A few notes on using pointers in the C language

R.C. Maher EE475 Fall 2004

A C language *pointer* is a variable that contains *the address* of a variable.

The address of a variable is a non-negative integer number that uniquely identifies a specific location in the storage available to the program. On small microprocessors the address is usually the actual physical address used by the hardware, while on larger systems it may be a virtual quantity. In either case the C language pointer concept can be used to perform fast and efficient indexing, referencing, and manipulation of data structures.

Using "&" (the "address of" operator)

The statement

```
int integer_value = 13;
```

tells the compiler to assign storage sufficient to hold one integer, and initializes the stored quantity to be the number 13. The program can use the variable name `integer_value` in expressions and other references now that it has been declared and initialized.

If we need to know *where* the contents of `integer_value` are actually stored, we can use the unary `&` operator to get its address. It is very unfortunate that the authors of C chose to use &, since this symbol is also used to indicate bit-wise logical AND operations. In any case, in the pointer-related form we interpret the operator '&' as "take address of".

For example, the statement

```
printf("integer_value=%d (address of integer_value=%d)\n",
      integer_value, (int) &integer_value);
```

when executed on a particular computer gives the output:

```
integer_value=13 (address of integer_value=7143380)
```

Note that a type cast `(int)` is used before the address value in order to print it out.

The address '`7143380`' is simply what happened to be returned when this program was run: the specific value will vary from machine to machine and instance to instance. We often do not really care about the numerical value of the address, BUT this may be important if we have specific hardware registers or ports to deal with.

This is how we can visualize the data storage for this example:

Address (arbitrary example)	*Contents*
…	
7143380 integer_value	13
…	

Declaring a Pointer Variable

For some reason, the creators of the C language chose to use the * symbol to indicate multiplication and to indicate several pointer-related operations. For example, here is how a pointer variable is declared:

```
int *integer_pointer;
```

This tells the compiler to assign storage sufficient to hold the address of an integer, and to associate the storage location with the name `integer_pointer`. Note that this pointer variable has not been initialized, so the address it holds initially is garbage. Then if we assign an address to be held by the pointer variable, like

```
integer_pointer = &integer_value;
```

the pointer variable `integer_pointer` now contains the *address* of the integer variable `integer_value`. Continuing this example, the statement

```
printf("address of integer_value=%d, address stored by
integer_pointer=%d (address of integer_pointer=%d)\n",
  (int) &integer_value,
  (int) integer_pointer,
  (int) &integer_pointer);
```

gives the output

```
address of integer_value=7143380, address stored by
integer_pointer=7143380 (address of integer_pointer=7143316)
```

Address (arbitrary examples)	*Contents*
…	
7143380 integer_value	13
…	
7143316 integer_pointer	7143380
…	

Using "" (the "get value pointed at by" operator)*

We use the unary operator * (yet again!) to cause indirect access to the quantity *located at the address* stored in the pointer variable. Read '*' as "get the value pointed at by".

For example,

```
printf("integer_value=%d, value pointed at by integer_pointer=%d\n",
      integer_value, *integer_pointer);
```

gives the output

```
integer_value=13, value pointed at by integer_pointer=13
```

Pointers and Arrays

An array in C defines a set of consecutive addresses. The declared name of the array is actually a pointer to the first element of the array (lowest address). This means that we can declare a pointer variable and assign it an address within the array, then access other array elements using pointer arithmetic.

```
int integer_array[4]={500,501,502,503};

integer_pointer = integer_array;
```

The statement

```
printf("integer_pointer=%d, *integer_pointer=%d, integer_pointer+1 =%d,
*(integer_pointer+1)=%d\n",
  (int) integer_pointer, *integer_pointer, (int) (integer_pointer+1),
*(integer_pointer+1));
```

gives the output

```
integer_pointer=7143320, *integer_pointer=500, integer_pointer+1
=7143324, *(integer_pointer+1)=501
```

Address (arbitrary examples)	*Contents*
…	
7143380 integer_value	13
…	
7143332 ← integer_array+3	503
7143328 ← integer_array+2	502
7143324 ← integer_array+1	501
7143320 ← integer_array ←	500
7143316 integer_pointer	7143320
…	

Note that *for this example* the size of an `int` variable happens to be 4 bytes, so the compiler has caused the pointer arithmetic to increment properly (+4) according to this type.

Pointers and Function Arguments

One of the important uses of pointers in C is allow “call-by-reference” in addition to the usual “call-by-value” mechanism for function calls.

The default call-by-value process means that the *value* of a variable is passed to a function, and the function sets up its own local storage to hold that quantity while executing the function’s statements. The only way a call-by-value function can change something outside its scope is by its return value or by using a global variable.

If it is necessary or desirable to have a *called* function alter the contents of a variable in the *calling* function, the caller can pass a pointer to that variable as a function argument. The prototype of the called function must also define the argument as a pointer, but then it is able to alter the contents of the variable itself since the called function now knows the variable's address.

For example, let’s say we want a function that will increment each value in an array.

```
void increment_array(int increment, int *arr, int arsize)
{
      int i;
      for(i=0;i<arsize;++i) *(arr++) += increment;
}
```

This function could be called by `increment_array(2,integer_array,4)` , which would cause each of the 4 elements in the `integer_array` to be incremented by 2 (502, 503, 504, 505).

There is one other important detail in this example. Note that the function arguments are kept as *local copies* within the context of the function. For example, the argument `arr` contains a local copy of the pointer to the array, so this local pointer can be manipulated (e.g., `arr++`) without changing the original `integer_array` pointer back in the calling program.

Pointers to Functions

Pointer variables can be declared for any storage type (char, int, float, arrays, structures, etc.), and they can also be declared for functions. This ability to define function pointers is quite useful in embedded programming since we often want to set up vector tables (e.g., interrupt vectors) that turn control over to a subroutine using a jump table.

For example, to declare a pointer variable `test_func` that points to a function that has one integer argument and returns an integer, use

```
int (*test_func)(int);
```

The declared name of a function is actually a pointer to the function. So, if the program includes a function `int do_task(int)` , the address of that function is simply its name, `do_task`. So, we can do the assignment

```
test_func = do_task;
```

and even call the function with an argument by using, for example,

```
integer_value = (*test_func)( 29 );
```

An array of pointers to functions can also be declared. The following is an example showing how this might be used.

```
void do_task_0( )
{
 …
}
```
etc., for other functions

```
/* declare an array of three pointers to void functions
with no arguments */
void (*func_arr[3])(void);

func_arr[0]=do_task_0;
func_arr[1]=do_task_1;
func_arr[2]=do_task_2;
```

The memory arrangement for this example could be:

Address (arbitrary examples)	*Contents*
…	
7143398 func_arr[2]	55396
7143394 func_arr[1]	54560
7143390 func_arr[0]	54390
…	
55396 start of do_task_2()	<machine instructions…>
…	
54560 start of do_task_1()	<machine instructions…>
…	
54390 start of do_task_0()	<machine instructions…>
…	

Storage Arrangements (Little Endian and Big Endian)

In the examples above the size of an int is 4 bytes. One question is whether the 4 bytes comprising the integer are stored with the least-significant byte at the lowest of the four addresses or the least-significant byte in the highest of the four addresses.

To help understand this question we can use pointers and a pointer type cast.

Consider the following statements:

```
int integer_value = 272;
char *character_pointer;

character_pointer = (char *) &integer_value;
```

The pointer `character_pointer` now contains the address of the variable `integer_value`, but what's more, any pointer arithmetic with `character_pointer` will now be done with the size of a `char` instead of the size of an `int`, due to the explicit type cast `(char *)`.

The statement

```
printf("integer_value=%d, bytes[0-4](hex)= %02x %02x %02x %02x\n",
integer_value,
(int) *character_pointer,
(int) *(character_pointer+1),
(int) *(character_pointer+2),
(int) *(character_pointer+3) );
```

produces the output

```
integer_value=272, bytes[0-4](hex)= 10 01 00 00
```

In this example we see that the individual bytes of the integer number 272 are stored 'little endian', meaning the least significant byte containing hex 10 (= decimal 16) is stored at the lowest address, the next byte (hex 01 = 256) is stored higher, and so forth.

Intel microprocessors use little endian storage, while most other processors (including Motorola) use big endian. The result of the code segment above would be "00 00 01 10" on a big endian machine.

We will do a homework problem involving converting data from big endian to little endian and vice versa.