ME 461 C review Session
Fall 2009
S. Keres

DISCLAIMER: These notes are in no way intended to be a complete reference for the C programming material you will need for the class. They are intended to help focus your review efforts toward the concepts that will be the most useful.

Digital Representations

You are already familiar with ***decimal*** (base 10) representation.

Binary (base 2) is similar to decimal, except that the digits represent ones, twos, fours, eights, 2^4s, 2^5s, etc, instead of ones, tens, hundreds, 10^3s, 10^4s, etc.

Hexadecimal (base 16) is the same, except the digits represent ones, sixteens, 256s, 16^3s, 16^4s, etc. Numbers 10-16 are represented by the letters A-F.

Dec	Bin	Hex	Dec	Bin	Hex
1_{10}	1_2	1_{16}	9_{10}	1001_2	9_{16}
2_{10}	10_2	2_{16}	10_{10}	1010_2	A_{16}
3_{10}	11_2	3_{16}	11_{10}	1011_2	B_{16}
4_{10}	100_2	4_{16}	12_{10}	1100_2	C_{16}
5_{10}	101_2	5_{16}	13_{10}	1101_2	D_{16}
6_{10}	110_2	6_{16}	14_{10}	1110_2	E_{16}
7_{10}	111_2	7_{16}	15_{10}	1111_2	F_{16}
8_{10}	1000_2	8_{16}	16_{10}	10000_2	10_{16}

- Hexadecimal is very convenient when working directly with memory and registers (as we will be) because each ***nibble*** (group of 4 binary digits) can be represented with a single hexadecimal digit.
- In C, we cannot write directly in binary, but we can write in hex. Hexadecimal numbers are prefixed by '`0x`' in C. Example: `a = 0x52` is the same as `a = 82`, which is the same as binary 101 0010.
- It is important to note that C does not distinguish between representations (everything is binary to a computer). The ability to write numbers using different representations is a facility provided for our convenience.

Operators

Arithmetic:

+	Addition	++	increment
-	Subraction	- -	decrement

*	Multiplication	+=	assignment by addition
/	Division	-=	assignment by subtraction
%	Modulus (remainder)	*=	assignment by multiplication
%=	assignment by modulus	/=	assignment by division

- The increment and decrement operators do just that; the timing depends on whether they are postfixed or prefixed to the operand. Example: `a = b++;` assigns b's value to a, then increments b. `a = ++b;` increments b, then assigns the value to a.
- Any of the arithmetic operators besides increment and decrement may be combined with the assignment operator to act as shorthand as in the following example. Assignment by addition: `a += b` is the same as `a = a + b`. Likewise, `a *= b` is the same as `a = a * b`, and so on.

Logical:

`a && b`	AND – true if neither operand is zero
`a || b`	OR – true if either operand is nonzero
`! a`	NOT – true if operand is zero
`a == b`	equality – true if operands are the same number
`a != b`	inequality – true if operands are not the same number
`a > b`	greater than – true if a is larger than b
`a < b`	less than – true if a is less than b
`a >= b`	greater than or equal to
`a <= b`	less than or equal to

Bitwise:

`a | b`	bitwise OR
`a & b`	bitwise AND
`a ^ b`	bitwise XOR – exclusive OR
`~ a`	bitwise complement – "flips" bits from 1 to 0 and 0 to 1
`a << b`	left shift – shifts the bits of a left and inserts zeros to the right
`a >> b`	right shift – shifts the bits of a right and inserts leading zeros

- All of the bitwise operators except the complement can be combined with the assignment operator to act as shorthand as in the following example. `a |= b` is the same as `a = a | b`. Likewise `a &= b` is the same as `a = a & b`, and so on.

Examples:

```
0x15 | 0x98          0x15        0001 0101
                   | 0x98        1001 1000
                                 1001 1101   = 0x9D
```

```
~ 0xA6                  0xA6          1010 0110
                        complement    0101 1001    = 0x59

5 ^ 3                   5                  0101
                      ^ 3                  0011
                                           0110    = 6

9 << 2                  9                  1001
                        left shift       10 0100   = 0x24 = 36
```

Application: setting/clearing bits in registers

Set the 5th and 6th bits in an 8-bit register called P1OUT without changing the others or knowing their previous state

```
P1OUT |= 0x30           Before P1OUT =             xxxx xxxx
                                   | 0x30         0011 0000
                        After  P1OUT =             xx11 xxxx
```

Clear the 1st and 7th bits in P1OUT

```
P1OUT &= ~0x41          Before P1OUT =             xxxx xxxx
                                   & ~0x41       1011 1110
                        After  P1OUT =             x0xx xxx0
```

Can you think of an operation we can use to "flip" (1→0 and 0→1) bits of our choosing without affecting the others or knowing their values?

Other:

`a = b`	assignment
`a[b]`	array subscripting (indexing) – accesses the b[th] element of array a
`& a`	reference – returns the memory address of a, i.e. constructs a pointer to a
`* a`	dereference – accesses the value stored at location a in memory, "the thing that a points to"

- The ability to work with pointers is essential to the creation of a sophisticated program. Our most common use of pointers will involve passing variables by reference to functions.

Basic Data Types

Integer types:

`char` – usually 8 bits long, may be used to store a single character

`int` – usually 16 bits long, but more generally has more precision than a char

Basic type modifiers: **`long`**, **`short`**, **`signed`**, **`unsigned`**

Floating-point types:
`float` – IEEE 32-bit floating point number
`double` – double-precision floating-point number (64 bits)

Other:
`void` – valueless, carries no data
`*` – when postfixed to a type specifier constructs a ***pointer*** to a variable of that type

Variable declarations

Variables are declared like

```
type var_name = value;
```

where `type` is a valid type preceded by one or more modifiers, `var_name` is a name for the variable, and `value` is an optional initial value for the variable.

Example: `unsigned int counter1 = 0;  float error = 0.0;  char letter = 'a';`

Arrays are declared like

```
type my_arr[dim1][dim2]…[dimN];
```

and may be initialized like

```
int my_2by2[2][2] = {{1, 2},{5, 9}};
```

- Variables declared outside of functions are ***global*** and carry their values throughout the program.
- Variables declared inside functions are ***local*** and carry their values only through a call to the function.

Type Conversion

- Explicit – A variable's type may be changed (cast) to another temporarily during an operation.

A ***type cast*** is performed by prefixing the new type in parentheses to the quantity of interest.

Example:

```
int wholenumber = 0;    float z = 5.0;
wholenumber = (int)(z/2.0) + 5;
```

- Implicit – Variables are promoted according to C conversion rules during operations between variables of different types

In the following example, the integer is promoted to a float due to the floating point divisor and there is no loss of precision; the result is `z = 2.5`.

```
int wholenumber = 5;          float z = 0;
z = wholenumber/2.0;
```

If instead of 2.0 we had just used 2, the result would have been `z = 2.0` due to an integer division.

- Take some time to familiarize yourself with C's conversion rules using your C reference. Also, if you're not already proficient, review discrete math fundamentals.

Functions

Functions are created like

```
type function_name(argument list) {
variable list
function body
return statement
}
```

Example:

```
float my_avg(int a, int b, int c) {   function header
   int sum = 0;                       local variables
   sum = a + b + c;                   function body
   return sum/3.0;                    return statement
}
```

This function takes three integers as inputs and returns their average. A call to the function would look like

```
float z = 0.0;
z = my_avg(3, 4, 11);
```

- The function call itself can be treated as any variable of the same type as the output of the function. It can be added, subtracted, multiplied, divided, type cast, etc.
- Note that we can create void functions that have no output. In this case the return statement is optional. In the same way, we can create functions that have no input arguments. We can also create functions that have neither inputs nor outputs. Example: `Init_PWM();` is a complete line of code assuming the function has neither inputs nor outputs.
- Functions must be prototyped by stating the header information in the global variable area before they are used if their definition occurs later in the file or in a different c file.

Program Control Statements

If-Else

An if-else statement allows the programmer to execute code depending on the result of a conditional statement.

```
if(condition) {
statement block
} elseif(condition) {
statement block
```

```
} else {
statement block
}
```

Example:

```
char up = 1;      float a = 0.0;
if(up) {
   a += 0.1;
   if(a >= 10) up = 0;
} else {
   a -= 0.1;
   if(a <= -10) up = 1;
}
```

- The **elseif** statement allows the programmer to allocate code blocks to several mutually exclusive alternatives.

While

The while statement executes a block of code repeatedly as long as the conditional is true.

```
while(condition) {
statement block
}
```

Do-while

Same as the while, except the code block executes once regardless of the conditional.

```
do {
statement block
} while(condition);
```

For

Executes a block of code repeatedly as long as the conditional (usually) based on the loop variable is true.

```
for(initialization; conditional; operation) {
statement block
}
```

- At the beginning of the loop, the *initialization* is performed. After each execution of the *statement block*, the *operation* is performed. When the *conditional* is false, the program continues execution after the **for** statement (after the closing bracket).

Example:

```
int i, j = 0, my_arr[10];
```

```
for(i = 0; i < 10; i++) {
   my_arr[i] = j;
   j += 5;
}
```

Can you write out the contents of the array `my_arr`?

Switch-Case

Executes a statement block based on the value of a variable.

```
switch(control variable) {
   case constant1:
      statement block
      break;
   case constant2:
      statement block
      break;
   ⋮
   default:
      statement block
      break;
}
```

- The code block associated with the case matching the *control variable*'s value is executed once. If no match is found among the cases, the code associated with the default statement is executed.
- If the break statements are omitted, code execution will continue through every case once a match is found. This is rarely desirable.
- As with all of the control statements, nesting switch statements is allowed and often very useful.

Preprocessor Directives

`#include` – used to tell the compiler what libraries to "look in" for function declarations, variables, macros, etc.

Example: `#include <msp430x22x2.h>`

- Use <> around the file name when it is in the search path (usually defined in a properties section) and double quotes around the file name when it is in the project directory

`#define` – used as a "dumb" text find & replace macro

Example: `#define PI 3.141592654`

This causes all instances of the text "`PI`" in the c file to be replaced by the constant 3.141592654.

- **#define** is very useful for parameterizing your programs. Use the **#define** macro whenever you will use a special constant throughout your program for clarity and ease of modification.

#define can also be made to accept input arguments.

Example: **#define** `AVG(A,B) (A+B/2.0)`

This is useful for creating very simple in-line functions.

- Remember, **#define** is a simple text replace. Careful use of parentheses is required!

#pragma – directive used to specify compiler-specific instructions.

Example: **#pragma** `vector=TIMERA0_VECTOR`

The "vector" pragma is used before a function definition to tell the compiler that the function should be placed in the interrupt vector address.

#if, **#ifdef**, **#ifndef**, **#elif**, **#else**, **#endif** – conditional compilation. These will see limited, if any, use in our class.

C References

In lab:
Schildt, *Teach Yourself C* (1997).

Kernighan, Ritchie, *The C Programming Language* (1988).

Harbison, Steele, *C: A Reference Manual* (2002).