Lecture 3: C Basics

CSE 30: Computer Organization and Systems Programming
Winter 2010

Rajesh Gupta
Dept. of Computer Science and Engineering
University of California, San Diego

Acknowledgement: Ryan Kastner
C Basics

SECOND EDITION

THE C ANSI PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES
Has there been an update to ANSI C?

- Yes! It’s called the “C99” or “C9x” std
  - You need “gcc -std=c99” to compile

- References
  - http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html

- Highlights
  - Declarations anywhere, like Java (#15)
  - Java-like // comments (to end of line) (#10)
  - Variable-length non-global arrays (#33)
  - `<inttypes.h>`: explicit integer types (#38)
  - `<stdbool.h>` for boolean logic def’s (#35)
  - restrict keyword for optimizations (#30)
## C vs. Java™ Overview (1/2)

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚫ Object-oriented (OOP)</td>
<td>⚫ No built-in object abstraction. Data separate from methods.</td>
</tr>
<tr>
<td>⚫ “Methods”</td>
<td>⚫ “Functions”</td>
</tr>
<tr>
<td>⚫ Class libraries of data structures</td>
<td>⚫ C libraries are lower-level</td>
</tr>
<tr>
<td>⚫ <strong>Automatic</strong> memory management</td>
<td>⚫ <strong>Manual</strong> memory management</td>
</tr>
<tr>
<td></td>
<td>⚫ <strong>Pointers</strong></td>
</tr>
</tbody>
</table>
## C vs. Java™ Overview (2/2)

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>High memory overhead from class libraries</td>
<td>Low memory overhead</td>
</tr>
<tr>
<td>Relatively Slow</td>
<td>Relatively Fast</td>
</tr>
<tr>
<td>Arrays initialize to zero</td>
<td>Arrays initialize to garbage</td>
</tr>
<tr>
<td>Syntax:</td>
<td>Syntax:</td>
</tr>
<tr>
<td>/* comment */</td>
<td>/* comment */</td>
</tr>
<tr>
<td>// comment</td>
<td>printf</td>
</tr>
<tr>
<td>System.out.print</td>
<td></td>
</tr>
</tbody>
</table>
C Syntax: General

- Header files (.h) contain function declarations - the function interface
- The .c files contain the actual code.

```c
File.h
void func1(int, char *);
int func2(char *, char *);

File.c
void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = 'a';
    }
}

int func2(char *a, char *b)
{
    ...
}
```

- Comment your code:
  - only /* */ works, & they can’t be nested
  - // doesn’t work in C
To get the main function to accept arguments, use this:

```c
int main (int argc, char *argv[])
```

What does this mean?

- `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
- Example: `unix% sort myFile`
- `argv` is a pointer to an array containing the arguments as strings (more on pointers later).
C Syntax: Variable Declarations

- All variable declarations must go before they are used (at the beginning of the block).
- A variable may be initialized in its declaration.
- Examples of declarations:
  - **correct**: 
    ```c
    { 
        int a = 0, b = 10;
        ...
    }
    ```
  - **incorrect**: 
    `for (int i = 0; i < 10; i++)`
    (but OK for C99)
There is a difference between assignment and equality:

- `a = b` is assignment
- `a == b` is an equality test

This is one of the most common errors for beginning C programmers!

- One solution (when comparing with constant) is to put the var on the right!
  - If you happen to use `=`, it won’t compile.
  ```c
  if (3 == a) { ... }
  ```
C Syntax: True or False?

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer: more on this later)
  - no such thing as a Boolean
- What evaluates to TRUE in C?
  - everything else…
C syntax : control flow

- Within a method / function
  - if-else
  - switch
  - while and for
  - do-while

If-else control structure

```c
... if(a == 0) { i++; } else if(a == 1) { i--; } else if(a == 2) { i = 2; } else { i = 3; }
```

How do we convert this into an equivalent case control structure?

For control structure

```c
for(i = 0; i < 20; i++) { a[i] = b[i]; }
```

How do we convert this into an equivalent while control structure?
Address vs. Value

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.

- Don’t confuse the address referring to a memory location with the value stored in that location.
Pointers

- An address refers to a particular memory location. In other words, it points to a memory location.

- **Pointer**: A variable that contains the address of a variable.
Pointers

- How create a pointer:
  - `&` operator: get address of a variable

```c
int *x, y;

y = 3;

x = &y;
```

- How get a value pointed to?
  - `*` “dereference operator”: get value pointed to

```c
printf("x points to %d\n", *x);
```
How change variable pointed to?
- Use dereference \(*\) operator to left of =

\[ \text{How change variable pointed to?} \]

\[ \text{Use dereference \(*\) operator to left of =} \]

\[ *x = 5; \]

\[ x \quad y \]

\[ x \quad y \]

\[ 3 \]

\[ 5 \]
Pointers and Parameter Passing

- C passes a parameter “by value”
  - procedure/function gets a copy of the parameter, so changing the copy cannot change the original

```c
void addOne (int x) {
    x = x + 1;
}
int y = 3;
addOne(y);
printf("The value of y is \%d", y);
```

- What will be displayed?
Pointers and Parameter Passing

- How to get a function to change a value?

```c
void addOne (int *x) {
    *x = *x + 1;
}

int y = 3;

addOne (&y);
```

- What will be displayed?
Pointers

- Normally a pointer can only point to one type (`int`, `char`, a `struct`, etc.).
  - `void *` is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs!
More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.

What does the following code do?

```c
void f()
{
    int* x;
    *x = 5;
}
```
After declaring a pointer:

```c
int *ptr;
```

`ptr` doesn’t actually point to anything yet. We can either:

- make it point to something that already exists,
- or
- allocate room in memory for something new that it will point to… (next lecture)
Pointers & Allocation

- Pointing to something that already exists:

```c
int *ptr, var1, var2;
var1 = 5;
ptr = &var1;
var2 = *ptr;
```

- `var1` and `var2` have room implicitly allocated for them.