Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Introduction to C

"the Universal Assembly Language"

- "Some" C experience is required before CS61C
- C++ or Java OK

"C"

“Programming Language”

- Based on pre-semester survey:
  - 83% already know JAVA
  - 54% already know C++
  - 34% already know C
  - 7% already know C#
  - About 10% have not taken 61B or equivalent
- If you have no experience in these languages, then
  start early and ask a lot of questions in discussion!

Disclaimer

- You will not learn how to fully code in C in these lectures! You’ll still need your C reference for this course:
  - K&R is a must-have
  - Check online for more sources
  - “JAVA in a Nutshell,” O’Reilly
    - Chapter 2, “How Java Differs from C”
    - http://oreilly.com/catalog/javapn/excerpt/
  - Brian Harvey’s helpful transition notes
    - On CS61C class website
    - http://inst.eecs.berkeley.edu/~cs61c/resources/
    - HarveyNotesC1-3.pdf
- Key C concepts: Pointers, Arrays, Implications for Memory management

Levels of Representation/ Interpretation

High Level Language
Program (e.g., C)

Compiler
Assembly Language
Program (e.g., MIPS)

Machine Language
Program
(MIPS)

Machine Interpretation

Hardware Architecture Description
(e.g., block diagrams)

Architecture Implementation

Logic Circuit Description
(Circuit Schematic Diagrams)

Introduc)on to C (Part I)

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/sp11

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Levels of Representation/ Interpretation

High Level Language
Program (e.g., C)

Compiler
Assembly Language
Program (e.g., MIPS)

Machine Language
Program
(MIPS)

Machine Interpretation

Hardware Architecture Description
(e.g., block diagrams)

Architecture Implementation

Logic Circuit Description
(Circuit Schematic Diagrams)

Introduc)on to C (Part I)

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/sp11

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Levels of Representation/ Interpretation

High Level Language
Program (e.g., C)

Compiler
Assembly Language
Program (e.g., MIPS)

Machine Language
Program
(MIPS)

Machine Interpretation

Hardware Architecture Description
(e.g., block diagrams)

Architecture Implementation

Logic Circuit Description
(Circuit Schematic Diagrams)

Introduc)on to C (Part I)

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/sp11

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Summary

Levels of Representation/ Interpretation

High Level Language
Program (e.g., C)

Compiler
Assembly Language
Program (e.g., MIPS)

Machine Language
Program
(MIPS)

Machine Interpretation

Hardware Architecture Description
(e.g., block diagrams)

Architecture Implementation

Logic Circuit Description
(Circuit Schematic Diagrams)
<table>
<thead>
<tr>
<th>Type of Language</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program-ming Unit</td>
<td>Function Oriented</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>Compilation</td>
<td>gcc hello.c creates machine language code</td>
<td>javac Hello.java creates Java virtual machine language bytecode</td>
</tr>
<tr>
<td>Execution</td>
<td>a.out loads and executes program</td>
<td>java Hello interprets bytecode</td>
</tr>
<tr>
<td>Storage</td>
<td>Manual (malloc, free)</td>
<td>Automatic (garbage collection)</td>
</tr>
</tbody>
</table>

### Compilation: Overview

- **C compilers** map C programs into architecture-specific machine code (string of 1s and 0s)
  - Unlike Java, which converts to architecture independent bytecode
  - Unlike most Scheme environments, which interpret the code
  - These differ mainly in exactly when your program is converted to low-level machine instructions ("levels of interpretation")
  - For C, generally a two part process of compiling, c files to .o files, then linking the .o files into executables; Assembling is also done (but is hidden, i.e., done automatically, by default)

### Compilation: Advantages

- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- Fair compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
- Why C?: we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism

### Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on CPU type and the operating system
- Executable must be rebuilt on each new system
  - i.e., "porting your code" to a new architecture
- "Change → Compile → Run [repeat]" iteration cycle can be slow, during the development cycle

### Basic C Concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Creates usable programs from C source</td>
</tr>
<tr>
<td>Typed variables</td>
<td>Kind of data that a variable contains</td>
</tr>
<tr>
<td>Typed functions</td>
<td>The kind of data returned from a function</td>
</tr>
<tr>
<td>Header files (.h)</td>
<td>Declare functions and variables in a separate file</td>
</tr>
<tr>
<td>Structs</td>
<td>Groups of related values</td>
</tr>
<tr>
<td>Enums</td>
<td>Lists of predefined values</td>
</tr>
<tr>
<td>Pointers</td>
<td>Aliases to other variables</td>
</tr>
</tbody>
</table>

These concepts distinguish C from other languages you may know

### Typed Variables in C

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer numbers, including negatives</td>
</tr>
<tr>
<td>unsigned int</td>
<td>integer numbers (no negatives)</td>
</tr>
<tr>
<td>float</td>
<td>floating point decimal numbers</td>
</tr>
<tr>
<td>char</td>
<td>single text character or symbol</td>
</tr>
<tr>
<td>double</td>
<td>greater precision FP number</td>
</tr>
<tr>
<td>long</td>
<td>larger signed integer</td>
</tr>
</tbody>
</table>

- You have to declare the type of data a variable will hold
- Types can’t change

These concepts distinguish C from other languages you may know
Typed Functions in C

- You have to declare the type of data you plan to return from a function.
- Return type can be any C variable type, and is placed to the left of the function name.
- You can also specify the return type as void.
- Also necessary to define types for values passed into a function.
- Variables and functions MUST be defined before they are used.

```c
int numberOfPeople ()
{
    return 3;
}

float dollarsAndCents ()
{
    return 10.33;
}

char firstLetter ()
{
    return 'A';
}
```

Structs in C

- Structs are structured groups of variables, e.g.,
- typedef struct {
    int lengthInSeconds;
    int yearRecorded;
} Song;
- Song song1;
- song1.lengthInSeconds = 213;
- song1.yearRecorded    = 1994;
- Song song2;
- song2.lengthInSeconds = 248;
- song2.yearRecorded    = 1988;

Dot notation: x.y = value

Confs and Enums in C

- Constant is assigned a value once in the declaration; value can’t change until the program is restarted.
- const float goldenRatio = 1.618;
- const int daysInWeek = 7;
- You can have a constant version of any of the standard C variable types.
- Enums: a group of related constants used to parameterize libraries.

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
A First C Program: Hello World

Original C:
main()
{
    printf("Hello World\n");
}

ANSI Standard C:
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return (0);
}

A Second C Program: Compute Table of Sines

#include <stdio.h>
#include <math.h>

void main()
{
    int angle_degree;
    double angle_radian, pi, value;

    /* Print a header */
    printf("Compute a table of the sine function\n");

    /* obtain pi once for all */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f\n", pi);

    printf("angle Sine\n");
    angle_degree = 0; /* initial angle value */

    while (angle_degree <= 360) /* loop until angle_degree > 360 */
    {
        angle_radian = pi*angle_degree/180.0;
        value = sin(angle_radian);
        printf(" %3d % 7f\n", angle_degree, value);
        angle_degree = angle_degree + 10; /* increment the loop index */
    }
}

C Syntax: main

• To get arguments to the main function, use:
  – int main (int argc, char *argv[])

• What does this mean?
  – argc contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here argc is 2:
    unix % sort myFile

  – *argv is a pointer to an array containing the arguments as strings (more on pointers later)

C Syntax: Variable Declarations

• Similar to Java, but with a few minor but important differences
• All variable declarations must appear before they are used (e.g., at the beginning of the block)
• A variable may be initialized in its declaration; if not, it holds garbage!
• Examples of declarations:
  – Correct: {
      int a = 0, b = 10;
      ...
  }
  – Incorrect: for (int i = 0; i < 10; i++)

C Syntax: Flow Control (1/2)

• Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  – if-else
    • if (expression) statement
    • if (expression) statement1
      else statement2
  – while
    • while (expression) statement
    • do statement
      while (expression);
C Syntax: Flow Control (2/2)

• Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  - for
    • for {initialize; check; update} statement
  - switch
    • switch {expression}{
      case const1: statements
      case const2: statements
      default: statements
    }
    • break

C Syntax: True or False

• What evaluates to FALSE in C?
  – 0 (integer)
  – NULL (a special kind of pointer: more on this later)
  – No explicit Boolean type

• What evaluates to TRUE in C?
  – Anything that isn’t false is true
  – Same idea as in scheme: only #f is false, anything else is true!

Agenda

• Scheme vs. Java vs. C
• Administrivia
• Quick Start Introduction to C
• Pointers

Pointers in C

• A pointer is just another kind of value
  – A basic type in C

```
int *ptr;
```

Variable “ptr” is a pointer to an “int”

Address vs. Value

• Consider memory to be a single huge array
  – Each cell of the array has an address associated with it
  – Each cell also stores some value
  – Do you think they use signed or unsigned numbers? Negative address?!

• Don’t confuse the address referring to a memory location with the value stored there

Some of these slides come from “Arrays and Pointers in C” by Alan Cox and T.K. Ng, Rice University. 
www.chevronrix.edu/courses/221/notes/notes03-arrays-pointers.pdf
Pointers

- An address refers to a particular memory location; e.g., it points to a memory location
- Pointer: A variable that contains the address of a variable

Pointer Operations in C

- Creation
  - & variable Returns variable’s memory address
- Dereference
  - * pointer Returns contents stored at address
- Indirect assignment
  - *pointer = val Stores value at address
- Of course, still have … assignment
  - pointer = ptr Stores pointer in another variable

Pointer Examples

- How to create a pointer:
  - & operator: get address of a variable
  ```c
  int *p, x;
  p = &x;
  x = 3;
  p = &x;
  ```
  - Note the ‘&’ gets used 2 different ways in this example. In the declaration to indicate that p is going to be a pointer, and in the printf to get the value pointed to by p.
- How get a value pointed to?
  - "dereference operator": get the value that the pointer points to
  ```c
  printf("p points to %d\n", *p);
  ```

Pointers in C

- If T is a type, T *p declares p a pointer to that type
- You can use p as a pointer to a T
- You can use *p as a T
- *p++ increments p by the size of a T
  - Important because of the way arrays are treated
  - You can make a pointer to any variable
    - If x is any variable, then &x is its address

Pointers in C

- If T is a type, T *p declares p a pointer to that type
- You can use p as a pointer to a T
- You can use *p as a T
- p++ increments p by the size of a T
  - Important because of the way arrays are treated
  - You can make a pointer to any variable
    - If x is any variable, then &x is its address

Pointer Examples

- How to change a variable pointed to?
  - Use the dereference operator * on left of assignment operator
    - ```c
      p = 5;
      *p = 3;
      ```

Pointer Examples

- How to change a variable pointed to?
  - Use the dereference operator * on left of assignment operator
    - ```c
      int i1;
      int i2;
      int *ptr1;
      int *ptr2;
      i1 = 1;
      i2 = 2;
      ptr1 = &i1;
      ptr2 = ptr1;
      *ptr1 = 3;
      i2 = *ptr2;
      ```
Pointer Examples

```c
int int1 = 1036; /* some data to point to */
int int2 = 8;
int *int_ptr1 = &int1; /* get addresses of data */
int *int_ptr2 = &int2;
*int_ptr1 = int_ptr2;
*int_ptr1 = int2;
```

What happens?

Type check warning: int_ptr2 is not an int

int1 becomes 8

Pointers and Parameter Passing

- Java and C pass parameters “by value”
  - Procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```c
void addOne (int x) {
    x = x + 1;
}
int y = 3;
addOne(y);
y remains equal to 3
```

Pass By Reference

- In C, the default passing strategy is pass by copy
- To pass by reference, we use pass by copy – because in C, everything is pass by copy
- Value that we pass by copy is address of the actual argument: we achieve the through the address operator &
- In C, pass by reference is actually pass by copy – because you copy the address

Getting Pass-by-Reference

```c
void set_x_and_y(int *x, int *y) {
    *x = 1001;
    *y = 1002;
}
void f(void) {
    int a = 1;
    int b = 2;
    set_x_and_y(&a, &b);
}
```
Pointers

- Pointers are used to point to any kind of data (int, char, a struct, etc.)
- Normally a pointer only points to one type (int, char, a struct, etc.).
  - void * is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs, and security issues, and other bad things!

Peer Instruction Question

```c
void main(); {
  int *p, x=5, y; // init
  y = *p = &x + 1;
  int z;
  flip-sign(p);
  printf("x=%d,y=%d,p=%d\n",x,y,p);
}
flip-sign(int *n){*n = -*n)
```

How many syntax + logic errors in this C code?

<table>
<thead>
<tr>
<th>Errors</th>
<th>Red: 1</th>
<th>Orange: 2</th>
<th>Green: 3</th>
<th>Yellow: 4</th>
<th>Pink: 5</th>
</tr>
</thead>
</table>

More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- Local variables in C are not initialized, they may contain anything (aka “garbage”)
- What does the following code do?
  ```c
  void f()
  {
    int *ptr;
    *ptr = 5;
  }
  ```

Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it’s easier / faster / etc. to pass a pointer than the whole thing
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
  - Most problematic with dynamic memory management— which you will know by the end of the semester, but not for the projects (there will be a lab later in the semester)
  - Dangling references and memory leaks

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn’t produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today’s compilers produce much better code, so may not need to use pointers
  - Compilers even ignore hints since they do it better!

FYI—Update to ANSI C

- “C99” or “C9x” standard
  - gcc -std=c99 to compile
- References
  - http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html
- Highlights
  - Declarations in for loops, like Java
  - Java-like // comments (to end of line)
  - Variable-length non-global arrays
  - <inttypes.h>: explicit integer types
  - <stdbool.h>: for boolean logic types and definitions
And In Conclusion, ...

- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - `*` "follows" a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers
- C is an efficient language, but leaves safety to the programmer
  - Array bounds not checked
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs