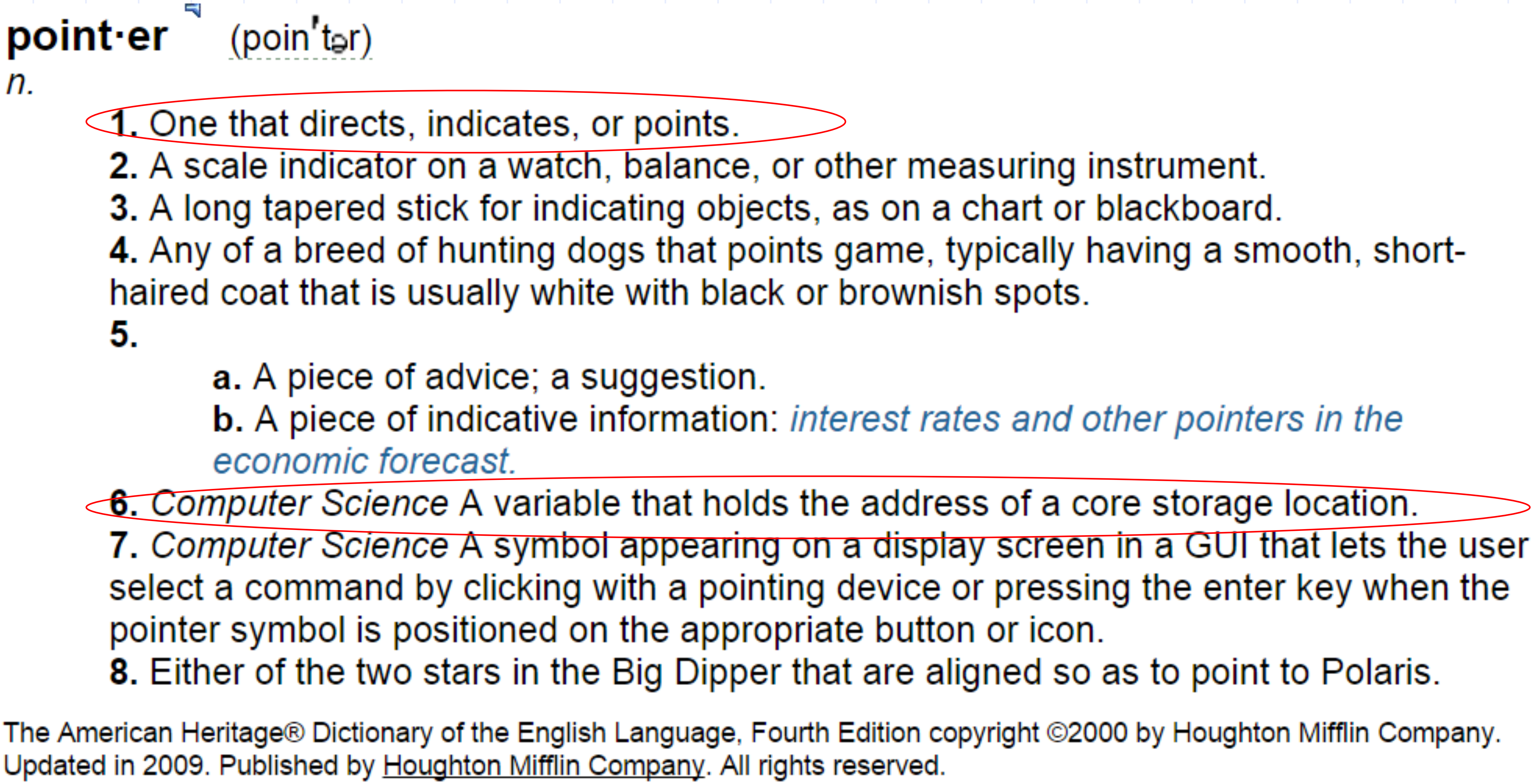
ESC101: Introduction to Computing

Pointers

Pointer: Dictionary Definition



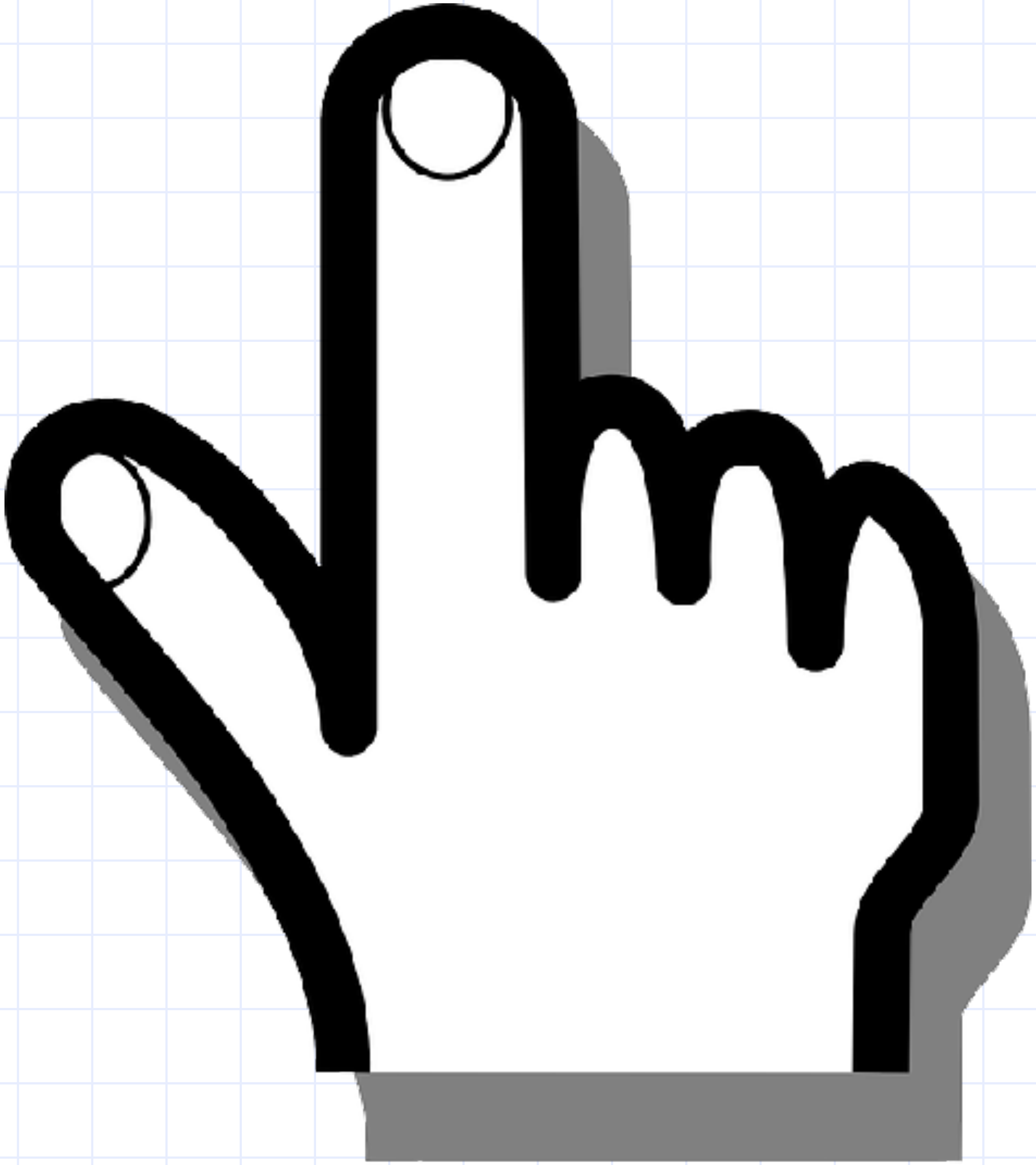
point·er (poin′tər)

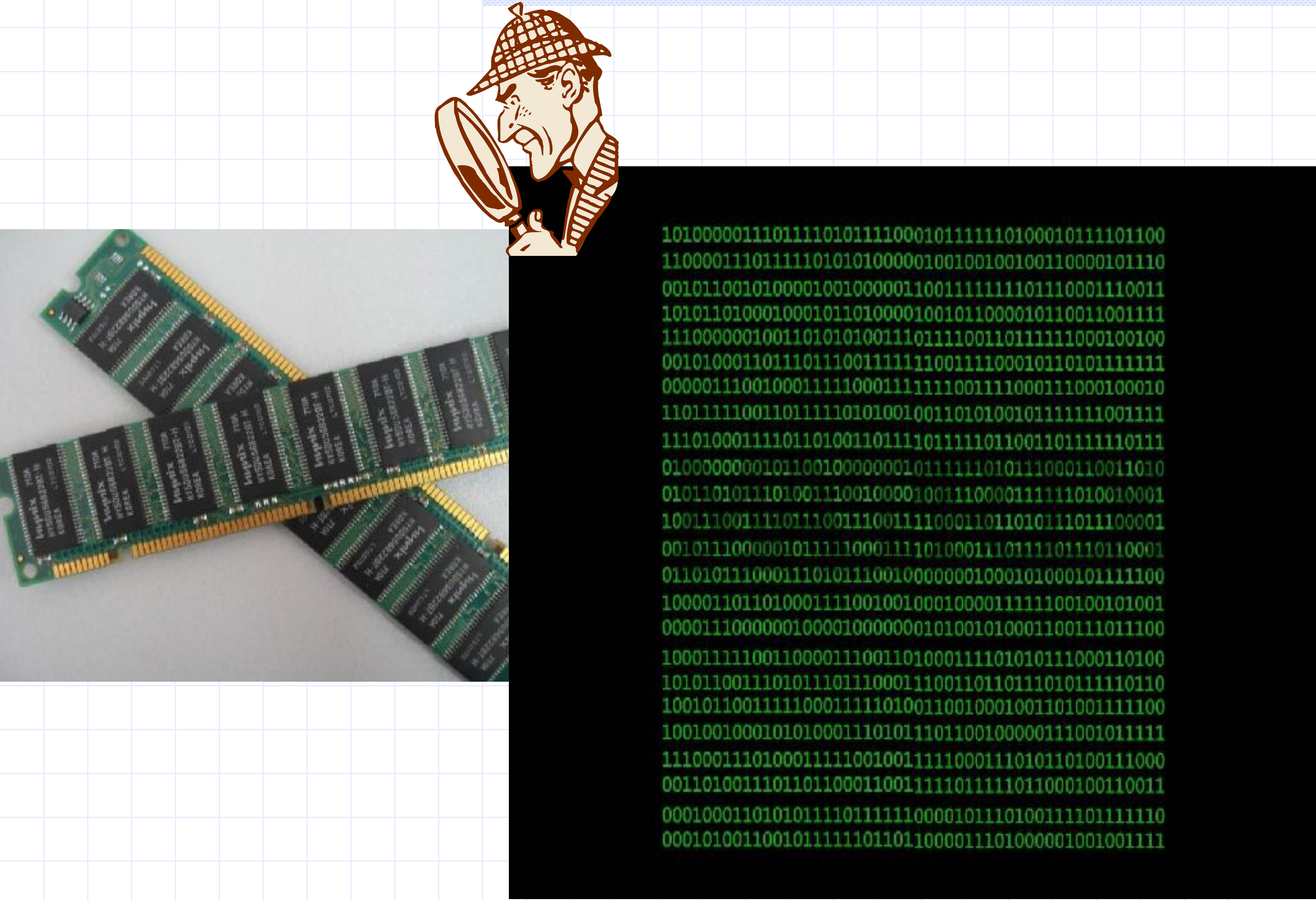
n.

1. One that directs, indicates, or points.
2. A scale indicator on a watch, balance, or other measuring instrument.
3. A long tapered stick for indicating objects, as on a chart or blackboard.
4. Any of a breed of hunting dogs that points game, typically having a smooth, short-haired coat that is usually white with black or brownish spots.
5.
a. A piece of advice; a suggestion.
b. A piece of indicative information: *interest rates and other pointers in the economic forecast.*
6. *Computer Science* A variable that holds the address of a core storage location.
7. *Computer Science* A symbol appearing on a display screen in a GUI that lets the user select a command by clicking with a pointing device or pressing the enter key when the pointer symbol is positioned on the appropriate button or icon.
8. Either of the two stars in the Big Dipper that are aligned so as to point to Polaris.

The American Heritage® Dictionary of the English Language, Fourth Edition copyright ©2000 by Houghton Mifflin Company. Updated in 2009. Published by Houghton Mifflin Company. All rights reserved.

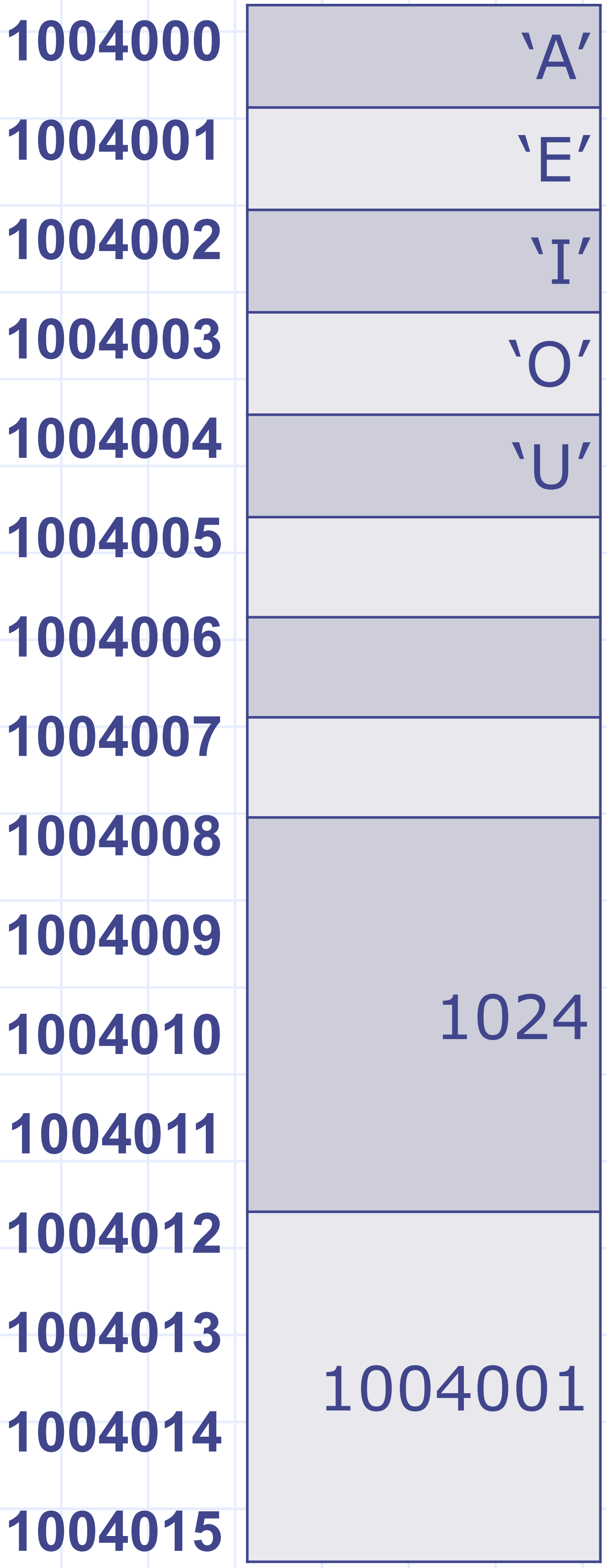
Pointer we are all born with





Simplified View of Memory

- “Array” of blocks
- Each block can hold a byte (8-bits)
- “char” stored in 1 block
- “int” (32-bit) stored in 4 consecutive blocks
- Finite number of blocks
  - Limited by the capacity of (Virtual) Memory
  - Blocks are addressable – [0… 2^N-1]



1004000	‘A’
1004001	‘E’
1004002	‘I’
1004003	‘O’
1004004	‘U’
1004005	
1004006	
1004007	
1004008	1024
1004009	
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

Simplified View of Memory

- Blocks are addressable.
- Address range: [0…2^N-1]
- N is the number of bits in address (number of digits in binary world)
- Any integer in the above range
  - Can be used as an index in the MEMORY ARRAY
- Since memory array is unique, we can use this index alone
  - If context is clear

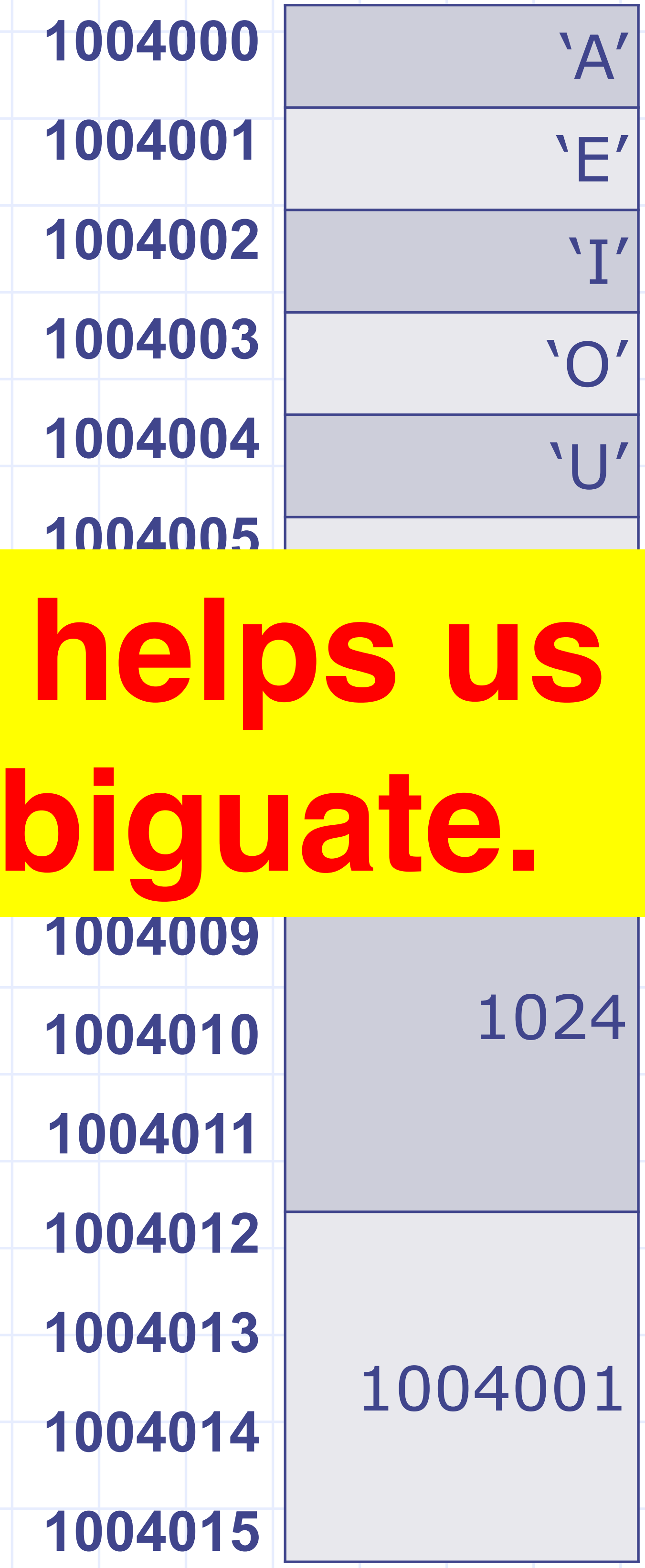
1004000	‘A’
1004001	‘E’
1004002	‘I’
1004003	‘O’
1004004	‘U’
1004005	
1004006	
1004007	
1004008	1024
1004009	
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

Simplified View of Memory

- Content of the 4-blocks starting at address 1004012
  - ✓1004001
- Without knowing the context it is not possible to determine t significance of number
  - ✓It could be an integer value 1004001
  - ✓It could be the "location" of the block that stores 'E'

"Type" helps us disambiguate.

How do we decide what it is?



What is a Pointer

- **Pointer**: A **special type** of variable that contains an address of a memory location.
- Think of a pointer as a new data type (a new kind of box) that holds memory addresses.
- Pointers are almost always associated with the type of data that is contained in the memory location.
  - For example, an integer pointer is a memory location that contains an integer.
  - Character pointer, float pointer
  - Even pointer to pointer (more on this later …)

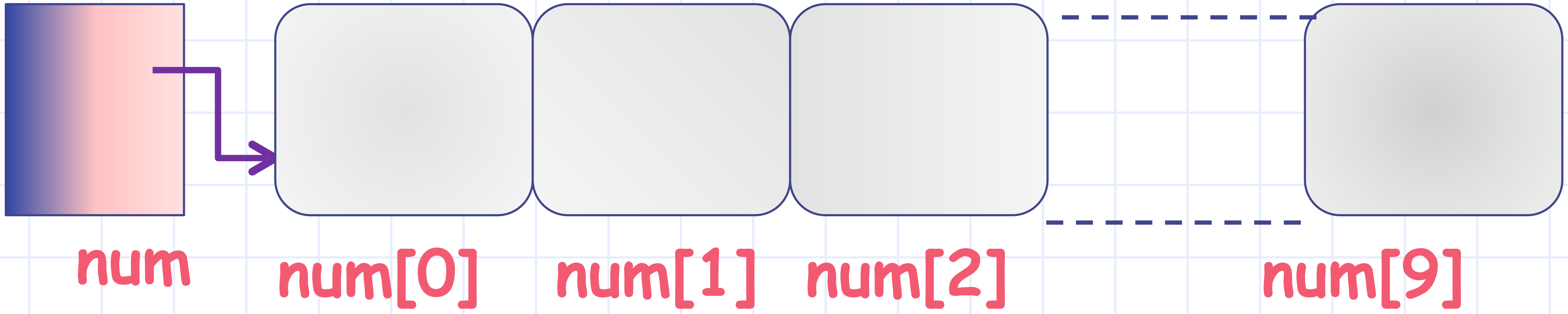
Remember Arrays?

The memory allocated to array has two components:

A consecutively allocated segment of memory boxes of the same type, and

A box with the same name as the array. This box holds the address of the base (i.e., first) element of the array.

int num[10];



This definition for num[10] gives 10 of type int and 1 of type address of an int box.

1. We represent the **address of a box x** by an **arrow to the box x**. So **addresses** are referred to as **pointers**.
2. The contents of an address box is a pointer to the box whose address it contains. e.g., num points to num[0] above.

What can we do with a box? e.g., an integer box?

```
int num[10];
```

But what is the type of **ptr**? And how do i define **ptr**?

ptr would be of type **address of int**. In C this type is **int ***.

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

We can do operations that are supported for the data type of the box.

For integers, we can do + - * / % etc. for each of num[0] through num[9].

We can also take the address of a box.We do this when we use scanf for reading using the & operator.

Suppose I want to take the address of num[1] and store it in an address variable ptr.

```
ptr = &num[1];
```

To see the meaning of ptr=&num[1], let's look at the memory state.

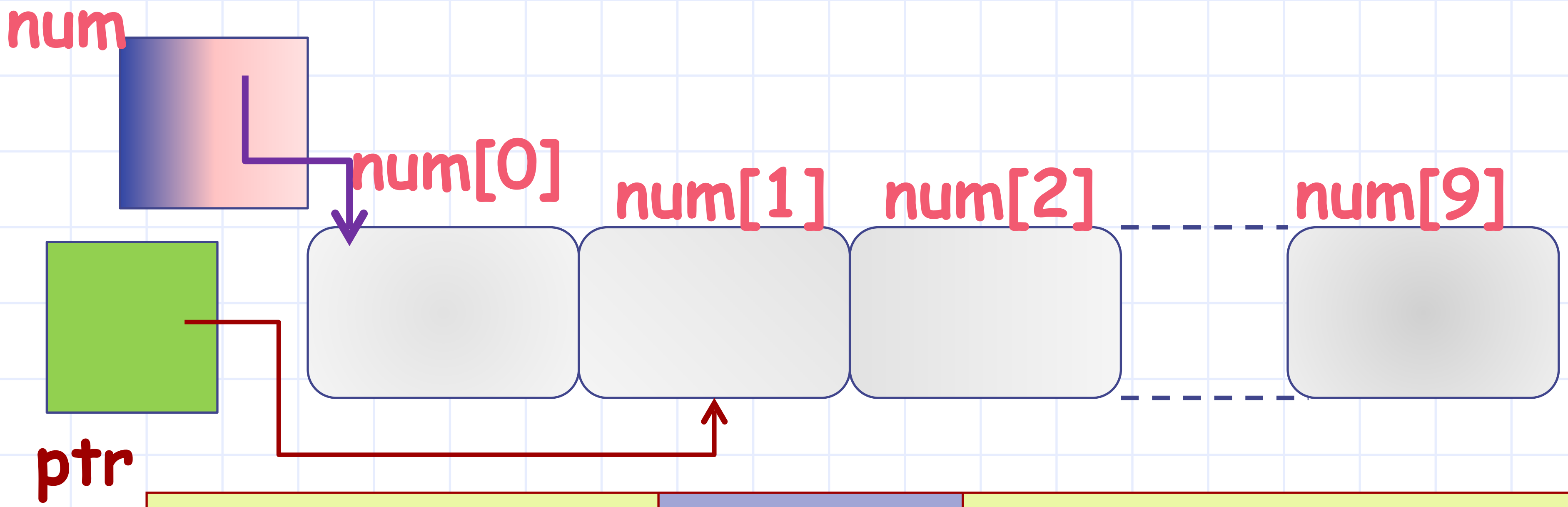
Here is the state after int num[10] gets defined.

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

OK, ptr is of type pointer to integer. But what does

```
ptr = &num[1];
```

mean?

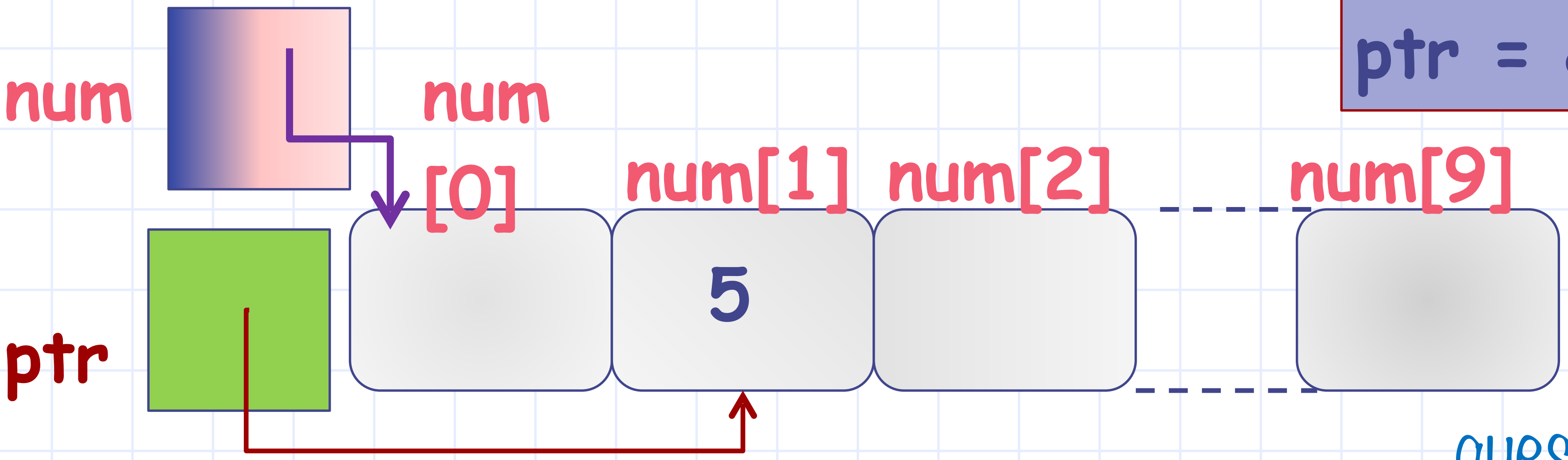


The statement int *ptr; creates a new box of type "**address of an int box**", more commonly referred to as, of type "**pointer to integer**".

The statement ptr = &num[1]; assigns to ptr the address of the box num[1]. Commonly referred to as: **ptr now points to num[1].**

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

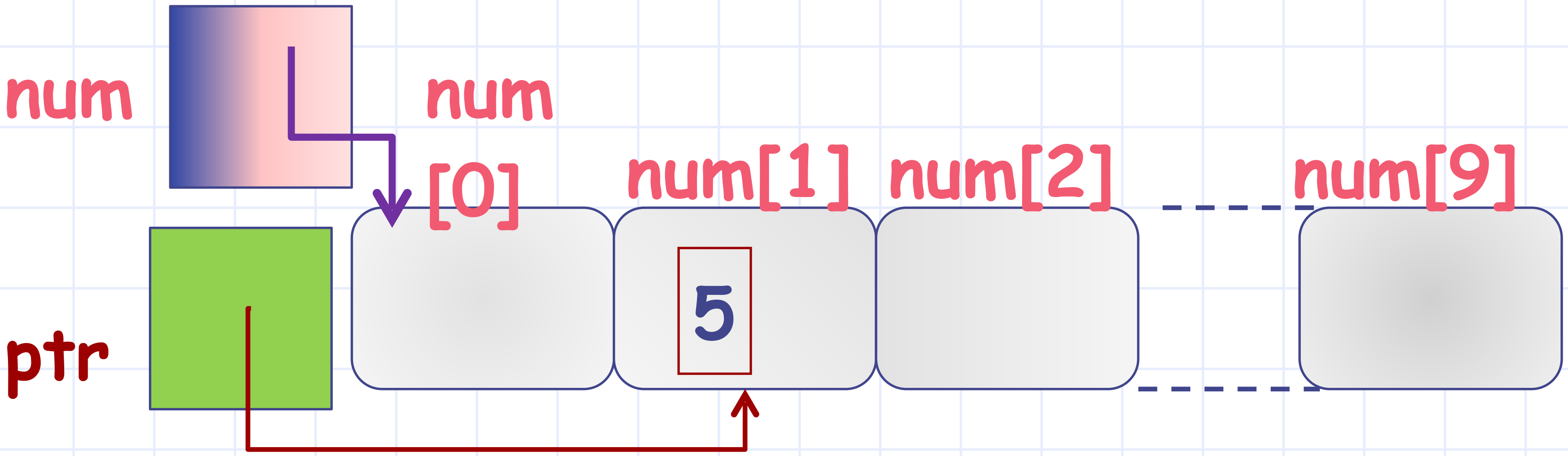
```
scanf("%d",ptr);
```

and input is :

Input

5

Does num[1] become 5?

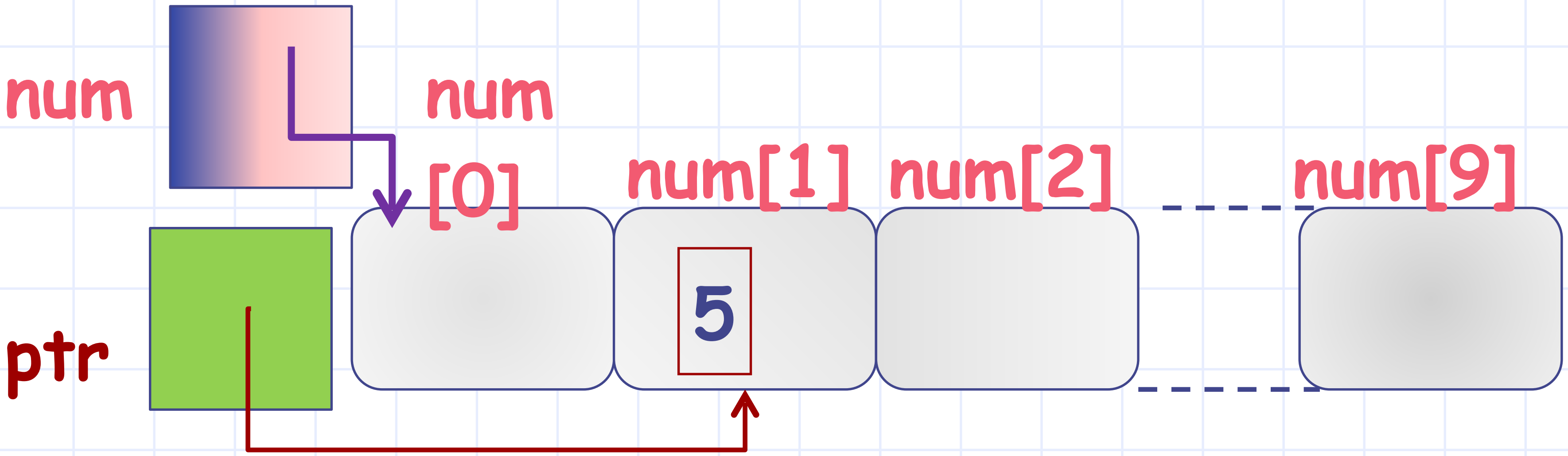


num is of type int [] (i.e., array of int). In C the box num stores the pointer to num[0]. Internally, C represents num and ptr in the same way. So the type **int** * can be used wherever int[] was used.

Well, what else can you do with a

You can

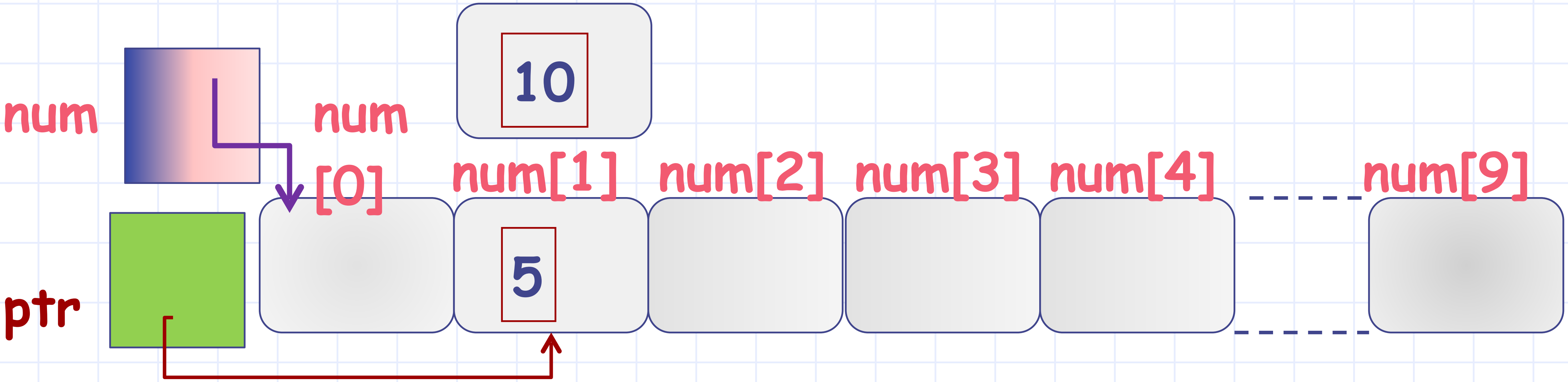
1. de-reference the pointer.
2. do simple arithmetic + - with pointers.
3. compare pointers and test for ==, <, > etc., similar to ordinary integers.



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.

How to output the address stored in a pointer

You can print the pointer using the following command printf(“%p”,ptr)



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

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

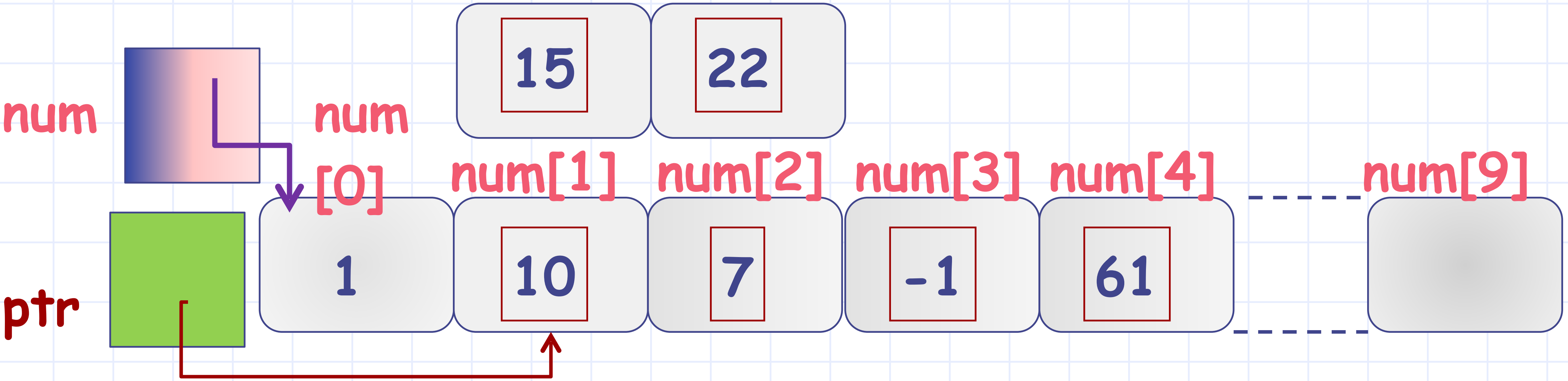
Output 5

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

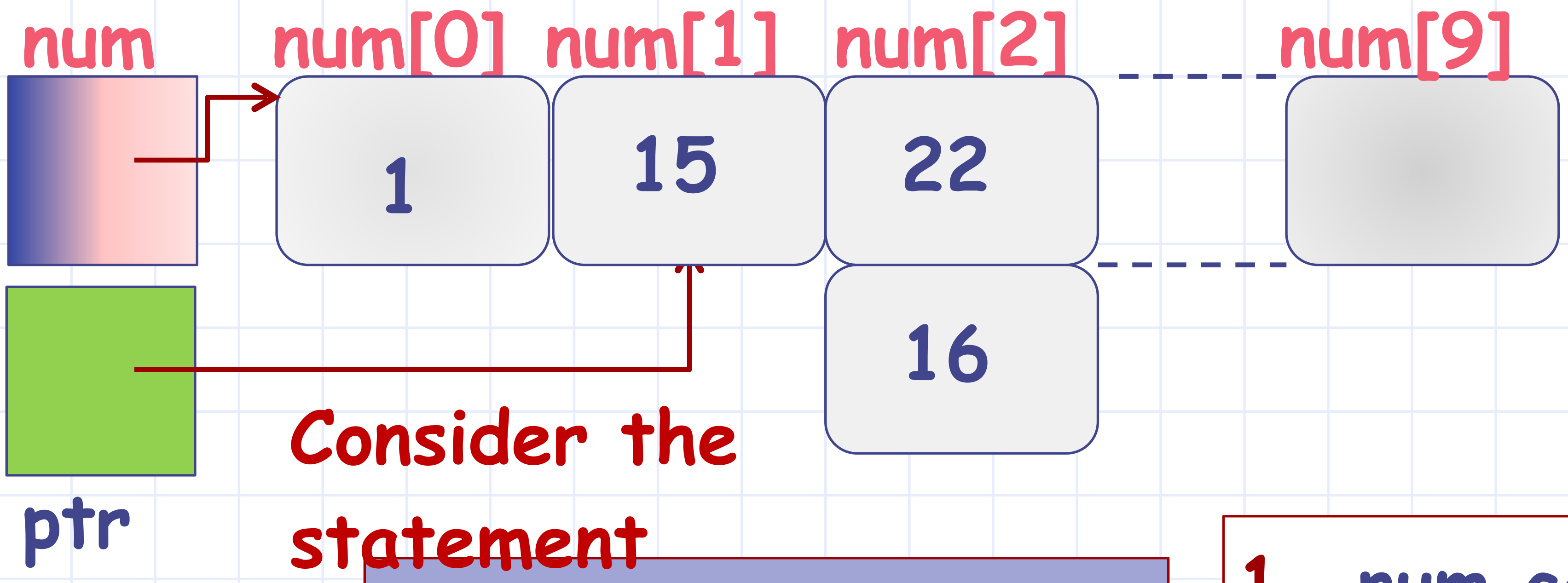


De-referencing a pointer **ptr** gives the box pointed to by **ptr**. The de-referencing operator in C is *.

Consider the statements. Execute them on above memory state.

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

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].



Consider the statement

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

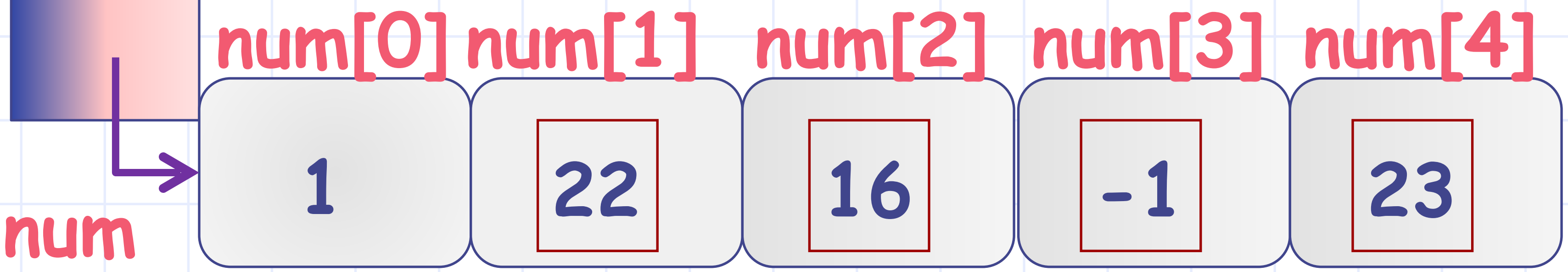
Is it a legal statement? What would be the result?

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.

Pointer Arithmetic

Let

```
int num[] = {1,22,16,-1,23};
```



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].

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));
```

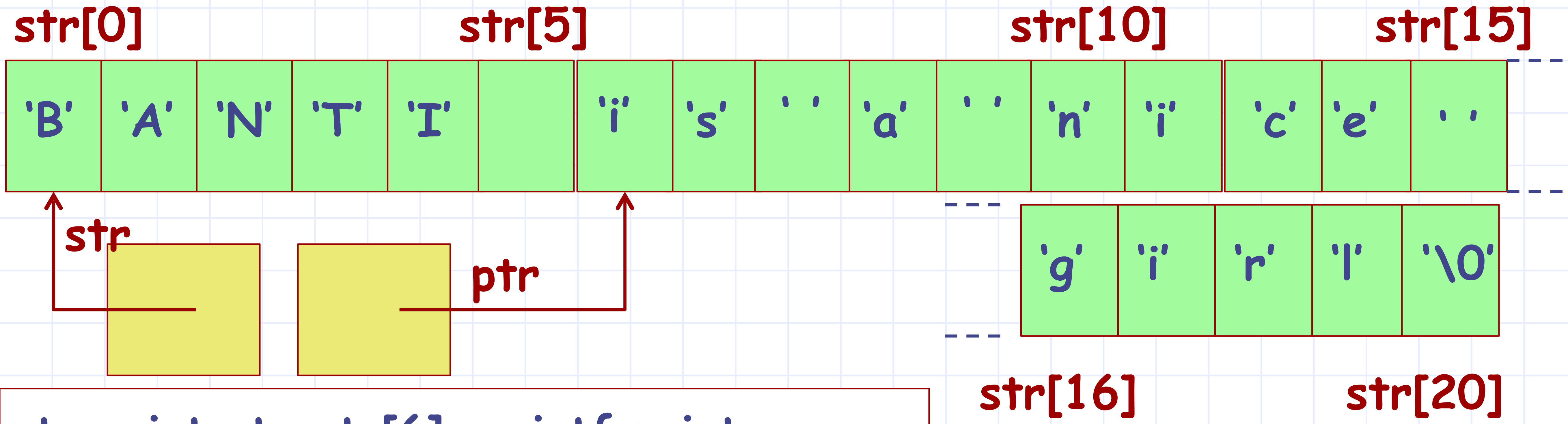
22 16 -1

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.



str[0] str[5] str[10] str[15]

‘B’ ‘A’ ‘N’ ‘T’ ‘I’ ‘i’ ‘s’ ‘ ’ ‘a’ ‘ ’ ‘n’ ‘i’ ‘c’ ‘e’ ‘ ’

‘g’ ‘i’ ‘r’ ‘l’ ‘\0’

str[16] str[20]

str ptr

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

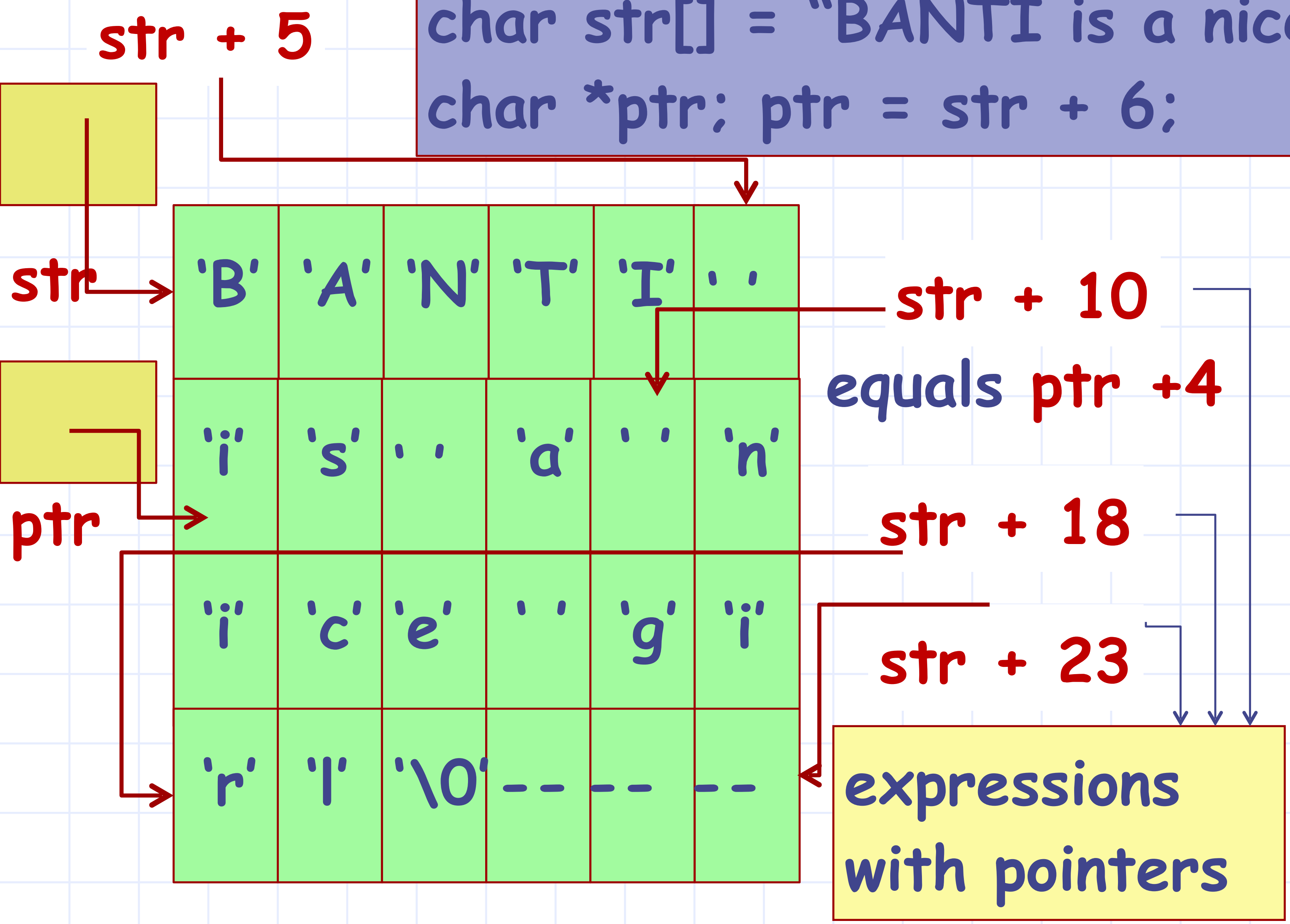
Output is a nice girl

The char array str[] is 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.

Here are some other pointer expressions-are they correct?

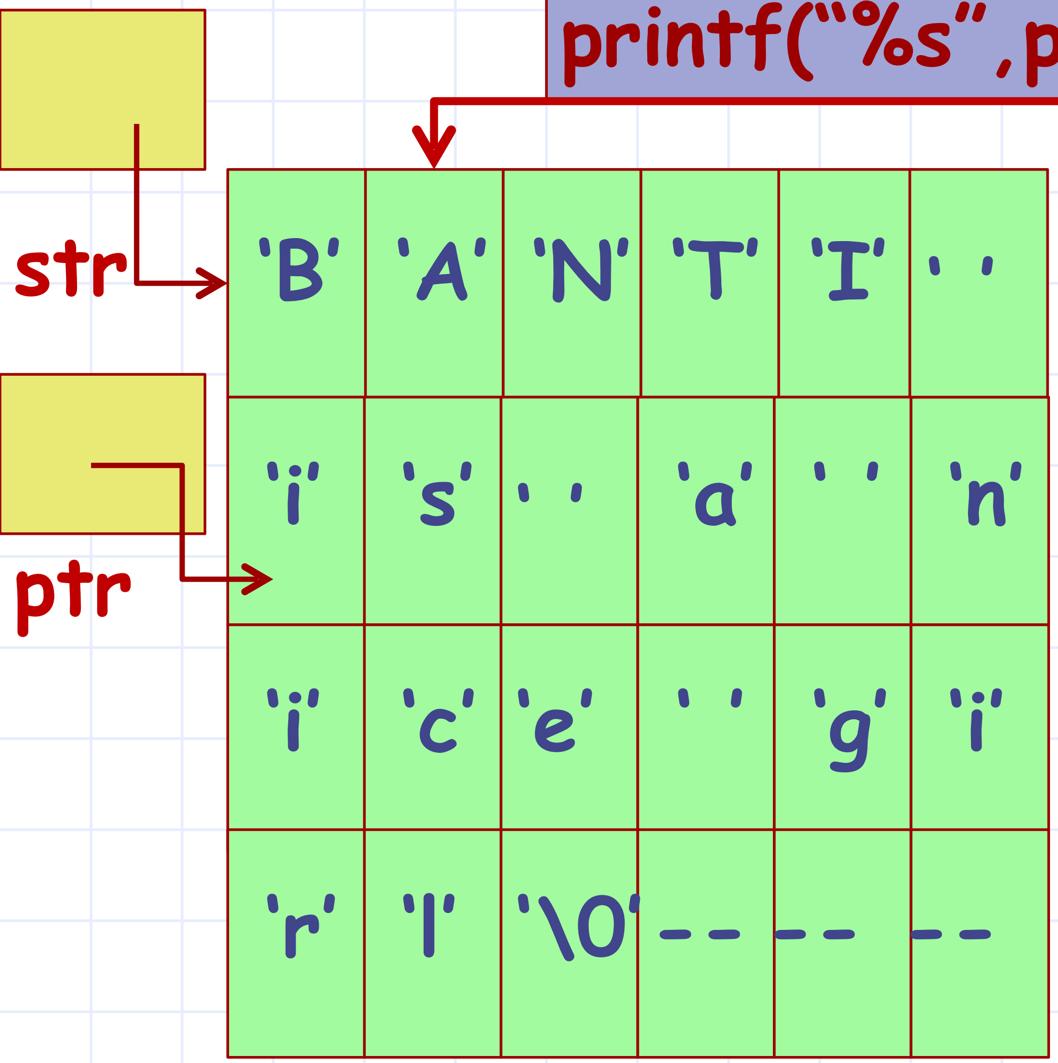


‘B’	‘A’	‘N’	‘T’	‘I’	‘ ’
‘i’	‘s’	‘ ’	‘a’	‘ ’	‘n’
‘i’	‘c’	‘e’	‘ ’	‘g’	‘i’
‘r’	‘l’	‘\0’	--	--	--

Yes, they’re all correct.

What is the output of: `printf(“%s”,ptr-5);`

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



'B'	'A'	'N'	'T'	'I'	' '
'i'	's'	' '	'a'	' '	'n'
'i'	'c'	'e'	' '	'g'	'i'
'r'	'l'	'\0'	--	--	--

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.

Output ANTI is a nice girl

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

Let’s start with the old example.

```
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);
}
```

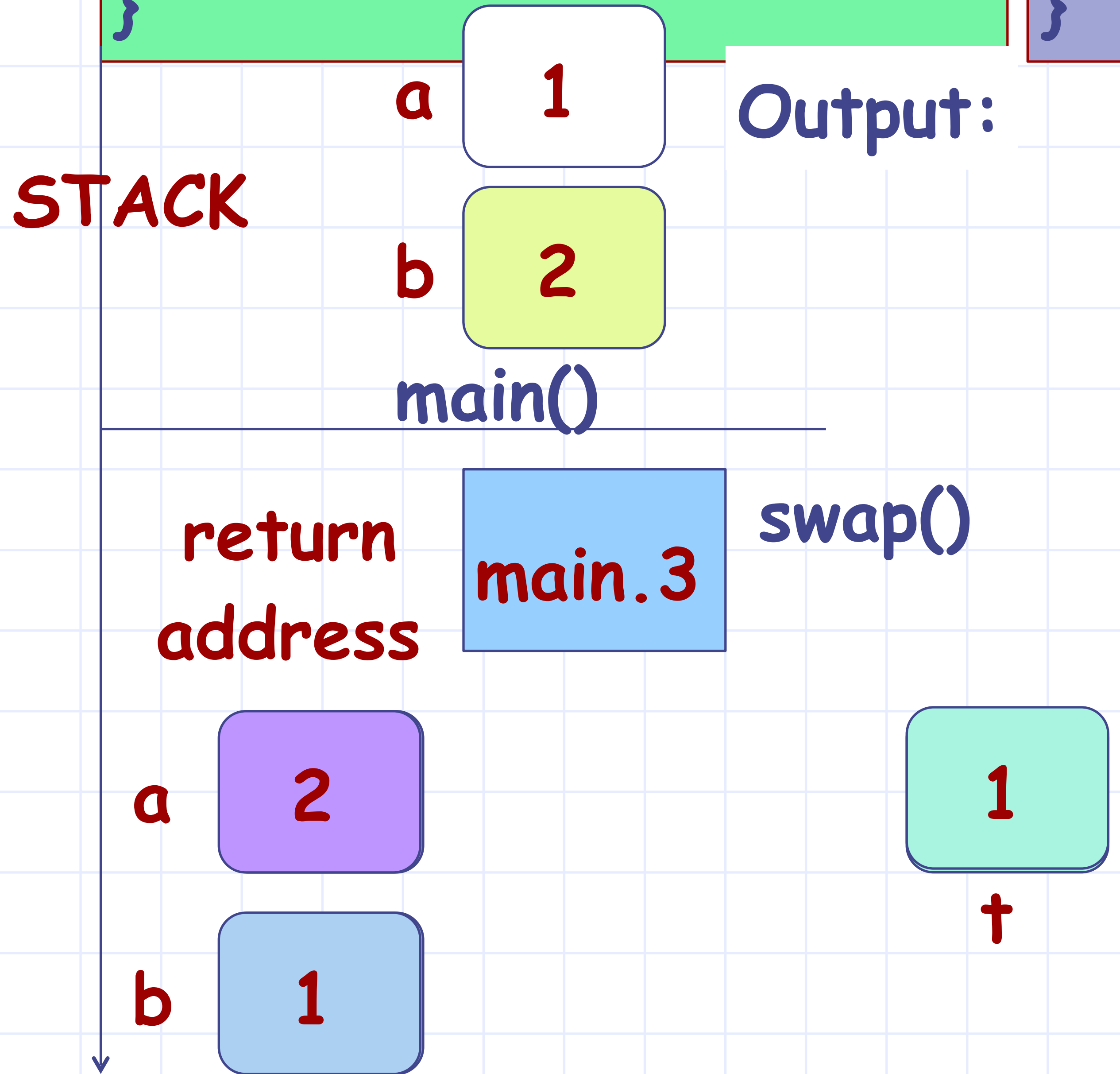
The swap(int a, int b) function is intended to swap (exchange) the values of a and b.

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);
}
```



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);
}
```

Output:

```
From swap a = 2 b = 1
```

TACK

a 1

b 2

main()

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

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
```

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!

```
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
```

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.

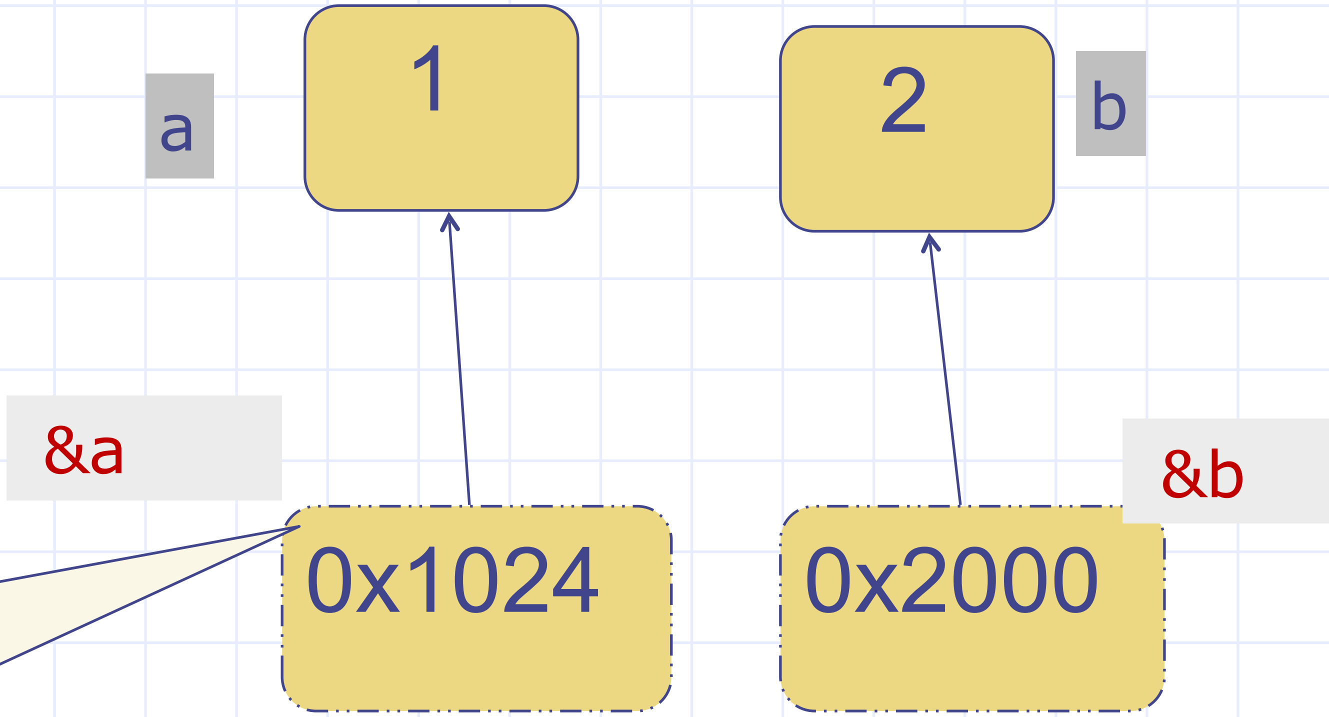
```
void
swap(int *ptra, int *ptrb)
{
    int t;
    t = *ptra;
    *ptra= *ptrb;
    *ptrb =t;
}
```

```
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

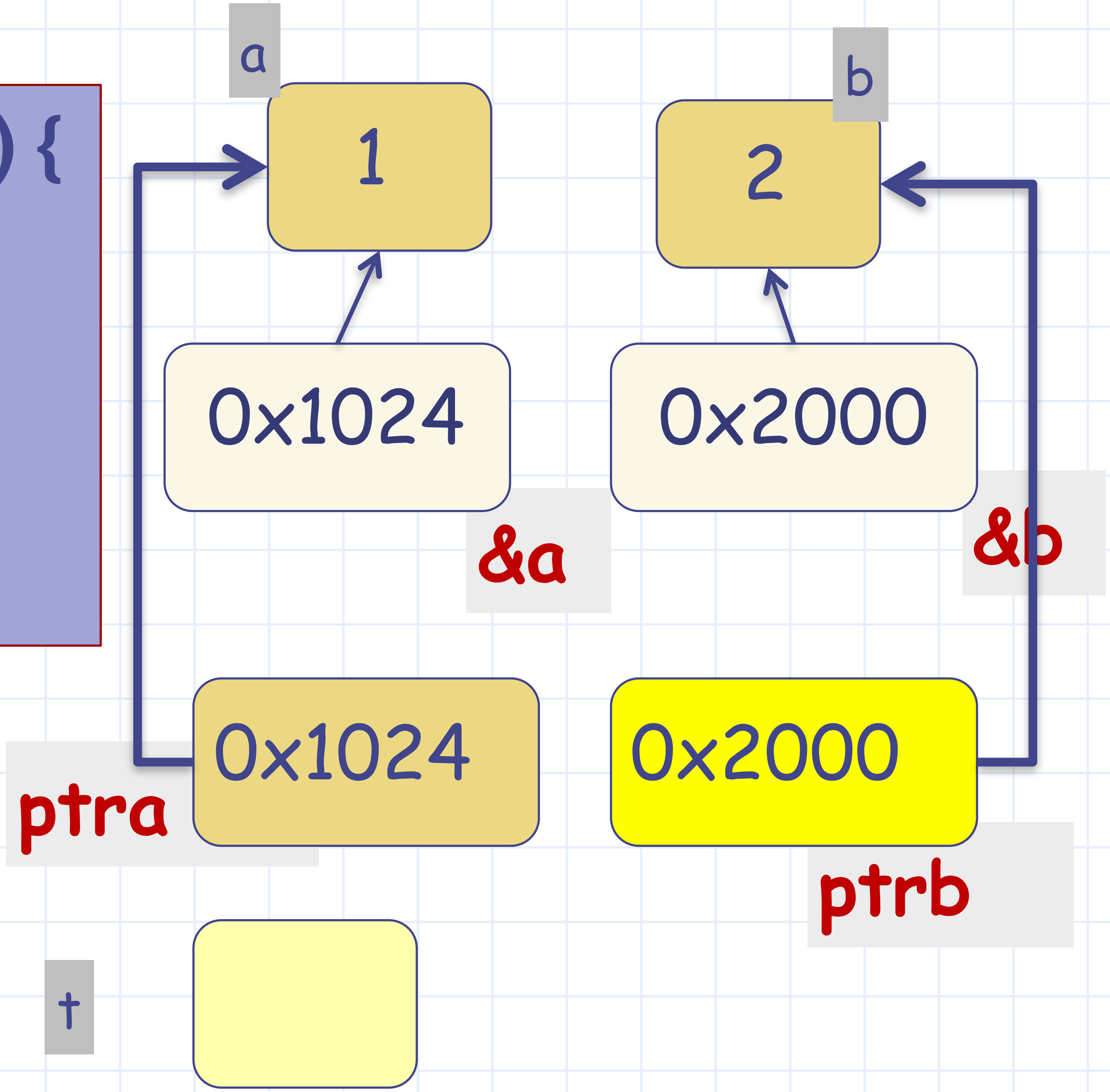
```
int main() {
    int a = 1, b = 2;
    swap(&a, &b);
}
```



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

Question: Will the following code perform swap correctly?

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



What is the output of following code?

```c
#include <stdio.h>

int foo(char *parr)
{
        int cnt=0;
        while(*parr!='\0')
        {
                printf("%s\n",parr);
                parr++;
                cnt++;
        }
        return cnt;
}
int main()
{
        char arr[]="text";
        char *parr = arr;
        printf("%d\n",foo(parr) );
        return 0;
}
```

What is the output of following code?

```
#include <stdio.h>

int foo(char *parr)
{
        int cnt=0;
        while(*parr!='\0')
        {
                printf("%s\n",parr);
                parr++;
                cnt++;
        }
        return cnt;
}
int main()
{
        char arr[]="text";
        char *parr = arr;
        printf("%d\n",foo(parr) );
        return 0;
}
```

Output is:

```
text
ext
xt
t
4
```