Pointers Explained

John Tsiombikas

Abstract

Over the last few years I have watched various people who tried to take their first steps with C or C++. I even took the responsibility of teaching the subject on various occasions. From these experiences I concluded that one of the concepts that novices find most difficult to understand, is pointers. For this reason I decided to write this article, explaining in simple terms the use of pointers in C and C++, to help people grasp this powerful concept, which is central to these languages.

1 Introduction

The C programming language is designed in such a way, as to provide full control of the underlying machine to the programmer. This design decision, makes C a very powerful programming language, able to be used for tasks that simply cannot be done with most other high level languages.

One of the most powerful tools that C provides is the ability to directly access and manipulate the memory of the computer, through a construct called *a pointer*. Note that although I am referring to C, everything applies to C++ as well, since pointers are exactly the same in both languages.

2 Memory and Variables

First let's review the way memory is organized in a computer.

The memory can be thought of as a big linear array of bytes, each one identified by an address, which is essentially the index in that big array. The only difference is that address 0, as far as C is concerned, is not a valid memory location. It is important to understand that the actual address of any byte in memory is just an integer, and nothing more. [1]

The CPU can generally access (read or write) any byte of the main memory, by sending its address to the memory controller in order to "select it" before writing or reading the actual data.

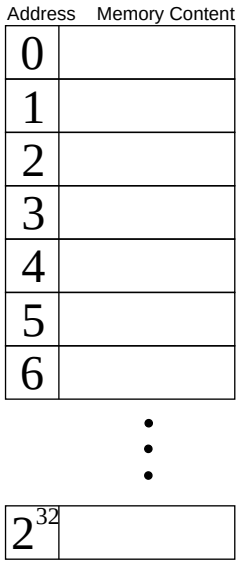

Figure 1: memory organization in a system with 32bit addressing

When we create a variable in a C program, for example: *int x*, the compiler sets aside a block of contiguous memory locations, big enough to fit this variable, and keeps an internal tag that associates the variable name x with the address of the first byte allocated to it. So when we access that variable like this: $x = 10$, the compiler knows where

[1] Technically, this is not specified by the C standard, and in fact some old 16bit systems used a complex addressing scheme involving segments and offsets. But this is beyond the scope of this introductory document, and in general of no interest to anyone writing C for modern computers, or even for those old computers under most circuimstances.

that variable is located in memory, and it simply changes the value there to 10.

2.1 The Size of a Variable

We can easily determine how much space does a variable occupy, with the sizeof operator. For example, on my computer, *sizeof x* gives me the value 4. Which means that an integer needs 4 consecutive bytes in memory to be stored. If the address of x would be 24, then the actual memory locations used by x would be identified by the addresses: 24, 25, 26, and 27.

2.2 The Address of a Variable

C provides a mechanism to determine the address of a specific variable. Remember that if a variable spans multiple bytes, its address is the first byte it occupies.

This is done with a special operator that is called the *address-of* operator, and is represented by the symbol '`&`'. To clearly illustrate the use of the address-of operator, let's see a minimal code fragment that uses it to print the address of a variable.

```
int a;
printf("The address is: %p\n", &a);
```

In the example above, we used the '*%p*' format specifier which tells printf that it is printing an address. In C++ the same thing can also be done with the iostream mechanism, like this:

```
int a;
cout << "address of a is: " << &a
     << endl;
```

C++ does not need any format specifiers, it detects the type of the variable and automatically uses the correct mechanism to print it.

Addresses are usually printed in hexadecimal, as you will see if you try this example, since it is more compact, and easier to read for large numbers (and addresses tend to get pretty large).

As we saw, we can get the address of a variable easily, but where do we store it? I mentioned earlier that addresses are actually just integers, so it would seem a good idea to use an int variable for that purpose. Indeed that is actually what we do when we program directly in assembly. C however, is a high level language which provides us with facilities to manipulate these addresses in a more intuitive way. And in order to be able to do that, the compiler must know that it deals with an address of some variable, and also exactly what is the type of the variable stored in that address. Thus it makes sense to have a specialized type to hold such addresses.

3 Pointers

Pointers are just a special kind of variables that can hold the address of another variable, nothing more, nothing less. There are some new operators that we can use with pointers, and some of the existing operators change their behaviour slightly when acting on pointers, but all in all nothing out of the ordinary.

3.1 Pointer Declaration

When we declare a pointer, we have to specify the type of the variable it will point to. Or to put it differently, the type of the variable whose address it's going to hold. The syntax of the declaration is very simple, we just write the type of the variable that the pointer points to, followed by a an asterisk, followed by the name of the pointer itself. See some examples of pointer declarations below:

```
int *p1;
float *p2;
unsigned int *p3;
char *p4;
void *p5;
```

Here *p1* is a pointer that can point to an *int* variable, *p2* can point to a *float*, *p3* to an *unsigned int*, and *p4* to a *char*. Finally *p5* is a pointer that can point to anything. These pointers are called *void pointers*, and there are some restrictions on what we can do with them.

3.2 Using Pointers

As we said, pointers hold addresses. So it follows that we can assign to a pointer, the address of a variable as obtained by the '&' operator. Let's see a small sample and explain what happens.

```
int a = 10;
int *ptr = &a;
```

Here we first declare an integer named *a* and initialize it to the value 10. Then we create a pointer to int named *ptr* and assign the address of *a* to it. This is called, "making the pointer *ptr* point to *a*." If we depict the first bytes of the memory of that program graphicaly, it will look like Figure 2.

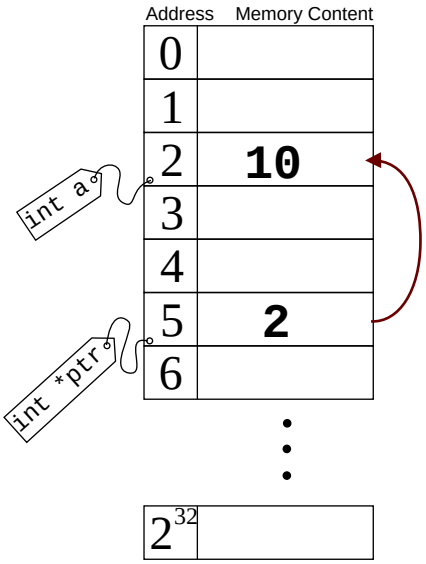

Figure 2: Memory diagram

A very common operation we can do with a pointer is what is called *indirection*, which is a way to access the contents of the memory that it points to. That can be done with the *indirection operator*, that is again represented by the asterisk symbol. Do not confuse this operator with the use of the same asterisk symbol in the delclaration of pointers, they are *not* the same thing.

So if we'd like to access the contents of the memory where *ptr* points to, we would do it like this: **ptr*. Let's see a small code fragment that illustrates pointer indirections.

```
int a = 10;
int *ptr = &a;

printf("a contains the value %d\n", a);
printf("ptr points to %p\n", ptr);
printf("there lies the value %d\n", *ptr);
*ptr = 25;
printf("now a contains the value %d\n", a);
```

Here we first declare *a* and *ptr* just like before. Then we print *a* (which is 10), followed by *ptr*, i.e, the contents of the variable *ptr* which is an address; the address of *a* to be precise. And finally we print **ptr*, which is the value of the memory location where *ptr* points to (again 10, since it points to the location occupied by the variable *a* in memory). Finally we go on and change the contents of the location where *ptr* points to by writing **ptr = 25*, which means *assign the value 25 to wherever* ptr *points to*.

Notice that when we do that, in essence we modify the value of *a*. That should not be surprising, as we have seen that *ptr* holds the address of *a*, and changing the contents of the memory at that address obviously affects the value of *a*.

Below is the output of the program, if the variables were placed as seen in figure 2.

```
a contains the value 10
ptr points to 2
there lies the value 10
now a contains the value 25
```

One limitation of *void pointers*, i.e. pointers that can point to any type, is that they cannot be dereferenced. That makes sense, if we consider the fact that each variable type takes different amount of memory. On a 32bit computer for example usually an *int* needs 4 bytes, while a *short* 2 bytes. So in order to read the actual value stored there, the compiler has to know how many consecutive memory locations to read in order to get the full value.

3.3 Pointer Arithmetic

Another very useful thing we can do with pointers is to perform arithmetic operations on them. This might be obvious to the careful reader, since we said that pointers are just integers. However, there are a few small differences on pointer arithmetic that make their use even more intuitive and easy. Try the following code:

```
char *cptr = (char*)2;
printf("cptr before: %p ", cptr);
cptr++;
printf("and after: %p\n", cptr);
```

We declare a pointer named *cptr* and assign the address 2 to it. We print the contents of the pointer (i.e. the address 2), increment it, and print again. Sure enough the first time it prints 2 and then 3, and that was exactly what we expected. However try this one as well:

```
int *iptr = (int*)2;
printf("iptr before: %p ", iptr);
iptr++;
printf("and after: %p\n", iptr);
```

Now the output, on my computer, is `iptr before: 2 and after: 6`! Why does this pointer point to the address 6 after we incremented it by one and not to 3 as the previous pointer? The answer lies with what we said about the size of variables.

An int is 4 bytes on my computer. This means that if we have an int at the address 2, then that int occupies the memory locations 2, 3, 4 and 5. So in order to access to the *next* int we have to look at the address 6, 7, 8 and 9. Thus when we add one to a pointer, it is not the same as adding one to any integer, it means *give me a pointer to the next variable* which for variables of type int, in this case, is 4 bytes ahead.

The reason that in the first example with the char pointer, the actual address after incrementing, was one more than the previous address is because the size of char is exactly 1. So the next char can indeed be found on the next address.

Another limitation of void pointers, is that we cannot perform arithmetic on them, since the compiler cannot know how many bytes ahaid is the next variable located. So void pointers can only be used to keep addresses that we have to convert later on to a specific pointer type, before using them.

3.4 Arrays and Pointers

There is a strong connection between arrays and pointers. So strong in fact, that most of the time we can treat them as the same thing. The name of an array can be considered just as a pointer to the beginning of a memory block as big as the array. So for example, making a pointer point to the begining of an array is done in exactly the same way as assigning the contents of a pointer to another:

```
short *ptr;
short array[10];
ptr = array;
```

and then we can access the contents of the array through the pointer as if the pointer itself was that array. For example this: *ptr[2] = 25* is perfectly legal. Furthermore, we can treat the array itself as a pointer, for example, **array = 4* is equivalent to *array[0] = 4*. In general **(array+n)* is equivalent to *array[n]*.

The only difference between an array, and a pointer to the begining of an array, is that the compiler keeps some extra information for the arrays, to keep track of their storage requirements. For example if we get the size of both an array and a pointer using the *sizeof* operator; *sizeof ptr* will give us how much space does the pointer itself occupies (4 on my computer), while *sizeof array* will give us, the amount of space occupied by the whole array (on my computer 20, 10 elements of 2 bytes each).

4 Dynamic Memory Allocation

An important use of pointers, is to hold addresses of memory locations that do not have a specific compile-time variable name, but are allocated dynamically while the program runs. In order to do that, we have to call the *malloc* function, which allocates the requested amount of memory, and returns a pointer to that memory. Of course, to deallocate that block of memory, we have to call *free*, passing the pointer as an argument to that function. For example see the following code fragment, in which an array of 10 integers is allocated dynamically, and then freed.

```
int nelem = 10;
int *arr = malloc(nelem * sizeof(int));
/* ... use arr as a regular array .. */
free(arr);
```

In this example, the expression *nelem * sizeof(int)* calculates the amount of bytes we need to allocate for the array, by multiplying the number of elements, to the size of each element (i.e. the size of one integer).

Note that a slightly preferable way to find the size of each element would be *sizeof *arr*, because in

the first example, if we decide to change the type of elements in the array, we would have to also change the sizeof.

5 Function Pointers

Executable code (machine instructions), in the Von Neumann architecture which is tha basis of all computers, are also stored in memory, like data. Thus, we can obtain the address of a C function, store it in a *function pointer*, and then call the function indirectly through that pointer. However, the manipulation of function pointers is limited to assignment and indirection; we cannot do arithmetic on function pointers, since there is no ordering imposed on functions. The following example illustrates how to create and use function pointers.

```
double (*foo)(double);
foo = sin;
printf("sine of 0 is: %f\n", foo(0.0));
foo = cos;
printf("cosine of 0 is: %f\n", foo(0.0));
```

First we create a pointer that can point to functions accepting a double as an argument and returning double, named *foo*. Then we make *foo* point to the standard library function *sin*, and proceed to call it through the *foo* pointer, to print the sine of 0. Finally, we change *foo* to point to *cos*, and call it again in exactly the same manner, to print the cosine of 0.