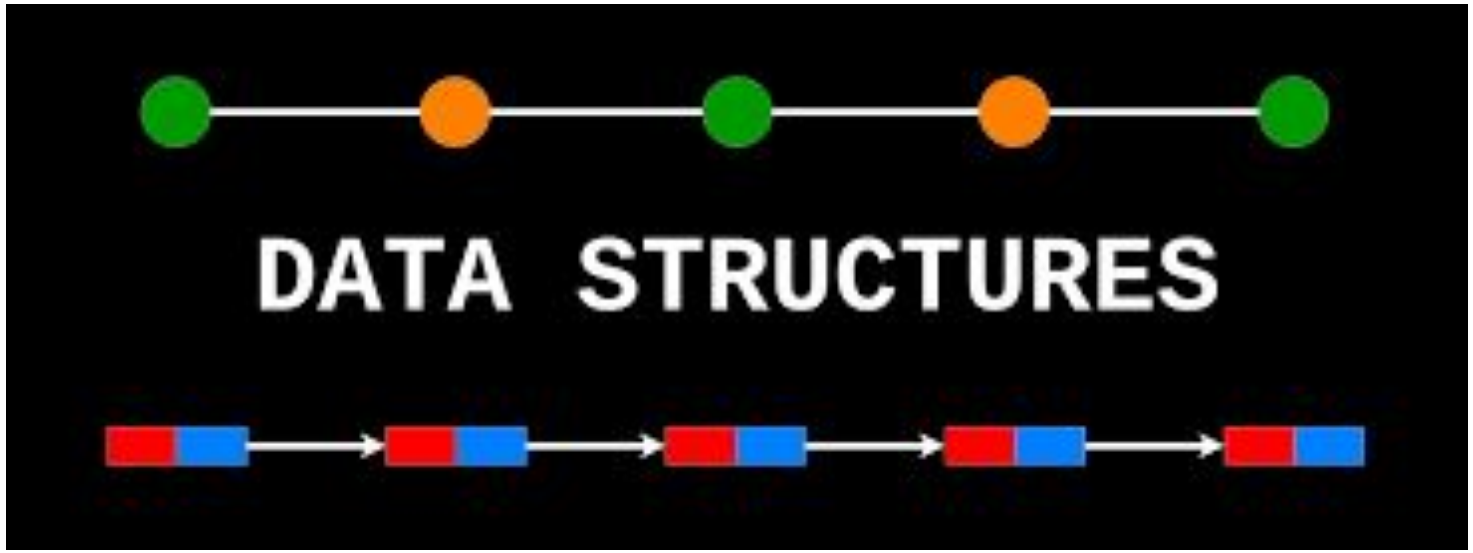
DATA STRUCTURES USING C



22BCA2C05

POINTERS

The pointer in C language is **a variable that stores the address of another variable**. This variable can be of type int, char, array, function, or any other pointer.

Example

```
1.int n = 10;
2.int* p = &n; // Variable p of type pointer is pointing to the address of t
  he variable n of type integer.
```

DECLARING A POINTER

- The pointer in c language can be declared using * (asterisk symbol). It is also known as an indirection pointer used to dereference a pointer.

```
int *a;//pointer to int
char *c;//pointer to char
```

EXAMPLE

```c
#include <stdio.h>

int main() {
    int n=10;
    int *p=n;
    int *c=&n;
    printf("value is %d Addrss of pointer %d", p,c);

    return 0;
}
```

Output

```
/tmp/817TFI59t2.o
value is 10 Add of pointer -1427657884
```

ADVANTAGE OF POINTER

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieve strings, trees, etc.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Address Of (&) Operator

- The address of operator '&' returns the address of a variable. we can also use **%u to display the address of a variable**.

```
#include<stdio.h>
int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);
return 0;
}
```

EXAMPLE

```c
#include <stdio.h>

int main() {
    int n=10;
    int *p=n;
    int *c=&n;
    printf("value is %d Add of pointer %u add without & %d", p,&n,c);


    return 0;
}
```

Output

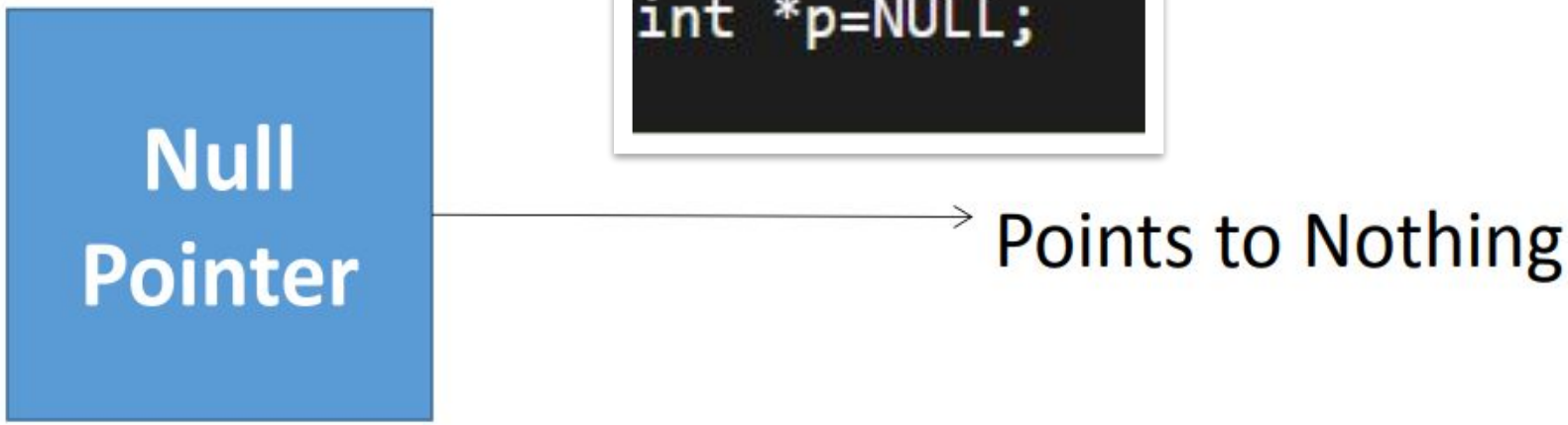
```
/tmp/817TFI59t2.o
value is 10 Add of pointer 551172644 add without & 551172644
```

NULL Pointers

A pointer that is assigned NULL is called a **null** pointer.

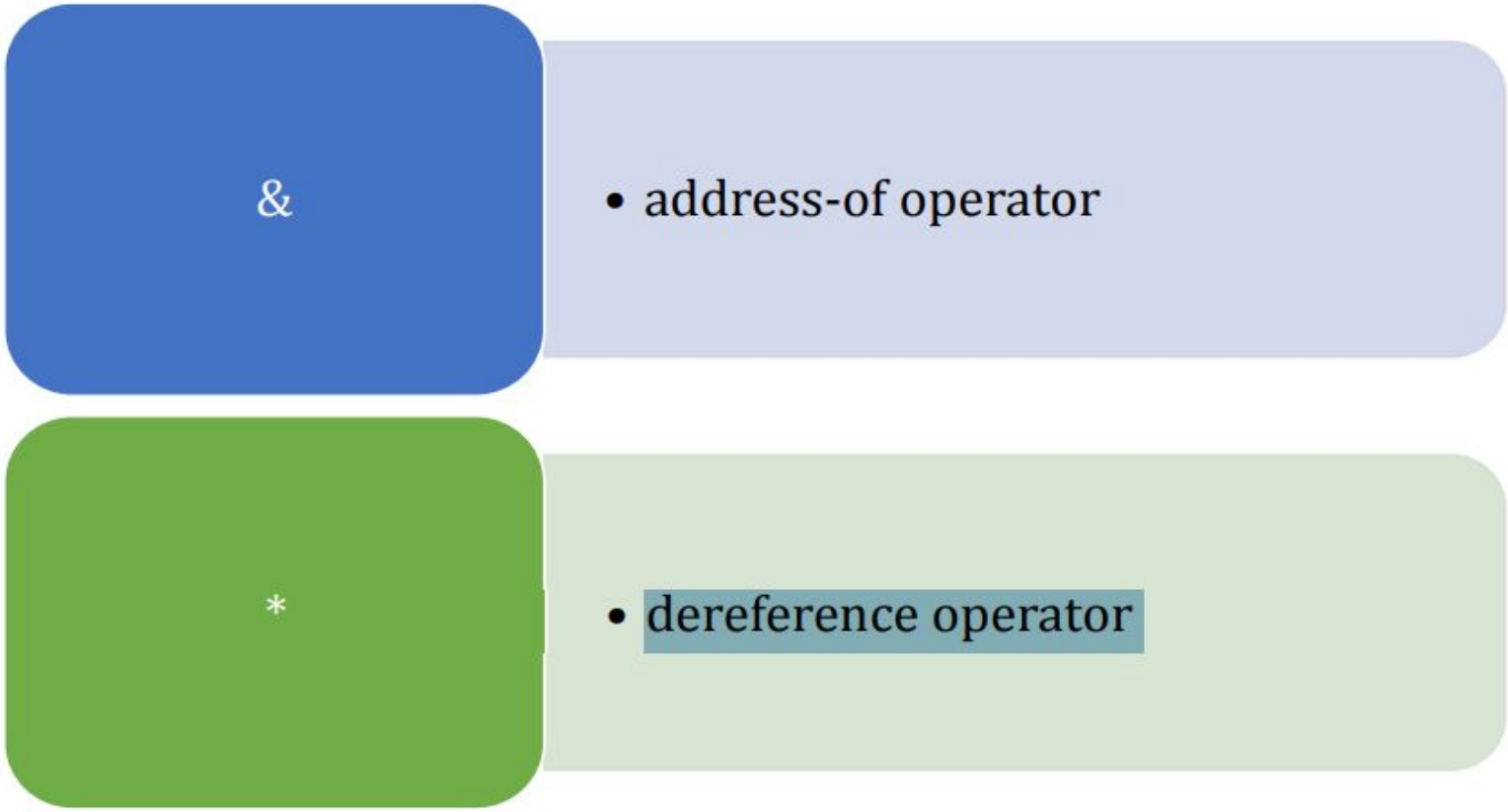
The NULL pointer is a constant with a value of zero defined in several standard libraries.

```
int *p=NULL;
```



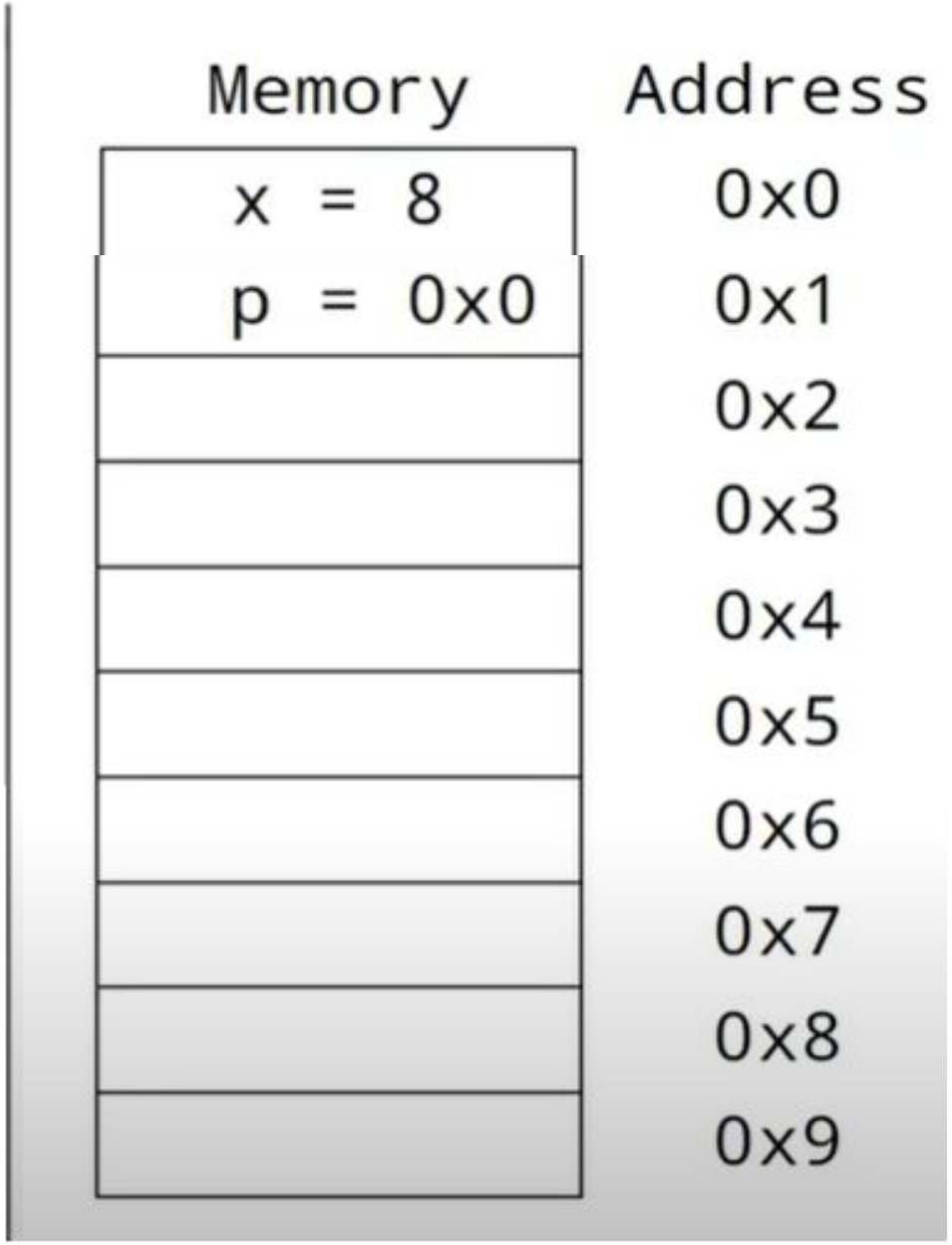
Accessing a variable through its pointer

Direct access



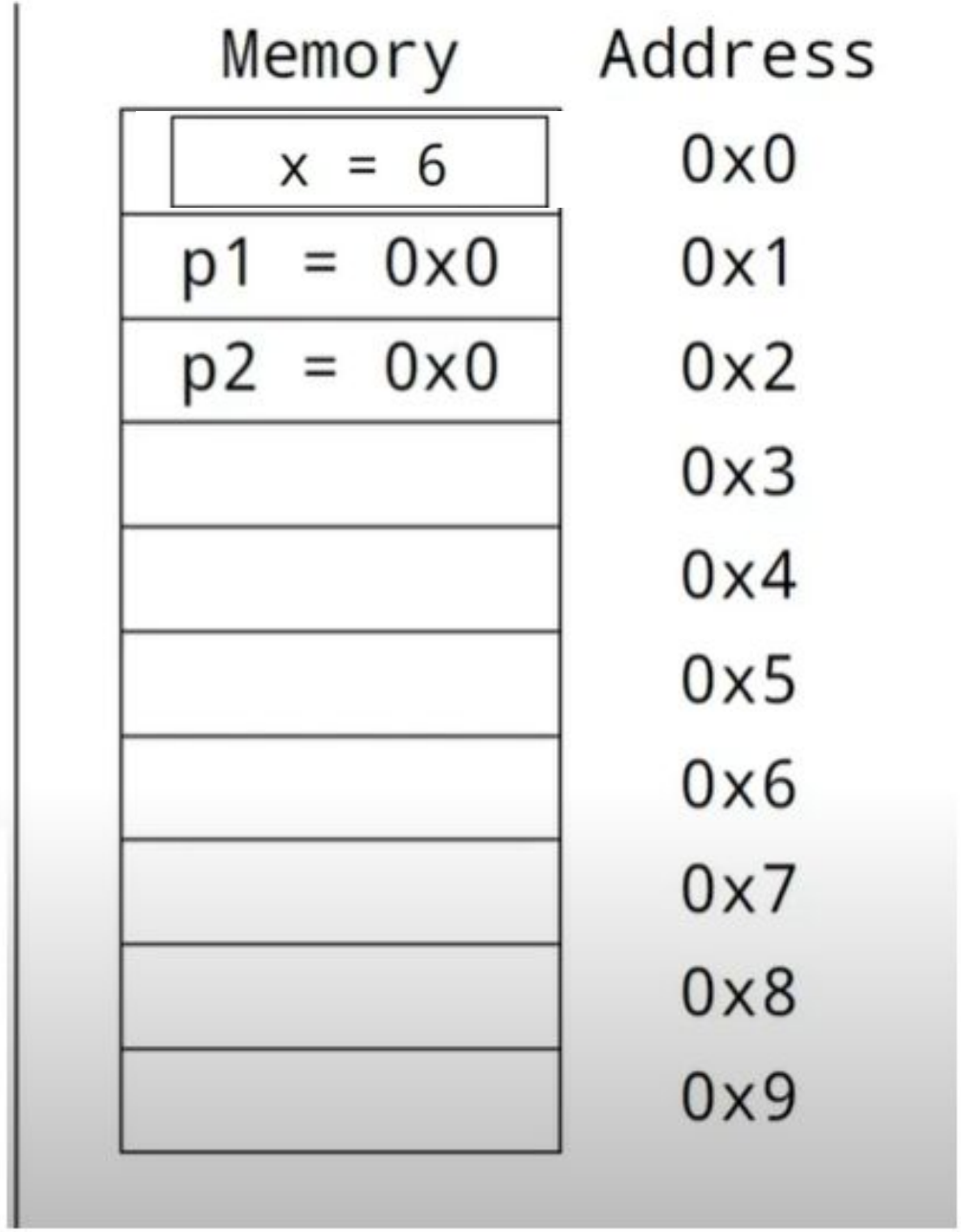
Dereference operator, also known as an indirection operator, operates on a pointer variable. It returns the location value

```c
int x;
int *p;

x = 5;

p = &x;

*p; /* same as x */

*p = 8; /* same as x = 8 */
```



```
int x;
int *p1, *p2;

x = 5;
p1 = &x;
p2 = &x;
```

```
*p1 = 2
*p2 = 6
```



Accessing a variable through its pointer

Once a pointer has been assigned the address of any variable, we can use the value of that variable and manipulate it as per the requirement.

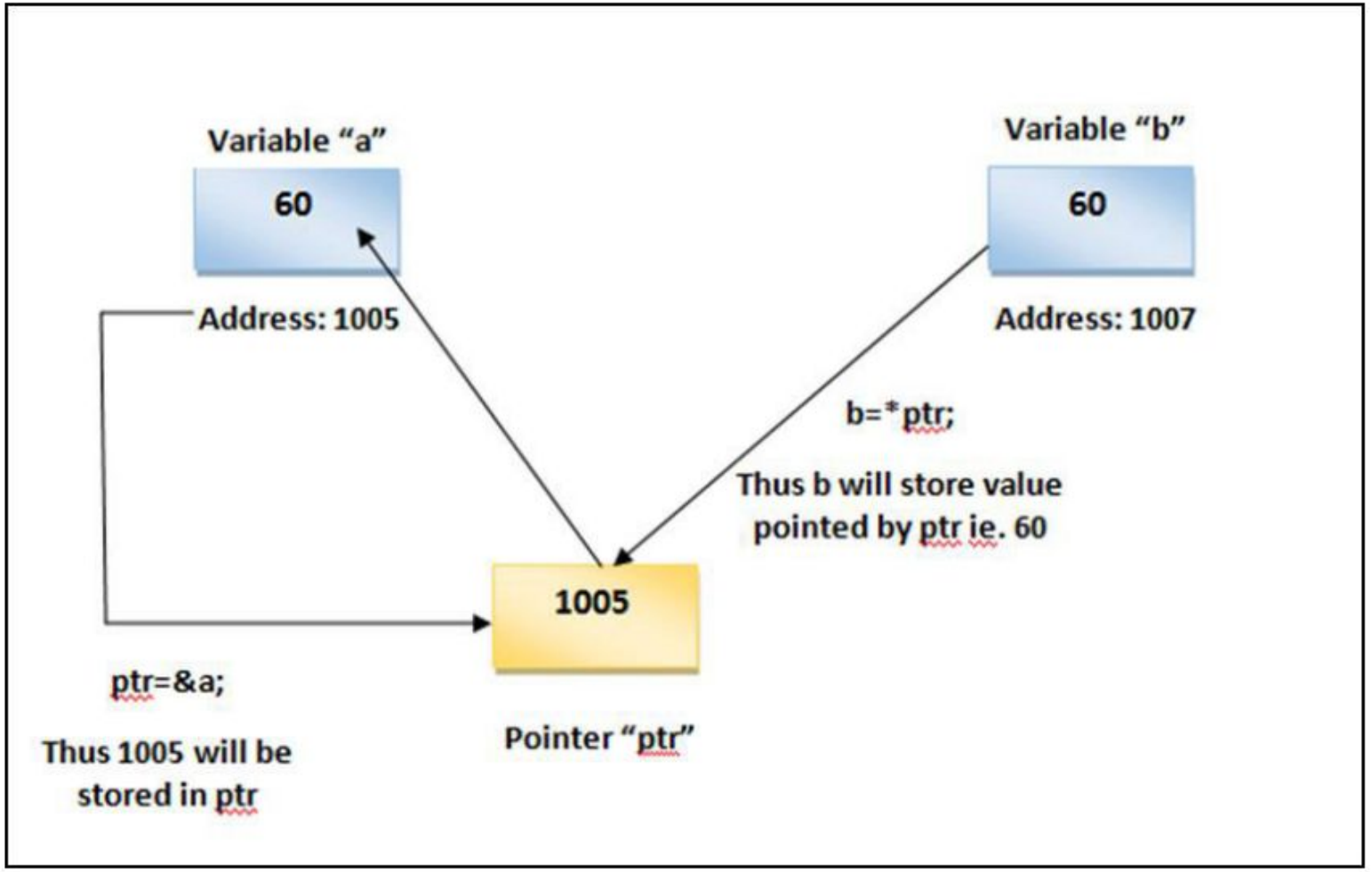
Accessing an object by its address is called indirect access.

The indirection operator, *, accesses an object of a specified type at an address.

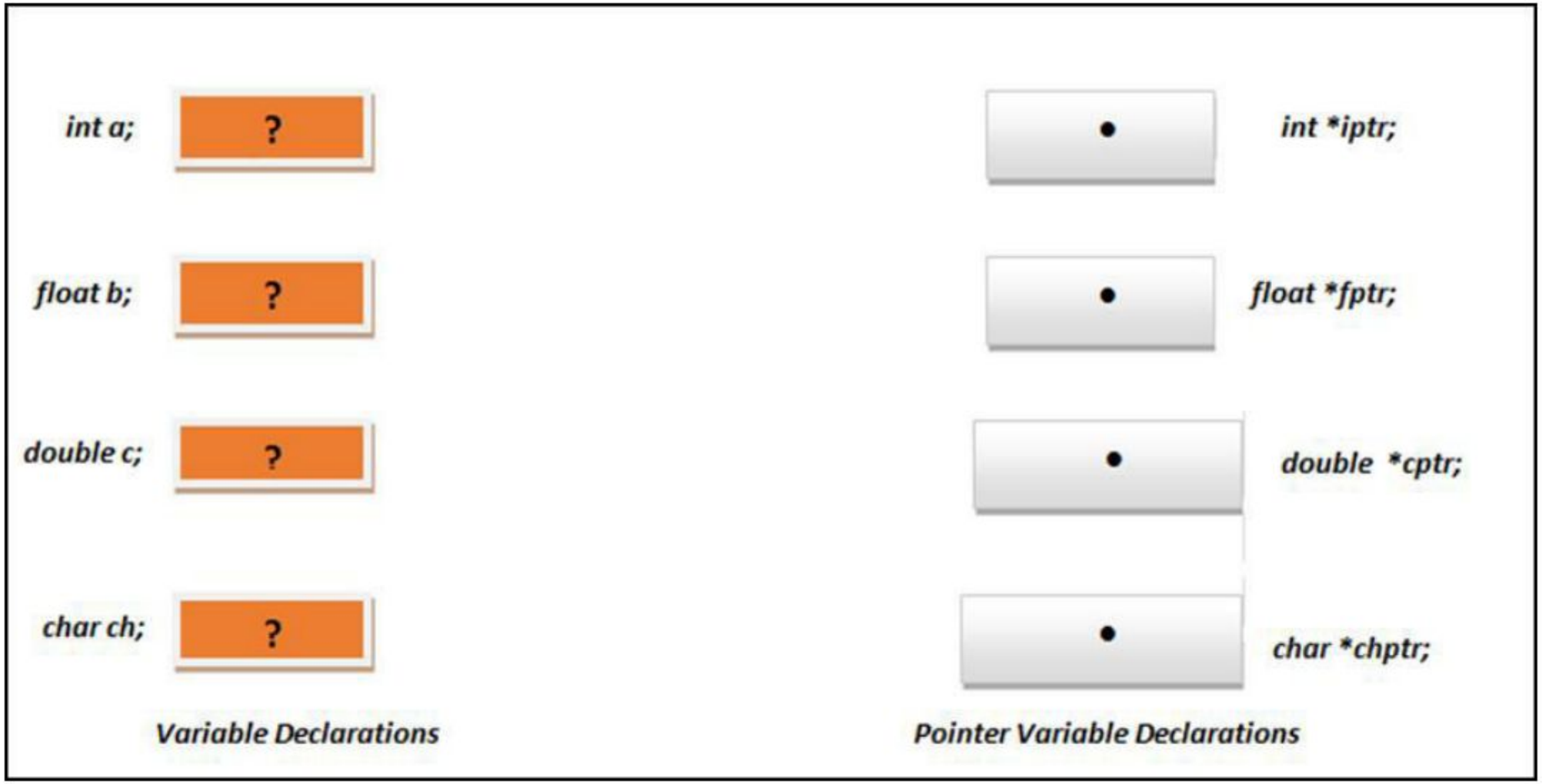
Example:

```
int a=60, b;
int *ptr;
ptr= &a;
b= *ptr;
```

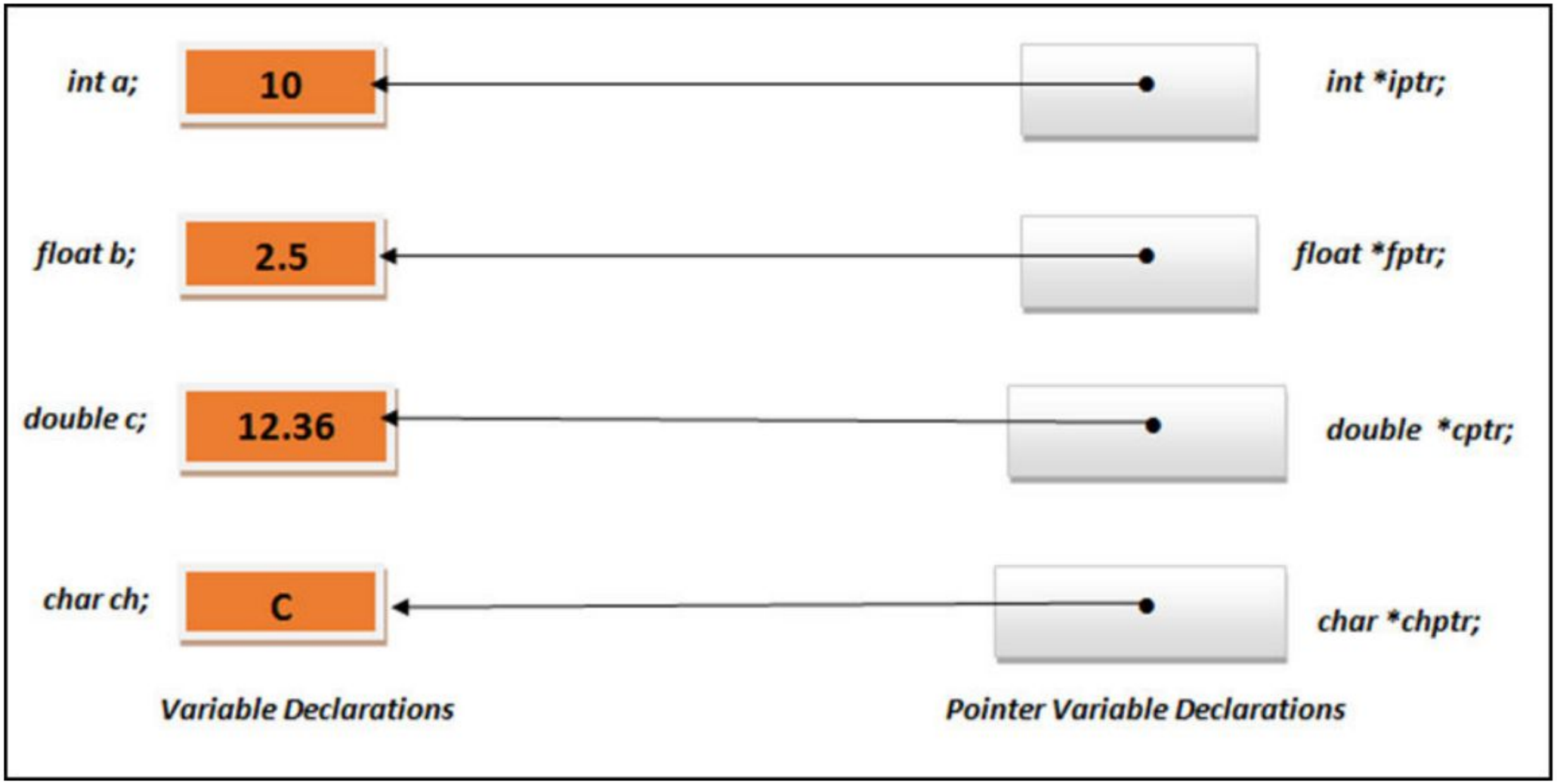
Accessing a variable through its pointer



Declaration of Pointer variables



Effect of indirect Access and Assignments of Pointers



Memory Allocation Functions

Memory Allocation Functions

Dynamic memory allocation is a process of allocating memory to the data during program execution.

- Dynamic memory allocation is necessary to manage available memory.
- Compile time memory management wastes the memory space.
- Memory allocation decisions are made during the run time.

These functions are defined in the **stdlib.h** header file.

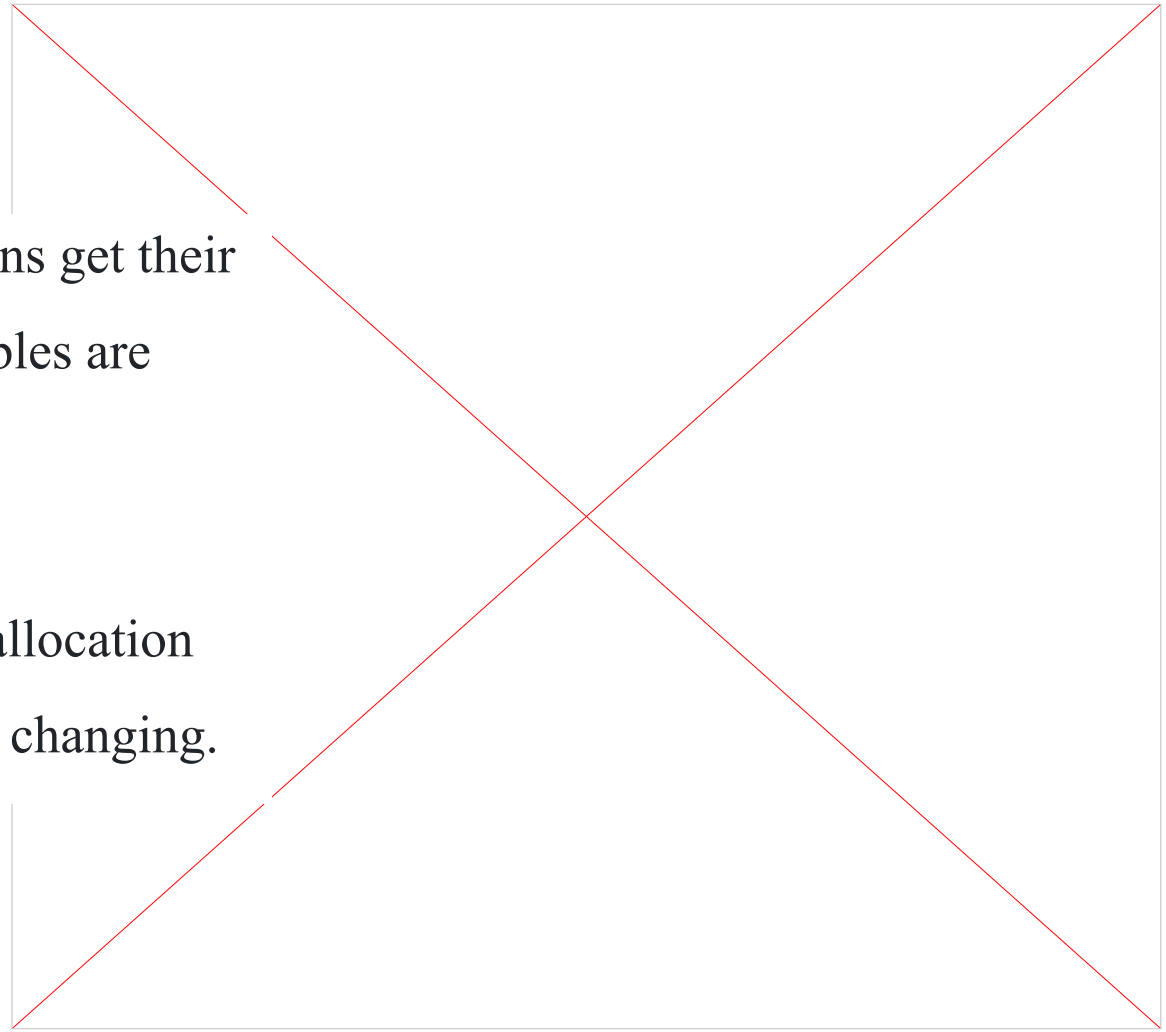
Types of Memory Allocation Functions

The <stdlib.h> contains four functions that can be used to manage dynamic memory

Function	Description
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	Deallocate the previously allocated space
realloc()	Change the size of previously allocated space

MEMORY ALLOCATION PROCESS

- **Global** variables, static variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**.
- The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keeps changing.



malloc() : Function **create a single block of memory of a specific size**.

Syntax:

```
void* malloc(size_t size)
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  // Dynamically allocated variable, sizeof(char) = 1 byte.
  char *ptr = (char * )malloc(sizeof(char));

  if (ptr == NULL) {
    printf("Memory Error!\n");
  } else {
    *ptr = 'S';
    printf("%c", *ptr);
  }

  return 0;
}
```

Output :

```
S
```

Let's take another example:

```c
#include<stdio.h>
#include <stdlib.h>
int main(){
    int *ptr;
    ptr = malloc(5 * sizeof(int)); /* a block of 5 integers */
```

calloc() Function

calloc() Assign **multiple blocks of memory to a single variable**, such as arrays and structures.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

DIFFERENCE BETWEEN MALLOC() AND CALLOC()

S.No.	malloc()	calloc()
1.	malloc() function **creates a single block of memory** of a specific size.	calloc() function **assigns multiple blocks of memory to a single variable**.
2.	The number of **arguments in malloc() is 1.**	The **number of arguments in calloc() is 2**.
3.	malloc() is **faster**.	calloc() is **slower.**
4.	malloc() has **high time efficiency**.	calloc() has **low time efficiency**.
5.	The memory block allocated by malloc() **has a garbage value**.	The **memory block allocated by calloc() is initialized by zero.**
6.	malloc() **indicates memory allocation**.	calloc() **indicates contiguous allocation**.

realloc() Function

The realloc() function is used to change the memory size that is already allocated dynamically to a variable

If we want to change the size of memory allocated by malloc() or calloc() function, we use realloc() function.

Without losing the old data, it changes the size of the memory block

Time for an Example: realloc() function

Let's see, how we can use this function.

```
int *x;
x = (int*)malloc(50 * sizeof(int));
x = (int*)realloc(x, 100);   //allocated a new memory to variable x
```

free() Function

Let's start by knowing the syntax of this function:

```
void free(void *p);
```

Example

```
int *x;
x = (int*)malloc(50 * sizeof(int));
free(x);
```

Recursion and its Advantages

Recursion

Recursion can be defined as defining anything in terms of itself. It can be also defined as repeating items in a self-similar way.

A recursive call is a call made to a function from within the same function.

Example:

```
fact(n):
if n == 0: return 1
else: return n * fact(n-1)
```

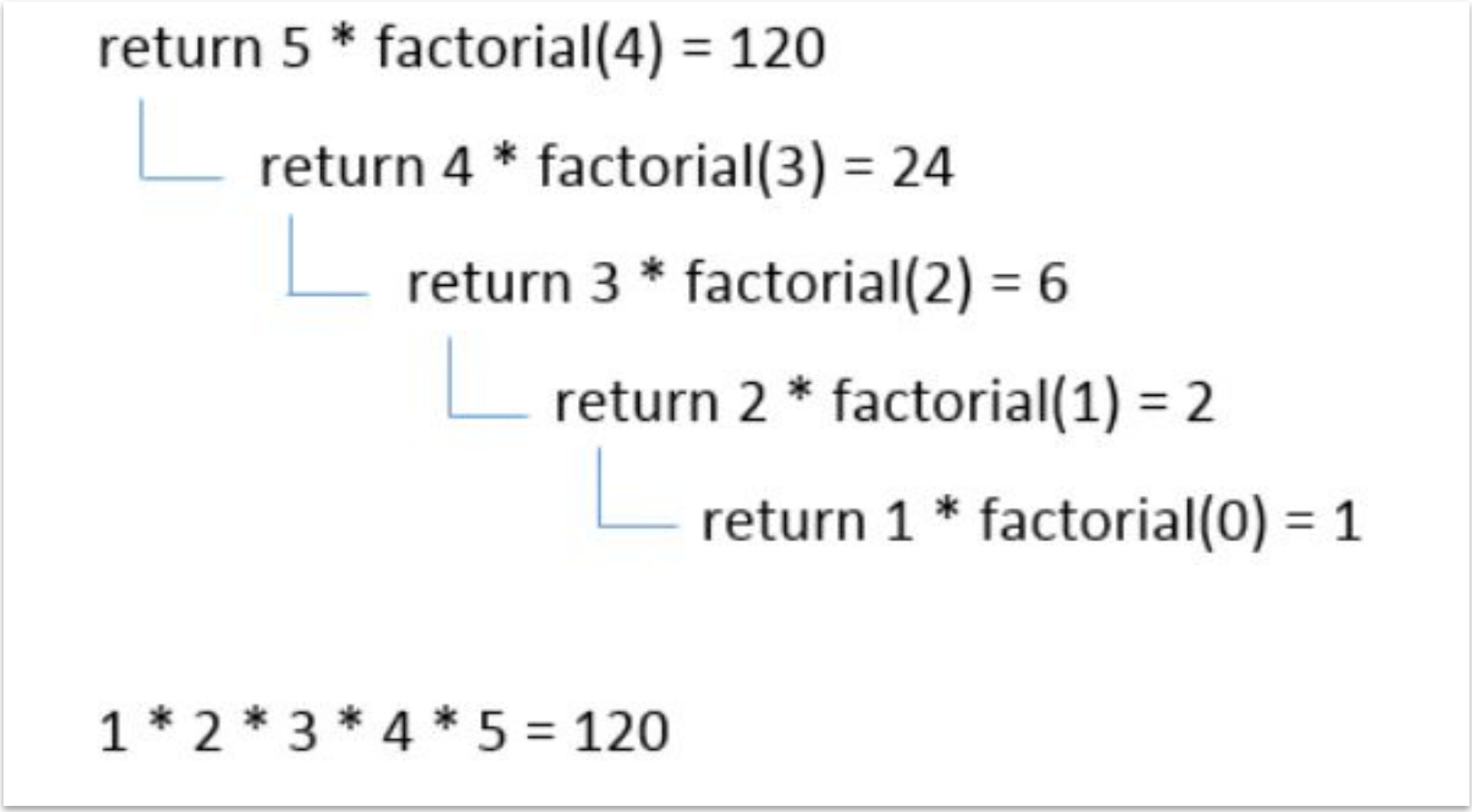
Any function **which calls itself is called recursive function**, and **such function calls are called recursive calls**. Recursion involves several numbers of recursive calls.

```c
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want t
o calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
```

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

Output

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```



Advantages of Recursion

1. Reduce unnecessary calling of function.

2. Easier to understand.

3. Solving problems becomes easy when its iterative solution is very big and complex and cannot be implemented with loops.

3. Extremely useful when applying the same solution

Recursive Programs:

1. **Fibonacci series**

To generate a Fibonacci series like 0,1, 1, 2, 3, 5, 8, 13, 21and so on.

2. **Binomial Coefficient**

Binomial coefficient C (n, k) counts the number of ways to form an unordered collection of k items selected from a collection of n distinct items

3. **GCD (Greatest Common Divisor)**

The Greatest Common divisor of two or more integers is the largest positive integer that divides the numbers without a remainder.

Thank you