

```

#include <stdio.h>

int main(void)
{
    int n;
    int a[5];
    int *p;

    a[2] = 1024;
    p = &n;
    /*
    * write your line of code here...
    * Remember:
    * - you are not allowed to use a
    * - you are not allowed to modify p
    * - only one statement
    * - you are not allowed to code anything else than this line of code
    */
    *(p + 5) = 98;
    /* ...so that this prints 98\n */
    printf("a[2] = %d\n", a[2]);
    return (0);
}

```

why we use `*(p + 5)` instad of `*(p + 3)`

The reason the code uses `*(p + 5)` instead of `*(p + 3)` to make `a[2]` equal to 98 is due to the memory layout of local variables in the stack and the behavior of compilers.

Let's break it down:

- 1 Local variables are typically stored on the stack, and the stack grows downwards in memory (i.e., from high memory addresses to low ones).
- 2 The declaration order of variables can determine the order they appear in the stack. In your code, the variable `n` is declared before the array `a[]`. Therefore, on many platforms and compilers, `n` is placed at a higher memory address than `a[]`.
- 3 The statement `p = &n;` sets `p` to point to the address of `n`.

Now, if you were to visualize the memory layout:

```

Higher addresses
+-----+
|   n   |
+-----+
| a[0]  |
+-----+

```

```

|  a[1]  |
+-----+
|  a[2]  |
+-----+
|  a[3]  |
+-----+
|  a[4]  |
+-----+
Lower addresses

```

If p is pointing to n , then $*(p + 1)$ would typically point to $a[0]$, $*(p + 2)$ to $a[1]$, $*(p + 3)$ to $a[2]$, and so on.

However, in the code you provided, $*(p + 5)$ is used to access $a[2]$ instead of the more intuitive $*(p + 3)$. This discrepancy might be due to a combination of reasons:

- 1 **Padding/Alignment:** Compilers sometimes add padding to ensure data is aligned correctly in memory. This might result in some "gaps" between the memory locations of variables, especially if there are optimizations or specific architectures in play.
- 2 **Compiler Optimizations:** Depending on optimization settings, the compiler might rearrange local variable storage for efficiency.
- 3 **Stack Layout:** There might be other elements (like saved registers) between the local variables on the stack.

Without more information, it's hard to pin down the exact reason. However, relying on such behavior is generally a bad idea because it's undefined behavior according to the C standard. This code may not work consistently across different compilers, optimization levels, or architectures.

i don't under stand

Alright, let's simplify this:

Imagine you have a row of boxes. Some of these boxes contain your variables. The box for n is located to the left, and the boxes for the $a[]$ array are located to its right.

```

|  n  |  a[0]  |  a[1]  |  a[2]  |  a[3]  |  a[4]  |

```

When you do $p = \&n;$, you're making p point to the n box.

```

|  n*  |  a[0]  |  a[1]  |  a[2]  |  a[3]  |  a[4]  |

```

The asterisk (*) indicates where p is pointing.

Now, if everything was super straightforward, $*(p + 3)$ should change the value of $a[2]$ since it's the third box to the right of n . But in the code you showed, we use $*(p + 5)$ to change the value of $a[2]$.

This suggests that there might be some "invisible" boxes (or gaps) between n and the $a[]$ array. So the memory might look more like this:


```
| n | ??? | ??? | a[0] | a[1] | a[2] | a[3] | a[4] |
```

In this hypothetical layout, there are two "invisible" boxes (or gaps) between `n` and `a[0]`, and this is why you have to use `*(p + 5)` to modify `a[2]`.

But why are there gaps?

- Computers sometimes add gaps for efficiency.
- The program's environment might add some information in those gaps.
- The way your code is compiled might introduce these gaps.

It's important to understand that this behavior is not guaranteed. If you move to a different computer or compiler, these gaps might change or disappear. So, using such techniques can lead to unpredictable results. Always ensure you're accessing memory that you've properly allocated or declared to avoid unintended consequences.

From ChatGPT, designed with  FancyGPT