Pointers

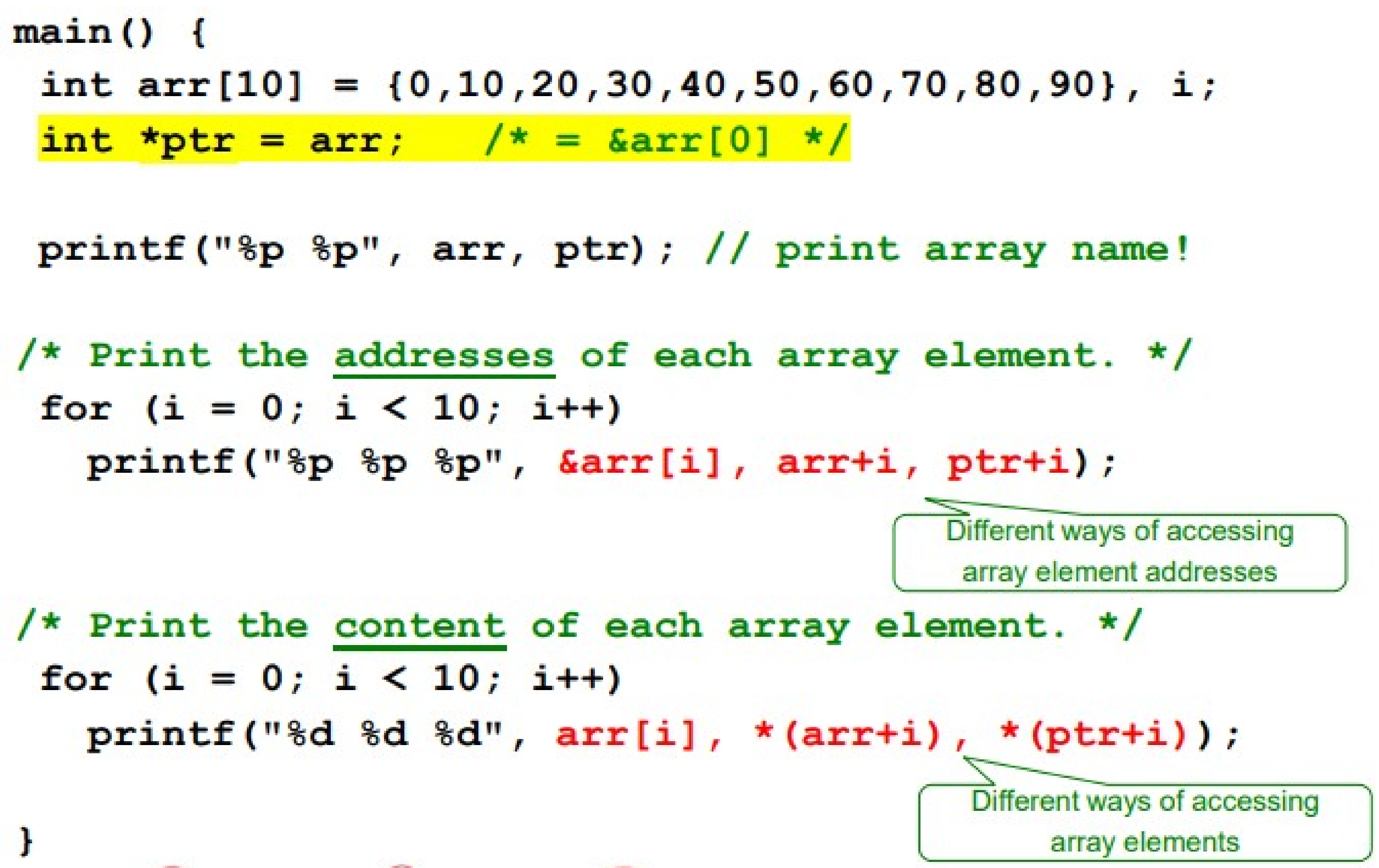
EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

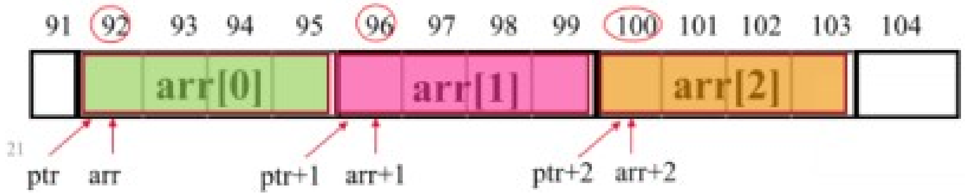
Use of pointer arithmetic in array

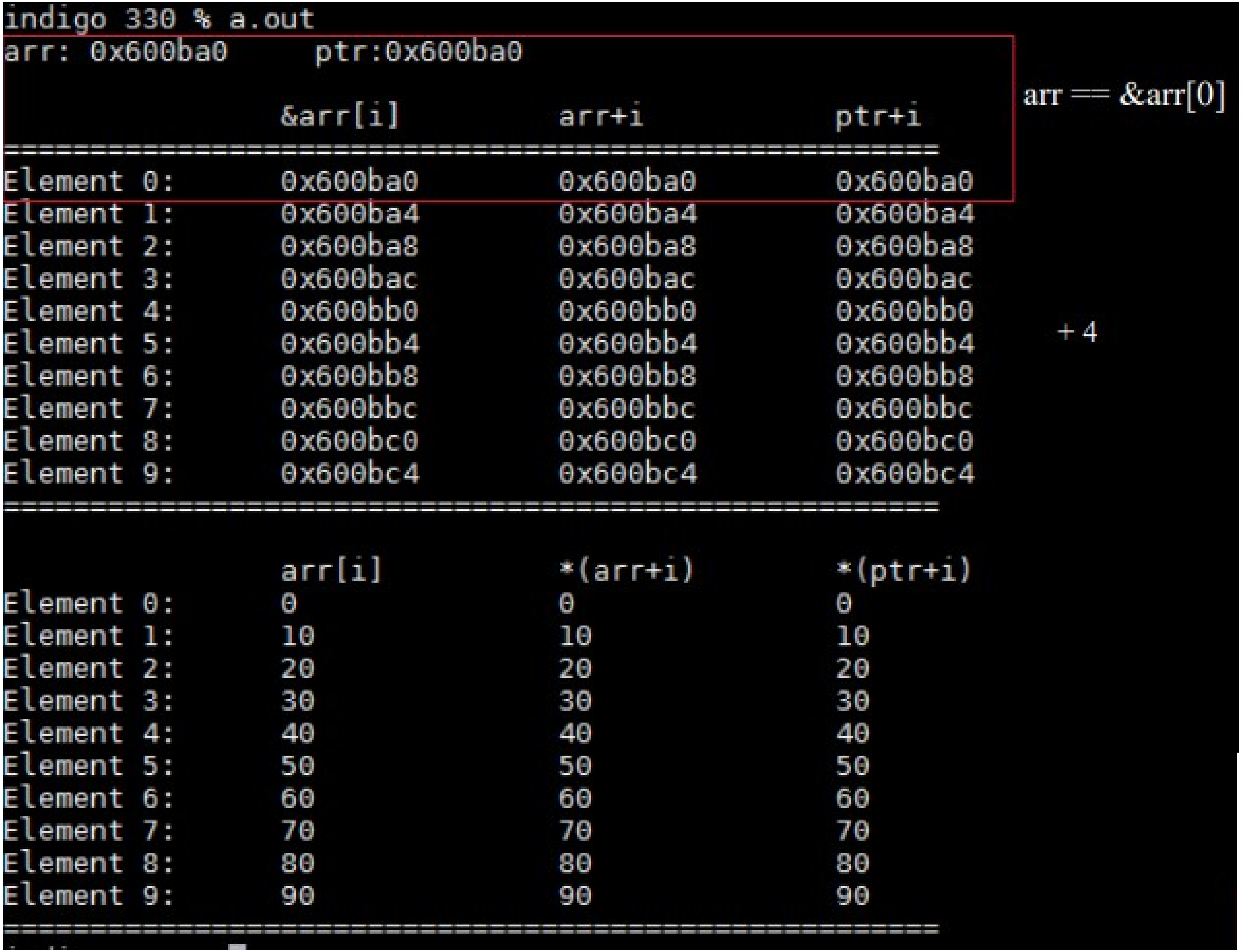


```
main() {
 int arr[10] = {0,10,20,30,40,50,60,70,80,90}, i;
 int *ptr = arr;    /* = &arr[0] */

 printf("%p %p", arr, ptr); // print array name!

/* Print the addresses of each array element. */
 for (i = 0; i < 10; i++)
   printf("%p %p %p", &arr[i], arr+i, ptr+i);

/* Print the content of each array element. */
 for (i = 0; i < 10; i++)
   printf("%d %d %d", arr[i], *(arr+i), *(ptr+i));

}
```

Different ways of accessing array element addresses

Different ways of accessing array elements



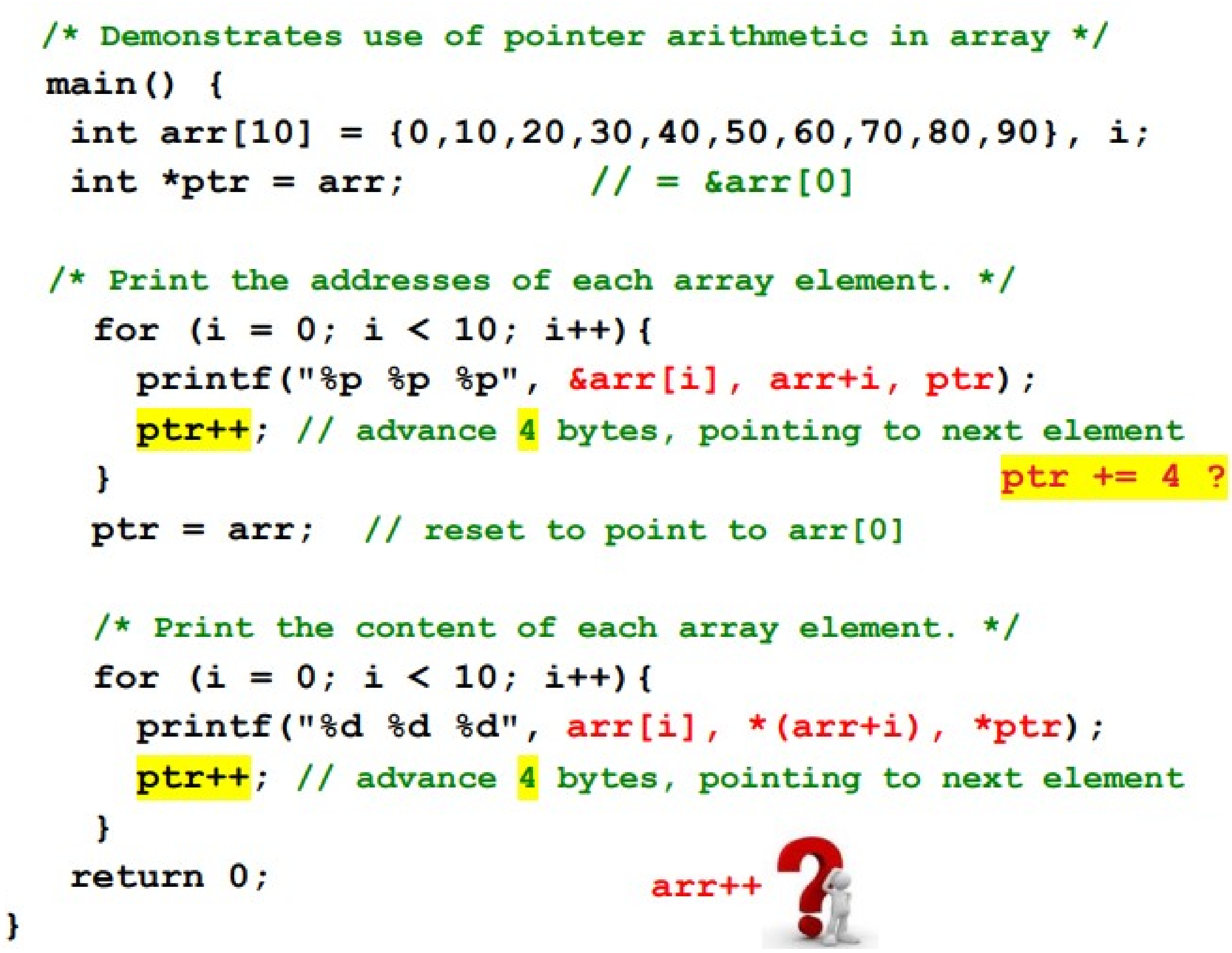


```
indigo 330 % a.out
arr: 0x600ba0     ptr:0x600ba0

                &arr[i]          arr+i            ptr+i
=====================================================
Element 0:      0x600ba0         0x600ba0         0x600ba0
Element 1:      0x600ba4         0x600ba4         0x600ba4
Element 2:      0x600ba8         0x600ba8         0x600ba8
Element 3:      0x600bac         0x600bac         0x600bac
Element 4:      0x600bb0         0x600bb0         0x600bb0
Element 5:      0x600bb4         0x600bb4         0x600bb4
Element 6:      0x600bb8         0x600bb8         0x600bb8
Element 7:      0x600bbc         0x600bbc         0x600bbc
Element 8:      0x600bc0         0x600bc0         0x600bc0
Element 9:      0x600bc4         0x600bc4         0x600bc4
=====================================================

                arr[i]           *(arr+i)         *(ptr+i)
Element 0:      0                0                0
Element 1:      10               10               10
Element 2:      20               20               20
Element 3:      30               30               30
Element 4:      40               40               40
Element 5:      50               50               50
Element 6:      60               60               60
Element 7:      70               70               70
Element 8:      80               80               80
Element 9:      90               90               90
=====================================================
```

arr == &arr[0]

+ 4

Another way ++



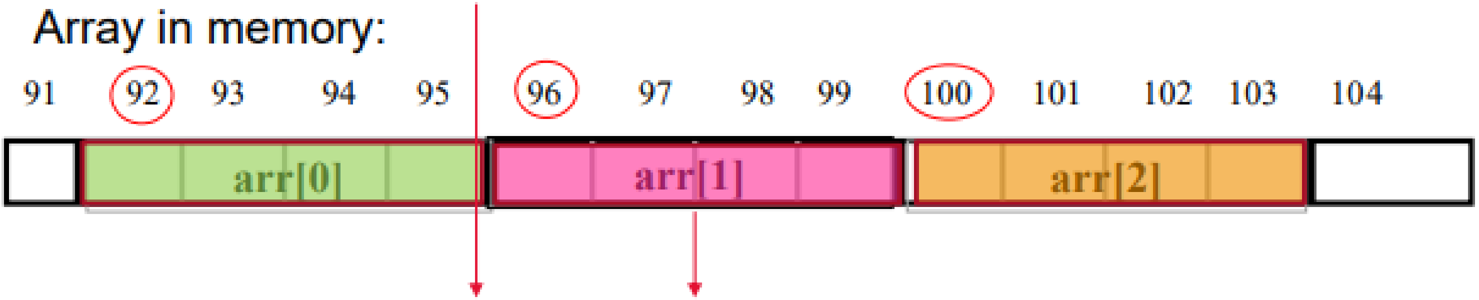
```
/* Demonstrates use of pointer arithmetic in array */
main() {
 int arr[10] = {0,10,20,30,40,50,60,70,80,90}, i;
 int *ptr = arr;          // = &arr[0]

/* Print the addresses of each array element. */
   for (i = 0; i < 10; i++){
     printf("%p %p %p", &arr[i], arr+i, ptr);
     ptr++; // advance 4 bytes, pointing to next element
   }
   ptr = arr;  // reset to point to arr[0]

   /* Print the content of each array element. */
   for (i = 0; i < 10; i++){
     printf("%d %d %d", arr[i], *(arr+i), *ptr);
     ptr++; // advance 4 bytes, pointing to next element
   }
 return 0;
}
```

ptr += 4 ?

arr++

Pointer arithmetic summary

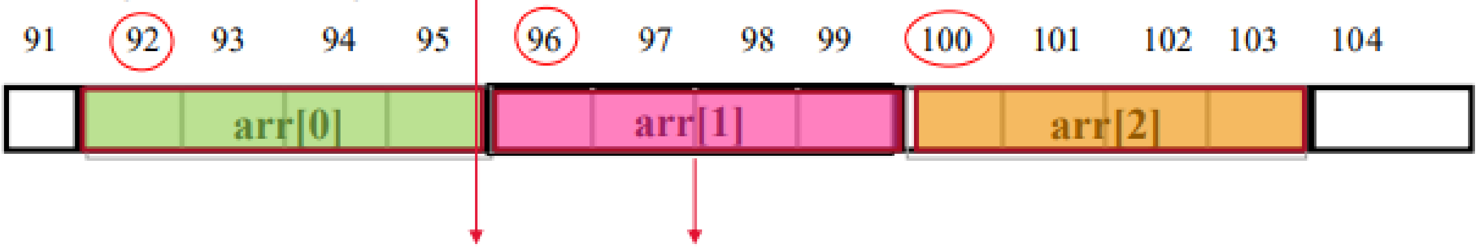
- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100   p + 2 = 96 + 2*4 = 104
- Array in memory:



- Suppose `p` points to array element `k`, then `p+1` points to `k+1` (next) element. `p + i` points to `arr[k+i]`.
  - `p = &arr[k]:        p + i == &arr[k+i]` → `*(p+i) == arr[k+i]`
  - `k=0:  p=&arr[0]:  p + i == &arr[i]` → `*(p+i) == arr[i]`

Pointer arithmetic summary

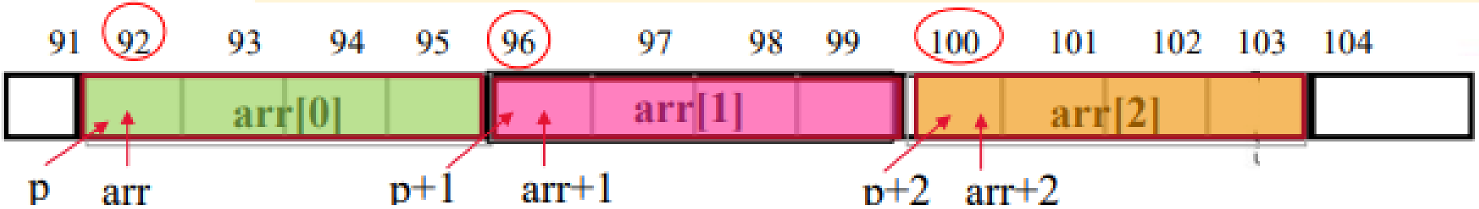
- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100 p + 2 = 96 + 2*4 = 104
- Array in memory:



- Suppose `p` points to array element `k`, then `p+1` points to `k+1` (next) element. `p + i` points to `arr[k+i]`.
  - `p= &arr[k]:        p + i == &arr[k+i]   → *(p+i) == arr[k+i]`
  - `k=0:  p=&arr[0]:  p + i == &arr[i]    → *(p+i) == arr[i]`

- **Array name contains pointer to 1st element** `arr==&arr[0]`
  - `arr==&arr[0]:       arr+i == &arr[i]   → *(arr+i) == arr[i]`

`p = arr:      p + i == &arr[i]  →    *(p+i) == arr[i]`



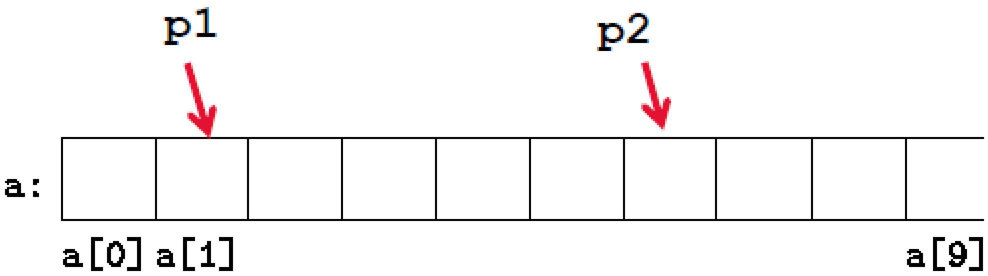
Pointer arithmetic (revisit + extension)

- `+n -n ++ --`
- If `p1, p2` points to different elements of the same array
  - Differencing: `p1 - p2`

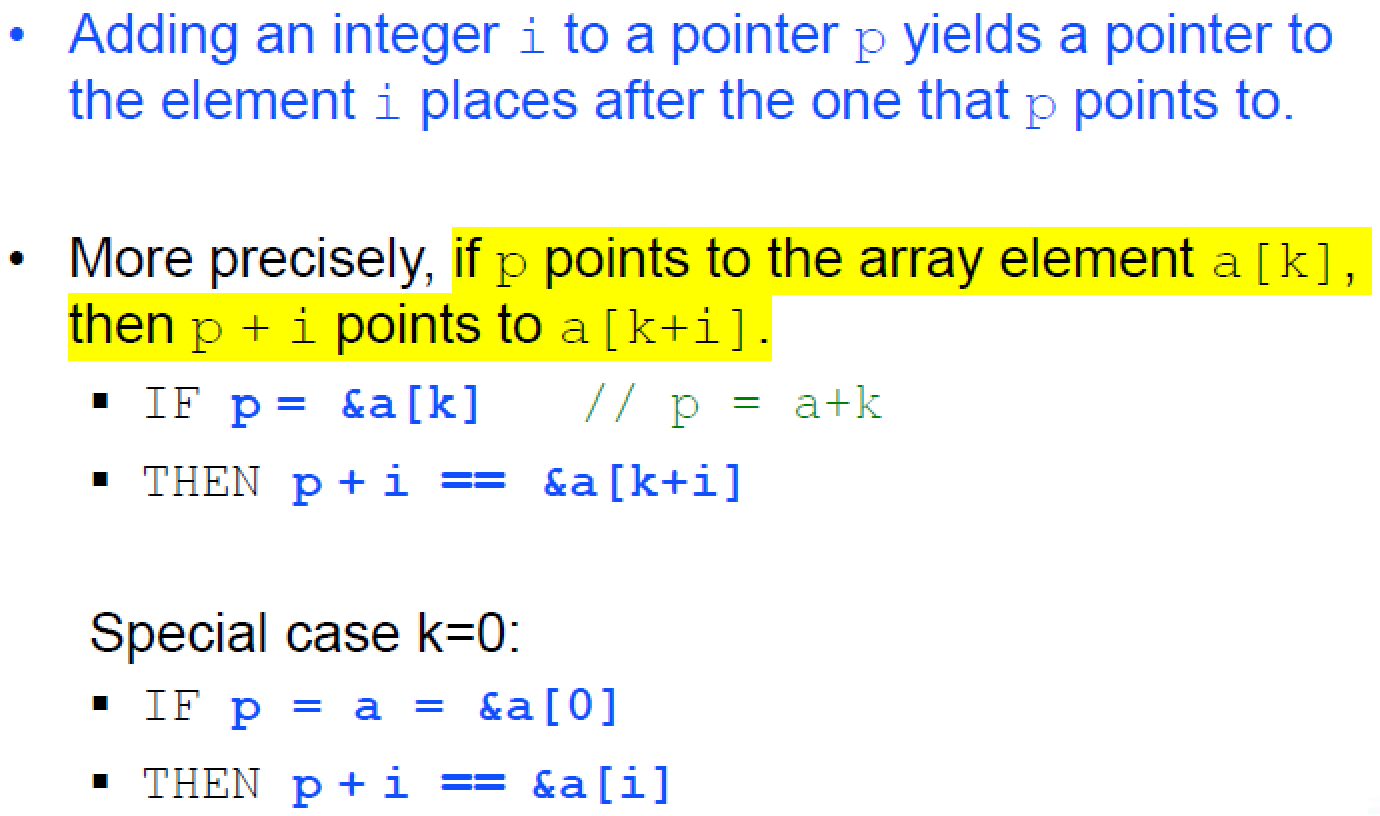
    result is how far apart in term of # elements

  - Comparison : `== != > < >= <=`

    `p1 < p2` is true (1) if `p1` points to earlier elements than `p2`



Pointer arithmetic on arrays
Adding an Integer to a Pointer +i



- Adding an integer `i` to a pointer `p` yields a pointer to the element `i` places after the one that `p` points to.

- More precisely, if `p` points to the array element `a[k]`, then `p + i` points to `a[k+i]`.
  - `IF p= &a[k]    // p = a+k`
  - `THEN p+i == &a[k+i]`

  Special case k=0:
  - `IF p = a = &a[0]`
  - `THEN p+i == &a[i]`

Pointer arithmetic on arrays
Adding an Integer to a Pointer -i

- Subtracting an integer i to a pointer p yields a pointer to the element i places before the one that p points to.

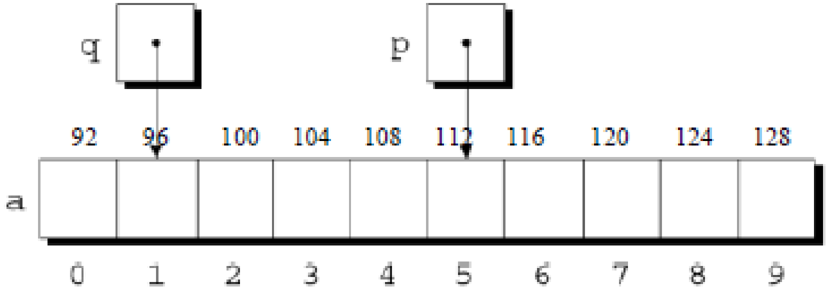
- More precisely, if p points to the array element a[k], then p – i points to a[k-i].
  - IF p = &a[k]   // p = a+k
  - THEN p - i == &a[k-i]

Pointer arithmetic on arrays (extended) Comparing Pointers

- Pointers can be compared using the relational operators (< <= > >=) and the equality operators (== and !=).
  - Using relational operators is meaningful only for pointers to elements of the same array.

- The outcome of the comparison depends on the relative positions of the two elements in the array.

```
p = &a[5];
q = &a[1];
p <= q      "false"  0
p >= q      "true"   1
```



Arrays passed to a Function

- The name/identifier of the array passed is actually a pointer/address to its first element. arr == &arr[0];

```
char a[20] = "Hello";
strlen(a); /* strlen(&a[0]). 96 is passed */
```

- The call to a function does not copy the whole array itself, just a *address* *(starting address -- a single value)* to it.

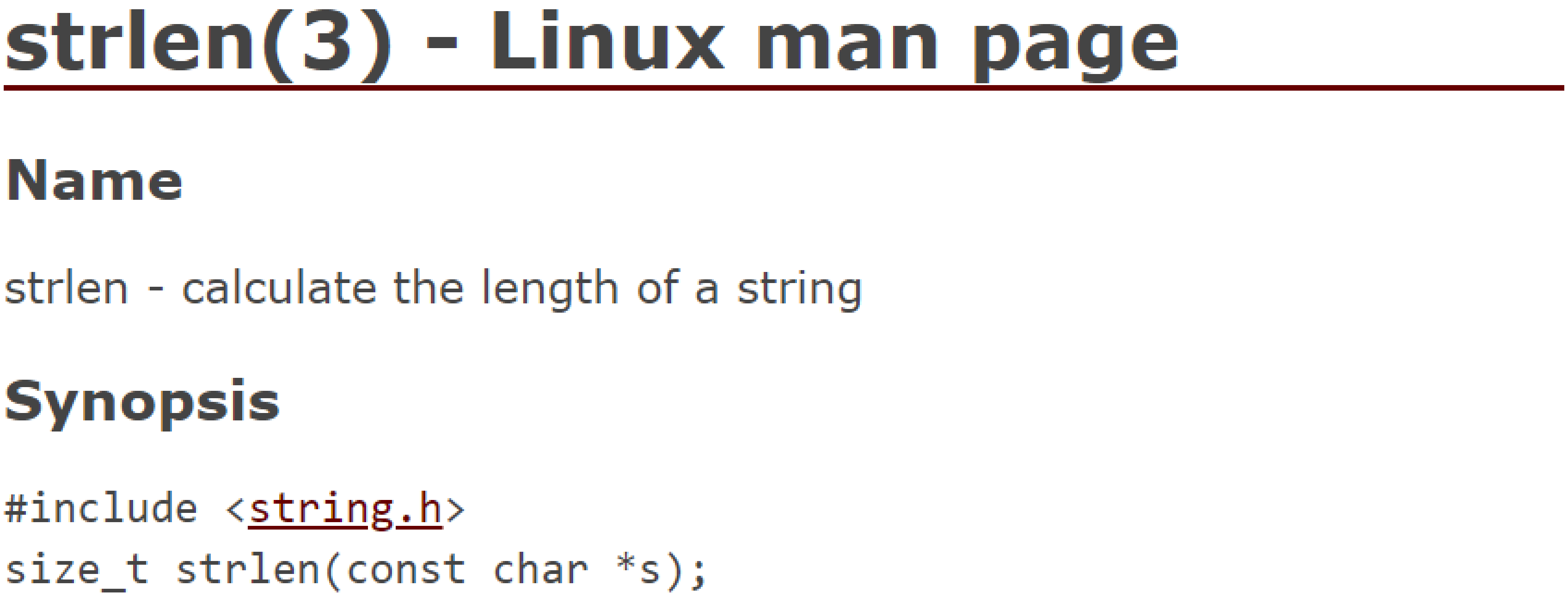
- Thus, function expecting a char array can be declared as either

```
strlen(char s[]);
```

  or

```
strlen(char * s);
```

Actual prototype man 3 strlen



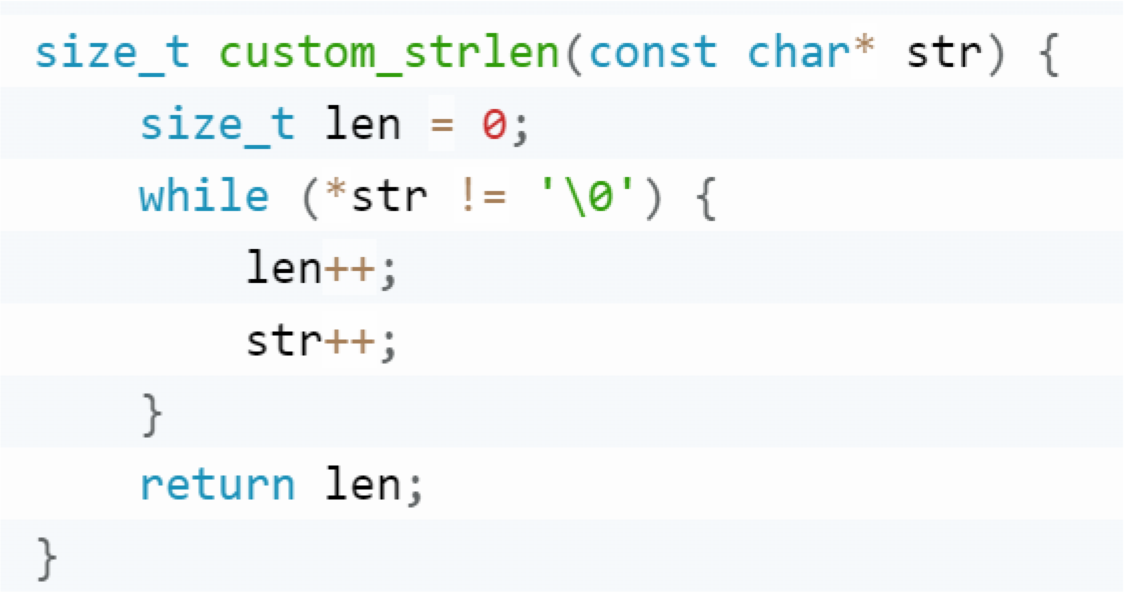
strlen(3) - Linux man page

Name

strlen - calculate the length of a string

Synopsis

```
#include <string.h>
size_t strlen(const char *s);
```



```
size_t custom_strlen(const char* str) {
    size_t len = 0;
    while (*str != '\0') {
        len++;
        str++;
    }
    return len;
}
```

Arrays Passed to a Function

- Thus, function expecting a char array can be declared as either

```
strlen(char s[]);
```

or

```
strlen(char * s);
```

Actual prototype   man 3 strlen

- The call to this function does not copy the whole array itself, just *a address* *(starting address -- a single value)* to it.

“decay”

```
char a[20] = "Hello";
char * ps = a;
strlen(a); /* strlen(&a[0]). 96 is passed */
strlen(ps);
```

Pass by value: 96 is passed and copied to s

```
s = a = &a[0] //s is a local pointer variabl
s = ps = a = &a[0] // in function
```

Arrays Passed to a Function

- Arrays passed to a function are passed by starting address.
- The name/identifier of the array passed is treated as a pointer to its first element. arr == &arr[0];

array-to-pointer conversion "decay"

By passing an array by a pointer (its starting address)

1. Array can be passed (efficiently)
   - a single value (e.g, 96, no matter how long array is)
2. Argument array can be modified
   - no **&** needed

```
strcpy(arr, "hello");
scanf("%s %d %f %c", arr, &age, &rate, &c);
sscanf (table[i], "%s %d %f %c", name,&age,&rate,&c)
```

Passing Sub-arrays to Functions

- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the sub-array.

```
char arr[20] = "hi world";
char * p = arr;  // &arr[0]
strlen(&arr[0]);
strlen(arr);
strlen(p);     8
```

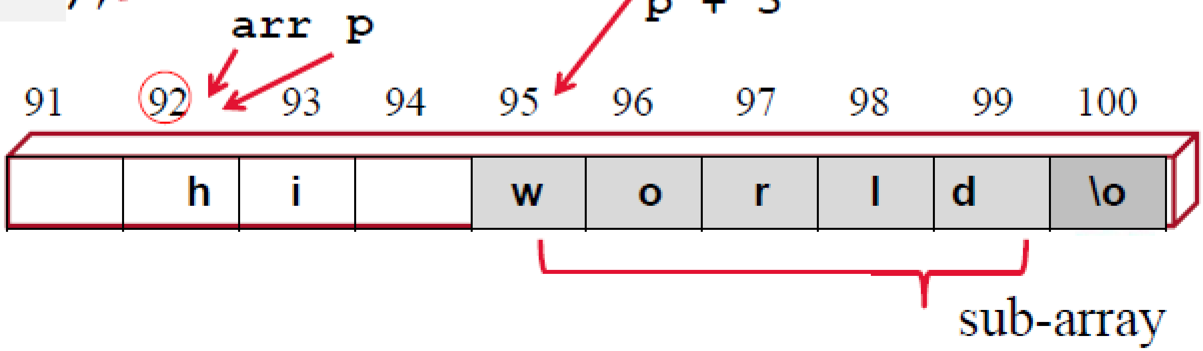
Functions receive address 92

```
printf("%s", p); // arr &arr[0]
```

```
//length of world
strlen (         );
strlen (         );
strlen (         );
```

Functions receive address 95

```
&arr[3]
arr + 3
p + 3
```



```
print world?
```

Passing Sub-arrays to Functions

- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the sub-array.

```
char arr[20] = "hi world";
char * p = arr;  // &arr[0]
strlen(&arr[0]);
strlen(arr);
strlen(p);      8
```

Pointer/address level

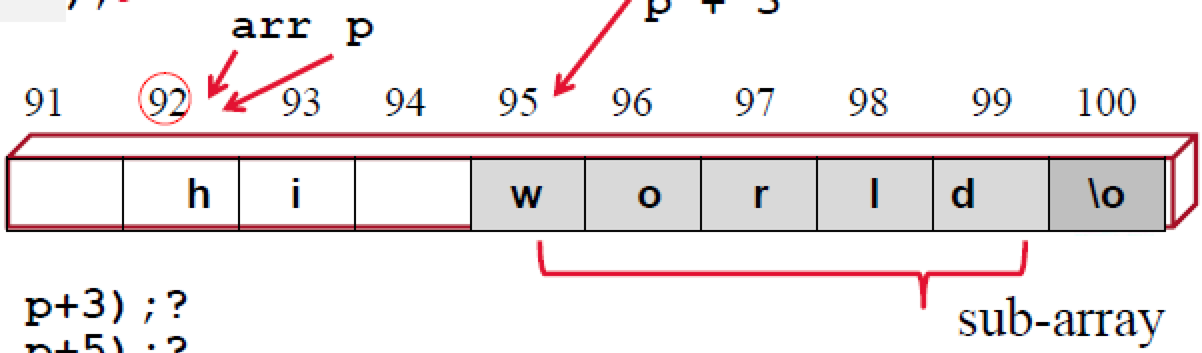
Functions receive address 92

```
printf("%s", p); // arr &arr[0]
```

```
//length of world
strlen (          );
strlen (          );
strlen (          );
```

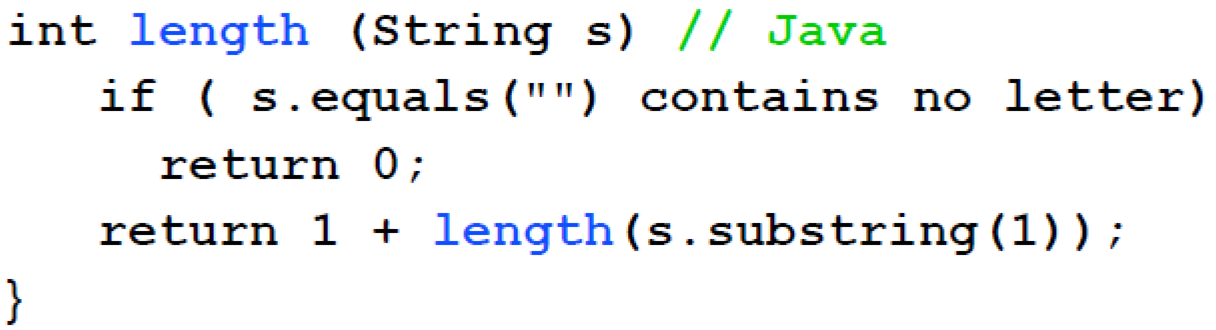
Functions receive address 95

```
&arr[3]
arr + 3
p + 3
```

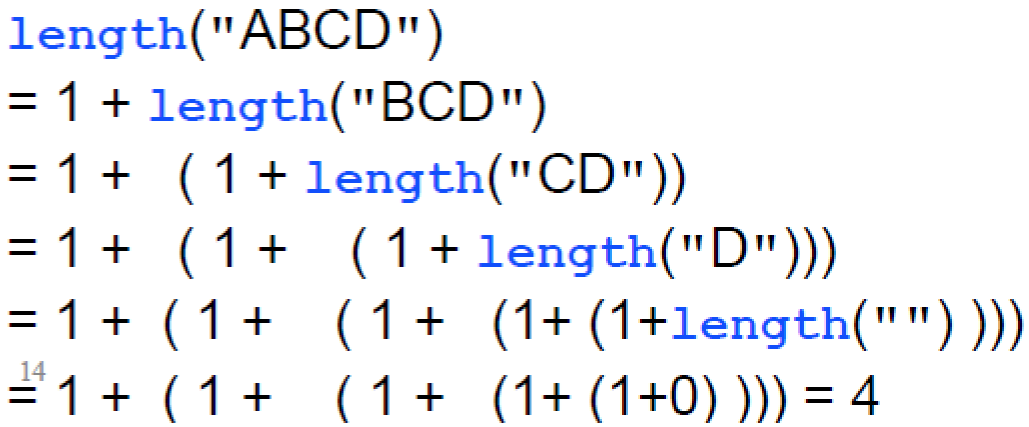


```
print world?

printf("%s", p+3);?
printf("%s", p+5);?
```

Passing Sub arrays to Functions -- Recursion



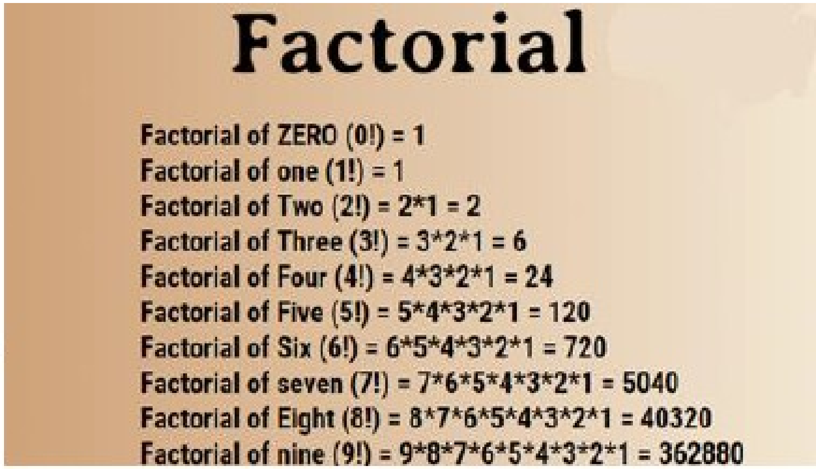
```
int length (String s) // Java
   if ( s.equals("") contains no letter)
     return 0;
   return 1 + length(s.substring(1));
}
```



```
length("ABCD")
= 1 + length("BCD")
= 1 +  ( 1 + length("CD"))
= 1 +  ( 1 +   ( 1 + length("D")))
= 1 + ( 1 +   ( 1 +  (1+ (1+length("") )))
= 1 + ( 1 +   ( 1 +  (1+ (1+0) ))) = 4
```

Recursion

- C supports recursion
- Think/define recursively

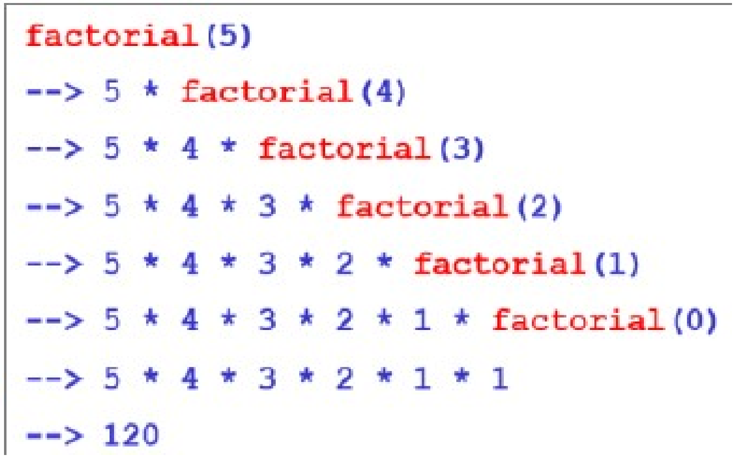


Factorial

Factorial of ZERO (0!) = 1
Factorial of one (1!) = 1
Factorial of Two (2!) = 2*1 = 2
Factorial of Three (3!) = 3*2*1 = 6
Factorial of Four (4!) = 4*3*2*1 = 24
Factorial of Five (5!) = 5*4*3*2*1 = 120
Factorial of Six (6!) = 6*5*4*3*2*1 = 720
Factorial of seven (7!) = 7*6*5*4*3*2*1 = 5040
Factorial of Eight (8!) = 8*7*6*5*4*3*2*1 = 40320
Factorial of nine (9!) = 9*8*7*6*5*4*3*2*1 = 362880

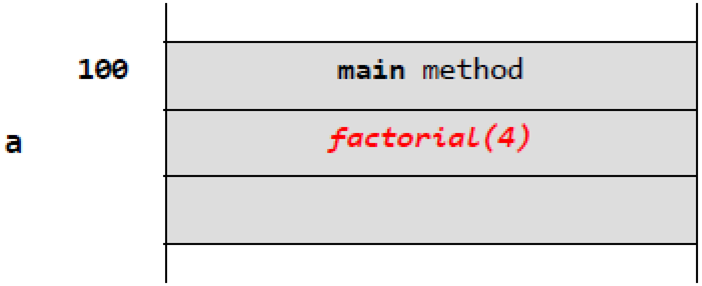
$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot factorial(n-1) & otherwise \end{cases}$$

```c
int factorial (int n)
{
  if(n == 0) /* base case  */
   return 1;
  else
   return n * factorial (n - 1);
}
```



```
factorial(5)
--> 5 * factorial(4)
--> 5 * 4 * factorial(3)
--> 5 * 4 * 3 * factorial(2)
--> 5 * 4 * 3 * 2 * factorial(1)
--> 5 * 4 * 3 * 2 * 1 * factorial(0)
--> 5 * 4 * 3 * 2 * 1 * 1
--> 120
```

```
int a = factorial(4)
```

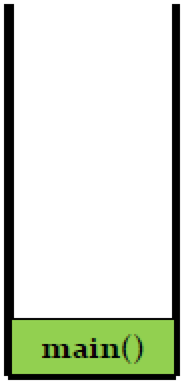


100

main method

a

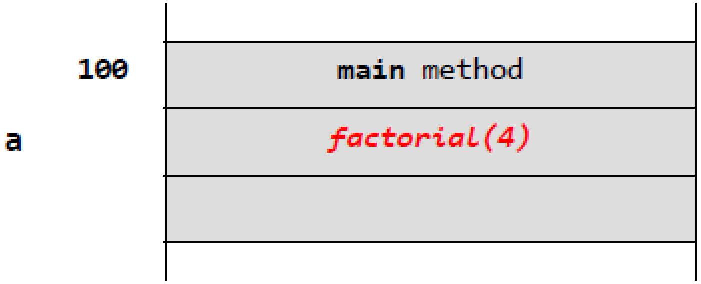
factorial(4)



main()

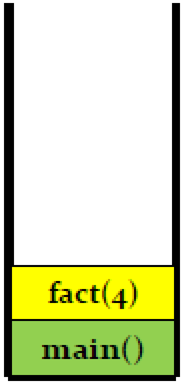
call/execution/program stack

```
int a = factorial(4)
```



100 **main** method

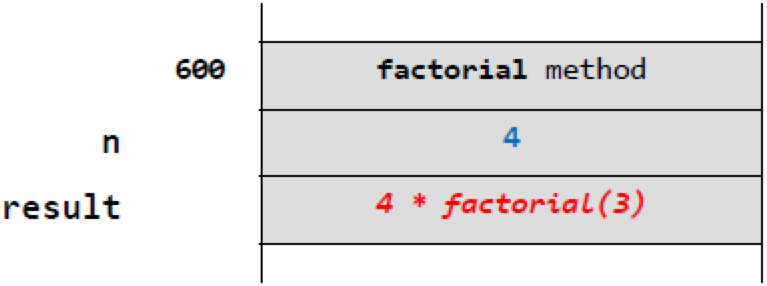
a *factorial(4)*



fact(4)

main()

call/execution/program stack



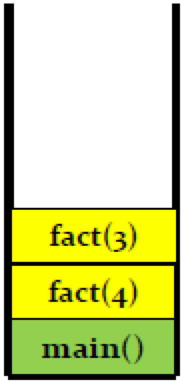
600 **factorial** method

n 4

result *4 * factorial(3)*

```
int a = factorial(4)
```

100	**main** method
a	*factorial(4)*

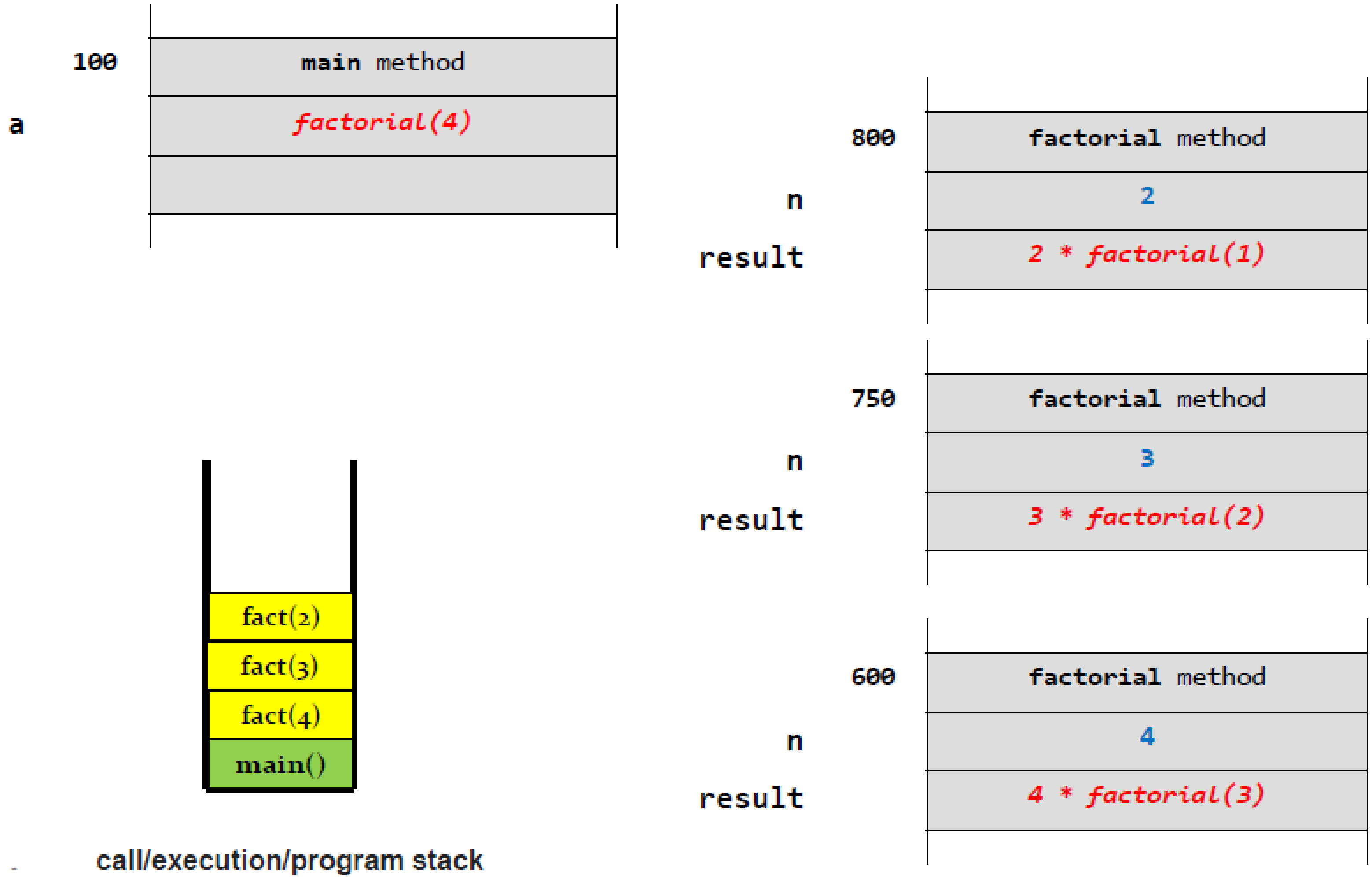


call/execution/program stack

750	**factorial** method
n	3
result	*3 * factorial(2)*

600	**factorial** method
n	4
result	*4 * factorial(3)*

```
int a = factorial(4)
```



100 main method

a *factorial(4)*

800 factorial method

n 2

result *2 * factorial(1)*

750 factorial method

n 3

result *3 * factorial(2)*

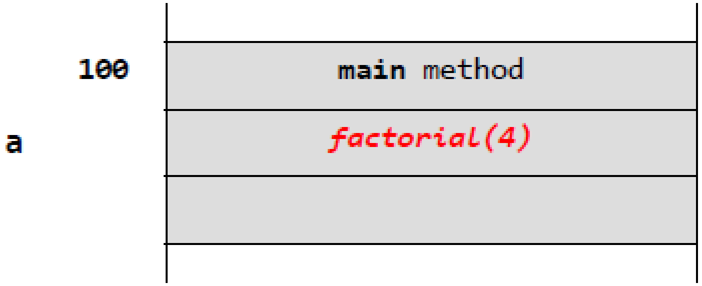
600 factorial method

n 4

result *4 * factorial(3)*

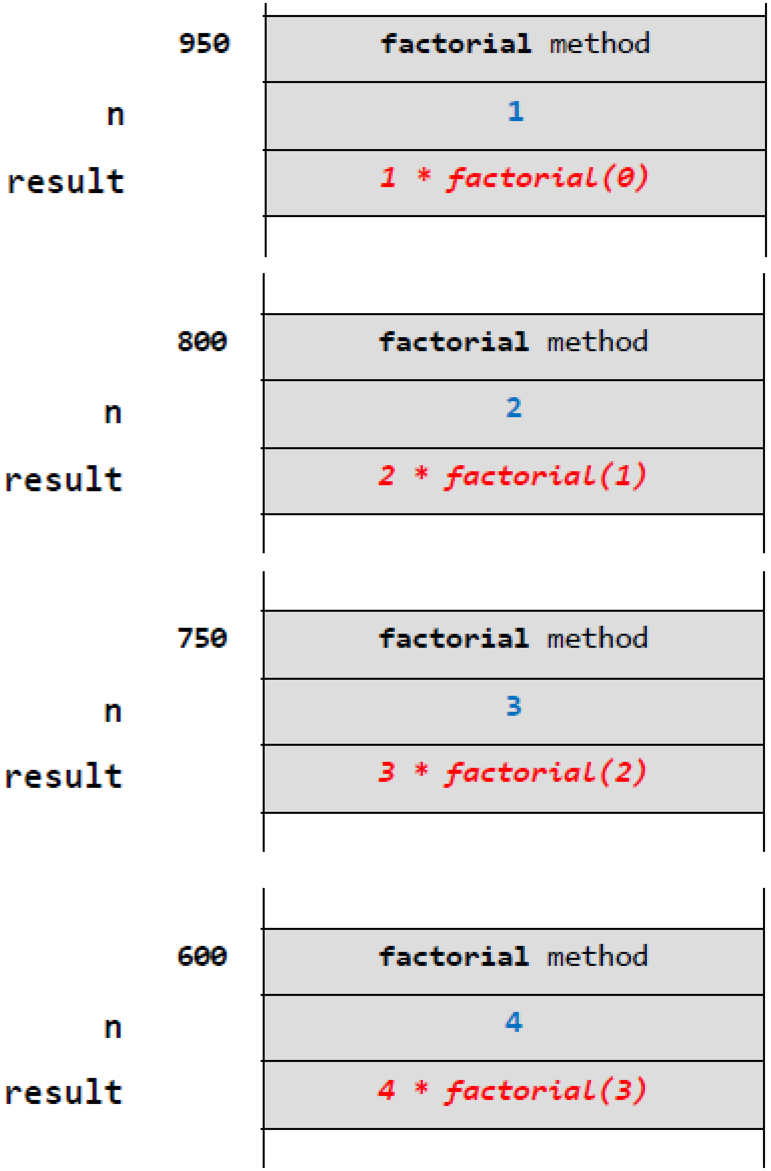
call/execution/program stack

```
int a = factorial(4)
```



100 **main** method

a *factorial(4)*



950 **factorial** method

n 1

result *1 * factorial(0)*

800 **factorial** method

n 2

result *2 * factorial(1)*

750 **factorial** method

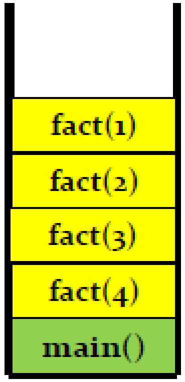
n 3

result *3 * factorial(2)*

600 **factorial** method

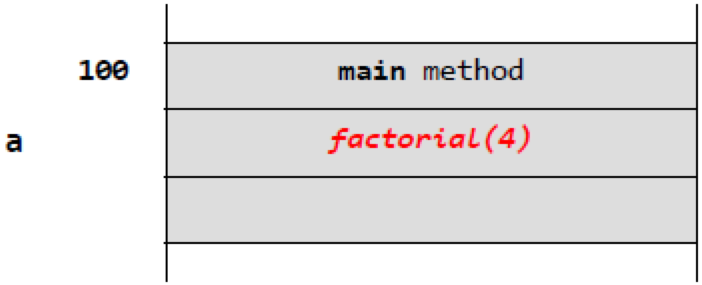
n 4

result *4 * factorial(3)*



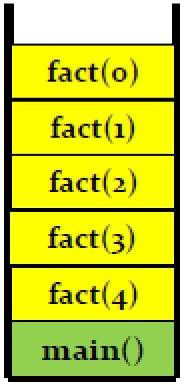
call/execution/program stack

```
int a = factorial(4)
```



100 main method

a *factorial(4)*



fact(0)

fact(1)

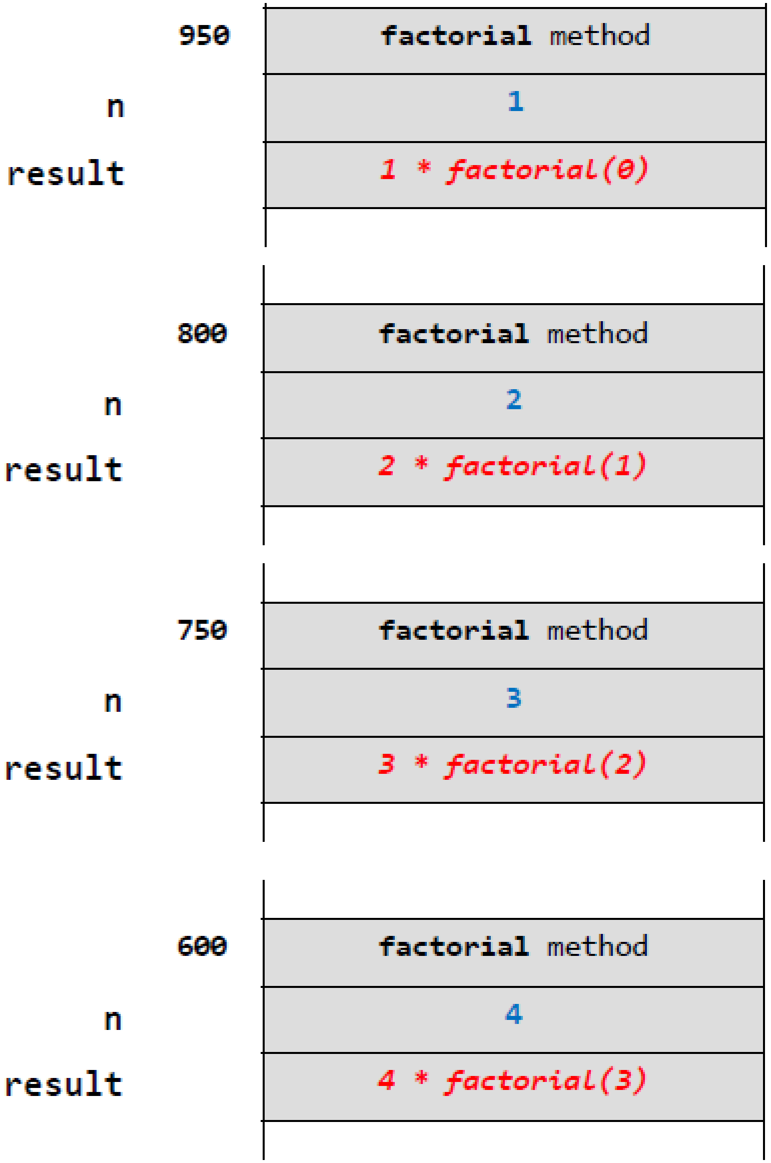
fact(2)

fact(3)

fact(4)

main()

call/execution/program stack



950 factorial method

n 1

result *1 * factorial(0)*

800 factorial method

n 2

result *2 * factorial(1)*

750 factorial method

n 3

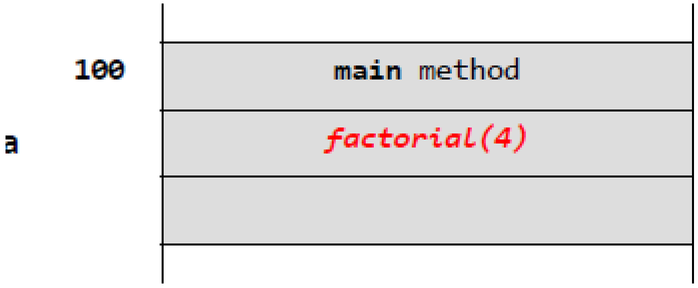
result *3 * factorial(2)*

600 factorial method

n 4

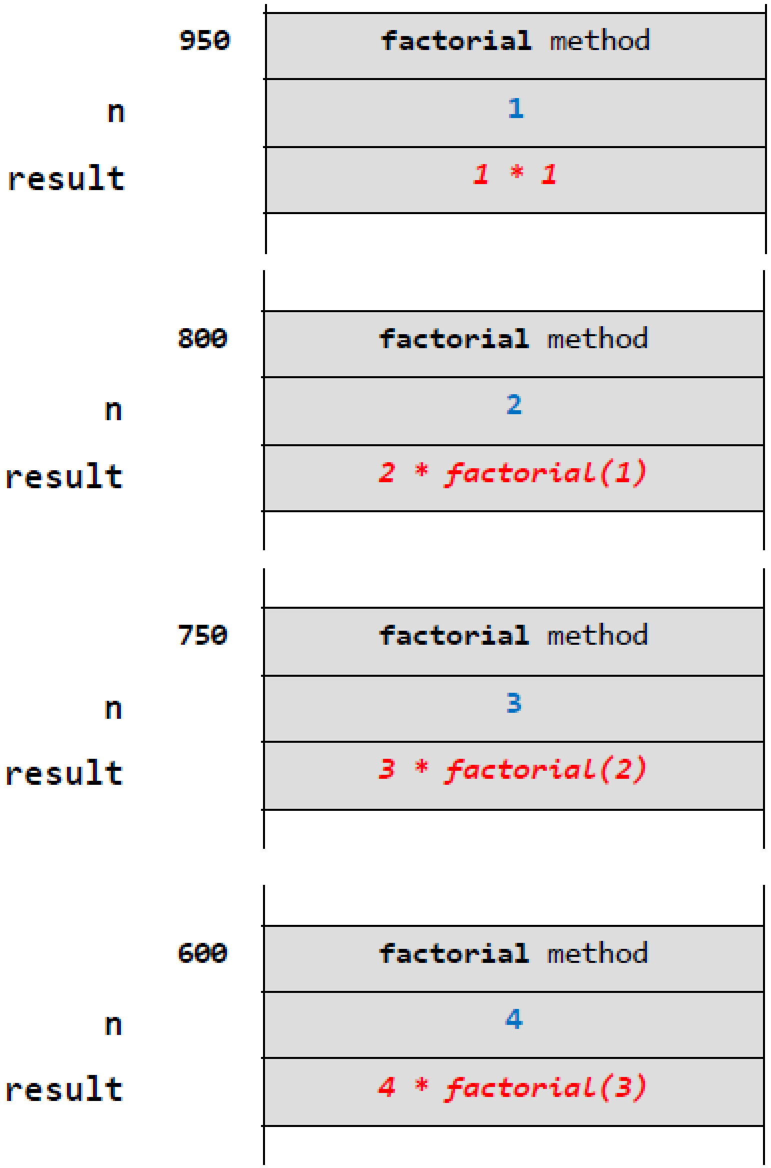
result *4 * factorial(3)*

```
int a = factorial(4)
```



100 main method

a *factorial(4)*



950 factorial method

n 1

result *1 * 1*

800 factorial method

n 2

result *2 * factorial(1)*

750 factorial method

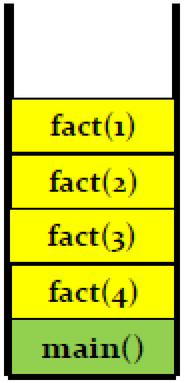
n 3

result *3 * factorial(2)*

600 factorial method

n 4

result *4 * factorial(3)*



fact(1)

fact(2)

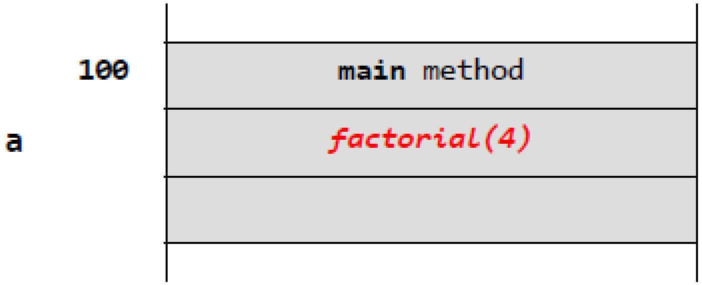
fact(3)

fact(4)

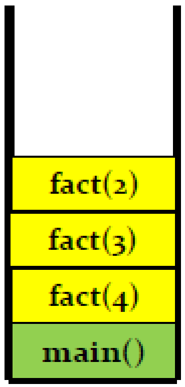
main()

call/execution/program stack

```
int a = factorial(4)
```

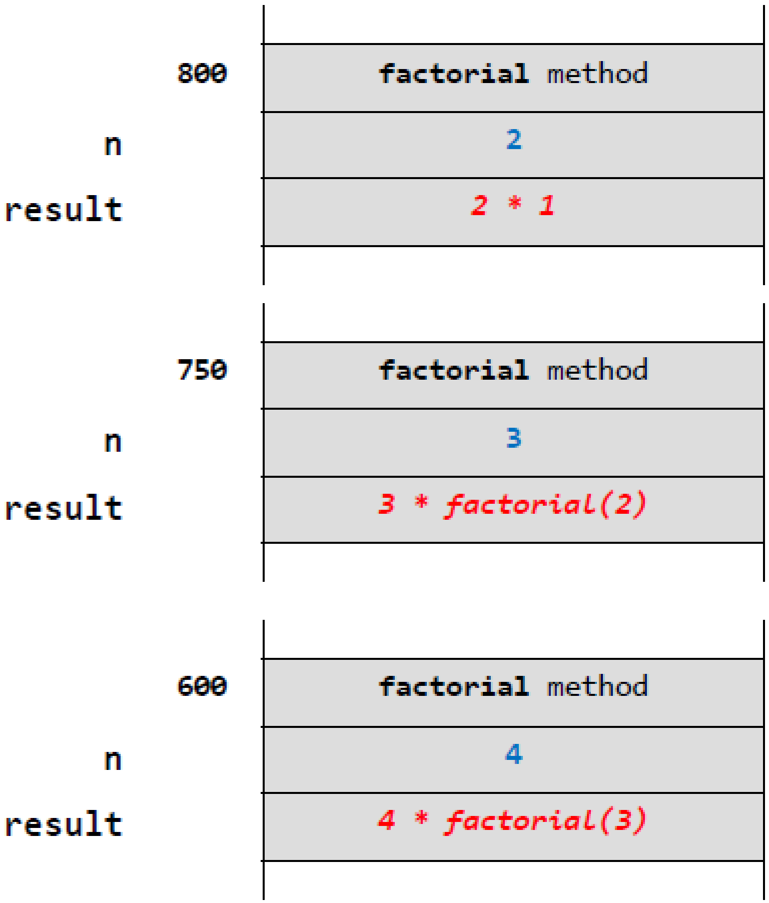


	100	**main** method
a		*factorial(4)*



fact(2)
fact(3)
fact(4)
main()

call/execution/program stack

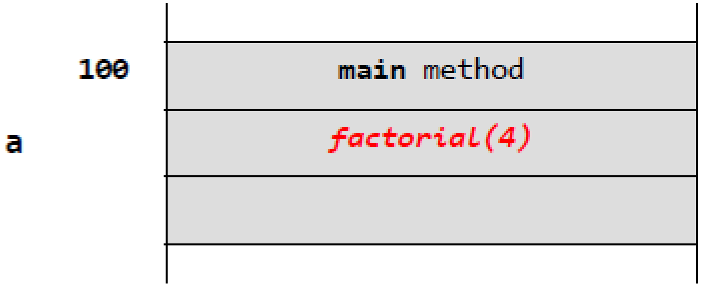


	800	**factorial** method
n		2
result		*2 * 1*

	750	**factorial** method
n		3
result		*3 * factorial(2)*

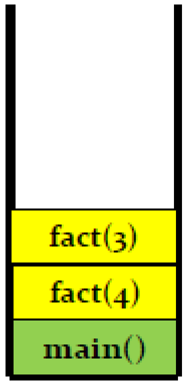
	600	**factorial** method
n		4
result		*4 * factorial(3)*

```
int a = factorial(4)
```

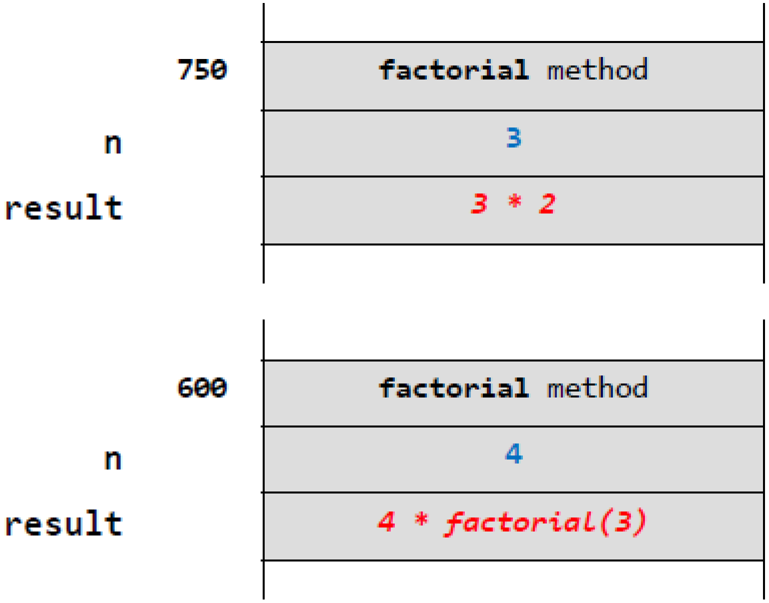


100 **main** method

a *factorial(4)*



call/execution/program stack



750 **factorial** method

n 3

result *3 * 2*

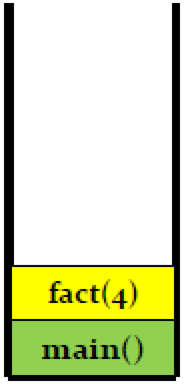
600 **factorial** method

n 4

result *4 * factorial(3)*

```
int a = factorial(4)
```

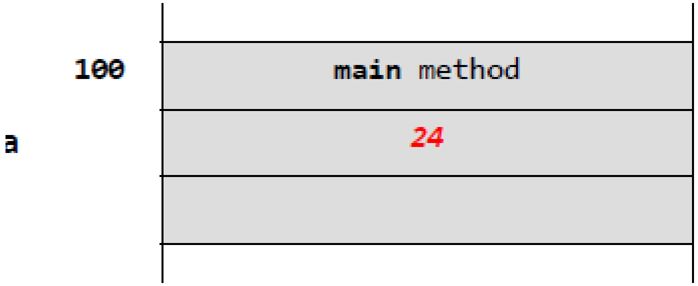
	100	**main** method
a		*factorial(4)*

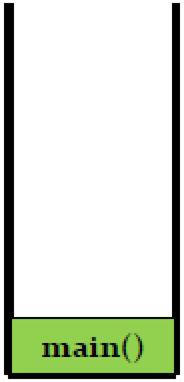


call/execution/program stack

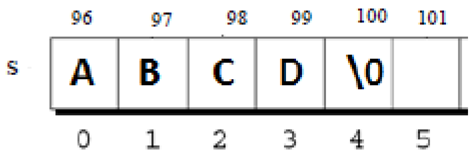
	600	**factorial** method
n		4
result		*4 * 6*

```
int a = factorial(4)
```

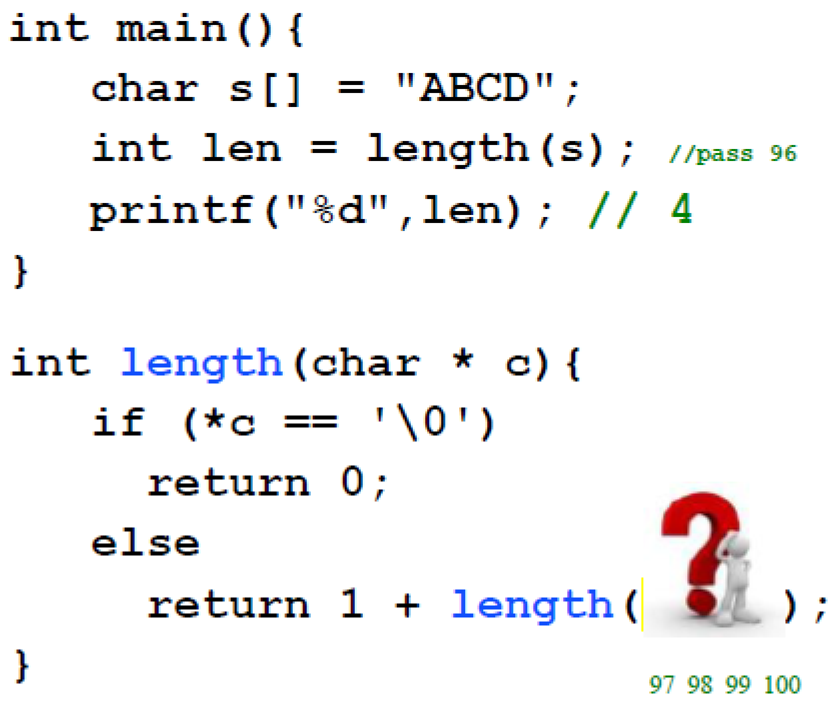




call/execution/program stack



	96	97	98	99	100	101
s	A	B	C	D	\0	
	0	1	2	3	4	5



```
int main(){
    char s[] = "ABCD";
    int len = length(s); //pass 96
    printf("%d",len); // 4
}

int length(char * c){
    if (*c == '\0')
      return 0;
    else
      return 1 + length(  );
}
```

97 98 99 100

length("ABCD")
= 1 + length("BCD")
= 1 + (1 + length("CD"))
= 1 + (1 + (1 + length("D")))
= 1 + (1 + (1 + (1+ (1+length(""))))
= 1 + (1 + (1 + (1+ (1+0)))) = 4

General array as function argument

- Pass an array/string by only the address/pointer of the first element
  - `strlen("Hello");`

"decay"

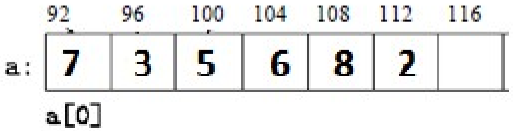
- You need to take care of where the array ends, the function does not know if it is an array or just a pointer to a char or int

- Two possible approaches:
  1. Special token/sentinel/terminator at the end (case of "string" `'\0'`)
  2. Pass the length as additional parameter

Function: `arrayLen(int *)`   `arraySum(int *)`

Caller: `int a[20]; arrLen(a); arraySum(a);`

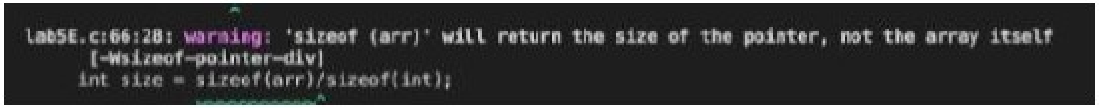
```
int main(){
   int a [] = {7,3,5,6,8,2};

   int max = findMax(a);
   ...
}
```



```
/* find max in the int array */
int findMax (int arr[]){ // (int * arr)

   int len = sizeof(arr)/sizeof(int);   //  8/4=2

   while ( i <  len ){
      ...
```

✗

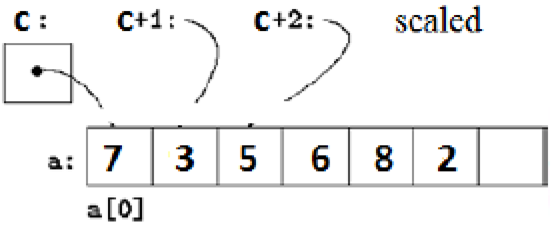
sizeof does not
work in function



```
lab5E.c:66:28: warning: 'sizeof (arr)' will return the size of the pointer, not the array itself
      [-Wsizeof-pointer-div]
     int size = sizeof(arr)/sizeof(int);
```

Some nice compiler (MAC. not lab gcc ☹)

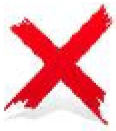
```
int main(){
     int arr [] = {17,3,5,19,8,2};
     finaMax(arr, 6);
}

/* find max in the int array. */
int findMax (int *c, int leng){
   int max = *c;
   int i=1;
   while ( i < leng ){
      ......
   }
   return max;
 }
```



Problems with pointers

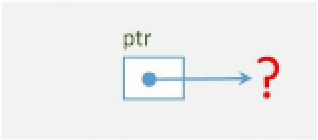
```
int *ptr;           /* I'm a pointer to an int */
ptr= &a              /* I got the address of a */
*ptr = 5;            /* set contents of the pointee a */
```

```
int *ptr;           /* I'm a pointer to an int */
*ptr = 5;          /* set contents of the pointee to 5 */
```



- `ptr` is uninitialized. “points to nothing”. “dangling”
  Has some random value `0x7fff033798b0`
  - may be your OS!

Dangling Pointers

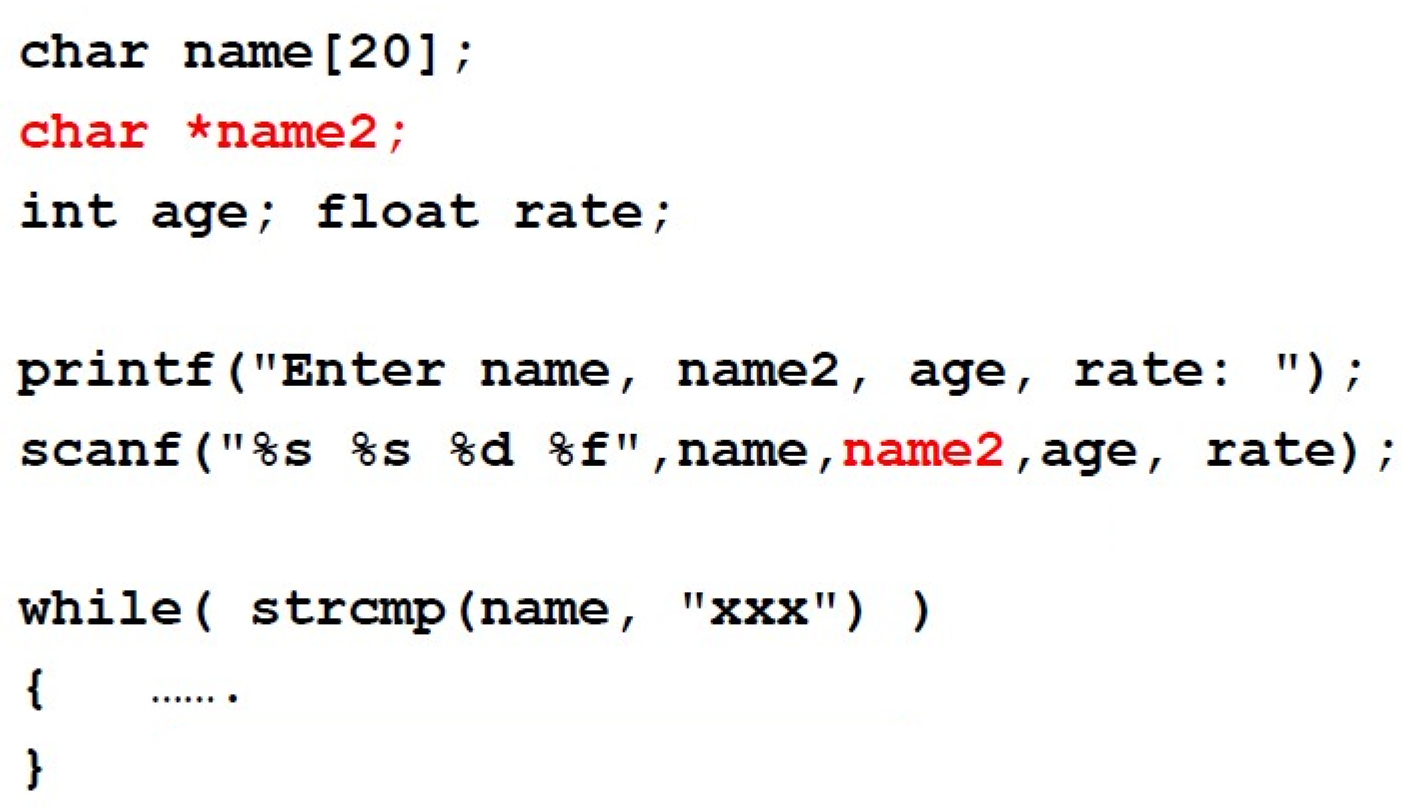


- dereferencing an uninitialized pointer? Undefined behavior!

- Always make `ptr` point to sth! How?

```
1) int a; ptr =&a;    int arr[20]; ptr=&arr[0];
2) ptr = ptr2    /* indirect. assuming ptr2 is
3) ptr = malloc (.....)
```

Problems with pointers



```
char name[20];
char *name2;
int age; float rate;

printf("Enter name, name2, age, rate: ");
scanf("%s %s %d %f",name,name2,age, rate);

while( strcmp(name, "xxx") )
{    …….
}
```