Lab 7: Dynamic Memory & Valgrind

CSE/IT 113L

NMT Department of Computer Science and Engineering

"A program is never less than 90% complete, and never more than 95% complete. "

— Terry Baker

"Programming without an overall architecture or design in mind is like exploring a cave with only a flashlight: You don't know where you've been, you don't know where you're going, and you don't know quite where you are.."

— Danny Thorpe

"Manually managing blocks of memory in C is like juggling bars of soap in a prison shower: It's all fun and games until you forget about one of them."

— anonymous Usenet user

Contents

1 Introduction

In this lab you will learn how to dynamically allocate memory for your variables with `malloc()`, and how to use Valgrind in checking for memory leaks.

2 Overview

In order to complete this lab, read the material in this section and answer the questions and problems presented in Section 3.

2.1 Reading

Chapters 11, 12, and 13 in *C Programming: A Modern Approach*

2.2 malloc() and free()

Looking at the code for pstr_ncat() in str.c you soon realize that this function doesn't work very well as the sizes of the arrays have to be determined at compile time. What you would like is the ability to dynamically allocate memory as needed at run time. To do this you need to use malloc(). malloc() dynamically allocates memory for you, placing it on the *heap*. See man malloc for details. The prototype for malloc() is found in stdlib.h.

For dynamic memory allocation, the first thing you have to do is determine how many elements you want to allocate. The second thing is to determine the size of the type of object. The call to malloc() then allocates $num\_elem \times sizeof(object)$ bytes of contiguous memory.

2.3 A better version of pstr_cat()

For concatenation, the array you allocate has to be large enough to hold the original string sans `'\0'`, plus the string you want to append to it sans `'\0'`, plus the NULL terminator.

Lets say you what to append the string `char *t = ", go c, go.";` to the string
`char s[] = "c run, c run unix, run unix run";`

Since you can't change the size of `s[]` at runtime, you first need to copy `s` into a new memory location. To do this you will use malloc() to create a space big enough to hold a copy of `s`, the string you want to append to it (`t`), plus space for `'\0'`. So the call to malloc() looks like:

```c
char *s = "c run, c run unix, run unix run";
char *t = ", go c, go!";
char *u;

 /* 1 for the NULL char */
size_t len = str_len(s) + str_len(t) + 1;

/* how many objects times the size of the object */
u = malloc(len * sizeof(char));

/* or as one call */
/* char *u = malloc((str_len(s) + str_len(t) + 1) * sizeof(char));
```

A successful call to malloc() returns a pointer to the base address of the memory just allocated.

Once you have called malloc(), it is your responsibility to make sure the memory was actually allocated. As the call to malloc() may fail (e.g. you are out of memory) for reasons beyond your control, you need to handle the error. A simple check is to check to make sure malloc() did not return NULL, which indicates malloc() failed.

```c
/* malloc may fail so need to check to make sure it worked  */
if (u == NULL) {
        printf("malloc failed. goodbye...\n");
        exit(MALLOC_FAILED); /* #define MALLOC_FAILED 1000 */
}
```

exit () is defined in stdlib .h as well. When exit () is called the program exits with the status given as the function parameter.

Now that you have setup enough memory to hold the new string, you can 1) copy the original string into the new memory location, and 2) append the string you want to it. The following code uses functions you just wrote:

```c
/* the + 1 makes room for the NULL terminator */
pstr_ncpy(u, s, str_len(s) + 1);
printf("u = \"%s\"\n", u);

pstr_cat(u, t);
printf("u = \"%s\"\n", u);
```

The only remaining thing to do, now is to quit the program. However, before you exit there is one task left to do. Since you allocated memory, you are responsible for *freeing* the memory so it can be reused by the OS again. If you allocate memory without freeing it, you will get what is called a *memory leak* as the OS cannot reclaim the unused memory until the program exits.

Putting this all together, the code looks like this:

```c
/* dynamic.c */
#include <stdio.h>
#include <stdlib.h>
#include "str.h"

#define MALLOC_FAILED 1000

int main(void)
{
        char *s = "c run, c run unix, run unix run";
        char *t = ", go c, go!";
        char *u;

        /* 1 for the NULL char */
        size_t len = str_len(s) + str_len(t) + 1;

        /* how many objects times the size of the object */
        u = malloc(len * sizeof(char));

        /* malloc may fail so need to check
           to make sure it worked  */
        if (u == NULL) {
                printf("malloc failed. goodbye...\n");
                exit(MALLOC_FAILED);
        }

        /* the + 1 makes room for the NULL terminator */
        pstr_ncpy(u, s, str_len(s) + 1);
        printf("u = \"%s\"\n", u);

        pstr_cat(u, t);
        printf("u = \"%s\"\n", u);

        /* you allocated it you free it */
        free(u);

        return 0;
}
```

The above code is available in the file dynamic.c.

2.4 strings.h

As a C programmer, you will commonly manipulate strings in your code. Rather than rolling your own like you just did for instructional purposes, C provides a standard library to manipulate strings (string .h) and in production code that is what you should use. To become familiar with the string

library and its capabilities, you should read the man page man −s3 string and the individual man pages for details.

If you read the BUGS section of the man page for strcpy it states: "Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine.". To help prevent buffer overflows you will notice the man page mentions a strncpy function which only copies n characters. As you read more of the man pages on string functions, you will typically see two versions of the functions: one without a n in its name and one with a n in its name. The one with the n in its name is the one you should use to help prevent buffer overflow exploits. Why the two versions? C originally just had `strcpy()`, but as users realized that `strcpy` is exploitable, the n-versions were developed. However there is still a lot of (legacy) code that uses strcpy() so you can't simply get rid of the function call in the standard library.

2.5 Valgrind

Valgrind is a tool used, among other things, to check for memory leaks in the heap. If it is not already installed you can install it on Ubuntu with sudo apt−get install valgrind

Read Valgrind's quick-start guide `http://valgrind.org/docs/manual/quick-start.html` to figure out how to get started using Valgrind. Using the command from section 3 of the quick-start guide, "Running your program under Memcheck", check that your version of dynamic.c does not have a memory leak.Valgrind's `HEAP SUMMARY` contains the relevant information:

```
==18442== HEAP SUMMARY:
==18442==     in use at exit: 0 bytes in 0 blocks
==18442==   total heap usage: 1 allocs, 1 frees, 43 bytes allocated
==18442==
==18442== All heap blocks were freed -- no leaks are possible
```

Comment out the call to free, recompile and run. This time you should get an error as you made one allocation (malloc) without freeing it.

```
==18875== HEAP SUMMARY:
==18875==     in use at exit: 43 bytes in 1 blocks
==18875==   total heap usage: 1 allocs, 0 frees, 43 bytes allocated
==18875==
==18875== LEAK SUMMARY:
==18875==    definitely lost: 43 bytes in 1 blocks
==18875==    indirectly lost: 0 bytes in 0 blocks
==18875==      possibly lost: 0 bytes in 0 blocks
==18875==    still reachable: 0 bytes in 0 blocks
==18875==         suppressed: 0 bytes in 0 blocks
```

Every time you are dynamically allocating memory, it is a good idea to run your program through Valgrind to make sure you are freeing memory correctly.

Uncomment the call to free in dynamic.c.

2.6 Dynamic Arrays

Rather than determining the size of arrays at compile-time you can determine them at runtime using malloc(). For instance, the following would allow you to declare an array of `doubles` of arbitrary size at run-time, enter data, and print the data out:

```c
/* dbl.c */
#include <stdio.h>
#include <stdlib.h>

#define MALLOC_FAILED -99
#define LEN 100

int main(void)
{
        int size;
        double *d;
        char buf[LEN];
        int i = 0;

        printf("Enter a size for your array: ");

        fgets(buf, LEN, stdin);
        sscanf(buf, "%d", &size);

        d = malloc(size * sizeof(double));

        if (d == NULL) {
                printf("malloc failed. goodbye...\n");
                exit(MALLOC_FAILED);
        }

        /* initialize elements to zero */
        for(i = 0; i < size; i++)
                *(d + i) = 0.0;

        /* reset i */
        i = 0;

        while (i < size) {
                printf("Enter a value: ");

                /* ctrl+d pressed or some other error,
                   fgets returns NULL */
                if (fgets(buf, LEN, stdin) == NULL) {
                        printf("\n");
                        break;
                }
```

```
43                  sscanf(buf, "%lf", &d[i++]);
44          }
45
46
47          for (i = 0; i < size; ++i)
48                  printf("%lf\n", *(d + i));
49
50          return 0;
51
52  }
```

Compile and run dbl.c. What happens when you enter ctrl + d?

3 Lab Specifications

Exercise 1 (str.c, str.h, test.c, free.script, mem-leak.script).

1. str.c has a number of array based versions of the basic string functions which you have to convert to pointer based ones. You will also write several from scratch. Read the man pages for the behavior of the function (e.g. man strlen). Duplicate the behavior of the standard function in the functions you write.

   `test.c` is a driver for the various functions you will write in str.c.

   Using redirection, capture the output of the various different functions in `s.out`. To append to an already existing file with redirection use >>

   The file `str.h` is provided for you.

   (a) Write a pointer version of `strlen()`, named str_len(char *s), which returns the length of the string s. Do not count the NULL terminator. Read the man page for strlen.

   (b) Write a pointer version of `strncpy()`, named str_ncpy(char *dest, char *src, int n), which copies n characters from src and copies them into dest. Read the man page for strncpy for details. Use malloc to allocate enough memory to store `n` characters from `src` in `dest`. Don't forget to `free()` your malloced memory when appropriate.

   (c) Write a pointer version of `strcat()`, named str_cat(char *s, char *t), which concatenates `t` to the end of s; s must be big enough to hold `t`. Use `realloc` to make sure that `s` is large enough. An array version is given str_cat Update this version to use pointers. Dont't forget to `free()` your reallocated string when appropriate.

   (d) Write a pointer version of `strncmp()`, named str_ncmp(char *s, char *t, int n) which compares the first n characters. Read the man page for the behavior of this function (man strncmp).

(e) Write a pointer version of index(), named pindex(char *s, int c) which finds the first occurrence of c in s, returning a pointer to its location, and NULL otherwise. Read the man page for the behavior of this function (man index).

(f) Write a character swap function, named cswap(char *c, char *d)

(g) Write a pointer version of reverse(), named reverse(char *s) which will reverse the elements in the array. Make sure you call cswap. An array version of (reverse()) is given.

2. Create a makefile to compile and run with Valgrind/leaks `test.c` with the functions you wrote for string processing.

3. You will now capture the output of make into two scripts - one with your free() lines commented out and one with your free() line working correctly. Name the first script `mem-leak.script` showing your code running with the frees commented out (memory leak). Name the second script `free.script` showing your code running with no memory leaks. Use the `make` command to compile and run your programs with Valgrind/leaks.

Exercise 2 (dynamic_array.c, da.out).

1. Write a program that reads in the following 11 integers dynamically:

$$4, 6, 2, 4, 9, 11, 14, 16, 1, 15, 3$$

Use `realloc()` to create your array dynamically. Don't forget to free your array before exiting your program. Name this file `dyn_array.c`.

2. Add functions to find the min, max, average and median of the list. These functions should each have one `pointer` as a parameter - a pointer to the structure that stores the array, along with the array's min, max, average. These functions should all be void functions.

3. Write a function to print the array, the min, the max, and the average. This print function should take in only one parameter as well - a pointer to the structure storing the array, and its min, max, and average.

4. Compile and run your code using Valgrind/leaks to show that there are no memory leaks. Using redirection, capture the output in a file named `da.out`.

4 README

Exercise 3 (README).
Every lab you turn in will include a README file. The name of the file is README with no file extension.The README will always include the following two sections *Purpose* and *Conclusion*:

- **Purpose:** describe what the program does (i.e. what problem it solves). Keep this brief.
- **Conclusion:**
  - What did you learn. What new aspect of programming did you learn in this lab? This is the portion of the lab where you want to be analytical about what you learned.
  - Did the Pair Programming prelab help you in solving this solo lab?
  - Did you encounter any problems? How did you fix those problems?
  - Did your code still contain bugs in your final submission? If so, list those bugs.
  - What improvements might you make? In other words, if you were to refactor your lab code, what would you update?

The conclusion does not have to be lengthy, but it should be thorough.

Some labs will require a special **Pseudo-code** section. If a lab requires a pseudocode section, it will be explicitly mentioned in that lab.

5 Getting Help

If you need help with this Lab, please go to tutoring offered weekdays in Cramer 213 and weekends on discord. Please also ask questions in class to the TA's or instructor. These concepts are important for future labs and exams, so if you don't understand them now it is important to get help!

Submitting

You should submit your code as a tarball file that contains all the exercise files for this lab. The submitted file should be named (**note the lowercase**)

`cse113_firstname_lastname_lab7.tar.gz`

If you require a refresher on generating a tar archive file, instructions can be found in Section 6 - Creating Tar archives of Prelab 0.

Upload your `.tar.gz` file to Canvas.

List of Files to Submit

Exercises start on page 6.