A crash course in C

CS378H - Spring 2011

Christian Miller

Okay so...

- I have two hours to teach you the basics of C
- You will use it for most of your assignments this semester
- Also the most useful language to know, if you want to do pretty much anything

Computer access

- You'll be doing all your work on the CS machines
- The Painter or ENS Linux machines will work
- You can SSH into them from home
  - e.g. ssh bart@mondello.cs.utexas.edu
  - list of public hosts here: http://bit.ly/e2IfEV
  - use PuTTY if you're on a Windows machine

Editors

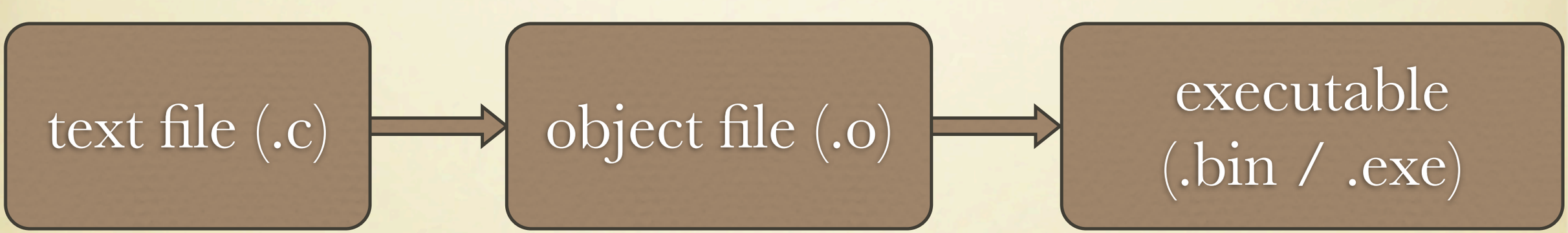
- You'll need to get comfortable with a text editor
- Most UNIX editors are arcane and bizarre
- vim and emacs are the usual choices
- Pick one and look online for tutorials
- Be prepared to spend a lot of time learning them

What is C, really?

- It's a relatively old programming language (1972)
- Much lower-level than Java
- No classes, purely procedural
- Has the ability to manipulate raw bits and memory
- Most systems-level coding is done in C, as well as a huge amount of application-level stuff

Some basics

C is a compiled language



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 here`
    - Some older C compilers don't support these

#include statements

- Use `#include <...>` to access code libraries
- e.g. `#include <stdio.h>`
- Put them at the top, before you use them
- Common ones: `stdlib.h` (standard utilities), `stdio.h` (basic I/O), `string.h` (string functions), `time.h` (time functions), `math.h` (math functions)

Function declarations

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

- Functions are pretty much the same as Java, except there are no visibility specifiers (public, private, etc.)
- Functions have one return type (can be void)
- There can be any number of parameters

Main function

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

- The main function is the entry point to your program
- Return value indicates success (0) or failure (nonzero), though this is usually ignored
- argc and argv hold command line arguments

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;
}
```

- You can't use a function before declaring it!
    - To use a function before defining it, declare it first with a function prototype
- Parameters passed to functions are copied, so changes made to them disappear when the function ends (use pointers to circumvent this)

Printf

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

- `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

Building and running

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

- 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

Variables

```
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;
}
```

- The compiler tries to enforce types, and will attempt to convert or error out as appropriate
- Explicit typecasts can force conversions
- Variables must be defined at the beginning of a function, before any other code!

Data types

- char: one byte (eight bits) signed integer
- 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

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 precendence (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 much different (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

- 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

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

- 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

Switch

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

- A convenient way of doing lots of equality checks
- The break statements in each case are necessary!

Loops

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

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

- 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

Arrays

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

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

- 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 (not bounds checked!)

Pointers

- New concept time!
- 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

Declaring pointers

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

- 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

Using pointers

```c
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
```

Pointers and arrays

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

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

- 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

Strings

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

- 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*.

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 or a segfault.)
  - 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

Pointer arithmetic

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

- 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!

Dynamic memory

```
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
}
```

- 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

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!

Pointer caveats

- 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.

Pointer caveats

- Q: What happens if you accidentally free memory twice?
- A: You crash.

Pointer hygiene

- 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

Structures

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

name s1, s2;
```

- 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

Using structures

```
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
```

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

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

```
typedef oldtype newtype;
```

- Typedef allows you to rename types
- For example: `typedef unsigned short uint16;`
- Really handy for complicated pointer and struct types

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

```
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
```

- 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

Invoking make

```
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
```

- 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’

There's more...

- But that's it for now
- Some topics not covered:
  - C preprocessor
  - Multidimensional arrays
  - Unions
  - Ternary operator
  - Etc. etc. etc.

These slides are online

- Get them here:
- http://www.cs.utexas.edu/~ckm/crashcourse.pdf