

Announcements

- Program #6
 - due TODAY
- Final
 - **Saturday, May 12 4-6 in PHY 1412**

Review Lecture

- Final covers all material from the class
 - Comprehensive
 - relatively equally weighted between 1st, 2nd and after 2nd exam
- Today's lecture is set of highlights
 - Be aware that there are things not in today's review which will likely also be on the final!

Why Study Low-level Programming? and Why Study the C language?

- Provides Understanding of How Things Work
 - Compilers, Processors, Data Structures
- Allows access to Hardware when required
- Allows efficiency and control
 - Writing device drivers and operating systems
 - If you are careful and good it will be faster
 - BUT could also be Dangerous or Slower
- Widely Used for System Programming
- “Middle Aged” Language – created in late 1970’s
- C’s design goals Goals
- Major Differences from JAVA
 - Explicit Memory Allocation and Deallocation
 - Procedural rather than object oriented
 - *Compiles* directly to machine code (rather than byte codes)

Phases of Compilation

- Preprocessor
 - Not everything is in one file
 - Definitions of routines are in separate files
 - Function *Prototypes*
 - Define name, parameters, and return types
 - #include, #if, #endif, #ifndef
 - macros, constants, conditional compilation
- translation (compilation)
 - source to object code
 - of individual files
 - making sure prototypes match definitions and calls
- linking
 - combining object files
 - making sure all parts are there

Readability, Reliability and Efficiency of Your program

- **Readability**
 - Comments
 - Should be surrounded by /* and */
 - May span multiple lines
 - White Space
 - vertical and horizontal
- **Testing**
 - Write Tests as you write software
 - When the software is “done”, tests continue as part of it
 - Types of Testing
 - line coverage, white box and black box
- **Efficiency**
 - Code motion
 - Loop Unrolling
 - Dead-Code Elimination

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**

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**

CMSC 212 – F05 (lect 28)

7

What do Instructions Look Like?

- **For Project #1, instructions:**
 - are 32 bits long
 - contain three parts

Purpose	Opcode	Register ₁	Register ₂	Register ₃	Memory
Size (bits)	4	4	4	4	16

- **Extension Later – Maintaining the Stack**
 - Calling Functions
 - for calls to the same function from other places in the program
 - for recursion
 - parameter passing to functions
 - return values of functions

CMSC 212 – F05 (lect 28)

8

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:

```
typedef 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
```

CMSC 212 – F05 (lect 28)

13

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
 - , (comma operator)

CMSC 212 – F05 (lect 28)

14

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 keyboard)
 - `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 with 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

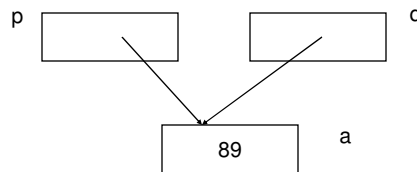
- **C lets you assign variables of different types**
 - called type conversion
 - useful when dealing with
 - external devices
 - `int *fooDev = (int *) 100;`
 - we know the foo device is at location 100
 - when type information read from a file is based on file contents.
- **WARNING: This can be very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

- Singly Linked Lists
- Doubly Linked Lists
- Trees
- Hash
 - simple
 - linked
- Graphs
 - Directed vs. undirected
 - Cyclic vs Non-Cyclic
 - DAGS – directed acyclic graphs
- How they are stored
- Algorithms for traversal

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;
```

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.

CMSC 212 – F05 (lect 28)

25

Floating Point Continued

- **Computers normally use a radix of 2**
- **Examples of floating point numbers**
 - $1001.11 = .1001110 \times 2^4$
 - $10.5 = 1010.1$ or 10101×2^4
 - $17.451 = 111.011100 = 1110111 \times 2^3$
- **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

CMSC 212 – F05 (lect 28)

26

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);
    }
}
```

CMSC 212 – F05 (lect 28)

27

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

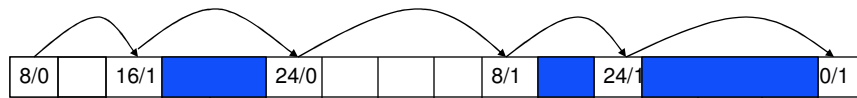
```
typedef enum { Flag, StrParam, IntParam };
typedef struct {
    char *name;
    optionTypes options;
    void *parameter;
} option;
```

CMSC 212 – F05 (lect 28)

28

Simple Data Structure for Heap

- Divide heap into blocks
- Each Block has a 32 bit 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



CMSC 212 – F05 (lect 28)

29

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

CMSC 212 – F05 (lect 28)

30

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
 - $\text{location} = \text{baseAddr} + \text{subscript} * \text{sizeof}(\text{object})$
- **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

How Processors Spend their time

- **Not all instructions take the same amount of time**
- **Some instructions are more expensive**
- **Caches**
 - processors have caches to store values
 - each cache item stores several values (called line size)
 - The same instructions may take different amounts of time
 - misses from the cache can be 10-100x slower
 - Goal is to re-use same cache items multiple times
 - Consider:

```
for (i=0; i < limit; i++) {  
    for (j=0; j < limit; j++) {  
        for (k=0; k < limit; k++)  
            a[i,j] += b[i,k] * c[k,j];  
    }  
}
```

Issues in Conducting Measurements

- **Number of Runs**
 - A single run is not sufficient
 - many things go on in a computer
 - operating system functions
 - other programs running
 - Multiple runs provide increased accuracy
 - take mean of K fastest runs
- **Workload**
 - what data is the program given for measurement runs?
 - Does it look like a "typical" use of the program
 - Many algorithms might look good if the measured workload
 - is too small - $O(n^2)$ algorithms similar to $O(n)$
 - is too large - not typical of usage pattern

Sources of Performance Problems

- **Too Small of I/O Ops**
 - calling read one or a few characters at a time
- **Left debugging printf's in program**
- **Poor Basic Algorithms**
 - used $O(n^2)$ for large n and frequent operations
- **Algorithms compose poorly**
 - AVL tree - implementing deleteTable via deleteItem
 - forces frequent rebalancing
- **Cache memory performance**
 - Use each cache item once

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 happen automatically
 - `System.loadLibrary("hello")`
- **Invoke Native Method**
 - `new HelloWorld().displayHelloWorld();`