ENPM691 Homework Assignment 1

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to determine the number of bytes assigned by the compiler to each variable type. The code collects the start/end memory addresses; the difference is the size of the datatype.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

In C, the compiler(e.g., GNU gcc) assigns memory during the program lifecycle. Depending on the type of declaration syntax (static vs dynamic), the memory can be assigned to the stack or heap. Static memory allocation happens in case of automatic variables during compile time. The term automatic means the variables are defined inside a block or a function. Also, C allows for defining storage classes (such as register keyword), and we get the ability to store on registers as well.

There is a concept of alignment, which basically means the memory is divided into even byte-sized words (each being 4 or 8) and padding if required in between. So when the compiler allotts memory to variables(for automatic) in the stack, the smallest logical unit is a byte. Each slot, being 8 bits in memory, has an address representing the indices.

The goal of the assignment is to find out the total bytes reserved for each variable type. Even though C gives an in-built function to check the size, the code will expose the hardware address for our experiment.

II. Methodology

A. The basic datatypes in C

There are seven basic datatypes, namely character, integer, short, long, float, double, and long double. User-defined datatypes like 'struct' and 'union' exist, but they are composed of the basic ones. Bool is another one, but C standards earlier than C99 do not support it and require a 'typedef' or bool library.

With the 'signed' keyword, the data storage can be extended to negative values. The benefit of 'unsigned' is that it allows a wider set of characters on the positive side. Either way, the total memory reserved is same. Array is a special set of structures that blocks continuous slots which again needs to be defined using the basic types.

So the experiment focuses on the seven fundamental ones.

B. Pointers and memory addresses

A pointer is a variable that can store the address of another variable using the unary operator '&'. By this declaration, the pointer points to the starting position of a memory block in the system's memory:

$$p = \&c; \tag{1}$$

Pointers can be combined with arithmetic operations, so the unary operator '++' can be used on them. This effectively tells it to increment to the next address and store its value:

$$*p++; \tag{2}$$

C. Code analysis and verfication

First, the program declares all the fundamental variables and their respective pointers in the 'main()' function. This being the entry point of the program, the compiler assigns memory accordingly. Then, the addresses of these variables are stored as values in the pointers.

Secondly, the initial address stored in the pointers is printed using the 'printf()' function. Here, '%d' instead of '%p' is used for easier calculation in integers. Next, the pointer is incremented and printed in the console to check the next slot in memory. Depending on the type of variables, the next address is some even bytes away from the initial address. The difference between the addresses is the amount of bytes saved for the datatype.

Third, we verify the difference found after manual calculation with the output of 'sizeof()' function. Every time, the result obtained is similar to the in-built function output. Hence, our experiment and theoretical assumption match.

III. Results

The following table captures the data derived from the console output when the program is executed. The difference is calculated manually.

TABLE I
Output of variable addresses and calculation

Sl. No.	Variable Type	Starting Address	Next Address	Difference
1	character	6422275	6422276	1
2	integer	6422268	6422272	4
3	short	6422266	6422268	2
4	long	6422260	6422264	4
5	float	6422256	6422260	4
6	double	6422248	6422256	8
7	long double	6422224	6422236	12

TABLE II
OUTPUT OF SIZEOF() FUNCTION

Sl. No.	Variable Type	sizeof(var)
1	character	1
2	integer	4
3	short	2
4	long	4
5	float	4
6	double	8
7	long double	12

```
[illegible]uments\dipam>temp_con.exe
address of variable ch: 6422275
next address after ch: 6422276
sizeof ch: 1
address of variable i: 6422268
next address after i: 6422272
sizeof i: 4
address of variable s: 6422266
next address after s: 6422268
sizeof s: 2
address of variable l: 6422260
next address after l: 6422264
sizeof l: 4
address of variable f: 6422256
next address after f: 6422260
sizeof f: 4
address of variable d: 6422248
next address after d: 6422256
sizeof d: 8
address of variable ld: 6422224
next address after ld: 6422236
sizeof ld: 12
```

Fig. 1. Actual console output

IV. DISCUSSIONS

A. Calculation of memory size

Table I states the difference between two memory addresses, calculated by:

$$Difference = NextAddress - StartingAddress \quad (3)$$

This gives us the number of bytes preserved for each datatype. Table II is for the confidence of our calculation by utilizing the 'sizeof' function. In the experiment, both these results go hand in hand. Hence, we got the capacity of the automatic variables.

B. Unexpected Results

The 'long' variable type generally is assigned at least 8 bytes for x64-based systems. From our results, we got only 4 bytes. Another is 'long double', which gets somewhere around 10 bytes, is here 12. Though this difference is present, it is following the C standards as expected.

C. Limitations and Considerations

The experiment is done on a Windows x64 device and a MinGW GCC compiler. The values found are not absolute and will differ for example in Linux machines. Apart from OS, x32-based processors may assign lower bytes to some of the types.

There are other compiler options, like GNU GCC, MSVC, or Clang each with its uniqueness in memory management. Adding to that, storage in each slot will be either in 'Little-Endian' or 'Big-Endian' format.

APPENDIX

C code

For this experiment, the following C code has been written to output the memory addresses of variables.

```
/*code that collects memory addresses of
variables through pointers*/
/*pointers are incremented to see if they
point to the next address*/
/*verify the allotted bytes between them*/

#include <stdio.h>

int main (void) {
 char ch,*p1; /*Defining the automatic variables*/
 int i,*p2;  /*and their pointers to memory*/
 short s, *p3;
 long l, *p4;
 float f, *p5;
 double d, *p6;
 long double ld, *p7;

 p1 = &ch;  /*assigning memory addresses*/
 p2 = &i;  /*pointers*/
 p3 = &s;
 p4 = &l;
 p5 = &f;
 p6 = &d;
 p7 = &ld;

 /*character datatype*/
 /*print start address*/
 printf("address of variable ch: %d\n",p1);

 /*incrementing pointer*/
 *p1++;

 /*to print next address*/
 printf("next address after ch: %d\n",p1);

 /*verify with sizeof func*/
 printf("sizeof ch: %d\n",sizeof(ch));

 /*integer datatype*/
 /*same as above, but for int*/
 printf("address of variable i: %d\n",p2);
 *p2++;
 printf("next address after i: %d\n",p2);
 printf("sizeof i: %d\n",sizeof(I));

 /*short datatype*/
 printf("address of variable s: %d\n",p3);
 *p3++;
 printf("next address after s: %d\n",p3);
 printf("sizeof s: %d\n",sizeof(s));

 /*long datatype*/
 printf("address of variable l: %d\n",p4);
 *p4++;
 printf("next address after l: %d\n",p4);
 printf("sizeof l: %d\n",sizeof(l));

 /*float datatype*/
 printf("address of variable f: %d\n",p5);
 *p5++;
 printf("next address after f: %d\n",p5);
 printf("sizeof f: %d\n",sizeof(f));

 /*double datatype*/
 printf("address of variable d: %d\n",p6);
 *p6++;
```

```
 printf("next address after d: %d\n",p6);
 printf("sizeof d: %d\n",sizeof(d));

 /*long double datatype*/
 printf("address of variable ld: %d\n",p7);
 *p7++;
 printf("next address after ld: %d\n",p7);
 printf("sizeof ld: %d\n",sizeof(ld));

return 0;
}
```