Discussion #2: C Basics

Written by Justin Hsia (6/22/2011)

Note: All the nitty-gritty details can be found in K&R, so I will only make points I feel are worthwhile here before referring you to the appropriate book pages.

Note: For programming, I believe in learning by doing. The answers to the exercises below are not given because you can easily code them up to test/verify. I encourage you to consult with other students as well, since coming up with *a* solution does not mean that is it the *best* solution. Also knowing *why* you get a particular outcome is more important than the outcome itself.

Generic C program layout:

```
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables and structs */

int main(int argc, char *argv[]) {
      /* the innards */
}

/* define other functions */
```

This exact structure is not required, but good formatting practice. Individual parts explained below.

The C Preprocessor

Runs through your code first and makes **SUBSTITUTIONS ONLY**. Does not create any variables or allocate memory or anything like that. C preprocessor commands are started with the '#' symbol.

#include *filename* – dumps contents of a file directly into your code. *Filename* does not have to have the extension .h, but good practice to signify header files. In general, header files (.h) should contain only declarations and code files (.c) should contain definitions. If a known system file (see K&R Appendix B), surround with <*filename*>. If a local file, surround with "*filename*".

#define *name replacement_text* – does a macro substitution. Finds token *name* in your code and replaces text EXACTLY AS IS with *replacement_text*. Not actual variables, so cannot assign to. Can do "function-like" macros such as:

```
#define max(A,B) ((A)>(B)?(A):(B))
```

Check: What is the value of `result`?

```
#define NUM 44-1
result = NUM*2;
```

The Innards: Variables

Variable types: `char` and `int` are integral types
`float` and `double` are floating types (exponential and decimal numbers)

Can convert/typecast `char` to `int` to `float` to `double` without loss, but not the other way around.

Variable qualifiers: The “adjectives” to the variable type “nouns.”

- `unsigned` (p. 36)
- `const` (p. 40)
- `short`/`long` (p. 36)
- `static` (p. 83)

The Innards: Basic Operators

For precedence/order of execution, see Table 2-1 on p. 53.

Arithmetic:	`=  +  -  *  /  %  ++ --`	Last three are modulus, increment, decrement.
Comparison:	`== != >  <  >= <=`	Will return 1 (true) or 0 (false).

Check: Write the function `int nonneg(int n)` that returns `1` if n is non-negative and `0` if n is negative. Use only a single statement (one semi-colon).

Check: Again using a single statement, rewrite the above function into `int sign(int n)`, which instead returns `-1` if n is negative. Use only the operators above and only integer constants.

Logical:	`!  &&  ||`	Use to modify/combine comparisons.

Check: What is the result of `int result = !(3 > 5) + 3*(35 == '#')`?

Bitwise:	`~ &	^ << >>`	Useful for masking or selecting bits.

Check: Write the function `int getbit(int n, int p)`, which returns the p’th bit in the integer n, counting from the right. You can assume the integer 1 is represented by all zeros with the rightmost bit set to 1.

Compound: Let ‘●’ be an operator in the set {`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`}.
Then `a ●= b` is equivalent to `a = a ● b`.

Check: What is the result of the following block of code?

```
int i = 1;
i <<= 3;
i ^= 15;
i %= 9;
```

The Innards: Control Flow

Statements:	`if-else if-else` (p. 55-58)	`switch` (p. 58-59)	`conditional` (p. 51-52)
Loops:	`for` (p. 60)	`while` (p. 60)	`do-while` (p. 63-64)

Use `break` to exit a loop or `switch` statement, and `continue` to skip directly to the next loop iteration.

Check: Assume you can call the `int sign(int n)` function defined above. Write the function `int mod(int n,int m)` that returns the equivalent of `n%m` using a `for` loop with an empty body and a `return` statement. It should handle both positive and negative values of `n`.

Check: Write the function `void printHexDigit(int d)` that prints the hex digit character equivalent of the decimal input `d` between 0 and 15. Use a single statement (one semi-colon).

Extra: See if you can implement `printHexDigit` as described above as a macro.

Functions

```
return_type function_name(argument list);

return_type function_name(argument list) {
      /* code here */
      return expr;
}
```

The first line above is the function declaration (function prototype) and is only needed if function is called before it is defined. Not required, but good practice to use exact same argument names in function prototype. There is no function overloading in C!

Variable scope: If a variable is declared within a function, it "disappears" when that function returns.

Check: What is terrible about the following function?

```
int *blasphemy(void) {
      int x; return &x;
}
```

Pass by value: Arguments (and return values) are passed BY VALUE in C. This means that a copy of the data is stored under the argument name in the function, so changing that value does not affect the original data at all.

main: When, called program execution always starts in `main()`, which should return an `int`. Use the following argument list to accept command-line arguments:

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

Command-line arguments include the name of the executable that is being run. argv is an array of character pointers (covered later). Each entry in the array is a pointer to an array of characters for each command-line argument (everything stored as characters).

Check: Assume a program is run from the UNIX shell with the following command:

```
$ ./manipulateCommandLine hello 2.56
```

What would the following expressions evaluate to inside that program?

```
argc
*argv[0]
*(argv[1]+4)
*argv[2]+1
```

Structs and typedef

Structs are user-defined collections of variables. A structure definition goes as follows:

```
struct structure_tag {
      type1 member1;
      ...
      typen membern;
};
```

Structure tag: The structure tag is optional and is used to refer to this structure in the future. If you are only going to create a single instance of this particular structure, then the structure tag may be unnecessary. In general it's always best to give a structure tag. This structure type is later used as `struct structure_tag`.

Structure members: A structure can hold an arbitrary collection of members of different variable types, meaning that the size of a structure can vary greatly. The only exception is that a structure cannot hold an instance of itself (but a pointer of the structure type is okay). Members are accessed using the operator '`.`'.

Struct variables: Unlike other variable types, structures need to be defined at first. It turns out that you can combine a structure definition with a variable declaration and even initialization! This can get a little confusing and long, but the following are all valid:

```
struct {int x; int y;} var;
```
Defines a variable `var` of an unnamed structure type that contains two integers `x` and `y`.

```
struct point {int x; int y;};
```
Defines the type `struct point`.

```
struct point {int x; int y;} pt1;
```
Defines the variable `pt1` of type `struct point`.

```
struct point {int x; int y;} pt1 = {1,2};
```
Defines the variable `pt1` of type `struct point` and initializes it to `pt1.x`=1 and `pt2.y`=2.

For clarity, most structure definitions are spaced like the generic form shown at the top of this section.

These two-word variable names `struct structure_tag` are a bit cumbersome, so we often combine them with `typedef`, which allows you to create new data type names such as:

```
typedef unsigned int uint;
```
New variable name `uint` refers to `unsigned int`.

For structs, combine typedef with struct definition to make a more manageable variable type:

```
typedef struct point {
      int x;
      int y;
} Point;
Point pt1 = {1,2};
```

Check: What's wrong with the following code?

```
struct {int val1; int val2;} mystruct;
struct mystruct s1 = {7,13};
```