In this 2d array
Int a[2][2]
How *(a+1) = &a[1][0]
and *(a+1) +1 = &a[1][1]
and *(*(a+1)+1) = a[1][1]

In C, a 2D array is stored in memory as a contiguous block. Let's break down the expressions you provided:

1. `*(a+1) = &a[1][0]`: This is due to pointer arithmetic. `a+1` moves to the next row, and `*(a+1)` gives the address of the first element in the second row, which is equivalent to `&a[1][0]`.
2. `*(a+1) + 1 = &a[1][1]`: Here, `*(a+1)` points to the first element of the second row, and adding 1 moves to the next element in that row, resulting in the address of `a[1][1]`.
3. `*(*(a+1)+1) = a[1][1]`: This expression first moves to the second row with `*(a+1)`, then adds 1 to point to the second element in that row, which is equivalent to `a[1][1]`. The outer `*` dereferences this address, giving the value at `a[1][1]`.

So, these expressions are related to the memory layout of a 2D array in C and how pointer arithmetic works with arrays.

How he know that he have to move to the next row or to thr next column

In C, when you perform pointer arithmetic on a multi-dimensional array, the behavior is determined by the way the array is stored in memory. A 2D array in C is essentially a contiguous block of memory, and the compiler interprets it as a one-dimensional array.

Consider the declaration `int a[2][2]`. The memory layout is as follows:

```
a[0][0] | a[0][1] | a[1][0] | a[1][1]
```

When you use pointer arithmetic, the pointer moves based on the size of the elements it points to. In this case, each element is of type `int`, so moving by 1 in the pointer arithmetic corresponds to moving to the next `int` in memory.

For `*(a+1)`, it moves to the next row because it jumps over the elements of the first row.

For `*(a+1) + 1`, it moves to the next column within the same row because it adds 1 to the address of the first element in the second row.

For `*(*(a+1)+1)`, it first moves to the second row with `*(a+1)`, then adds 1 to move to the next column in that row.

The behavior is determined by the data type of the array elements and the row-major order storage of multi-dimensional arrays in C.

I don't get it

Let me simplify it. In C, a 2D array is stored in memory as a contiguous block, row by row.

For the array `int a[2][2]`:

```
a[0][0] | a[0][1]
a[1][0] | a[1][1]
```

- `a` is the address of the beginning of this block.
- `a + 1` moves to the next row because each row has two elements (2 ints), and adding 1 means moving to the next row.
- `*(a + 1)` gives the address of the first element in the second row, which is equivalent to `&a[1][0]`.
- `*(a + 1) + 1` then adds 1 to this address, moving to the next column in the second row, resulting in the address of `a[1][1]`.
- `*(*(a + 1) + 1)` dereferences this address, giving the value stored at `a[1][1]`.

So, it's about understanding the memory layout of the array and how pointer arithmetic works in conjunction with that layout. If it's still unclear, feel free to ask more specific questions!

(a+1)+1

`(a+1)+1` refers to moving to the next element in memory after the one pointed to by `(a+1)`. In the context of a 2D array `int a[2][2]`, let's break it down:

- `a` is the address of the beginning of the array.
- `(a+1)` moves to the next row because each row consists of two elements (2 ints), and adding 1 means moving to the next row.
- `((a+1)+1)` then adds 1 again, which means moving to the next element in the same row.

So, `(a+1)+1` effectively points to the element in the next row and the next column in the original 2D array.

Why , how *(a + 1) gives the address of the first element in the second row, which is equivalent to &a[1][0].

Let's break down the expression `*(a + 1)` in the context of a 2D array `int a[2][2]`.

1. `a` represents the address of the beginning of the array.
2. `(a + 1)` increments the address by the size of one row. In this case, each row has two elements (2 ints), so adding 1 moves to the next row.
3. `*(a + 1)` dereferences this address, effectively giving the value stored at the beginning of the second row.

Now, since `*(a + 1)` gives the value at the beginning of the second row, it is equivalent to `&a[1][0]`, which represents the address of the first element in the second row of the 2D array.

In simpler terms, `*(a + 1)` moves to the second row, and the dereference (`*`) gives the value at the beginning of that row. This value is the address of the first element in the second row, which is what `&a[1][0]` also represents.

(a+1)+1 = *(a+1)+1 ?

No, `(a+1)+1` and `*(a+1)+1` are not the same.

1. `(a+1)+1`: This moves to the next row in the 2D array because `(a+1)` represents the address of the first element in the second row, and adding 1 moves to the next row.
2. `*(a+1)+1`: This involves dereferencing first. `*(a+1)` gets the value at the beginning of the second row (which is the address of the first element in the second row), and then adding 1 moves to the next element in the same row.

So, while both expressions involve moving to the next row and column, the dereference operation in the second expression affects how the addition is applied.

Is this like we have **p that point to *p[0] and *p[1]

Yes, the analogy you provided is somewhat similar to the concept of a 2D array in C.

If you have a pointer to a pointer, such as `**p`, it's often used to represent a dynamic 2D array. Each `*p[i]` points to a 1D array (or a row in a 2D array), and `**p` points to the first element of the first row.

Here's a breakdown of the analogy:

- `**p`: Equivalent to the first element in the 2D array.
- `*p[0]`: Equivalent to the first row of the 2D array.
- `*p[1]`: Equivalent to the second row of the 2D array.

In the case of a 2D array `int a[2][2]`, you can think of it as a fixed-size version of a similar dynamic 2D array represented by `**p`. The syntax and memory organization differ, but the conceptual idea is similar.

That you say that *p it is the value that store in p[0] that is the address of the first element

Yes, exactly. When you have a pointer `*p`, it typically represents the value stored at the memory location it points to. In the context of a dynamic 2D array:

- `*p` would represent the value stored at the memory location pointed to by `p`, and in the analogy, it could be considered similar to the address of the first element in the 2D array.
- If `*p` is used in the context of a dynamic array of pointers, it might represent the address of the first element in the array, which itself could be another pointer, leading to the idea of a pointer to a pointer (`**p`).

Keep in mind that the exact interpretation depends on the specific context and how the pointers are being used.