ENPM691 Homework 1:Measuring Memory Size of C Data Type

Kalpesh Bharat Parmar M.Eng Cybersecurity
University of Maryland, College Park
[redacted]
UMD Directory ID – [redacted]
Course and section - ENPM691 0101

***Abstract*—This report presents a method to measure memory sizes and alignment of common C data types (`char`, `short`, `int`, `long`, `long long`, `float`, `double`, `long double` and their unsigned versions) without using the `sizeof()` operator. A struct-based method is used to compute sizes by subtracting memory addresses. Results are verified against `sizeof()` and discussed.**

***Index Terms*—C programming, data types, memory alignment, sizeof, struct method**

I. Purpose

This assignment aims to investigate basic memory organization concepts in C programming, with a particular emphasis on variable size and alignment. To determine the number of bytes for common variable types like `int`, `long`, `float`, `double`, and others, a C program was created. The main measurement technique entails building structures and examining the address differences between members. The built-in `sizeof()` operator is used to validate the results. The methodology, findings and a detailed discussion of the results, including any discrepancies caused by the padding imposed by the compiler, are provided in this report.

II. Method

- Create a struct containing two variables of the same type.
- Subtract the addresses of the two variables to determine the size of the type:

  `(char*)&s.b - (char*)&s.a`
- Repeat for all signed and unsigned integer types, as well as floating-point types.
- Verify each measurement using `sizeof()`.

III. Results

Table I shows measured sizes of various data types using the struct method and `sizeof()` for verification. It also includes typical Linux 64-bit sizes for comparison.

TABLE I
Data Type Sizes (Struct Method vs sizeof()) on Windows 64-bit and Linux 64-bit

Type	Struct Method (Windows)	sizeof() (Windows)	Struct Method (Linux)	sizeof() (Linux)
char	1	1	1	1
unsigned char	1	1	1	1
short	2	2	2	2
unsigned short	2	2	2	2
int	4	4	4	4
unsigned int	4	4	4	4
long	4	4	8	8
unsigned long	4	4	8	8
long long	8	8	8	8
unsigned long long	8	8	8	8
float	4	4	4	4
double	8	8	8	8
long double	16	16	16	16

IV. Discussion

The outcomes of the struct-based approach closely resemble the sizes found using the built-in `sizeof()` operator, proving that calculating the memory size of variables in C can be done with accuracy by measuring the address differences between struct members. This demonstrates that the struct method can be used as a substitute for `sizeof()` in situations where it is prohibited or for teaching purposes to comprehend memory layout.

The findings allow for the following observations:

- **Integer types:** The same number of bytes are occupied by signed and unsigned integer types (`char`, `short`, `int`, `long`, and `long long`). The range of values that signed and unsigned types can store is the only distinction between them; memory size is not.
- **Floating-point types:** Because their representation by default contains a sign bit, the floating-point types (`float`, `double`, and `long double`) do not have unsigned variants. Their usual sizes are four bytes for `float`, eight bytes for `double`, and sixteen bytes for `long double` on the majority of 64-bit systems—were validated by the struct method.
- **Memory alignment and padding:** The struct-based approach sheds light on memory alignment as well. In order to guarantee effective access, compiler-imposed padding may be used to align data structures on memory boundaries. This is especially evident for larger data types, such as `long double`, where alignment may result in a greater member difference than is strictly required for the type size.
- **Platform dependence:** Depending on the compiler and operating system, some types' sizes, particularly those of `long` and `long double`, may change. For instance, `long` is typically 4 bytes on Windows 64-bit systems and 8 bytes on Linux 64-bit systems. This highlights how crucial it is to check sizes when writing portable C code.
- **Address computation and format specifiers:** The num-

ber of bytes between two members can be found by casting the addresses to `char*` and subtracting them when utilizing the struct method. Accurate printing and the avoidance of undefined behavior depend on using the proper format specifiers (`%ld` for the difference, `%zu` for `sizeof()`).

- **Comprehending memory organization:** This practice helps reinforce important ideas in C programming, such as the function of compiler padding, the effect of data type alignment, and how variables are stored in memory. It also emphasizes how basic operations, like pointer arithmetic, can be used to examine memory layouts without the need for built-in functions.

All things considered, this task gave practical experience in comprehending memory alignment, variable size, and platform-dependent behavior of C data types. It proved that combining `sizeof()` with struct-based address calculations provides a thorough method of verifying memory organization in C programs.

V. Appendix: C Program

```c
#include <stdio.h>

// creating struct for signed char, short, int, long and long long.
struct CharStruct { char a, b; };
struct ShortStruct { short a, b; };
struct IntStruct { int a, b; };
struct LongStruct { long a, b; };
struct LongLongStruct { long long a, b; };

// creating struct for unsigned char, short, int, long and long long.
struct UCharStruct { unsigned char a, b; };
struct UShortStruct { unsigned short a, b; };
struct UIntStruct { unsigned int a, b; };
struct ULongStruct { unsigned long a, b; };
struct ULongLongStruct { unsigned long long a, b; };

// creating struct for float, double, longdouble.
struct FloatStruct { float a, b; };
struct DoubleStruct { double a, b; };
struct LongDoubleStruct { long double a, b; };

int main() {


    struct CharStruct sc;             // creating a struct variable of CharStruct
    struct UCharStruct suc;           // creating a struct variable of UCharStruct
    struct ShortStruct ss;            // creating a struct variable of ShortStruct
    struct UShortStruct sus;          // creating a struct variable of UShortStruct
    struct IntStruct si;              // creating a struct variable of IntStruct
    struct UIntStruct sui;            // creating a struct variable of UIntStruct
    struct LongStruct sl;             // creating a struct variable of LongStruct
    struct ULongStruct sul;           // creating a struct variable of ULongStruct
    struct LongLongStruct sll;        // creating a struct variable of LongLongStruct
    struct ULongLongStruct sull;      // creating a struct variable of ULongLongStruct
    struct FloatStruct sf;            // creating a struct variable of FloatStruct
    struct DoubleStruct sd;           // creating a struct variable of DoubleStruct
    struct LongDoubleStruct sld;      // creating a struct variable of LongDoubleStruct

    /*Printing the difference between the address of two same datatype from the struct
    and seeing if the difference is same which we get from sizeof function and forming
    it in a tabuler form for easy viewing.*/

    printf("Type␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣Struct␣Method␣|␣sizeof()\n");
    printf("-----------------------------------------------\n");

    printf("char␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sc.b - (char*)&sc.a), sizeof(char));
    printf("unsigned␣char␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&suc.b - (char*)&suc.a), sizeof(unsigned char));

    printf("short␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&ss.b - (char*)&ss.a), sizeof(short));
    printf("unsigned␣short␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sus.b - (char*)&sus.a), sizeof(unsigned short));

    printf("int␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&si.b - (char*)&si.a), sizeof(int));
    printf("unsigned␣int␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sui.b - (char*)&sui.a), sizeof(unsigned int));

    printf("long␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sl.b - (char*)&sl.a), sizeof(long));
    printf("unsigned␣long␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sul.b - (char*)&sul.a), sizeof(unsigned long));

    printf("long␣long␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sll.b - (char*)&sll.a), sizeof(long long));
    printf("unsigned␣long␣long␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sull.b - (char*)&sull.a), sizeof(unsigned long long));

    printf("float␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sf.b - (char*)&sf.a), sizeof(float));
    printf("double␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sd.b - (char*)&sd.a), sizeof(double));
    printf("long␣double␣␣␣␣␣␣␣␣␣␣|␣%ld␣bytes␣␣␣␣␣|␣%zu␣bytes\n",
           (long)((char*)&sld.b - (char*)&sld.a), sizeof(long double));

    return 0;
}
```

Listing 1. C program to measure C data type sizes using structs

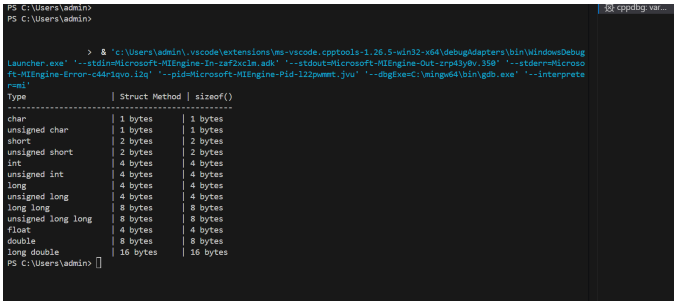

Fig. 1. Program output showing data type sizes using the struct method and sizeof() when the C program is compiled and executed on a Windows 64-bit operating system.

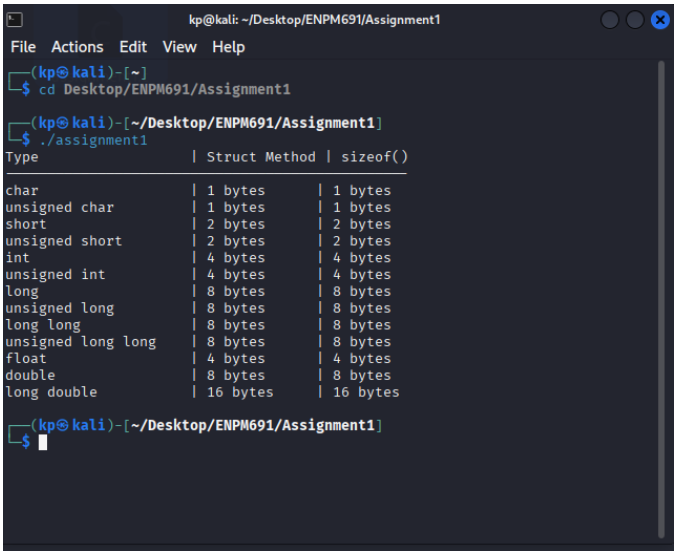

Fig. 2. Program output showing data type sizes using the struct method and sizeof() when the C program is compiled and executed on a Linux 64-bit system.