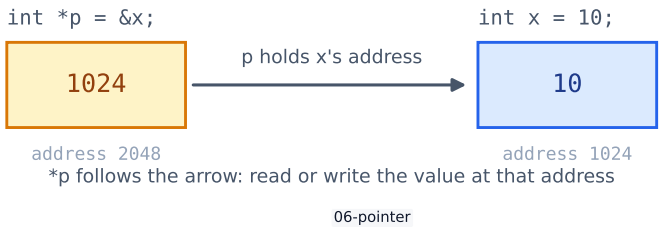
Pointers

C Reference

&, * and pass-by-pointer

A pointer 指针 stores the address 地址 of a variable. `&x` gives the address of `x`; `*p` dereferences 解引用 the pointer — it reads or writes the value stored there. Passing a pointer lets a function change the caller's variable (pass-by-pointer).



A pointer stores an address; dereferencing it follows the arrow to the value

```c
#include <stdio.h>

void addOne(int *p) {     // p holds an address
    *p = *p + 1;          // change the value at that address
}

int main(void) {
    int x = 10;
    int *ptr = &x;        // & takes the address of x
    printf("%d\n", *ptr); // * reads the value: 10
    addOne(&x);
    printf("%d\n", x);    // 11 - changed through the pointer
    return 0;
}
```


The classic use is a swap function. Passing values only copies them — the swap is lost. Passing pointers lets the function reach the caller's variables:

```c
#include <stdio.h>

void swap_values(int a, int b) {      // copies: the caller sees nothing
    int t = a; a = b; b = t;
}

void swap_pointers(int *a, int *b) {  // addresses: the swap is real
    int t = *a; *a = *b; *b = t;
}

int main(void) {
    int x = 1, y = 2;
    swap_values(x, y);
    printf("%d %d\n", x, y);   // 1 2  (unchanged!)
    swap_pointers(&x, &y);
    printf("%d %d\n", x, y);   // 2 1  (swapped)
    return 0;
}
```

An array name acts as a pointer to its first element, and pointer arithmetic 指针运算 steps by whole elements:

```c
#include <stdio.h>

int main(void) {
    int a[] = {10, 20, 30};
    int *p = a;                  // same as &a[0]
    printf("%d\n", *p);          // 10
    printf("%d\n", *(p + 2));    // 30 - same as a[2]
    return 0;
}
```

- `NULL` is the pointer that points at nothing — check for it before you dereference.

Common mistakes

- `&x` is the address of `x`; `*p` is the value stored at `p`.
- Never use `*p` on an uninitialised or `NULL` pointer — it crashes or corrupts memory.
- To change a caller's variable, pass its address and write through the pointer.