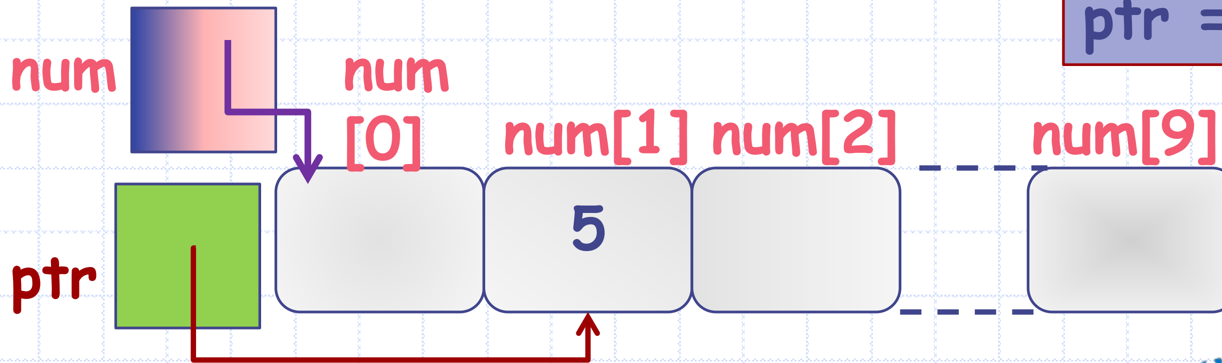
OK, i see. The program fragment below results in this memory state.

```
int num[10];
int * ptr;
ptr = &num[1];
```



question

1. Yes! scanf("%d",ptr) reads input integer into the box pointed to by the corresponding argument.
2. The box pointed to by ptr is num[1].
3. So num[1] becomes 5.

Suppose I now add the following statement after above fragment

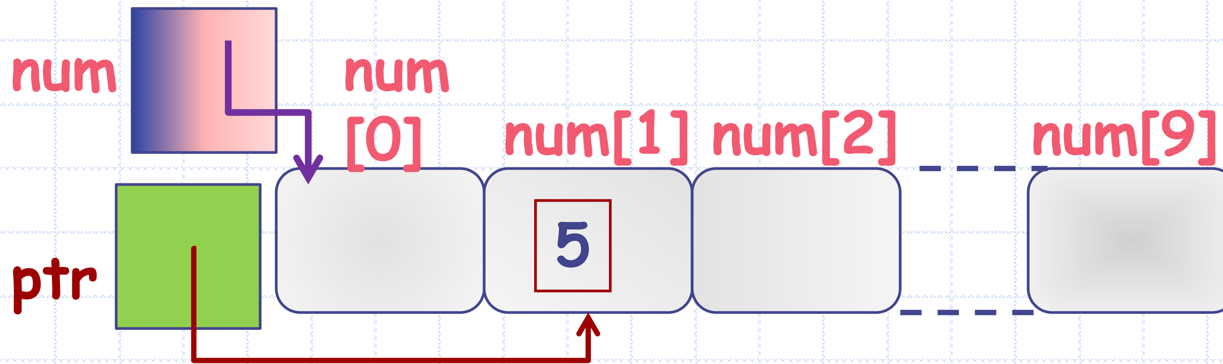
```
scanf("%d",ptr);
```

and input is :

Input

5

Does num[1] become 5?



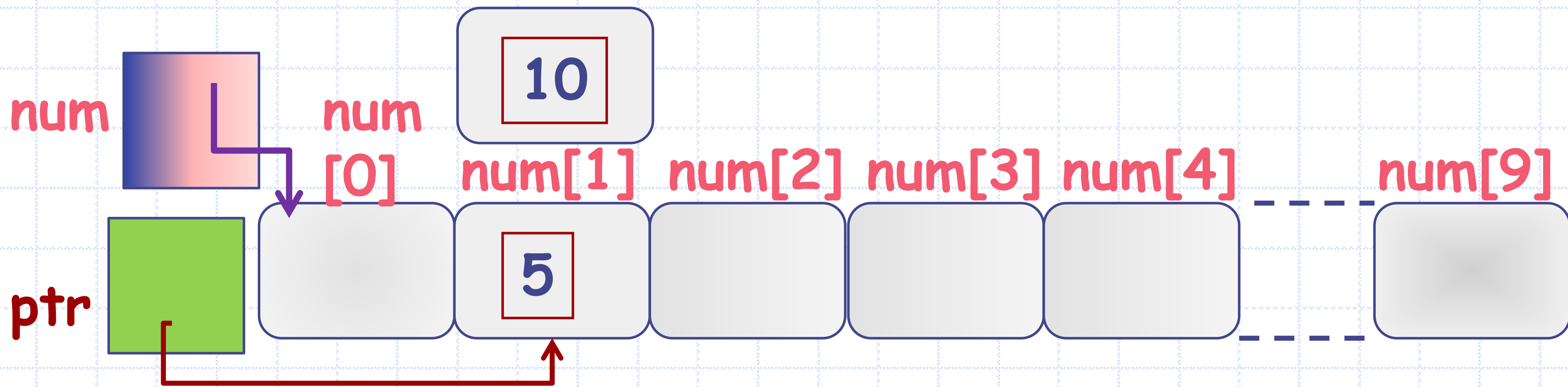
num is of type int [] (i.e., array of int). In C the box num stores the pointer to num[0]. Internally, C represents num and ptr in the same way. So the type **int *** can be used wherever **int[]** was used.

Well, what else can you do with ptr?

What's so interesting? Please give examples.

Here are the interesting parts! You can

1. de-reference the pointer.
2. do simple arithmetic + - with pointers.
3. compare pointers and test for ==, <, > etc., similar to ordinary integers.



De-referencing a pointer **ptr** gives the box pointed to by **ptr**. The de-referencing operator in C is also *.

```
printf("%d", *ptr);
```

Output 5

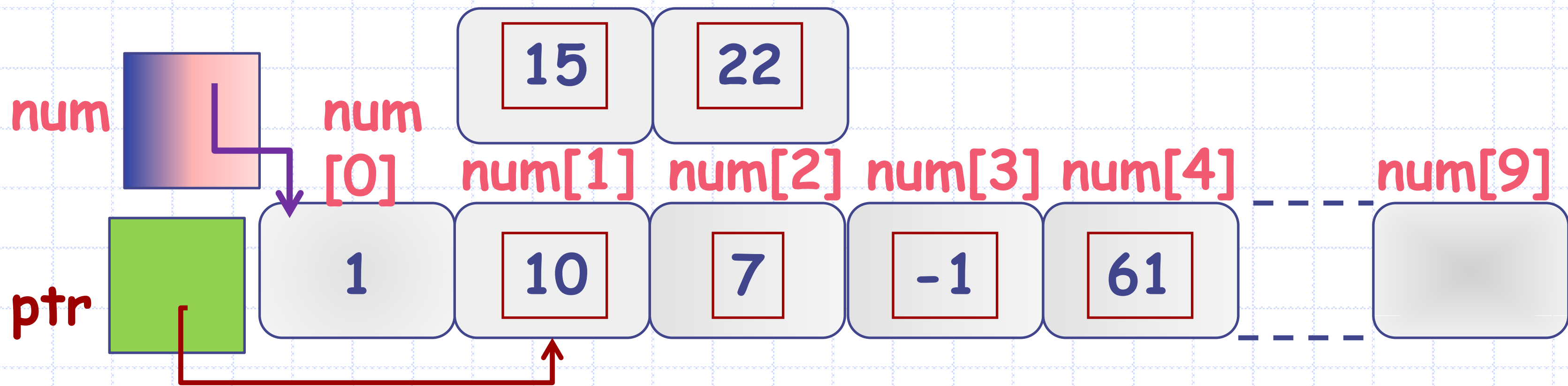
Hmm...

Since ptr points to num[1], *ptr is the box num[1]. Printing it gives the output 5.

Consider statement

```
*ptr = *ptr + 5;
```

This will add 5 to the value in box pointed by ptr. So num[1] will become 5+5 = 10



Recall rule about pointers:

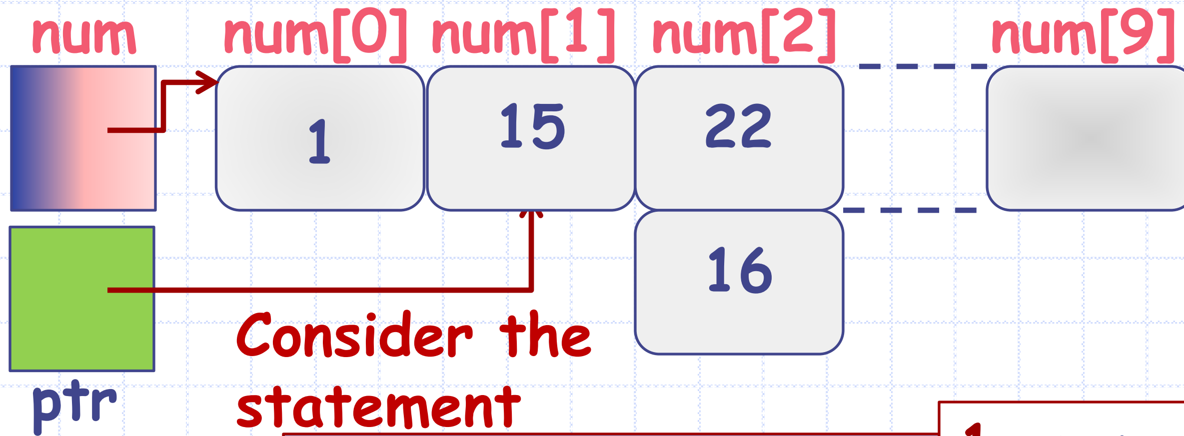
De-referencing a pointer **ptr** gives the box pointed to by **ptr**. The de-referencing operator in C is *****.

Consider the statements. Execute them on above memory state.

```
*ptr = *ptr + 5;
num[2] = num[1]+num[2];
```

OK....

1. 1st statement will add 5 to the value in box pointed by ptr. So *ptr becomes 10 + 5 = 15.
2. But *ptr and num[1] are the same box. So 2nd statement assigns 15 + 7 equals 22 to num[2].



Hmm... seems **legal**— 'cause

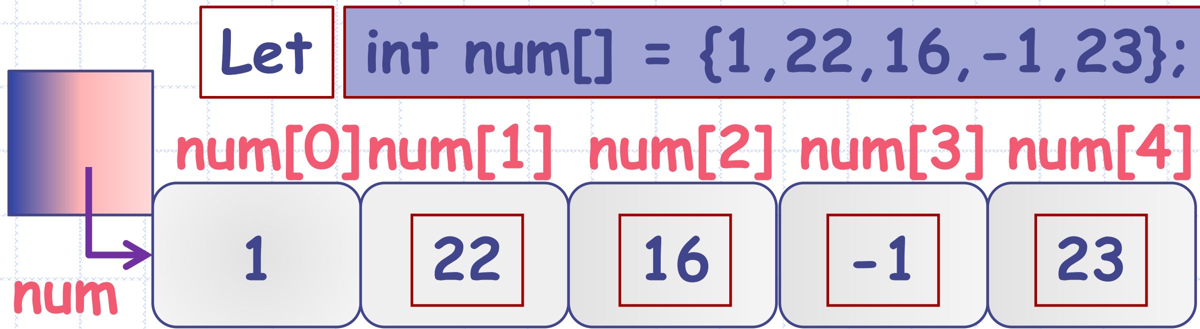
Consider the statement

```
num[2]= *num + *ptr;
```

Is it a legal statement? What would be the result?

1. num can be thought to be of type int *, and, ptr is of type int *.
2. So *num is of type int, which is 1 and *ptr is of type int with value 15
3. So num[2] is set to 16.

Let me show you some cool stuff: pointer arithmetic.



num+1 points to **integer box just next to the integer box pointed to by num.** Since arrays were consecutively allocated, the integer box just next to num[0] is num[1].

Okay, What's cool?

So num+1 **points to** num[1]. Similarly, num+2 **points to** num[2], num + 3 **points to** num[3], and so on.

Can you tell me the output of this printf statement?

```
printf("%d %d %d", *(num+1),
              *(num+2),*(num+3));
```

Hmm... Output would be

22 16 -1

Let us predict the output of some simple code fragments.

```
char str[] ="BANTI is a nice girl";
char *ptr = str + 6; /*initialize*/
printf("%s",ptr);
```

What is printed by the above program?

First let us draw the state of memory.

owlie

str[0]					str[5]					str[10]					str[15]
'B'	'A'	'N'	'T'	'I'		'i'	's'	' '	'a'	' '	'n'	'i'	'c'	'e'	' '

str[16]				str[20]
'g'	'i'	'r'	'l'	'\0'

str

ptr

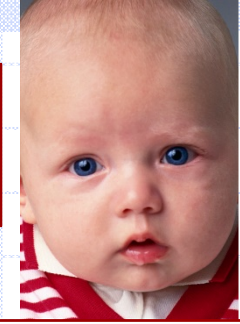
ptr points to str[6]. printf prints the string starting from str[6], which is

Output: is a nice girl

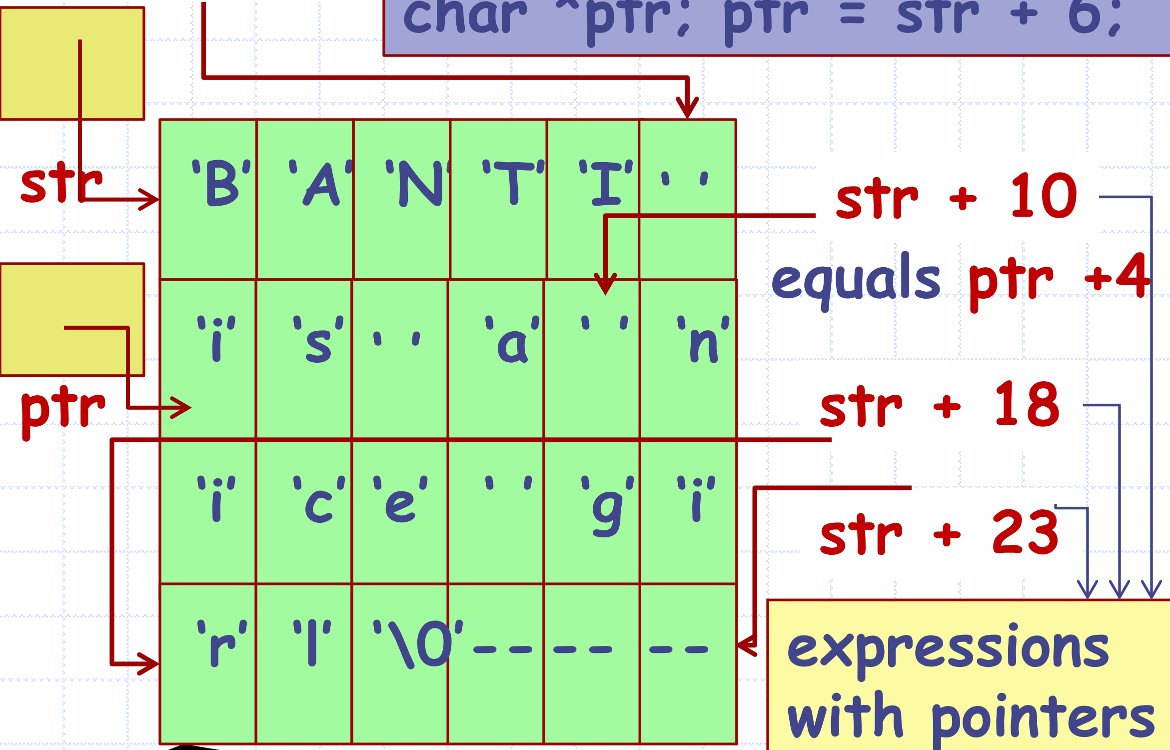
Hmm... OK

OK. Let me understand. The char array str[] was initialized as below.

```
char str[] = "BANTI is a nice girl";
char *ptr; ptr = str + 6;
```



str is of type char *. So str + 6 points to the 6th character from the character pointed to by str. That is ptr. Correct?



expressions with pointers

Here are some other pointer expressions-are they correct?

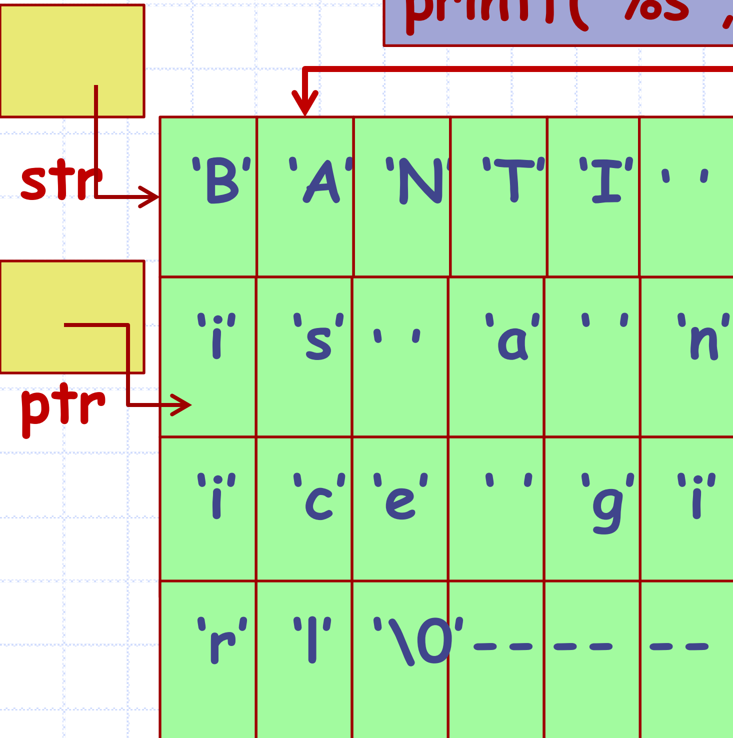
Yes, that's correct

Yes, they're all correct. Can you tell me the output of:

```
printf("%s",ptr-5);
```

Hmm... I'm thinking

```
char str[] = "BANTI is a nice girl";
char *ptr; ptr = str + 6;
printf("%s",ptr-5);
```



ptr -5 should point to the 5th char backwards from the char pointed to by ptr. So ptr-5 points here

The string starting from this point is "ANTI is a nice girl". That would be the output. Correct?

Output	ANTI is a nice girl

Yes, that's correct

Pointers play an important role when used as parameters in function calls.

Let's start with the old example.

OK. How so?

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf(“From main”);
    printf(“a = %d”,a);
    printf(“b=%d\n”,b);
}
```

```
void swap(int a, int b) {
        int t;
        t = a; a=b; b =t;
        printf(“From swap”);
        printf(“a = %d”,a);
        printf(“b= %d\n”,b);
}
```

The swap(int a, int b) function is intended to swap (exchange) the values of a and b.

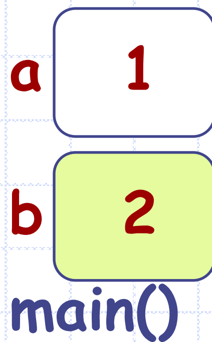
Could you explain again? i'm not so sure

But, if you remember, the value of a and b do not change in main(), although they are swapped in swap().

OK, let's first trace the call to swap

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
        int t;
        t = a; a=b; b =t;
        printf("From swap ");
        printf("a= %d",a);
        printf("b= %d\n",b);
}
```



STACK

1

t

Output:

From swap a= 2 b= 1

Now swap() returns:

1. Return address is line 3 of main(). Program counter is set to this location.
2. Stack for swap() is deleted.

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
        int t;
        t = a; a=b; b =t;
        printf("From swap ");
        printf("a = %d",a);
        printf("b = %d\n",b);
}
```

STACK

a	1
b	2

main()

Output: From swap a = 2 b = 1

Returning back to main(),we resume execution from line 3.

Hmm…

But the variables a and b of main() are unchanged from what they were before the call to swap(). They are printed as is.

Changes made by swap() remained local to the variables of swap(). They did not propagate back to main().

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
        int t;
        t = a; a=b; b =t;
        printf("From swap ");
        printf("a = %d",a);
        printf("b = %d\n",b);
}
```

a	1
b	2

Output:

From swap a = 2 b = 1

From main a = 1 b = 2

OK, i remember now

1. Passing int/float/char as parameters does not allow passing "back" to calling function.
2. Any changes made to these variables are lost once the function returns.

Pointers will help us solve this problem!

Here is the changed program.

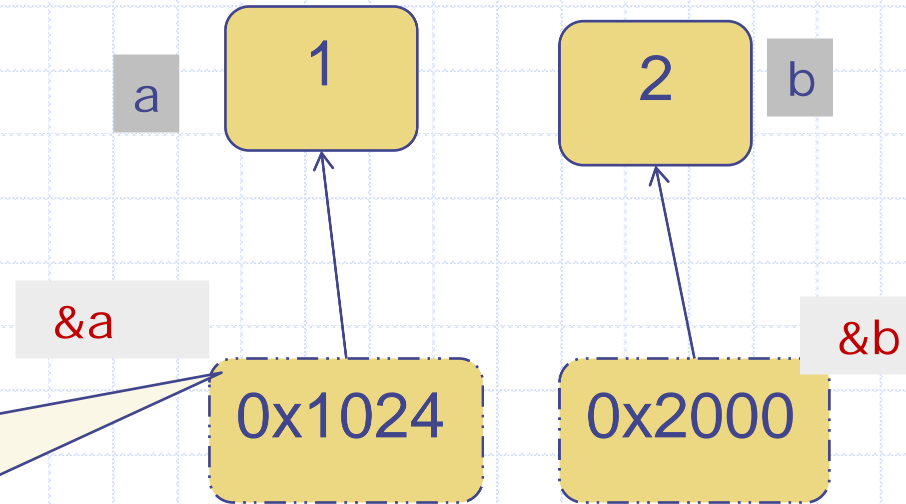
```
void
swap(int *ptra, int *ptrb)
{
     int t;
     t = *ptra;
     *ptra= *ptrb;
     *ptrb =t;
}
```

```
int main() {
   int a = 1, b = 2;
   swap(&a, &b);
   printf("a=%d, b=%d",
            a, b);
   return 0;
}
```

1. The function swap() uses pointer to integer arguments, int *ptra and int *ptrb.
2. The main() function calls swap(&a,&b), i.e., passes the addresses of the ints it wishes to swap.

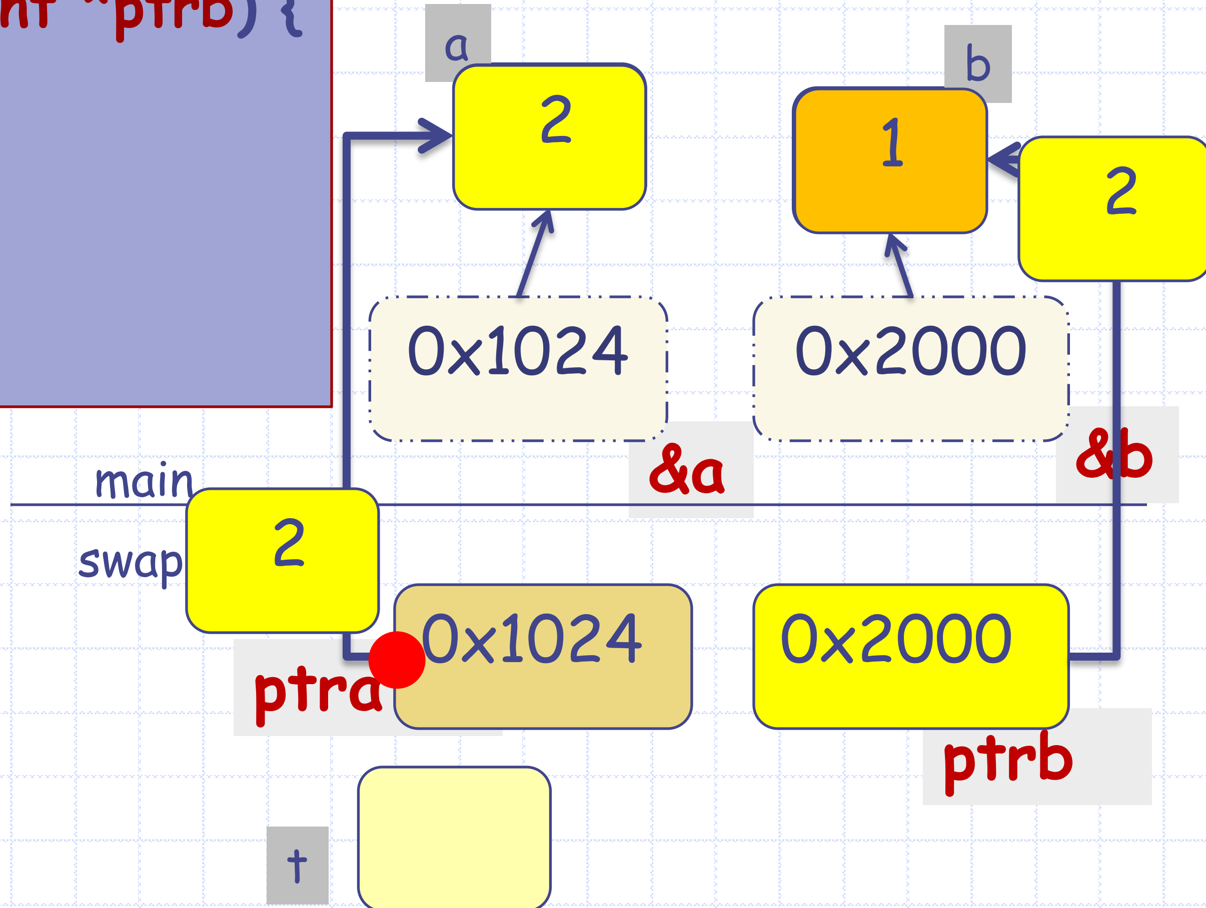
Tracing the swap function

```
int main() {
    int a = 1, b = 2;
    swap(&a, &b);
}
```



Address of a. (a is situated at memory location 0x1024)

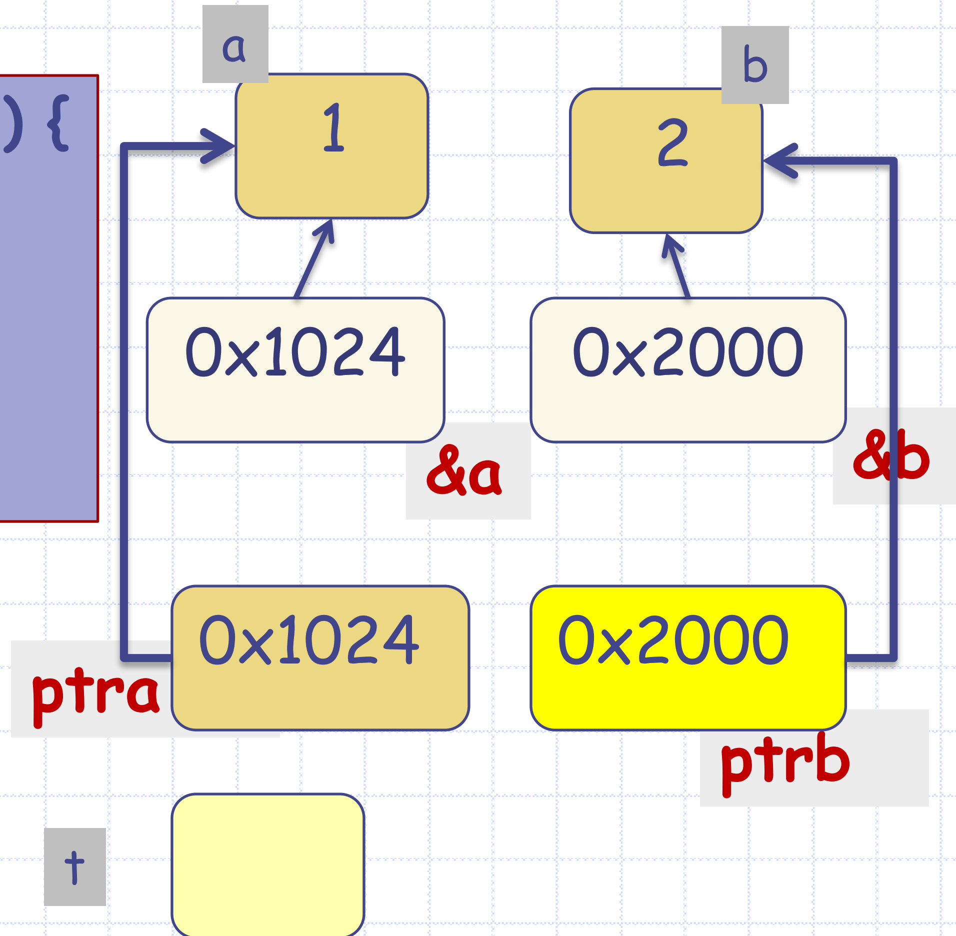
```
void swap(int *ptra, int *ptrb) {
   int t;
   t = *ptra;
  *ptra= *ptrb;
  *ptrb =t;
}
```



Homework ☺

Will the following code perform swap correctly?

```
void swap(int *ptra, int *ptrb) {
      int *ptrt;
      ptrt = ptra;
      ptra= ptrb;
      ptrb =ptrt;
}
```



Memory: Recap

Give examples please

4 ... starting
4012

...ng the context it is
... deter...
... numbe...
...an inte...

"Type" helps us disambiguate.

✓It could be the "location" of the block that stores 'E'

How do we decide what it is?

Address	Value
1004000	'A'
1004001	'E'
1004002	'I'
1004003	'O'
1004004	'U'
1004009	1024
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

Simplified View of Memory

- In programming also, "Type" helps us decide whether 1004001 is an integer or a pointer to block containing 'E' (or something else)

```
#include<stdio.h>
int main() {
        char x[5] = {'A', 'E', 'I', 'O', 'U'};
        int y = 1024;
        char *p = x+1;
        ...
}
```

```
#include<stdio.h>
int main() {
        char x[5] = {'A', 'E', 'I', 'O', 'U'};
        int y = 1024;
        int p = 1004001;
        ...
}
```

Declaration of a pointer to char box