Reading Material

From *Teach Yourself C*:

- Chapter 6: 6.1 – 6.3
A. Pointers to Variables

Each variable declared in a program is associated with a memory address. A pointer is a certain type of variable that holds the information about the memory address of a “normal” variable. We then say that the pointer points to the value of the “normal” variable.

If a variable var has been defined, then we can get its memory address using: &var.

If pointer p represents the memory address of a variable, then we can get the value stored at that address, i.e., the value of the variable, with *p. Since *p refers to the value at the location where the pointer points to, we can assign a data type to it: e.g., int *p; this is how we declare a pointer p.

We can think of the two pointer operators, & and * as being inverse operations that allow us to switch between address pointers and values of variables. They also cancel each other: var ≡ *(&var).

Since *p refers to the value stored, we can use such expressions in all the same ways normal variables are used (in arithmetic operations, “if” statements, etc). However it is very important always to remember which part of the memory we are accessing, which values we are changing, and which remain unchanged.

Examples: Let us say that a pointer p points to an address where the value stored is 3, i.e., *p=3.

If we pass the value to a variable var: var=*p; and then change the value of var: var=var+10; we have NOT changed *p. The reason is that after the first command p is NOT equal to &var.

If instead var=3; and we associate p with the address of var: p=&var;, this means *p=3. If then we change the value of var: var=var+10, then *p also changes and becomes equal to 13.

Let us now deal with just pointers. We declare three pointers to integers and go through a sequence of assignments:

```c
int *pa, *pb, *pc;
*pa=3;
*pb=*pa;
pc=pa;
*pa=*pa+5;
```
The values associated with each pointer now are: \( *pa=8 \), \( *pb=3 \), and \( *pc=8 \). The reasons are that initially the three pointers point to three different addresses, and, after the assignments are executed, pointers \( pc \) and \( pa \) point to the same memory address, whereas pointer \( pb \) points to its original (different) address.

B. Call-by-reference

Pointers are most useful when used with functions. Normally variables that are passed to functions as input parameters are essentially copied as new variables with new memory allocations. This means that any changes to the input parameters within the function do not affect the values of these variables in the main function.

Often though we need the values of variables passes to a function to be changed by the function operations and have these changes be reflected to the main function. This can be achieved if a variable is passed to a function with the use of a pointer. The effect is similar to that of global variables. Most common is the case where we would like a function to have more than one output. Then we can define a function where the input parameters are the values of pointers, and the function is called with the variable addresses:

```c
#include <stdio.h>
#include <math.h>

void swap(int *a, int *b) {
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

int main () {
    int num1,num2,temp;
    num1=5;
    num2=10;
    printf("%i %i \n",num1,num2);
    swap(&num1,&num2);
    printf("%i %i \n",num1,num2);
    return 0;
}
```

Within a function it is also possible to transfer the pointer values to normal variables and go through the operations. However it is important at the end of the function we must remember to transfer the changed values back to the pointers. See the example
below for a variation of the *swap* that uses extra variables. Let us say that we called the *swap* function with two variable addresses: *swap*(&num1,&num2), and the function is written as follows:

```c
void swap (int *a, int *b) {
  int temp, va, vb;
  va=*a;
  vb=*b;
  temp=va;
  va=vb;
  vb=temp;
  *a=va;
  *b=vb;
}
```

Would the result be the same, i.e., would the values of num1 and num2 in the main function be swaped, if you substituted the last two lines of code with:

```c
a=&va;
b=&vb;
```
C. Pointers and One-Dimensional Arrays

Pointers can also be associated with arrays and in this case the pointer points to the address of the 0th element.

When a pointer is defined then a memory space just big enough to hold a value of the data type of the pointer is reserved. However, after the definition of the pointer, we can let the compiler know that we would like that pointer to point to an array of a given size and ask it to allocate memory space big enough to hold all of the array elements. This is done with the malloc command (from memory allocation):

```
pointer-name = (data-type*) malloc (length of the array*sizeof(data-type));
```

Array elements are stored sequentially and their addresses are increasing sequentially. This allows us to access array elements using pointers:

- \( *p \) gives us the value of the 0th element
- \( *(p+1) \) gives us the value of the 1st element
- \( *(p+2) \) gives us the value of the 2nd element
- \( *(p+i) \) gives us the value of the ith element

For example:

```c
int *par;
par = (int*) malloc (10*sizeof(int));
```

then load \( par[i] \) with values. Note that the expressions \( par[i] \) and \( *(par+i) \) are equivalent.

When passing an array to a function with pointers, we use the name of the array without any brackets.

**NOTE:** The advantage of using pointers and allocating memory for a whole array with the malloc command is crucial: it allows us to define arrays the size of which is not known at the time of writing the program, but instead gets defined during run-time. This is called *dynamic memory allocation* and cannot be done by simply declaring an array, because in that case the size (or the maximum possible size) of the array must be known before run-time.

**NOTE:** Whenever you use malloc it is necessary to include one other header file with the statement: `# include <stdlib.h>`. 

D. How to Declare, Open, Close, Write to, and Read Data from a File

Just like programs can be used to input/output information from and to the screen, they can also be used to access (input/output) information in (data) files. Files are actually pointers ...

Declare: Before a file can be opened or accessed it must be declared just like a variable has to be declared before being used. A file is declared as a pointer:

FILE* file-pointer-name;

Open: A file must be opened before information can be taken in or out of them:

file-pointer-name = fopen ("file-name","r");

r indicates that the file is opened as a read–only file. It can be replaced by w for a write–only file or by a to append data to a file that already exists.

Read from: To read data from a file, a command similar to scanf is used:

fscanf (file-pointer-name, "variable-type identifiers", &variable name);

Write to: To write data to a file, a command similar to printf is used:

fprintf (file-pointer-name, "variable-type identifiers", variable name);

Close: A file must be closed after access:

fclose (file-pointer-name);
Programming Hints

• Align first and last line of loop statements and indent the statements to be executed repeatedly.

• Remember to initialize variables used within loop structures.

• To decide which of the two structures you should use for a specific problem, ask yourself whether you know the number of iterations, in which case the for statement is more appropriate, or whether you know what is the condition that determines the termination of the loop, in which case the while statement is more appropriate.

• Try to test your loop structure with plugging-in numbers and working out some simplified cases of the loop’s functionality.