



Computers in Engineering

COMP 208

Pointers

Michael A. Hawker



Function Prototypes

Before using a function C must *know* the type it returns and the parameter types.

Function prototypes allow us to specify this information before actually defining the function

This allows more structured and therefore easier to read code.

It also allows the C compiler to check the *syntax* of function calls.



Function Prototypes

- ✱ If a function has been defined before it is used then you can just use the function.
- ✱ If NOT then you must *declare* the function prototype. The prototype declaration simply states the type the function returns and the type of parameters used by the function.



Function Prototypes

A function prototype has

1. the type the function returns,
2. the function name and
3. a list of parameter types in brackets

e.g.

```
int strlen(char []);
```

This declares that a function called `strlen` returns an integer value and accepts a string as a parameter.



Function Prototypes

- ✱ It is good practice to prototype all functions at the start of the program, although this is not strictly necessary.

```
void swap(int, int);  
int main() {  
    int x, y; ... swap(x, y); ...  
}  
void swap(int x, int y) {...};
```



Swap Two Values

```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```



What Happens?

```
void main () {  
    int a, b;  
    a = 27;  
    b = 103;  
    swap(a, b);  
    printf ("%d    %d \n", a, b);  
}
```



Surprise!

```
>swap
```

```
27    103
```

We wanted to see:

```
103    27
```

What happened here?

This worked in Fortran!

Why not in C?



Parameter Passing

- ✱ It turns out that Fortran and C handle parameters very differently
- ✱ In C all parameters are passed by value
- ✱ The parameters are treated as new local variables that are initialized to the argument values
- ✱ Any changes made are local and do not effect the arguments



Fortran ?

- ✱ Remember how arguments were passed in Fortran?
 - ✱ Expressions or constants had their values put in new local variables representing the parameters (**call by value**)
 - ✱ If the argument was a variable the parameter was treated as an alias for that variable (**call by reference**)



C ?

- ✱ C is more uniform
- ✱ It treats all arguments the same way, the way Fortran treats expressions
- ✱ But ...

That means that we have problems with functions like swap where we want the argument values to change



What's the solution?

- ✱ C allows us to manipulate addresses, called pointers
- ✱ If we pass a pointer as an argument, the value of the argument doesn't change
- ✱ But . . .
 - The value in the cell pointed to could change



Swapping Values in C

```
void swap(int *px, int *py)
{ int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```



Let's Compare

```
void swap(int x, int y)
{ int temp;

  temp = x;
  x = y;
  y = temp;
}
```

```
void swap(int *px, int *py)
{ int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```



What is a Pointer?

- ✱ A pointer is a variable which contains the memory address of another variable.



Declaring a Pointer

To declare a pointer to an integer variable:

```
int *ip;
```

The variable `ip` will be able to store the **address** of an integer cell


In general:

```
type *name;
```




Type

- ✱ We can have a pointer to any variable type (cell “shape”)
- ✱ Once we declare it, the pointer can only be associated with a variable of the specific type we declared



How do we find the value pointed to?

- ✱ If the variable `ip` contains an address, how do we find out what is stored in the cell pointed to?
- ✱ We use a dereferencing operator.
- ✱ The dereference operator `*` returns the “contents of the object pointed to”

How do we get the address of a variable?

Suppose we have an integer variable `x` that contains some value, say 37.

If we want the address of `x`, we can use the `&` operator.

For example we could write

```
ip = &x
```

That is, `ip` contains the address of `x` and `*ip` has the value 37



A Simple Example

```
int x = 1, y = 2;
```

```
int *ip;
```

```
ip = &x;    /* ip gets the address of x */
```

```
y = *ip;    /* y gets assigned 1 */
```

```
x = ip;     /* x gets the its address */
```

```
*ip = 3;    /* x gets assigned 3 */
```



Swapping Values in C

```
void main () {  
    int a, b;  
    a = 27;  
    b = 103;  
    swap (&a, &b);  
    printf ("%d    %d \n", a, b);  
}
```



"The more you know..."

- ✿ In the `scanf` function, we used `&a` to read into the variable `a`
- ✿ Without the `&a`, the value of `a` would not have changed
- ✿ `scanf` needs the reference to the variable in order to change it, therefore we need to have the `&`.



Pointers and Arrays

- ✿ Pointers are very closely linked to arrays in C
- ✿ There is a duality between an array, which is a block of memory cells, and a pointer to a memory location
- ✿ The array is a pointer to the first of these cells



Pointers and Arrays

```
int a[10], x;
```

```
int *pa;
```

```
pa = &a[0]; /* pa gets address of a[0] */
```

```
x = *pa; /* x gets contents of pa (a[0]) */
```




Pointer Arithmetic

- ✱ C allows us to add integer values to pointers
- ✱ Adding a value, i , to a pointer gives the address of the i th memory cell following
- ✱ If the pointer, `arr`, references an array `arr + i` is equivalent to `arr[i]` (but more efficient)



More on pointer arithmetic

- ✱ Suppose `pa` is a pointer to an array
 - ✱ What does `pa + 1` do?
 - ✱ What does `(pa) + 1` do?
 - ✱ What does `* (pa+1) + 1` do?



A Note on Parameters

- ✱ We want to pass an array as a parameter to the sorting algorithm
- ✱ We want the contents of the array to be modified
- ✱ Do we have to pass the parameter as a pointer?
 - ✱ Nope, an array is a pointer.



Finding Max

To find the smallest, just invert the comparison

```
int find_biggest(int arr[], int size)
{
    int index_of_big = 0, i;

    for(i = 0; i < size; ++i)
        if(arr[i] > arr[index_of_big])
            index_of_big = i;

    return index_of_big;
}
```



Linear Search

The algorithm to find the location of the a value in an array has a similar structure.

```
int linear_search(int val,
                  int arr[], int size) {
    int i;

    for(i = 0; i < size; ++i)
        if(arr[i] == val)
            return i;

    return -1;
}
```



A Sorting Algorithm

- ✱ Computers are frequently used to sort data stored in arrays
- ✱ We will soon look at several different ways this can be done
- ✱ For now we will look at a sorting algorithm that illustrates the use of a swap



Bubble Sort

- ✱ We can compare pairs of values working backwards through the array.
- ✱ When two values are out of order, swap them
- ✱ When we are finished one pass, the smallest value is at the front of the array (it “bubbles” down)
- ✱ We repeat this process until all the values are in order



Bubble Sort

```
void bubble_sort(int arr[], int size) {  
    int i, j;  
    for (i=0; i<size-1; i++) {  
        for (j=size-1; j>i; --j) {  
            if (arr[j] < arr[j-1]) {  
                swap (&arr[j], &arr[j-1]);  
            }  
        }  
    }  
}
```


Binary Search for Sorted Arrays

```
int binary_search(int val, int arr[], int size)
{
    int left = 0, right = size, middle;

    do {
        middle = (left + right) / 2;

        if(arr[middle] < val)
            left = middle + 1;
        else if(arr[middle] > val)
            right = middle - 1;
        else
            return middle;

    } while(left <= right);

    return -1;
}
```



Random Numbers

- ✱ Many applications use random numbers
- ✱ Before we go on with pointers, lets have a quick look at how to generate a sequence of numbers that looks random
- ✱ It really isn't random, hence **pseudorandom numbers**



Pseudo Random Numbers

Make sure to include the needed libraries

```
#include <stdlib.h>
#include <time.h>
```

The first step is to seed the pseudo-random number generator.

For testing, we can always reproduce the same sequence if we start with the same seed.

For “production” we might choose an arbitrary seed

```
srand((unsigned int) time(NULL));
```



Pseudo-Random Sequence

Once the random number generator has been seeded, the next number can be generated with

```
rand()
```

This generates a number in the range from 0 to `RAND_MAX` (which is often 32767 but may vary with different implementations)



Restricting the Range

To generate a random real number between 0 and 1, you could use

```
(double) rand() / RAND_MAX
```

To get a number in the range from x_0 to x_1 , you could generate a number between 0 and 1 as above and then scale it as follows

```
num * (x1-x0) + x0
```



File Input

- ✱ To read from a file, you first declare a file pointer of type `FILE`

```
FILE* name;
```

- ✱ `FILE` is uppercase because it is an implementation dependent macro

- ✱ Once declared, you use `fopen` to associate the name with an actual file

```
FILE* datafile = fopen("test.data", "r");
```



File Specifications

```
FILE* datafile = fopen("test.data", "r");
```

- ✱ The “r” specifies that the file is to be read from
- ✱ To write to a file, we use “w”
- ✱ To append to a file, we use “a”



Reading and Writing

- ✱ In place of scanf, we use fscanf and specify the file to read from

```
fscanf (datafile, "%f%d", &value, &count);
```

- ✱ To write to a file we use fprintf instead of printf and also specify the file

```
FILE* results;  
results = fopen("test.result", "w");  
fprintf (results, "The average of %d values "  
"is %f.\n", totalcount, ave);
```




Closing a File

- ✱ As in Fortran, when we are finished reading from or writing to a file, the file should be closed:

```
fclose(results) ;
```

Writing to a File

- ✱ When we open a file, the filename is assigned a nonzero value if the operation is successful and a value of zero if it fails.

```
FILE* results = fopen("test.result",  
                      "r");  
  
if (results)  
{  
    . . .  
}  
else printf("Could not open the file");
```