CS61C – Machine Structures

Lecture 3 – Introduction to the C Programming Language

1/23/2006

John Wawrzynek

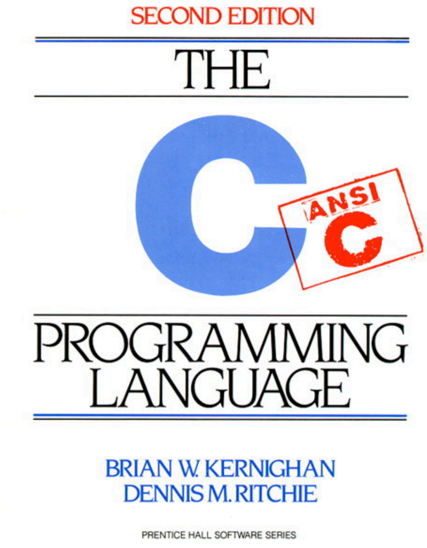
(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Administrivia : Near term

- Get cardkeys from CS main office Soda Hall 3rd floor.
- Reading for this week:
  - K&R Ch 1-4 (today, Ch 5-6 (W, F)
- HW
  - HW1 due Wednesday 11:59pm.
  - HW2 will be posted Wednesday.
- Project 1 - C Programming
  - Goes online tomorrow AM
  - Due Monday 2/6 (2 weeks from today)

Introduction to C



Why learn C?

Disclaimer

- **Important: You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course.**
  - **K&R is a great reference.**
    - **But… check online for more sources.**
  - **"JAVA in a Nutshell" – O'Reilly.**
    - **Chapter 2, "How Java Differs from C".**
  - **Brian Harvey's course notes.**
    - **On class website.**

Compilation : Overview

C *compilers* take C and convert it into an architecture specific machine code (string of 1s and 0s).

- Unlike Java which converts to architecture independent "bytecodes".
- Unlike most Scheme environments which interpret the code.

(These differ mainly in *when* your program is converted to machine instructions.)

For C generally a 2 part process of compiling .c files to .o files, then linking the .o files into executables

Compilation : characteristics

- Great run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- OK compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Compilation : Disadvantages

- All compiled files (including the executable) are architecture specific, depending on *both* the CPU type and the operating system.
- Executable must be rebuilt on each new system.
  - Called "porting your code" to a new architecture.
- The "change→compile→run [repeat]" iteration cycle is slow

C vs. Java™ Overview (1/2)

Java	C
Object-oriented (OOP)	No built-in object abstraction. Data separate from methods.
"Methods"	"Functions"
Class libraries of data structures	C libraries are lower-level
Automatic memory management	Manual memory management
	Pointers

C vs. Java™ Overview (2/2)

Java	C
• High memory overhead from class libraries	• Low memory overhead
• Relatively Slow	• Relatively Fast
• Arrays initialize to zero	• Arrays initialize to garbage

• Syntax:
`/* comment */`
`// comment`
`System.out.print`
	• Syntax:
	`/* comment */`
	`printf`

Newer C compilers allow Java style comments as well!

C Syntax: Variable Declarations

- Very similar to Java, but with a few minor but important differences
- 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:

```
{
    int a = 0, b = 10;
    ...
```

  - incorrect: `for (int i = 0; i < 10; i++)`

C compiler now allow this in the case of “for” loops.

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…
  - (same idea as in scheme: only `#f` is false, everything else is true!)

C syntax : flow control

- Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  - `if-else`
  - `switch`
  - `while and for`
  - `do-while`

C Syntax: `main`

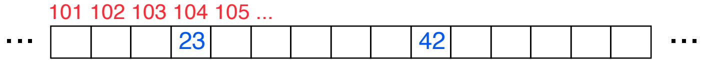
- To get the main function to accept arguments, use this:

```
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).

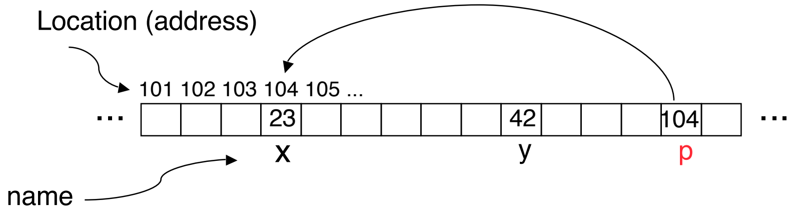
Address vs. Value

- 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
  - Do you think they use signed or unsigned numbers? Negative address?!
- 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 another variable.



Pointers

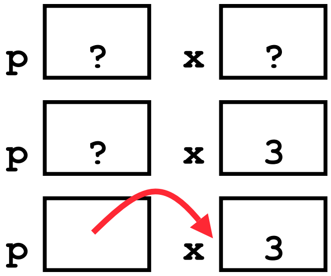
- How to create a pointer:

  & operator: get address of a variable

```
int *p, x;

  x = 3;

  p =&x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

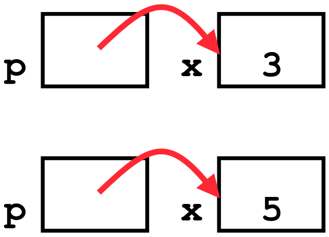
- How get a value pointed to?

  `*` "dereference operator": get value pointed to

```
printf("p points to %d\n",*p);
```

Pointers

- How to change a variable pointed to?
  - Use dereference `*` operator on left of =



```
*p = 5;
```

Pointers and Parameter Passing

- Java and C pass a parameter "by value"
  - procedure/function gets a copy of the parameter, so changing the copy cannot change the original

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

int y = 3;

addOne(y);
```

`y` is still = 3

Pointers and Parameter Passing

- How to get a function to change a value?

```
void addOne (int *p) {
  *p = *p + 1;
}
int y = 3;


addOne(&y);
```

`y` is now = 4

Pointers

- Of course pointers are used to point to any data type (`int`, `char`, a `struct`, etc.).
- Normally a particular pointer variable can only point to one type.
  - `void *` is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!

Find the Errors:

```
void main(); {
  int *p, x=5, y; // init
  y = *(p = &x) + 10;
  int z;
  flip-sign(p);
  printf("x=%d,y=%d,p=%d\n",x,y,p);
}
flip-sign(int *n){*n = -(*n)}
```

How many errors?

And in conclusion…

- All declarations go at the beginning of each function.
- Only 0 and NULL evaluate to FALSE.
- All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
- A pointer is a C version of the address.
  - * "follows" a pointer to its value
  - & gets the address of a value