CS3600 C Primer for Java Programmers
15 September 2011

Stolen from CMU:
http://www.cs.cmu.edu/afs/cs/academic/class/15213-f02/www/CPrimer/CPrimer.ppt

Outline
• Overview comparison of C and Java
• Good evening
• Preprocessor
• Command line arguments
• Arrays and structures
• Pointers and dynamic memory

What we will cover
• A crash course in the basics of C
• You should read the K&R C book for lots more details
Like Java, like C

- Operators same as Java:
  - Arithmetic
    - i = i+1; i++; i--; i *= 2;
    - +, -, *, /, %
  - Relational and Logical
    - <, >, <=, >=, ==, !=
    - &&, ||, &,

- Syntax same as in Java:
  - if ( ) { } else { }
  - while ( ) { }
  - do { } while ( );
  - for(i=1; i <= 100; i++) { }
  - switch ( ) { case 1: ... }
  - continue; break;

Simple Data Types

<table>
<thead>
<tr>
<th>datatype</th>
<th>size</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>3.4E+/-38 (7 digits)</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>1.7E+/-308 (15 digits long)</td>
</tr>
</tbody>
</table>

Java programmer gotchas (1)

```
{ int i
  for(i = 0; i < 10; i++)
    ...

  NOT

{ for(int i = 0; i < 10; i++)
  ...
```
Java programmer gotchas (2)

- Uninitialized variables
  - catch with `-Wall` compiler option

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;
    factorial(i);
    return 0;
}
```

Java programmer gotchas (3)

- Error handling
  - No exceptions
  - Must look at return values

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    /* print a greeting */
    printf("Good evening!\n");
    return 0;
}
```

```
$ ./goodevening
Good evening!
```

“Good evening”

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    /* print a greeting */
    printf("Good evening!\n");
    return 0;
}
```

```
$ ./goodevening
Good evening!
```

```
$ ./goodevening
```

$
Breaking down the code

• #include <stdio.h>
  - Include the contents of the file stdio.h
    • Case sensitive – lower case only
    • No semicolon at the end of line
• int main(...)
  - The OS calls this function when the program starts running.
• printf(format_string, arg1, ...)
  - Prints out a string, specified by the format string and the arguments.

format_string

• Composed of ordinary characters (not %)
  - Copied unchanged into the output
• Conversion specifications (start with %)
  - Fetches one or more arguments
  - For example
    • char %c
    • char* %s
    • int %d
    • float %f
• For more details: man 3 printf

C Preprocessor

#define HELLO "Hello world\n"

int main(int argc, char* argv[])
{
  printf(HELLO);
  return 0;
}
After the preprocessor (gcc -E)

```c
int main(int argc, char* argv)
{
    printf("Hello world\n");
    return 0;
}
```

Conditional Compilation

```c
#define CS3600

int main(int argc, char* argv)
{
    #ifdef CS3600
    printf("Hello world\n");
    #else
    printf("Goodbye, cruel world\n");
    #endif
    return 0;
}
```

After the preprocessor (gcc -E)

```c
int main(int argc, char* argv)
{
    printf("Hello world\n");
    return 0;
}
```
Command Line Arguments (1)

- int main(int argc, char* argv[])
- argc
  - Number of arguments (including program name)
- argv
  - Array of char's (that is, an array of 'c' strings)
  - argv[0]: = program name
  - argv[1]: = first argument
  - ...
  - argv[argc-1]: last argument

Command Line Arguments (2)

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;
    printf("%d arguments\n", argc);
    for(i = 0; i < argc; i++)
        printf(" %d: %s\n", i, argv[i]);
    return 0;
}
```

Command Line Arguments (3)

```sh
$ ./cmdline Hello world
3 arguments
0: ./cmdline
1: Hello
2: world
$```
Arrays

- **char foo[80];**
  - An array of 80 characters
  - `sizeof(foo)`
    - $= 80 \times sizeof(char)$
    - $= 80 \times 1 = 80$ bytes
- **int bar[40];**
  - An array of 40 integers
  - `sizeof(bar)`
    - $= 40 \times sizeof(int)$
    - $= 40 \times 4 = 160$ bytes

Structures

- Aggregate data

```c
#include <stdio.h>

struct name {
    char* name;
    int age;
}; /* <= DO NOT FORGET the semicolon */

int main(int argc, char* argv[])
{
    struct name bovik;
    bovik.name = "Harry Bovik";
    bovik.age = 25;
    printf("%s is %d years old\n", bovik.name, bovik.age);
    return 0;
}
```

Pointers

- Pointers are variables that hold an address in memory.
- That address contains another variable.
Memory layout and addresses

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>12.5</td>
<td>9.8</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4300</td>
<td>4304</td>
<td>4308</td>
<td>4312</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4317</td>
<td>4316</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using Pointers (1)

```
float f; /* data variable */
float *f_addr; /* pointer variable */

f_addr = &f; /* & = address operator */
```

Pointers made easy (2)

```
*f_addr = 3.2; /* indirection operator */

float g = *f_addr; /* indirection: g is now 3.2 */
f = 1.3; /* but g is still 3.2 */
```
Function Parameters

- Function arguments are passed “by value”.
- What is “pass by value”?
  - The called function is given a copy of the arguments.
- What does this imply?
  - The called function can’t alter a variable in the caller function, but its private copy.
- Three examples

Example 1: swap_1

```c
void swap_1(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Q: Let x=3, y=4, after swap_1(x,y); x=? y=?

A1: x=4; y=3;
A2: x=3; y=4;

Example 2: swap_2

```c
void swap_2(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Q: Let x=3, y=4, after swap_2(&x,&y); x=? y=?

A1: x=3; y=4;
A2: x=4; y=3;
Example 3: scanf

```c
#include <stdio.h>
int main()
{
    int x;
    scanf("%d\n", &x);
    printf("%d\n", x);
}
```

Q: Why using pointers in scanf?
A: We need to assign the value to x.

Dynamic Memory

- Java manages memory for you, C does not
  - C requires the programmer to explicitly allocate and deallocate memory
  - Unknown amounts of memory can be allocated dynamically during run-time with `malloc()` and deallocated using `free()`

Not like Java

- No `new`
- No garbage collection
- You ask for `n` bytes
  - Not a high-level request such as “I'd like an instance of class `String`"
malloc

- Allocates memory in the heap
  - Lives between function invocations
- Example
  - Allocate an integer
    - `int* iptr = (int*) malloc(sizeof(int));`
  - Allocate a structure
    - `struct name* nameptr = (struct name*) malloc(sizeof(struct name));`

free

- Deallocates memory in heap.
- Pass in a pointer that was returned by `malloc`
- Example
  - `int* iptr = (int*) malloc(sizeof(int));`
     `free(iptr);`
- Caveat: don’t free the same memory block twice!