Pointers in C

Objectives:
Having read this section you should be able to:

1. Program using pointers;
2. Understand how C uses pointers with arrays.

Point to Point:
Pointers are a very powerful, but primitive facility contained in the C language. Pointers are a throwback to the days of low-level assembly language programming and as a result they are sometimes difficult to understand and subject to subtle and difficult-to-find errors. Still it has to be admitted that pointers are one of the great attractions of the C language and there will be many an experienced C programmer spluttering and fuming at the idea that we would dare to refer to pointers as 'primitive'!

In an ideal world we would avoid telling you about pointers until the very last minute, but without them many of the simpler aspects of C just don't make any sense at all. So, with apologies, let's get on with pointers.

A variable is *an area of memory that has been given a name*. For example:

```
int x;
```

is an area of memory that has been given the name x. The advantage of this scheme is that you can use the name to specify where to store data. For example:

```
x=lO;
```

is an instruction to store the data value 10 in the area of memory named x. The variable is such a fundamental idea that using it quickly becomes second nature, but there is another way of working with memory.

The computer doe not access its own memory using variable names – instead it uses a *memory map* with each location of memory uniquely defined by a number, called the *address* of that memory location.

A pointer is a variable that stores this location of memory. In more fundamental terms, *a pointer stores the address of a variable*. In more picturesque terms, a pointer points to a variable.

A pointer has to be declared just like any other variable - remember a pointer is just a variable that stores an address. For example,

```
int *p;
```

is a pointer to an integer. Adding an asterisk in front of a variable's name declares it to be a pointer to the declared type. Notice that the asterisk applies only to the single variable name that it is in front of, so:

```
int *p , q;
```

declares a pointer to an int and an int variable, not two pointers.

Once you have declared a pointer variable you can begin using it like any other variable, but in practice you also need to know the meaning of two new operators: **&** and *****. The & operator returns the *address of a variable*. You can remember this easily because & is the 'A'mpersand character and it gets you the 'A'ddress. For example:

```
int *p , q;
```

declares p, a pointer to int, and q an int and the instruction:

```
p=&q;
```

stores the address of q in p. After this instruction you can think of p as pointing at q. Compare this to:

```
p=q;
```

which attempts to store the value in q in the pointer p - something which has to be considered an error.

The second operator * is a little more difficult to understand. If you place * in front of a pointer variable then the result is the value stored in the variable pointed at. That is, p stores the address, or pointer, to another variable and *p is the value stored in the variable that p points at.

The * operator is called the *de-referencing operator* and it helps not to confuse it with multiplication or with its use in declaring a pointer.

This multiple use of an operator is called *operator overload*.

Confused? Well most C programmers are confused when they first meet pointers. There seems to be just too much to take in on first acquaintance. However there are only three basic ideas:

1. To declare a pointer add an * in front of its name.
2. To obtain the address of a variable us & in front of its name.
3. To obtain the value of a variable use * in front of a pointer's name.

Now see if you can work out what the following means:

```
int *a , b , c;
b = 10;
a = &b;
c = *a;
```

Firstly three variables are declared - a (a pointer to int), and b and c (both standard integers). The instruction stores the value l0 in the variable b in the usual way. The first 'difficult' instruction is `a=&b` which stores the address of b in a. After this a points to b. Finally `c = *a` stores the value in the variable pointed to by a in c. As a points to b, its value i.e. 10 is stored in c. In other words, this is a long winded way of writing:

```
c = b;
```

Notice that if a is an int and p is a pointer to an int then :

```
a = p;
```

is nonsense because it tries to store the address of an int, i.e. a pointer value, in an int. Similarly:

```
a = &p;
```

tries to store the address of a pointer variable in a and is equally wrong! The only assignment between an int and a pointer to int that makes sense is:

```
a = *p;
```

Swap Shop:

At the moment it looks as if pointers are just a complicated way of doing something we can already do by a simpler method. However, consider the following simple problem - write a function which swaps the contents of two variables. That is, write `swap(a,b)` which will swap over the contents of a and b. In principle this should be easy:

```
function swap(int a , int b);
 {
  int temp;
  temp = a;
  a    = b;
  b    = temp;
 }
```

the only complication being the need to use a third variable temp to hold the value of a while the value of b overwrites it. However, if you try this function you will find that it doesn't work.

You can use it - `swap(a,b);` - until you are blue in the face, but it just will not change the values stored in a and b back in the calling program. The reason is that all parameters in C are *passed by value*. That is, when you use the `swap(a,b)` function the values in a and b are passed into the function swap via the parameters and any changes that are made to the parameters do <u>not</u> alter a and b back in the main program. To repeat, the function `swap` does swap over the values in a and b within the function, but <u>doesn't do so in the main program</u>.

The solution to this very common problem is to pass <u>not</u> the values stored in the variables, but the <u>addresses</u> of the variables. The function can then use pointers to get at the values in the variables in the main program and modify them. That is, the function should be:

```
 function swap(int *a , int *b);
 {
  int temp;
  temp = *a;
  *a   = *b;
  *b   = temp;
 }
```
Notice that now the two parameters a and b are pointers and the assignments that effect the swap have to use the de-reference operator to make sure that it is the values of the variables pointed at that are swapped. You should have no difficulty with:

```
temp = *a;
```

this just stores the value pointed at by a into temp. However,

```
*a = *b;
```

is a little more unusual in that it stores that value pointed at by b in place of the value pointed at by a. There is one final complication. When you use swap you have to remember to pass the addresses of the variables that you want to swap. That is not:

```
swap(a,b)
```

but

```
swap(&a,&b)
```

The rule is that whenever you want to pass a variable so that the function can modify its contents you have to pass it as an address. Equally the function has to be ready to accept an address and work with it. You can't take any old function and suddenly decide to pass it the address of a variable instead of its value. If you pass an address to a function that isn't expecting it the result is usually disaster and the same is true if you fail to pass an address to a function that is expecting one. For example, calling swap as `swap(a,b)` instead of `swap(&a,&b)` will result in two arbitrary areas of memory being swapped over, usually with the result that the entire system, not just your program, crashes!!

The need to pass an address to a function also explains the difference between the two I/O functions that we have been using since the beginning of this course. `printf` doesn't change the values of its parameters so it is called as `printf("%d",a)` but `scanf` does, because it is an input function, and so it is called as `scanf("%d",&a)`.

Pointers And Arrays:

In C there is a very close connection between pointers and arrays. In fact they are more or less one and the same thing! When you declare an array as:

```
int a[10];
```

you are in fact declaring a pointer a to the first element in the array. That is, a is exactly the same as `&a[0]`. The only difference between a and a pointer variable is that the array name is a constant pointer - you cannot change the location it points at. When you write an expression such as `a[i]` this is converted into a pointer expression that gives the value of the appropriate element. To be more precise, `a[i]` is exactly equivalent to `*(a+i)` i.e. the value pointed at by `a + i` . In the same way `*(a+1)` is the same as `a[1]` and so on.

Being able to add one to a pointer to get the next element of an array is a nice idea, but it does raise the question of what it means to add 'one' to a pointer. For example, in most implementations an int takes two memory locations and a float takes four. So if you declare an int array and add one to a pointer to it, then in fact the pointer will move on by two memory locations. However, if you declare a float array and add one to a pointer to it then the pointer has to move on by four memory locations. In other words, adding one to a pointer moves it on by an amount of storage depending on the type it is a pointer to. This is, of course, precisely why you have to declare the type that the pointer is to point at! Only by knowing that a is a pointer to int and b is a pointer to float can the compiler figure out that :

```
a + 1
```

means move the pointer on by two memory locations i.e. add 2, and

```
b + 1
```

means move the pointer on by four memory locations i.e. add 4. In practice you don't have to worry about how much storage a pointer's base type takes up. All you do need to remember is that pointer arithmetic works in units of the data type that the pointer points at. Notice that you can even use ++ and -- with a pointer, but <u>not</u> with an array name because this is a constant pointer and cannot be changed. So to summarise:

1. An array's name is a constant pointer to the first element in the array that is `a==&a[0] and *a==a[0].`
2. Array indexing is equivalent to pointer arithmetic - that is `a+i=&a[i]` and `*(a+i)==a[i]`.

It is up to you whether you want to think about an array as an array or an area of storage associated with a constant pointer. The view of it as an array is the more sophisticated and the further away from the underlying way that the machine works. The view as a pointer and pointer arithmetic is more primitive and closer to the hardware. In most cases the distinction is irrelevant and purely a matter of taste.

One final point connected with both arrays and functions is that when you pass an entire array to a function then by default you pass a pointer. This allows you to write functions that process entire arrays without having to pass every single value stored in the array - just a pointer to the first element. However, it also tempts you to write some very strange code unless you keep a clear head. Try the following - write a function that will fill an array with random values `randdat(a,n)` where a is the array and n is its size. Your first attempt might be something like:

```
 void randdat(int *pa , int n)
  {
   for (pa = 0 ; pa < n ; pa++ ) *pa = rand()%n + 1;
  }
```

Well, I hope your first attempt wouldn't be like this because it is wrong on a number of counts! The problem is that the idea of a pointer and the idea of an index have been confused. The pointer pa is supposed to point to the first element of the array, but the `for` loop sets it to zero and then increments it though a series of memory locations nowhere near the array. A lesser error is to suppose that n-1 is the correct final value of the array pointer! As before, you will be lucky if this program doesn't crash the system, let alone itself! The correct way of doing the job is to use a `for` loop to step from 0 to n-1, but to use pointer arithmetic to access the correct array element:

```
int randdat(int *pa , int n)
{
  int i;
  for ( i=0 ; i< n ; ++i)
   {
     *pa = rand()%n + 1;
     ++pa;
   }
}
```

Notice how the `for` loop looks just like the standard way of stepping through an array. If you want to make it look even more like indexing an array using a for loop you could write:

```
for(i=0 ; i<n ; ++i) *(pa+i)=rand()%n+1;
```

or even:

```
for(i=0 ; i<n ; ++i) pa[i]=rand()%n+1;
```

In other words, as long as you define pa as a pointer you can use array indexing notation with it and it looks as if you have actually passed an array. You can even declare a pointer variable using the notation:

```
int pa[];
```

that is, as an array with no size information. In this way the illusion of passing an array to a function is complete.