

Gathering of Gray Presents:

An Introduction to Programming for Hackers
Part VI - Pointers, Data Structures and Dynamic
Memory

By Lovepump, 2004

Visit:

www.gatheringofgray.com

Goals:

At the end of Part VI, you should be a thorough understanding of pointers, references and data structures.

Review:

Please ensure you understand the following terms:

- Function Calls
- Variable Scope
- Return Values

If you are unsure of any of these terms, go back and review Part V now.

Memory:

In Part I, we learned about memory in the x86 architecture. It is a series of “boxes”, each able to hold a number between 0 - 255 (0x00 - 0xFF). Each box has an address. A memory address in x86 is one word, or 4 bytes so memory addresses range from 0x00000000 - 0xFFFFFFFF.

A memory “box” can store a number that can be used directly, or can store an address to “point” at another piece of memory. Why would we want to “point” somewhere, when we can just refer to it directly? One reason is for the memory address pointed at to be available for other pieces of code.

Pointer and References:

We learned earlier that you can declare a variable in C:

```
int x;
```

This declares an integer, *x*. We can use *x* in our calculations, take input to it, print it, etc. Similarly we can also do this:

```
int *x;
```

This declares a **pointer** to an integer. The **pointer** is called *x*. We use the ***** to indicate a pointer. This **declaration** creates a **pointer** called *x*, not an integer. *x* does not store an int. It stores a memory location that points to an int. We can't use this to store an integer yet, because there is no where to store it. A diagram will help:



Here we have created our pointer to an int, x. Let's pretend it was created in memory location 2000. Since we have done nothing else, the value of x is unknown, so it really doesn't point anywhere for us yet. Using x at this point would not be wise. We need to initialize our pointer.

```
#include <stdio.h>

int main(int argc, char **argv) {

    int y;
    int *x;

    printf("x before initialization = 0x%X\n", x);

    x = &y;

    printf("After initialization, x = 0x%X, &y = 0x%X\n", x, &y);

    y = 3;

    printf("y = %d\n", y);

    *x = 25;

    printf("Now y = %d\n", y);

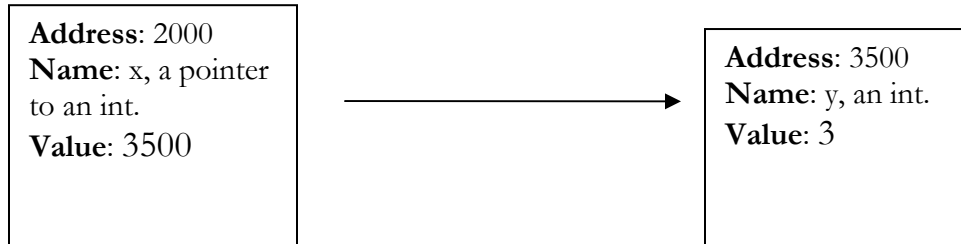
}
```

In the code above, we create an int, y, and a pointer to an int x. The first printf outputs the value of x before it is initialized. The next line is very important. It introduces a new operator - **&**. The **&** (ampersand) means "**reference**". If you put the **&** before a variable, it returns the **address of the variable**, not the value. It is sometimes called "the address of operator".

The line:

```
x = &y;
```

Sets our pointer, x , to the **address of** y . The next `printf` is designed to show both x and the address of y . They should be the same now. If we assume $y = 3$, this is our diagram:



As you can see, x now “points” to y . The **value** of x is the **memory address** of y .

Now that x is initialized, we can use it. First, we use the old method of directly setting a variable:

```
y = 3;
```

Then, we use the pointer to set the variable. Remember that ‘ x ’ holds a memory address. We can’t say:

```
x = 25;
```

This will change where x points. Instead, we use the `*` operator again:

```
*x = 25;
```

The `*` operator says “set the value x points to”. Here is my output of the code above:

```
-sh-2.05b$ ./point
x before initialization = 0x0
After initialization, x = 0xFEf28CE4, &y = 0xFEf28CE4
y = 3
Now y = 25
```

Your memory addresses will likely be different, but the output will be similar. See that the value in x is a word sized address. It is not the value 3, or 25. The int ‘ y ’ lives at address 0xFEf28CE4 in this example.

Recall back to our lesson on scanf. We saw the & operator, but didn't know why we used it:

```
scanf("%d", &i);
```

Now we can say that scanf expects a **pointer to a variable**, not a variable. In this case, we want to scanf into i, so we must give scanf the address where i lives. The & operator is the "address of" operator, so &i tells scanf where i lives.

One special note about pointers. We learned about arrays in a previous part, but there is a relationship between pointers and arrays. If we declare an array:

```
int stuff[100];
```

It is now possible for us to access the array via the array index, such as stuff[12]. An interesting note is that 'stuff' with no index is actually a pointer to the start of the array. This can prove interesting and we will explore this more in 'pointer arithmetic'.

Take a look at some pointer and reference examples:

```
static int
next_signal(struct sigpending *pending, sigset_t *mask)
{
    unsigned long i, *s, *m, x;
    int sig = 0;
```

Taken from signal.c of the Linux kernel, this code snippet shows two pointers. The unsigned long declaration creates two "regular" unsigned longs, i and x, and two pointers to unsigned longs, s and m.

```
char *svRunRegKey;
    if(g_bIsWinNT) {
        svRunRegKey="SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run";
    } else {
        svRunRegKey="Software\\Microsoft\\Windows\\CurrentVersion\\RunServices";
    }
```

In this code we see a pointer to a char declared, svRunRegKey. Taken from our favourite remote admin suite, Back Orifice 2k.

```
struct iphdr *ip;
struct udphdr *udp;
```

The code above, from knock-0.2 shows the creation of two pointers. `ip` points to a struct `iphdr`. `udp` points to a struct `udphdr`. It seems they are pointers to an ip header and a udp header respectively, but what is “struct”? We will learn about struct’s a little later in this part. For now, we will learn more about pointers.

All this learning about pointers may seem pointless (unintentional humour). Why would we want to create a pointer to a variable? Why not just create a variable. There are a number of reasons. The first good reason has to do with functions. If you remember from the last part, a function takes an argument, or arguments and returns a value. There are limitations in what can be done to arguments and in the return value. From what we saw, functions cannot perform tasks directly on our ‘main’ variables due to scope. Here’s an example:

```
/* pointless.c */
#include <stdio.h>

int point_func(int x);

int main(int argc, char **argv) {

    int a = 0;

    point_func(a);

    printf("a = %d \n", a);

}

int point_func(int x) {

    a = 14;

}
```

We have a function called `point_func` that we want to set our variable `a = 14`. Looks good, right? Let’s compile it:

```
-sh-2.05b$ gcc pointless.c -o pointless
pointless.c: In function `point_func':
pointless.c:18: error: `a' undeclared (first use in this
function)
pointless.c:18: error: (Each undeclared identifier is reported
only once
pointless.c:18: error: for each function it appears in.)
```

Oops. The compiler is telling us 'a' is undefined in line 18, function 'point_func'. That can't be right because we declared a as integer above in 'main'. Oh, right. Variable scope is the problem. 'a' is not in scope in point_func. Let's try and modify our code slightly:

```
/* pointless.c */
#include <stdio.h>

int point_func(int *x);

int main(int argc, char **argv) {

    int a = 0;

    point_func(&a);

    printf("a = %d \n", a);

}

int point_func(int *x) {

    *x = 14;

}
```

The change is in the prototype for point_func. It no longer takes an int as an argument. It now takes a pointer to an int (int *x). Since we changed the prototype, we have to change the way we use it as well. In main, we now call point_func like this:

```
point_func(&a);
```

Since point_func is expecting a pointer to an int, we must use the address of operator (&) for our int a. This now tells point_func where our int, a, lives in memory.

When point_func receives the pointer, it now know where the int a lives in physical memory. Scope rules don't apply here. point_func can directly change a. So when we say:

```
*x = 14;
```

We are changing the value pointed at by 'x'. 'x' points at the home of 'a', so we should see a = 14 when we return. Let's have a look:

```
-sh-2.05b$ gcc pointless.c -o pointless
-sh-2.05b$ ./pointless
a = 14
```

It worked. We had a function modify a variable that was “out of scope”. This is one very important use of pointers.

Structures:

Earlier we saw a pointer declared:

```
struct iphdr *ip;
```

What does ‘struct’ mean? It means structure. A structure is a like a grouping of variables that we wish to use as a unit. For example:

```
struct s_data
{
    char name[50];
    char address[100];
    int grade;
    int age;
};
```

In this example we have defined a new data type called ‘s_data’. Our new type is a collection of 4 pieces of information: name, address, grade and age. Notice the important syntax element - the ; after the closing brace.

Now we can declare a variable of our new type:

```
struct s_data student;
```

‘student’ is now a variable structure of type s_data. It is also common to see ‘inline’ declarations of structures:

```
struct s_data
{
    char name[50];
    char address[100];
    int grade;
    int age;
} student;
```

This declares ‘student’ as an s_data in one line. To refer to the individual elements of student, we use the ‘.’ operator:

```
student.name = "Chris Kringle";
student.address = "123 North Pole";
student.grade = 12;
student.age = 17;
```


Cool. We can now use the elements of student like any other variables. We can pass student as an argument to a function, but it's important to note that the 'struct' keyword must be used in the prototype like this:

```
int checkgrades(struct s_data student);
```

We can also declare a pointer to our struct like this:

```
struct s_data *student;
```

Pointers to struct's work differently than pointers to regular variables. To access the individual data elements of the structure, no * is required. The other difference is in the element operator. Instead of '.', when using pointers we use:

```
struct s_data *student;

student->name = "Different Name";
student->grade = 8;
```

Note that the only difference between using an actual struct and a pointer to a struct is the "element" operator. When using a pointer, use ->, when using a struct, use '.'

We will be using struct's more in the next part of this series. Let's see some examples before we move on:

```
struct radeon_regs {
    /* CRTC regs */
    u32 crtc_h_total_disp;
    u32 crtc_h_sync_strt_wid;
    u32 crtc_v_total_disp;
    u32 crtc_v_sync_strt_wid;
    u32 crtc_pitch;
    u32 crtc_gen_cntl;
    u32 crtc_ext_cntl;
    u32 dac_cntl;

    u32 flags;
    u32 pix_clock;
    int xres, yres;

    /* DDA regs */
    u32 dda_config;
    u32 dda_on_off;

    /* PLL regs */
    u32 ppll_div_3;
    u32 ppll_ref_div;
    u32 vclk_ecp_cntl;
```

```

        /* Flat panel regs */
        u32 fp_crtc_h_total_disp;
        u32 fp_crtc_v_total_disp;
        u32 fp_gen_cntl;
        u32 fp_h_sync_strt_wid;
        u32 fp_horz_stretch;
        u32 fp_panel_cntl;
        u32 fp_v_sync_strt_wid;
        u32 fp_vert_stretch;
        u32 lvds_gen_cntl;
        u32 lvds_pll_cntl;
        u32 tmds_crc;
        u32 tmds_transmitter_cntl;

#ifdef __BIG_ENDIAN
        u32 surface_cntl;
#endif
};

```

Wow. That's a big one. This one was included to show that struct's can be as large as you like. This one is from the ATI Radeon driver "radeonfb.c" in the Linux 2.6 kernel. It appears to hold all of the Radeon's GPU registers.

```

struct tcphdr
{
    u_int16_t th_sport;          /* source port */
    u_int16_t th_dport;        /* destination port */
    tcp_seq th_seq;            /* sequence number */
    tcp_seq th_ack;            /* acknowledgement number */
#ifdef __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4;           /* (unused) */
    u_int8_t th_off:4;          /* data offset */
#endif
#ifdef __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4;          /* data offset */
    u_int8_t th_x2:4;           /* (unused) */
#endif
    u_int8_t th_flags;
#define TH_FIN      0x01
#define TH_SYN      0x02
#define TH_RST      0x04
#define TH_PUSH     0x08
#define TH_ACK      0x10
#define TH_URG      0x20
    u_int16_t th_win;           /* window */
    u_int16_t th_sum;           /* checksum */
    u_int16_t th_urp;           /* urgent pointer */
};

```

Ah-hah. Here is one of our examples from earlier. This is tcphdr, the TCP packet header we will see more of when we dive in to network programming. It is found in tcp.h.

Dynamic Memory Allocation

Earlier, we learned how to create a variable, then point a pointer at that variable. This seems like a lot of work. There must be a way to create some memory space for our variable and then point at that space. There is. The function we will look at is `malloc()` (memory allocate). Here is the prototype:

```
void* malloc(size_t size);
```

Let's read it as we learned in the last part: "malloc is a function that takes a `size_t` as an argument and returns a pointer to void".

Interesting. First off, it takes a type "size_t". We haven't seen that before. What is it? If we refer to `stddef.h`, we see that `size_t` is an "unsigned long". It is used for defining sizes of strings and memory blocks. It is ok to simplify it and think of it as a positive integer. You will see it often in C code. The `size_t` argument is the size of memory block you want to be allocated to your code by the operating system.

Next, it returns a pointer to a void. What that is really saying is that a pointer is returned, but it is not type specific. In x86 pointers are one word in size, so a pointer to an int is the same size as a pointer to a char.

Let's examine the usage:

```
char *text;  
text = malloc(1000);
```

You can see the idea from the code above, however there is a problem with this usage. The compiler will complain about the `malloc` returning a `void*` and you are assigning a `char*` to it. We can correct this using a technique call a "cast". We can tell the compiler to ignore the actual type of return and pretend it's something else. The proper usage is:

```
char *text;  
text = (char *)malloc(1000);
```

This usage tells the compiler to "cast" or pretend that `malloc` is returning a pointer to a char (`char*`). Our pointer, "text", now points at a brand new patch of memory, 1000 bytes in size. It is ours, given to us by the operating system at run-time. We can use it as we please. But there are some rules:

1. malloc does not initialize the memory for you. You have no idea what it contains when you receive your pointer.
2. When getting new chunks of memory from malloc, the proper form is `text=malloc..` not `*text=malloc`. When we use `*text`, we are referring to the values pointed at, not the address. malloc returns an address to your new memory. If malloc returns NULL, an error has occurred. Always check!
3. You must give your memory back when you are done. If you don't it is **possible** it will remain yours, even after your program exits. If you don't give it back before you exit, it may stay allocated "forever". This is a memory "leak". It reduces the amount of memory available in your systems because your OS still thinks it's in use, when in fact it isn't. (**Happy note - most OS's will take back any malloc'd memory when your program exits.)

Item 3 is important. How do we give our malloc'd memory back to the OS. Simple.

```
free(text);
```

That's it. Once we call 'free' our memory is de-allocated and given back to the free store. After calling free, it is very dangerous to use your pointer. While it still points to a valid memory location, the contents are no longer owned by you. Someone else may be using it!

We can malloc space for other types other than char as well. int, float, struct's, any type can be malloc'd. How do we know how much memory to malloc? An easy way to ensure adequate memory is allocated for our purposes is to use the sizeof function. Here is the prototype:

```
size_t sizeof(type);
```

sizeof is a function that takes a type and returns a size_t (unsigned long). sizeof returns the size, in bytes, of whatever object is in the parentheses. For example, if:

```
char text;  
int num;  
size_t x, y, z;  
x = sizeof(text);  
y = sizeof(num);  
z = sizeof(int);
```

In x86 Linux, $x = 1$, $y = 4$ and $z = 4$ in the code above. As you can see, it's acceptable to use built-in types in sizeof (sizeof(int)). Since we can't be sure

about our type sizes, we should always use `sizeof` in our `malloc` calls. Using this information, our previous example would read:

```
text = (char *)malloc(sizeof(char[1000]));
```

It looks like a lot but it isn't. We assign a pointer to `text` that points at a block of memory the size of 1000 char's.

For our struct from earlier, we could declare a pointer and give it some space:

```
struct s_data *student;
student = (struct s_data*) malloc(sizeof(struct s_data));

student->age = 12;
```

Here our `malloc` is "cast" to a pointer to "struct s_data". The amount of memory allocated is the size of our structure, `s_data`.

Conclusion:

We covered a lot of topics in this part. The idea of pointers, references and dynamic memory are key concepts in C. It would be wise to spend some time reading this part again, to ensure the concepts are grasped.

Next:

In the next part, we will start a programming project involving all of the concepts learned to date.