Prelab 7: Strings and Pointers

CSE/IT 113L

NMT Department of Computer Science and Engineering

"Programs must be written for people to read, and only incidentally for machines to execute. "

— Abelson and Sussman

"Low-level programming is good for the programmer's soul."

— John Carmack

"Programming can be fun, so can cryptography; however they should not be combined."

— Kreitzberg and Shneiderman

Contents

1 Introduction

In this prelab, you will explore how to use pointers, pointer arithmetic, and strings. You will also learn how to pass pointers as function parameters.

2 Overview

In order to complete this lab, read through the material and examples provided and answer the questions presented in Section 3

2.1 Reading

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

Chapters 1 in *Understanding and Using C Pointers*

Chapter 6 in *The Linux Command Line*

2.2 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:

```
1 int main(void)
2 {
3         int *x;
4         double *d;
5
6         return 0;
7 }
```

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

Just because you have declared a pointer doesn't 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:

```c
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 – you use the **dereference 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);
```

```
33
34          return 0;
35  }
```

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.

2.3 Arrays, Pointers and Incrementing

Pointers and arrays are very closely tied in C. Pointer versions of array manipulations are generally faster, that is why it 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 element of the array (or a[0]). In other words, 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 operator, (*ip) which is equivalent to a[0]. To get to the next element in the array you can add 1 to ip alongside the dereference operator. *(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: Because ip also points to a[0], 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) will access a[i]. For instance, if you just did *a + 1 you would add 1 to whatever was stored at 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**.

For example, let's assume you have the following array (double d[10]). If you wanted to access the sixth element (d[5]) through pointer arithmetic, you would use (d + 5), which would add $5 * 8 = 40$ bytes to the base address, thereby accessing 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(int)) */
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 */
/* note the use of the postfix operator, when does ip get updated? */
for ( ; i < len; i++)
        printf("%d\n", *ip++);
```

Incrementing Pointers

Adding one is a common operation in CS, and the increment (or 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++.

NB: you cannot use increment/decrement operators with a because it is pointing to an array. In C, arrays are **constant pointers**, which means you, as the programmer, are not allowed to alter what a is pointing to.

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

```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 the value that *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 ++.

2.4 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.

2.5 Declaring Strings

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

char 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 individual elements of the

string later in your code.

char *s allocates space for the string, NULL terminates it and points s to the first character in the string, but with this s you *cannot* change the individual elements of the string later in your code.

In other words, char s[] is mutable, while *s is immutable.

Compile and run `string_error.c`

```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.

2.6 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 this swap function passes 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.

2.7 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, because you are accessing the physical location in memory where the original variable is located.

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 the called function is expecting as a parameter.

2.8 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.

3 Lab Specifications

Exercise 1 (fundamental.c, array.c, string_error.c, swap.c, makefile, pl7.script).

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`.
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) modify print_array to use pointer arithmetic.

   (b) modify inc_array to use pointer arithmetic.

   (c) modify add_array so that elements of all arrays are accessed via secondary incremented pointers.

   (d) change all `int a[]`'s to int `*a`'s in function declarations.

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

3. In string_error.c uncomment the char *s line and comment the char s[] line. Compile and run the program. Use gdb and use backtrace to determine what line causes the segmentation fault to occur. Add a comment at that line in the program explaining why it caused the segmentation fault.
4. Rewrite swap.c so that it correctly swaps values.
5. Create a makefile to compile **and execute** fundamental.c, array.c, and swap.c.
6. Create a script file called pl7.script that shows you running the make command to compile and execute your programs.

4 Getting Help

If you need help with this Lab, please go to tutoring offered weekdays in Cramer 213 and via discord on the weekends. 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_groupNUMBER_prelab7.tar.gz`

Replace the `NUMBER` with your assigned group number and have one group member upload the tarball to Canvas before the due date.

Only one person in your group needs to upload your group's tarball to Canvas. If, upon being uploaded, all members of your group do not see that the file has been submitted, please contact the instructor or head TA as soon as possible.

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

List of Files to Submit

Exercises start on page 8.