Announcements

- Midterm
  - none, enough work as it is
- Welcome to the next phase of the course (cue dramatic music) later today
- C

- not B or A
Today

- last little perl stuff
- intro to c

- reading:
  - perl object and packages

Advanced topics

- Multi threading
  - Fork processes
    - this is not an eating utensil
  - Process space
    - not parking space

- Communication between programs
  - Pipes
  - Sockets
Testing Environment

- One word on testing in the real world:
  - need as much as you can get!

- Large projects
- Bugs cost time and money
- Bugs hurt morale
- Human are programmers...humans make mistakes
- Formula for this actually

Automated testing

- Humans hate testing...
- Fast verification that new feature has not broken code
- Verify all code on a regular basis
- No grumble if test to rerun test 😊
packages

- There are packages out there (Test::Simple and Test::Harness) to automatically run tests
- Verify what happens on good/bad input
- Verify variables/method behavior
- Usually .t files have tests

Advanced Random Stuff

- $| = 1
  - flushes the output to make sure you see what is being printed right away
- Can choose your own delimiters when matching
- m#shlomo#
- s!cheese!milk!
- s{something}(else)
Slicing

- similar to ranges, can fetch set of values from hash by preceding hash variable with @ sign

```perl
%phonebook;
#do bunch of reads/inserts
@numbers = @phonebook($n1, $n2, $n3);

@phonebook($n1, $n2) = (718,516);
```

tieing

- way of connecting a class with a variable

- example counting scalar
  - want to increment a scalar every time we read it
  - either can create a specific class
  - or use a normal scalar, but tie it to the class
  - list of specific sub you need to override to make it work
Perl Skills

- We’ve covered a lot of perl skills since beginning the course
- What you should have
- What we haven’t covered
- Where to take perl from here

Shift gears

- Will now start C component of the course
- This is not C#
  - any ideas why it is called C#
C Phase

- Intro to C
  - Background
  - Compiling
  - Basic data structures
  - Basic I/O
  - Types conversion
  - Loops
  - Branching

Roadmap

- How this all fits together
  - We covered perl (duct-tape programming)
  - CGI programming USING perl
  - Will now move to c, which is a more low level programming language
  - Will learn to work with c, and then CGI+c
  - Then CGI+perl+c etc
  - Get to use the best of any programming language in a project
Why Learn C?

- C provides stronger control of low-level mechanisms such as memory allocation, specific memory locations.
- C performance is usually better than Java and usually more predictable (very task dependant).

Why Learn c continued

- Java hides many details needed for writing code, but in C you need to be careful because:
  - memory management responsibility left to you
  - explicit initialization and error detection left to you
  - generally, more lines of (your) code for the same functionality
  - more room for you to make mistakes

- Most older code is written in C (if you are lucky) might need there skills if you will be hired to upgrade or interface with in place tech.
Background

C
- Dennis Ritchie in late 1960s and early 1970s
- Systems programming language
- Goal: make OS portable across hardware platforms
- Not necessarily for real applications—could be written in Fortran or PL/I

Background II

C++
- Bjarne Stroustrup (Bell Labs), 1980s
- object-oriented features

Java
- James Gosling in 1990s, originally for embedded systems
- object-oriented, like C++
- ideas and some syntax from C
Background III

- C is early-70s, procedural language

- C advantages:
  - direct access to OS primitives (system calls)
  - more control over memory
  - fewer library issues—just execute

- C disadvantages:
  - language is portable, but APIs are not
  - no easy graphics interface
  - more control over memory (i.e., memory leaks)
  - pre-processor can lead to obscure errors

C vs Java

- Java program
  - collection of classes
  - class containing main method is starting class
  - running java StartClass invokes StartClass.main method
  - JVM loads other classes as required

- C program
  - collection of functions
  - one function – main() – is starting function
  - running executable (default name a.out) starts main function
  - typically, single program with all user code linked in— but can be dynamic libraries (.dll, .so)
C vs Java …Running

- Java programs are compiled and interpreted:
  - `javac` converts `foo.java` into `foo.class`
  - class file is not machine-specific—it is byte code
  - byte code is then interpreted by JVM
  - and each JVM is machine-specific

- C programs are compiled into object code and then linked into executables
  (to allow for multiple object files and libraries to be compiled together into one program):
  - `gcc` compiles `foo.c` into `foo.o` and then links `foo.o` into `a.out`
  - you can skip writing `foo.o` if there is only one object file used to create your executable
  - `a.out` is executed by OS and hardware
  - the C compiler is machine-specific, creating code that executes on specific OS/hardware

Outline

- Working with C
  - Compiling
  - Basic data structures
  - Basic I/O
  - Types conversion
  - Loops
  - Branching
  - Compiling
  - Control flow
  - Arrays
  - Pointers
  - strings
  - string library
  - string tokenizing
  - Memory allocation intro

- Reading
  - Deitel book: posted online, will update with more information
Code Example

- Java

```java
public class hello {
    public static void main(String[] args) {
        System.out.println("hello world!");
    }
}
```

- C

```c
#include <stdio.h>
int main() {
    printf("hello world!");
    return 0;
}
```

- `#include <stdio.h>` to include header file stdio.h
- `#` lines processed by pre-processor
- No semicolon at end of pre-processor lines
- Lower-case letters only—C is case-sensitive
- `int main() { ... }` is the only code executed
- `printf( " /* message you want printed */ " );`
- `\n = newline, \t = tab`
- `\ (escape character) in front of other special characters`
Brief Overview

- For the c section of the course, here are some tips
  1. Write your course code
  2. Try to compile
  3. Debug compile bugs, goto step 1
  4. Try step 2 again
  5. Run debugger to catch run time bugs
  6. Run memory profiler to catch memory bugs
  7. Have running product
  8. Add one last cool feature and jump to step 3 😊

or use eclipse and do everything at once 😊

How to make your c code run - manual

- gcc is the C compiler we’ll use in this class
- it’s a free compiler from Gnu (i.e., Gnu C Compiler)
- gcc translates C program into executable for some target machine platform
- default file name a.out
- behavior of gcc is controlled by command-line switches
- Will create files to help in compiling out programs

$ gcc hello.c
$ ./a.out
hello world!
Compiling your program

two-stage compilation
1. pre-process and compile: gcc -c hello.c
2. link: gcc -o hello hello.o

linking several modules:
> gcc -c a.c
   == a.o
> gcc -c b.c
   == b.o
> gcc -o hello a.o b.o
   == hello

using a library, for example the “math” library (libm):
> gcc -o calc calc.c -lm

C control flow

- blocks are enclosed in curly brackets
- functions are blocks
- main() is a function
- blocks have two parts:
  - variable declaration (“data segment”)
  - code segment
- in C, variables have to be declared before they are used
- initializations can occur at the end of the declaration section, but before the code section
Break down of running program

- A program is a collection of functions
- The function named main is launched first
- When main ends, your program is done
  - or can crash the system earlier 😊

First C program

/* First C program */

int main(void){

    printf("Hello Everyone\n");

    return 0;
}


compile

- gcc –o test simple.c
- ./test

Steps to running program

- Write code
  - Platform independent (for the most part)
- Preprocess the code
  - Understand and reinterpret parts
- Compile the code generate object files
  - Turn it into machine code, use optimizers
- Link object files to executable
- Load executable to running code
Your Own Environment

- **Windows:**
  - can use cygwin (free) with gcc (free)
  - gcc 3.4.4.1

- **Mac**
  - get gcc

- **Unix:**
  - cunix has it already
  - gcc 4.1.1

- **Eclipse:**
  - need to make sure you have c/cpp plugin

Split personalities

- In c and cpp normal to divide definition of code (header files .h) and working code (.c files)

- So will have function declaration
  - `int foo();`

- And function definitions
  - `int foo(){. . . . . }`
A macro

- A macro is a section of code, which has been given a name
- Can do a lot with macros

Important to understand:
- When you use the name, the preprocessor will replace it with the code contents
- Compiler only sees changed code

C pre-processor

- the C pre-processor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it for the compiler
  - pre-processor directives start with # at beginning of line
- used to:
  - include files with C code (typically, "header" files containing definitions; file names end with .h)
  - define new macros
  - conditionally compile parts of file (later – not today)
- gcc -E shows output of pre-processor
- Can be used independently of compiler
Example

- `#define BUFFER_SIZE 1024`
- Convention to use upper case
- Will be replaced exactly with the stuff after the name
- `int x = BUFFER_SIZE;`
- Why would this be useful?

pre-processor II

- file inclusion
  - `#include "filename.h"
  - `#include <filename>
  - inserts contents of filename into file to be compiled
  - "filename.h" relative to current directory
  - `<filename>` relative to `/usr/include` or in default path (specified by `-I` compiler directive); note that file is named verb+filename.h+

- import function prototypes (in contrast with Java import)
  - examples:
    - `#include <stdio.h>
    - `#include "mydefs.h"
    - `#include "/home/shlomo/programs/defs.h"`
Pre-processor cont.

#define name const-expression
#define name (param1,param2,...) expression
#undef symbol

- replaces name with constant or expression
- textual substitution
- symbolic names for global constants
- in-line functions (avoid function call overhead)
- type-independent code

#define MAXLEN 255

Example

#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i)) { ...}

- becomes
  if ((i) < 100) { ...}

- Caution: don’t treat macros like function calls
#define valid(x) ((x) > 0 && (x) < 20)
#define valid(x) ((x) > 0 && (x) < 20)

- is called like:
  if (valid(x++)) { ...}
- and will become:
  valid(x++) -> ((x++) > 0 && (x++) < 20)
- and may not do what you intended...
- conditional compilation
- pre-processor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code
- can be used to comment out chunks of code—bad!
- (but can be helpful for quick and dirty debugging :-)

**example:**
```c
#define OS linux
...
#if OS == linux
puts( "Wow you are running Linux!" );
#else
puts( "why are you running something else???" );
#endif
```

- **ifdef**
  - for boolean flags, easier:
    ```c
    #ifdef name
code segment 1
#else
code segment 2
#endif
    ```
  - pre-processor checks if name has been defined, e.g.:
    ```c
    #define USEDB
    ```
  - if so, use code segment 1, otherwise 2
Comments

/* any text until this */

- convention for longer comments:
  /*
   * AverageGrade()
   * Given an array of grades, compute the average.
   * /

- Don’t get carried away with comment boxed
- **** boxes - hard to edit, usually look ragged.

Where to begin?

- Lets talk about what are the primitive data types:
Data Types

- Very important when trying to resource memory/cpu
- float has 6 bits precision
- double has 15 bits precision
- Range can change depending on machine type, generally int is native to the machine type

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>

Types II

- unsigned char
- unsigned short
- unsigned int
- unsigned long

- Byte size is the same, but can now have greater range
- Can look at /usr/include/limits.h
Use in functions

- in strict c mode:
  - Variables must be declared in the beginning of the function to be used

- Common mistake: forgetting to declare at top of function

Intro arrays

- An array is a group of memory locations with the same name and type
- To get to a particular element in the array we need
  - data type
  - name
  - Length or position
- Array length can be determined:
  - statically— at compile time (when we code)
    - e.g., char str1[10];
  - dynamically— at run time (more on this later)
    - e.g., char *str2;
Defining a variable is called “allocating memory” to store that variable.

Defining an array means allocating memory for a group of bytes.

Individual array elements are indexed:
- starting with 0
- ending with length -1

Indices follow array name, enclosed in square brackets ([ ])
e.g., name[25]

Initializing the arrays are your problem:

```c
int a[3];
```
```
X = a[1];
```

Bound checking is your problem:

```c
printf("%d",a[100]);
```

```c
printf("%d",a[100]);
```
int C[5]

-45 0 17 4 82

We can say for example
int varX = C[4] / 5;

Declarations:
int b[100], v[3];

More arrays
- Can also create arrays in the following manners
  1. int a[] = {1, 2, 3};
  2. int b[3] = {6, 3, 7};
  3. int n[10] = {0};
  - In general: you need to initialize the array elements, 3 is a trick case.
Function

- **Declaration:**
  - Return-type function-name (parameters if any);

- **Definition:**
  - Return-type function-name (parameters if any){
    declarations

    statements

  }
Library

- Access libraries using the include statement
- Generally include header files
- Compiler links them automatically
- Example:
  - Standard input/output: stdio.h
  - To look up information use the man page:
    man stdio
stdio.h

- Access stdio functions by
  - using `#include <stdio.h>`
  - compiler links it automatically

- defines stdin, stdout, stderr
- use for character, string and file I/O (later)

- Example: `printf`

printf Function

- The way printf works is it takes a format to print out and then the data to add to the format

- One or more of the following:
  - `%[flags][width][.precision][modifiers]type`
    - “%d”
      - Means a single number
    - “%d %d %d”
      - ??
- `printf ("%d %d", a, b);`

### `stdio.h`: `printf`, type specifier

- `int printf(const char *format, ...)` formatted output to stdout

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Format Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>Character</td>
<td><code>sample</code></td>
</tr>
<tr>
<td><code>d</code> or <code>i</code></td>
<td>Signed decimal integer</td>
<td><code>392</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td>Scientific notation (mantissa/exponent) using e character</td>
<td><code>3.9265e2</code></td>
</tr>
<tr>
<td><code>E</code></td>
<td>Scientific notation (mantissa/exponent) using E character</td>
<td><code>3.9265E2</code></td>
</tr>
<tr>
<td><code>f</code></td>
<td>Decimal floating point</td>
<td><code>392.65</code></td>
</tr>
<tr>
<td><code>g</code></td>
<td>Use shorter %e or %f</td>
<td><code>392.65</code></td>
</tr>
<tr>
<td><code>G</code></td>
<td>Use shorter %E or %f</td>
<td><code>392.65</code></td>
</tr>
<tr>
<td><code>o</code></td>
<td>Signed octal</td>
<td><code>610</code></td>
</tr>
<tr>
<td><code>s</code></td>
<td>String of characters</td>
<td><code>sample</code></td>
</tr>
<tr>
<td><code>u</code></td>
<td>Unsigned decimal integer</td>
<td><code>7235</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td>Unsigned hexadecimal integer</td>
<td><code>7fa</code></td>
</tr>
<tr>
<td><code>X</code></td>
<td>Unsigned hexadecimal integer (capital letters)</td>
<td><code>7FA</code></td>
</tr>
<tr>
<td><code>p</code></td>
<td>Address pointed by the argument</td>
<td><code>8800:0000</code></td>
</tr>
<tr>
<td><code>n</code></td>
<td>Nothing printed. The argument must be a pointer to integer where the number of characters written so far will be stored.</td>
<td></td>
</tr>
</tbody>
</table>
printf flags

- %[flags][width][.precision][modifiers]type

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left align within the given width. (right align is the default).</td>
</tr>
<tr>
<td>+</td>
<td>Forces to precede the result with a sign (+ or -) if signed type. (by default only - (minus) is printed).</td>
</tr>
<tr>
<td>Blank</td>
<td>If the argument is a positive signed value, a blank is inserted before the number.</td>
</tr>
<tr>
<td>#</td>
<td>Used with o, x or X type the value is preceded with 0, 0x or 0X respectively if non-zero.</td>
</tr>
<tr>
<td></td>
<td>Used with e, E or f forces the output value to contain a decimal point even if only zeros follow.</td>
</tr>
<tr>
<td></td>
<td>Used with g or G the result is the same as e or E but trailing zeros are not removed.</td>
</tr>
</tbody>
</table>

example

```c
int class_size = 35;
char class_name[15] = “3157 adv prog”;

printf(“Welcome to our test program\n”);

printf( “the %s class size is %d”,
    class_name, class_size);
```
int array

1. #include <stdio.h>
2. #define MAX 6
3. int main( void ) {
4.   int arr[MAX] = { -45, 6, 0, 72, 1543, 62 };
5.   int i;
6.   for ( i=0; i<MAX; i++ ) {
7.     printf( "[%d] = %d \n", i, arr[i] );
8.   }
9. } /* end of main() */

stdio.h: scanf

- int scanf(const char *format, ...)

32
Example: scanf/printf

```c
#include <stdio.h>
void main( void ) {
    int n = 0; /* initialization required */
    printf( "how much wood could a woodchuck chuck\n" );
    printf( "if a woodchuck could chuck wood?" ); /* prompt user */
    scanf( "%d", &n ); /* read input */
    printf( "the woodchuck can chuck %d pieces of wood!\n", n );
    return;
}
```

output

```
$ a.out
how much wood could a woodchuck chuck if a woodchuck could chuck wood? 12345
the woodchuck can chuck 12345 pieces of wood!
```
Loops

- loops in C are just like in Java
  - there are 2 methods for looping:
    - counter-controlled (loop for a fixed number of times)
    - sentinel-controlled (loop while a condition is true)
  - there are 3 statements for implementing the 2 methodologies:
    - for
    - while
    - do...while
  - as always: beware the infinite loop!
  - Ctrl-C interrupts your executing C program

Branching

- branching in C is just like in Java
  - there are 2 ways to do branching:
    - if/else
    - switch
  - questions:
    - which is more flexible and powerful?
    - one can always be translated into the other, but not the other way around— which is which?
Pointer power

- Variables that contain memory addresses as their values
- Data types we've learned about in C use direct addressing
- Pointers facilitate indirect addressing
- Declaring pointers:
  - pointers indirectly address memory where data of the types we've already discussed is stored (e.g., int, char, float, etc.)
  - declaration uses asterisks (*) to indicate a pointer to a memory location storing a particular data type
  - Called dereferencing a pointer
- example:
  ```c
  int *count;
  float *avg;
  ```

Pointers: nitty gritty

- ampersand & is used to get the address of a variable (dereference a pointer)
- example:
  ```c
  int count = 12;
  int *countPtr = &count;
  &count returns the address of count and stores it in the pointer variable countPtr
  ```
Another example

- here’s another example:
  ```c
  int i = 3, j = -99;
  int count = 12;
  int *countPtr = &count;
  printf ( "%d", *countPtr);
  ```
- Here is the memory picture:

Arrays as pointers

- an array is some number of contiguous memory locations
- an array definition is really a pointer to the starting memory location of the array
- and pointers are really integers
- so you can perform integer arithmetic on them
- e.g., +1 increments a pointer, -1 decrements
- you can use this to move from one array element to another
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i, *j, arr[5];
    srand( time( NULL ));
    for ( i=0; i<5; i++ )
        arr[i] = rand() % 100;
    printf( "arr=%p\n",arr );
    for ( i=0; i<5; i++ ) {
        printf( "i=%d arr[i]=%d &arr[i]=%p\n", i,arr[i],&arr[i] );
    }
    j = &arr[0];
    printf( "\nj=%p *j=%d\n",j,*j );
    j++;
    printf( "after adding 1 to j:\n j=%p *j=%d\n",j,*j );
}

Output

arr=0xbfffff4f0
i=0 arr[i]=29 &arr[i]=0xbfffff4f0
i=1 arr[i]=8 &arr[i]=0xbfffff4f4
i=2 arr[i]=18 &arr[i]=0xbfffff4f8
i=3 arr[i]=95 &arr[i]=0xbfffff4fc
i=4 arr[i]=48 &arr[i]=0xbfffff500
j=0xbfffff4f0  *j=29
after adding 1 to j:
j=0xbfffff4f4  *j=8
Pointer operations

- Difference between
  - `ptr++`
  - `*ptr++`

- `int b[5] ....`
  - `int *bPtr;`
    - `bPtr = b`     //or
    - `bPtr = &b[0]`

- Careful when moving pointers

  - `bPTr += 2;`

  the memory location isn’t simply incremented by 2.....depends on size of type being pointed to.
Strings

- storing multiple characters in a single variable
- data type is still char
- BUT it has a length
- last character the is terminator: '\0', aka NULL
- string constants are surrounded by double quotes: "
- example:
  - char s[6] = "ABCDE";

String II

- char s[6] = “ABCDE”;
- Memory storage looks like:
  
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>\0</td>
</tr>
</tbody>
</table>

- Need to remember that you are really accessing indices 0 – (length-2) since the value at length-1 is always \0
Using strings

- **printing strings**
- **format sequence**: `%s`
- **example**:

```c
#include <stdio.h>

int main() {
    char str[6] = "ABCDE";
    printf( "str = %s\n", str );
} /* end of main() */
```

String Library

- to use the string library, include the header in your C source file:
  ```c
  #include <string.h>
  ```
- **string length function**:
  ```c
  int strlen( char *s );
  ```
  - this function returns the number of characters in `s`; note that this is NOT the same thing as the number of characters allocated for the string array
- **string comparison function**:
  ```c
  int strcmp( const char *s1, const char *s2 );
  ```
  - "This function returns an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared."

  `man strcmp`
copying functions:

char *strcpy( char *dest, char *source );
- copies characters from source array into dest array up to NULL

char *strncpy( char *dest, char *source, int num );
- copies characters from source array into dest array; stops after num characters (if no NULL before that); appends NUL

Search

char *strchr( const char *source, const char ch );
- returns pointer to first occurrence of ch in source; NULL if none

char *strstr( const char *source, const char *search );
- return pointer to first occurrence of search in source
String Parsing

char *strtok( char *s1, const char *s2 );
- breaks string s1 into a series of tokens, delimited by s2
- called the first time with s1 equal to the string you want to break up
- called subsequent times with NULL as the first argument
- each time is called, it returns the next token on the string
- returns null when no more tokens remain

Example

char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
rank = strtok( null,"+");
serial_num = strtok( null,"+");
Formatting functions

```c
int sscanf(char *string, char *format, ...) {
  parse the contents of string according to format
  placed the parsed items into 3rd, 4th, 5th, ... argument
  return the number of successful conversions
}
```

```c
int sprintf(char *buffer, char *format, ...) {
  produce a string formatted according to format
  place this string into the buffer
  the 3rd, 4th, 5th, ... arguments are formatted
  return number of successful conversions
  format characters are like printf and scanf (see notes from earlier lectures)
}
```

Memory allocations

- One of the most powerful features of C is the ability of the programmer to create more memory space during the execution of the program.
- Limited by physical machine memory
- If you want to be able to create memory, you also need to free it manually
malloc /sizeof / free

- charPtr = malloc ( sizeof ( ... ) );
- free (charPtr)

Compiling problems

- errors can come from multiple sources:
  - pre-processor: missing include files
  - parser: syntax errors
  - assembler: rare
  - linker: missing libraries and references
  - e.g., undefined names will be reported when linking:
    
    ```
    undefined symbol first referenced in file _print program.o
    ld fatal: Symbol referencing errors
    No output written to file.
    ```

- if gcc gets confused, there can be hundreds of messages!
  - fix first message first, and then retry—ignore the rest
- gcc will produce an executable with warnings
- gcc is more forgiving than javac!
For Next Time

- hopefully done with the homework
- Do Reading
  - important for this week’s lab, we are starting c section
  - will have tutorial for setting up eclipse in the lab and on your machine