CS 2001 Programming Fundamental	Lab 09a Pointers

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2023.

Contents

National University of Computer and Emerging Sciences 1
Aims and Objectives 3
Introduction 3
C Variables 3
C Pointers 4
The Dereferencing Operator '*' 5
Pointers and Arrays 6
Passing by Value and Passing by Reference 8
TASKS TO DO 10

AIMS AND OBJECTIVES

This lab manual aims to provide an introduction to C pointers and make you comfortable in using them and reading code that uses pointers. After the lab you will be able to: -

1. Identify pointer variables.
2. Understand basics of pointer arithmetic in C.
3. Understand the importance of datatype for a pointer variable and its impact on pointer arithmetic and use.
4. Use pointers to pass variables as reference in functions.
5. Use pointers and arrays interchangeable.

INTRODUCTION

One of the several things that make C language so powerful is its ability to provide direct access to memory addresses to programmers via pointers. However, 'With great powers, come great responsibility.' So, with access to memory addresses programmers themselves are the managers of memory addresses, cleaning, managing, and other memory related tasks. C lang does not have Garbage Collectors as found in other languages like C# or Java.

We start off by revisiting what a variable is and then extending that definition to pointer variables.

C VARIABLES

We take the definition of a variable from, *"A Tutorial On Pointers And Arrays In C By Ted Jensen"*

> *A variable in a program is something with a name, the value of which can vary. The way the compiler handles this is that it assigns a specific block of memory within the computer to hold the value of that variable.*

The size of the block reserved for a variable depends on its data type and the system on which the compiler is being used. E.g., in older systems and integer variable used to be 2 bytes, now a days it is usually 4 bytes. Some embedded systems might still use 2-byte integers these days. Another example can be character variable which is usually 1 byte long.

When we compile the following statement, several things are happening at the same time.

```
int k;
```

1. Compiler sets aside 4 bytes (on a PC) to hold the value of an integer.
2. A symbol table is created where the symbol 'k' is added along with its memory address, where the 4 bytes were set aside earlier.

Thus, with every variable there are 4 properties associated with it.

a. Its name, the identifier. (in our case the symbol, ‘k’).
b. Its memory address. (the 4 bytes set aside reserved for ‘k’) Also called its **L-Value**
c. The data, stored at the memory location. (In our case some garbage value). Also called the **R-Value**.
d. Datatype of the variable (size it takes in the memory). In our case integer takes up 4-bytes.

Consider the code below: -

```
01      int j, k;
02      k = 2;
03      j = 7;
04      j = k;
```

At line 2 when k is used on the left side of the assignment operator the L-Value of k in invoked (from the symbol table) and the value computed on the right side of the assignment operator (here simply the literal 2 thus 2) is stored at that L-value (the memory address location of k).

However, at line 4, when k is used on the right side of the assignment operator the R-Value of k, the data stored at the memory block reserved for k, is invoked i.e., 2. So 2 is copied to the memory address designated by the L-Value of j.

C POINTERS

C **Pointers** are special variables that are designed to hold L-Value (an memory address). In C a pointer variable is specified to the compiler by preceding a variables name with ‘*’ asterisk. Each pointer variable must also have a datatype so that the compiler knows which type of data is being pointed to by the pointer.

```
05      int *ptr;
```

In example above ptr is the pointer variable’s name just we were using k and j to refer to our integer variables. The ‘*’ tells that ptr is a pointer variable and stores memory addresses. The compiler, therefore, always reserves the same size memory block for all pointer variables which is equal to space needed for 1 memory address. In modern systems, this is 4 bytes. The int specifies that the address stored in R-Value of ptr will be an integer.

Right now, we have not stored any address in the variable ptr. Its best practice to set pointer variable to NULL. Pointers initialized with NULL value are called **Null Pointer.**

Back to using our new variable **ptr**. Suppose now that we want to store in **ptr** the address of our integer variable k. To do this we use the unary & operator and write:

```
06      ptr = &k;
```

What the & operator does is retrieve the L-Value (address) of k, even though k is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer **ptr**. Now, **ptr** is said to **"point to" k.**

THE DEREFERENCING OPERATOR '*'

The dereferencing operator (*) is the asterisk and is used to dereference the pointer variable.

When used on the left side of '=' of assignment operator, like this: -

```
*ptr = 11;
```

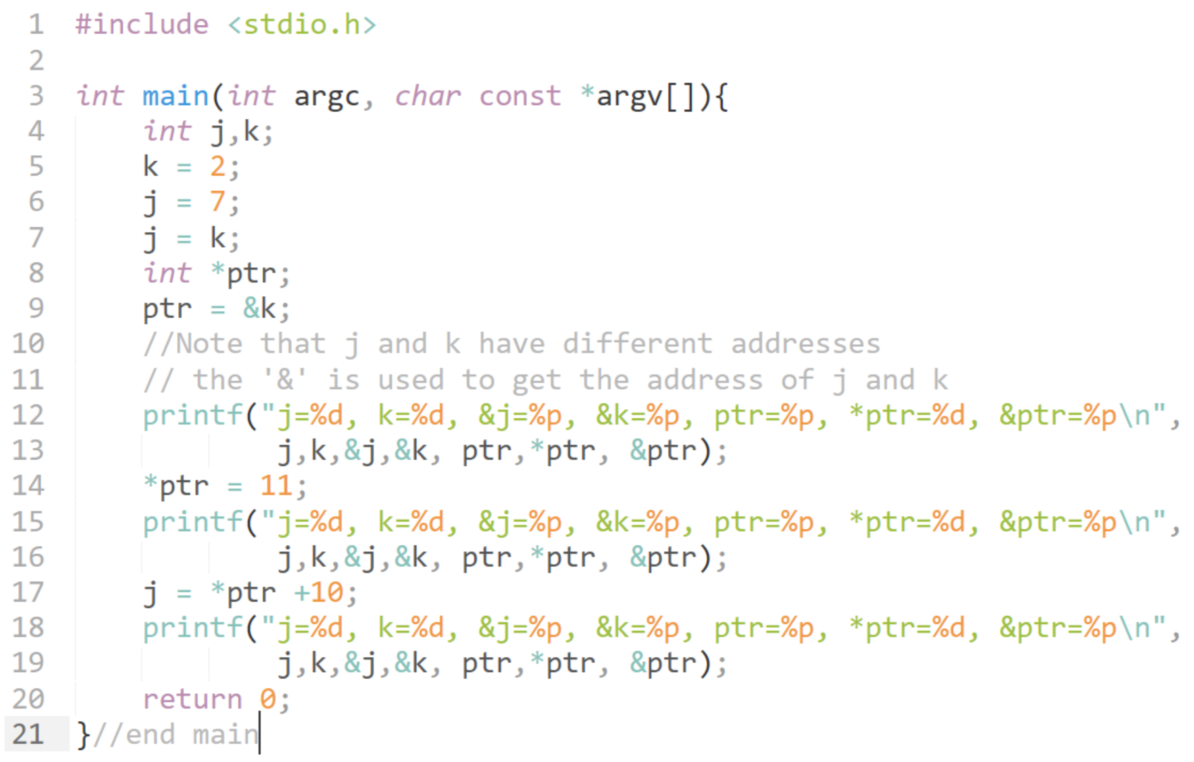
Instead of getting the L-Value of ptr, you get the R-Value and 11 is copied at that address. In our case it will be the address of variable k and now the 11 is stored in that memory location.

When used on the right side of the assignment operator, like this: -

```
j = *ptr + 10;
```

instead of getting R-Value of ptr (i.e. the memory address of variable 'k' which is actually the R-Value of ptr) the value stored at R-Value of ptr is fetched in our case 11 and therefore j will be equal to 21;

Below is the above code combined with some printing statements to make things clear. Type and check it yourself.



```
#include <stdio.h>

int main(int argc, char const *argv[]){
    int j,k;
    k = 2;
    j = 7;
    j = k;
    int *ptr;
    ptr = &k;
    //Note that j and k have different addresses
    // the '&' is used to get the address of j and k
    printf("j=%d, k=%d, &j=%p, &k=%p, ptr=%p, *ptr=%d, &ptr=%p\n",
            j,k,&j,&k, ptr,*ptr, &ptr);
    *ptr = 11;
    printf("j=%d, k=%d, &j=%p, &k=%p, ptr=%p, *ptr=%d, &ptr=%p\n",
            j,k,&j,&k, ptr,*ptr, &ptr);
    j = *ptr +10;
    printf("j=%d, k=%d, &j=%p, &k=%p, ptr=%p, *ptr=%d, &ptr=%p\n",
            j,k,&j,&k, ptr,*ptr, &ptr);
    return 0;
}//end main
```

Appreciate that the pointer variable has its own address too printed by using &ptr.

POINTERS AND ARRAYS

This section is taken from the reference shared as is.

Okay, let's move on. Let us consider why we need to identify the type of variable that a pointer points to, as in:

```
int *ptr;
```

One reason for doing this is so that later, once ptr "points to" something, if we write:

```
    *ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by ptr. If ptr was declared as pointing to an integer, 2 bytes would be copied, if a long, 4 bytes would be copied. Similarly for floats and doubles the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, consider a block in memory consisting if ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integers.

Now, let's say we point our integer pointer ptr at the first of these integers. Furthermore, let’s say that integer is located at memory location 100 (decimal). What happens when we write:

```
    ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 2 to ptr instead of 1, so the pointer "points to" the next integer, at memory location 102. Similarly, were the ptr declared as a pointer to a long, it would add 4 to it instead of 1. The same goes for other data types such as floats, doubles, or even user defined data types such as structures (will be discussed in later labs). This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using "pointer arithmetic", a term which we will come back to later.

Similarly, since ++ptr and ptr++ are both equivalent to ptr + 1 (though the point in the program when ptr is incremented may be different), incrementing a pointer using the unary ++ operator, either pre- or post-, increments the address it stores by the amount sizeof(type) where "type" is the type of the object pointed to. (i.e. 2 for an integer, 4 for a long, etc.).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
    int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to my_array, i.e. using my_array[0] through my_array[5]. But we could alternatively access them via a pointer as follows: -

```
int *ptr;
ptr = &my_array[0]; /* point our pointer at the
                       first integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
01    #include <stdio.h>
02
03    int my_array[] = {1,23,17,4,-5,100};
04    int *ptr;
05
06    int main(){
07          int i;
08          ptr = &my_array[0]; /* point our pointer to
09                   the first element of the array */
10          printf("\n\n");
11          for (i = 0; i < 6; i++){
12            printf("my_array[%d] = %d ",i,my_array[i]);
13            printf("ptr + %d = %d\n",i, *(ptr + i));
14          }
15          return 0;
16    }
```

Compile and run the above program and carefully note lines 12 and 13 and that the program prints out the same values in either case. Also observe how we dereferenced our pointer in line 13, i.e., we first added **i** to it and then dereferenced the new pointer. Change line 13 to read:

```
printf("ptr + %d = %d\n",i, *ptr++);
```

and run it again... then change it to:

```
printf("ptr + %d = %d\n",i, *(++ptr));
```

and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use **&var_name[0]** we can replace that with **var_name**, thus in our code where we wrote:

```
ptr = &my_array[0];
```

we can write:

```
ptr = my_array;
```

to achieve the same result.

This leads many texts to state that the name of an array is a pointer. I prefer to mentally think "the name of the array is the address of first element in the array". Many beginners (including myself when I was learning) have a tendency to become confused by thinking of it as a pointer. For example, while we can write: -

```
ptr = my_array;
```

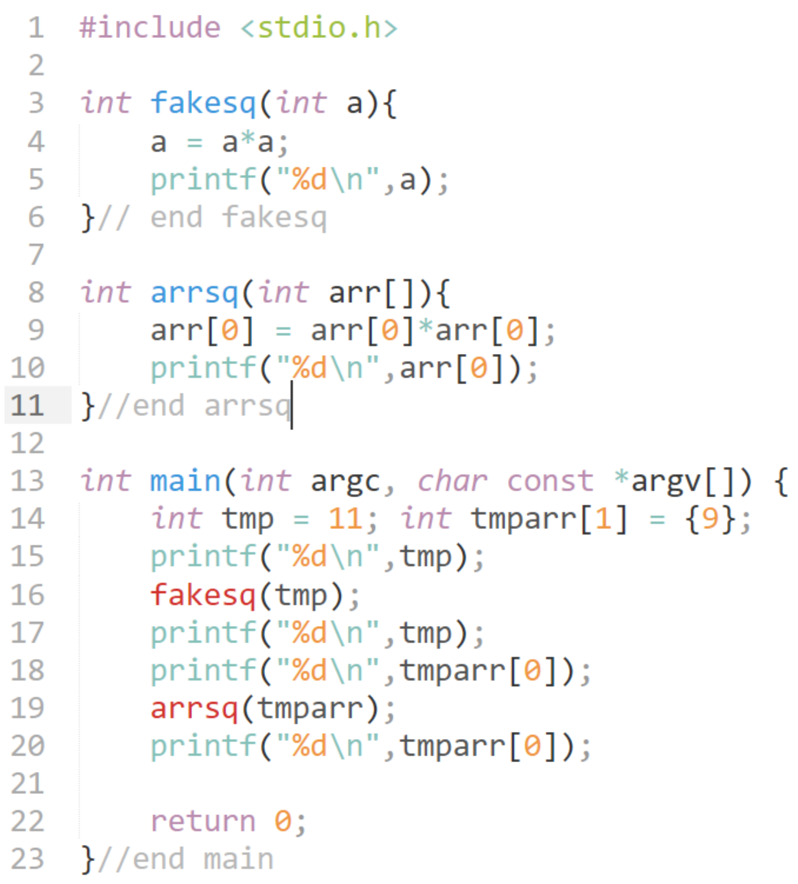
we cannot write: -

```
my_array = ptr;
```

The reason is that while **ptr** is a variable**, my_array** is a constant. That is, the location at which the first element of **my_array** will be stored cannot be changed once **my_array[]** has been declared.

Passing by Value and Passing by Reference

Consider the code below. What will be the output?



```c
#include <stdio.h>

int fakesq(int a){
    a = a*a;
    printf("%d\n",a);
}// end fakesq

int arrsq(int arr[]){
    arr[0] = arr[0]*arr[0];
    printf("%d\n",arr[0]);
}//end arrsq

int main(int argc, char const *argv[]) {
    int tmp = 11; int tmparr[1] = {9};
    printf("%d\n",tmp);
    fakesq(tmp);
    printf("%d\n",tmp);
    printf("%d\n",tmparr[0]);
    arrsq(tmparr);
    printf("%d\n",tmparr[0]);

    return 0;
}//end main
```

The value of tmp variable is unchanged after use in fakesq function, while the value of tmparr[0] has changed after use in arrsq function. Remember, by default all parameter/arguments passing to function in C are by Value. I.e., the value of tmp is send to fakesq and a new variable is created referred to as a with the same value as of tmp. All the statements in the function fakesq are working on this new variable ‘a’ while the original variable ‘tmp’ remains unchanged.

HOWEVER, all arrays are passed as references. Meaning their address is passed to the function e.g., **arrsq** will get the address of **tmparr** stored in the array variable **arr** and will be pointing to the same memory as **tmparr**. Therefore, any change made to **arr** is reflected in **tmparr**.

Using pointers, we can pass addresses of our variables instead of a copy of their value, this way changes made in the function to that memory address will be reflected in our variables as well. As shown in program below: -

```
 1  #include <stdio.h>
 2
 3  int realsq(int *a){
 4      *a = *a * *a;
 5      printf("%d\n",*a);
 6  }
 7
 8  int main(int argc, char const *argv[]) {
 9      int tmp = 11;
10      printf("%d\n",tmp);
11      realsq(&tmp);
12      printf("%d\n",tmp);
13      return 0;
14  }//end main
```

Yes, line 4 is very confusing. This is intentional. Go through pointers again and visit this code again. You will get the hang of it.

TASKS TO DO

TASK 1:

You are given the program below: -

```
1  #include <stdio.h>
2
3  void swap(int a, int b){
4      int tmp =a;
5      a = b;
6      b = a;
7  }
8
9  int main(){
10     int j = 2, k = 5;
11     printf("j=%d, k=%d\n",j,k );
12     swap(j,k);
13     printf("j=%d, k=%d\n",j,k );
14     return 0;
15 }
```

A. Desired result, that is the values of j and k are not swapped. Why is this?
B. Modify the function swap and its call at line 12 to get the desired result.

Write the answer to question A in the comments of the program for part B.

TASK 2:

Given the function prototype below, implement the function that reverses the array passed to its arguments. Also write the main function that demonstrates this by taking 10 inputs from a user and storing them in an array. Print the array, then use our function reverse and print the array again to show that array has been reversed successfully. Use pointers in the function reverse.

```
void reverse(int *arr, int size){
}
```

TASK 3:

Write a C program that makes 3 arrays of size N (N is to be provided by the user, use variable length arrays) of type char, int, long long int. Initialize them with random values. Print the addresses and values stored in all these arrays one by one using pointer arithmetic. Explain in comments why adding to pointers of different types works differently.