Introduction to C

CS 429H SPRING 14

VIVEK NATARAJAN

Outline:

- I have 2 hours to teach you basics of C
- You will be using it for most of your assignments
- Main topics : Basic C program structure, functions, operators. control structures, pointers, structures, make
- I will hand out exercises. You can ssh into the CS Lab machines and try them - http://apps.cs.utexas.edu/unixlabstatus/
- ssh natviv@aero.cs.utexas.edu and invoke gcc
- Use putty to login from Windows

Editors

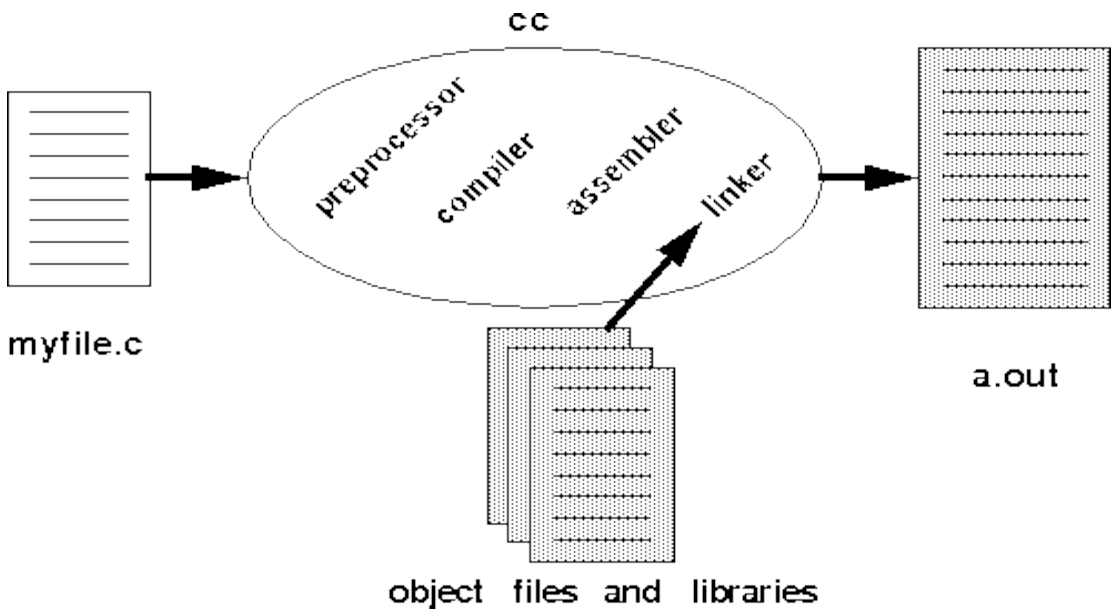
- Pick and choose an editor
- Emacs and Vim are the way to go
- Look online for resources
- Be prepared to spend lots of time learning them
- Number of cool tricks which would save you loads of time

C vs Java

- http://introcs.cs.princeton.edu/java/faq/c2java.html
- Much lower level than Java
- Java is object oriented while C is a procedural
- No classes in C
- Ability to manipulate raw bits and memory
- Linux kernel written in C

Basics

- Compiler vs Interpreter
- Preprocessor, Compiler, Assembler, Linker, Loader



Preprocessor

- Uninteresting – does textual substitution, semantics of the language not taken into consideration
- Expands macro definitions and includes (anything that begins with #)
- What if you had no preprocessor directives?
- Independent of the target architecture

Compiler

- Entire courses on compilers, Interesting
- Translates preprocessed C code into assembly code making optimizations and register allocations on the way
- The generated assembly code is target dependent
- Use gcc –S directive to see the assembly code generated

Assembler

- Assemble code translated to machine code – resulting file is called an object file (.o file)
- Gives symbolic memory location (through offsets) to your variables and instruction because its linked with other object files and libraries – cannot give absolute locations
- Makes list of all unresolved references present in other object files and libraries, eg printf

Linker

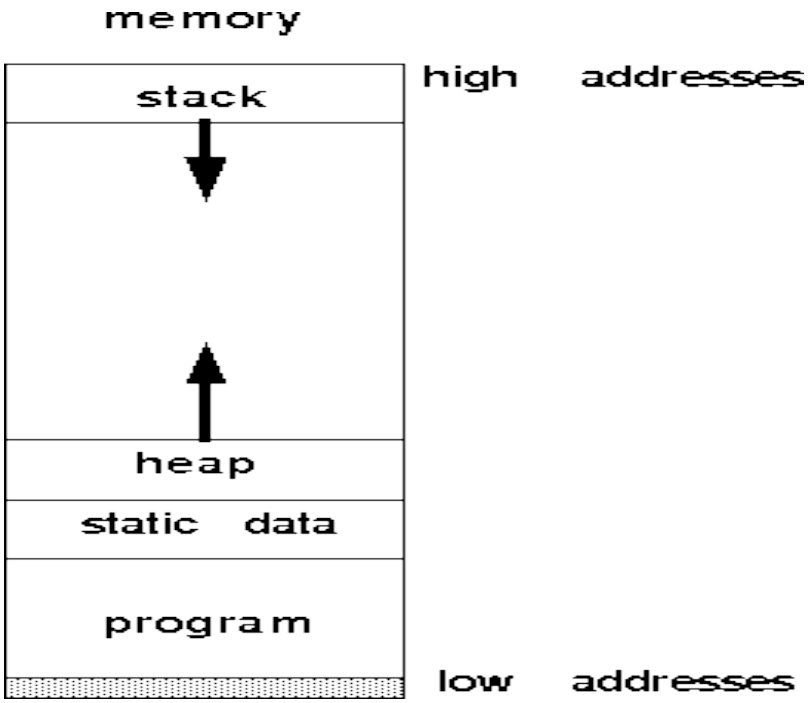
- Assembler writes ‘notes’ on how it assumed things were layered in the .o files
- Linker takes these notes and assigns absolute memory locations and resolves any unresolved references
- Might do optimizations like align procedures on page boundaries to avoid thrashing
- Produces a binary executable that can be run from the command line

Loader (Runtime)

- On typing a.out on the command line, a number of things happen which the loader does for you
- Creates a process by reading the file and creating the address space
- PTE for instruction, data, stack is created and register set initialized
- Executes a jump instruction to the first pgm instruction
- Causes a page fault and your program to load into memory
- Sometimes DLL references (Dynamically loaded libraries) must be resolved similar to a linker

Loader

- Static linkage (compile time) vs Dynamic linkage (run time)
- Responsibility to map instructions, data to appropriate locations
- Data allocated via malloc placed in the heap



Hello World

```c
/*
 hello.c
 A simple hello world program.
*/

#include <stdio.h>

int main(int argc, char **argv)
{
   printf("Hello, world!\n"); // greetings
   return 0;
}
```

Comments

- Block Comments /* ….
  Several lines here …..*/
- Single line comments //One line comment
- Some arcane compilers don't support these

#include statements

- Use # include <…> to access code libraries ex: #include <stdio.h>
- Header files contain constants, structure, function declarations and macro definitions to be shared between several source files. Put them at the top
- System header files contain declarations and definitions to invoke OS, make system calls and libraries
- Custom header files grouping related declarations and definitions to be used in several source files
- stdlib.h (standard utilities), stdio.h (basic IO), string.h (string functions), time.h (time functions), math.h (math functions)

Function declarations

- C program can be thought of as a collection of functions/procedures
- Same as in Java (to be precise static methods), except no visibility specifiers like public, private, etc
- Have one return type which can be void
- Can have any number of parameter inputs

```
type fn_name(type param1, type param2)
{
  // code here
  return something;
}
```

Main function

- The main function is the entry point to your program
- Return value indicates success or failure although ignored
- argc and argv hold command line arguments

```
int main(int argc, char **argv)
{
  // code goes here
  return 0;
}
```

Function Caveats

```c
int foo(); // function prototype

int main(int argc, char **argv)
{
   printf("%i\n", foo()); // foo called before it's defined
   return 0;
}

int foo() // foo defined down here
{
   return 99;
}
```

- Cant use a function before declaraing it. Declare a prototype of the function to use it before defining it
- Arguments passed by value i.e they are copied to the function parameters. Changes disappear when function ends. We use pointers to pass values by reference and circumvent this

printf

- printf handles console output, declared in stdio.h
- The first argument is the format string, other parameters are for substitutions
- Example: printf("Hello, world!\n");
- Example: printf("Login attempt %i:", attempts);
- There are tons of format specifiers, look them up

```
printf(char *format, type val1, type val2, ...)
```

Building and Running

- gcc cross compiler runs on many machines, works in phases
- By default, gcc will compile and link your program
- The -o flag tells it the name of the output binary
- Use ./name to run something

```
horatio-150:~ ckm$ gcc hello.c -o hello
horatio-150:~ ckm$ ./hello
Hello, world!
horatio-150:~ ckm$
```

Command line arguments

```
#include "stdio.h"

main( int argc, char *argv[], char *env[])
{
    int i;
    if( argc == 1 )
        printf( "The command line argument is:\n" );
    else
        printf( "The %d command line arguments are:\n",
            argc );

    for( i = 0; i < argc; i++ )
        printf( "Arg %3d: %s\n", i, argv[i] );
}
```

`argc` is the argument count, including the name of the program.
`argv` is an array of those strings.

Output

Here's a compilation and run of the program:

```
> gcc –o commargs commargs.c
> commargs x y z 3
The 5 command line arguments are:
Arg   0: commargs
Arg   1: x
Arg   2: y
Arg   3: z
Arg   4: 3
```

Assembler output from gcc

- Will be using it in Homework 1

You can produce assembler output, without running the assembler.

```
int sum( int x, int y)
{
   int t = x + y;
   return t;
}
```

To generate the assembler in file `sum.s`:

```
gcc –S –O2 –c sum.c
```

Output sum.s

```
            .file "sum.c"
            .text
            .p2align 4,,15
.globl sum
            .type   sum, @function
sum:
            pushl   %ebp
            movl    %esp, %ebp
            movl    12(%ebp), %eax
            addl    8(%ebp), %eax
            popl    %ebp
            ret
```

Variables

- The compiler tries to enforce types, and will attempt to convert or error out as appropriate. C not as strongly typed as Java. Casting not always safe
- Explicit typecasts can force conversions. Look up C implicit and explicit casting rules
- Variables must be defined at the beginning of a function, before any other code!

```
double foo()
{
   int a = -5;
   unsigned int b = 3;
   int c;
   c = a * (int)b; // cast b to int, just to be sure
   double q; // illegal, must be at top of function
   q = c; // implicitly converts c to double
   return q;
}
```

Data Types

- char: one byte (eight bits) signed integer – byte equivalent of Java
- short: two byte signed integer (same as short int)
- int: four byte signed integer (same as long int)
- unsigned: add to the above to make them unsigned
- float: four byte floating point
- double: eight byte floating point
- const: add to a data type to make its value constant

Assignment

- The equals operator copies the right hand side to the left hand side
- It also returns the value it copied, which enables some cool tricks
- In C, all strings end with a null character '\0'

```
char s[ ]= "Hello"
void func (char* s, char* d) {
while(*d++=*s++);
}
```

Logical operators

- Logic is supported as usual
- In order of precendence: ! (not), && (and), || (or)
- No boolean type; any integer zero is considered
- false, any integer nonzero is true
- !0 = 1, usually
- For example: 1 && !1 || !0 && -999 // true

Math Operators

- Math is the same as usual, with normal operator
- precedence (use parentheses when unsure)
- Supported operators are: + - * / %
- In-place versions as well: ++, --, +=, *=, etc.
- Integers round down after every operation
- No operator for exponent, ^ means something different (what ?) (look for pow() in math.h)

Comparison Operators

- Comparisons are also what you'd expect
- == (equals), != (not equals), < (less than), <= (less than or equals), > (greater than), >= (greater than or equals)

Bitwise operators

- You have an entire lab on this
- These treat data as a simple collection of bits
- Useful for low-level code, you'll use them a bunch
- They are: & (bitwise and), | (bitwise or), ~(bitwise not), ^ (bitwise xor), << (shift left), >> (shift right)
- Also useful: you can write hex numbers using 0x
- For example: 0x5B == 91

If/else

- Evaluates the given conditions in order, and will execute the appropriate block
- Can have any number of else ifs
- Else if and else are optional

```
if (condition)
{
        // condition is true
}
else if (other_condition)
{
        // condition not true, but other_condition is
}
else
{
        // none of the above were true
}
```

Switch

- A convenient way of doing lots of equality checks
- Why break? Why default?

```
switch (var)
{
    case 0:
            // if var == 0
            break;
    case 3:
            // if var == 3
            break;
    default:
            // if none of the other cases
            break;
}
```

Loops

- Loops work the same as in Java
- Remember to declare your loop variables at the top of the function
- Also do / while loops: same as while, but automatically execute once

```
for (i = 0; i < n; i++)
{
   // will execute n times
}

while (i != 0)
{
   // loop body
}
```

Arrays

- To declare an array, specify the size in brackets
- Size is fixed once an array is declared
- You can also provide an initializer list in braces
- If you omit the dimension, the compiler will try to figure it out from the initializer list
- Use brackets to index, starting with zero (note that bounds arent checked!)

```
int array[15];
int array2[] = { 3, 4, 99, -123, 400 };

for (i = 0; i < 5; i++)
   printf("%i\n", array2[i]);
```

Pointers

- A pointer is just a number (an unsigned int) containing the memory address of a particular chunk of data
- There is special syntax for dealing with pointers and what they point to
- They are by far the easiest and most effective way to shoot yourself in the foot –You will see a lot of them and hopefully master them by the end of the course

Declaring pointers

- Pointers are created by adding * to a variable declaration
- In the example above, ip is a pointer to an int,
- string and buffer are pointers to chars
- NULL is just zero, and is used to represent an uninitialized pointer

```
int *ip = NULL;
char *string, *buffer;
```

Using Pointers

```
int x = 10, y = 25; // declare two ints
int *p = NULL, *q = NULL; // and two pointers to ints

printf("%i\n", x); // 10
printf("%i\n", y); // 25

// & gets the address of a variable
p = &x; // p now points to x

// p just contains the number of a location in memory,
// so printing it won't mean much to a human
printf("%i\n", p); // some weird number

// use * to dereference (get the contents of) a pointer
printf("%i\n", *p); // 10

// you can change what a pointer points to
p = &y; // p now points to y
printf("%i\n", *p); // 25

// it's possible for two pointers to point to the same thing
q = p; // q now points to the same thing p does
printf("%i\n", *q); // 25

// since they point to the same thing, if you change the
// contents of one, you change the contents of the other!
*q = 9;
printf("%i\n", *p); // 9
```

Call by value vs Call by reference

Pointers and Arrays

- Arrays don't keep track of their length in C, you have to do that yourself
- The syntax shown earlier is just for convenience, arrays are actually just pointers to the first element of a contiguous block of memory
- Pointers can be interchanged with arrays, and indexed the same way

```
int buf[] = { 9, 8, 7, 6, 5 };
int *p = buf;

printf("%i\n", p[2]); // prints 7
```

Strings

- There's no special string type in C, strings are just arrays of characters ending in a null character \0
- You have to keep track of string length yourself strlen() in string.h will count up to the null for you
- strcpy() will copy strings.
- String literals are of type const char*.

```
const char *str = "Hi"; // same as const char str[3] = { 'H', 'i', '\0' }
```

Pointer Arithmetic

- You can increment and decrement pointers using the ++ and -- operators
- This will automatically move to the next or previous entry in an array
- Nothing will stop you when you hit the end of the array, so be careful!
- Dereferencing a pointer (*p).x => p->x. Will come back when we talk about structures

```
void strcpy(char *dst, const char *src)
{
   while(*dst++ = *src++);
}
```

More on Pointers

- Null pointer vs void pointer (value vs type)
- Null pointer is a special reserved value of a pointer
- Void pointer refers to the type of the data being pointed to in memory. Its void
- Null pointer used to indicate pointer that has not been allocated memory or something like end of a linked list
- Dangling pointer do not point to valid data of the appropriate type. Arise when data has been deleted or deallocated (using free) but pointer has not been modified. Unpredictable behaviour occurs on dereferencing the pointer

Pointer Caveats

- Q: What happens if you try to dereference a pointer that doesn't point to anything?
- A: CRASH! (Usually politely called an access violation (Windows) or a segfault in Unix.)
- Actually, that's the easy case. It may accidentally seem to work fine some of the time, only to break something else.
- Also happens if you index an array out of bounds

Dynamic Memory

- Allows you to create arrays of any size at runtime
- Include <stdlib.h> to get malloc() and free()
- malloc() gives you memory, free() releases it
- Difference between malloc, calloc, realloc and free
- calloc allocates and initializes to zero, realloc used to increase/decrease allocated memory block (where?)

```
int len = 97;
int *data = NULL;

// try to grab some memory
data = (int*)malloc(len * sizeof(int));

if (data)
{
   // alloc successful, work with data
   free(data); // release when done
}
```

Dynamic Memory

- The argument to malloc is the size of the requested memory block, in bytes
- sizeof() will give you the size of a datatype in bytes
- You have to cast the result of malloc to the pointer type you are using malloc() will return NULL if unsuccessful
- free() memory when you're done with it!
- Q: What happens if you don't free memory once you're done with it?
- A: You never get it back! That's called a memory leak. If you leak enough memory, you'll eventually run out, then crash.
- Q: What happens if you accidentally free memory twice?
- A: You crash.

Pointer tips

- If you're not using a pointer, set it to NULL
- This includes when the pointer is declared, otherwise it will initialize with random garbage
- Before dereferencing or using a pointer, check to see if it's NULL first
- Carefully track your memory usage, and free things when you're done with them using something like valgrind

Structures

- Structs allow you to group together several variables and treat them as one chunk of data
- Once defined, you can then instantiate a struct by using its name as a type

```
struct name
{
    type var1;
    type var2;
    // ...
};

name s1, s2;
```

Using Structures

- Use the dot operator to extract elements from a struct
- Use the arrow operator to pull out elements from a pointer to a struct

```
struct point
{
   float x, y;
};

point a, *p;

a.x = -4.0f;
a.y = 10.0f;

p = &a;

printf("%f\n", p->x); // -4.00000
printf("%f\n", p->y); // 10.00000
```

Structure Caveats

- When you pass a struct to a function, you get a copy of the whole thing
- This isn't bad for small structs, but copying larger ones can impact performance
- Pass pointers to structs instead, then use the arrow operator to manipulate its contents
- Don't forget the semicolon at the end of a structure definition

Typedef and enum

- Typedef allows you to rename types
- For example: typedef unsigned short uint16;
- Really handy for complicated pointer and struct types
- A type instead of int is used to represent a restricted set of values – enumeration like for days of the week. Can associate integers with them and cast and do operations
- typedef enum {RANDOM = 1, IMMEDIATE = 2, SEARCH =3} strategy_t;
  strategy_t my_strategy = IMMEDIATE;

Make

- Most UNIX projects are made of a ton of source files, which all need to be compiled and linked together
- Doing this all by hand would be annoying
- There's a program called make that does it for you

Makefiles

- Make knows what to build by looking in makefiles
- These are specially formatted rulesets that tell make how to build everything
- You don't normally need to know how they work
- It's good to know, but we won't teach you here

```
CC = gcc
CFLAGS = -O -Wall -m32
LIBS = -lm

all: btest fshow ishow

btest: btest.c bits.c decl.c tests.c btest.h bits.h
	$(CC) $(CFLAGS) $(LIBS) -o btest bits.c btest.c decl.c tests.c

fshow: fshow.c
	$(CC) $(CFLAGS) -o fshow fshow.c
```

Invoking Make

- Typing ‘make’ on the command line will automatically try to build the project described by ‘Makefile’ in the current directory
- Lots of stuff will happen, and make will report success or failure of the build
- You can also specify project-specific targets, like ‘make clean’

```
horatio-150:datalab ckm$ ls
Makefile        README          grade           src             writeup
horatio-150:datalab ckm$ make
##################################
# Build the btest test harness sources
##################################
(cd src; make clean; make)
rm -f *.o *~ btest fshow ishow bits-handout.c bits-middle.c bits.c bits.p.c decl.c tests.c bits.h *.exe
gcc -O1 -g -Wall -m32  -c btest.c
cpp -P -C -DTEST selections.c -Ipuzzles > tests-middle.c
```

Structures vs Unions

- Supposed to use only one element, allocated size of largest element. All stored in the same place
- union foo

  { int a; // can't use both a and b at once

  char b;

  } foo;

  union foo x;

  x.a = 0xDEADBEEF;

  x.b = 0x22;

  printf("%x, %x\n", x.a, x.b);
- Output: deadbe22, 22

Some other stuff

- Macros vs inline functions (Function calls expensive to avoid overhead. Macros by preprocessor, Inline functions at compile time. Leads to code bloat. Macros leads to bugs, have binding issues)
- # define DOUBLE(x) x*x //y=3; DOUBLE(++y)

We did not cover:

- Multi dimensional arrays
- Preprocessor
- GCC options go through
- Ternary operator and more

Info regarding upcoming labs

- Canvas or turnin?
- Please write comments in your code
- Possibly a report of what you are doing.
- These slides will be posted on Piazza and the course web site
- Now onto the exercise