C: Pedal to the (bare)Metal

Nik Ingrassia and Riccardo Mutschlechner
UPL Coordinators

What is C?

- Kernighan and Ritchie
- Unix
- 1969-1973

- Linux Kernel - used in Computers, Phones
- Java Virtual Machine (C and C++)

Why use C?

- Bare-metal programming
- Small RAM/code size
- Absolute control over execution
- Lightning fast
- Truly lets you know what is going on “under the hood”

Getting Started (on your own, later)

- With Java, you have Eclipse, NetBeans, etc
- For C, there are really no “good” IDEs. instead:
  - emacs or vim for editing a plaintext file in the terminal - lots of resources out there to learn
  - Sublime Text or Notepad++ for a nice “free” graphical editor
  - JetBrains working on CLion: cross-platform IDE, free for students!

Hello, World! (Proper)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(“hello, world!\n”);
  return 0; //all went well!
}
```

(can be compiled with “`gcc filename.c`” on Unix, but not out of the box on Windows)

Compiling

- `gcc filename.c` will make a file called `a.out` that you can run by typing `./a.out`
- Some helpful flags for compilation [1]:
  - `-o` to specify the output file name: `gcc -o progName filename.c`
  - `-Wall` to give much better warnings
  - `-Werror` to prevent compilation until warnings fixed
  - `-g` to give better debugging info

Variables and Types

- Variables
  - Locations in memory where data is stored
- Types
  - Determines what kind of data will be stored in a variable.
- Type examples - very similar to Java, others
  - `int` - Integer: 1, 1000, -30
  - `char` - Character - ‘a’, ‘b’, ‘x’, ‘\0’
  - `float` - Floating point number - 0.23, 1.5

Exception: Bool

- By default, no boolean (called *bool* in C) type in C!
  - Must #include <stdbool.h> to be able to use them.
    - Just macros for 0 and 1, so, not required!
  - C Alternative: Simply use an int; 0 is false, anything else is true.

Using Variables

```
int a, b, c, d; //declarations
a = 10; //initializations …
b = 3;
c = a+b;
d = c - 4;
```

`printf()`

Function signature from man page:

```
int printf(const char *format, ...);
```

- In simpler terms:
  - First argument is a “string”, such as “hello world”, with (infinitely* many) optional format specifiers in it such as %s for string, %d for int, %c for char.
  - Second (and on) arguments replace specifiers

`printf()` Example

```
printf("hello, world!\n");
printf("%s\n", "hello, world!"); //same output; the %s gets replaced by the matching argument

int x = 2, y = 3;
printf("%d + %d = %d\n", x, y, x+y);
```

Control Statements and Loops

- **If/Else If/Else**
  - if (some condition)
  - else if (previous ***if(s)*** is false, some new condition is true)
  - else (if the previous condition(s) is false)
- **For**
  - for (an initialization; an end condition; a change)
- **While**
  - while (a condition remains true)

Using If/Else If/Else

```
int test = 2;
if(!test) { //”C way” of checking for 0 or null
   printf(“Test is 0.\n”);
} else if(test == 1) {
   printf(“Test is 1.\n”);
} else {
   printf(“Test is not 0 or 1!\n”);
}
```

Using For Loops

```
int i; //must declare i beforehand!


printf(“The numbers 1-10 follow:\n”);
for(i = 1; i < 11; i++) {
   printf(“%d\n”, i);
}
```

Using While Loops

```
int button;
button = readButton();
while(!button){
   button = readButton();
   printf(“Please press a button.\n”);
}
```

Functions

- Group of statements run with a single call.
- Useful to abstract or repeat code.
- Anatomy of a function:
  - `ret_type function_name(type arg1, type arg2)`
  - `ret_type` is the type of the return value - what the function will evaluate to
  - `arg1`, `arg2` are inputs to the function with given types.

Arrays

- A space in memory to hold multiple variables of the same type!
- Used for strings, images, sounds, etc.
- Syntax:
  - `type array_name[size];`

C Strings

- Null-terminated array of characters
- Doesn’t track own length!
- No string type - only *`char *`* or *`char[n]`*
- Example - “Hello World”
  - When we look at it as a set of characters, it is:
  - `Hello World\0`
  - `\0` is the null terminator

Structs

- How do you store sets of data that aren’t all the same type?
- Structs!
  - Struct definition:

```
struct struct_name {
    type item_1;
    type item_2;
};
```

Using Structs

- For the previous struct:
- `struct struct_name *our_struct;`
- `struct struct_name other_struct;`
  - `our_struct->item_1` - this accesses an item from the struct pointer
  - `other_struct.item_2` - this accesses an item from the struct
  - We'll talk more about pointers later on.

#include? What is that?

- Preprocessor directive
  - Very similar to “import …” in Java
- Allows combination of files/editing to be done at compile time!
- Very important/useful to organize code
- Allows use of outside code/libraries!

And now the FUN stuff

Pointers!

- A pointer is a type of variable.
- Pointers don’t hold data.
- Pointers hold the address of data!
- Pointers make C very useful for hardware.
- Pointers make C very dangerous!

Pointers Explanation

- * is used to define the pointer. (i.e. `int *x;`)
- * is also the dereferencing operator.
  - This gets the item/data **stored at** a given location when used on the RHS (i.e. `int a = *x;`)
- & is the referencing operator
  - This gets the location a given item is stored at.

Pointers Example

```
int a, b;
int *c; //or int * c, int* c. All the same
a = 2;
b = a;
c = &a;
*c = 1;
b = 4;
```

Pointers Answer

- `a == 1`
- `b == 4`
- `c == &a`
- `*c == a == 1`

Dynamic Memory Allocation

- All of our examples have been allocated at compile time
- What if you don’t know how big something is at compile time?
- Dynamic Memory Allocation solves these problems.
- And introduces some new ones.

`malloc()` and `sizeof()`

- `malloc()` - allocates some bytes of memory and returns a pointer to that memory.
- `sizeof()` - returns the size (in bytes) of whatever you pass into it. Different types have different sizes!
- Typical use of `malloc/sizeof`
  - `struct ListNode *ln = malloc(sizeof(struct ListNode));`
  - This gives us the memory for a new thing at `ln`.

Memory Management

- What happens when I’m done using a pointer?
  - Nothing!
- What happens if I try to write to/read from a bad address?
  - Crashes and faulty data!
- How do I deal with these issues?

`free()`

- `free()` “frees” the memory used by a pointer
- Use it when you won’t use the memory allocated by malloc anymore.

Linked List

- List of variable size!
- Composed of multiple nodes
- Nodes contain next (and optionally previous) node pointer

Linked List Node Struct Example

```
struct listnode {
   void *data; //void * = ptr to any type!
   struct _listnode *next;
   struct _listnode *prev;
};
```

Declared as: `struct listnode ln;`

Using typedef on structs

```
typedef struct _listnode {
   ...
} ListNode;
```

- Now we can declare a ListNode as:
  - ListNode ln;

How to get help?

- man pages: “RTFM”
  - usage: “man <function>” in terminal
  - http://linux.die.net/man/ same thing, but online
- StackOverflow.com
  - Be careful! People on the site can be rude:
    - Read the rules first at http://stackoverflow.com/tour
    - Look up your question beforehand!
    - If asking a new question, be VERY specific with errors, warnings, etc. Avoid “my code doesn’t work”
- UPL! Some coords know C well, are happy to help

Where do I go from here?

- “Homework” from us after this is done
- Take CS 354 - Machine Org. and Assembly
  - After that: CS 537 (Operating Systems) , CS 640 (Networks), CS 642 (Security) if you want to learn and use more low-level stuff
- Read K+R C book, dense but very great reference
- Read *Advanced Programming in the Unix Environment*

Questions?

References

[1]http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf