Computer Science 1MC3

Lab 7 – Pointers

A pointer is nothing more then a memory location which stores an address to something. This simple notion is something that characterizes C as a versatile language. Pointers, as we will soon see, have a tremendous amount of use in c programming.

Declaration

There are two ways to declare a pointer.

```
data_Type *var_name;    ~or~  data_Type* var;
```

What we have just done is declare a pointer to a certain type of data type. To demonstrate what we can do with this pointer let consider the following concrete example.

```
int *arrow, real_value=12;
```

Now say we would like to have `arrow` point to our `real_value`, we would need to know a few things.

`*var_name;` refers to the pointer or `var_name`
`&var_name;` refers to the address of `var_name`

All pointers point to addresses, so to have our `arrow` pint to `real_value` we must make `arrow` (a pointer), equal to the address of `real_value`. It is done like this:

```
*arrow=&real_vaule;
```

We now have a pointer to an address, which is what we want. We can now manipulate `*arrow` and `real_value` in the exact same way, `*arrow=5;` would be equivalent to `real_value=5;`.

How things go wrong

Using pointers are very tricky. In fact many people try to avoid pointers unnecessarily complicating their programs! Here are some typical mistakes made when using pointers.

incorrect

```
arrow=&real_value;
arrow=14;
```

This is an attempt to change the value of `real_value` indirectly using the `arrow` pointer. However, `arrow` is an address. Saying `arrow=14` is really setting it to a value at address 14.

correct

```
arrow=&real_value;
*arrow=14;
```

This correct code says change whatever `arrow` is pointing too.

incorrect

```
*arrow=&real_value;
```

This code wouldn't change where `arrow` is pointing to, but the value to which `arrow` is pointing.

correct

```
arrow=&real_value;
```

Passing Pointers

It is possible to pass a pointer to a function. Passing a pointer gives you the advantage of changing the value of variables in other procedures without using more memory expensive global variables. For instance consider the following two pieces of code:

```
#include <stdio.h>

void proc (int *a);

int prog1( void ) {

      int x;

      x=7;

      proc(&x);

      printf("%d \n",x);

      return 0;
}

void proc (int *a) {

      *a=12;

}
```

```
#include <stdio.h>

void proc (int a);

int prog2( void ) {

      int x;

      x=7;

      proc(x);

      printf("%d \n",x);

      return 0;
}

void proc (int a) {

      a=12;

}
```

Notice that the `prog1` uses pointers and `prog2` does not. What will each of these programs print to the screen? Well `prog1` will send a pointer to `x` to `proc` so the value of `x` will be changed by the procedure. So 12 would be printed to the screen. In `prog2` a copy of `x` is made by the `proc` and it is manipulated. However the value of `x` is not changed in the main program so 7 would be printed to the screen.

What we have done in `prog1` is known as **pass by reference**, since we have passed a pointer to a variable that we can manipulate. What we have done in `prog2` is called **pass by value** since we have only passed a value which was copied by the procedure.

Arrays

You may have thought about passing an array to a procedure. Well now we know that this array can be passed by value or by reference. We generally pass arrays by reference since we don't want the procedure to take an array and copy it for its own purposes, this is tremendously wasteful. Also, it is actually not that easy to pass an array by value anyways!

Lets declare an array.

```
int x[]={1,2,3,4,5}
```

What you may not of realized it that arrays are pointers, `x` is an address. So if you had a variable `*a` it would suffice to put `*a=x;` **not** `*a=&x;` (an address of an address does not exist).

```
#include <stdio.h>

void proc (int *a);

int main( void ) {

      int x[]={1,2};

      proc(x);

      printf("%d \n",x[1]);

      return 0;
}

void proc (int *a) {

      a[1]=12;

}
```

This program sends a pointer to array `x` to a procedure. The procedure can now manipulate the values of `x` in any way you would like. Notice that we can say `a[1]` which is equivalent to `x[1]` since `a` and `x` are both pointing to the same memory location.

A Final Note

When using pointers always remember that:

1. A pointer must equal a memory location `a=&x;`
2. When changing the value of what something is pointing to, you must say `*a=12;`
3. The `sizeof(pointer)` is always 4 (an integer).
4. An array name, by itself, is an address. `pointer=array_name`

Homework

1. Edit this program to pass by reference instead of value. (i.e the value that the procedure set should be printed.)

```
#include <stdio.h>

void proc (float point);

int main( void ) {

    float realval=45.67;

    proc(realval);

    printf("%f \n",realval);

    return 0;
}

void proc (float point) {

    point=12.45;

}
```

2. Create a procedure to take an array from your main program and double every value in the array. It will be necessary to pass the length of the array to you procedure.

3. Outline the differences between pass by reference and pass by value.