Pointers, Arrays and Strings

Slide 2: Remember

Before we start, remember the two operators we have discussed earlier. `&` operator fetches the address of a variable and stores the address in a pointer variable. `*` operator is used in dereferencing i.e. `*` operator (always used on a pointer) goes to the address the pointer points to and retreives the content of the address.

Slide 3: Pointer to an Array

As the picture shows, to get the address of `v[0]`, assuming it to be like a variable, we fectch it's address using the `&` operator as usual i.e. `vPtr = &v[0]`. The address `3000` is fetched and stored in the pointer `vPtr`.

Slide 4: Pointer to an Array

Similarly, to fetch the address of `v[2]`, we use the `&` operator as above i.e. `vPtr = &v[2]`. Now the content of `vPtr` becomes 3008. Notice that since the array consists of integers, each element occupies 4 bytes of memory!

Slide 5: Pointer to an Array

We can fetch the address of each and every element in the array in the above fashion. The following code in fact does the same and prints the address. Notice that the pointer contents are printed using the `%p` format specifier.

```
for(i=0; i<5; i++){
  vptr = &v[i];
  printf("%p ", vptr);
}
```

The above code snippet fetches the address of each and every element in the array and prints it i.e. the output of the code snippet would be `3000 3004 3008 3012 3016`. However, it is worth noting that arrays are stored in contiguous memory locations and we are still using `&` operator based address fetching that is predominantly used for arbitrary variables located in arbitrary memory locations. We are not clearly taking advantage of the array property that being all elements are in continous memory locations. There must clearly be a better way of fetching *consecutive* addresses in case of arrays!!

Slides 6, 7 and 8: Array Organization

In fact if you notice, in case of arrays, only the starting address i.e. address of `v[0]` is crucial. The rest of the addresses can be obtained using simple math if we know the type of the array elements!! Assuming that `vPtr` now points to `v[0]` that is it holds the address of `v[0]` which is 3000, the expression `vPtr = vPtr + sizeof(int) * i` if used in a loop will fetch the address of all consecutive array elements. Find below the expression worked out for increasing values of `i`. Can you see that for each value of `i`, each address is computed.

```
vPtr = vPtr + sizeof(int) * i
vPtr = 3000 + 4 * 0 = 3000
vPtr = 3000 + 4 * 1 = 3004
vPtr = 3000 + 4 * 2 = 3008
vPtr = 3000 + 4 * 3 = 3012
vPtr = 3000 + 4 * 4 = 3016
```

Once again, notice that you need three information to fetch the address of any arbitrary element in an array viz. *The starting address, the type of the array elements and the index of the array element you need to fetch the address for!!!*.The last two information being obvious, essentially what we need is the starting address of an array.

<u>Slide 9: Pointer Arithmetic</u>

If you look closely at the expression `vPtr = vPtr + sizeof(int) * i`, you can see that the `vPtr` might have been already declared as an integer pointer, so the `sizeof(int)` is redundant as the type can be deduced from the type of the pointer itself!! So `vPtr = vPtr +` ~~`sizeof(int) *`~~ `i`. So the computation becomes,

```
vPtr = vPtr + i
vPtr = 3000 + 0 = 3000
vPtr = 3000 + 1 = 3004
vPtr = 3000 + 2 = 3008
vPtr = 3000 + 3 = 3012
vPtr = 3000 + 4 = 3016
```

You can notice above that addition of each integer is substituted by the size of integer i.e. `3000 + 1 = 3004 (not 3001!!)`. In fact the pointer type knows its *arithmetic*.

<u>Slide 10: Pointer Arithmetic</u>

So the code snippet shown in Slide 5, can be rewritten as

```
vPtr = &v[0];
for(i=0; i<5; i++){
 printf("%p ", vPtr+i);
}
```

Fortunately, since the pointers type know their arithmetic in fetching addresses, the above succint code snippet precisely does what the code snippet in Slide 5 did!!

Slide 11: Increment Pointer

Considering the fact that `vPtr+i` is a regular arithmetic expression and that `i` changes incrementally, we can use incremental short hand operator notation here too such that `vPtr+i` becomes `vPtr++`. Subsequently, the code snippet becomes

```
vPtr = &v[0];
for(i=0; i<5; i++){
 printf("%p ", vPtr++);
}
```

The pointer arithmetic has been seamlessly made a part of the C syntax!!

Slides 12, 13, 14 and 15

The pointer arithmetic in fact works for subtraction too (but not for multiplication and division as you can see clearly why so?!). Consequently, the code snippet which we have been discussing can be rewritten as below

```
vPtr = &v[4];
for(i=0; i<5; i++){
 printf("%p ", vPtr-i);
}
```

Slides 13-15 show the computation of expression and decrement operator for subtraction.

Slide 16: Pointer Arithmetic

Another (consistent) aspect of pointer arithmetic is shown in this slide. Assuming that `vPtr` and `v2Ptr` respectively points the first `v[0]` and fith array element `v[4]`, their difference obviously shows that there are 4 integer elements between them. The resultant 4 has been explained as `v2Ptr` pointing to 4th element from starting address and `vPtr` as pointing to the starting address (i.e. address of the first element). Hence their difference is supposed to yield 4 (which can be confirmed by the number of elements between these two addresses).

Slide 17: Pointer Arithmetic

All the above discussions culminating in the pointer arithmetic till the last slide holds good only in the case of arrays (or as a matter of fact to any compound data type that stores its elements in contiguous memory locations). That definitely makes sense!!

Slides 18 and 19: Pointer Arithmetic

Another interesting observation in all the code snippets is that the `vPtr` always points to the first element in the array. The expression `vPtr = &v[0]` is still a remnant of our earlier usage for arbitrary variables. If by default, we want the pointer to an array always point to the address of the first element, the above expression is still redundant.

Slide 20: Base Address

Instead of using the expression `vPtr = &v[0]` to fetch the address of the first element of an array, in C it has been designed in such a way that the name of the array itself, by default, points to the address of the first element!!! So the above expression can be simply written as `vPtr = v;` The address of the first element of the array is usually called as the base address of the array.

Slide 21: Base Address

That means, going by the pointer arithmetic, `v+1`, `v+2`, `v+3` and `v+4` respectively points to the 2nd, 3rd, 4th and 5th elements of the array. Note that `v+0` (which essentially is `v`) points to the base address.

Slide 22: Base Address

So the name of the array is indeed a pointer, pointing to the first element of the array. However the name of the array `v`, in the context of pointer, is a constant pointer. So expressions like `v += 3` is invalid and causes error if attempted. This is obvious because in C the name of the array in the context of pointer always points to the base address. If the above expression is possible, then `v` will point to the fourth element which violates the very design.

Slide 23: Pointer Offset Notation

Once we ae convinced about the expressions, `v+1`, `v+2`, `v+3` and `v+4` and understand what respectively they point to, it is straightforward to see that derefencing operator can be used on them to access the respective element values i.e `*(v+1)`, `*(v+2)`, `*(v+3)` and `*(v+4)` respectively access the values of 2nd, 3rd, 4th and 5th elements of the array.

Slides 24, 25, 26 and 27

If `v` is the name of the array, then `vPtr = v` (where `vPtr` is a pointer) copies the base address to `vPtr`. Then as `vPtr` is same as `v`, `vPtr[1]`, `vPtr[2]`, `vPtr[3]` and `vPtr[4]` (like `v[1]`, `v[2]`, `v[3]` and `v[4]`) are valid notations, that represent respectively the values of 2nd, 3rd, 4th and 5th elements of the array. As you can see we use subscript notation using a pointer. However, so far you have seen subscript notation used often for arrays.

So the following notations can be used interchangeably to point the base address of an array `v`, `vPtr`, `(v+0)` and `(vPtr+0)`.

Similary the following notations can be used interchangeably to represent the first element of the array (it's value not the address!!) `v[0]`, `vPtr[0]`, `*(v+0)` and `*(vPtr+0)`.

Slide 28: Initialize Strings

Though we have seen initialization of strings, this slides present another way of initializing strings in the context of pointers. In `char *c = "abcd"` the declaration part states that `c` is a pointer that points to characters and initialization part states that `c` points to the base address of the array (which is essentially a string) `"abcd"` that is the address where the character `a` is stored.

Slide 29: If Strings are arrays

In the following function

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
  printf("msg occupies %i bytes\n", sizeof(msg));
}
```

When we pass the array, what we essentially pass is the base address of the array!! So `msg` is actually a pointer and when is called in `printf()` it understands the context and accesses the entire array/string using pointer arithmetic. `sizeof()` operator usually fetches the size of the input type. In this case, what gets printed in the second statement is the size of the pointer not the string!!

Slide 30: An Array isn't a Pointer

In fact the title slide shows one of the "The Simpsons" characters the Bart Simpson writing *An Array is not a pointer*. This slide kind of explains the differences between the two.

When you take the `sizeof()` of `s` and `t` you see the difference. While the former returns 15, the latter returns either 4 or 8 as the case may be. When you take the address of `s` it is `s` itself!! Because `&s` gives the base address which is what `s` stands for!! But if you take `&t` as you know it gives the address of the pointer which is a piece of memory different than the array itself.

Slide 31: Update String Literal

In the first declaration-cum-initialization, `*cards` is just a char pointer. However, the string literal `"JQK"` is read-only. In case of second declaration-cum-initialization, `*cards` is a constant pointer holding the address of first location of 3 continuous memory locations. So the string can be modified at individual locations.

Slides 32 and 33 are self explanatory

Slides 34 and 35

Slide 34 provides the rationale for why functions could also have pointers and slide 35 provides an example. Though we are not getting into the details, if you are interested you can pursue it yourselves.