11 Pointers (Optional)

11.1 Introduction

Pointers are the most powerful, and the most confusing, aspect of "C". They are also essential to successful "C" programming and you will have to master the concepts as some point. Now is good time to try! You have been using *pointers* throughout this course but without actually knowing about them in detail. This section is optional at this stage. Only study this if your are confident in your programming, have completed the basic set of 5 checkpoints are are looking "for more to do".

11.2 What is a Pointer

Since in "C" we have to deal with *addresses* of variables fairly frequently, for example in `scanf`, there are a set of data-types designed to hold and manipulate them; these are *pointer*. As with other variables different types of *pointers* hold different types of *addresses* and in particular each data-type has its corresponding *pointer* type being declared with at additional `*` as follows.

Type	Declaration	Value is
`int` pointer	`int *`	Address of an `int`
`float` pointer	`float *`	Address of a `float`
`double` pointer	`double *`	Address of a `double`
`char` pointer	`char *`	Address of a `char`

Pointers are manipulated by the two operators `&` and `*` which mean

`&`	Address of variable
`*`	Value stored at address

So for example we can write,

```
int i = 5 , j ;                /* Declare two integers i & j */
int *p_i;                      /* Declare an int pointer p_i */
p_i = &i;                      /* Set pointer p_i to address of i*/
j = *p_i + 10;                 /* Add 5 to value pointed as by p_i
                                  and store in j */
```

then this

1. Declares two `int` variables, setting the first one to `5`.
2. Declares an `int` *pointer*.
3. Set the *value* of `p_i` to be the *address* of `i`.
4. Takes the *value* in the *address* held by `p_i` and adds `10`, storing the result in the variable `j`, which has *value* `15`.

similarly check you understand what this does:

```
#include <stdio.h>
#include <stdlib.h>
int main()
  {
   float x = 10 , y ;
   float  *px, *py;
   px = &x;
   py = &y;
   *py = 2*(*px);
   printf("x has value : %f\n",x);
   printf("y has value : %f\n",x,y);
   exit(EXIT_SUCCESS);
  }
```

if not, type it in and see what it does!

These examples show that pointers allow you an alternative method of accessing the *values* in memory. This is the real power, and massive danger of pointers since if *not* used correctly they allow you to write totally opaque code that neither you, or anybody else can work out what it has just done!

11.3 Pointers and Arrays

Although pointers to single variables are useful in function arguments, see below, the main use of pointers is associated with arrays. Consider the declaration

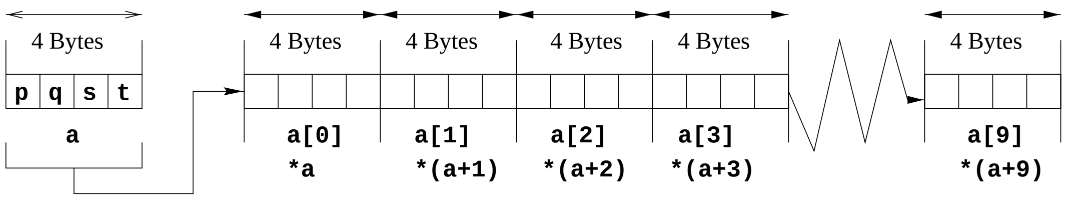
```
float a[10];
```

what *really* happens is that 40 bytes of memory are allocated for the array (as seen before) *and* a `*float` pointer called `a` is created, its *value* is set to the *address* of the first element of the array. Therefore,

$$*\mathtt{a} = \mathtt{a}[0]$$

so the *name* (`a`) of the array, that we have seen before, is *really* a pointer.

If we "add" 1 to the value of a pointer then it "points" at the next value in memory. For our array of 10 `floats` we have that,



so we have two notations for addressing the *values* of array elements with an integer index `i`, these being

$$*(\mathtt{a}+\mathtt{i}) \quad \text{and} \quad \mathtt{a}[\mathtt{i}]$$

which are functionally identical. One makes the pointer structure more visible than the other.

The above example could have used `char`, `int` or `double` arrays. The amount of memory allocated for each type of array would have been different, but the structure and methods of access are identical.

Multi-dimensional arrays are rather more complicated than this are are beyond the scope of this course. It is actually logical, with for example a two-dimensional array being a pointer to and array-of-points, each of which hold the *address* of the first elements of a one-dimensional array. *See textbooks for details.*

11.4 Pointer arithmetic

In the above example we rather casually "added" an integer to a pointer without really thinking, or explaining what this means. With pointers "addition" of "1" actually means *address of the next element*, so if we have

```
float a[10];
float *ptemp;
ptemp = a + 1;
```

then `a` contains the *address* of the first element of the array and `ptemp` the address of the second. Note, the actual *value* of `a` and `ptemp` will differ by the size of a `float`, which is typically 4 bytes!

Similarly if we used a `double` array,

```
double a[10];
double *ptemp;
ptemp = a + 1;
```

then the *value* of `a` and `ptemp` will differ by the size of a `double`, which is typically 8 bytes!

So the meaning of "addition" depends on the type of pointer we have declared. This all works exactly as expected, provided that you always make sure that your pointers "point" at the arrays of the corresponding type, otherwise total chaos occurs!

Pointer arithmetic is also commonly used with the *increment* (`++`) and *decrement* (`--`) operators, for example the following piece of code set an array of a 100 `float` elements to the value 10.

```
float a[100];
float *pa,*pend;
pa = a;
pend = a + 100;
while(pa < pend)
{
  *pa = 10;
   pa++;
}
```

The second line declares two `float` pointers and the third line sets them to the "start" and "end" addresses of the array, so `pa` will be numerically "less than" `pend`. The `while` loop the sets each element of `a` to 10, and *increments* the pointer `pa` until it reaches the end of the array.

People who are bemused, confused and panicked by this are likely to be asking "so what, I can do this with a `for` loop with much less pain, so why bother…". For access of simple arrays, the answer is "no reason at all", but pointer arithmetic allows more flexible access to data and the above `while` loop will actually be computationally faster than the corresponding `for` loop. As you use more complex "C" constructs you will have to use *pointers*.

11.5 Pointers in Function Arguments

In the *Functions II* section we saw how pointer in arguments allow functions to write to variables in the calling program, for example the *rotation* function of,

```
#include <math.h>
void rotation(float *px, float *py, float theta)
{
   float x_new, y_new;

   x_new = (*px)*cos(theta) + (*py)*sin(theta);
   y_new = (*py)*cos(theta) - (*px)*sin(theta);
   *px = x_new;
   *py = y_new;
   return;
 }
```

should now be clear. The arguments are pointers, so contain the *address* of the `x` and `y` variable which are then written to. In calls it is always the *value* of the argument that is passed, but the *value* will be an *address* if the argument is a pointer.

This now should explain why we had to use the `&` in the `scanf` calls.

Note: Many books on "C" *will talk about "pass by value" and "pass by reference". This is an alternative, and in my opinion, more confusing method of looking at argument passing. In* "C" *it is always the "value" of an argument that is passed, and how it is used depends of what type of argument is passed.*

When we pass an array as an argument we normally give the array *name* as in the `dot_product` function we looked at earlier.

```
float dot_product(float a[], float b[], int n)
{
   float dot = 0;
   int i;
   for (i = 0; i < n; i++)
   {
        dot = dot + a[i]*b[i];
   }
   return dot;
```

```
    }
```

However we now know that the array *name* is just a pointer that gives the *address* of the first element. The above function can therefore be declared as:

```
    float dot_product(float *a, float *b, int n)
    {
       float dot = 0;
       int i;
       for (i = 0; i < n; i++)
       {
            dot = dot + a[i]*b[i];
       }
       return dot;
    }
```

It is now clearer what is happening. Then the function is called the *address* of the first element is passed to the function. The function then access elements values offset from this address. Looked at this way the passing of arrays to functions obeys exactly the same rules as other arguments.

Note: The declaration of array arguments given in the previous section *is correct* but is very unusual. All book and almost all programs will use the `float *a` notation rather than the `float a[]`.

What Next?

You have now completed the basic structure of "C" which will allow you to write fairly complex and useful programs. There are three important areas that you have not covered in "C", which will be dealt with next year, these being:

Memory Allocation: So far all your programs have to declare the size of all the arrays they use when the program is written, for example the maximum length of the PDF array in checkpoint 5b. This makes writing very general programs difficult. There are five memory allocation functions, `malloc`, `calloc`, `realloc`, `free` and the rather non-standard `alloca` to get round this problem and allow much greater flexibility. These are reasonable easy to use but do require a good understanding of *pointers*.

Scope of Variable: So far all variable you have used are local to the program or function in which they were declared. Also variable declared within a function are re-created each time the function is called so loosing any previous values. This action can be modified and variable can be declared as `static` or common to a set of functions. "C" has rather complex and non-obvious rules for this.

Structures: Within "C" we can declare data structures, for example two `floats` linked together to form a complex number. This `struct` then effectively become an additional data type. This advanced features allows the programmer to create new date types tailored to the programming task in hand which can considerable simplify the coding of complex tasks. This is perhaps the most complicated aspect of "C" especially when when these additional data-types are arranged in arrays. Again a fundamental understanding of *pointers* is essential.

If you are interested in computing, you are encouraged to look up these topics in your favorite “C” book and try of some programming examples for yourself.