1 Lab 5

Date: Sep 19, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to familiarize you with the use of pointers in C. After completing this lab, you should be familiar with the following topics:

- Using pointers to traverse arrays.
- Pointer arithmetic.
- Casting between different pointer types.

1.2 Background

C allows the programmer to manipulate memory addresses. Given any C expression *E* which is stored in memory, then the address where *E* is stored can be extracted simply as `&`*E*. Note that the `&` unary operator must have an operand which has a memory address; it is an error if it is an expression like `x + 2` which does not have a memory address.

A *pointer* in C is nothing but a variable which holds a memory address. Hence the result of the `&` operator is often assigned to a pointer. A pointer `p` pointing to a value of type `T` is declared as `T *p;`.

For example:

```
int i;
int *ip = &i;          //ok
int *ip1 = &(i + 1);   //not ok: (i + 1) is a temporary and
                       //does not have a memory address.
```

C pointers are typed: i.e. each pointer points to a specific type. Hence dereferencing a pointer using the `*` prefix operator will result in an expression of that specific type.

```
int i = 5;
int *ip = &i;          //ok
int j = 3 * (*ip);     //ok: *ip is 5 and j will be set to 15.
char *p = *ip;         //not ok: attempt to assign an int
```

```
                        //to a char * pointer
```

C allows pointer arithmetic. It is possible to add/subtract a constant integer to a pointer, or subtract one pointer from another provided both pointers point to the same type. This arithmetic is scaled: i.e., the constant integer is scaled by the size of the underlying type.

```
int is[5] = {1, 2, 3, 4, 5};
char cs[5] = { 'a', 'b', 'c', 'd', 'e' };

int *ip = &is[0];       //equivalently, simply use is[];
                         //pointing ip to base of is[].
*(ip + 2) = 30;          //change is[2]; note that if an int
                         //is 4 bytes, then ip + 2 will add 8 to ip.
int *ip1 = &is[3];
int k = ip1 - ip;       //set k to 3
```

Though this lab hints at some of the possibilities of pointer manipulation in C which can result in buggy code, it is worth emphasizing that tricky pointer manipulation code is not usually necessary. If pointers are used in a stylized limited way, then it is easy to avoid pointer bugs.

1.3 Exercises

1.3.1 Starting up

Use the startup directions from the earlier labs to create a `lab5` directory and fire up a terminal whose output you are logging using the `script` command. Make sure that your `lab5` directory contains a copy of the files directory.

All of the following exercises have pointers traverse 2 arrays in different ways. The declarations for the arrays are as follows:

```
char cs[] = { 'a', 'b', 'c', 'd', 'e' };
int is[] = { 1, 2, 3, 4, 5 };
```

1.3.2 Exercise 1: Illustrating Pointer Increments

Change over to the ex1 directory and look at the pointers.c program. Once you have looked at the source code, build the `uints` executable by typing `make`. Run the program by typing `./pointers`.

The program uses the `cp` pointer to traverse the `cs[]` array and the `ip` pointer to traverse the `is[]` array and prints out the pointer value and what it points to after each step. Note that even though the code increments each pointer only by 1, the `cp` pointer increments by 1, but the `ip` pointer increments by 4 (the size of the pointed to `int` type).

If you run the program a second time, you may notice that the pointer values are different. This is due to added security in Linux: by randomly changing memory addresses slightly at each run, it is harder for crackers to exploit program vulnerabilites.

1.3.3 Exercise 2: Deriving Pointer Values

Change over to the ex2 directory and look at the in-pointers.c program. Once you have looked at the source code, build the executable by typing `make`. Run the program by typing `./in-pointers`.

The program requires you to type in pointers which point to specific elements in the `is[]` array. Provide the value in hex. If correct, you will get a `ok` message, if not correct, you will be asked to retry. The program will terminate when all cases have been answered correctly. (If you want to terminate the program early, use `^C`).

1.3.4 Exercise 3: Using Pointers with Incorrect Types

Change over to the ex3 directory and look at the bad-types.c file contained there. The code uses a `char *` pointer to traverse the `int[]` array and a `int *` pointer to traverse the `char[]` array.

Build the program by typing `make`, ignoring the warning message you get during the compilation. Run it by using `./bad-types`. You will notice that the program does print out memory, but since the pointers are pointing to the wrong object, the printed contents seem like garbage. However, if you look at the output more carefully, you will see that the `char *` pointer is printing out the bytes of the integers stored in `is[]` (in little-endian order) and the `int *` pointer is printing out the `char`'s in the `cs[]` array (note that the ASCII code for `a` is `0x61`), before taking off beyond it. Note that even though the program is accessing invalid memory using the `int *` pointer, the program continues, printing out the garbage contents of the memory.

This exercise illustrates why it is usually a bad idea to ignore compiler warnings.

1.3.5 Exercise 4: Casting Pointers

Change into the ex4 directory and examine the cast-types.c file contained there. It shows that even though we are using a `char *` pointer to traverse `is[]` and a `int *` pointer to traverse `cs[]` we can do so correctly if we treat them as the right type of pointer before we do pointer arithmetic on them. This is done using casts.

Specifically, `cp = (char *)(((int *)cp) + 1)`, casts `cp` to a an `int *` pointer, adds 1 to it (thus incrementing it by `sizeof(int)`) and then casts it back to a `char *` pointer so that it can be assigned back to `cp`. OTOH, `ip = (int *)(((char *)ip) +1)`, casts `ip` to a a `char *` pointer, adds 1 to it (thus incrementing it by `sizeof(char)`) and then casts it back to an `int*` pointer so that it can be assigned back to `ip`.

Type `make`. Then run the program `./cast-types`. Notice that the external behavior is quite reasonable.

1.3.6 Exercise 5: void pointers

Generic `void *` pointers are used only for storage and must be cast to a specific pointer type before being dereferenced or participitating in pointer arithmetic.

Change into the ex5 directory and examine the void-pointers.c file contained there. Type `make`. Then run the program `./void-pointers`. This shows that you can use a `void *` pointer to access both arrays correctly.

1.3.7 Exercise 6: Input void Pointers

Change into the ex6 directory and examine the in-voids.c file contained there. Type `make`. Then run the executable using `./in-voids`.

The program requires you to type in pointers which point to specific elements in the `is[]` and `cs[]` arrays. Provide the value in hex. If correct, you will get a `ok` message, if not correct, you will be asked to retry. The program will terminate when all cases have been answered correctly. (If you want to terminate the program early, use `^C`).

1.4 References

Jon Erickson, *Hacking: The Art of Exploitation*, 2nd Edition, No Starch Press, 2008. Source of most of the exercises.