A slow introduction to pointers in C

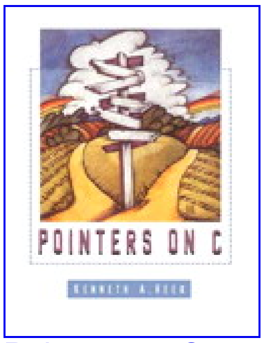
The C11 programming language has a very powerful feature for addressing and modifying memory which, for now, we'll casually term *"pointers in C"*.

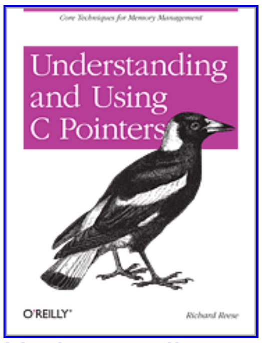
- If used correctly pointers can enable very fast, efficient, execution of C programs.
- If misunderstood or used incorrectly, pointers can make your programs do very strange, incorrect things, and result in very hard to diagnose and debug programs.

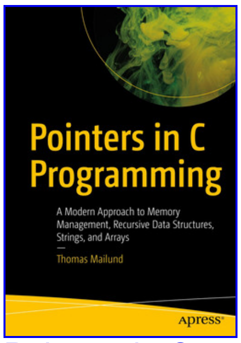
The primary role of pointers - to allow a program (at run-time) to access its own memory - sounds like a useful feature, but is often described as a very dangerous feature.

There is much written about the power and expressiveness of C's pointers, with general agreement that C's pointers are a threshold concept in Computer Science.

(Recently much has been written about Java's lack of pointers. More precisely, Java *does* have pointers, termed *references*, but the references to Java's objects are so consistently and carefully constrained at both compile- and run-times, that little information about the run-time environment is exposed to the running program, and little can go wrong).


Pointers on C, Kenneth Reek, Addison-Wesley, 636pp, 1998.

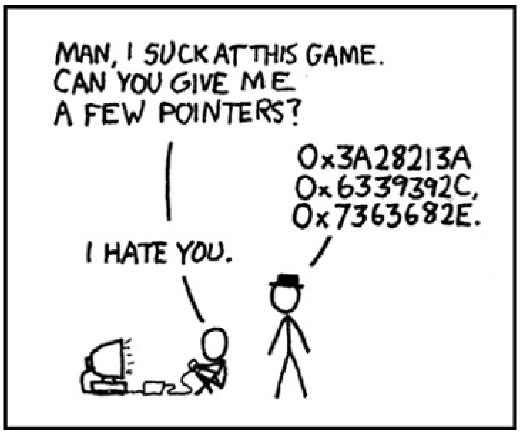

Understanding and Using C Pointers, Richard Reese, O'Reilly Media, 226pp, 2013.


Pointers in C Programming, Thomas Mailund, APress, 537pp, 2021.

What are pointers?

We know that C has both "standard" variables, holding integers, characters, and floating-point values, termed *scalar* variables.
In addition, we've seen *arrays* and *structures* of these, termed *aggregate* variables.

Let's follow this simplified explanation:



- We understand that variables occupy memory locations (1 or more bytes) of a computer's memory.
- Each variable requires enough (at least) bytes to store the values the variable will (ever) need to hold. For example, on typical desktop and laptop computers a simple C **int**eger will require 4 bytes of memory. However a **bool** value, only *requiring* 1-bit, will typically occupy 1 byte.
- Similarly, an array of 100 integers, will require 400 bytes of *contiguous* memory - there is no padding between elements.
- Computers have a large amount of memory, e.g. our lab computers have *16 gigabytes* of memory (16GB), or nearly 17.1 *billion* bytes.
- Each of a computer's memory bytes is uniquely numbered, from 0 to some large value. Each such number is termed the byte's *memory address*.
- We often refer to memory locations as just *addresses* and the action of identifying an address as *addressing*.

With these points in mind, we can make 3 simple statements:

1. **Pointers are *variables* that *hold* the address of a memory location.**
2. **Pointers are *variables* that *point* to memory locations.**
3. **Pointers (usually) *point* to memory locations being used to hold variables' values/contents.**

The & operator, the *address-of* operator, the *ampersand* operator

The punctuation character **&**, often pronounced as the *address-of operator*, is used to find a variable's address.

For example, we'd pronounce this as:

```
int total;

.... &total ....
```

"the address of total", and if the integer variable *total* was located at memory address 10,000 then the value of *&total* would be 10,000.

We can now introduce a variable named *p*, which is a *pointer to an integer* (pedantically, *p* is a variable used to store the address of a memory location that we expect to hold an integer value).

```
int total;
int *p ;

    p = &total ;
```

If the integer variable *total* was located at memory address 10,000 then the value of *p* would be 10,000.

If necessary (though rarely), we can print out the *address of a variable*, or the *value of a pointer*, by first *casting* it to something we can print, such as an *unsigned integer*, or to an *"generic"* pointer:

```
int total;
int *p     = &total ;

    printf("address of variable is:  %lu\n", (unsigned long)&total );
    printf(" value of pointer p is:  %lu\n", (unsigned long)p );
    printf(" value of pointer p is:  %p\n", (void *)p );
```

Dereferencing a pointer

We now know that a pointer may point to memory locations holding variables' values.

It should also be obvious that if the variable's value (contents) changes, then the pointer will *keep* pointing to the same variable, (which now holds the new value).

We can use C's concept of *dereferencing a pointer* to determine the value the pointer points to:

```
int total;
int *p      = &total ;

    total   =  3;

    printf("value of variable total is:       %i\n", total );
    printf("value pointed to by pointer p is: %i\n", *p );

    ++total;        // increment the value that p points to

    printf("value of variable total is:       %i\n", total );
    printf("value pointed to by pointer p is: %i\n", *p );
```

Even though the variable's value has changed (from 3 to 4), the pointer still points at the variable's location.

The pointer first pointed at an address containing 3, and the pointer kept pointing at the (same) address which now contains 4.

Dereferencing a pointer, *continued*

We now know that *changing the value that a pointer points to* does not change the pointer (good!).

Now we'd like to *change the value held in the address that the pointer points to*

Similarly, this will not change the pointer itself.

```
int total;
int *p      = &total ;
int bigger;

    total   =  8;

    printf("value of variable total is:       %i\n",   total );
    printf("value pointed to by pointer p is: %i\n\n", *p );

    *p  =  *p + 2 ;      // increment, by 2, the value that p points to

    printf("value of variable total is:       %i\n",   total );
    printf("value pointed to by pointer p is: %i\n\n", *p );

    bigger  =  *p + 2 ; // just fetch the value that p points to

    printf("value of variable total is:       %i\n", total );
    printf("value of variable bigger is:      %i\n", bigger );
    printf("value pointed to by pointer p is: %i\n", *p );
```

An array's name is an address

When finding the address of a scalar variable (or a structure), we precede the name by the *address of* operator, the ampersand:

```c
int total;
int *p = &total ;
```

However, when requiring the address *of an array*, we're really asking for the address of the first element of that array:

```c
#define  N     5

int totals[N];

int *first  = &totals[0];    // the first element of the array
int *second = &totals[1];    // the second element of the array
int *third  = &totals[2];    // the third element of the array
```

As we frequently use a pointer to *traverse* the elements of an array (see the following slides on pointer arithmetic), we observe the following equivalence:

```c
      int *p = &totals[0] ;
// and:
      int *p = totals ;
```

That is: *"an array's name is synonymous with the address of the first element of that array."*

Pointer Arithmetic

Another facility in C is the use of *pointer arithmetic* with which we may *change a pointer's value* so that it points to *successive* memory locations.

We employ pointer arithmetic in the same way that we specify numeric arithmetic, using **++** and **— —** to request pre- and post- increment and decrement operators.

We generally use pointer arithmetic when accessing *successive* elements of arrays.

Consider the following example, which initializes all elements of an integer array:

```
#define  N     5

int totals[N];
int *p = totals; // p points to the first/leftmost element of totals

    for(int i=0 ; i<N ; ++i) {
        *p = 0;  // set what p points to to zero
        ++p;     // advance/move pointer p "right" to point to the next integer
    }

    for(int i=0 ; i<N ; ++i) {
        printf("address of totals[%i] is:  %p\n", i, (void *)&totals[i] );
 printf("  value of totals[%i] is:  %i\n", i,          totals[i] );
    }
```

How far does the pointer move?

It would make little sense to be able to "point anywhere" into memory, and so C automatically 'adjusts' pointers' movement (forwards and backwards) by values that are *multiples of the size of the base types* to which the pointer points(!).

In our example:

```
for(int i=0 ; i<N ; ++i) {
    *p = 0;   // set what p points to to zero
    ++p;      // advance/move pointer p "right" to point to the next integer
}
```

p will initially point to the location of the variable:

- *totals[0]*, then to
- *totals[1]*, then to
- *totals[2]* ...

Similarly, we can say that *p* has the values:

- *&totals[0]*, then
- *&totals[1]*, then
- *&totals[2]* ...

Combining pointer arithmetic and dereferencing

With *great care* (because it's confusing), we can also combine pointer arithmetic with dereferencing:

```
#define N    5

int totals[N];
int *p = totals ;

    for(int i=0 ; i<N ; ++i) {
        *p++ = 0; // set what p points to to zero, and then
                  // advance p to point to the "next" integer
    }

    for(int i=0 ; i<N ; ++i) {
 printf("value of totals[%i] is:  %i\n", i, totals[i] );
    }
```

In English, we read this as:

> *"set the contents of the location that the pointer p currently points to the value zero, and then increment the value of pointer p by the size of the variable that it points to"*

Similarly we can employ pointer arithmetic in the control of *for* loops. Consider this excellent use of the preprocessor:

```
int array[N];
int n, *a;

#define FOREACH_ARRAY_ELEMENT  for(n=0, a=array ; n<N ; ++n, ++a)

    FOREACH_ARRAY_ELEMENT {
        if(*a == 0) {
            .....
        }
    }
```

Functions with pointer parameters

We know that pointers are simply variables.
We now use this fact to implement functions that receive *pointers as parameters*.

A pointer parameter will be *initialized with an address* when the function is called.

Consider two equivalent implementations of C's standard *strlen* function - the traditional approach is to employ a parameter that "looks like" an array; new approaches employ a pointer parameter:

```
int strlen_array( char array[] )
{
    int   len = 0;

    while( array[len] != '\0' ) {
        ++len;
    }

    return len;
}
```

```
int strlen_pointer( char *strp )
{
    int   len = 0;

    while( *strp != '\0' ) {
        ++len;
        ++strp;
    }
    return len;
}
```

```
int strlen_pointer( char *strp )
{
    char   *s = strp;

    while( *s != '\0' ) {
        ++s;
    }
    return (s - strp);
}
```

During the execution of the function, any changes to the pointer will simply change what it points to.
In this example, *strp* traverses the *null-byte* terminated character array (a string) that was passed as an argument to the function.

We are not modifying the string that the pointer points to, we are simply accessing adjacent, contiguous, memory locations until we find the *null-byte*.

Returning a pointer from a function

Here we provide two equivalent implementations of C's standard *strcpy* function, which copies a string from its source,*src*, to a new destination,*dest*.

The C11 standards state that the *strcpy* function function must return a *copy* of its (original) destination parameter.

In both cases, we are returning a copy of the (original) *dest* parameter - that is, we are *returning a pointer* as the function's value.

We say that *"the function's return-type is a pointer"*.

```
char *strcpy_array( char dest[], char src[] )
{
    int   i = 0;

    while( src[i] != '\0' ) {
 dest[ i ] = src[ i ];
 ++i;
    }
    dest[ i ] = '\0';

    return dest;     // returns original destination parameter
}
```

```
char *strcpy_pointer( char *dest, char *src ) // two pointer para
{
    char *origdest = dest;  // take a copy of the dest parameter

    while( *src != '\0' ) {
        *dest = *src;       // copy one character from src to des
        ++src;
        ++dest;
    }
    *dest = '\0';

    return origdest; // returns copy of original destination para
}
```

Note:

- in the array version, the function's *return type* is a pointer to a *char*. This further demonstrates the equivalence between array names (here,*dest*) and a p to the first element of that array.
- in the pointer version, we *move* the *dest* parameter after we have copied each character, and thus we must first save and then return a *copy* of the paran original value.
- if *very careful*, we could reduce the whole loop to the statement **while((*dest++ = *src++));**
  But don't.