C Pointers for Experts

Sean McNealy

April 9, 2017

1 Introduction

Pointers are not scary. If you've sent arguments to a python function or "new"ed up a Java class you've used pointers. In many languages there isn't a way to **not** use pointers. C just makes you think about them. This is an introduction to using pointers in C, without assuming you want to know about all the types, if/else, switch, precompiler, headers, and many other features. This is for the programmer who has experience in a language that handles pointers differently and who wants to learn how C does it or any student that has a graduate class project due in a week and "Oh God no! It's in C!"

2 Strings

Strings are intuitive data structures found in every programming language. In C they can be used as teaching examples because there are simple functions to write strings to output streams or read strings from input streams or program arguments. However, they can also be used to show some of the worst methods to mess with pointers. Operating directly on strings is how many security vulnerabilities are created. It is important to understand what not to do before writing production code.

The string "Hello World!" can be stored in memory several ways. The most common, even though it is quite limiting, is ASCII using 1 byte per character. In order to use that string in our program there will be a location in memory that has the following values:

H	e	l	l	o		W	o	r	l	d	!	null
0x48	0x65	0x6C	0x6C	0x6F	0x20	0x77	0x6F	0x72	0x6C	0x64	0x21	0x00

And indeed, if we run "`od -A x -t x1 hello`" to view a compiled hello world program in hex we can see that sequence of numbers in the program:

```
...
0005f0 01 00 02 00 48 65 6c 6c 6f 20 77 6f 72 6c 64 21
000600 00 48 65 6c 6c 6f 20 77 6f 72 6c 64 20 61 67 61
...
```

In this program the string "Hello world!" exists between the memory locations `0x5f4` and `0x600`, taking 13 bytes of memory. This program will be loaded from the disk into memory when it is executed. When we want to do anything with this string, like print it to the `stdout` output stream, the program will reference it by its memory location. And here we have our first example of a pointer! Let's see it used in the hello world program.

Listing 1: hello.c

```c
#include <stdio.h>
/* Simple Hello World! program. */

void print_hello_world(){
    puts("Hello world!");
}

void print_hello_world_again(){
    puts("Hello world again!");
}

int main()
{
    print_hello_world();
    print_hello_world_again();

    return 0;
}
```

This is a simple C program that will to the standard output stream:

```
Hello world!
Hello world again!
```

To compile this program run "`gcc -Wall -Wpedantic -g -o hello hello.c`", and run the program "`./hello`".

Looking at this program, the main program calls 2 functions that write constant strings to standard out using the `puts` function defined in `stdio.h`. It may appear that we are passing the entire string "Hello world!" to the `puts` function. We are not! Let's check the argument type `puts` expects using the man page by running "`man -s3 puts`". We'll also contrast it with `putchar`, a function that writes a single character.

```
int putchar(int c);

int puts(const char *s);
```

So `puts` expects a pointer to a single character. `const` here only promises us that the characters will not be changed when calling the function, a `char*` or `const char*` are allowed arguments. What we can tell from this is that when the API takes a `char*` as an argument it is always assumed to be a string. This is because, as seen on `putchar`, we would not have to use a pointer in the case we write a single character. The C language does not require all `char*`s to be

strings, and you can write functions that take pointers to single characters, but by convention you would not pass character pointers around and it would be very confusing to those reading your program. The same convention does not apply to more complex types, but it helps most frequently on strings anyway.

To see this string pointer in action we compile the program into assembly using `gcc`'s `-S` option. Each line in assembly code can easily be translated to the actual machine binary that can execute on the target CPU. Lines with `movl`, `nop`, and `call` are actual instructions that will be sent to the CPU by high and low voltage patterns to control which logic gates are activated. Directives beginning in '.' are not instructions, but tell how to lay out non-instruction pieces of memory necessary for the program (or are just comments, like '.file').

Listing 2: hello.s

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world!"
        .text
        .globl  print_hello_world
        .type   print_hello_world, @function
print_hello_world:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   print_hello_world, .-print_hello_world
        .section        .rodata
.LC1:
        .string "Hello world again!"
        .text
        .globl  print_hello_world_again
        .type   print_hello_world_again, @function
print_hello_world_again:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC1, %edi
        call    puts
        nop
```

```
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   print_hello_world_again, .-
            print_hello_world_again
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        call    print_hello_world
        movl    $0, %eax
        call    print_hello_world_again
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0
            20160609"
        .section        .note.GNU-stack,"",@progbits
```

The assembly program places our 2 strings in memory as null-terminated strings using the `.string` directive. It also declares `.LC0` and `.LC1` to be the memory location of where those strings are stored. Inside the `print_hello_world` function the memory location `.LC0` is stored in the register `edi` (the register the first argument goes into by convention), and then the function `puts` is called. `puts` will start reading characters from our string pointer in `edi` and write characters to the output stream until it reaches a null character.

Thinking about string pointers this way, we can implement our own version of `puts` using `putchar` and incrementing the pointer as we go along the string. We just take a starting pointer and add characters to the output stream until the pointer points at a null terminator.

Listing 3: puts.c

```c
#include <stdio.h>
/* We can write a string character by character by incrementing
    its pointer. */

void our_own_puts(const char* s){
    // while the single character s points to is not null
    while(*s != 0){
        // putchar the character s is pointing to
        putchar(*s);
```

```
        // increment the address stored in s,
        // so it points to the next character
        s++;
    }
    // puts writes a newline to the stream at the end of the
        string
    putchar('\n');
}

int main()
{
    const char* hello_world = "Hello world!";
    our_own_puts(hello_world);
    our_own_puts(hello_world);
    return 0;
}
```

3 Using Pointers

Many features of C's pointers will be intuitive to a programmer of other languages. If a pointer is passed to a function the called function may change the data being pointed to. If two threads are using the same pointer it may be necessary to implement some read and write locking.

In Java, for example, every variable except primitives is actually dealing with a pointer. If that variable is null instead of a memory address you will find the all too familiar NullPointerException at runtime when accessing methods or fields. The same behavior can be see with C pointers, and let's see a program that does that. The function `strlen` tells us the length of a string. It has one argument, a `const char*` type and returns an number.

```
size_t strlen(const char *s);
```

Listing 4: npe.c

```
#include <string.h>
/* Cause a segmentaton falut. */

int main()
{
    char* null_pointer = NULL;
    int i = strlen(null_pointer);

    return 0;
}
```

`NULL` is defined as `(void*)0`, simply a zero value that will not give a compiler warning when assigning to a pointer variable. Running this program causes a segmentation fault, where the program has tried to access memory outside its memory segment. Here is some output when debugging using `gdb`.

```
Program received signal SIGSEGV, Segmentation fault.
```

```
(gdb) bt
#0  strlen () at ../sysdeps/x86_64/strlen.S:106
#1  0x0000000000400542 in main () at npe.c:7
```

So on line 7 in the function `main`, `strlen` is called with a null pointer. The `strlen` function tries to use this pointer. The operating system detects a memory access request to the bad address and closes the program entirely, returning the segmentation fault signal to the shell that executed the program. You can haldle this signal yourself without entirely exiting, but in most cases this default behavior is best, and avoiding a segfault in the first place is the easiest solution.

4 Allocating Memory

So let's say we want to combine the two strings "Hello" and "world" into one string that we send to the output stream. We could define a space in memory we want to copy both strings into using something like `char[100]`. This gives us a 100 character space on the stack memory area that we can safely do anything we want in. In the next program we copy strings into that space as well as edit specific characters to add a space, exclamation mark, and the null terminator.

Listing 5: hello_static.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Hello World! program that generates the output string
   from "Hello" and "world" strings. */

int main()
{
    const char* hello = "Hello";
    const char* world = "world";

    char hello_world[100];

    // hello_world is a char* pointing to a space of 100
        characters we have allocated
    // it can be used as the destination of a strcpy operation
    strcpy(hello_world, hello);

    // we can access specific places in memory
    hello_world[strlen(hello)] = ' ';

    // we can add to the pointer and use it as a char* copy
        destination
    strcpy(hello_world + strlen(hello) + 1, world);

    hello_world[strlen(hello) + 1 + strlen(world)] = '!';
    hello_world[strlen(hello) + 1 + strlen(world) + 1] = '\0';

    puts(hello_world);
```

```
    return 0;
}
```

Note that the `hello_world` string is used as a `char*` argument to `strcpy` function calls, but it is also used with the array accessor. The following two lines of code are equivalent, as the array accessor is able to check the data type of `hello_world`.

```
hello_world[strlen(hello)] = ' ';

*(hello_world + strlen(hello)) = ' ';
```

Both just write a " " (space) character to the same address. In many architectures that have large words like 32 and 64 bit, this is a simple indirect addressing mode, which can fit in a single instruction. This is pointer addition in action. Whatever the pointer type, you can access into memory this way.

In the static allocation version of this function we don't use all 100 characters in this space and we could allocate a smaller amount. 13 would have been enough. But what if we had only allocated 10? This is dangerous, as `strcpy` would happily copy all 13 characters into the space we allocated, writing over other important information. Sometimes the return address will be overwritten, and your function will return not to where it should but will try to execute some random location in memory. The solution to this is to never use `strcpy`, but its responsible sibling `strncpy` instead, which is safer since it takes an argument of the size of memory allocated and will not overrun that space.

We may also want to combine two strings that we don't know the size of beforehand. In that case, 100 characters may not be enough. This would be very difficult using statically allocated memory. So we'll let the operating system give us dynamically allocated memory on the heap. The function `malloc` will allocate memory whatever size we ask for and return a pointer to the beginning of that memory. Like the `char[100]` before, we can do whatever we want inside that memory.

Listing 6: hello_heap.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Hello World! program that generates the output string
   from "Hello" and "world" strings (or any other strings)
   passed in as arguments. */

int main(int argc, const char** argv)
{
    if(argc != 3){
        puts("Usage: hello_heap word1 word2");
        return 0;
    }
    const char* hello = argv[1];
    const char* world = argv[2];
```

```
    // remember how much memory we malloc, for strncpy
    size_t hello_world_len = strlen(hello) + 1 + strlen(world) +
         2;
    char* hello_world = (char*)malloc( hello_world_len );
    if(hello_world == NULL){
      // malloc may not give us memory, in that case exit
      return -1;
    }
    strncpy(hello_world, hello, hello_world_len);
    hello_world[strlen(hello)] = ' ';
    strncpy(hello_world + strlen(hello) + 1, world,
        hello_world_len - strlen(hello) - 1);
    hello_world[strlen(hello) + 1 + strlen(world)] = '!';
    hello_world[strlen(hello) + 1 + strlen(world) + 1] = '\0';

    puts(hello_world);

    free(hello_world);
    return 0;
}
```

This example uses `strncpy` even though it's simple to prove we have allocated enough memory for these copy operations. This is considered good programming even if it is unnecessary. An experienced C programmer will notice this example leaves out `sizeof(char)` that would be multiplied by the number of characters to find the `size_t hello_world_len` we pass to `malloc`, but we assume `sizeof(char)` = 1.

Note that we would never really want to combine strings like this. Since we are outputting to a stream, the following accomplishes the same goal much more efficiently. And `snprintf` will write to an allocated `char*` just as easily.

```
printf("%s %s!\n", argv[1], argv[2]);
```

5 Structs

Pointers to structs have a simple access method that makes reading code easier. Both a hard to read way `(*p).field` and an easier way to read `p->field` are shown in the function that prints the struct out to show they are equivilant. It is easy to declare a struct on the stack if it is only used by functions called from the current scope. In this example `employee_1` is the better way to declare the struct and send it to the print function. However, we could return `employee_2` to another function if we trust it to `free` the memory when it is done. If we returned a pointer to `employee_1` the memory could be corrupted before it is used since it is using the same stack memory later computations would use.

Listing 7: struct.c

```
#include <stdlib.h>
#include <stdio.h>
/* Pointers and structs. We store one struct on the stack
   and one on the heap. */
```

```c

typedef struct {
    char* name;
    int number;
} Employee;

void print_employee(Employee* e){
    // dereference pointers like this
    printf("Name: %s\n", e->name);
    // ugly way to do the same thing
    printf("Number: %i\n", (*e).number);
}

int main()
{
    Employee employee_1;
    employee_1.name = "John Smith";
    employee_1.number = 1234;

    Employee* employee_2 = (Employee*)malloc(sizeof(Employee));
    if(employee_2 == NULL){
      puts("Failed allocating memory");
      return -1;
    }
    employee_2->name = "Jane Smith";
    employee_2->number = 5678;

    // both employees can be sent to the print
    // function that accepts a pointer to a struct
    print_employee(&employee_1);
    print_employee(employee_2);

    free(employee_2);

    return 0;
}
```

Our struct contains a `char*`, which is not the memory location where "John Smith" is stored. The actual characters are stored just like "Hello world!" was stored in the compiled program, and our struct contains a pointer to that string.

Pointers that are not `char*` may seem weird when adding integers. They actually jump full sizes of the type pointed to. So a pointer to an array of structs will go to the next struct when adding one. But a pointer can be cast to any other type of pointer without issue. To add any arbitrary number to an address, cast to `char*`. While `void*` may work, it is undefined and might not work on another system or compiler and you should be getting compiler warnings. Here is an example with 3 ways to access into an array of structs.

Listing 8: arr_struct.c

```c
#include <stdlib.h>
#include <stdio.h>
/* Pointers and arrays of structs. Adding and casting. */

typedef struct {
```

```
    char* name;
    int number;
} Employee;

void print_employee(Employee* e){
    printf("Name: %s\n", e->name);
    printf("Number: %i\n", e->number);
}

int main()
{
    const int num_employees = 2;
    Employee* employees = (Employee*)malloc(num_employees *
        sizeof(Employee));
    if(employees == NULL){
      puts("Failed allocating memory");
      return -1;
    }
    employees[0].name = "John Smith";
    employees[0].number = 1234;

    employees[1].name = "Jane Smith";
    employees[1].number = 5678;

    // these do the same thing
    // first is finding a pointer to the first element
    print_employee(&employees[0]);
    // second is just using the pointer to the array, which is
    // equal to the pointer to the first element
    print_employee(employees);

    // these also do the same thing
    // adding 1 to a Employee* increases the pointer by sizeof(
        Employee)
    print_employee(employees + 1);
    // or if cast to a char*, we can add sizeof(Employee)
    // which adds directly to the address
    char* char_ptr_to_employees = (char*)employees;
    char* char_ptr_to_employee_2 = char_ptr_to_employees +
        sizeof(Employee);
    print_employee((Employee*) char_ptr_to_employee_2);
    // or the simple way that worked for the first employee
    print_employee(&employees[1]);

    free(employees);

    return 0;
}
```

6 Function Pointers

Function pointers are just like the other pointers so far. We declare a pointer to a function that has both a return type and all its arguments. Within a data type, here everything is an int, it's easy to implement map and reduce functions

that apply argument functions to an array. The function `map_reduce` takes pointers to a mapping and a reducing function as arguments.

Listing 9: function.c

```c
#include <stdio.h>
/* Function pointers example. */

int add_1(int x){
    return x + 1;
}

int noop(int x){
    return x;
}

int sum(int x, int y){
    return x + y;
}

int mul(int x, int y){
    return x * y;
}

int* map(int* list, int(*f)(int), size_t len){
    for(int i = 0; i < len; i++){
      list[i] = f(list[i]);
    }
    return list;
}

int reduce(int* list, int(*f)(int, int), size_t len){
    int acc = list[0];
    for(int i = 1; i < len; i++){
      acc = f(acc, list[i]);
    }
    return acc;
}

int map_reduce(int* list, int(*mapper)(int),
        int(*reducer)(int, int), size_t len){
    return reduce(map(list, mapper, len), reducer, len);
}

int main()
{
    int list1[4] = {1,2,3,4};
    printf("add_1 and sum = %i\n",
        map_reduce(list1, add_1, sum, 4));

    int list2[3] = {2,2,2};
    printf("noop and mul = %i\n",
        map_reduce(list2, noop, mul, 3));

    return 0;
}
```

7 Additional Reading

`const` is important and should be used correctly. `http://publications.gbdirect.co.uk/c_book/chapter8/const_and_volatile.html`

Think you've got all of this? Try the advanced version combining too many types and pointers. `http://c-faq.com/decl/spiral.anderson.html`