Prelab 3: Loops and Arrays

CSE/IT 113

"Thats whats cool about working with computers. They dont argue, they remember everything, and they dont drink all your beer."

— Paul Leary

"Embrace the SUCK!"

— Anonymous

Unix Command of the Week

Read the man page on `wc`. Find out how to use it and what it does. In particular, learn what the flags `-l, -c, -w` do. You will be asked to demonstrate them for a TA and they will be on your quiz!

1 Introduction

In order to accomplish this lab, you will need to master two new concepts: loops and arrays. There are three different types of loops we will apply in this lab. You should learn to apply them to various problems.

2 Requirements

In order to complete Lab 3, you must complete the following:

1. Read in *C Programming: A Modern Approach* Chapter 6, Sections 8.1,

2. Review Chapters 1 - 5 in *The Linux Command Line*.

3. Answer the questions in the Questions Section of this prelab, create a tarball of your files named `cse113_firstname_lastname_prelab3.tar.gz` and submit it to Canvas before the due date.

3 Material

In order to complete this prelab, read the following material and answer the questions in Section 9.

3.1 Loops

Loops are handy stuff: they let you do something in your program as many times as you like without copying code.

There are 3 different types of loops:

1. for
2. while
3. do...while

3.1.1 For Loops

Let's say you want to print "hello, world!" ten times. You can either do this:

```
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
printf("hello, world!\n");
```

or write a for loop:

```c
int i = 0;

for (i = 0; i < 10 ; i++) {
        printf("hello, world!\n");
}
```

As you can see, the for loop is much easier to code.

Note the increment operator `i++` in the for loop. The increment operator is just shorthand for `i = i + 1`. And since this happens so often in programming, the inventors of C created a convenient shorthand syntax for the operation.

Lets break down what happens in the for loop:

1. `i` is assigned the value of 0. This is the start value of the for loop.
2. The condition (`i < 10`) is evaluated. The condition contains the end value of the for loop, in this case 10. What happens next depends on whether the condition is true or false.
3. If the condition is true then the following happens:
   (a) The body of the for loop executes.
   (b) the value of `i` is updated (`i++`).
   (c) repeat step 2 with the new value of `i`.
4. If the condition false, then the for loop terminates and execution picks up after the }

In general, a **`for`** loop has this basic structure:

```c
for (initialization ; condition ; update) {
        /* for loop body */
}
```

Its important to understand that (for the most part) the initialization, condition, and update all work on the same variable.

As another example of a for loop, say you want to print the squares of numbers between 1 and 100 that are divisible by 3:

```
int i = 0;

for (i = 1; i <= 100; i++) {
        if (i % 3 == 0)
                printf("%d^2 = %d\n", i, i * i);
}
```

or you could write it as:

```
int i = 0;

for (i = 3; i <= 100; i += 3) {
        printf("%d^2 = %d\n", i, i * i);
}
```

Note the syntax `i += 3`. This is shorthand for the statement `i = i + 3`. You do not need to always update for loops by one!

Problems that can be solved with **`for`** loops are fairly common. Remember the basics of how it is done.

Typically, for loops are used for a finite and defined number of executions. This is referred to as"definite iteration."

3.1.2 While Loops

`while` loops are syntactically simpler than for loops. They look like this:

```
while (condition) {
        /* while body */
}
```

In C, for and while loops are equivalent. However, **`while`** loops are typically used for "indefinite iteration." This means that the number of executions is not known until the loop is executed. It could also be possible that there are no executions at all.

REMEMBER: If your condition is false the first time, your program will skip right over the while loop without entering the body at all.

Consider this code:

```
int i = 0;

printf("Enter an integer: ");
scanf("%d", &i);

while (i != 0) {
        printf("Enter an integer: ");
        scanf("%d", &i);
}
```

This will keep reading numbers until `i` is false. When is `i` false? When `i == 0`.

3.1.3 Do...While Loops

A **do...while** loop is similar to a **while** loop. It looks like this:

```
do {
        /* do block */
} while (condition);
```

Do-loops always run the body of the loop once and then the test (while) is performed. If the test is false, the do-loop stops executing; if the test is true the body of the do-loop executes again.

The above while loop example could be rewritten as:

```
int i = 0;

do {
        printf("Enter an integer: ");
        scanf("%d", &i);
} while (i != 0) ;
```

You use a **do...while** loop when you always want to do the loop at least once.

4 Lists

It is often the case in computer science that you will process a *list* of items. A list in C is a *homogeneous* collection of items (e.g. integers, characters, strings, or floating point numbers, etc) that allows for duplicates. Lists differ from sets in that sets do not allow duplicates.

For instance $A = \{0, 3, 4, 5, 6, 3, 4, 5\}$ is a list ($3, 4$ and 5 are repeated), while $B = \{0, 3, 4, 5, 6\}$ is a set. However, a set is a list.

Mathematically, you can think of a list as a sequence defined as

$a_1, a_2, a_3, \cdots, a_n$

where n is some integer and a_i is an integer, or a real number, or a string, etc (whatever type your list consists of). a_i references the i-th element in the list. The $_i$ is known as a subscript.

For example, assume $n = 1000$ and a_i are the natural numbers. The sequence you would generate is:

$1, 2, 3, 4, 5, \cdots, 998, 999, 1000$

How would you represent this list in C? You could define a 1000 different variables. For example,

```c
int a1 = 1, a2 = 2, a3 = 3, a4 = 4, a5 = 6;
/* keep declaring variables a6 through a1000 */
int a998 = 998, a999 = 999, a1000 = 1000;
```

You would very quickly grow tired of typing variable names. And what happens if you change n to a million. Nobody wants to manually create a million variables, let alone a thousand variables.

As you can see, one difficulty with lists is the naming of variables. There are various ways in which you represent lists in C. The simplest is through an array.

5 Arrays

An array is a contiguous block of memory that occupies $n \cdot size\_of\_data\_type$ bytes. For example, with $n = 1000$ and a data type of `int`. The array would occupy $1000 * 4 = 4000$ bytes or 4K in memory as each `int` occupies 4 bytes in memory.

If $n = 1000000$ and the data type was a `double` (a double is a floating point number that occupies 8 bytes), the occupied space in memory would be $1000000 \cdot 8 = 8000000$ bytes or 8 MB of memory.

You access array elements (variables) in C through a notation similar to subscripting in mathematics. Rather than subscripts which you can't represent using the ASCII character set, C uses the notation `a[i]` to indicate the i-th element in the array. i is called the index of the array.

However, there is important distinction between mathematical and programming notation: The first element of the array is always `a[0]` not `a[1]` That is, **array indexing begins with zero in C**.

So for the sequence $1, 2, 3, 4, 5, \cdots 998, 999, 1000$, the first element in C would be represented by `a[0] = 1` and the last element by `a[999] = 1000`.

To declare arrays in C, you do this:

`type name[length]`

where `type` is the data type of the array, `name` is the name you use to reference the array, and the `length` is the number of elements in the array, which is always one more than the last index of the array.

To create an array to hold a 1000 integers, your declaration would like this:

```
  int a[1000];
```

or another way, which makes your code more maintainable, is to use a preprocessor constant

```
#define SIZE 1000

int a[SIZE];
```

Like other variables in C, declaring them does not initialize them.

There are a couple of ways to initialize an array.

If you would like to initialize every element to the list to zero, you could do something like the following:

```
#define SIZE 1000

int a[SIZE] = {0};
```

Another way you could initialize an array is with a for loop. The for loop visits each element of the array, setting it to zero:

```
#define SIZE 1000
int a[SIZE];
int i = 0;

for(i = 0; i < SIZE; i++)
       a[i] = 0;
```

If you wanted to set the array to the sequence 1, 2, ..., 999, 1000, you would do this:

```c
#define SIZE 1000
int a[SIZE];
int i;

for(i = 0; i < SIZE; i++)
        a[i] = i + 1;
```

Note how the for loop increments the index of the array. This allows the for loop to visit every element of the array and make an assignment. Or put another way, we say the for loop *iterates* over the array.

Using the index in the **for**() statement and in the body of the for loop is a very, very common way to process array elements.

If you want to make individual assignments, you write code like this:

```c
#define SIZE 1000
int a[SIZE] = {0}; /*set all elements of array to zero*/

a[0] = 100; /* first element is set to 100 */
a[1] = 50; /* second element is set to 50 */
a[998] = 50; /* second-to-last element is set to 50 */
a[999] = 100; /* last element is set to 100 */
```

The array would like this: $100, 50, 0, 0, \cdots 0, 0, 50, 100$ in memory.

5.1 Example

In case you're having trouble putting this all together, here's a program fragment. It will read in 10 integers and then print them out in reverse order.

```c
int someInts[10]; /* set up an array of ints of size 10 */
int i = 0;

for (i = 0; i < 10; i++) {
        printf("Enter a value for someInts[%d]: ", i);
        scanf("%d", &someInts[i]);
}

for (i = 9; i >= 0; i--) {
        printf("%d\n", someInts[i]);
}
```

The statement `i--` is equivalent to `i = i - 1`. The `--` operator decrements a variable by one; the `++` operator increments a variable by one.

6 Determining the length of the array

Often is the case that you need to figure out how many elements are in an array. Or put another way, you need to figure out the *length* or *size* of the array.

Another way to declare and initialize arrays is to explicitly define the array elements:

```c
int a[] = {7, 5, 6, 7, 3, 4, 8, 9, 10, 11, 14, 3, 2};
```

Notice that there is not a number between the brackets. You do not need to put the size of the array if you declare and initialize on the same line. What is the length of the array? The compiler figures out the size for you. In this example, the compiler allocates space for 13 `int`s and makes the assignments `a[0] = 7, a[1] = 5, a[2] = 6, ...., a[11] = 3, a[12] = 2` for you. This is convenient as you let the compiler figure out the size for you, but functions often need to know the length of the array to work correctly. The compiler provides no magic in this case.

To determine the size of an array you use the **`sizeof`** operator. The **`sizeof`** operator tells you how many bytes the array occupies. But the number of bytes is not what you want as that is how much memory is occupied by the array. Remember the number of bytes an array occupies is determined by $n \cdot size\_of\_data\_type$, where n is the number of elements in the array. To find the length or size or the number of elements of the array, you have to divide the size of the array by the $size\_of\_data\_type$.

For example, run this program and see what it outputs:

```c
#include <stdio.h>

int main(void)
{
        /* array declaration */
        /* allocates a contiguous block of memory */
        int i = 0;
        size_t size; /* sizeof returns size_t types */
        int data[] = {43, 5, 67, 44, 33, 25, 54, 33, 17, 6, 9, 12, 52};

        /* the token to display the size is %zu */
        printf("the array occupies %zu bytes\n", sizeof(data));

        /* determine the size of the array */
        size = sizeof(data)/sizeof(int);
```

```

        printf("the array has %zu elements\n", size);

        /* print elements of array */
        /* i++ equivalent to i = i + 1 and i += 1  */
        for(i = 0; i < size; i++) {
                printf("d[%d] = %d\n", i, data[i]);
        }

        return 0;
}
```

Notice that the variable `size` has the type `size_t`. This is the type that **`sizeof`** returns. Also, that the conversion specifiers are using `%zu` to print the variable size.

You will use the **`sizeof`** operator frequently in this class. Particularly, when you do dynamic memory allocation.

7 Functions and Arrays

To use an array as a parameter to a function, you make a function declaration like this:

```
int foo(int a[])
{
        /* function body */
}
```

Notice that the brackets are empty in function declarations.

To call this function you do something like this:

```
int foo(int a[])
{
        /* function body */
}

int main(void)
{
        int data[] = {1, 2, 3};
        foo(data);
        return 0;
}
```

You do not use braces when you are passing an array parameter to a function. Also, you cannot use arrays as a return type of functions. You cannot write code like this:

```c
/*THIS IS WRONG - WILL NOT COMPILE */
int[] foo(int a[])
{
        /* function body */
}
```

But passing the array as a parameter is not enough to use arrays properly in functions. Run the following program:

```c
#include <stdio.h>
int foo(int a[])
{
        printf("\nin function foo\n");
        printf("a occupies %zu bytes\n", sizeof(a));
        printf("a has %zu elements\n", sizeof(a)/sizeof(a[0]));
        printf("a[0] occupies %zu bytes\n", sizeof(a[0]));
        return 0;
}

int main(void)
{
        /* array declaration */
        /* allocates a contiguous block of memory */

        int data[] = {43, 5, 67, 44, 33, 25, 54, 33, 17, 6, 9, 12, 52};

        printf("data occupies %zu bytes\n", sizeof(data));
        printf("data has %zu elements\n", sizeof(data)/sizeof(int));
        printf("data[0] occupies %zu bytes\n", sizeof(data[0]));

        foo(data);

        return 0;
}
```

You should get the output:

```
/* this assumes a 64-bit machine */
data occupies 52 bytes
data has 13 elements
data[0] occupies 4 bytes

in function foo
the array occupies 8 bytes
```

```
the array has 2 elements
a[0] occupies 4 bytes
```

Probably not what you expected as the array `data` and the array `a` should be the same size! Why does the function report a size of 2 when you know the array has 13 elements? And without knowing the length of the array you can't use the bread and butter of array processing — for loops.

To save memory, the address of the first element of the array is passed to the formal parameter. Rather than making a copy of the array and then making the assignment to the function parameter, the compiler is passing the *address* of the first element of the array. It does this for efficiency and to save memory. Imagine if you had an array that occupied 100 MB. If every time you called a function with an array size of 100 MB and it had to make a copy of the array it would be slow and you would waste a lot of memory. To avoid these penalties, the compiler passes the address of the array rather than a copy of the array. Use of the address speeds things up and allows the function to access an already allocated region of memory. Addresses are 8 bytes on a 64-bit machine (4 bytes on a 32-bit machine). That is why the size of the array `a` in function `foo()` is only 8 bytes. It is the size of the array's first element's address.

To fix this problem, the length of the array needs to be a function parameter as well. The correct way to pass an array to a function is as follows:

```c
int foo(int a[], size_t size)
{
        int i = 0;

        for(i = 0; i < size; i++) {
                /* todo -- process array, element by element a[i] */
        }

        return 0;
}
```

You have to know the array length before you call the function, so the correct way to call the function `foo()` is:

```c
#include <stdio.h>

int foo(int a[], size_t size)
{
        int i = 0;

        for(i = 0; i < size; i++) {
               /* todo */
        }

```

```
        return 0;
}

int main(void)
{
        /* array declaration */
        /* allocates a contiguous block of memory */
        size_t size;
        int data[] = {43, 5, 67, 44, 33, 25, 54, 33, 17, 6, 9,
            12, 52};

        size = sizeof(data)/sizeof(int);

        foo(data, size);

        return 0;
}
```

8 Quiz

For your quiz this week, please be able to use basic UNIX commands to navigate through a terminal: know what your current directory is (`pwd`), list the contents of a directory (`ls`), move between directories (`cd`), make a directory (`mkdir`), copy files (`cp`), remove files (`rm`), and word count (`wc`). Also know what flags we want set when you compile (see below). You will have a one-on-one test with one of the TAs so please feel comfortable in front of a terminal.

9 Questions

Answer the following questions. When you compile, make sure you use the following command:

```
gcc -g -Wall filename.c -o filename
```

where filename.c is your source code and filename is the name of the file sans the .c extension. For example, if your source code was named `init.c` the correct way to compile it is with the command:

```
gcc -g -Wall init.c -o init
```

The `-g` option turns on debug symbols–we will get to debugging programs in a few labs. The `-Wall` turns on all warnings. Your code should compile so no warnings are produced.

1. Write a program that prints the numbers from 1 to a 1000 to the screen. One number per line. Use a for loop. Name your file `for_1000.c`.

2. Write a program that prints the numbers from 1000 to 1 to the screen. Use a while loop. Print the numbers with tabs between them. In C, the escaped character \t prints a tab. Name your file `while_1000.c`

3. Write a program that stores the numbers from 1 to a 1000 in an array. Initialize the array with a for loop and print the array to the screen using a for loop. Print one number per line. Name your file `init.c`

4. Modify the program `init.c` so that printing is now done via a function named

   `print_array(int a[], int len).`

   `int a[]` is the array you want to print and `len` is the size of or the length of the array (the total number of elements the array contains). Use **`sizeof`** to determine the size of the array and pass that to the print function. Name your file `init_print.c`. The function `print_array` will now let you print any integer array of any size! Print the values of the array, one per line.

5. Write a function named

   `multiply_range(int start, int stop)`

   that accepts two integers called `start` and `stop` as input and returns the product of every number between the start and stop,including start and stop. For example, if you enter 2 as your start and 5 as your stop, the function will return the value $2*3*4*5$, or 120. Hint: Use a for loop, and declare a variable named `product` to hold the running product. Name your file `multiply.c`. Your program should also work if the value of `start` ≥ `stop`.

6. Modify `multiply_range()` so it now accepts an array. Name the new function
   `array_multiply_range(int a[], size_t size, int start, int stop)`
   Check to make sure that `start` and `stop` are between 0 and `size`. If `start` or `stop` are less than zero, set it equal to zero and if `start` or `stop` is greater than the size of the array set it equal to size. The function should work even if `start` ≥ `stop`. This time, assuming `start` ≤ `stop`, the product returned is the product of $a[start] \times a[start+1] \cdots \times a[stop]$. Name your file `multiply_array.c`

7. Write another function called
   `array_mod(int a[], int div[], int size, int divisor)`
   that takes in two arrays, the size of the arrays, and an additional number (named `divisor`) to see if any of the numbers in array `a` are divisible by `divisor`. It is assumed that the size of the arrays is the same. If the number in array `a` is divisible by `divisor`, place a 1 in the corresponding element of array `div`. If it is not divisible

place a 0 in the corresponding element of array `div`. Print out both arrays with a tab between corresponding elements and one pair of elements per line. Modify the print function you wrote earlier.

Your output will look something like this:

```
10	1
4	0
13	0
5	1
9	0
35	1
10	1
22	0
134	0
135	1
```

Name your file `divisor.c`.

10 Testing the code

Create a script file named `prelab3.script`. Test your code as follows:

Just run the programs `for_1000`, `while_1000`, `init`, and `init_print`.

For `multiply_range()` use the ranges `(3, 7)` and `(19, 18)`.

For `array_multiply_range()` use the following values

```
int a[] = {6, 3, 5, 2, 3, 2, 4};
int start = 1;
int stop = 4;
```

For `array_mod` use the values

```
int a[] = {13, 44, 85, 23, 72, 99, 100, 108, 222, 1084}
int divisor = 4;
```

11 Submission

Tar your source code files and script file into a file named

`cse113_firstname_lastname_prelab3.tar.gz`.

and post to Canvas before the due date.