An Introduction to Pointers in C

Author:
Manuel Lemos
Mechatronics Engineering Undergraduate
McMaster University
lemosm1@mcmaster.ca

December 11^{th}, 2020

Abstract

Pointers are used in numerous applications of C. As a result, if one wants to use the C language to the fullest of it's capabilities, gaining a thorough understanding of pointers is required. This paper provides an introductory explanation of pointers and their uses, as well as examining various implementations of pointers in the C programming language. A basic understanding of C, in conjunction with knowledge of the concepts of programming, is assumed.

1 Introduction

To many, pointers are a seemingly insurmountable obstacle on the path to learning C. It is often the understanding of this tool, and its effective use, which draws the divide between a programmer who is still a novice to the language and one with more experience. Pointers permeate nearly all concepts of C and enable the flexibility which supports dynamic memory allocation and establishes the basis upon which arrays are notated. Furthermore, function pointers provide an element of control during program execution which cannot be achieved in their absence[1]. At its core, a pointer is an extremely simple concept. Even so, this simplicity is often overshadowed by notation which can catch inexperienced programmers off guard. However, if a strong framework is established, which captures fundamental themes, making use of pointer programming is attainable for any programmer, regardless of experience. Pointers store memory addresses. As a result, gaining a deeper understanding of their behavior can be achieved more easily when paired with comprehension of how memory is managed within the scope of a C program. Thus, at least a basic understanding of how C interacts with memory is strongly encouraged prior to learning pointers. This will make much of C pointer logic feel intuitive and less convoluted. The first section of this paper will provide a foundation to pointers and their basic applications. What they are, their general notation, and their uses in a variety of circumstances will be examined in detail with examples in C. For this section a traditional approach will be taken in that proven methods will be used to convey concepts and ideas. The second section will explore an abstract interpretation of pointers. This will include an in-depth, non-programming analogy which will aim to shed light onto pointers by drawing comparisons to real word examples. When learning a new topic it can be advantageous to think about the concepts as a story to provide an emotional tie to key ideas.

2 Lesson 1: Traditional Approach

2.1 Defining and Declaring a Pointer

On the most fundamental level, a pointer is a special variable whose value is the address of another variable. Recall that an address is a location in memory where data is stored. When a pointer is declared a data type is chosen, followed by an asterisk '*' and then a name is assigned to the pointer. In the following example a pointer of data type integer will be declared. The asterisk informs the compiler that a pointer variable is being created, and thus, the number of bytes to store an address must be allocated in memory[2]. The integer data type is a declaration that this pointer is intended to store the address of a variable of type integer.

```
1 int *ptr_Variable;
```

Listing 1: Declaring a Pointer Variable

It should be noted that the asterisk can be placed anywhere between the type declaration and the pointer name. Spacing is of no consequence.

2.2 Assigning and Dereferencing a Pointer

A pointer with a type declaration of int is said to 'point to an integer'. However, in its current state the pointer is not of much use. What would be useful is if the pointer pointed to, which is to say contained the address of, a variable. To assign an address to a pointer the bit wise & operator is used. Recall that the & operator preceding a operand provides the address of that operand. Logically, this assignment falls in line with expectation as a pointer stores an address while the & operator yields an address. In the listing below, the address of the variable 'a' is displayed. This address is dependent on the compiler being used, and as a result, it can vary from one compiler to another. That being said, the value stored in 'a' will always be 5 as this is assignment which has been made.

```
1 #include <stdio.h>
2 int main(){
3     int a = 5;
```

```
    int *ptr_Variable = &a;
    printf("The address of a: %p",&a);
    printf("\nThe value of a: %d",*ptr_Variable);
}
//output: The address of a: 0x7ffeefbff46c (varies)
//        The value of a: 5
```

Listing 2: Assigning a Pointer Variable

The true value of pointers can start to be unveiled as the discussion of dereferencing begins. Dereferencing occurs when one wants to append the data stored in a variable that a pointer points to. This is done by placing an asterisk as a prefix to the pointer variable name and setting it equal to the new required data. This assignment is demonstrated on line 6 in the listing below.

```
#include <stdio.h>
int main(){
    int a = 5;
    int *ptr_Variable = &a;
    printf("\nThe value of a: %d",a);
    *ptr_Variable = 10;
    printf("\nThe value of a: %d",a);
}
//output: The value of a: 5
//        The value of a: 10
```

Listing 3: Dereferencing a Pointer Variable

Recall previously that a type must be chosen when declaring a pointer. This is done so the compiler will know how many bytes to copy to the memory location stored in the pointer during dereferencing. It is important to note that a pointer only points to the first address in what could be a single data point several addresses long. Thus, the data type of the pointer is extremely important, and must be the same as the type of the data which it points to.

2.3 Pointers and Arrays

Up until now, a true application of pointers has yet to be seen. If a variable needs to be appended or passed to a function it can be done directly without the need for pointers. However, when an array is to be passed to a function, errors arise. C does not permit this because arrays 'decay' into pointers when passed to outside functions. An array in C is simply a series of data points with successive memory addresses. Such an array is described by a pointer, which points to the first memory address in the array, and it's length. Length must be specified prior to run time unless dynamic memory allocation is used. As a result, it is necessary for a programmer to understand pointers so they have the capability of passing arrays to functions. In the following example, an array of type int is declared, initialized, and passed to an outside function. At this point, it has decayed to a pointer, which points to the first address of the array. The array is received by a pointer variable of the same type as the array and manipulated[3].

```
#include <stdio.h>
void arr_Printer(int *passed_Arr){
    printf("%d\n",passed_Arr);
    for(int i = 1; i <= 4; i++){
        printf("%d ",*passed_Arr);
        passed_Arr++;
    }
}
void main(){
    int my_Arr[4] ={1,2,3,4};
    arr_Printer(my_Arr);
}
//output: -272632736
//         1 2 3 4
```

Listing 4: Passing an array to a Function

As a pointer is incremented, as can be seen in the above example on line 6, the value of the memory address it stores is incremented by the size of the type it is declared as. Thus, being of type integer, the memory address would be incremented by 2 or 4 bytes depending on the compiler used. Once again the importance of the type declaration of a pointer can be seen.

2.4 Pointers and Structures

Recall that a structure in C is a framework of a collection of variables. This collection is then used to create objects with the data points of the aforementioned framework. Structures are the basis on which C programmers implement object oriented design[3]. A potential structure could look like the below example which contains a default object "person1".

```
struct human{
    int age;
    char first_Name[30];
    char last_Name[30];
}person1;
```

Listing 5: Constructing a Structure

Imagine now, that hundreds of these structures exist in a file and a goal of reading each one and subsequently printing every first and last name is set about. Thus, the first data point "age" is not to be output. As a result of the large volume of data, it is evident that manually requesting every piece of information would be monotonous. Instead, this printing should be done with a function call. One could then pass that function a pointer to the current structure at hand. In the following example, a single structure will be used for simplicity and spacing, however, the goal remains as set about above; to pass a pointer, which points to a structure, to a function. To go about achieving this, it must first be realized that the pointer being used will point to a structure declared using `struct human`. This structure is used in continuation of the above example. Therefore, the pointer must also be declared `struct human`. From this point the address of the structure can be stored in the pointer, and the pointer passed to the function. The functionality of accessing a given member by dereferencing the pointer is now in place. However, dereferencing a pointer to a structure is slightly more nuanced than one might expect. Two methods are available to achieve this. The first method looks like '(*p).'. Pointing to a structure is extremely common in C. Thus, an alternative and more widely used syntax exists, which is of the form 'p->'. With all of the syntactical formalities out of the way, the function can now be observed.

```
#include <stdio.h>
#include <string.h>
struct human{
    char last_Name[20];
    char first_Name[20];
```

```
    int age;
}person1;
void display_Name(struct human *p){
    printf("%s ", (*p).first_Name);
    printf("%s ", p->last_Name);
}
void main(void){
    struct human *ptr_Passed;
    ptr_Passed = &person1;
    strcpy(person1.first_Name,"Manuel");
    strcpy(person1.last_Name,"Lemos");
    printf("%s ",person1.first_Name);
    printf("%s\n",person1.last_Name);
    display_Name(ptr_Passed);
}
output: Manuel Lemos
        Manuel Lemos
```

Listing 6: Passing a Structure to a Function

Unlike arrays, in C it is possible to pass entire structures to functions[3]. However, in the pursuit of pointer mastery, ignoring this fact is beneficial in constructing a reason to care about what is occurring in the problem. Furthermore, using pointers in this way is a much more efficient use of stack space. Passing structures to functions loads the entire structure onto the stack. If one were to use the pointer method discussed above, one object is loaded at a time. Therefore, in applications with extremely large data sets, or those which require optimization, this method is preferred.

3 Lesson 2: Abstract Interpretation

3.1 Assigning a Pointer: Parking Garage Analogy

One can argue that a number of parallels can be drawn between a pointer in C and a person's pointer finger. Imagine for a moment that two people are in an extremely large parking garage. Person one will be called the instructor and person two will be called the pointer. Not every space is filled in this parking garage, however, every space is labelled sequentially, starting at 1 and counting up. As the two are strolling through the lot, the instructor tells the pointer to point at spot number 7. The pointer respectfully obliges and points to the spot. In this parking spot a red Ferrari 458 is parked. Intuitively, it is known that the pointer finger of the pointer tells the instructor nothing about the car. It does not tell the instructor its colour, or its make and model for that matter. Nothing directly can be ascertained from the pointer's finger. What it does tell the instructor is that, if they look where the pointer is pointing to, they will be able to find the car which is located at the value which was requested. In this case that value is spot number 7. In this example, the spaces in the parking garage represent memory addresses and the instructor represent a programmer. First, the programmer assigns a location to the pointer. This is chosen as spot number 7. Then the programmer prints out the data located at this address, in other words, looking at what the pointer is pointing at. In C this would be represented as shown in the following example.

```
int a;
int *ptr_Variable = &a;
printf("%d",*ptr);
```

Listing 7: Assigning an Address to a Pointer Variable

The pointer variable itself tells a compiler nothing about 'a' aside from data type. The pointer does not tell the compiler what value is stored at 'a' nor does it inform the compiler of the name 'a'. However, because the pointer knows the memory address that 'a' is stored at, it can point the compiler to the location of 'a'. Similarly, the pointer in the analogy does not know what is located in spot 7. However, they know where spot 7 is and point to it for the instructor to observe.

3.2 Dereferencing a Pointer: Parking Garage Analogy Continued

In an identical setup to the previous analogy, an instructor and a pointer are once again strolling through the parking garage. The instructor then tells the pointer to point at spot number 7. Once again, the pointer obliges, and points towards the spot identified by its integer value. However, this time the instructor is not satisfied with the vehicle he finds when he looks where the pointer is pointing. Frustrated, the instructor tells the pointer to point to spot number 52. The pointer points, and so the two travel to the spot. Upon arrival, and once again looking at the car in the line of the pointers finger, the instructor is not satisfied. However, this time the instructor elects to instead swap out the car in that spot with another. Thus, the inspector enters the car, drives it out of the parking garage, and returns in a brand-new Lamborghini. All the while the pointer has not moved a muscle, still pointing at spot 52. This can be likened to a programmer declaring and initializing a variable 'a' and 'b' and then declaring a pointer and setting it equal to the address of the variable 'a'. Then the address stored in the pointer is updated to the address of the variable 'b'. Finally, the value stored in the variable 'b' is appended to a new value. In C this would be

```
#include <stdio.h>
void main(){
    int a = 5, b = 6;
    int *ptr = &a;
    ptr = &b;
    *ptr = 10;
    printf("a = %d, b = %d",a,*ptr);
}
output:  a = 5, b = 10
```

Listing 8: Dereferencing a Pointer Twice

This example is beneficial as it brings to light a sometimes confusing difference. Changing the address which the pointer points to is contrasted with, changing the actual value stored at an address. Both of these instances fall within the scope of dereferencing, however, they have different meanings and different implementations. Knowing the difference between these two is vital in understanding fundamentals of pointers.

3.3 Pointers and Arrays: Parking Garage Analogy Appended

In a similar setup to the previous analogies, an instructor and a pointer exist in a parking garage. However, this time there is also a third person, the customer. The instructor,

being rather brash, cannot talk to the customer as it would scare them away. The pointer can interact with the customer but all they can do is point to the one spot given to them by the instructor. The instructor informs the pointer that 3 cars are parked sequentially starting at spot number 25 and they are ready to be inspected by the customer. Recalling that the pointer can only remember the spot number, the instructor quickly hands the customer a paper with the number 3 written on it, careful not to scare them away. The pointer sets off with the customer and they arrive at spot 25 with the pointer pointing at the car in it. The customer, happy with the inspection of their first vehicle shouts next at the pointer. While the pointer can only remember one spot at a time, they are capable of incrementing the spot they are pointing at by one, and so they do. Following the successful second inspection, the customer shouts next once more, inspects the final car and then leaves. In this analogy the customer is the programmer in an outside function, while the inspector is the programmer in main. This can be likened to a programmer declaring and initializing an array of length 3. The array is then passed to an outside function, in the process decaying. Alongside the array, a variable containing the length of the array is also passed to the function. In the analogy this is represented by passing the customer a note with the number 3 on it. Upon arrival at the outside function, the array's first memory location is set equal to a pointer. The function then iterates over the length of the array printing out each of its values. This iteration occurs by incrementing the pointer by one after each value is printed. This is shown in the analogy by the customer shouting next. In C this would be represented as shown in the following example.

```
#include <stdio.h>
void customer(int *passed_Arr, int cars){
    for(int i = 1; i <= cars; i++){
        printf("car %d ",*passed_Arr);
        passed_Arr++;
    }
}
void main(){
    int my_Arr[3] ={1,2,3};
    customer(my_Arr,3);
}
output: car 1 car 2 car 3
```

Listing 9: Passing an Array to a Function "customer"

This example delves into the decay of arrays in C. For many less experienced programmers this behaviour is entirely unexpected. That being said, when one is aware of the existence of this phenomenon and can appreciate what is occurring, it becomes much easier to deal with.

4 Conclusion

The above content was written with the intention of providing an in depth introduction to pointers, and perhaps more pointedly, their applications in the C programming language. Traditional methods of teaching were used in conjunction with abstract analogies in an attempt to bring a more effective learning experience to readers. As one comes to thoroughly grasp pointers, a deep understanding for the extreme flexibility of coding in C can be appreciated.

References

[1] T. A. Standish, Data structures, algorithms, and software principles. Reading, Mass: Addison-Wesley, 1995.

[2] Y. P. Kanetkar, Let us C. New Delhi, India: BPB Publications, 2019.

[3] T. Jenson, A Tutorial On Pointers and Arrays In C. Independently published , 2017.