

## Computational Physics

### Memory and Pointers

Comments and questions to [John Rowe](#).

### Memory

Remember how variables are stored in memory. For example the computer might choose to store two variables `x` and `y` like this:

Byte number:	400   401   402   403   404   405   406   407
Value:	8.4                       11.3
	<---       x       ---->   <---       y       ---->

### Pointers

A *pointer* is a variable that contains the address of another variable.  
Kernighan and Ritchie.

**Pointers say it where something is stored in memory, not what its value is.**

#### Pointers to variables

Pointers to variables are the most-obvious and least-used use of pointers. We use them here merely to illustrate how pointers work, the more useful examples follow below.

C provides a special operator (`&`) to find the address of a variable and a special type of variable (known as a **pointer**) to hold that address. If we assume that `p` is a pointer and that, as above, the variable `y` is stored in memory starting at location 404 then the line:

```
p = &y;
```

would set the value of `p` to 404. We can even print out the value of `p` for debugging purposes using the `%p` operator:

```
printf("p has the value: %p\n", p);
```

The `*` operator does the opposite of `&`: given a pointer `p` whose value is `&x` `*p` is just a synonym for `x`, ie `*p` means 'the variable whose address is `p`'. We may then use `*p` **anywhere** where we might want to use `x`:

```
void example(void) {
    float x =17.1, y, *p, *q; /* p, q are pointers */

    p = &x; /* The value of p is now 400, the address of x.
             If we do anything with or to *p it's
             doing it to x: */

    printf(" p: %p\n *p: %f\n", p, *p);

    y = *p; /* Exactly the same as y = x; */
    *p = 3.14159; /* Exactly the same as x = 3.14159; */

    /*
     * Now let's make p point to y:
     */
    p = &y; /* The value of p is now 404, the address of y */
    *p = 1.414; /* Exactly the same as y = 1.414; */
    q = &x; /* The value of q is now 400, the address of x */

    unwise_function(p, q); /* The same as unwise_function(&y, &x); */

    /*
     * Now x has the value 73.2, y has the value 2.5
     */
}
```

```

    */
}

void unwise_function(float *pp, float *qq) {
/* What are the values of pp and qq? */
    *pp = 2.5;
    *qq = 73.2;
}

```

Notice the difference between:

`p = &x;` change the value of `p`

`*p = 3.14159;` change the value of the variable whose address is `p`.

Now see what happens when we pass `p` and `q` to the function `unwise_function()`. Inside `unwise_function()` `pp` and `qq` have the same value as `p` and `q` had in `main`, ie `&y` and `&x`. Therefore when `pp` and `qq` are dereferenced we are actually accessing `y` and `x` inside `main` and `y` and `x`'s values gets changed.

The process of finding the variable whose address is `p` is called **dereferencing** `p`.

## Use structures, not pointers to variables!

At first sight `unwise_function()` looks like the answer to the question "How do I write a function that returns more than one value?". (One return value is easy of course as in `x = sin(y)`.) But as the name `unwise_function()` implies it isn't! Remember two of our golden rules:

- A function does just one job considered from the perspective of its caller.
- Everything about a "thing" is contained in that thing's structure.

So if we find ourselves wanting to pass pointers to two or more variables to get round the "functions return at most one value" limitation it's a sure sign that we're either trying to do two things at once or that we've broken the "one thing, one structure" rule.

In practice the only time we'll ever use pointers to ordinary variables is when we call the `scanf` functions.

## Pointers to structures

These work just as we would expect. Let's remind ourselves of our ellipse structure from last week's lecture:

```

typedef struct ellipse {
    float centre[2];
    float axes[2];
    float orientation;
} Ellipse;

```

This takes twenty bytes. A particular Ellipse, say called "ellie", might be stored starting at byte 800 in which case its members would be laid out as follows:

Byte number:	800 - 803	804 - 807	808 - 811	812 - 815	816 - 819
Used for:	centre[0]	centre[1]	axes[0]	axes[1]	orientation
	<----- ellie ----->				

(If we had an array of Ellipses, `ellipsearray`, starting at byte 3000 then `ellipsearray[0]` would start at byte 3000, `ellipsearray[1]` at byte 3020, `ellipsearray[2]` at byte 3040, etc.)

A statement such as:

```
ellie.orientation = 11.4;
```

would mean that the computer would go to the start of `ellie` (800), go along 16 bytes to byte 816 and write the value 11.4 into the four bytes 816 - 819.

Alternatively, we could declare a pointer to an Ellipse, assign it the address of `ellie` (800) and use that pointer, as in the following rather contrived example:

```

#define PI 3.14159265358979

int main() {
    Ellipse ellie, *ep; /* ellie is a structure, ep just a pointer */

    ep = &ellie;
    (*ep).orientation = PI/4;
}

```

The notation `(*ep).orientation` is rather awkward and pointers to structures are used so often they have their own notation:

```
ep->orientation = PI/4;
```

The fact that they have their own notation should tell us how important pointers to structures are!

## Recap: the key point

The key to understanding pointers is simple but vital: the value of a pointer is the address in memory of another variable (most commonly a structure).

### Several pointers can point to the same object

Until now, the only way to access or change the value of a variable or structure member was to "hard code" it into the program:

```
ellie.orientation = PI/4;
```

But now any number of pointers may have the same value, meaning that they refer to the same underlying object. For example, if `p`, `q` and `r` are all pointers to an `Ellipse` and we have the statement:

```
p = q = r = &ellie;
```

then, until their values are changed, all three point to `ellie` and the following statements all have the same effect:

```
ellie.orientation = PI/4;

p->orientation = PI/4;
q->orientation = PI/4;
r->orientation = PI/4;
```

This gives us a great deal of flexibility, but with flexibility comes the possibility of confusion!

## Passing pointers-to-structures to functions

We can use pointers to deal with the obvious weakness of our 'structure' functions so far: that they only pass the structure values **into** the function without passing the new values back. We do this by passing a pointer to a structure, to a function.

This is an example of "several pointers pointing to the same object": although the pointer in the calling function and the pointer in the called function are different variables they both have the same **value** and so they both point to the same thing. When we dereference them we are therefore accessing the same original structure.

As an example, we will add a new member, `area`, to our structure and modify last week's `print_area()` function to calculate the area and store it in the structure. We will call the new function `calculate_area()`. In addition, we will define a new function to read in the values of the ellipse, imaginatively called "read\_ellipse()":

```
#include <stdio.h>

#define PI 3.14159265358979
typedef struct ellipse {
    float centre[2];
    float axes[2];
    float orientation;
    float area;
} Ellipse;

void read_ellipse(Ellipse *el);
void calculate_area(Ellipse *el);

int main() {
    Ellipse ellie;

    read_ellipse(&ellie);
    calculate_area(&ellie);

    printf("The area of the ellipse is: %f\n", ellie.area);
    return 0;
}

void read_ellipse(Ellipse *el) {
    int i;
```

```

printf("Centre? (x,y)\n");
scanf("%f %f", &el->centre[0], &el->centre[1]);

for(i = 0; i < 2; ++i) {
    do {
        printf("Length of axis number %d ( > 0 )?\n", i + 1);
        scanf("%f", &el->axes[i]);
    } while (el->axes[i] <= 0);
}

printf("Orientation to the vertical?\n");
scanf("%f", &el->orientation);
}

void calculate_area(Ellipse *el) {
    el->area = PI * el->axes[0] * el->axes[1];
}

```

Looking at the above example from the bottom upwards, the first thing we see is that `calculate_area()` is passed a pointer to the original ellipse and so is able to set and use quantities such as `el->area`, `el->axes[0]`, etc. See how changing the value of `el->area` inside the function `calculate_area()` is changing the original ellipse inside `main()`.

We pass just one pointer to `calculate_area()` and it is able to access as many of the members as we want.

Similarly, for `read_ellipse()` we pass just one pointer, to the ellipse, rather than three or five individual pointers to `ellipse.orientation`, `ellipse.centre[0]`, etc.

`&el->axes[i]` means what we want it to: it is equivalent to "`&(el->axes[i])`" as "`->`" binds closer than anything else.

## Our old friend FILE \*

We've actually used pointers already, when using a declaration such as

```
FILE *infile;
```

This means `infile` is a pointer to something. In practice this is almost certainly a structure and somewhere inside `stdio.h` is a section like:

```
typedef struct {
    Huge structure definition here...
} FILE;
```

## Pointers and arrays: incomplete dereferences

A one-dimensional array needs one index to identify a particular value (dereference it), a two-dimensional array two indices, etc.

C has a useful feature where we can follow an array name with fewer indices than it requires (e.g. a one-dimensional array with no indices, a two-dimensional array with zero or one index, etc.) Such a dereference is **incomplete** as it needs one or more further indices (dereferences) to complete it.

The basic rule is simple: the value of an incomplete dereference is the location of that part of the array in memory. This is most often used when passing all or part of an array to a function which expects an array of the same or fewer dimensions as its argument.

Most obviously, the name of an array is the address of the start of the array. This is true how ever many dimensions an array has.

For a two-dimensional array we have the additional option of specifying just the first index,. This allows us to think of a two-dimensional array such as:

```
float twodarray[NX][NY];
```

either as one two-dimensional array or as `NX` one-dimensional arrays. Both views are equally valid.

In this case `twodarray[j]` can be thought of as the  $j^{\text{th}}$  one-dimensional array and can be passed to any function expecting a one-dimensional array, since it requires one more index to complete the dereference.

In this example we see how a two-dimensional character array can also be treated as an array of one-dimensional arrays:

```

char name[MAXNAMES][MAXLEN];
int i;

for(i = 0; i < MAXNAMES; ++i) {
    printf("Please enter name %d without spaces\n". i + 1);
}

```

```
scanf("%s", name[MAXNAMES]);
}
```

**Important consequence:** when we pass the name of an array, or any incomplete array dereference, to a function we are passing a **pointer** to the array, or that part of the array, so changing the array in the function changes the original.

## The NULL pointer

C has a special value called NULL which it can be guaranteed no legitimate pointer will ever have.

We've seen this already: `fopen()` returns NULL if the open of a file failed:

```
FILE *fin;

fin = fopen("idatin.dat", "r")
if (fin == NULL) {
    /* Print an error message and die */
}
else {
    /* Do something useful */
}
```

In general, any function that returns a pointer to something useful will return NULL when it fails, and that should always be checked for.

## When pointers attack: random pointer values

**In order to dereference a pointer it is absolutely essential that its value has been explicitly set to the address of a legitimate variable.**

Why? Well, remember first that uninitialised variables values random values. So when we try to dereference an uninitialised pointer we will try to read from or write to a random memory location which will give us a random result. Worse still, if we write to this location some other variable in some other function will have its value changed and our program will fail or the whole machine crash.

The safest bet is whenever we declare a pointer, initialise it to NULL:

```
float *p = NULL;
```

Using a value of NULL by mistake is not safe but it is probably less dangerous than using any other value.

## Symptoms of pointer errors

### SIGSEGV, SIGBUS and STATUS\_ACCESS\_VIOLATION crashes

If we are lucky, a pointer error will cause the program to crash with one of the above errors. (Initialising pointers to NULL improves the chance of this happening which is why we recommend it.)

### Random results

If we are unlucky, a pointer error will cause our program to behave randomly. Putting in `printf()` statements to debug may cause the problem to go away...

### What to look for

In either case, look for:

- Uninitialised pointers, or pointers to something that no longer exists.
- Bad arguments to `scanf()`
- Array index errors.

## A couple of more-advanced errors

### Variation: pointers to things that no longer exist

A subtle, and therefore dangerous, version of the above is the pointer to something that used to exist but doesn't any more. Consider the following:

```
#include <stdio.h>
/*
 * Accept a number in the range 1-12 and return the name of
 * the month. (Contains a bug!)
 */
char *month_name(int month) {
```

This example is the big one. The following two are more advanced and less common.

```

char *months[] = { "January", "February", "March", "April",
                  "May", "June", "July", "August",
                  "September", "October", "November", "December" };

if (month > 0 && month <= 12)
    return months[month - 1];
else
    return NULL;
}

```

## Discussion

1. What happens to variables and arrays inside a function when that function is called?
2. What happens to them when that function exits?
3. Why is this a problem in the above example?
4. [Advanced.] Can you think of a fix to this problem?

## Copying strings vs. copying pointers

Consider the following program:

```

#include <stdio.h>
#include <string.h>

#define LENGTH 200
struct mystruct {
    float val;
    char str[LENGTH];
};

main() {
    struct mystruct struct1 = { 3.14259, "This is the value of Pi!" };
    struct mystruct struct2;

    struct2 = struct1;
    strncpy(struct1.str, "Oops! No it isn't!", LENGTH);
    printf("%s\n%s\n", struct2.str, struct1.str);
}

```

Each structure has 204 bytes of memory so the line

```
struct2 = struct1;
```

copies all 204 bytes of memory from `struct1` to `struct2`. Thus the program prints out:

```

This is the value of Pi!
Oops! No it isn't!

```

As each structure has its own 200 bytes of memory for the string and the call to `strncpy` only effects `struct1`. Suppose we decide to do things a bit better and allow a variable amount for the string:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct mystruct {
    float val;
    char *str;
    int length;
};

main() {
    char stringbuffer[100];
    struct mystruct struct1 = { 3.14259, stringbuffer, 100 };
    struct mystruct struct2;

    strncpy(struct1.str, "This is the value of Pi!", struct1.length);
    struct2 = struct1;
    strncpy(struct1.str, "Oops! No it isn't!", struct1.length);
    printf("%s\n%s\n", struct2.str, struct1.str);
}

```

Now each structure has just 12 bytes of memory and the value of `struct1.str` is just the value of `stringbuffer`, for example 600.

Now the line:

```
struct2 = struct1;
```

copies just 12 bytes of memory and `struct1.str` and `struct2.str` both have the value 600, ie they both point to the same piece of memory. The output of this program is:

```
Oops! No it isn't!  
Oops! No it isn't!
```

## Using pointers to express relationships between things

Consider a genealogy program. It might contain something like:

```
typedef struct {  
    char surname[MAXLEN];  
    char firstname[MAXLEN];  
    int gender; /* 0 male, 1 female, 2 "other" */  
} Person;
```

This has a couple of issues to do with maximum name length, etc, that we will deal with in the next week or two but one very obvious one: how do we express family relationships?

One way is to have two **pointers** to the parents and an array of pointers to the children. (This has a slight niggle in that we can't use "Person" until it has been defined, we work around that by giving the structure its own name.) The result is:

```
typedef struct person {  
    char surname[MAXLEN];  
    char firstname[MAXLEN];  
    int gender; /* 0 male, 1 female, 2 "other" */  
    struct person *father;  
    struct person *mother;  
    struct person *children[MAXCHILDREN];  
} Person;
```

It is important to notice that `father` and `mother` are pointers and `children` is an array of **pointers**, not an array of structures. (If this were the case our structures would be infinitely big!)

To print out the name of a person's father we might say:

```
printf("%s %s's father's name is %s %s\n", p->firstname, p->surname,  
      p->father->firstname, p->father->surname);
```

### A slightly more professional way

In reality we would probably represent the parents by an array of length two:

```
typedef struct person {  
    char surname[MAXLEN];  
    char firstname[MAXLEN];  
    int gender; /* 0 male, 1 female, 2 "other" */  
    struct person *parents[2];  
    struct person *children[MAXCHILDREN];  
} Person;
```

To print out the name of a person's father we would then say:

```
printf("%s %s's father's name is %s %s\n", p->firstname, p->surname,  
      p->parents[0]->firstname, p->parents[0]->surname);
```

Finally, although here we have used pointers to describe relationships between objects of the same type (people), it is even more common to use them to describe relationships between objects of different type.

## Optional material

### Passing pointers to strings (and arrays) to functions

We remind ourselves that **the name of an array is a synonym for the address of its first element.**

```
#include <stdio.h>
void printstr(char *);
main() {
    char str[] = "abcdxyz";

    printstr(str);
    strcpy(str, "Hi!");
    printstr(str);
}

void printstr(char tmp[]) {
    int i;
    for(i = 0; tmp[i] != '\0'; ++i)
        putchar(tmp[i]);
    putchar('\n');
}
```

The first line of `main` both declares and initialises `str`. The empty brackets `[]` tell the compiler to make the array equal to the size of the string (ie 8, the number of characters plus one for the final zero).

The pointer to the start of the string is then passed to `printstr` which goes over the string printing out each character in turn until it comes to the end.

We could have just written the test part of the `for` loop as just `"tmp[i]"`, rather than `"tmp[i] != '\0'"`. This is because a 'logical' test is just an integer value which is considered to be 'true' if that value is non-zero and Strings are terminated with a binary zero (often written `'\0'`)

This indented section is a more-advanced topic.

## Changing the value of the local pointer variable

Although the above version of `printstr()` is the most intuitive, a popular idiom is to increment the local pointer variable:

In an argument list the two constructs `char tmp[]` and `char *tmp` mean the same thing. This is because they both say "this is the start of an array which has been defined somewhere else."

```
void printstr(char *tmp) {
    for(; *tmp; ++tmp)
        putchar(*tmp);
    putchar('\n');
}
```

### The argument is still a copy of the original

The loop in this version of `printstr` looks strange because we are changing `tmp`. But remember, `tmp` is a variable just like any other. Let's assume that the string pointed to by `str` starts at memory location 400:

Byte number:	400   401   402   403   404   405   406   407
Value:	'a'   'b'   'c'   'e'   'x'   'y'   'z'   \0

(Remember, each printable character from the 'normal' latin alphabet is assigned an integer value in the range 1-127 which is the value actually stored in the computer's memory.)

When we call `printstr` then `tmp` is given its own (temporary) piece of memory to store its value (say bytes 640-643) and that value is initialised to 400:

Byte number:	640   641   642   643
Value:	400
	<- tmp (in printstr) ->

As we keep adding one to `tmp` it has the values 401, 402, etc until it reaches 407 and it stops because `*407` is zero. Then when `printstr` returns the memory at location 640 is freed and is available for use by variables as other functions are called and so the changes made to `tmp` are lost.

## Pointer arithmetic

We've seen how we can add one to a pointer and it does what we expect (and what we would like: since a char takes one byte adding one to `tmp` takes us to the next character). But what about floats which on our systems

take four bytes? Suppose we have an `float` array called `x`:

Byte number:	800   801   802   803   804   805   806   807
Value:	-12.8   73.4
	<----- x[0] -----> <----- x[1] ----->

Then `x` is a constant pointer of value 800 and if we have the following code:

```
float *p = x; /* 800 */
++p;
printf("%p\n", p);
```

at the end of this `p` has the value **804** not 801, ie adding one to a pointer always makes it point to the next item in the array. The compiler can do this because we declared `p` as `float *p` so it knew that the thing `p` pointed to took four bytes per item.

## Pointer subtraction

Pointers to items in the same array can even be subtracted:

```
float *p = &x[2];
float *q = &x[7];

printf("%d\n", q - p);
```

will print out 5.

- `array` is a synonym for `&array[0]`.
- `(array + n)` actually means `(array + n * sizeof *array)`
- `array[n]` is a synonym for `*(array + n)`.

## Summary

- A pointer is a variable whose value is the address of another variable.
  - They are mainly used to point to structures.
  - Several pointers can point to the same thing.
  - We are in big trouble if they end up pointing to random places.
  - Pointers can express relationships between things.
-