Announcements

- Program #5
  - is due Today
- Final
  - Monday 4-6 in J. M. Patterson 3201
- Pickup old assignments in my office
  - Midterms (including re-grades)
  - Programming assignments

Review Lecture

- Final covers all material from the class
- Today's lecture is set of highlights
  - Also, things not in today's review will likely be on the final!

Why Study Low-level Programming?

- Provides Understanding of How Things Work
  – Compilers, Processors, Data Structures
- Allows access to Hardware when required
  – Writing device drivers
  – Creating Operating Systems
- Can be faster
  – If you are careful and good
  – BUT can also be:
    - Slower – you write low level routines where faster ones exist
    - Dangerous – easier to have software that
      – Crashes program (or OS)
      – Fails in odd ways (memory corruption)
- It's Fun!
  – Or at least I think so

The “C” Programming Language

- Widely Used for System Programming
  - Writing Operating Systems and Compilers
- “Middle Aged” Language – created in late 1970’s
  - Reflects lessons learned from Fortran, COBOL
- Goals
  - Designed to allow low-level programming
  - Small language
    - Syntax is relatively simple
- Major Differences from JAVA
  - Explicit Memory Allocation and Deallocation
  - Procedural rather than object oriented
  - *Compiles* directly to machine code (rather than byte codes)

Preprocessor

- Not everything is in one file
  - Definitions of routines are in separate files
- Function *Prototypes*
  - Define name, parameters, and return types
  - Sort of like how methods are defined in class definitions
  - Example: int myFunc(int a, int b);
- #include
  - provides ability to include definitions from other files
  - Generally only include prototypes from external files
    - Actual code is kept in a separate file
    - Files are generally compiled seperately

Formatting Your program

- Comments
  - Must be surrounded by /* and */
  - May span multiple lines
    - Common bug is to forget to close a comment
    - Can not be nested
- White Space
  - Blank lines before a new function definition
  - Blank lines within a function to separate different steps
- Indentation
  - Generally corresponds to nesting level of program

Stored Program Computer

- A Program:
  - Consists of a sequence of operations
  - Is loaded into a computer's memory to execute
- Standardize set of allowed operations, called *Instructions*
  - each instruction performs a specific operation
  - Instructions can
    - read or change memory
    - compute a value
    - read or write output
  - What instruction to execute next:
    - normally it is the next instruction
    - provide control to select alternate next instruction
- Program and data are both stored in memory

Completeness of Instructions

- Need to ensure that the computer can execute any computable function.
  - more instructions may make the machine faster
    - but not more powerful
- Arithmetic
  - add, complement (negate)
  - AND and NOT (provides NAND which is enough)
- Data movement instructions
  - Load from memory, Store to memory
- Control instructions
  - check values of memory or other locations
  - provide a way to change the next instruction to execute
- Input/Output Instructions

What do Instructions Look Like?

- For Project #1, instructions:
  - are 32 bits long
  - contain three parts:
    - 4 bit Opcode (indicates what to do)
      - implies there are at most 16 operations
    - 3 registers (temporary values to use)
    - 16 bits of memory address
      - at most 65536 words of memory

Purpose	**Opcode**	**Register$_1$**	**Register$_2$**	**Register$_3$**	**Memory**
Size (bits)	**4**	**4**	**4**	**4**	**16**

Make: A Tool to Compile Programs

- Problem: Compiling programs is tedious
- Solution: Automate it!

- Done in Unix by the make command
  - Uses a file called Makefile
    - A makefile is a file (script) containing :
      - Project structure (files, dependencies)
      - Instructions for files creation
- Make is not limited to C programs

Adapted from: www.cs.tau.ac.il/~dcor/Software1/Makefile.**ppt**

Function Arguments

- Comma separated list of values
- All parameters are passed by value
  - called function can modify values
  - arrays are passed by the address of their 0th element
    - modifying elements of array seen by calling function
- Can pass address of variable to change values

```
int foo(int *a) {
    *a = 3;
}

int x;
foo(&x);
```

struct, union, and typedef

- Often useful to use typedef and struct together:

```
typdef struct {
    int a;
    char b;
} Simple;
```

- Declarations then look like:

```
Simple x;
Simple y[20], z;
```

- Header Files
  - If structure used by more than one file, put it in a header file.
  - Use #include to include definition in each file where used.
- Unions
  - Rather than storing each field, only stores **one** of the fields
  - Syntax: union tag { member-list } variable-list

Bit Fields

- Part of the Power of C is control over data layout
  - If you need a field with exactly 13 bits, you can define it
  - Useful for:
    - defining fields that interact with hardware
    - managing pre-defined file formats
    - saving every last bit of space
  - Syntax: type variable:size;
- Examples:

```
int foo:13;
unsigned int foo:4;
typedef struct {                /* from project #1 */
   unsigned int opCode:4;
   unsigned int r1:4;
   unsigned int r2:4;
   unsigned int r3:4;
   unsigned int address:16;
} instruction;
```

Bit Shift Operator

- C has operators to bit shift numbers
  - number of bits changed depends on size of variable

- Left Shift (number << xxx)
  - Move each bit of number to the left by xxx bit positions
  - Leftmost xxx bits discarded
  - Rightmost xxx bits gets 0
- Right Shift (number >> xxx)
  - Move each bit of number to the right by xxx bit positions
  - Rightmost xxx bits discarded
  - Leftmost xxx bits gets 0 **or replicate sign bit**
    - for unsigned gets 0
    - for signed, its **implementation dependent**

Type Conversion

- Promotion of char and short
  - in expressions, char and short are promoted to int

    char a, b, c;

    a = b + c;

    - b and c are converted to int, then the sum is truncated
- Arithmetic Conversions
  - can't be performed on different types
    - converted to "higher" type

      long double ← Highest Type

      double

      float

      unsigned long int

      long int

      unsigned int

      int

Precedence

- There is a set of rules about precedence of operator
  - Most important precedence rule:
    - When in doubt, put in () to ensure correct order
  - Order (highest to lowest):
    - ()
    - Function call, subscript, postfix increment/decrement
    - rest of unary operators
    - type conversion
    - Arithmetic operators
    - Relational operators
    - Bit operators
    - Assignment operators
    - ,

Standard I/O Library

- #include <stdio.h>
  - includes prototypes for standard I/O routines
- Most I/O is stream based
  - buffered to allow efficient operations
    - mostly to disks or networks
  - interactive I/O is normally flushed at useful points
    - printf("foo\n");   /* \n causes a flush */
- FILE type
  - used to describe an active I/O connection
  - three default ones supplied on entry to main:
    - stdin - an input stream for default input (often keybaord)
    - stdout - default output stream (often terminal window)
    - stderr - default output stream for errors

I/O Functions

- FILE *fopen(char *name, char *mode)
- int fclose(FILE *fp);
- int fgetc(FILE *stream);
- int fputc(int character, FILE *stream);
- int putchar(int character);
- int ungetc(int character, FILE *stream);
- char *fgets(char *buffer, int bufferSize, FILE *stream);
- **gets: never use this function!**
  - **it leads to buffer overflow problems**
- int fputs(char *buffer, FILE *stream);

Strings

- Zero or more characters followed by NUL
  - not counted as part of string
  - string.h defines prototypes for string routines
- Copying Strings
  - size_t strlen(char const *str);
    - returns count of characters in str
  - char *strncpy(char *dst, char const *src, size_t len);
    - copy src to dst
    - copy until NUL in src or at most len characters
    - pad extra characters will NUL
    - Safety tip: dst[len-1] = '\0'; to force termination
  - char *strncat(char *dst, char const *src, size_t len);
    - append src onto the end of dst
    - always appends NUL ('\0') to end of string

Arrays of Pointers

- int *api[10];
  - An array of 10 pointers to integers;

```
char *keyword[] = {
   "Load",
   "Store",
   "Negate",
   …
};
```

Pointers

- Can declare any variable to be of type pointer
  - are similar to references in Java
  - use * before variable name to create a pointer to the type
    - int *foo;
    - myTypeDef *myPtr;
    - float *bar;
  - The type determines both:
    - what is pointed to
    - the size of the object being pointed to
- Initialization
  - creating a pointer doesn't create the space it points to
  - int *foo;
  - *foo = 3; /* error: haven't defined what it points to */

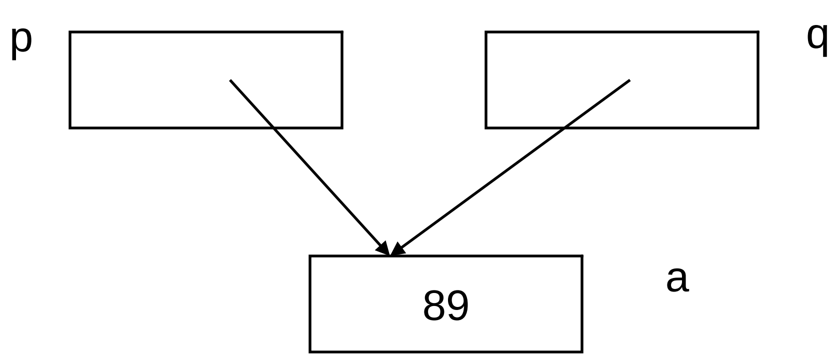
Type Conversion

- **C lets you assign variables of different types**
  - called type conversion
  - useful when dealing with
    - external devices
      - int *fooDev = (int *) 100;
      - we know the foo device is at location 100
    - when type information read from a file is based on file contents.
- **WARNING: This can be very dangerous**
- (type *) variable:
  - int *x;
  - float *y;
  - x = (int *) y;

Pointer Aliases

- Sometimes two pointers point to the same thing\
  - This can be useful
  - This can cause problems:

```
int *p, *q;
int a= 89;
p = &a;
q = p;
free(p);
if (*q) { … }        /* error: *q points to freed space */
```



Heap Memory Management

- Allocating memory:
  - void *malloc(size_t requestedSize);
    - allocate requestedSize bytes of memory
  - void *calloc(size_t requestCount, size_t objectSize);
    - allocate requestCount objects of size objectSize bytes each
    - set the requested memory to zero
  - Type Information
    - malloc/calloc return void *
    - need to convert to desired type using cast operator
- free(void *p);
  - returns space pointed to by p
  - does **not** change p
    - good idea p=NULL; right after deallocation.

Linked Data Structures

- Lists
  - Singly Linked Lists
  - Doubly Linked Lists
- Tree Like
  - Trees
  - Dags

Graphs - Directed

- Examples
  - List of web pages that link to each other
- Nodes can
  - have an arbitrary number of out arcs
  - have an arbitrary number of inbound arcs
- Declarations

```
typedef struct ARC {
    struct NODE *nodePtr;
    struct ARC *next;
} Arc;
typedef struct NODE {
    Arc *outArcs;
    int value;
} Node;
```

Tests are an integral part of the software

- Write Tests as you write software
  - Don't wait until code is done to write tests
  - Sometimes write test code **before** software
  - Test each function/method as you write it
- When the software is "done", tests continue as part of it
  - Need to be able to re-test when code changes
  - Need to validate code on new platforms
    - Sometimes even individual systems
- Types of Testing
  - White Box
  - Black box

Number Representation

- Integers - 2's complement representation
  - To compute a negative value:
    - flip all the bits of the positive value, add 1
  - Allows addition of signed and unsigned numbers
  - Valid range of numbers
    - -2^{n-1} to $2^{n-1}-1$ for n bits
    - Example: 16 bit number -32,768 to 32,767
- Real numbers
  - Each number has two parts
    - mantissa (represents a number between -1 and 1)
    - exponent (designates the position of the decimal point)
    - number = $m\ r^{e}$ where r is the radix
  - Normalization
    - convert to number between -1 and 1
    - if the most significant digit of mantissa is non-zero.

How do we represent real numbers?

- Each number has two parts
  - mantissa (represents a number between -1 and 1)
    - represented as a binary number i.e. 0.1 = 1/2
  - exponent (designates the position of the decimal point)
    - uses normal two's complement form
  - number = $m\ r^{e}$ where r is the radix
    - $6132.789 = +0.6132789 \times 10^{4}$
- Normalization
  - convert to number between -1 and 1
  - if the most significant digit of mantissa is non-zero.

Floating Point Continued

- Computers normally use a radix of 2
- Examples of floating point numbers
  - 1001.11 = .1001110 x 2^4
  - 10.5 = 1010.1 or 10101 x 2^4
  - 17.451 = 111.011100 = 1110111 x 2^3
- IEEE Floating point standard
  - 32 bit floating point (float)
    - 1 sign bit, 8 bits exponent, 23 bits mantissa
  - 64 bit floating point (double)
    - 1 sign bit, 11 bits exponent, 52 bits mantissa
    - most common for real applications
  - 128 bit floating point (quad)
    - 1 sign bit, 15 bits exponent, 112 bits of mantissa

Callbacks

```
typedef int (*compareFunc)(item *a, item * b);

int searchTree(Node *root, Item *target, compareFunc cmp) {
        if (!root) return -1;
        ret = (cmp)(target, root->data);
          if (ret == 0) {
              /* found it */
              return 1;
          } else if (ret < 0) {
              return searchTree(root->left, target, cmp);
          } else {
              return searchTree(root->right, target, cmp);
          }
}
```

Programming Command Line Arguments

- Generally, want arguments to be accepted in any order
  - ./foo -file myfile -help
  - ./foo -help -file myfile
- Solution:
  - Table driven command line parsing
    - Example of a common pattern, table drive parsing
  - Declare a structure
    - fields describe each valid option

```
typedef enum { Flag, StrParam, IntParam };
typedef struct {
    char *name;
    optionTypes options;
    void *parameter;
} option;
```

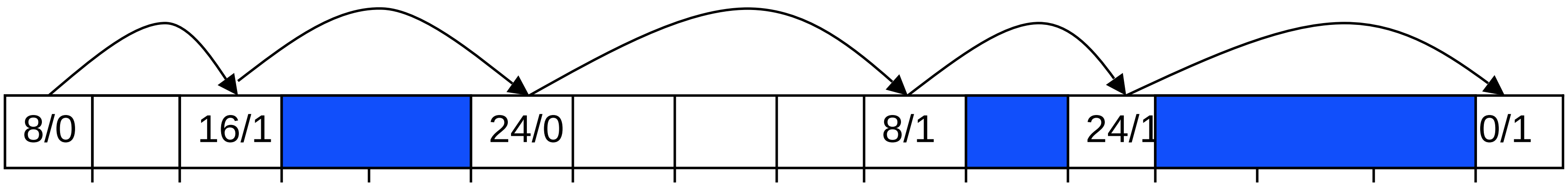
Command Line Arguments Continued

```
int debugFlag, limit;
char *inputFileName;
optionTable = [] {
        { "-debug", Flag, &debugFlag },
        { "-file", StrParam, &inputFileName },
        { "-limit", IntParam, &limit },
        …. /* other options go here */
 };

int optionCount = sizeof(optionTable)/sizeof(option);
```

Simple Data Structure for Heap

- Divide heap into blocks
- Each Block has a 32 bite header
  - 29 bits define size of block (in 8 byte chunks)
  - 1 bit is for free/allocated (1 indicates allocated)
  - 2 bits are un-used
- Can find next block if know the current block
  - add address of current block header to size of block



Garbage Collection

- What is Garbage?
  - memory the program can't get to anymore
  - Example:
    - ptr = malloc(40);
    - ptr = NULL;
- How to detect Garbage?
  - Need to know what is a pointer
    - know what everything is (Java)
    - guess anything that looks like a pointer is one (C)
      - any 32-bit quantity in globals, on stack, is assumed to be a pointer.
      - anything in a memory region pointed to
  - Compute reachability graph
    - Follow all pointers

Representing Arrays

- Arrays (C style)
  - sequence of consecutive memory locations
  - address of array is the starting point for the array
  - subscript defines offset into array
    - location = baseAddr + subscript * sizeof(object)
- Fortran90
  - sequence of logically consecutive bytes
  - array descriptor (runtime data) stores additional information
    - distance between array elements (for each dimension)
    - number of elements (for each dimension)
    - lower bound of array dimension
- Java
  - sequence of objects
  - array descriptor (runtime data) stores
    - type information about what this is an array of
    - length array

Functions and Parameters

- Calling Functions
  - Keep a stack of return addresses for function
  - Dedicate a register to managing the stack
    - called stack pointer (R15)
  - Pick a standard register to save return address in
    - caller always uses this one (R14)
    - callee knows where return address is and saves it
      - push return address onto stack onto stack
- Parameters are like local variables
  - values are only defined for lifetime of function
  - but, they have initial values
  - Implementation:
    - put some parameters into registers
    - registers are faster than memory

Typical Compiler Optimizations

- Code motion
  - previous example
- Loop Unrolling
  - often faster if a loop body is a bit bigger
    - modern computers can do several instructions at once
  - convert:

```
for (i=0; i < n; i++)
        c[i] = a[i] + b[i];
```

  - to:

```
for (i=0; i < n; i+=2) {
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] +b[i+1];
}
```

- Dead-Code Elimination
  - if compiler can infer code is never run, remove it

Using JNI Functions

- Declaring Functions
  - Declare like a normal method, but add “native” keyword
  - public native void displayHelloWorld();
- Loading the Code
  - Use loadLibrary call
  - Making it static forces it to happened automatically
  - System.loadLibrary("hello")
- Invoke Native Method
  - new HelloWorld().displayHelloWorld();