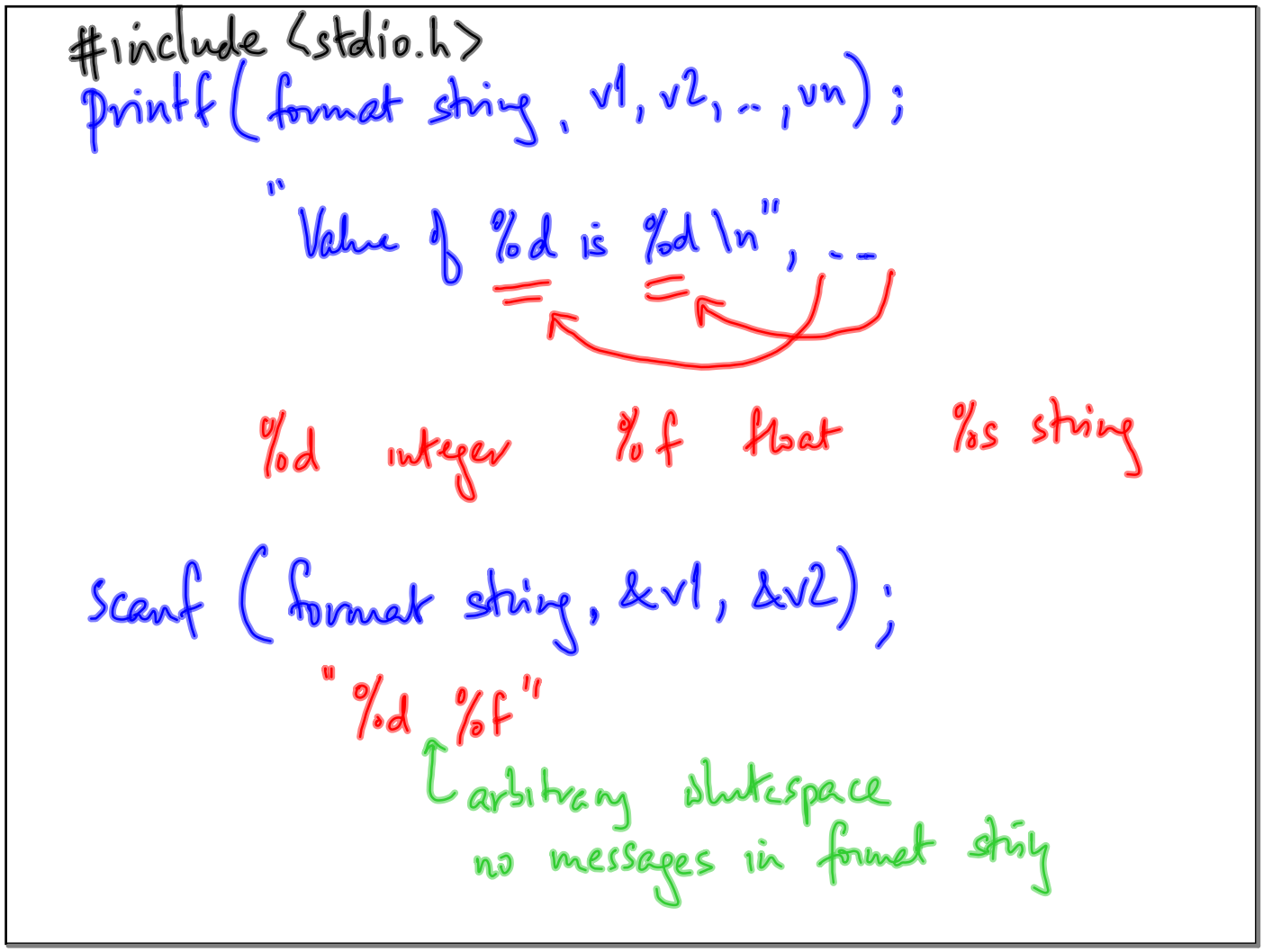
Call by value / call by reference

```
int i, *p;   // p is a "pointer" to int

p = &i ;     Make p point to location where
               i is stored

*p = 4 ;     "Dereference" p to update the
               location pointed to by p
```


#include <stdio.h>
printf (format string , v1, v2, .. , vn) ;
" Value of %d is %d \n", ..
%d integer
%f float
%s string
scanf (format string, &v1, &v2) ;
"%d %f"
arbitrary whitespace
no messages in format string

Structure of C program: Collection of fns, including main

```
#include <stdio.h>

int main(){
    int i;
    printf("Enter a number:");
    scanf("%d",&i);
    printf("%d! = %d\n",i,
                factorial(i));
}
```

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

Compiling a C program:

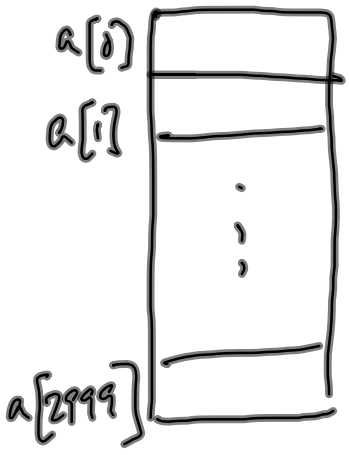
```
$ gcc factorial.c        ⇝ a.out

$ gcc factorial.c -o factorial
```

creates executable in "factorial"

Arrays

Fixed size contiguous block of identical values

```
int a[3000];  // a[0] to a[2999]
```



Memory Model

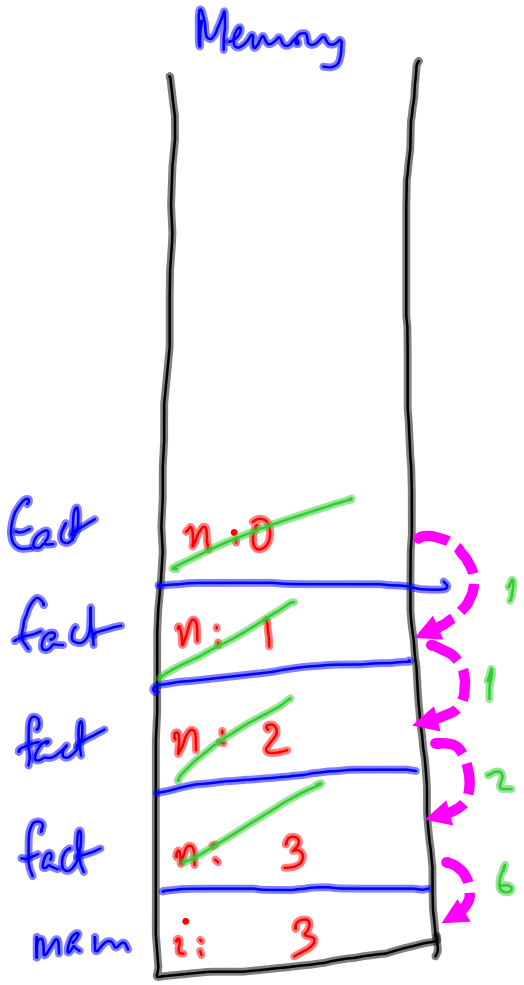
Compute factorial(3)

main: int i

factorial: int n

Each function gets space on stack

This space is discarded when function ends



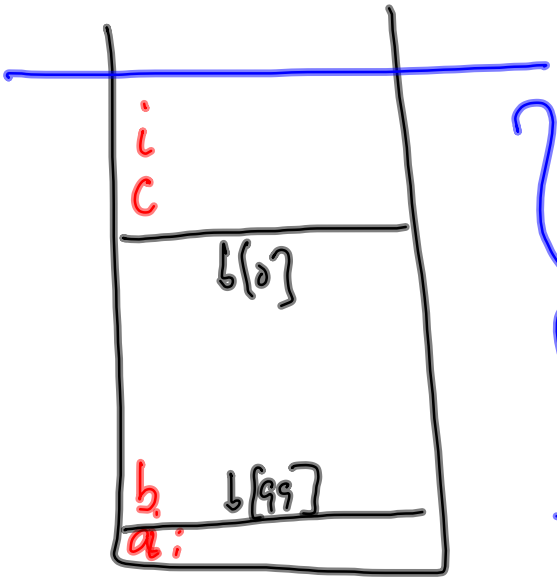
memorytest example

int a, b[100], c, i;

b[-1] ⇝ c

b[100] ⇝ a

∴ b[-2] = i ?



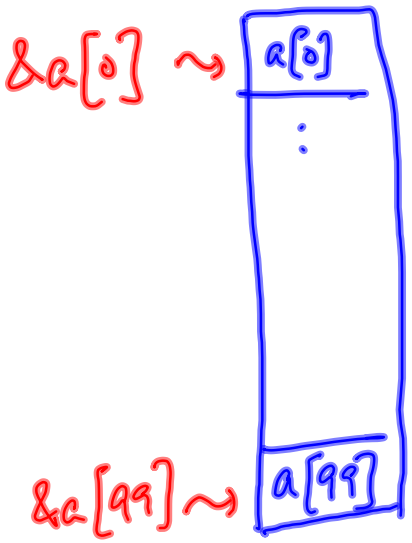
"Stack Frame" for main()

Back to arrays

int a[100]

By C convention:

a is a synonym for &a[0]



*a = 66 is same as a[0] = 66

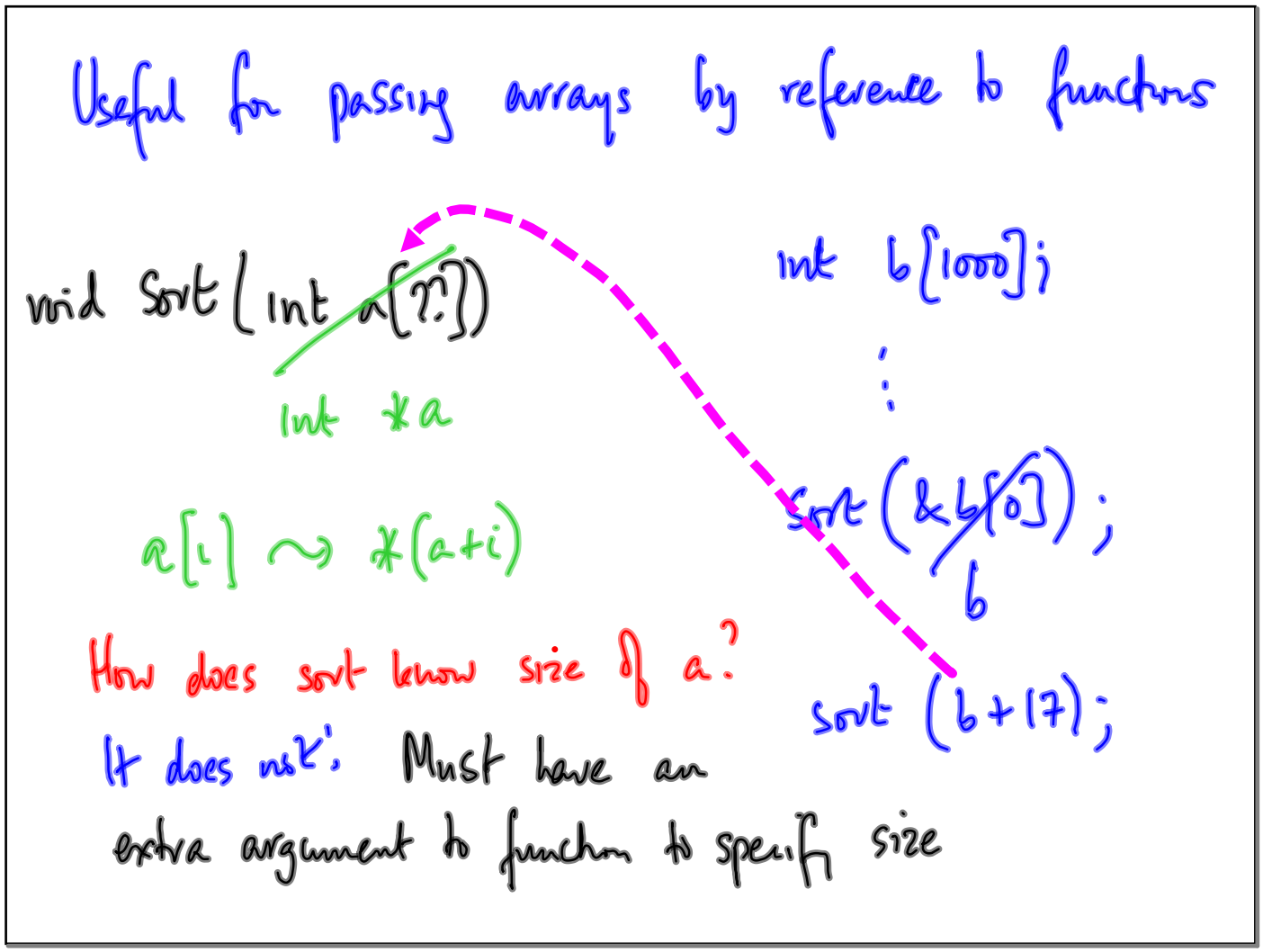
Furthermore, by definition,

$$\&a[1] = \&a[0] + 1 = a + 1$$
$$\vdots$$
$$\&a[i] = \&a[0] + i = a + i$$

regardless of type of a

"Pointer arithmetic"

$$*(a+i) \stackrel{\text{def}}{=} a[i]$$
$$a+i \stackrel{\text{def}}{=} \&a[i]$$


Useful for passing arrays by reference to functions
void sort(int a[??])
int *a
a[i] ~> *(a+i)
How does sort know size of a?
It does not. Must have an extra argument to function to specify size
int b[1000];
sort(&b[0]);
b
sort(b+17);

What happens to dynamic data structures

```
class Node:
    ⋮
def append(self,m):
    =
```

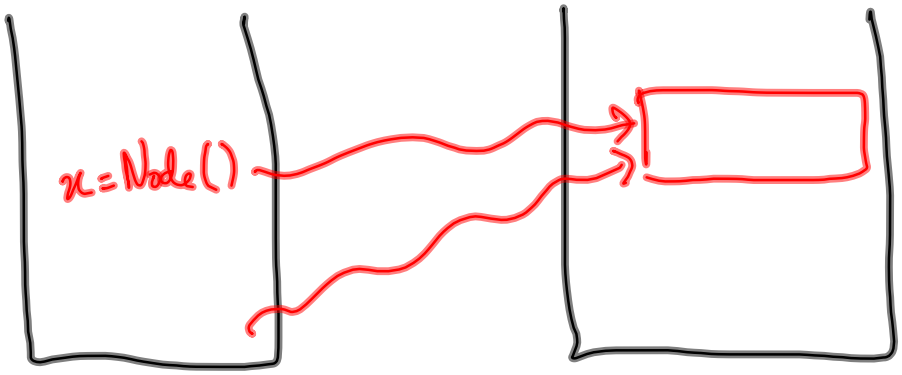
In C:

```
~ append(int n){
    =
}
```

must create space for a new node

Newly created node is not on stack ← Stack frame for append is erased when append finishes

C has two conceptually different "zones" in memory

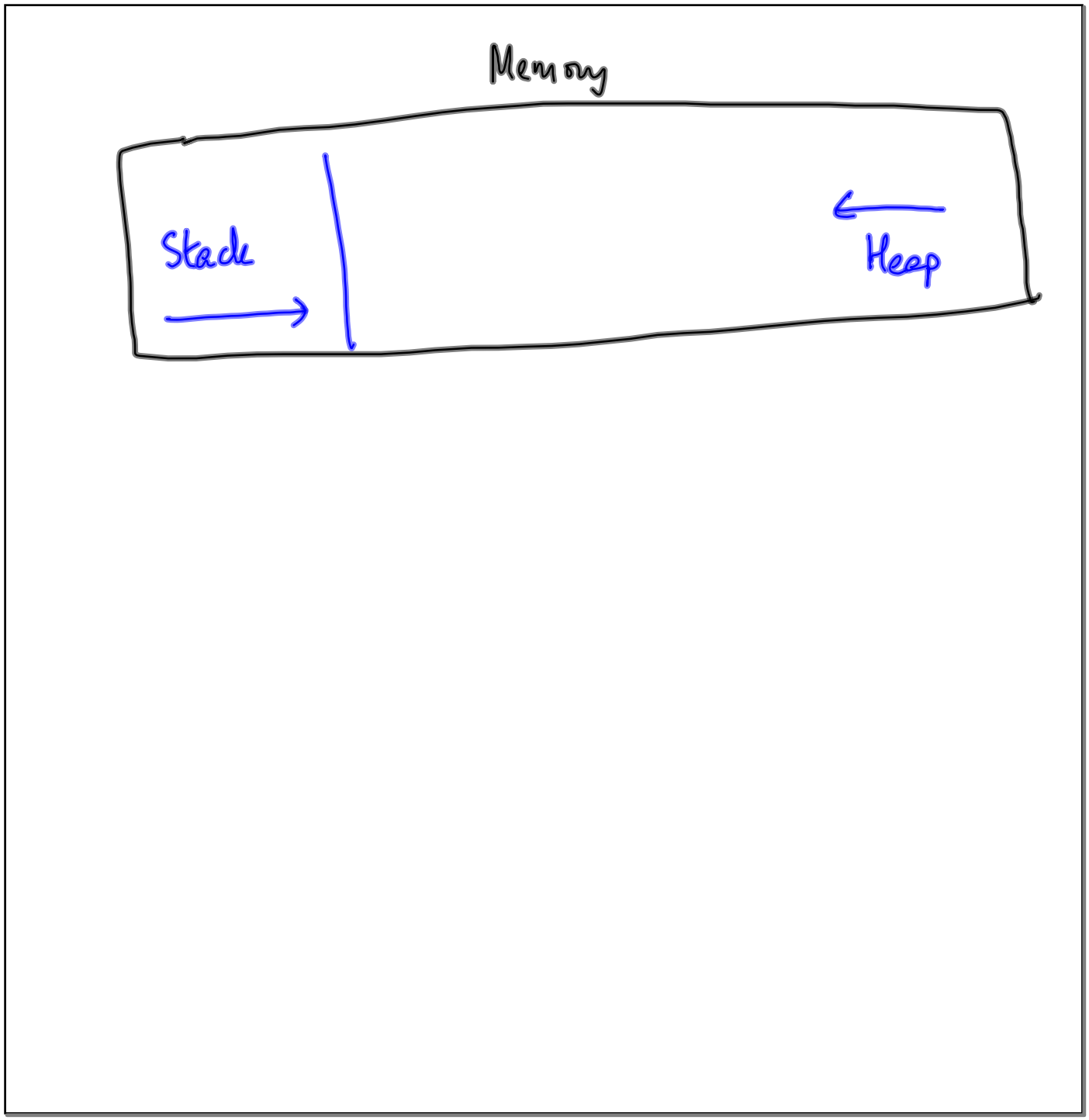


Call Stack

"Persistent" storage

≡ "HEAP"

nothing to do with priority queues


Memory
Stack
Heap