Pointers Explained

John Tsiombikas

July 18, 2004

Abstract

Over the last few years I have watched various people who tried to do their first steps with C or C++, and I even took on the responsibility of teaching the subject on various occasions. From these experiences I concluded that one of the concepts that most people have difficulty understanding, is pointers. Thus I decided to write this article on the subject of pointers in C and C++, to help people grasp this powerfull tool that these languages provide.

1 Introduction

The C programming language is designed in such a way, as to provide full control of the machine to the programmer. This design decision behind C, makes it a very powerfull programming language, that gives programmers the tools to do things that simply cannot be done with other languages. One of the most powerfull tools that C provides is what is called *pointers*. Pointers are central to C programming, as they provide the ability to manipulate the actual bytes on the memory of the computer. Note that although we are talking about C, everything applies to C++ as well, since pointers are exactly the same in both languages.

2 Memory Organization

First let's review the way memory is organized in a computer, from the viewpoint of the CPU. The CPU can access any part of the main memory, by specifying the address of the exact position in memory that it wants to read or write to the memory controller. This address is an integer, of varying range between computer systems. For the typical PC it is a 32bit number, so the processor can actually access up to 4GB of memory. The bottomline is that the memory is actually seen as a big linear array of values, each having a unique identifying address as we can see in Figure 1. When we create a variable in a C program, for example: `int x;` the compiler sets aside a memory position for this variable, and keeps an internal tag that says where in memory is `x` located. So when we access that variable like this: `x = 10;` the compiler knows where that variable is located in memory, and it simply changes the value there to 10.

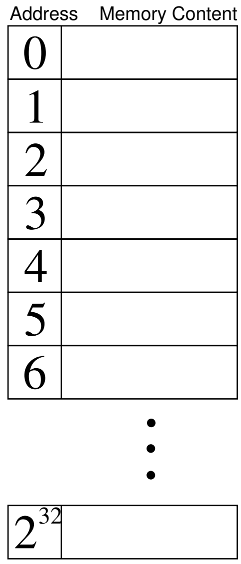

Figure 1: memory organization in a system with 32bit addressing

3 The Address of a Variable

As we said, each variable that we define in a C program is kept at some specific location in memory, and each location in memory has a unique address.

So, since C is designed to provide total control of the computer to the programmer, it also provides a way to get the actual address of the location in memory where a variable is stored in. This is done with a special operator that is called the *address-of* operator, and is represented by the symbol `&`. For start let's see a minimal code sample that illustrates how to print the address of a variable.

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

In the above example, the `%p` is the format specifier that tells printf that it is printing an address. In C++ the same thing can be done like this:

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

C++ does not need to be told what is sent to the cout stream, it detects it automatically and uses the correct printing form. The address is usually printed in hexadecimal form as you will see if you try the above example, since it is easier to read and more compact for large numbers.

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 integer numbers, so it would seem a good idea to use an integer for that purpose, and that is actually what we do when we program directly in assembly. C however, is a high level language where we can manipulate these addresses in a more intuitive way, and be able to tell (and manipulate accordingly) what type of variable exactly lies in that location of memory. So it makes sense to have a specialized type to hold such addresses.

4 Pointers

Pointers are just a special kind of variable 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 slightly when acting on pointers, but all in all it's nothing weird. Still, let's take things from the begining.

4.1 Pointer Declaration

When we declare a pointer, we have to specify the type of variables that it will point to (contain the address of). The syntax of the declaration is very simple, we just write the variable type that the pointer points to, followed by a `*` and the name of the pointer. See some examples of pointer declarations below:

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

Here `p1` is a pointer that can point to an `int` variable, `p2` points to a `float`, `p3` to an `unsigned int`, and `p4` to a `char`. Also there is a pointer type that can point to anything, however there are some restrictions on what we can do with that kind of pointers. These are called *void pointers* and are declared like this:

```
void *p5;
```

4.2 Using Pointers

As we said, pointers hold addresses. So it follows that we can assign to a pointer the address of a variable as taken with 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 10, and then we create a pointer to int named `ptr` and assign the address of `a` to it. If we graphicaly depict the memory of that program it will look like Figure 2.

A very common operation that we can do with a pointer is what we call *indirection*. That means, to access the contents of the memory that it points to, indirectly, through the pointer. That can be done with the *indirection operator*, that is again represented by this `*` symbol. Do not confuse this operator with the use of that symbol in delclaration of pointers, they are *not* the same thing. So if we want to access the contents of the memory location where `ptr` points to, we do it like this: `*ptr;`. Let's see a small code fragment that illustrates pointer indirections.

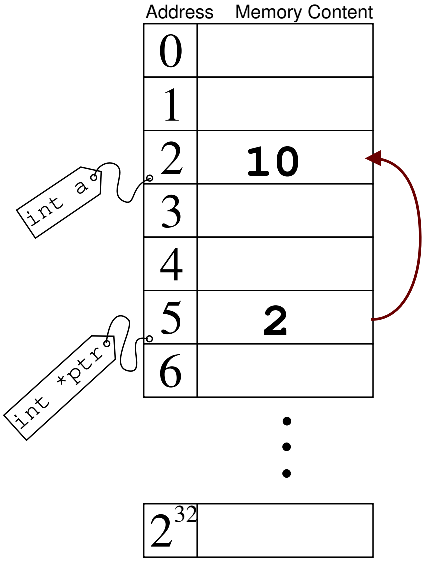

Figure 2: Memory diagram

```
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). After that we print `ptr`, that is, the contents of the variable `ptr` which is an address, the address of `a` to be specific. 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 of `a` in memory). Then 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*. That in essence changes the value of `a` which we then print to illustrate that change. Below is the output of the code as it would be printed if the figure 2 was real.

```
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* (pointers that point to anything), is that they cannot be dereferenced. And that makes sense, if we consider the fact that each data type takes different ammount of memory to be stored. On a 32bit PC for example an `int` needs 4 bytes, while a `short` 2 bytes. So in order to get the actual value stored there, the compiler has to know how many consecutive memory locations to read in order to get the full value.

4.3 Pointer Arithmetic

Another very usefull thing we can do with pointers is to do arithmetic on them. That might be expected of course, since as we said pointers are just integers, and you can do arithmetic operations on integers. However, there is one small difference on pointer arithmetic that makes 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, increment it, and print again. Surely 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 machine, 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 is that the arithmetic operations on pointers are designed to be intuitive. An `int` is 4 bytes on my machine, that means that if we have an `int` at the address 2, then that `int` occupies the locations 2, 3, 4 and 5. So to get to the *next* `int` we have to look at the address 6, 7, 8 and 9. So when we add one to a pointer, it is not the same as adding one to any integer, it means *move this pointer to the next value* which in the case of `int` on my machine 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 `char` data type needs exactly one byte of storage, so the next char is indeed on the next byte.

Another limitation of `void` pointers, is that we cannot perform arithmetic on them. Since the compiler cannot know how many bytes to increment it, to get to the next value. So `void` pointers can only be used to keep addresses there, that we then cast to a specific pointer type before using.

4.4 Arrays and Pointers

There is a strong connection in C between arrays and pointers. So strong indeed that most of the time, we can treat them as the same thing. The fact is that the name of an array can be considered as just a pointer to the begining of a memory block that is 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 the following: `*array = 4;` is equivalent to: `array[0] = 4;`, and 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 on an array. For example if we get the size of each using the `sizeof` operator, for the pointer we will get the size of a pointer in bytes, that is, how much space does a pointer occupy in memory. While in the case of the array, we will get the size that the array occupies in memory, which is the number of elements in the array times the size of each element. So on my machine, for the above example, `sizeof(ptr)` gives me 4, while `sizeof(array)` gives 20 (10 elements, 2 bytes each).