3: for
  4: SERIOUS
  5: programmers
$
Arrays

```c
char foo[80];

An array of 80 characters

sizeof(foo) = ???
```

Array elements are stored contiguously in memory.
Arrays

```c
char foo[80];

An array of 80 characters
```

```c
sizeof(foo)  
= 80 \times \text{sizeof(char)}  
= 80 \times 1  
= 80 \text{ bytes}
```

Array elements are stored contiguously in memory.
Arrays

```c
char foo[80];
   An array of 80 characters
   sizeof(foo)
    = 80 \times sizeof(char)
    = 80 \times 1
    = 80 bytes

int bar[40];
   An array of 40 integers
   sizeof(bar)
    =
    =
    =

Array elements are stored contiguously in memory.
```
Arrays

char foo[80];
   An array of 80 characters
   sizeof(foo)
       = 80 × sizeof(char)
       = 80 × 1
       = 80 bytes

int bar[40];
   An array of 40 integers
   sizeof(bar)
       = 40 × sizeof(int)
       = 40 × 4
       = 160 bytes

Array elements are stored contiguously in memory.
#include <stdio.h>

struct person {
    char* name;
    int age;
}; /* <= DO NOT FORGET the semicolon */

int main(int argc, char* argv[]) {
    struct person prof;
    prof.name = "Harry Porter";
    prof.age = 59;

    printf("%s is %d years old\n", prof.name, prof.age);
    return 0;
}
Pointers

Pointers are variables that hold an address in memory. The address “points to” another variable.

Very powerful idea!

```c
int x;
```

x:
Pointers are variables that hold an address in memory. The address “points to” another variable. Very powerful idea!

```c
int x;
int * myPtrVar;
```

myPtrVar: 

x: 
Pointers

Pointers are variables that hold an address in memory.
The address “points to” another variable.
Very powerful idea!

```c
int x;
int * myPtrVar;
myPtrVar = &x;
```
Using Pointers

```c
float f;    /* data variable */
```

```c
f:
```
Using Pointers

```c
float f; /* data variable */
```

```plaintext
f: 2300
```

memory
Using Pointers

```c
float f;    /* data variable */
float *f_addr;  /* pointer variable */
```

![Memory Diagram](image)
Using Pointers

```c
float f;    /* data variable */
float *f_addr; /* pointer variable */
```

```
f:     2300

f_addr: 3408
memory
```
Using Pointers

```c
float f;          /* data variable */
float *f_addr;    /* pointer variable */
f_addr = &f;      /* & = address operator */
```

```
f:  2300
    memory
f_addr: 3408
```
Using Pointers

```c
float f;        /* data variable */
float *f_addr;  /* pointer variable */
f_addr = &f;    /* & = address operator */
```
Using Pointers

```c
float f;  /* data variable */
float *f_addr;  /* pointer variable */
f_addr = &f;  /* & = address operator */
```
Using Pointers

```c
float f;    /* data variable */
float *f_addr;  /* pointer variable */
f_addr = &f;    /* & = address operator */
*f_addr = 3.2;  /* indirection operator */
```

```
f:  2300
```

```
f_addr: 3408
```

memory

```
3408 2300
```
Using Pointers

```c
float f;    /* data variable */
float *f_addr; /* pointer variable */
f_addr = &f; /* & = address operator */
*f_addr = 3.2; /* indirection operator */
```
Using Pointers

```c
float f;    /* data variable */
float *f_addr;   /* pointer variable */
f_addr = &f;    /* & = address operator */
*f_addr = 3.2;  /* indirection operator */
float g = *f_addr;  /* indirection: g is now 3.2 */
```
Using Pointers

```c
float f;    /* data variable */
float *f_addr;    /* pointer variable */
f_addr = &f;    /* & = address operator */
*f_addr = 3.2;    /* indirection operator */
float g = *f_addr;    /* indirection: g is now 3.2 */
```

memory

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>2300</td>
</tr>
<tr>
<td>f_addr</td>
<td>3408</td>
</tr>
<tr>
<td>g</td>
<td>5670</td>
</tr>
</tbody>
</table>
```

```
3.2
```
Using Pointers

```c
float f;       /* data variable */
float *f_addr; /* pointer variable */
f_addr = &f;   /* & = address operator */
*f_addr = 3.2; /* indirection operator */
float g = *f_addr; /* indirection: g is now 3.2 */
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2300</td>
<td>3.2</td>
</tr>
<tr>
<td>3408</td>
<td>2300</td>
</tr>
<tr>
<td>5670</td>
<td>3.2</td>
</tr>
</tbody>
</table>

- `f`: float variable at memory location 2300
- `f_addr`: pointer to `f` at memory location 3408
- `g`: value of `f` through `f_addr` at memory location 5670
Using Pointers

```c
float f;        /* data variable */
float *f_addr;  /* pointer variable */
f_addr = &f;    /* & = address operator */
*f_addr = 3.2;  /* indirection operator */
float g = *f_addr; /* indirection: g is now 3.2 */
f = 1.78;       /* but g is still 3.2 */
```

In memory:

- `f`: 2300, 3.2
- `f_addr`: 3408, 2300
- `g`: 5670, 3.2
Using Pointers

```c
float f;     /* data variable */
float *f_addr;  /* pointer variable */
f_addr = &f;   /* & = address operator */
*f_addr = 3.2; /* indirection operator */
float g = *f_addr; /* indirection: g is now 3.2 */
f = 1.78;     /* but g is still 3.2 */
```
Using Pointers

```c
float f;  /* data variable */
float *f_addr;  /* pointer variable */
f_addr = &f;  /* & = address operator */
*f_addr = 3.2;  /* indirection operator */
float g = *f_addr;  /* indirection: g is now 3.2 */
f = 1.78;  /* but g is still 3.2 */
```

`f`: 1.78

`f_addr`: 

`g`: 3.2
Arguments vs. Parameters

The caller / calling code

```c
x = 123;
w = foo (x*y, z, -1);
printf (...);
```

The callee / called function

```c
int foo (int a, int b, int c) {
    ...
    return (a+b+c);
}
```
Function call parameters

In C, all function arguments are passed “by value”.

“pass by value”
The called function is given a copy of the argument.
The data is copied from “caller” into the function.
Within the function, the parameter is a local variable.

Note:
The function can’t alter variables in the caller function!
Pass by Reference

What if you wish to modify an argument?

“pass by reference”
The called function is given a pointer to the argument.
The data is not copied.
Within the function, the original variable is modified.

Call-by-reference requires language support.
Call-by-reference is NOT SUPPORTED in “C”
... directly.
“C” has a mechanism that you can use:
Pointers!
In C all arguments are passed using “call-by-value.”
You can achieve call-by-reference, but you must program it.
**Example: Pass by Value**

```cpp
void swap1(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

**Before:**

```plaintext
x=3   y=4
swap1(x,y);
```

**After?**

```plaintext
x=3   y=4
or
x=4   y=3
```
Example: Pass by Value

```c
void swap1(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

Before:
    x=3   y=4
    swap1(x,y);

After?
    x=3   y=4
        or
    x=4   y=3
```
Example: Pass by Reference

```c
void swap2(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

**Before:**
- \( x = 3 \)
- \( y = 4 \)

```
swap2(&x,&y);
```

**After?**
- \( x = 3 \)
- \( y = 4 \)

**or**
- \( x = 4 \)
- \( y = 3 \)
Example: Pass by Reference

```c
void swap2(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Before:

```
x=3   y=4
```

swap2(&x,&y);

After?

```
x=3   y=4
```

or

```
x=4   y=3
```
Pass by value

A copy is made of the argument.
Within the function…
   The parameter is a local variable.
   You can update it, but the original will be unchanged.

*In C, all arguments are passed “by value”.*

Pass by reference

The value is not copied.
A pointer to the argument is passed to the function.
Within the function…
   The original variable is modified.

*You can implement “call-by-reference” using pointers.*
(The pointers are passed by value.)
Function calls (static)

Calls to functions normally resolved statically
(“Static” means done at compile-time.)

```c
void print_ints(int a, int b) {
    printf("%d %d\n", a, b);
}

int main(int argc, char* argv[]) {
    int i=3;
    int j=4;
    print_ints(i, j);
}
```
Function calls (dynamic)

Using function pointers, C can support late-binding of functions where calls are determined at run-time

```c
#include <stdio.h>

void print_even(int i){ printf("Even %d\n",i);}
void print_odd (int i){ printf("Odd %d\n",i); }

int main(int argc, char **argv) {
    void (*fp)(int);
    if !(argc%2)
        fp=print_even;
    else
        fp=print_odd;
    fp(argc);
}
```