L a b 4 : P o i n t e r s

Date: Sep 30, 2020

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

0.1 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.

0.2 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:

```c
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.

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

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

0.3 Exercises

0.3.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab4` branch and a new `submit/lab4` directory. You should have copied over the contents of `~/cs220/labs/lab4/exercises` over to your directory.

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

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

A `Makefile` is provided within the root of the exercises directory. When you are within an exercise directory, you can build the current exercise using:

```
$ make -f ../Makefile
```

Run through the following exercises in the `exercises` directory.

0.3.2 Exercise 1: Illustrating Pointer Increments

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

The program uses the `cp` pointer to traverse the `chars[]` array and the `ip` pointer to traverse the `ints[]` 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).

Run the program a second time and 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.

0.3.3 Exercise 2: Deriving Pointer Values

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

The program requires you to type in pointers which point to specific elements in the `ints[]` array. Provide the value in hex without any leading `0x` or `0X`. If correct, you will get an `ok` message, if not correct, you will be asked to retry. If you enter a value which points to an invalid address, you can crash the program. The program will terminate after you answer 3 successive attempts correctly. (If you want to terminate the program early, use `^C`).

0.3.4 Exercise 3: Using Pointers with Incorrect Types

Change over to the bad-types 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 -f ../Makefile`, 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 `ints[]` (in little-endian order) and the `int *` pointer is printing out the `char`'s in the `chars[]` 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.**

0.3.5 Exercise 4: Casting Pointers

Change into the cast-types 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 -f ../Makefile`. Then run the program `./cast-types`. Notice that the external behavior is quite reasonable.

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

0.3.7 Exercise 6: Input void Pointers

Change into the in-voids directory and examine the in-voids.c file contained there. Type `make -f ../Makefile`. 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`).

0.3.8 Exercise 7: Debugging Memory Allocation Errors Using valgrind

Change over to the directory bug-program where bug-program.c contains four memory allocation bugs. Look at the program which should be reasonably understandable.

For each word in an array of words, the program adds the word (referred to as `key`) and its index (referred to as `value`) into a linked list. Since the words are always added to the head of the linked list, it is not necessary to maintain a dummy node.

Build the program by typing `make`. Even with the multiple bugs, the program will probably run without a problem!! The fact that it may do so illustrate the insiduous nature of such bugs in that such buggy programs seem to work most of the time.

Can you understand why the indexes of the words are printed out in descending order?

Can you spot the bugs by simply inspecting it? Some hints:

- All bugs are within the `add_key_value()` and `free_key_values()` functions.
- The amount of memory needed to store a string **must** include space for the terminating `'\0' NUL` character.
- The strlen() function returns the number of characters in a string; it does **not** count the terminating `'\0' NUL` character.
- When `malloc()`'ing memory for some type `T`, the normal call will look like `T *pointerToT = malloc(sizeof(T));`.
- **All** allocated memory should be `free()`'d. Hence for every memory allocation call there should be a call to `free()`.
- Memory should not be accessed once it has been `free()`'d.

If you cannot spot the bugs, running the program under valgrind should help:

```
$ valgrind -v --leak-check=yes ./bug-program 2>bug-program.valgrind
```

This should record the standard error diagnostics in the `bug-program.valgrind` file. Now look at that file and the docs for valgrind and try to figure out the bugs. Look at the specific lines mentioned for `bug-program.c` to figure out the problems.

Since the report is quite long, search the file for errors (search for `Invalid`) mentioning `bug-program.c`. Look at the line specified by the line number mentioned in valgrind report. Consider the error reported by valgrind for that line and try to find the bug which may cause it. Note that the same bug may result in multiple errors.

Since it is very likely that each of the words in the Jabberwocky poem cause the same bugs, it may be a good idea to comment out all words other than the first and then run valgrind once again. This way the report will be shorter and it may be easier to find the bugs.

Once you have identified the bugs, fix them. Run a valgrind report so that valgrind reports a clean output without **any** errors. To avoid the `REDIR` messages, leave off the `-v` option.

0.4 References

Bryant and O'Halloran Recommended Text, section 9.9 up to and including 9.9.2 and section 9.11.

Valgrind.

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