CMPE-380: Applied Programming

Laboratory Exercise 02

Strings, Data Types, Memory management & gdb

Table of Contents

Pre-Lab – 10 pts 4

Part1 – Strings & Data 4

Strings 4

Simple Printing 5

Primitive Data Types 10

User Defined Data Types 11

Composite Data Types 11

Structures 11

Packed Structures 12

Unions 14

Composite Data Type access 15

Part 2 – memory management 16

Malloc 16

Calloc 17

Realloc 17

Free 18

Part 3 - Debugging (gdb) 19

Commands 19

gdb Logging 20

gdb UI 20

Interactive Exercises – 30 pts 21

Using Strings & Data structures 21

Memory Management 23

Summary of deliverables 23

Assignment – 60 pts 24

Objective 24

Implementation Details 24

Grading Criteria 25

Notes 25

Laboratory Grading Sheet 26

Data Types, Strings

Pre-Lab – 10 pts

In this lab session, we will review basic and more complex data types in C. We will review strings and string functions simple print functions and then move to user defined and composite data types.

Your lab TA will conduct a short oral quiz during prelab review to verify you read and understood the material!

You must turn in the signed grading sheet at the end of lab to receive credit!

Part1 – Strings & Data

Strings

C does not offer a native string data type. All C strings are character arrays terminated with a binary zero (0x00). The means that all string buffers MUST include ONE EXTRA BYTE for the zero!

- Strings: (not a data type)
  - **Array of characters** terminated by the **sentinel 0x00**
  - Constant strings enclosed with doubt quotes:
    - The string : **"A"** **[65][\0]** The string A is **TWO BYTES LONG**
    - The character : **'A'** **[65]** The letter A, one byte long

Examples:

char[] course="CMPE-380"; /* declare and initialize a string buffer of 9 bytes */

if (course[2] == 'P') {} ; /* True, strings are just byte arrays with a sentinel */

Simple Printing

Printing data in C is similar to most other languages because most other languages borrowed the syntax from C. There are two popular printing functions: **fprintf()** and **printf()**.

printf() — prints to standard output "stdout"

fprintf() — to any file, often used to print to "stderr". fprintf to "stdout" is identical to printf

protptype: int printf(const char *format, ...)
int fprintf(FILE *handle, const char *format, ...)

usages: printf("format-string, variable-list);
printf(stderr,"format-string, variable-list);

Important: printf *returns* and int with the *number of characters printed*. In case of *error* it returns a *negative value. Print* functions can be called with "any" number of arguments. The number of arguments depends on the "format-string" specifications. Many C compilers does not care if the format-string matches the number of arguments given!

Simple format strings:

%c: char

%d: integer

%s: string (i.e., *pointer to char array*)

%f: float or double (*)

\n: new line

Example:

```
printf("my name is %s and my age is %d \n", "Richard", 26);          //prints to stdout

fprintf(stdout, "my name is %s and my age is %d \n", "Richard", 26);  //prints to stdout
fprintf(stderr,  "Debug: data is %f\n", 2.2); //  prints a floating pointer number to stderr
```

Many of the data formatting commands allow for fractional formatting. For example the %f format allows the programmer to control the number of digits displayed and the precision. Printf will ignore the formation if the total length is too short to hold the value.

%f<total length>**.**<fractional digits>

e.g. printf("Different data '**%f**' '**%5.3f**' '**%9.4f**'\n", 123.456789, 123.456789, 123.456789);

Different data '123.456789' '123.457' ' 123.4568'

Note the leading space of #3 and rounding of #2

See **exercise\lab_a.c** code for other format data types.

The standard library **<string.h>** has functions to operate on strings, such as:

strcpy, strcmp, strcat, strstr, strchr, strncpy, strncmp, strncat

You can't compare strings directly in C, you MUST USE A FUNCTION.

Consider:

```
char *str1 = "Some Text1";

char *str2 = "Some Text1";
```

if (str1 == str2) **NEVER true**, only comparing POINTERS str1 and str2 and they are at different memory locations**.**

Consider: if (*str1 == *str2) **ALWAYS true in this case**, only comparing the FIRST CHARACTERS in str1 and str2

You **MUST** use the function **strcmp()** to logically **compare two strings**

int strcmp(const char *s1, const char *s2);

- s1, s2 - *strings to be compared, byte by byte*
- Returns –
  - 0 - if strings are identical
  - Negative - if, character by character, s1 is less than s2
  - Positive - if, character by character. s1 is greater than s2

Use: **if (!strcmp(s1, s2)) { printf("Identical");}**

The most common string function is “string copy” or strcpy. It is a dangerous function because it doesn’t verify the destination has enough room and is a classic way to cause system memory crashes. **Never use strcpy**, it is included here because many people use it anyway.

char *strcpy(char *dest, const char *src);

Copies src into dest, byte by byte **until 0x00 in src is reached**

Assumes dest is big enough (dangerous)

Returns – Pointer to dest

How it works:

```
char dest [3];                  // Only 3 bytes long
strcpy(dest, "ABC");            // ABC is REALLY 4 bytes long!!
```

dest [0] contains “A”

dest [1] contains “B”

dest [2] contains “C”

dest [3] contains 0x00 // Memory corruption!

All modern programs use the “safe” version: **strncpy**

char *strncpy(char *dest, const char *src , int num);

Copies src into dest

Copy byte by byte until 0x00 in src is reached BUT will not copy more than num bytes (safer). This function does NOT null terminate in the error case.

Won’t overflow the destination buffer!

Returns – Pointer to dest

How it works: strncpy(dest, "ABC“, 3);

Other useful, safe functions:

char ***strncat**(char *dest, const char *src, int num);

Appends src to the END of dest

Copy byte by byte until 0x00 is src is reached.

Will not copy more than num bytes (safer) but will NOT null terminate in the error case.

Num should be adjusted for the size of the existing dest contents (strlen(dest))

Returns – Pointer to dest

How it works:

```
char dest [20];

strncpy(dest, “abc“, 20);

strncat(dest, “123“, 20 - strlen(dest));    // need to take into account the existing string
```

results in dest containing “abc123”

Other functions:

strstr – find a string in another string,

strchr – find a character in a string

Primitive Data Types

Table 1: Data types in C (on 32 bit gcc compiler)

DATA TYPE	BYTES	RANGE	printf SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	12		%Lf

The list of data types, their sizes in bytes, range and format specifier are shown in Table 1. Please note that the **size of each data type could be different on different platforms**. An example showing the differences of some data types in two platforms is provided in Table 2. You can use sizeof() function to learn the size of a given type in your platform.

Table 2 Data type sizes for two example platforms

VLSI machines (AMD Opteron(TM) Processor 6272, 1400 MHz)			Raspberry pi (ARMv6-compatible processor rev 7, 795.44 MHz)		
Type	Bits	Bytes	Type	Bits	Bytes
char	8	1	char	8	1
short int	16	2	short int	16	2
int	32	4	int	32	4
long int	64	8	long int	32	4
float	32	4	float	32	4
double	64	8	double	64	8
long double	128	16	long double	64	8

User Defined Data Types

C provides a mechanism to create your own types, e.g., "typemarks" (even if that typemark is the same as a primitive data type),

e.g., take look at /usr/include/stdlib.h. Examples from <stdint.h> (C99):

```
typedef signed char int8_t;   typedef short int   int16_t;   typedef int       int32_t;
```

typedef creates a typemark (i.e. alias); after including <stdint.h> we can use "int8_t, int16_t and int32_t" to declare variables of type "signed char, short int and int'' respectively.

Composite Data Types

The structure and union are complex data types which can store any data type. However, an important difference between them is that the structure has a separate memory location for each of its members whereas, the members of a union share the same single memory location.

Structures

A structure is used to build more complex data types from simpler data types. Structures can be used by themselves or "typedefed" using a typedef into their own unique data types.

The following structure illustrates a "point" in 2D space

```
struct point {     double x;
                   double y;
};
```

We can use the structure "point" in our code as: struct point MyPoint = {0,0};

Notice in our example, our structure was named "point", it does NOT allocate any variable space! We then use the named "structure point" to create the "MyPoint" variable which DOES allocate variable space.

We can also convert our named structure to a full data type by using the typedef command:

```
typedef struct point Point;

Point MyPoint = (0,0};
```

Structures can include other structures so we could create a 3D point using:

```
struct point3D {
        Point xy;               // Using our typedef!
        double z;
};
```

Structures can include arrays or pointers too:

```
/*structure with pointers*/
typedef struct MyData {
        char* name;
        char   data [256];
} myData;
```

Packed Structures

The actual number of bytes used by a structure may be greater than you would expect. The compiler will insert "pad" bytes to force data elements onto word or double word boundaries. The compiler does this padding for performance reasons. Normally these pad bytes are not an issue if you use the same compiler for all your code and don't transmit the data structure to a different computer/compiler. If you really need to save the structure space, or you need to transmit data to a different computer/compiler then use the, compiler dependent, "pack feature".

The gcc compiler pack feature syntax**: __attribute__ ((packed)**

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  // Normal data structure with standard automatic padding enabled
  struct padd {
    int  number;      // 4 bytes
     char buff [3];   // 3 bytes + 1 hidden pad
  } Padd;

  // Custom data structure with standard automatic padding DISABLED
  struct noPadd {
     int  number;     // 4 bytes
     char buff [3];   // 3 bytes, no hidden pad
  } __attribute__ ((packed))  noPadd;

  // Print out the sizes
  printf("Size of Padd:   %ld bytes, native users expect 7 bytes\n\n",
           sizeof(Padd));
  printf("Size of noPadd: %ld bytes, expert users expect 7 bytes\n\n",
           sizeof(noPadd));
 exit(0);
}
```

Resultant Code:

```
Size of Padd:      8 bytes, native users expect 7 bytes
Size of noPadd: 7 bytes, expert users expect 7 bytes
```

This test code demonstrates that the compiler will insert pad bytes as necessary to improve performance. This also suggests that data structures should be laid out with word and double word variables adjacent, while any byte variables should be grouped at the end to minimize padding.

Unions

A union allows the programmer to store different data types in the same memory location. You can define a union with many members. Unions can be used to "reuse" static memory (carefully) or to aid in data access and formation. Often peripheral chips register segments are defined as nibbles (4 bits) but the processor only can access the physical register using word (32 bit) access. Unions provide an efficient way of using the same memory location for multiple purposes.

Consider the simple union for accessing the high and low byes of a 16-bit integer

```
Union aunion  { uint16_t   number;
        struct  { uint8_t high;   uint8_t low; } bytes;
} aUnion;
```

The size of a union is the size of the largest member.

Composite Data Type access

Unions and structures, and the elements inside the are accessed the same way. Composite types that are defined are accessed using the "**.**" syntax while pointer types are accessed with the "->" syntax.

Consider a defined variable :

```
union  aunion { uint16_t   number;
          struct  { uint8_t high;   uint8_t low; } bytes;
};
```

```
union aunion aUnion;
aUnion.bytes.high= 0xee;                    // Notice the "." (two in this case)
aUnion.bytes.low = 0xff;
Results in aUnion.number containing:   0xffee        // Notice one "." in this case
```

Consider pointer variable:

```
union aunion *aUnion_P = &aUnion
```

```
Results in aUnion_P->number having   0xffee              // Notice the "->"
```

Part 2 – memory management

In this lab section, we will first review memory management and explain the use of malloc(), calloc(), realloc() , and free(). Then, we will review debugging in C via gdb.

Memory Management

Some programming languages (like Java, Python) support automatic memory management with garbage collection. However, in C, you must explicitly allocate/deallocate memory. There are a couple of key functions in the standard C library to do that.

Memory allocation is not necessary for a simple data structure (like int), because the needed memory is allocated as soon as the variable is declared (int i). However, memory allocation (deallocation) is very important when managing more complex data structures like arrays.

Malloc

malloc is the primary function used for memory allocation.

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

Examples

- Allocate memory to store names with a maximum size of 50 chars:
  - char * names = (char *) malloc(50 * sizeof (char))
- Allocate memory to store 50 floats:
  - float * floats = (float *) malloc(50 * sizeof (float))

Notes & Tips

- malloc returns NULL (pointer) if the memory allocation request is NOT successful!
- Always check if allocation is successful (names!=NULL or floats != NULL)
- Always typecast the generic void pointer to your specific type (char *, float * etc).

Calloc

calloc is used to allocate memory and set allocated memory to "binary zero".

void* calloc (size_t count, size_t unitSize);

Examples

- Allocate memory to store 50 integers:
    - int * ptr = (int*) calloc(50, sizeof(int));

Notes & Tips

- callloc returns NULL (pointer) if the memory allocation request is NOT successful!
- Always check if allocation is successful (NULL != ptr)
- Always typecast the generic void pointer to your specific type (int *).
- Note that the allocated memory is set to binary zero.
- The classical C data type sizes can vary in different platforms. To make your code portable you may use more specific types like int16_t, uint16_t, int32_t, uint32_t (instead of int).

Realloc

realloc is used to resize and copy memory (**grow or shrink** previously allocated memory).

void* realloc (void* oldPtr, size_t newSize);

Examples

- Allocate memory to store 50 integers:
    - int * ptr = (int*) calloc(50, sizeof(int));
- Now extend this memory to store 100 integers:
    - int * ptr = (int*) realloc(ptr, 100*sizeof(int));

Notes & Tips

- realloc returns NULL (pointer) if the memory grow/shrink request is NOT successful!
- Always check if allocation is successful (ptr!=NULL)
- Always typecast the generic void pointer to your specific type (int*).

Free

Free is used to return the allocated memory back to the heap.

void * free (void* pointer);

Examples

- Deallocate the above allocated memory location (by ptr)
  - free (ptr);
  - ptr=NULL;

Notes & Tips

- After you free the pointer names, floats, ptr become a dangling pointer. To avoid using it unintendedly always set unused pointers to NULL.

Part 3 - Debugging (gdb)

Commands

Because we are human and prone to make mistakes, we may use the divide and conquer approach (https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm) to break down complex problems into simple ones. It is difficult to see all possible implications of complex code in advance!

We use gdb (GNU Debugger) to trace programs written in C.

- http://www.gnu.org/software/gdb/
- http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html

Command	Comment
gdb code	Start the gdb debugging the program "code"
gdb -args code var1 var2	Start the gdb debugging the program "code" passing the parameters var1 and var2
l	List out the source lines of "code"
b lineNum	Set a break point at the line number in the current file
b function	Set a break point at the start of the function specified, in any file.
r var1 var2	Run "code" as if var1 and var2 were specified on the command line. Code will run until the break point is hit
c	Continue from the current line to the next break point
n	Run the next line of code
print var	Print out the value of "var". Only simple types (int, float, strings, ptrs)
q	Quit the debugger
where	Shows the code line that caused a crash

Figure 1 gdb commands

gdb Logging

The gdb debugger has an activity logging feature which will record all gdb activities. This feature is useful for tracking gdb activities and results.

To enable gdb logging execute the following before running your debug sessions.

1) gdb <binary>	Start gdb like normal, running <binary>
2) set logging file <file>	Set the output file name to <file> e.g. gdb.out
3) set logging overwrite on	Deletes any old gdb.out file
4) set logging on	Start writing results to the gdb.out file
5) <normal gdb commands>	The gdb command will not be recorded BUT the results will be.

Use this feature to record your future gdb sessions to provide as proof to the TA's. Be sure to choose meaningful file names.

gdb UI

In addition to the simple command line debugging features, gdb offers a simple GUI interface which displays the program code in context with the debugging commands. To access this feature , start dgb with the "-tui" option

Interactive Exercises – 30 pts

Compile lab_a.c (*gcc -O0 -g -Wall -std=c99 lab_a.c -o lab_a*) and run it (*./lab_a*) to see the sizes of primitive data types on your Linux system. Redirect the output to a file called **exercise.txt** and identify the data size that is incorrect, there is an important C lesson here. C does not guarantee data size.

Using Strings & Data structures

Use the following compile command in this section: **gcc -O0 -g -Wall -std=c99 lab_b.c -o lab_b**

Edit the "lab_b.c" frame work to add the following features. Create a **exercise.txt** to hold the answers to any questions in this section.

1) Create a **union** of a float and integer (don't use a struct, use a union)
   Call the data union "anumber1" and the variable "aNumber1"
   Assign the integer the value 5 and then the float the value of 6.0
   Print out the value of the float AND integer, be clear in your print statement what you are printing.
   Are the values what you expected? Why is one of the numbers "garbage"

2) Create a structure of a float and integer
   Call the data structure "anumber2" and the variable "aNumber2"
   Assign the integer the value 5 and then the float the value of 6.0
   Print out the value of the float AND integer, be clear in your print statement what you are printing.
   Are the values what you expected?

3) Use the sizeof() operator on the data structures (e.g. sizeof(union anumber1)) to print out the size of the union and structure variables.

Be clear in your print statement what you are printing.

Which one is larger, why?

4) You are going to print out a list of Beatle band members and birth years using a typedef and a for loop.

   Typedef the given "bandMember" structure as "band"

   Create an array called "beatles" from the "band" typedef

   Initialize the fields in the array with the supplied information: John Lennon 1940, Paul McCartney 1942, George Harrison 1943, Ringo Starr 1940

   Use strncpy to copy strings

   Write a for loop to print the data out in the form: "Beatle John Lennon born 1940"

5) Create a run on string of Beatle members first names: e.g "Run on string of first names:JohnPaulGeorgeRingo:

   Using a for loop, strncat and the beatles array data from the previous question.

6) Write code to return the index number for the "Ringo. You will need to use the **strcmp()** function

   Clearly identify the index number: e.g "The Ringo index is 3" (your number can vary)

Memory Management

Use the following compile command in this section: **gcc -O0 -g -Wall -std=c99 lab_c.c -o lab_c**

Edit the "lab_c.c" frame work to add the following features. Add the answers to any questions to your **exercise.txt** file.

1) Malloc 12 bytes of memory for "text_p" with proper return code checking
2) Using the SAFE strncpy() function, copy the string" "CMPE380 lab malloc try 1.". You must be able to justify your choice of "n".
3) Print out the resulting string with single quotes around the %s string variable (e.g: "'%s'") so the exact string is clearly identified. Did the "string" print properly?
4) Use realloc() to resize "text_p" to 20 bytes with proper error checking.
5) Use the SAFE strncat() to add "-- Realloc" to "text_p". You must be able to justify your choice of "n". (hint: should include strlen())
6) Print out the resulting string with single quotes around the %s string variable (e.g: "'%s'") so the exact string is clearly identified. Did the "string" print properly?
7) Free the memory block.

Summary of deliverables

exercise.txt

lab_b.c, lab_c.c

Assignment – 60 pts

Objective

Implement a data driven **dynamic array** "abstract data type" as a C module and to get familiar with the use of memory allocation and deallocation functions in the standard library **stdlib**. Every call to malloc(), calloc(), realloc(), fopen(), fread() or equivalent status returning function MUST include error handling.

Implementation Details

Upload the file **lab02.tar** (available in MyCourses) to the **lab02** working directory. The tarball contains:

The header files **DynamicArrays.h** and **ClassErrors.h**

A C framework **DynamicArrays.c**

A **build** and **test** script, to build and test your code

A test harness **simpleTest.c** to test your code

1. Write the implementation of your dynamic array module (DynamicArrays.c) using the interface specification in the file DynamicArrays.h, where the elements of the array are variables of type Data containing a character–string field. Use the pseudo-code given in the code or Lecture Notes for reference.

2. Use the provided **simpleTest.c** to help test your code.
    1. Build your code: ***./build***
    2. You can manually debug your code using: **gdb ./simpleTest**
    3. You can automatically test your code: ***./test***
    4. Memory leak detection is NOT required for this assignment

    Note: you may have to run the following command ONCE:
    chmod +x build
    chmod +x test
    to make the two scripts executable.

Grading Criteria

1. (54 points) Correct implementation of Dynamic Array. (error codes, error messages, checking for all important return codes, etc.)
2. (6 points) Correct results reported.

Notes

1. Start work on this homework early, especially if you do not have experience using pointers or programming in C.
2. It should be clear that the implementation must be your own (not copied from some remote source in the Internet)

Student Name: _____

Laboratory Grading Sheet

Lab02 - Strings, Data Types, Memory management & Debugging (gdb)

Component	**Point Value**	**Points Earned**	**Comments and Signatures**
Pre-Lab: strings oral quiz (%d, %s, strncat, length, etc)	3		
Pre-Lab: Data types (unsigned, float, etc)	1		
Pre-Lab: Memory Management Oral quiz (malloc, etc.)	3		
Pre-Lab: Debugging oral quiz (break points, where, etc)	3		
Interactive Exercises: Strings & Data structures	15		
Interactive Exercises: Memory management	15		
Total	40		

You must turn this signed sheet in at the end of lab to receive credit!