C for Java Programmers

CS152

Chris Pollett

Sep. 8, 2008.

Outline:

- Introduction to C and UNIX Environment.
- A Basic C Program.
- Similarities between Java and C.
- Important differences.
- Program structure.
- Pointers.

Introduction to C and UNIX Environment:

- Next week we'll be talking about C based tools for writing lexers and parsers for programming languages.
- In order for these talks to make sense, we need to make sure everyone is on the same page as far as programming in C.
- Since it was first created at Bell Labs along with Unix in the early 70's, C has been ported to many OS's and environment.
- The particular OS/platform in which you program can have a significant effect on the kind of code you write.
- The same is true of Java. Although, the basic language is the same, the available libraries and stuff are different if you are using Java SE versus ME for instance.
- We'll focus in the class on using C in a Unix environment, and we will do things from the command line.

UNIX Environment Remarks:

- If you are using a Mac or Linux, there is nothing much involved in getting a command line environment – just launch a terminal.
- If you are using a Windows machine, it is recommended that you install Cygwin and make sure to also install the devel library. Then you can launch the Cygwin terminal and you should be able to do the same commands, as Mac and Linux users.
- I am most familiar with csh/tcsh as my Unix shell. If you prefer bash that is okay, just try to write your make files so they don’t rely on shell specific code.

More Unix:

- For those of you unfamiliar with Unix here are some of the most important shell commands to get started with :
  - man -- the Unix manual.
  - man ls -- would give the Unix manual entry for the ls command. man -k some_keyword -- lists all commands whose description has that keyword.

Paths are used to refer to documents somewhere on the computer. All directories are subdirectory of / . A path might be absolute and look like /foo/bar/ which refers to the bar subdirectory of the foo directory of the root dir. Or relative foo/bar -- bar subdir of foo dir of current directory. There are also some abbreviations for specialpaths: . -- current dir, .. --parent dir, ~ --user's dir, ~bob --user bob's dir.

- ls path -- list contents of directory given by path.

Still more Unix:

- cd path -- switches the current directory to be path.
- rm filename -- deletes filename. Can have flags like: rm -rf foo -- force deletes foo directory and all its children.
- cp f1 f2 -- copies file f1 to f2. cp -r dir1 dir2 copies contents of dir1 to dir2.
- mv f1 f2 -- moves/renames f1 to f2.
- set -- lists your environment settings. This can be particularly useful for checking if your path environment variable is set correctly. The path variable says which directories to look for the executables of the currently typed command.
- You can use whatever editors you like for editing your programs. Unix comes with a few built in editors such as vi, pico, emacs, but you can feel free to use a GUI editor.

A Basic C Program: hello.c :

```
#include <stdio.h> //for printf statement
int  main()
{
Printf ("HelloWorld\n");
return 0;
}
```

- To compile and run this program:

```
gcc hello.c -o hello
./hello
```

- Notice the main function is the main entry into a C program.
- #include is a preprocessor directive which says include the contents of the given header file a that location in the current document. stdio.h has the function prototype for printf, so that we can print to screen.

Similarities b/w Java and C:

- As of C99 you can use both /* */ or // for comments. ANSI C only had /* */ .
- Variable declarations are pretty much the same and similar types are available:

  int a, b; // declares two int's a and b.

  Have types float, double, char, etc.
- Array declarations also work similarly.
- int a[10]; //don't use a variable rather than 10 in ANSI C.
- Variable assignment also works like you would expect: a=5; a++; a--; ++a; --a; a+=100; //etc.
- If - statements, while statements, for statements, switch, etc. have the same syntax as Java.

Important differences:

- You don’t have a class construct. You can pretend -- if it helps -- as if everything is declared in the same class.
- Although, these rules were relaxed somewhat in C99 from ANSI C, you should have all your variable declarations at the start of your functions:

```
void foo()
{
    int   a=1;
   printf ("%d", a);
   int   b; // ANSI C’s head explodes
}
```

- Similarly, you either need to have a function prototype, or the function itself before the first place you use it -- See next slide.

Function examples:

```
//the following is okay:
char foo();
void bar()
{
  printf("%c", foo());
  }
  char foo( )
  {
return ‘a’;
}
```

```
//the following makes
ANSI C cry:
void bar()
{
 printf("%c", foo());
}
char foo()
{
return ‘a’;
 }
```

Program structure:

- Generally, for each module (say .c file) of your program, you put all its function prototypes, struct definitions, #define's, global variable declarations, etc. into a header file (a .h file) for that program and then include it with the line like:

```
#include "foo.h" /* double quotes unlike <foo.h>, also searches current directory */
```

- Since, you might use the same header file for several modules, the header file might have at its start some preprocessor code like:

```
#ifndef FOO_H
#define FOO_H
//header file code
#endif // this prevents the header from being included more than once.
```

Pointers:

- Roughly, pointers allow us to refer to a memory address.

  Example: int a, *b; /* b is declared as a pointer to an int memory address */.
  a = 5;
  b = &a; /* now b points at where a is stored in memory */.
  printf ("%d", *b); //prints 5.
  b = 6; / this changes the value of what's stored at the address b points to */.
  printf ("%d", a); //prints 6.

- We'll see pointers can be used very much like arrays.
- You can have pointers to pointers (handles):

char **p; // sort a like a 2d array.

- You can also have pointers to functions (useful for doing things like to simulate strategy pattern):

```
int add (int a, int b) { return a+b; }
int (* p) (int, int);
p=&add;
printf("sum: %d", (*p)(3,4)); //output 7.
```