Two’s Complement Review

• Suppose we had 5 bits. What integers can be represented in two’s complement?
  a) -32 to +31
  b) -31 to +32
  c) 0 to +31
  d) -16 to +15
  e) -15 to +15
  f) -15 to +16

Agenda

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

- Spring 2011 statistics
  - 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
- On CS61C class website
  - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- Key C concepts: Pointers, Arrays, Implications for Memory management

C vs. Java

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Language</td>
<td>Function Oriented</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>Programming Unit</td>
<td>Function</td>
<td>Class / Abstract Data Types</td>
</tr>
<tr>
<td>Compilation</td>
<td>Machine dependant assembly</td>
<td>Machine independent bytecode</td>
</tr>
<tr>
<td>Execution</td>
<td>Loads and executes program</td>
<td>JVM interprets bytecode</td>
</tr>
</tbody>
</table>
| Hello, world   | `include<stdio.h>`
  `int main(void) {
    printf("Hello\n");
    return 0;
  }` | `public class HelloWorld {
    public static void main(String[] args) {
      System.out.println("Hello");
    }
  }` |
| Memory management | Manual (malloc, free) | Automatic (garbage collection) |

Compilation: Overview

- C compilers map C programs into architecture-specific machine code (string of 1s and 0s)
  - Unlike Java, which converts to architecture independent byte code
  - Unlike most Scheme environments, which interpret the code
- These differ mainly in exactly when your program is mapped to low-level machine instructions
- 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
Typed Variables in C

- You have to declare the type of data a variable will hold.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>signed integer</td>
<td>5, -12, 0</td>
</tr>
<tr>
<td>short int</td>
<td>smaller signed integer</td>
<td>'a', 'd', 'r'</td>
</tr>
<tr>
<td>long int</td>
<td>larger signed integer</td>
<td>0.0, 1.618, -1.4</td>
</tr>
<tr>
<td>char</td>
<td>single text character or symbol</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>floating point non-integer numbers</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>greater precision FP number</td>
<td></td>
</tr>
</tbody>
</table>

- Integer sizes are machine dependant!
- Common size is 4 bytes, but can’t ever assume this.
- Can add “unsigned” before int or char type for unsigned
  eg unsigned int

Characters

- How do we encode characters?
- Relevant question: How many different characters do we want to encode?
  - 'a'-'z', 'A-Z', '0-9'.
  - What else?

Big Idea – Bits can represent anything!

- Say we have 0b 0110 0011
  - Interpret it as a char — 'c'
  - Interpret it as a (signed or unsigned) integer: 99
- 0b 1111 1111
  - Interpret it as a char: (not a valid char)
  - Interpret it as a signed int: -1
  - Interpret it as an unsigned int: 255

- It all depends on how the programmer looks at them.
- Just wait until Lecture 8, MIPS Instruction Formats, for an incredible example of this.

Types in C

- C is a “weakly” typed language. You can explicitly typecast from any type to any other.
- int i = -1;
  if(i < 0) printf("This will print\n");
  if((unsigned int) i < 0) printf("This will not print\n");
- Can be seen as changing the “programmer’s perspective” of the variable.
- Can typecast anything, even if it doesn’t really make sense:
  - struct node n; //We will see structs in a few slides
  - int j = (int) n;
  - More freedom, but easier to shoot yourself in the foot.

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
  - Just think of this as saying that no value will be returned
  - Also necessary to define types for values passed into a function
- Declaring the “prototype” of a function allows you to use it before the function’s definition.
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!

Structs in C

• Way of defining compound data types.
• A structured group of variables, possibly including other structs

```c
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;
song1.lengthInSeconds = 213;
song1.yearRecorded    = 1994;

Song song2;
song2.lengthInSeconds = 248;
song2.yearRecorded    = 1988;
```

Consts and Enums in C

• Constant is assigned a value once in the declaration; value can’t change until the program is restarted

```c
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

• C Language Overview
• Administrivia
• C Syntax and Control Flow
• Break
• Pointers
• Summary

Administrivia

• TA office hours have been decided. For this week
  – Alvin – MW 12-1. Alcoves
• HW1 will be posted today.
Actual C Code

```c
#include <stdio.h>
#define REPEAT 5

int main(int argc, char *argv[]) {
    int i;
    int n = 5;
    for (i = 0; i < REPEAT; i = i + 1) {
        printf("hello, world\n");
    }
    return 0;
}
```

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)

- `for`
  - `for (initialize; check; update) statement`
- `switch`
  - `switch (expression){
    case const1: statements
    case const2: statements
    default: statements
  }
  - `break`

Agenda

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

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

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

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

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
```
Pointers and Parameter Passing

• How can we get a function to change a value?

```c
void addOne (int *p) {
    *p = *p + 1;
}
int y = 3;
addOne(&y);
y is now equal to 4
```

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>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>&gt;4</td>
</tr>
</tbody>
</table>

Peer Instruction Answer

```c
#include <stdio.h>
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?

SIX

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;
  }
  ```
• 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 to 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

Pointers in C
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!

Has there been an update to ANSI C?

- Yes! It’s called the “C99” or “C9x” std
  - You need “gcc -std=c99” to compile
- References
  - http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html
- Highlights
  - Declarations in for loops, like Java (#15)
  - Java-like // comments (to end of line) (#10)
  - Variable-length non-global arrays (#33)
  - <inttypes.h> explicit integer types (#38)
  - <stdbool.h> for boolean logic def’s (#35)

And In Conclusion, ...

- Bits can represent anything!
  - chars, integers, floating point, ...
- C is an efficient language, but leaves safety to the programmer
  - Weak type safety
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs
- 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