Lab 8: Strings, Pointers, and Dynamic Memory

CSE/IT 113

Pointers and Addresses

A pointer is a variable that holds a memory address. Pointers can point to any data type, including user-defined ones (e.g. structures).

The following are valid declarations of pointers:

```
int main(void)
{
        int *x;
        double *d;

        return 0;
}
```

`x` is pointer to an integer type; `d` is a pointer to a double type.

Just because you have declared a pointer doesn't it mean it points to an initial value. It is a good practice to initialize pointers. There is a special value `NULL` which you use to initialize a pointer. The above really should be written as:

```
int main(void)
{
        int *x = NULL;
        double *d = NULL;

        return 0;
}
```

Note: initializing pointers to `NULL` is something that is checked during grading.

Even though you initialized the pointers, they still do not point to any addresses. The next step would be to point the pointers to valid addresses. To obtain the address of a variable you use the address operator `&`. To dereference pointers–that is, to get the value of what the pointer points to and this is usually what you want–you use the `*` operator. For example, to manipulate integers via an integer pointer you use a combination of address and dereference operators. Compile and run the following program `basic.c`:

```c
/* basic.c */
#include <stdio.h>

int main(void)
{
        int *x = NULL;
        int y = 0;
        int z = 10;

        printf("x's address is %p\tx's value is NULL\n", x);
        printf("y's address is %p\ty's value is %d\n", &y, y);
        printf("z's address is %p\tz's value is %d\n\n", &z, z);

        x = &y;     /* x points to y */
        printf("x's address is %p\tx's value is %d\n", x, *x);
        printf("y's address is %p\ty's value is %d\n", &y, y);
        printf("z's address is %p\tz's value is %d\n\n", &z, z);

        *x = 99;    /* change the value of what x points to (y) */
        printf("x's address is %p\tx's value is %d\n", x, *x);
        printf("y's address is %p\ty's value is %d\n", &y, y);
        printf("z's address is %p\tz's value is %d\n\n", &z, z);

        x = &z;     /* x now points to z */
        printf("x's address is %p\tx's value is %d\n", x, *x);
        printf("y's address is %p\ty's value is %d\n", &y, y);
        printf("z's address is %p\tz's value is %d\n\n", &z, z);

        *x = -101;   /* change the value of what x points to (z) */
        printf("x's address is %p\tx's value is %d\n", x, *x);
        printf("y's address is %p\ty's value is %d\n", &y, y);
        printf("z's address is %p\tz's value is %d\n\n", &z, z);

        return 0;
}
```

Note the use of and difference between `x`, `*x`, `&y`, `y`, `&z`, and `z` and how you can change the values of `y` and `z` via a pointer.

1. *Write a program that changes the value of* `double p` *from 3.14159 to 2.71828 using a pointer. Print out both the address and the value of* `p` *and the pointer. Name your source code file* `fundamental.c`*. Capture the output to a file named* `fundamental.out` *using redirection. At the prompt run* `$ fundamental > fundamental.out`

Arrays, Pointers and Incrementing

Pointers and arrays are very closely tied in C. Pointer versions of array manipulations are generally faster, that is why is important to manipulate arrays via pointers.

Arrays and Pointers

The declaration `int a[10]` defines an array of size 10, a block of 10 consecutive integers in memory named `a[0]`,`a[1]`, `a[2]`, `...`, `a[9]`. And the notation `a[i]` refers to the i-th element of the array. If you declare a pointer to an integer `int *ip` and make the assignment `ip = a`, `ip` points to the first or `a[0]` element of the array and `ip` contains the address of `a[0]`. It is important to understand that `a` is really shorthand for `&a[0]`. That is, `a` refers to the first or base address of the the array.

To dereference the pointer you use the dereference operation, `*ip` which is equivalent to `a[0]`. To get to the next element in the array you can add 1 to `ip` and dereference. `*(ip + 1)` is the same as `a[1]`, and `*(ip + 2)` is the same as `a[2]`, and so on.

Pointer Arithmetic

To refer to the i-th element of the array `a[i]` with pointers you can write `*(a + i)`. (Note: you can use `*(ip + 1)` as well). The parenthesis around `(a + i)` are important. `(a + i)` is the address of `a[i]`, (i.e. `&a[i]`), and `*(a + i)` is the contents `a[i]`. For instance, if you just did `*a + 1` you would add `1` to `a[0]`.

Adding one to a pointer `(a + 1)`, means you point to the next element in the array. It does not not mean you literally add one to an address which would move you only one byte. This is true for any pointer to an array of any type: adding one will take you to the next element in the array. This is called pointer arithmetic.

Pointer arithmetic adds the number of bytes that each element occupies to the base address. For example, if you have `double d[10]`, then `d + 5`, would add $5 * 8 = 40$ bytes to the base address, which is `d[5]`'s location in memory.

Walking Through Arrays with Pointers

You can use either pointer arithmetic or pointers to walk though array elements.

If you want to walk through an array using pointer arithmetic you can do this:

```
int a[] = {1,2,3,4};
int i = 0;
size_t len = sizeof(a)/sizeof(int);

/* pointer arithmetic a + i =  base address + i * sizeof(a) */
for ( ; i < len; i++)
        printf("%d\n", *(a + i));
```

And if you want to walk through the array using pointers you can do this:

```
int a[] = {1,2,3,4};
int i = 0;
int *ip = a; /* point to the beginning of the array */
size_t len = sizeof(a)/sizeof(int);

/* ip++ moves to the next element in the array */
for ( ; i < len; i++, ip++)
        printf("%d\n", *ip);
```

Incrementing Pointers

Adding one is a common operation in CS, and the increment (decrement) operator (either ++ or −−) can be used with pointers. But there are some subtleties regarding precedence and how operators associate.

For example, given the declaration `int a[10]` and `int *ip = a` (you can do the declaration and assignment at the same time like any other variable). The following syntax works to increment the address:

`++ip` or `ip++`.

As with other post and prefix operations, there are side effects you have to be aware of. Compile and run `side.c`

```
/* side.c */
#include <stdio.h>

int main(void)
{
        int a[] = {1,2,3,4,5,6,7,8,9,10};
        int *p = a;
        int *q = NULL;

        q = p++;
        printf("a's address is %p\ta's value is %d\n", a, *a);
        printf("p's address is %p\tp's value is %d\n", p, *p);
        printf("q's address is %p\tq's value is %d\n\n", q, *q);

        q = ++p;
        printf("a's address is %p\ta's value is %d\n", a, *a);
        printf("p's address is %p\tp's value is %d\n", p, *p);
        printf("q's address is %p\tq's value is %d\n", q, *q);

        return 0;

}
```

You can also increment what the pointer points to. Both `++*p` and `(*p)++` increment what `*p` points to. The parenthesis are necessary around `(*p)++` otherwise you would increment `p`, the address. This happens because `++` associates right to left, but `*` has higher precedence than `++`.

Also note `*p++` is not the same as `(*p)++`. `*p++` the dereference is made first and then you increment the pointer (the address). `(*p)++`, you dereference and then increment what it points to (the value).

2. *The source code* `array.c` *gives you array based versions of printing an array (*`print_array()`*), incrementing the values of an array (*`inc_array()`*), and adding two arrays together (*`add_array()`*).*

   Your job is to modify `array.c`*:*

   (a) *write a pointer arithmetic version of* `print_array`
   (b) *write a pointer based version of* `inc_array`
   (c) *write a pointer based version of* `add_array`
   (d) *capture the output into a file using redirection. Name the output file* `array.out`

   Note: as function parameters `int a[]` and `int *a` are equivalent. They both refer to the base address of the array. In `array.c` change all `int a[]` to `int *a`.

Strings, char s[] and char *s

Unlike integers and other types where you have to pass the length of the array to functions to correctly process the array, strings use the `NULL` terminator to indicate the end of a string. This makes processing strings different than other types of arrays.

Declaring Strings

When you are declaring two strings, there is a difference between:

```
s[] = "c run, run c run";
```

and

```
char *s = "c run, run c run";
```

`char s[]` declares an array named `s`, allocates space for the string, and appends `'\0'` to the string. `s` refers to the beginning address of the string and you can *change* the elements of the string later in your code.

`char *s` allocates space for the string, null terminates it, points `s` to the first character in the string, but `s` is a *string constant*. You cannot change it later in code. You can move the pointer so it points to a different address, but if you do that you have lost the location of the string constant unless you keep track of it.

Compile and run `string_error.c`

```
/* string_error.c */
#include <stdio.h>

int main(void)
{
         char s[] = "c run, run c run";
         /* char *s = "c run, run c run"; */

        char *p = s;  /* p points to the first element of s */

        printf("the original string:\n");
                printf("%s\n", s);

        while(*p != '\0') {      /* dereference what p points to */
                if (*p == 'r')
                        *p = 'f'; /* change the value */
                p++;            /* increment the address */
        }

        printf("\nthe changed string:\n");
                printf("%s\n", s);

        return 0;
}
```

As written, `string_error` works fine. You can change the values of `s`. Now uncomment the `char *s` and comment the `char s[]` line. Recompile and run your code. You will get a *segmentation fault* as you cannot change the string constant. A segmentation fault (or seg fault for short) occurs when you try to access memory you cannot access.

As a review of `gdb`, start `$ gdb ./string_error` and run the program. The program will exit with a seg fault. Use the backtrace command (`bt`) to figure out what line the seg fault occurs at.

3. *Uncomment the line char *s and comment the char s[] line. Capture the output of `string_error` in a script name `string_error.script`. Also capture the gdb output including the results of the backtrace command into the script.*

Pointers as Function Parameters

Passing pointers to functions allows you to change variables without relying on the return type. This means that you can now change multiple items when you call a function and the changes will remain on the function's return. The classic example is a swap function, which occurs often in sorting algorithms. The following program swaps the integers inside the function but does not keep them swapped on the return.

```c
/* swap.c */
#include <stdio.h>

void print_vars(int a, int b)
{
        printf("a = %d\tb = %d\n", a, b);
}

/* a first try at writing a swap function */
/* swap the contents a with b and b with a */
void swap(int a, int b)
{
        int tmp = a;
        a = b;
        b = tmp;

        printf("in function swap: ");
        print_vars(a,b);
}

int main(void)
{
        int a = 6;
        int b = 9;
        printf("before swap: ");
        print_vars(a,b);

        swap(a, b);

        printf("after swap: ");
        print_vars(a,b);

        return 0;
}
```

Compile and run `swap.c`.

The swap doesn't work because functions pass arguments by value in C. This means that the called function is given copies of of its arguments. Remember how the call stack and automatic variables work.

When `swap(a, b)` is called, the `swap` gets a copy of `a` and `b`. So it is not changing the original variables in `main`. Instead, it is swapping copies of `a` and `b`.

Calling Functions that use Pointers

To allow you to change the value of variables that you pass in as function arguments you have to use pointers. Through pointers the function will get the address of the memory location of the variable in the calling function. The function can now change the value of the variable and changes will remain after the function returns.

To call `swap` correctly, you need to change the parameters of the function to pointers and call the function with the address operator: `swap(&a, &b)`.

Remember pointers are variables that hold a memory address. Use of the address operator gives the address of the variable, which is precisely what you need

4. *Rewrite swap.c so that it correctly swaps values. Using redirection, capture the output in a named swap.out*

Strings Redux

C provides a number of string functions in the standard library strings.h. Use `man -s3 string` to see a list of string functions. In production code you would use the functions in `string.h` to manipulate strings. But writing our own versions of these functions is instructive.

5. *`str.c` has a number of array based versions of the basic string functions which you have to convert to pointer based ones. You also have to 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` calls the various functions using conditional compilation. Uncomment the appropriate #define statements to compile `test.c`. Make sure you comment the functions you write following the doxygen coding style. Using redirection, capture the output of the various different functions in `str.out`. To append to an already existing file with redirection use >>. `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 `man strlen`.*

   (b) *Write a pointer version of `strncpy()`, named `pstr_ncpy(char *dest, char *src, int n)`, which copies `n` characters from `src` and copies them into `dest`. Read the man page `man strncpy` for details.*

   (c) *Write a pointer version of `strcat()`, named `pstr_cat(char *s, char *t)`, which concatenates `t` to the end of `s`; `s` must be big enough to hold `t`. An array version is given `str_cat`*

   (d) *Write a pointer version of `strncmp()`, named `pstr_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 pointer version of* ***`squeeze()`***, *named* ***`psqueeze(char *s, char c)`***, *which removes* `c` *from the string* `s`*. An array version of (*`squeeze()`*) is given.*

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

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

malloc() and free()

Looking at the code for `pstr_ncat()` 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.

A better version of pstr_cat()

For concatenation the array you allocate has to be 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:

```
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.

```
/* 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:

```
/* 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:

```
/* 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`.

6. *Compile and run* ***`dynamic.c`*** *with the functions you wrote for string processing. Using redirection, capture the output to a script named* ***`dynamic.out`***

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.

7. *Your job is to rewrite* ***`dynamic.c`*** *using standard C library string calls. Name your source code file* ***`dynamic_strings.c`****. Read the man pages for the details of strlen, strncpy, strncat. Make sure you use the n-versions of the standard library calls. Using redirection capture the output to a script named* ***`dynamic_strings.out`***

Valgrind

Valgrind is a tool used, among other things, to check for memory leaks in the heap. 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. Running your program under Memcheck” of the quick start guide, check that you version of `dynamic.c` does not have a memory leak. If you just changed the string calls you will 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`.

8. *Capture the output of Valgrind into a script with free and free commented in* ***`dynamic.c`****. Name your scripts* ***`free.script`*** *and* ***`no_free.script`***

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:

```
/* 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;
                }
                sscanf(buf, "%lf", &d[i++]);
        }


        for (i = 0; i < size; ++i)
                printf("%lf\n", *(d + i));

        return 0;

}
```

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

9. *Write a program that reads in the following 11 integers dynamically: 4, 6, 2, 4, 9, 11, 14, 16, 1, 15, 3. Find the min, max, average, median of the list. You already did this in Lab 3, but now rewrite those functions using pointers. Print the array, the min, the max, the average, and median to stdout. Name your source code file* ***`dyn_array.c`***. *Using redirection, capture the output in a file named* ***`dyn_array.out`***

What you need to turn in

A summary of what you need to turn in:

1. `fundamental.c` and `fundamental.out`
2. `array.c` and `array.out`
3. `string_error.c` and `string_error.script`
4. `swap.c` and `swap.out`
5. `str.c`, `str.h`, and `str.out`
6. `dynamic.c` and `dynamic.out`
7. `dynamic_strings.c` and `dynamic_strings.out`

8. `free.script` and `no_free.script`

9. `dyn_array.c` and `dyn_array.out`

Submit your *.c, *.h, *.out, *.script, Makefile, and README files as a tarball to Moodle before the due date.

The name of your tarball follows that of previous labs:
`cse113_firstname_lastname_lab8.tar.gz`