MODULE-4

FUNCTIONS and POINTERS

Contents

Pointers: Definition, Initialization
Pointers arithmetic
Pointers & Arrays and Dynamic memory allocation.

Functions: Prototype declaration
Function definition
Function call
Types of functions
Difference between built-in and user-defined functions.

POINTERS

Introduction to pointers

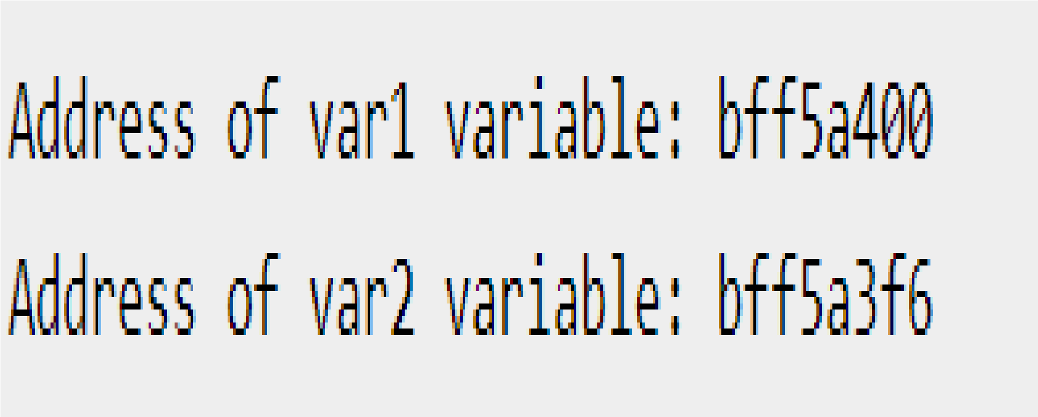
- The pointers in C language refer to the **variables that hold the addresses of different variables of similar data types**.

- We **use pointers to access the memory** of the said variable and then manipulate their addresses in a program.

- Every variable is a memory location and every memory location has its address defined which can be accessed using **ampersand (&) operator**, which denotes an address in memory.

Introduction to pointers

```
#include <stdio.h>
Void main ()
{
 int var1;
 char var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
 }
```



```
Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6
```

Pointer Declaration

Declaration of a pointer is done before using it to store any variable address. The general form of a pointer variable declaration is −

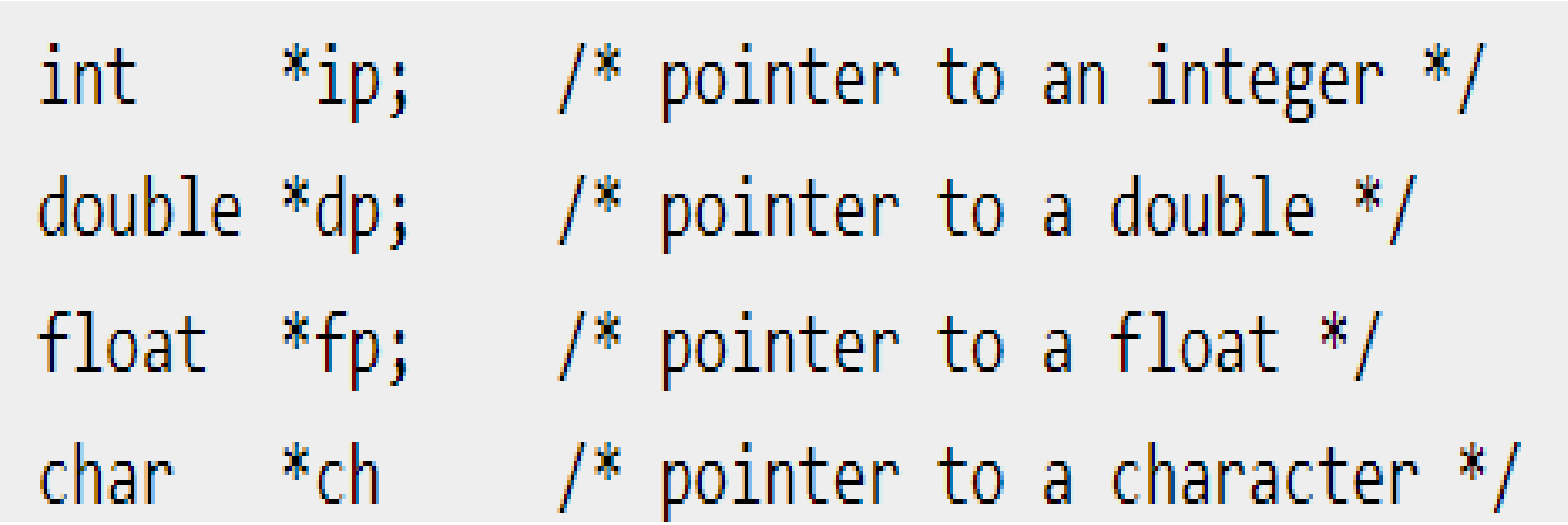
```
type *var-name;
```

type is the **pointer's base type**; it must be a valid **C data type** and **var-name** is the name of the pointer variable.

The **asterisk *** used to **declare a pointer** is the same asterisk used for multiplication. However, in this statement the **asterisk is being used to designate a variable as a pointer**.

Pointer Declaration

```
type *var-name;
```



```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Pointer Declaration

- Declaration and Initialization of pointers :- The operators used to represent pointers are
  - Address Operator (&)
  - Indirection Operator (*)

- Syntax :-

ptr_data_type *ptr_var_name;

ptr_var_name = &var_name;

- where var_name is a variable whose address is to be stored in the pointer.

Pointer Declaration

- Example :-

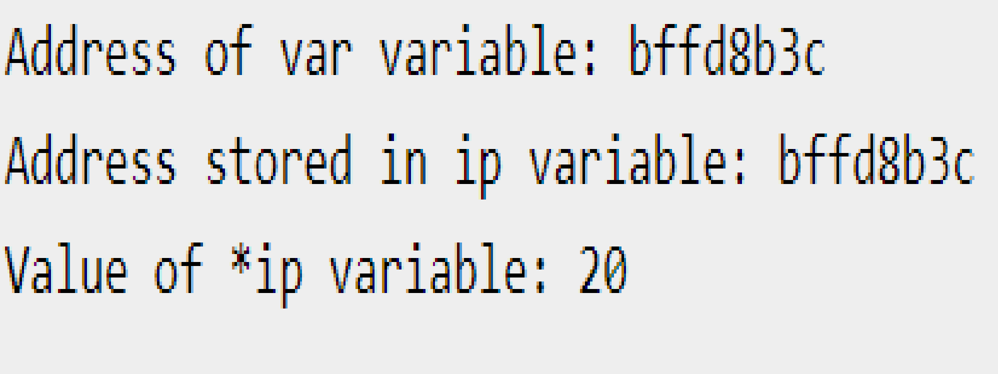
```
int a=10;
int *ptr;
then
        ptr = &a;
        *ptr = a;
```

ptr is a pointer holding the address of variable ‘a’

*ptr holds the value of the variable a.

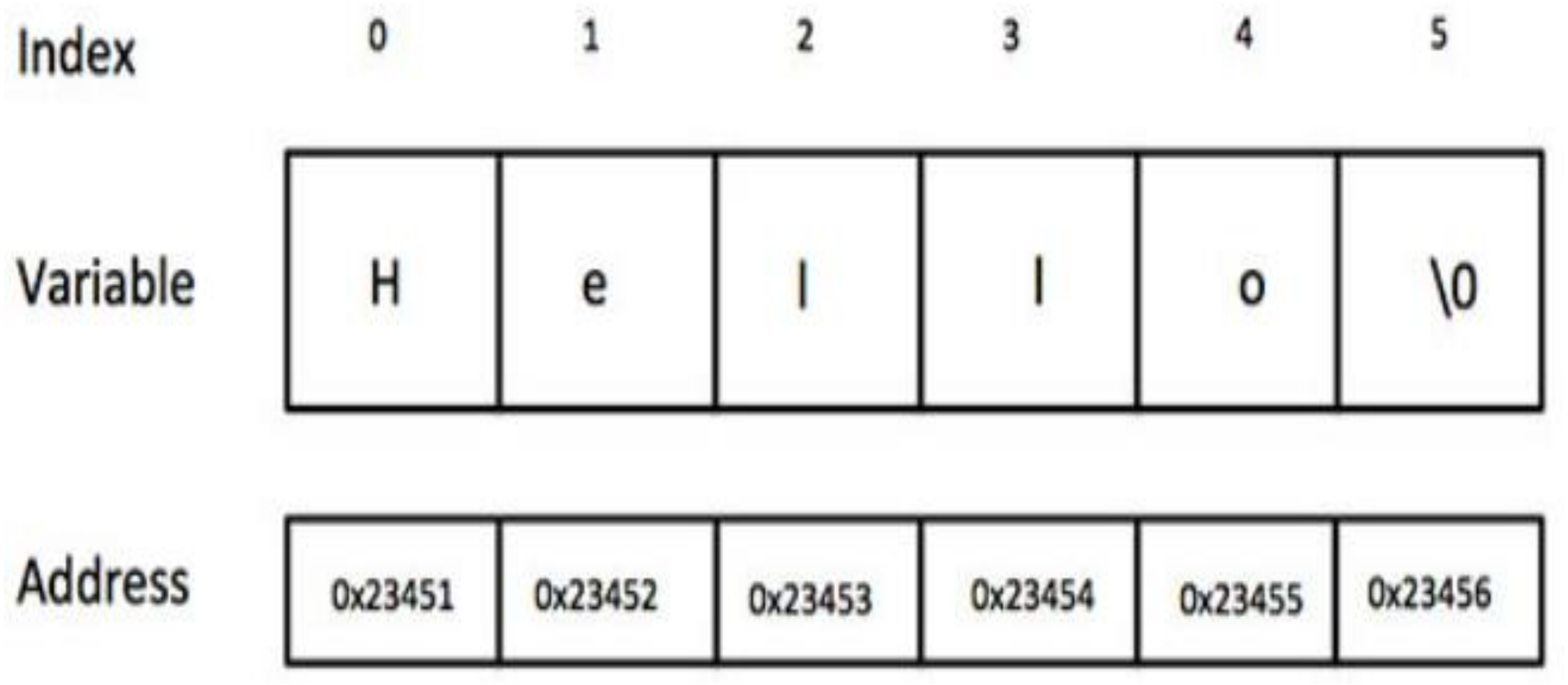
Pointer Declaration

```
#include <stdio.h>
void main ()
{
int var = 20;      /* actual variable declaration */
 int *ip;          /* pointer variable declaration */
ip = &var;       /* store address of var in pointer variable*/
printf("Address of var variable: %x\n", &var );
                    /* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );
                    /* access the value using the pointer */
}
```



```
Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of *ip variable: 20
```

Pointer Declaration



Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Pointer Types

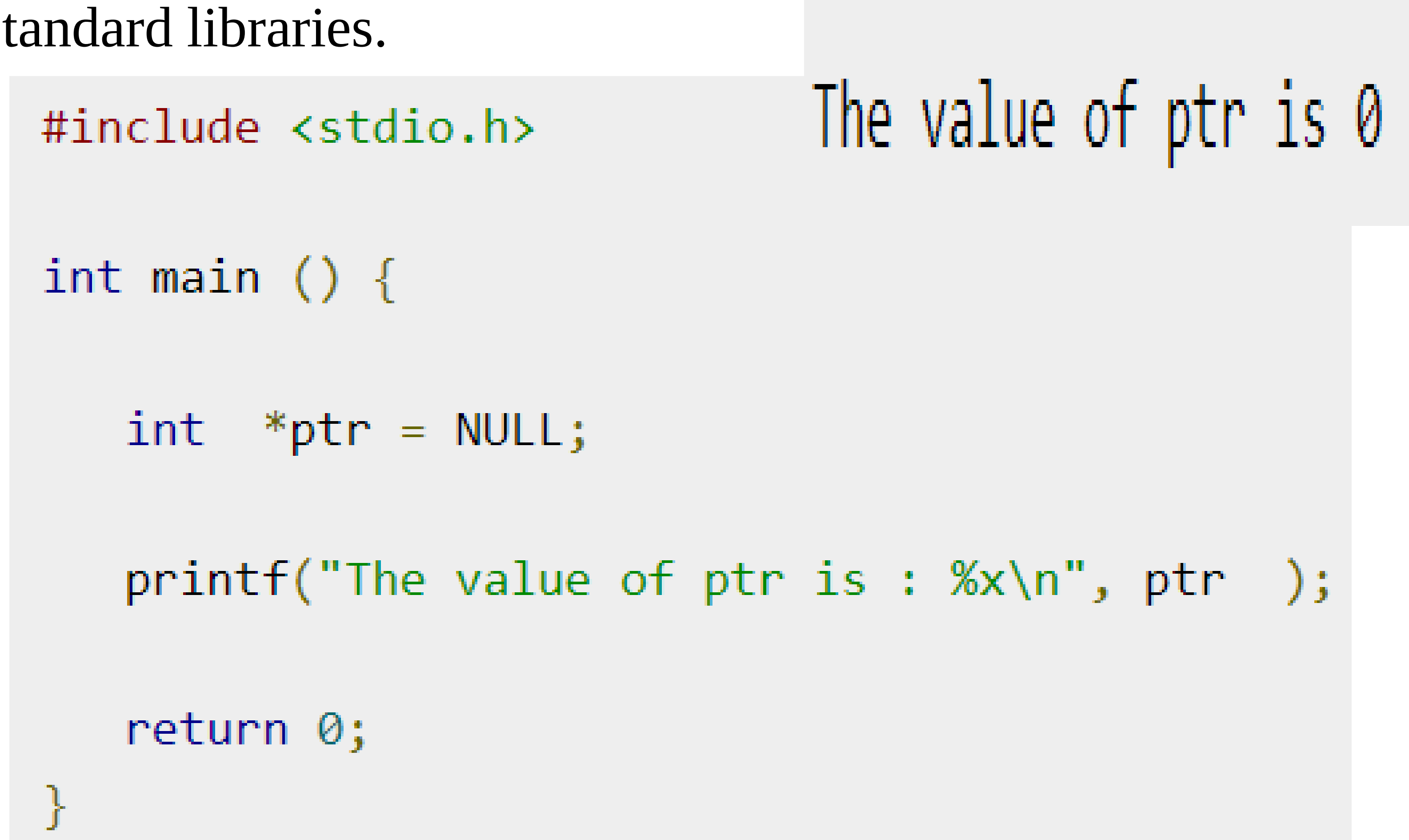
There are eight different types of pointers which are as follows −

- Null pointer
- Void pointer
- Wild pointer
- Dangling pointer
- Complex pointer
- Near pointer
- Far pointer
- Huge pointer

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.



```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

```
The value of ptr is 0
```

Generic Pointers(Void pointer)

- When a variable is declared as being a pointer to type **void,** it is known as a ***generic pointer*.**

- Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn’t have any specific type.

- If we **assign address of char data type to void pointer** it will become **char**

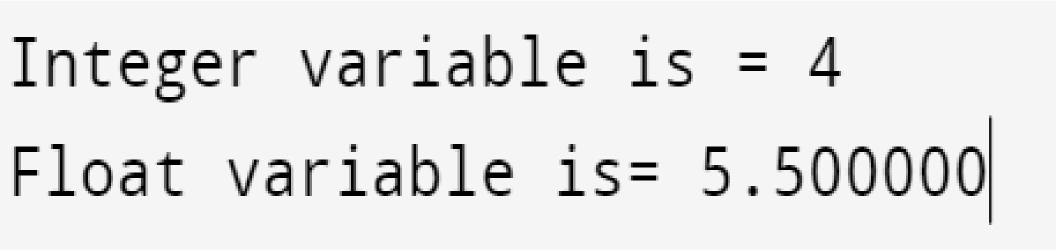
Generic Pointers(Void pointer)

- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as an integer pointer, character pointer.

Declaration : **void * pointer_name;**

Generic Pointers(Void pointer)

```
#include<stdio.h>

void main() {

 int x = 4; float y = 5.5; //A void pointer

 void *ptr;

ptr = &x;

printf(″Integer variable is = %d″, *( (int*) ptr) ); //type

ptr = &y;

printf(″¥nFloat variable is= %f″, *( (float*) ptr) );

}
```



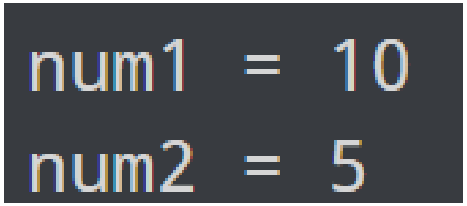
```
Integer variable is = 4
Float variable is= 5.500000
```

Passing Arguments to function using pointer

When we **pass a pointer** as **an argument instead of a variable then the address** of the variable is passed instead of the value. So any **change made** by the **function** using the **pointer** is **permanently made at the address of passed variable**.

Passing Arguments to function using pointer

```
#include <stdio.h>
void swap(int *n1, int *n2);
void main()
{
   int num1 = 5, num2 = 10;          // address of num1 and
num2 is passed
   swap( &num1, &num2);
   printf("num1 = %d\n", num1);
   printf("num2 = %d", num2);
}
void swap(int* n1, int* n2)
{
   int temp;
   temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
```



```
num1 = 10
num2 = 5
```

Pointer Expressions and Arithmetic Pointer

Pointer Expressions

Expressions involving **pointers conform to the same rules as other expressions**. Expressions in C programing language combine **operands, operators, and variables**. The operator denotes the action or operation to be performed.

Arithmetic Pointer

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer.

It is **a method of calculating the address of an object with the help of arithmetic operations on pointers and use of pointers in comparison operations**.

Pointer Arithmetic in C

- We can perform arithmetic operations on the pointers like addition, subtraction, etc.

- However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.

- In pointer-from-pointer subtraction, the result will be an integer value.

- Following arithmetic operations are possible on the pointer in C language:

  Increment , Decrement , Addition , Subtraction Comparison

Incrementing Pointer in C:-

- If we increment a pointer by 1, the pointer will startpointing to the immediate next location.

- This is somewhat different from the general arithmetic since the value of the pointer will get increased by the sizeof the data type to which the pointer is pointing.

The Rule to increment the pointer is given below:

new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

1. Increment/Decrement of a Pointer

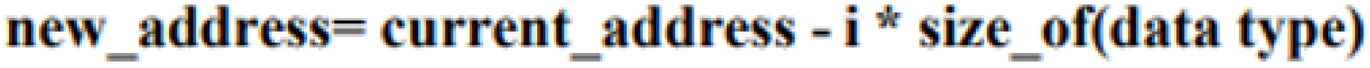
Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

new_address= current_address + i * size_of(data type)

For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**

Decrement a Pointer: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

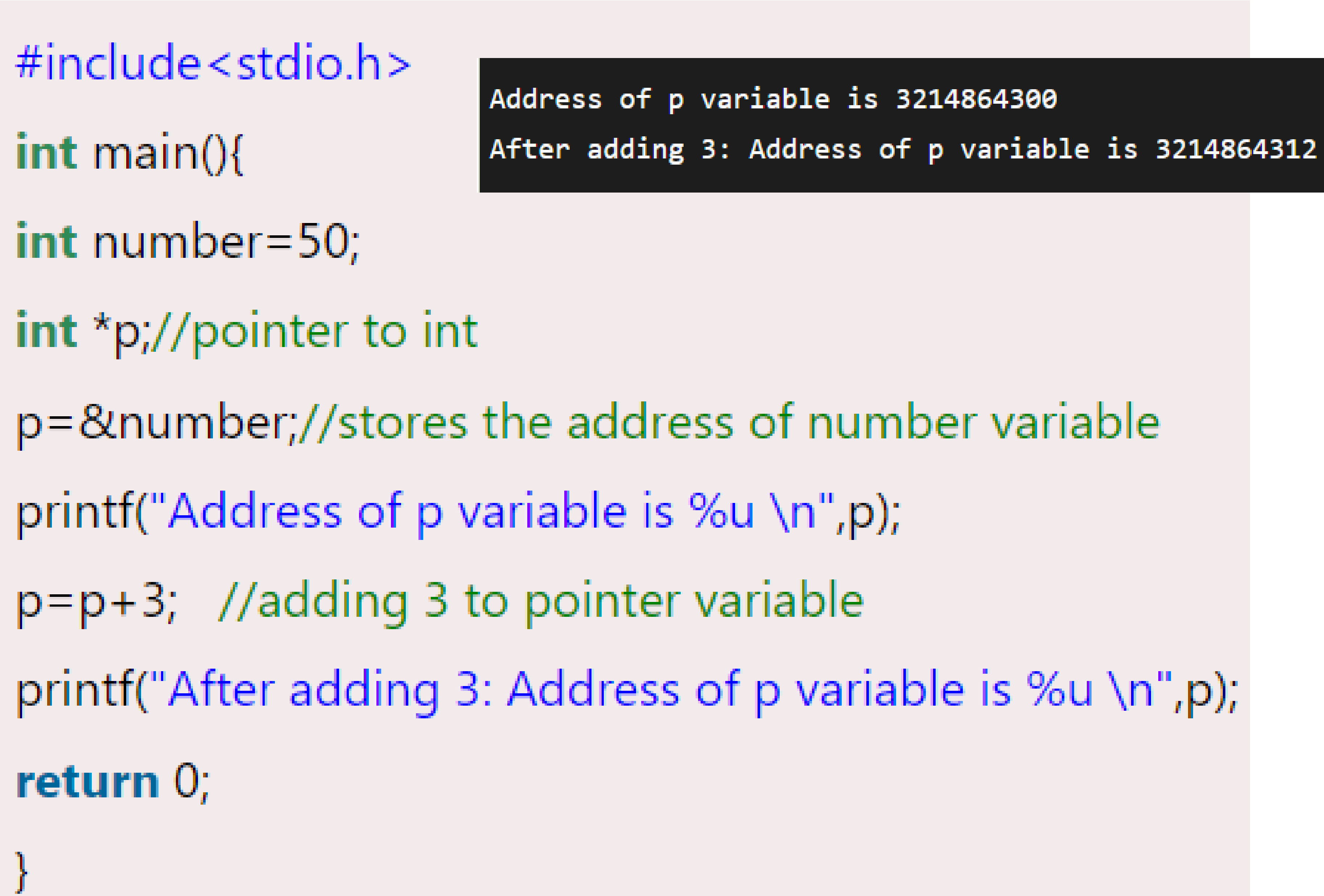


new_address= current_address - i * size_of(data type)

ForExample:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**.

Pointer Expressions and Arithmetic Pointer

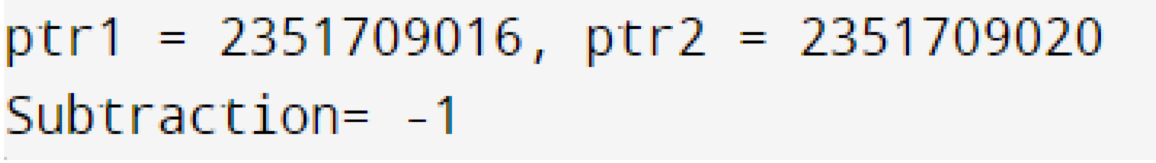


```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

Pointer Expressions and Arithmetic Pointer

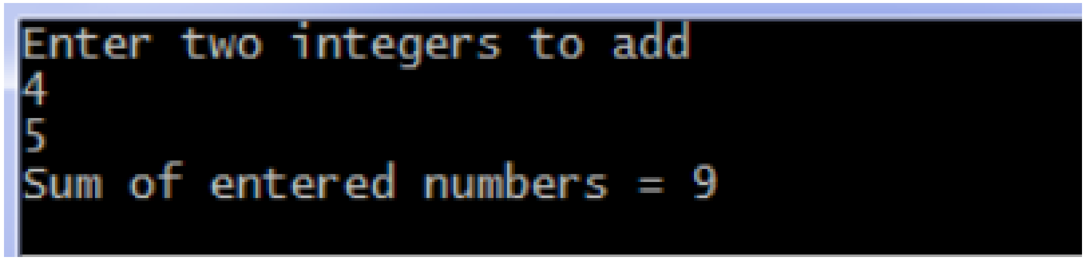
```
#include <stdio.h>
void main()
{
    int x = 6;
    int N = 4;
   int *ptr1, *ptr2;
    ptr1 = &N; // stores address of N
    ptr2 = &x; // stores address of x
    printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
     x = ptr1 - ptr2;
    printf("Subtraction= %d\n", x);
 }
```



```
ptr1 = 2351709016, ptr2 = 2351709020
Subtraction= -1
```

Pointer Expressions and Arithmetic Pointer

```
#include <stdio.h>
void main()
{
   int first, second, *p, *q, sum;
   printf("Enter two integers to add\n");
   scanf("%d%d", &first, &second);
   p = &first;
   q = &second;
   sum = *p + *q;
   printf("Sum of the numbers=%d\n",sum);
}
```

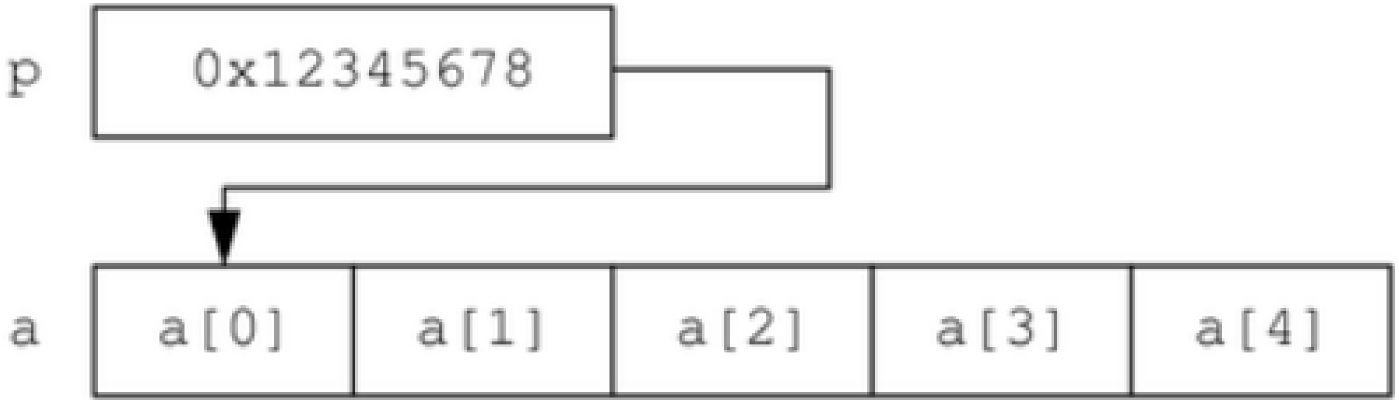


```
Enter two integers to add
4
5
Sum of entered numbers = 9
```

Pointers and Arrays

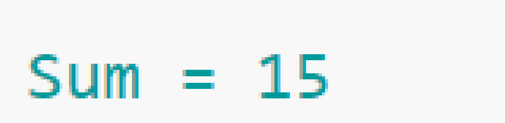
Array name ≈ a pointer to the initial (0th) array element

An array is represented by a variable that is associated with the address of its first storage location. A pointer is also the address of a storage location with a defined type, so it is allowed to use of the array [ ] index notation with both pointer variables and array variables.



Pointers and Arrays

```
#include <stdio.h>

 #define N 5

int main()

 {

int i, * ptr, sum = 0;

int nums[N] = {1, 2, 3, 4, 5};

 for (ptr = nums; ptr < & nums[N]; ++ptr)

 sum += * ptr;

printf("Sum = %d ", sum);

 }
```

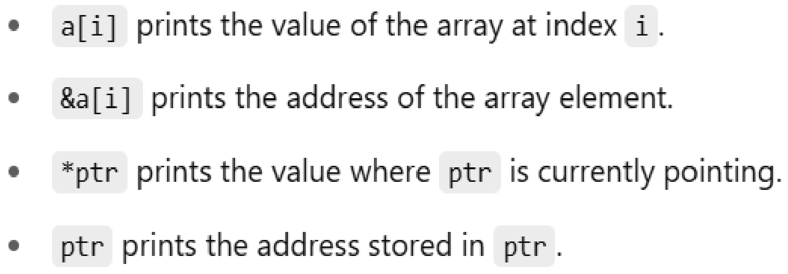


```
Sum = 15
```

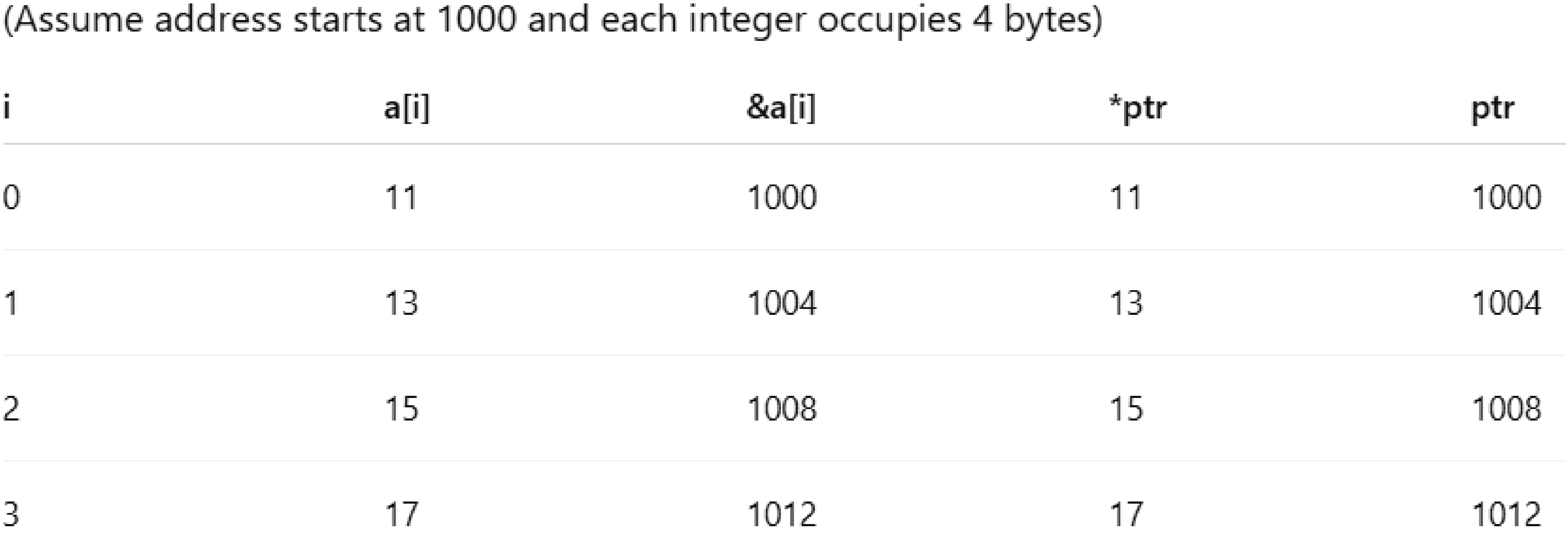
Pointers and Arrays

/* c program to demonstrate arrays with pointers */

```
#include<stdio.h>
void main()
{                    int a[10]={11,13,15,17};
                     int *ptr;
                     int i;
                     ptr=a;
                     for(i=0;i<4;i++)
                     {
                     printf("%d\t",a[i]);
                     printf("%d\n",&a[i]);
                     printf("%d\t",*ptr);

                     printf("%d\n",ptr);

                     ptr++;
                     }
}
```



- `a[i]` prints the value of the array at index `i`.
- `&a[i]` prints the address of the array element.
- `*ptr` prints the value where `ptr` is currently pointing.
- `ptr` prints the address stored in `ptr`.



(Assume address starts at 1000 and each integer occupies 4 bytes)

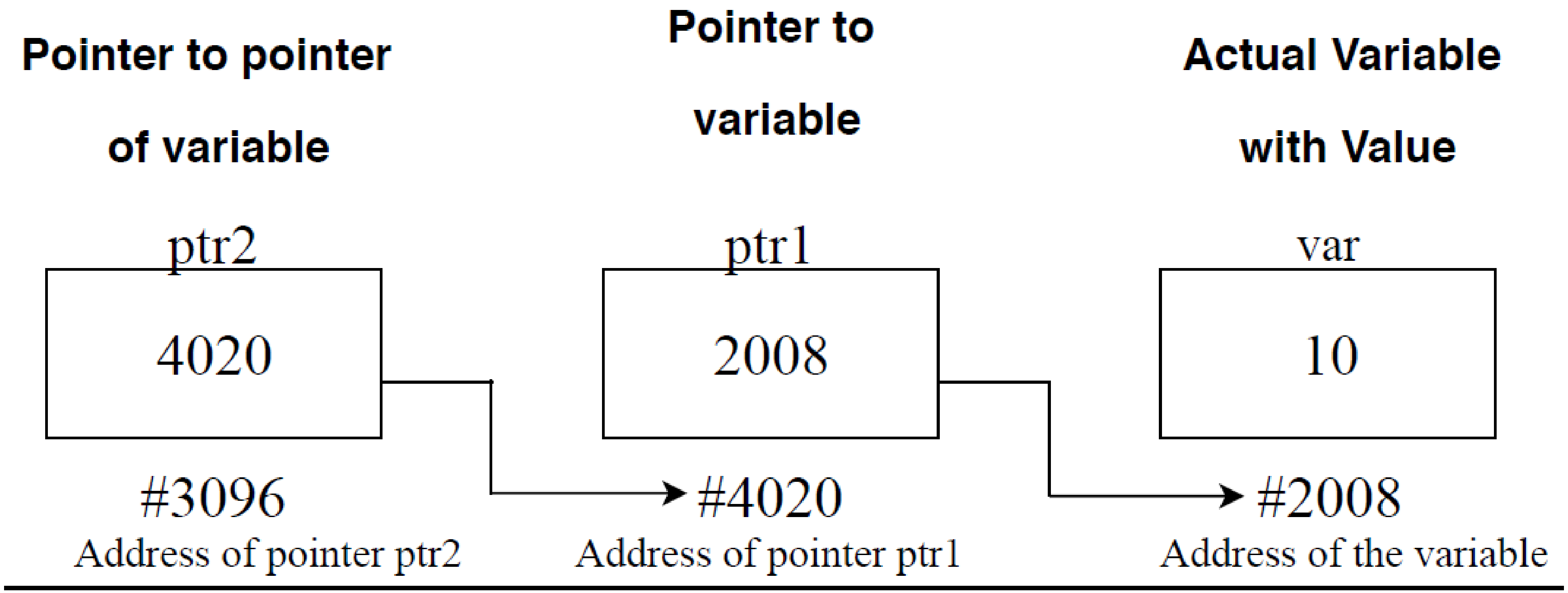
i	a[i]	&a[i]	*ptr	ptr
0	11	1000	11	1000
1	13	1004	13	1004
2	15	1008	15	1008
3	17	1012	17	1012

Double Pointer: When a pointer holds the address of another pointer then such type of pointer is known as **pointer-to-pointer** or **double pointer**.

- Here the **first pointer** is used to store the address of the variable
- The **second pointer** is used to store the address of the first pointer.

Declaration of double pointer :

Syntax: **datatype **ptr;**



Pointer to pointer of variable

Pointer to variable

Actual Variable with Value

ptr2

4020

#3096

Address of pointer ptr2

ptr1

2008

#4020

Address of pointer ptr1

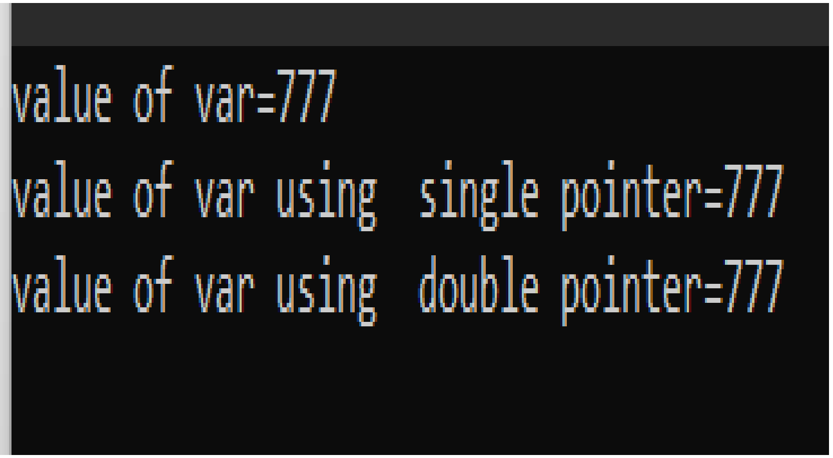
var

10

#2008

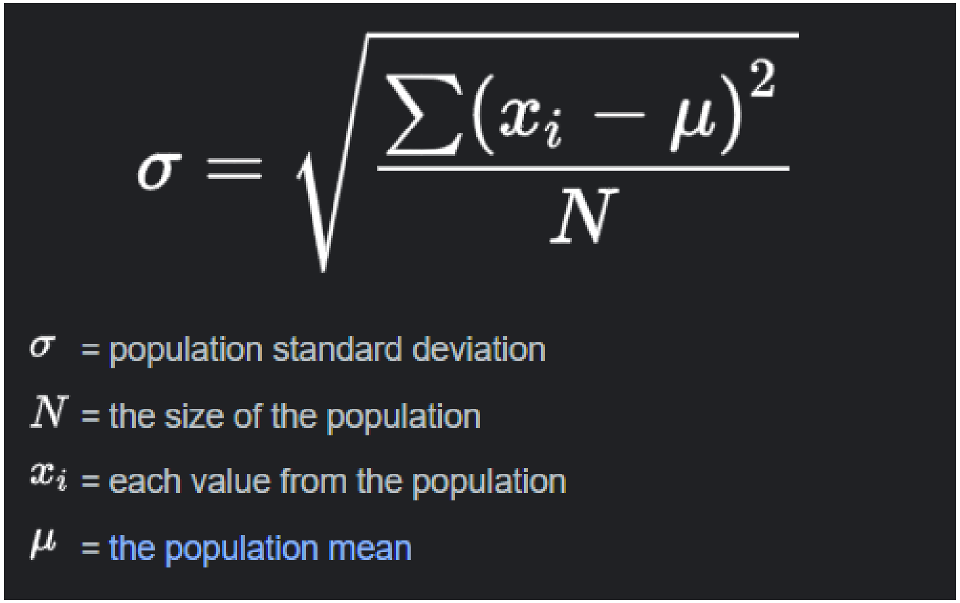
Address of the variable

```
#include<stdio.h>
void main()
{
        int var=777;
        int *ptr2;
        int ** ptr1;
        ptr2=&var;
        ptr1=&ptr2;
        printf(“value of var=%d\n”,var);
        printf(“value of var using  single pointer=%d\n”,*ptr2);
        printf(“value of var using  double pointer=%d\n”,**ptr1);
}
```



```
value of var=777
value of var using  single pointer=777
value of var using  double pointer=777
```

Develop a program using pointers to compute the sum, mean and standard deviation of all elements stored in an array of n real numbers.



$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

σ = population standard deviation

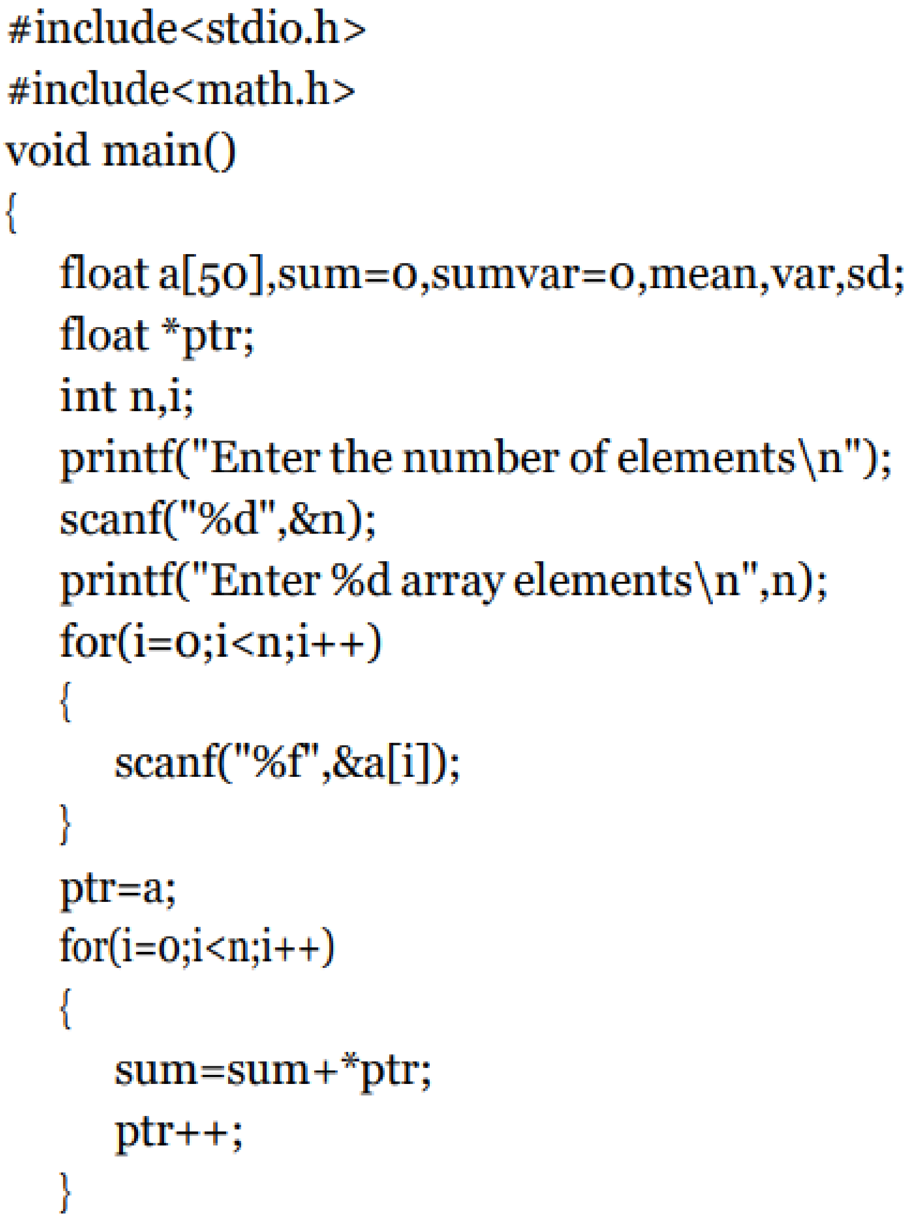
N = the size of the population

x_i = each value from the population

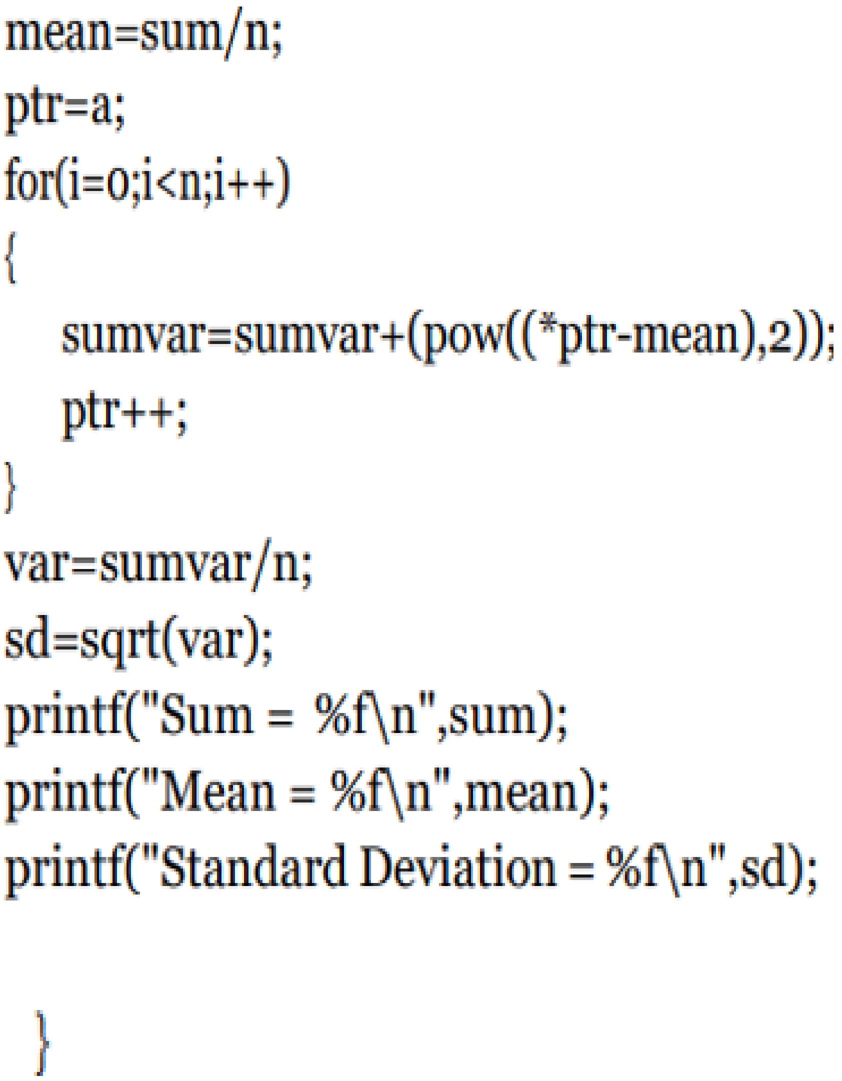
μ = the population mean

Step 1: Find the mean.
Step 2: For each data point, find the square of its distance to the mean.
Step 3: Sum the values from Step 2.
Step 4: Divide by the number of data points.

Develop a program using pointers to compute the sum, mean and standard deviation of all elements stored in an array of n real numbers.

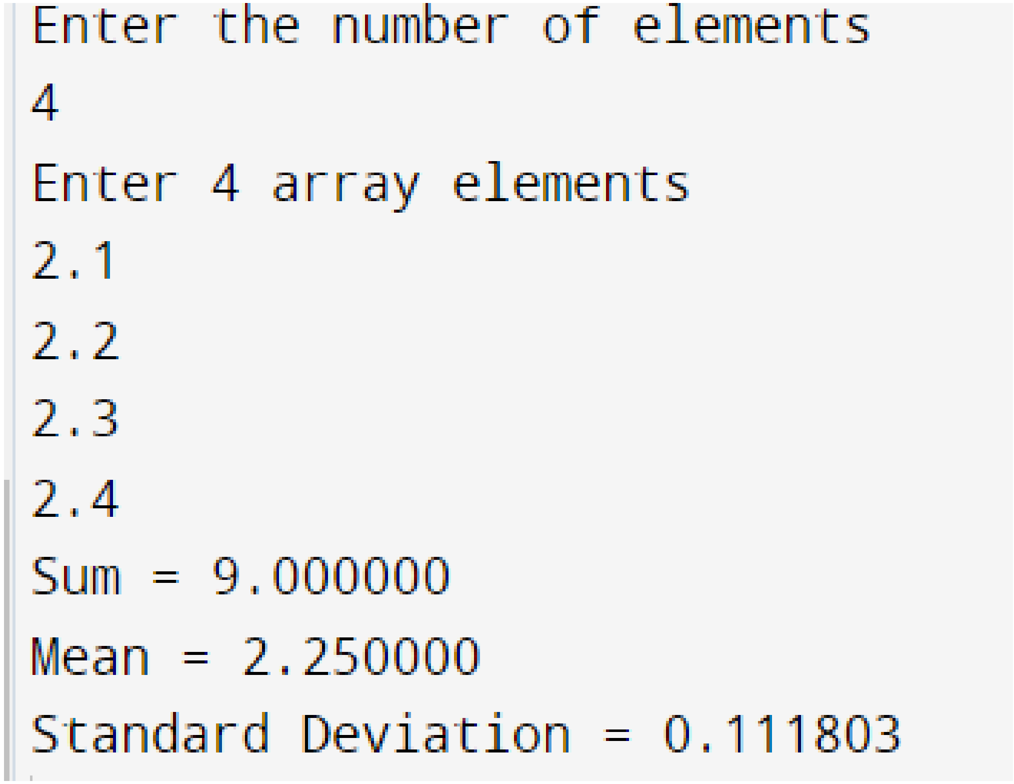


```
#include<stdio.h>
#include<math.h>
void main()
{
    float a[50],sum=0,sumvar=0,mean,var,sd;
    float *ptr;
    int n,i;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter %d array elements\n",n);
    for(i=0;i<n;i++)
    {
        scanf("%f",&a[i]);
    }
    ptr=a;
    for(i=0;i<n;i++)
    {
        sum=sum+*ptr;
        ptr++;
    }
```

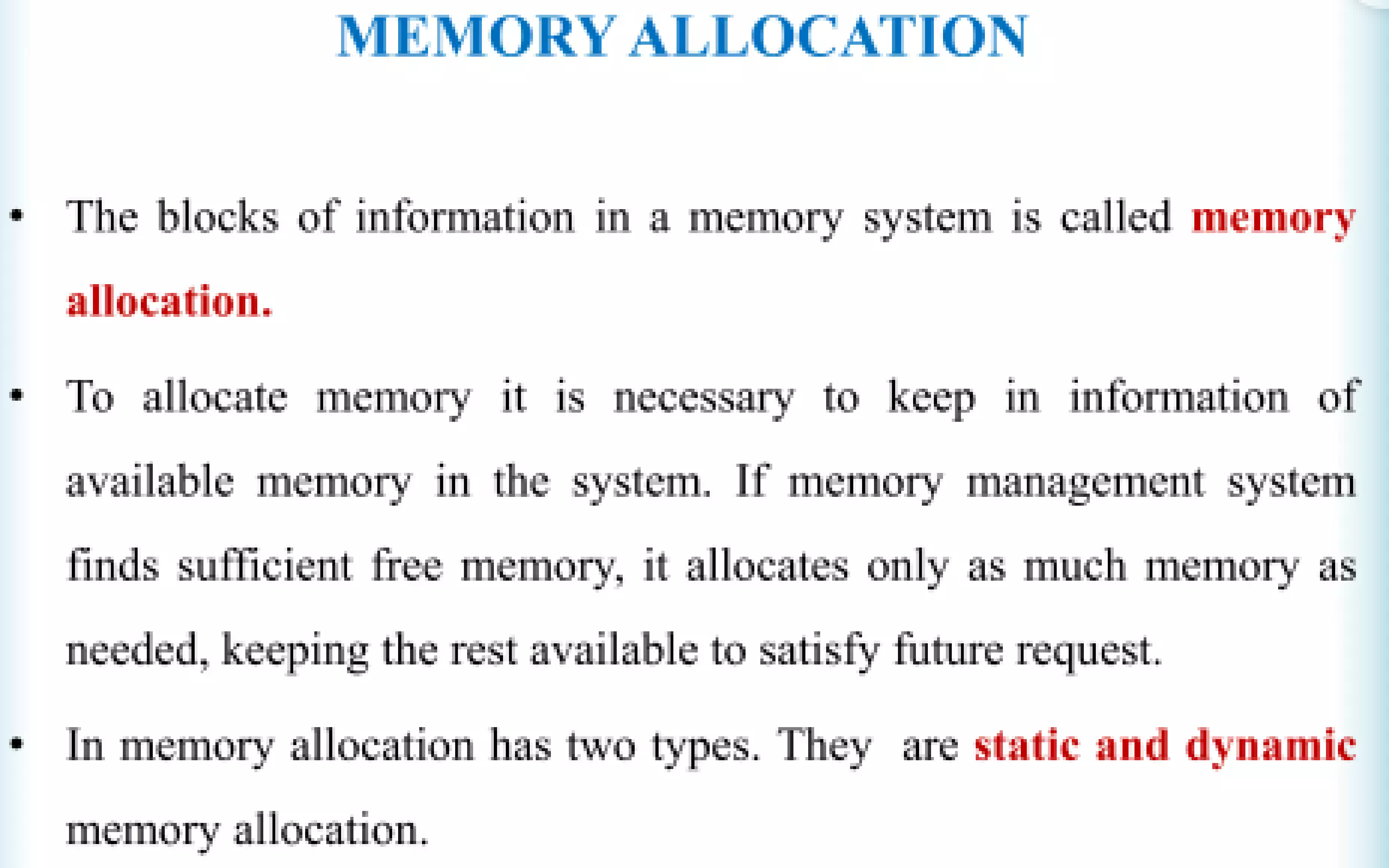


```
mean=sum/n;
ptr=a;
for(i=0;i<n;i++)
{
    sumvar=sumvar+(pow((*ptr-mean),2));
    ptr++;
}
var=sumvar/n;
sd=sqrt(var);
printf("Sum = %f\n",sum);
printf("Mean = %f\n",mean);
printf("Standard Deviation = %f\n",sd);


 }
```

Develop a program using pointers to compute the sum, mean and standard deviation of all elements stored in an array of n real numbers.

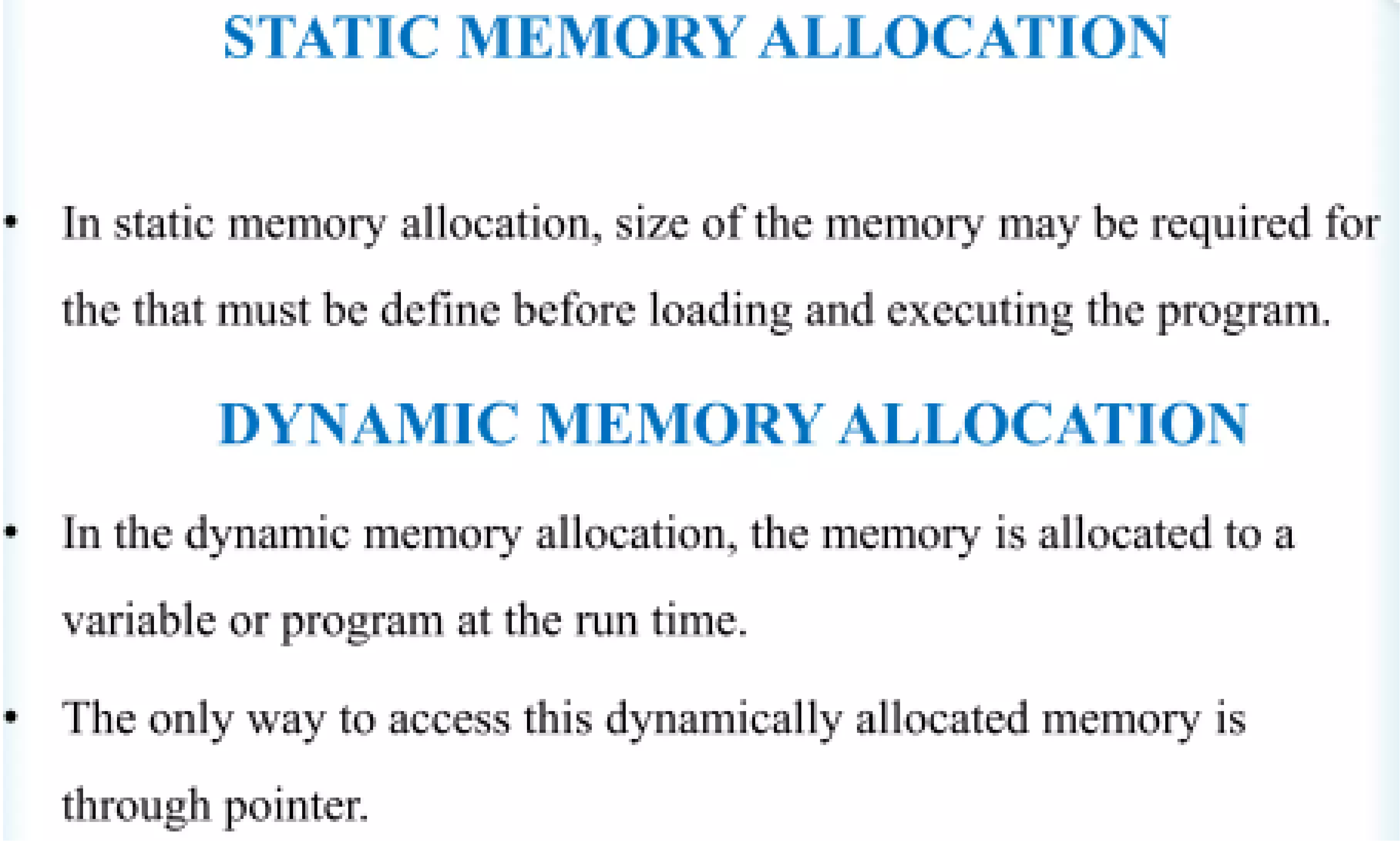


```
Enter the number of elements
4
Enter 4 array elements
2.1
2.2
2.3
2.4
Sum = 9.000000
Mean = 2.250000
Standard Deviation = 0.111803
```



MEMORY ALLOCATION

- The blocks of information in a memory system is called **memory allocation.**
- To allocate memory it is necessary to keep in information of available memory in the system. If memory management system finds sufficient free memory, it allocates only as much memory as needed, keeping the rest available to satisfy future request.
- In memory allocation has two types. They are **static and dynamic** memory allocation.

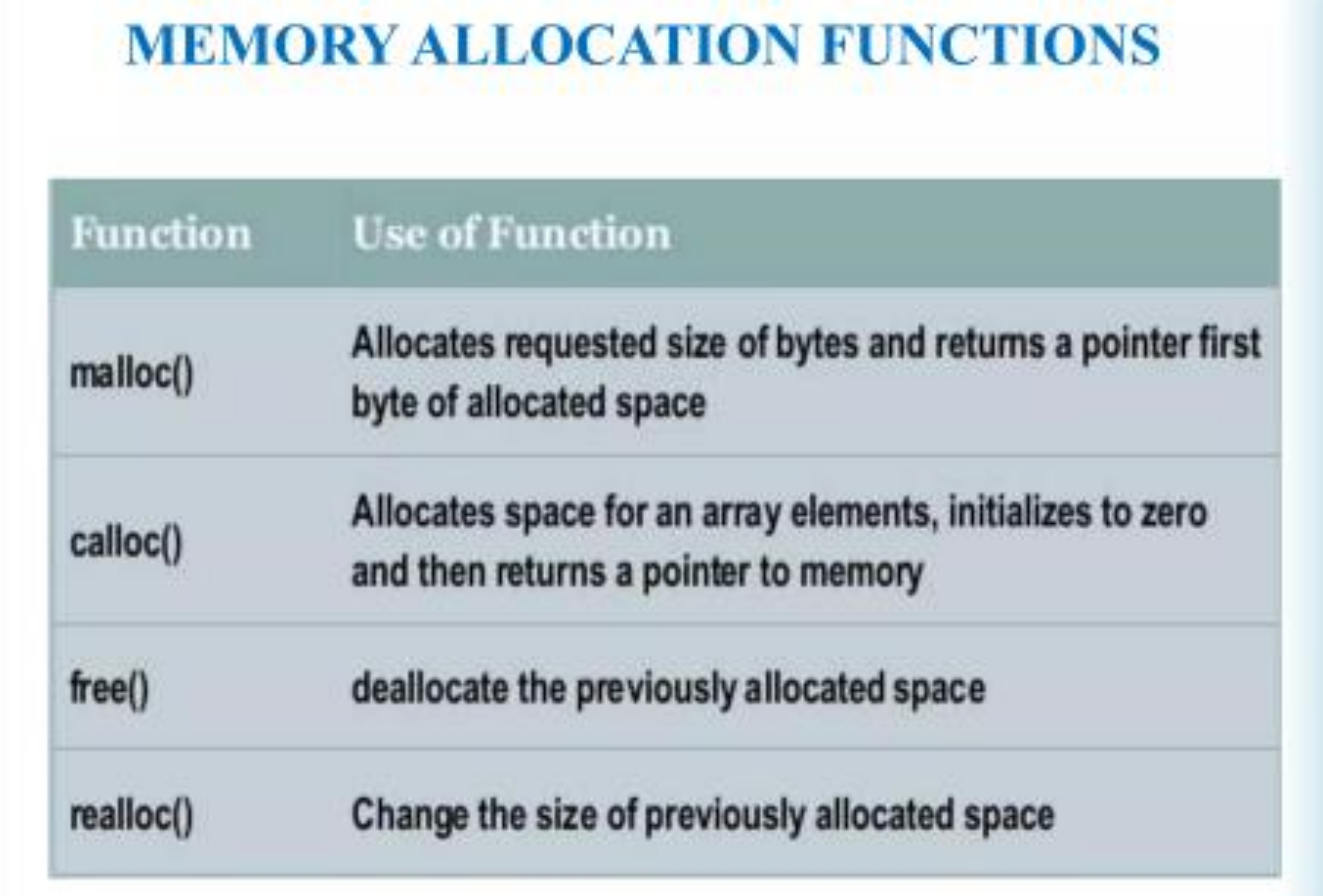


STATIC MEMORY ALLOCATION

- In static memory allocation, size of the memory may be required for the that must be define before loading and executing the program.

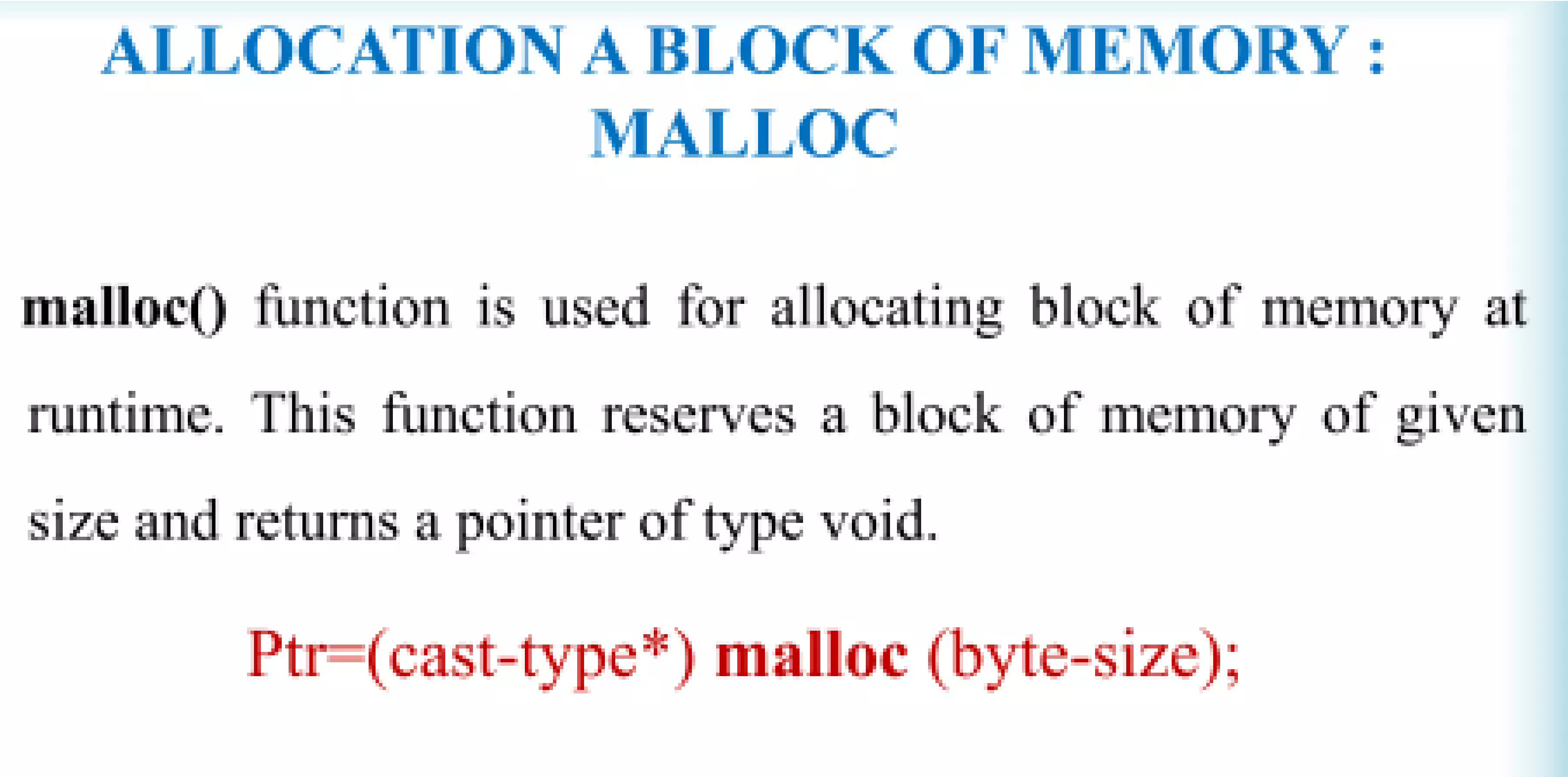
DYNAMIC MEMORY ALLOCATION

- In the dynamic memory allocation, the memory is allocated to a variable or program at the run time.
- The only way to access this dynamically allocated memory is through pointer.



MEMORY ALLOCATION FUNCTIONS

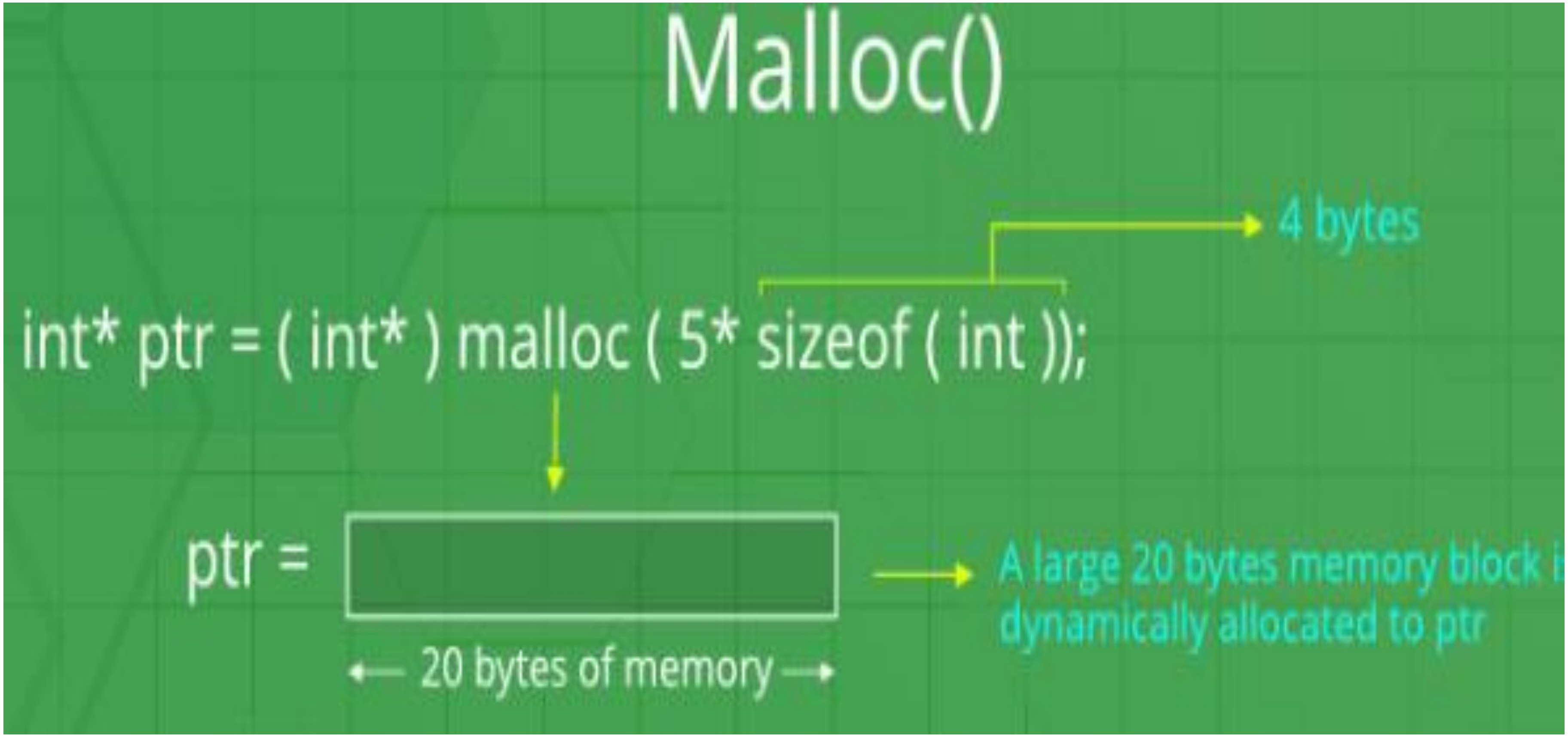
Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of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



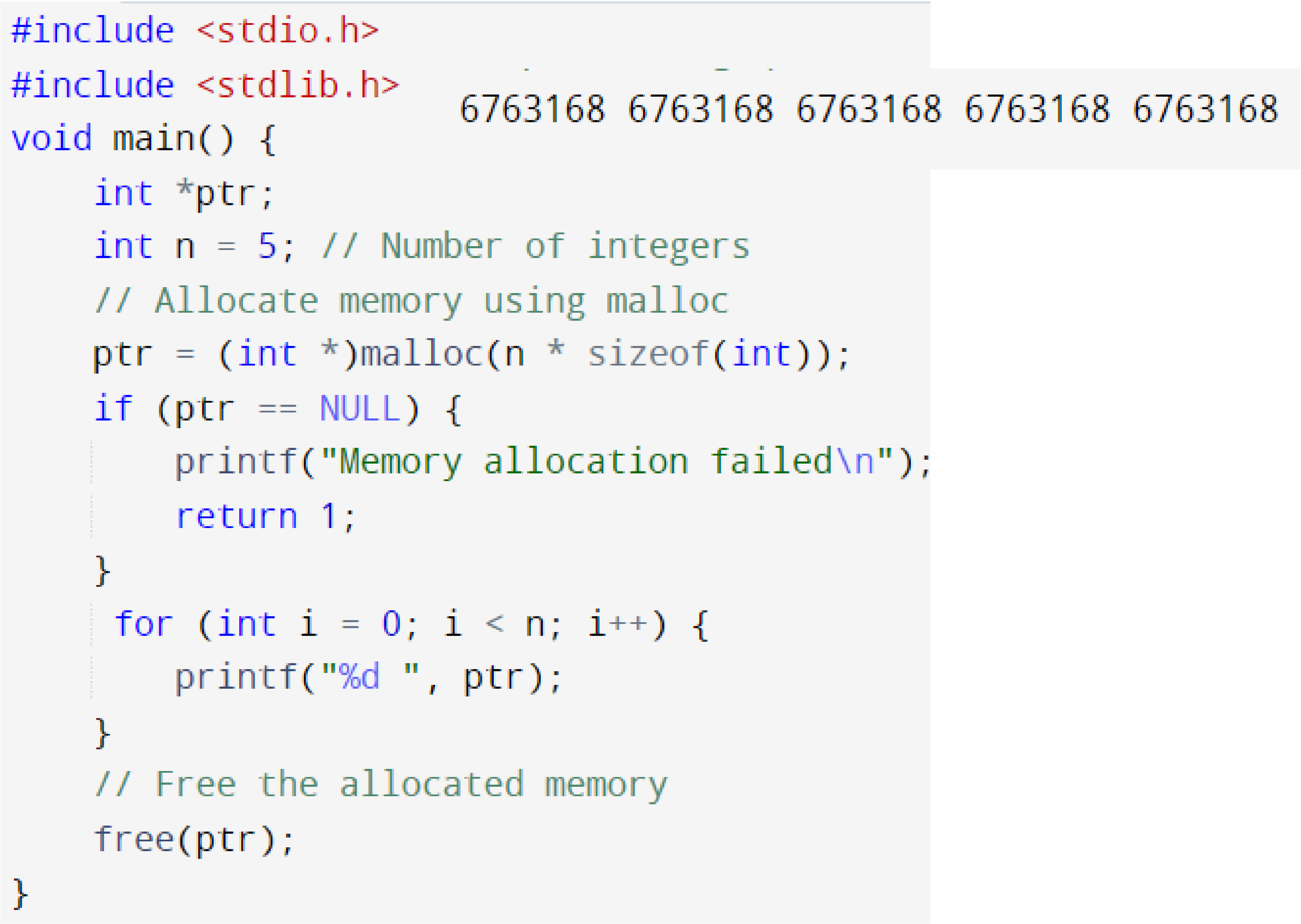
ALLOCATION A BLOCK OF MEMORY : MALLOC

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void.

Ptr=(cast-type*) **malloc** (byte-size);

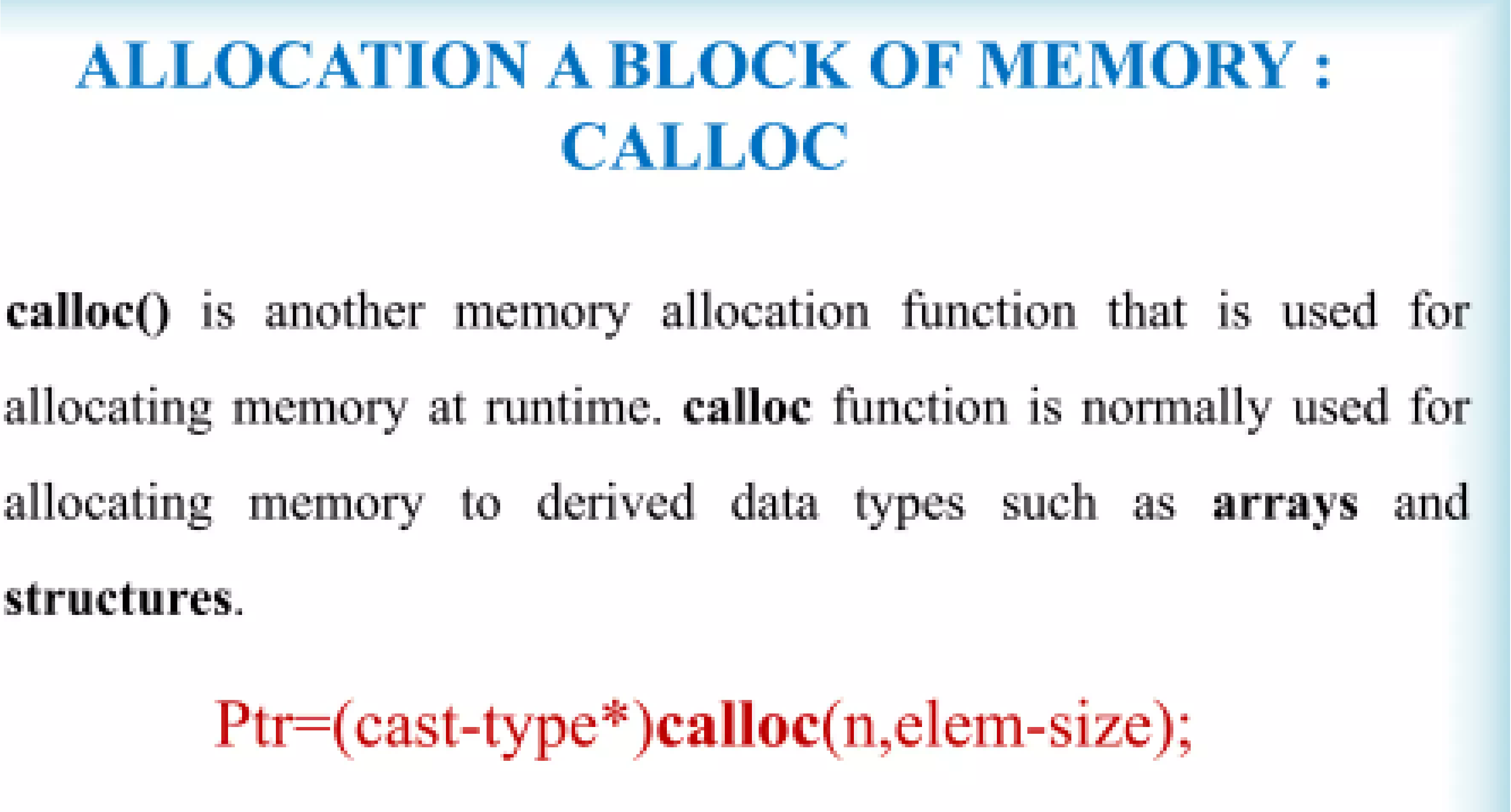


```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```



```c
#include <stdio.h>
#include <stdlib.h>
void main() {
    int *ptr;
    int n = 5; // Number of integers
    // Allocate memory using malloc
    ptr = (int *)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
     for (int i = 0; i < n; i++) {
        printf("%d ", ptr);
    }
    // Free the allocated memory
    free(ptr);
}
```

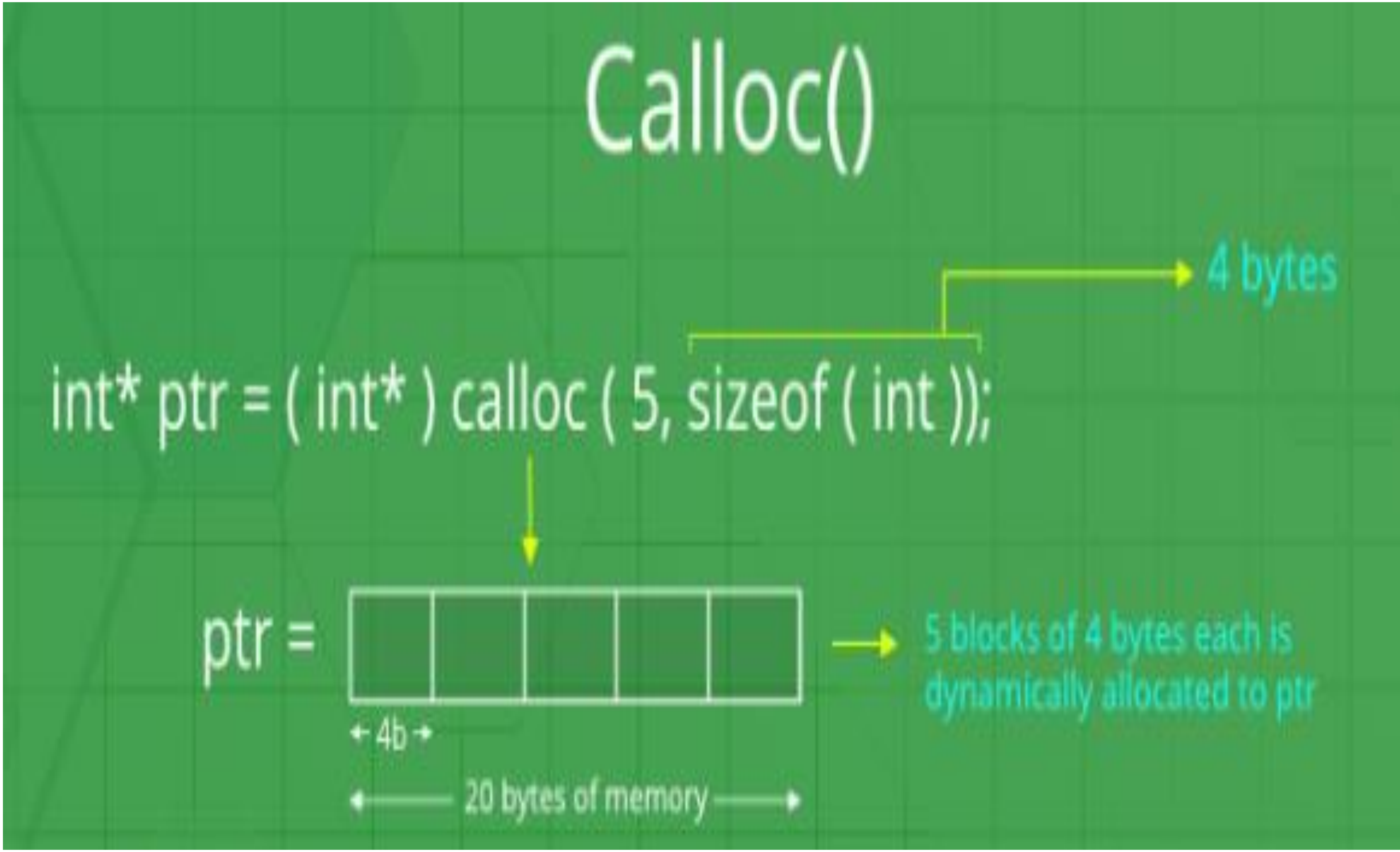
```
6763168 6763168 6763168 6763168 6763168
```



ALLOCATION A BLOCK OF MEMORY : CALLOC

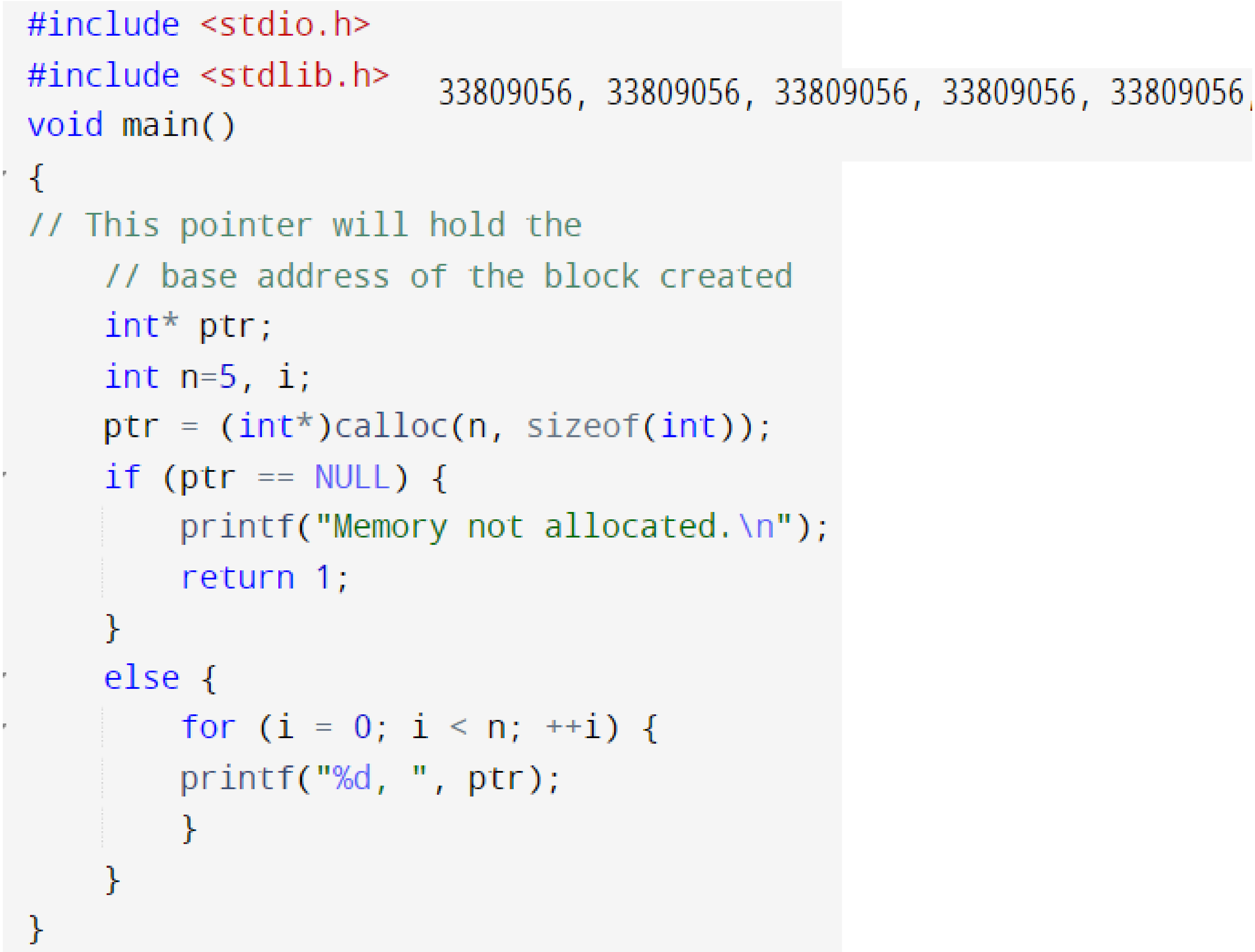
calloc() is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**.

```
Ptr=(cast-type*)calloc(n,elem-size);
```



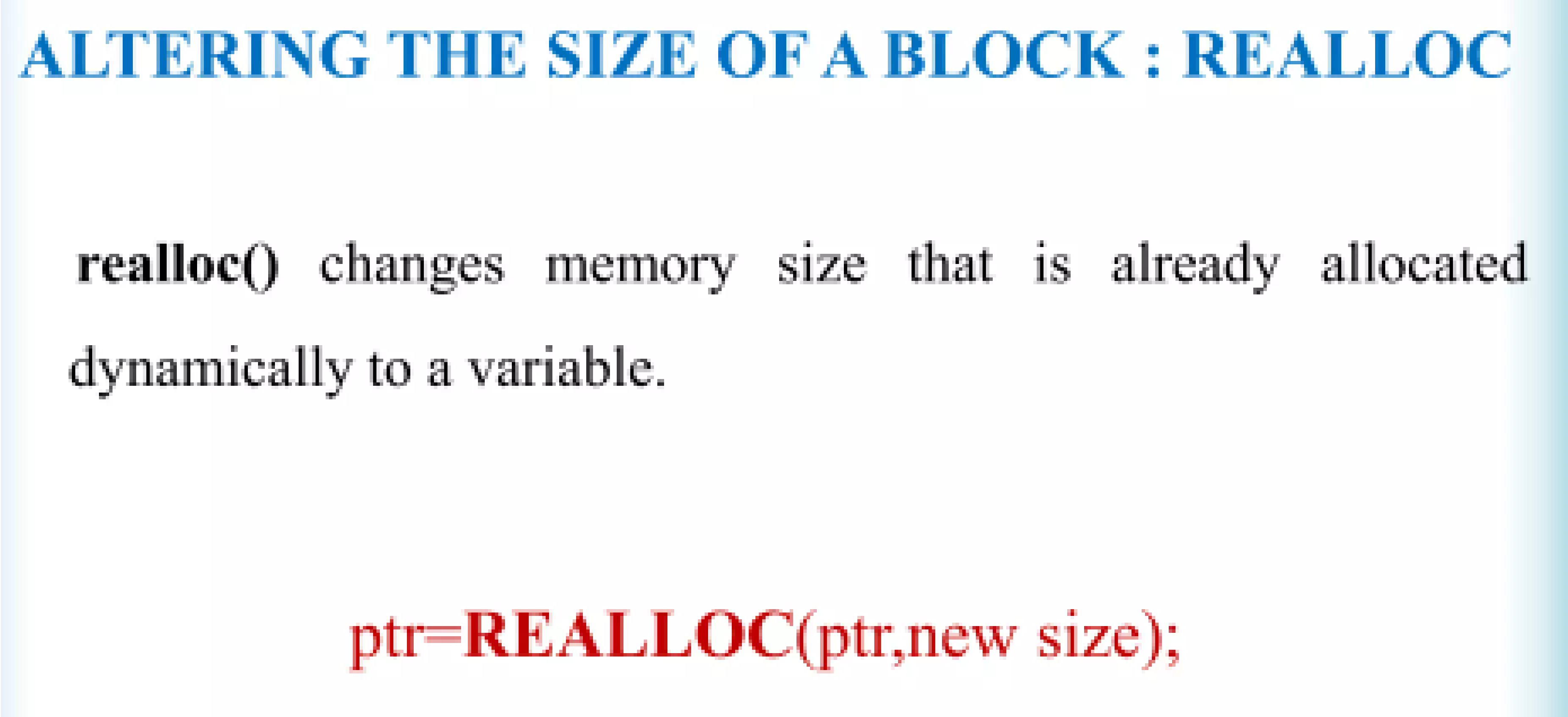
Calloc()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```



```
#include <stdio.h>
#include <stdlib.h>
void main()
{
// This pointer will hold the
    // base address of the block created
    int* ptr;
    int n=5, i;
    ptr = (int*)calloc(n, sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }
    else {
        for (i = 0; i < n; ++i) {
        printf("%d, ", ptr);
        }
    }
}
```

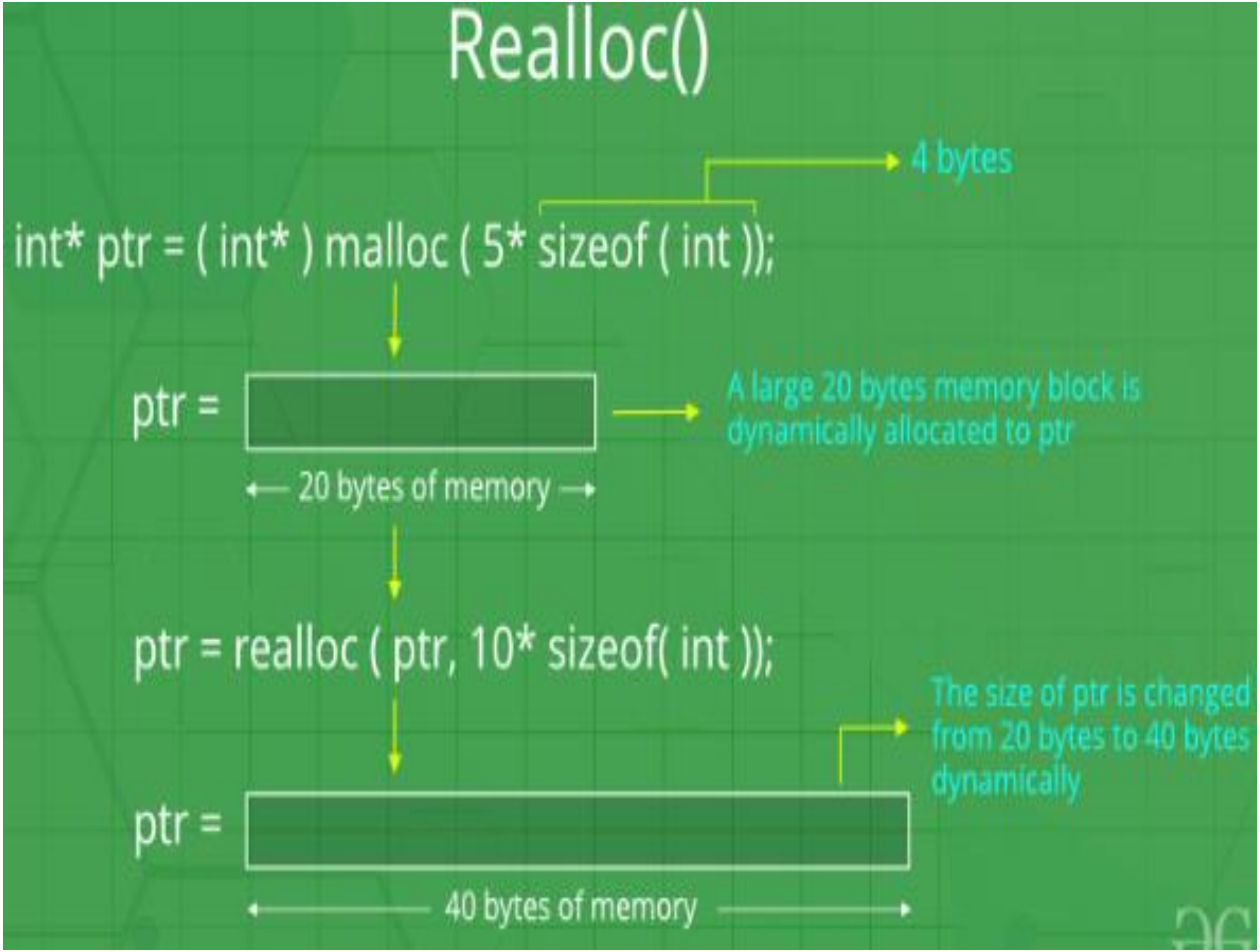
```
33809056, 33809056, 33809056, 33809056, 33809056,
```



ALTERING THE SIZE OF A BLOCK : REALLOC

realloc() changes memory size that is already allocated dynamically to a variable.

ptr=**REALLOC**(ptr,new size);



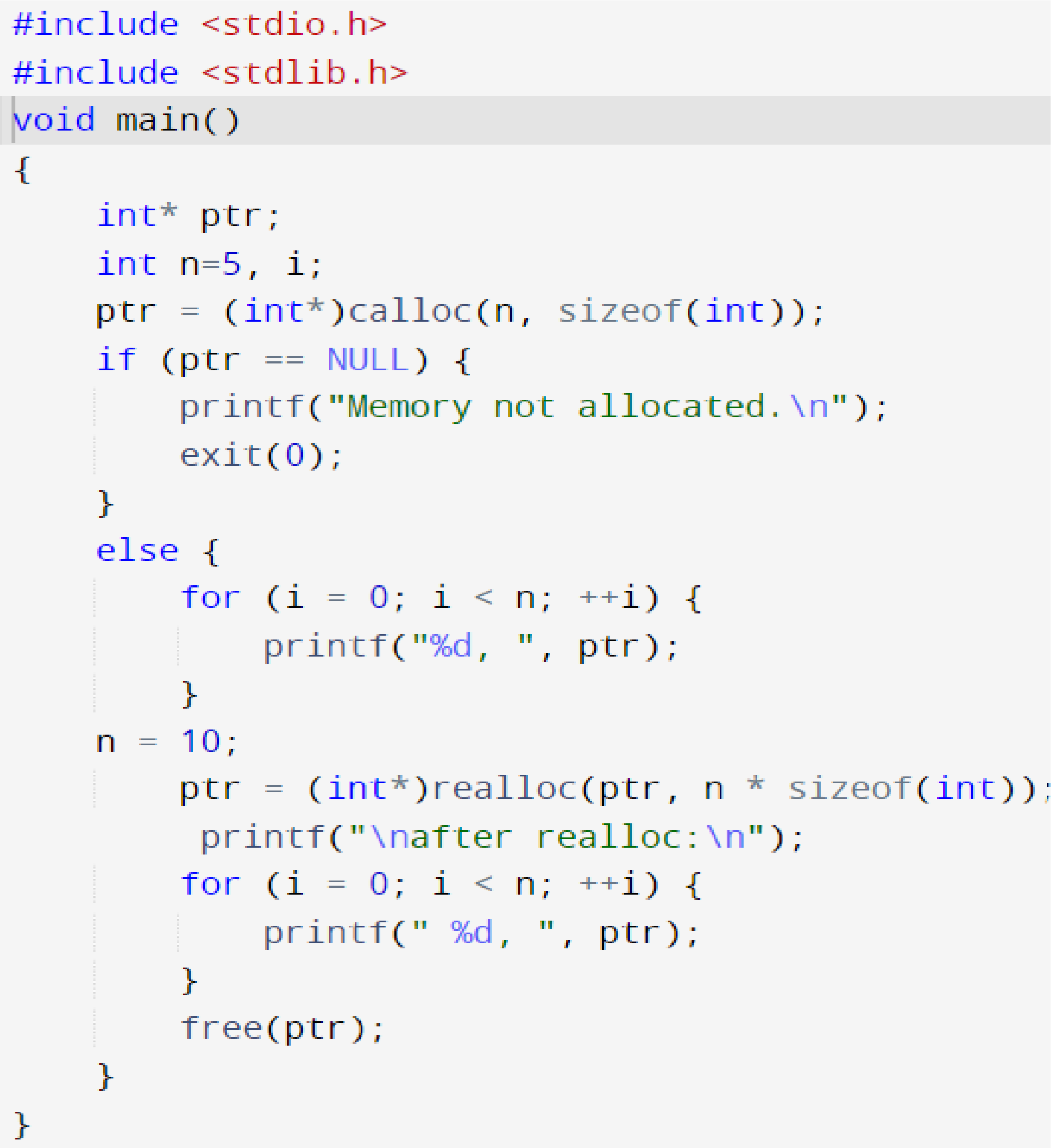
Realloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

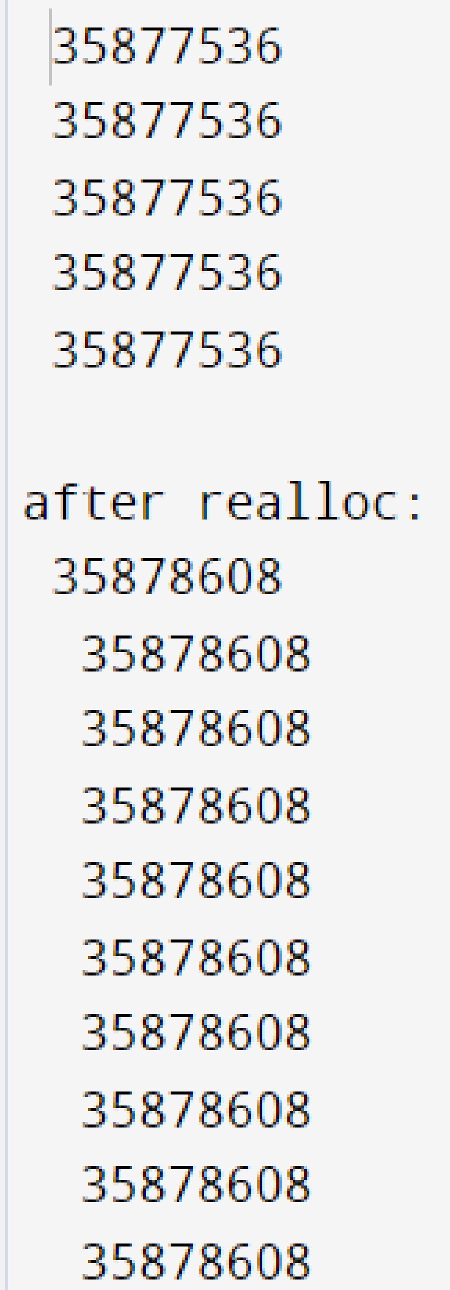
A large 20 bytes memory block is dynamically allocated to ptr

```
ptr = realloc ( ptr, 10* sizeof( int ));
```

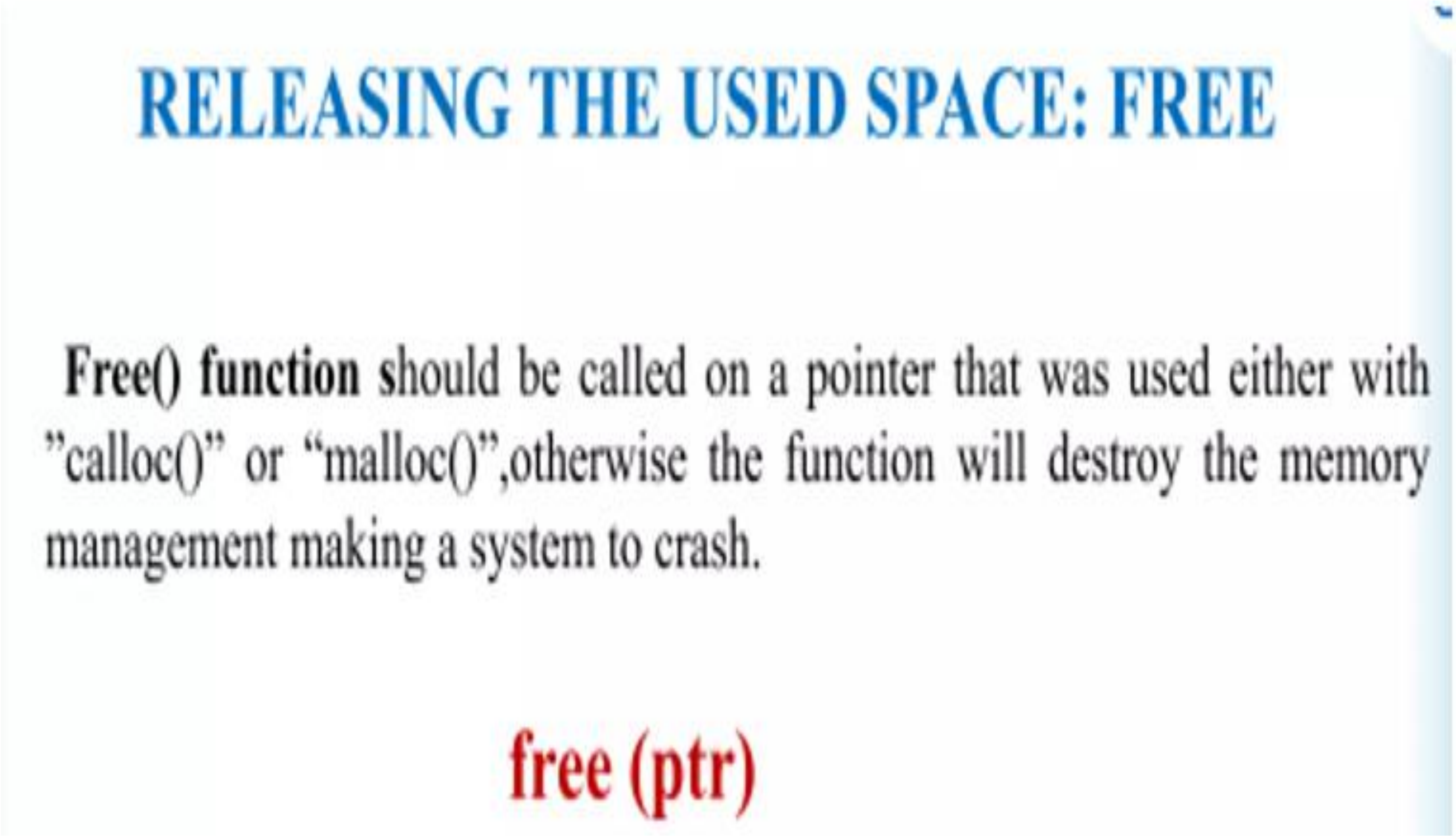
The size of ptr is changed from 20 bytes to 40 bytes dynamically



```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int* ptr;
    int n=5, i;
    ptr = (int*)calloc(n, sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr);
        }
    n = 10;
        ptr = (int*)realloc(ptr, n * sizeof(int));
         printf("\nafter realloc:\n");
        for (i = 0; i < n; ++i) {
            printf(" %d, ", ptr);
        }
        free(ptr);
    }
}
```



```
 35877536
 35877536
 35877536
 35877536
 35877536

after realloc:
 35878608
  35878608
  35878608
  35878608
  35878608
  35878608
  35878608
  35878608
  35878608
  35878608
```



RELEASING THE USED SPACE: FREE

Free() function should be called on a pointer that was used either with ”calloc()” or “malloc()”,otherwise the function will destroy the memory management making a system to crash.

free (ptr)

Functions

- A function is a collection of statements that perform a specific task
- These functions are very useful to read write and debug complex programs ;

Types of Functions

- These can be broadly classified into two types
  - Built-in functions
  - User defined functions

Why are functions needed

1.Improve Modularity

- We can divide a large program into multiple small modules.
- If we write programs using modules, it very easy to understand the program.
- And it's also easy to debug (say, which part doesn't work properly) the program.

2.Code Reusability

- **Call a function multiple times**, thereby allowing reusability and modularity in C programming.
- It means that instead of writing the same code again and again for different arguments, you can simply enclose the code and make it a function and then call it multiple times by merely passing the various arguments

Why are functions needed

3.Reduce workload:

A big program can be broken into smaller function, then divide the workload by writing different functions.

4.Speed:

Functions CAN make code faster by coding logic once instead of repeating several times

- **User defined functions** :-

The user defined function is defined by the user according to its requirements.

- instead of relying only on the built-in functions C allows us to create our own function called user defined function
- Parts of user defined function.
  - Function Declaration or Function prototype
  - Function call or calling Function
  - Function Definition or defining a function

- **Function Declaration or Function prototype** :-
- It will inform the compiler about the return type, function name and number of arguments along with the data types.
- syntax:

return_type function_name(argument _list);

Function Declaration or Function prototype :-

return_type function_name(argument _list);

- **return_type** :- is the data type of the value that is returned or sent from the function.
- **Function_name** :-function should be given a descriptive name.
- **argument _list** :- contains type and names of the variables that must be passed to the function.

Function Declaration or Function prototype :-

- Example:-

int large (int x, int y);

- is a function declaration with function_name “large” with return _type “integer” and has two arguments “x” and “y” of integer type.
- NOTE:-
  - if we define a function before main ( ) function the there is no need of function declaration
  - if we define the function after main ( ) function then it is mandatory to declare the function because it will inform the compiler.

Function call or calling function :-

- Invoking the function with valid number of arguments and valid data type is called as function call.
- To call a function one simply needs to pass the required parameters along with the function name and if the function returns the value then one can store the returned value.

- Syntax:

function_name(argumement_list);

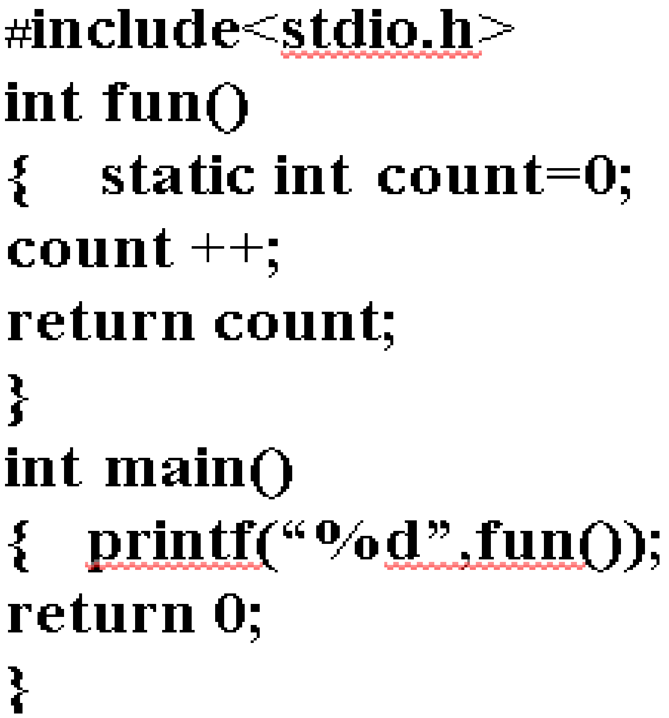
- argumement_list :- consist of constant(s) , variable(s), or Expression(s).

Calling function and called function :-

- The **function main( )** that calls another function is called calling function
- **The function** being called by the calling function is known as called function.

Called function

Calling function



```
#include<stdio.h>
int fun()
{    static int count=0;
count ++;
return count;
}
int main()
{   printf(“%d”,fun());
return 0;
}
```

- Example:-

large (m,n);

- The function can be invoked in various ways
  - large(m,n); //m and n are variables.
  - large(5,8); //5 and 8 are constants
  - large(5+2,6); // The first argument is an expression which is evaluated to 7
  - large(2*3,5+3); //is an expression which is equivalent to large(6,8);

Function definition or Defining a function

- The declared function must define the same to perform the specific task.
- Syntax

```
return_type  function_name(argument _list)
      {
                local_variable_declaration;
                Body of the function;
      }
```

- return_type :- when the function is called the function may or may not return a value

- If the function returns a value then the return_type will be any appropriate data type (int, float, char etc) and we use the keyword “return” to return the value.
- If the function does not return a value then the return_type will be “void” and no need to use the keyword “return”

- **function_name** :- is the name of the function.
- **argument _list** :- these are also called as parameters. the argument_list refers to the type order and number of parameters of the function.
- **local_variable_declaration** :-these are temporary variables which are required only within this function.

- **Function body**:- The body of the function contains the collection of statements that define what the function does.
- when the program makes the function call the program control is transferred to the called function. This called function performs the defined task and returns the program control back to the main( ) function.

/* C program to find area of circle using functions */

```
#include<stdio.h>
float area(float r); // function declaration
void main()
{
        float r,x;
        printf("Enter the radius\n");
        scanf("%f",&r);
        x=area(r); // function call
         printf("Area ofcircle= %f\n",x);
}

float area(float r) // function defination
{
         float x;
         x=3.142*r*r;
         return x;
}
```

return Statement

- A return statement **ends the execution of a function, and returns control to the calling function**.

- Syntax.   **return <expression>;**

Example: return 10; return a; return a+b;

- The **value** will be **passed back** to the function where it was called.
- **Return** statement **may or may not return** the value to the calling function.
- For functions that **have no return statement**, after execution of last statement of called function control returns to the ***calling function.***
- Function that has **void** as its return statement **cannot return** any value to the calling function,

Parameter passing mechanism

There are two methods by which parameters or arguments can be passed to the function

- **Call by value**
- **Call by reference**

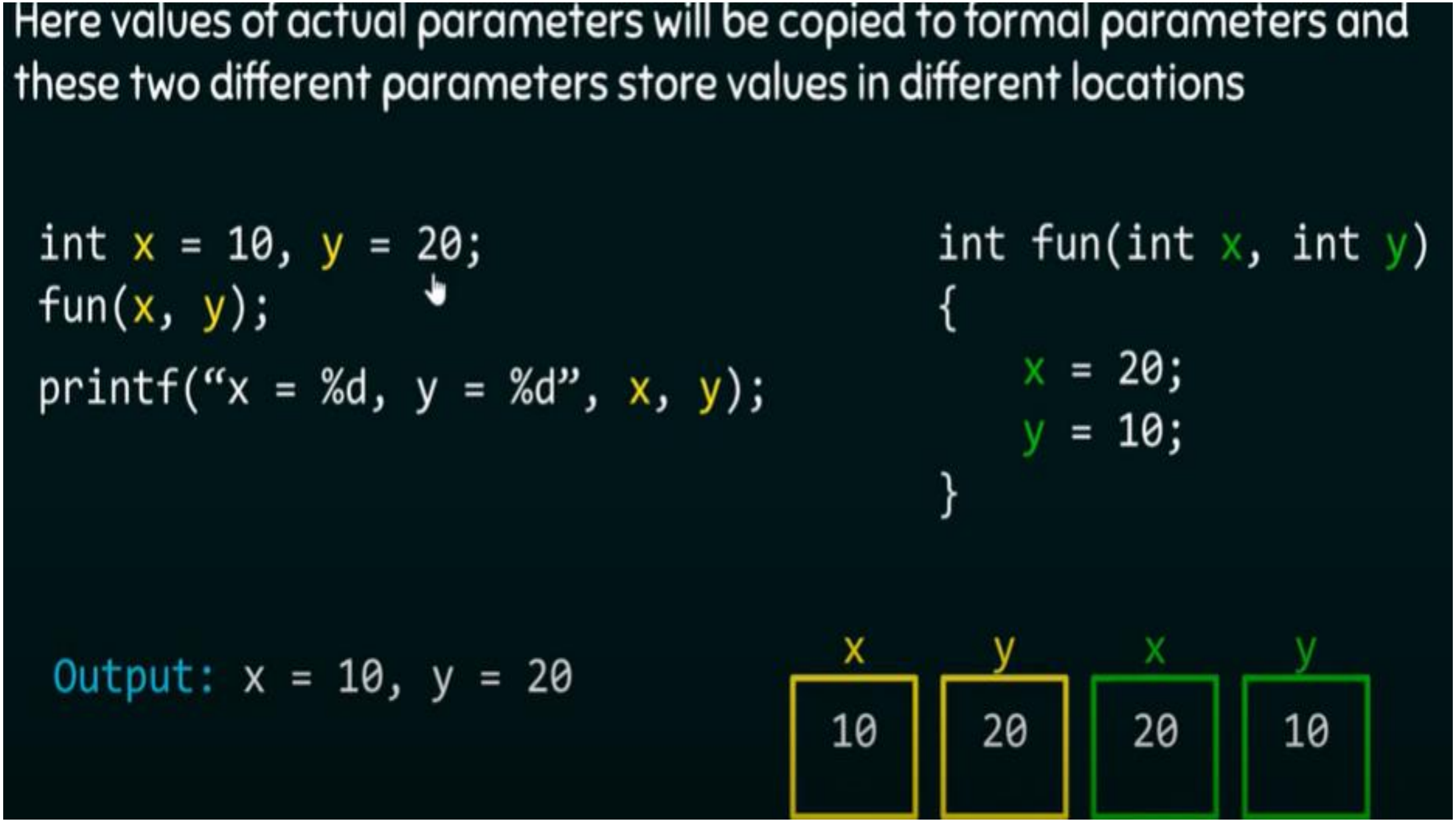
Call by value or Argument passing by value

- When an variable or value is passed to an function during its call such function invocation(call) is called as **call by value**.

Call by reference or Argument passing by reference

when the address of the variable is passed to the function during its invocation(call) such a function is called as **call by reference.**

Call by value



Here values of actual parameters will be copied to formal parameters and these two different parameters store values in different locations

```
int x = 10, y = 20;
fun(x, y);

printf("x = %d, y = %d", x, y);
```

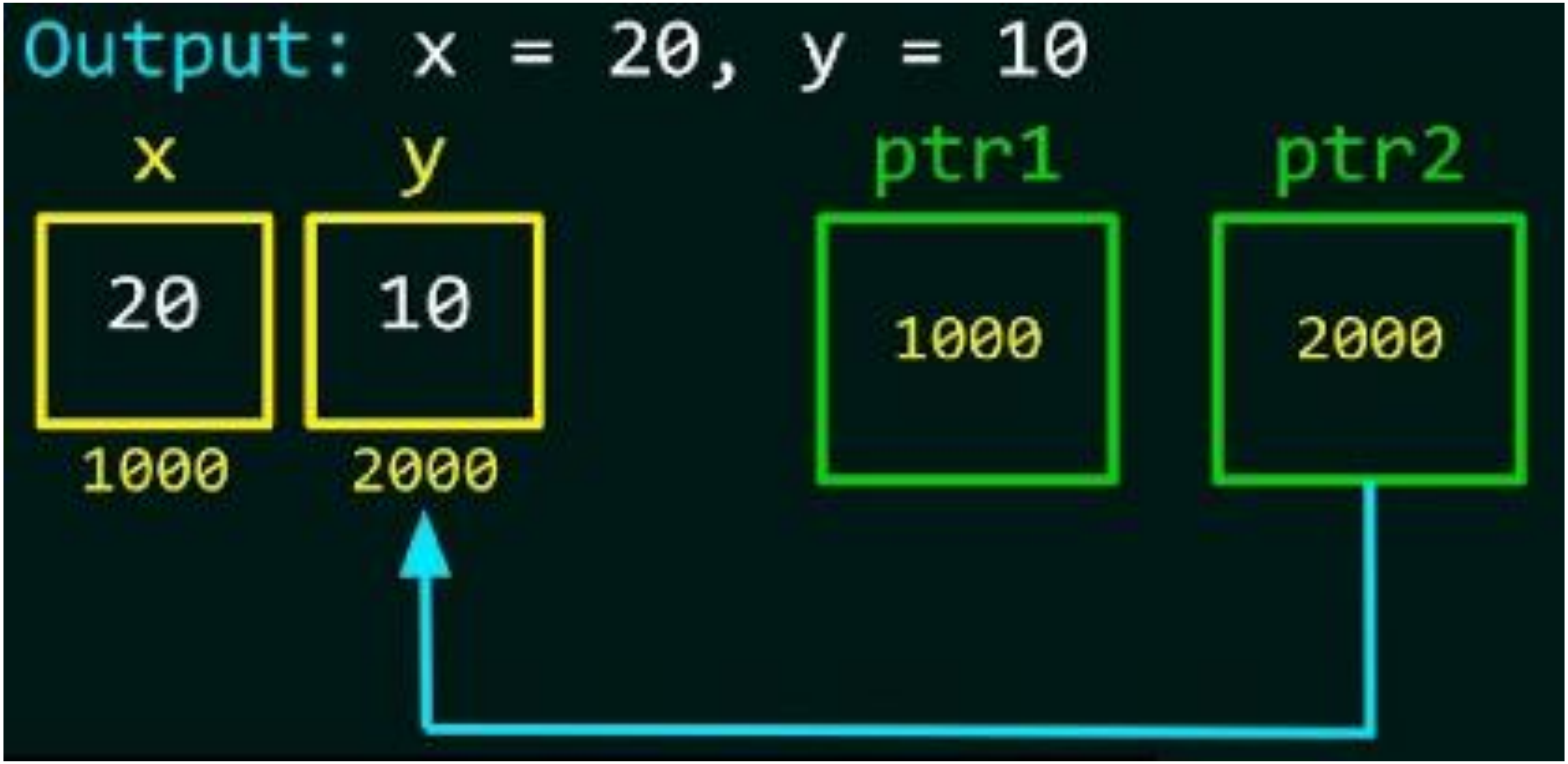
```
int fun(int x, int y)
{
    x = 20;
    y = 10;
}
```

Output: x = 10, y = 20

C program to demonstrate call by value

```
#include<stdio.h>
int sum(int n);
void main()
{
             int n,x;
            printf("Enter the value of n\n");
            scanf("%d",&n);
            x=sum(n);
            printf("Sum of natural numbers=%d\n",x);
}
int sum(int n)
{
            int res=0,i;
             for(i=1;i<=n;i++)
                         res=res+i;
            return res;
}
```

Call by reference



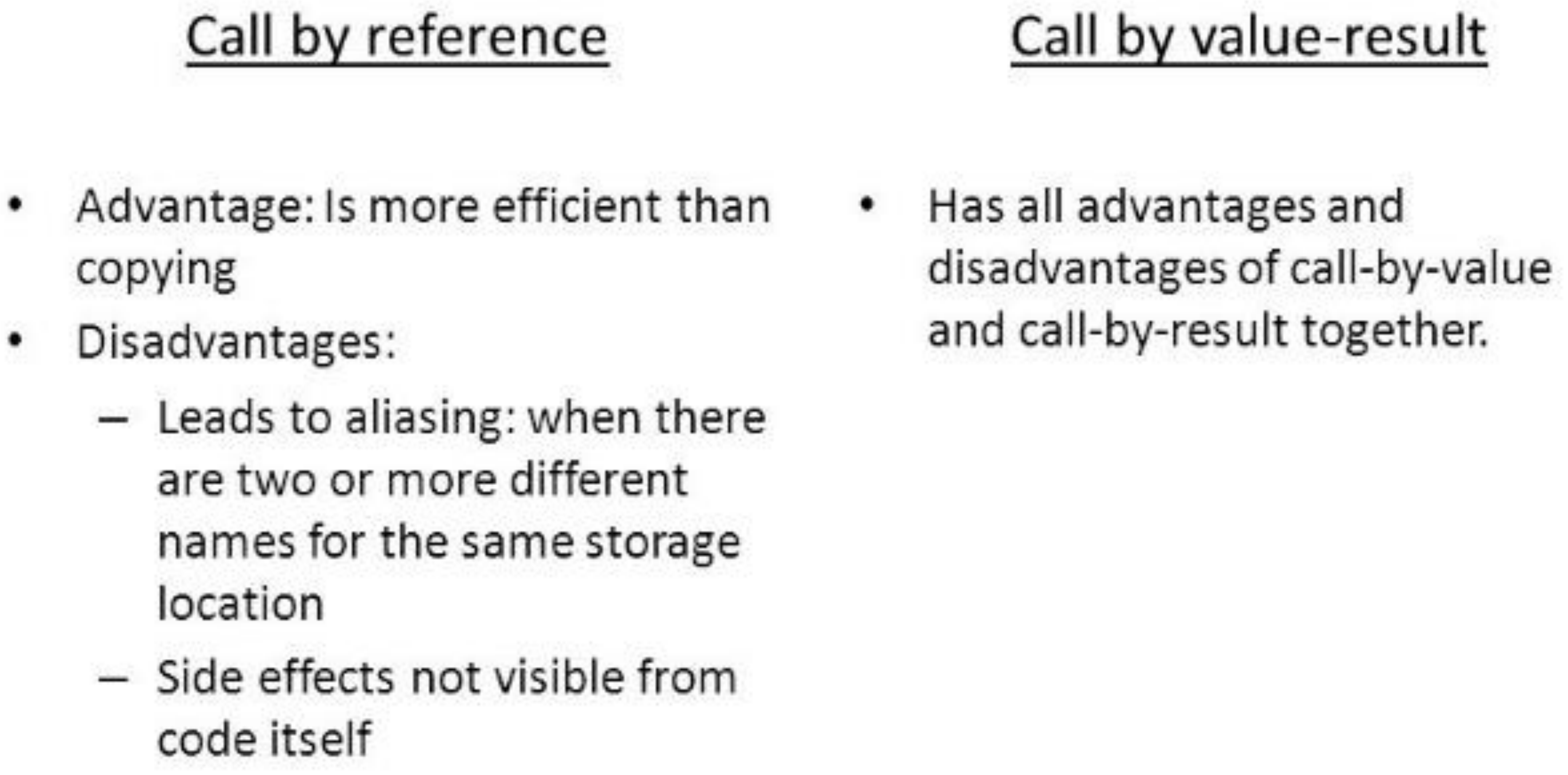
```
Output: x = 20, y = 10
```

Here the values are not passed to called function, the addresses of values are passed to the called function

C program to swap two numbers using call by reference or Argument passing by reference

```
#include<stdio.h>
void swap(int *a,int *b);
void main()
{
        int a,b;
        printf("Enter two numbers\n");
         scanf("%d%d",&a,&b);
         printf("Before Swapping\n a=%d\t b=%d\n",a,b);
         swap(&a, &b);
         printf("After Swapping\n a=%d\t b=%d\n",a,b);
}
void swap(int *a, int *b)
{
         int temp;
        temp=*a;
         *a=*b;
         *b=temp;
}
```

Advantages and Disadvantage of Call by value and Reference



Call by reference

- Advantage: Is more efficient than copying
- Disadvantages:
  - Leads to aliasing: when there are two or more different names for the same storage location
  - Side effects not visible from code itself

Call by value-result

- Has all advantages and disadvantages of call-by-value and call-by-result together.

Types of function based on arguments and return values or Types of user defined function

- – Function with argument/parameter with return value.
- – Function with argument/parameter without return value.
- – Function without argument/parameter with return value.
- – Function without argument/parameter without return value.

Function with argument with return value

The arguments are passed from calling function to the called function.

- based on the received argument values the called function performs the required action and returns the value back to calling function (main( ) function).

/* C program to demonstrate Function with argument with return value */

```
#include<stdio.h>
int add(int a, int b);
void main()
{
          int a,b,sum;
          printf("Enter two numbers\n");
          scanf("%d%d",&a,&b);
          sum=add(a,b);
          printf("The Sum of two numbers=%d\n",sum);
}

int add(int a, int b)
{
           int sum;
          sum=a+b;
          return sum;
}
```

Function with argument without return value

- The arguments are passed from calling function to the called function.
- based on the received argument values the called function performs the required action but does not return any value back to calling function (main( ) function).

/* C program to demonstrate Function with argument without return value */

```
#include<stdio.h>
void add(int a, int b);
void main()
{
        int a,b;
        printf("Enter two numbers\n");
         scanf("%d%d",&a,&b);
        add(a,b);
}

void add(int a, int b)
{
        int sum;
         sum=a+b;
         printf("The Sum of two numbers=%d\n",sum);
}
```

Function without argument with return value

- Here no arguments are passed from calling function to the called function.
- The called function performs the required action by taking the necessary arguments and returns the value back to calling function (main( ) function).

/* C program to demonstrate Function without argument with return value */

```
#include<stdio.h>
int add();
void main()
{
        int sum;
         sum=add();
        printf("The Sum of two numbers=%d\n",sum);
}

int add()
{
        int a,b,sum;
         printf("Enter two numbers\n");
        scanf("%d%d",&a,&b);
         sum=a+b;
        return sum;
}
```

Function without argument without return value

- Here no arguments are passed from calling function to the called function.
- The called function performs the required action by taking the necessary arguments but does not return any value back to calling function (main( ) function).

/* C program to demonstrate Function without argument without return value */

```
#include<stdio.h>
void add();
void main()
{
          add();
}

void add()
{
         int a,b,sum;
         printf("Enter two numbers\n");
         scanf("%d%d",&a,&b);
         sum=a+b;
         printf("The Sum of two numbers=%d\n",sum);
}
```

Scope of variables

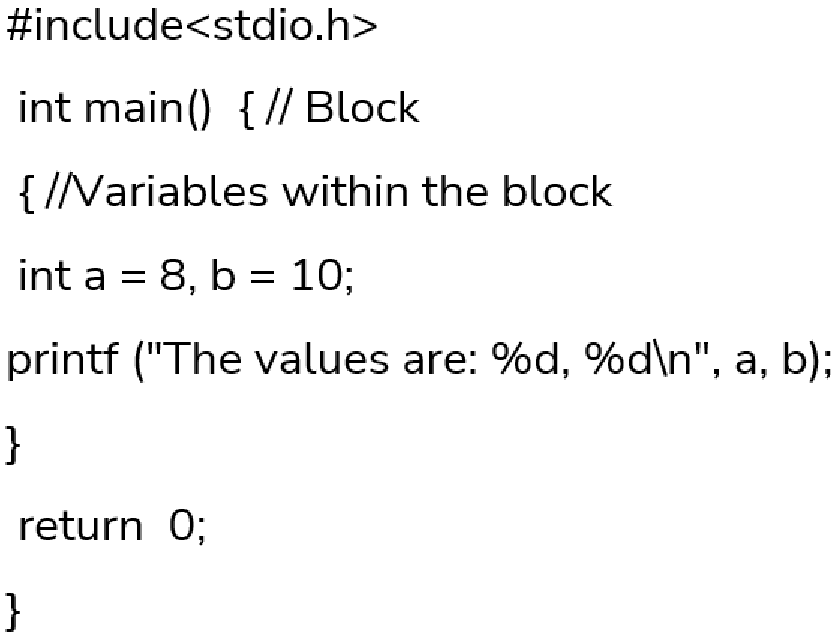
The scope of a variable is the block of code in the entire program where the variable is declared, used, and can be modified.

1. **Block Scope:** A **Block** in C is a set of statements written within the right and left braces.

A block may contain more blocks within it, i.e., nested blocks.

The right and left braces are as follows:

{ }

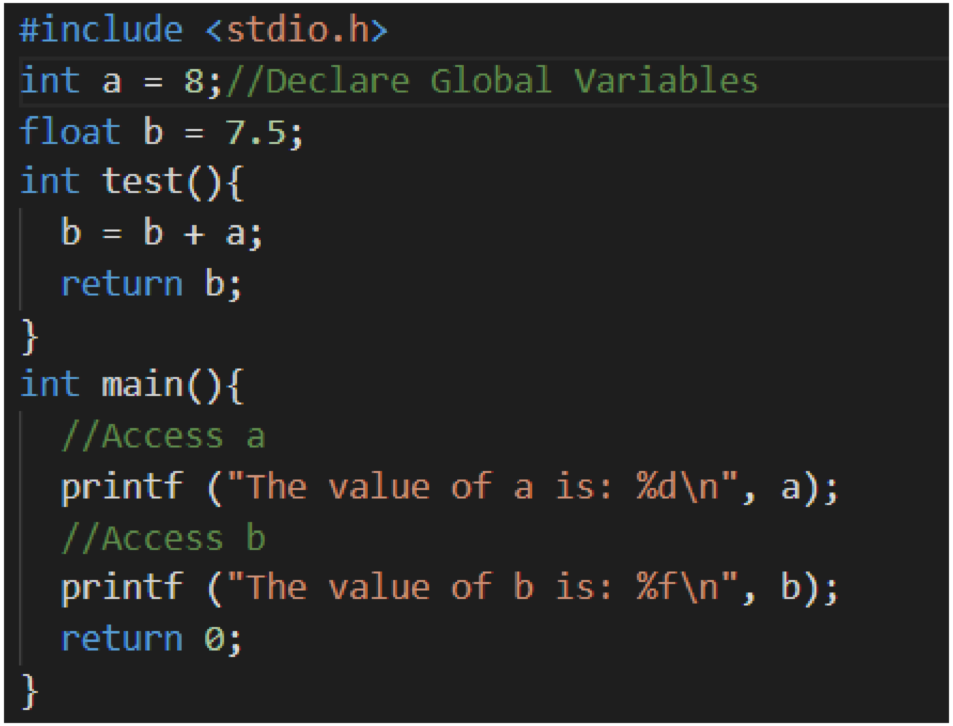


```
#include<stdio.h>
 int main()  { // Block
 { //Variables within the block
 int a = 8, b = 10;
printf ("The values are: %d, %d\n", a, b);
}
 return  0;
}
```

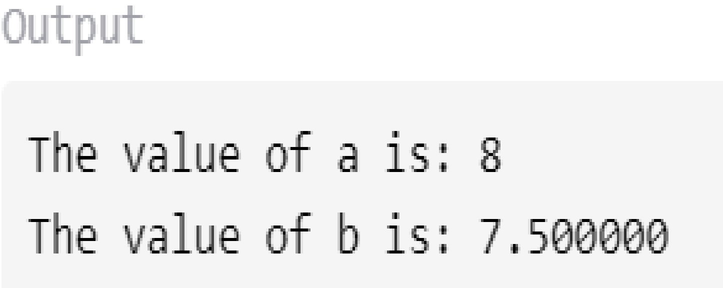
Scope of variables

2. **Program scope**

Global variables declared outside the function bodies have a **program scope**. The availability of global variables stays for the entire program after its declaration



```c
#include <stdio.h>
int a = 8;//Declare Global Variables
float b = 7.5;
int test(){
  b = b + a;
  return b;
}
int main(){
  //Access a
  printf ("The value of a is: %d\n", a);
  //Access b
  printf ("The value of b is: %f\n", b);
  return 0;
}
```



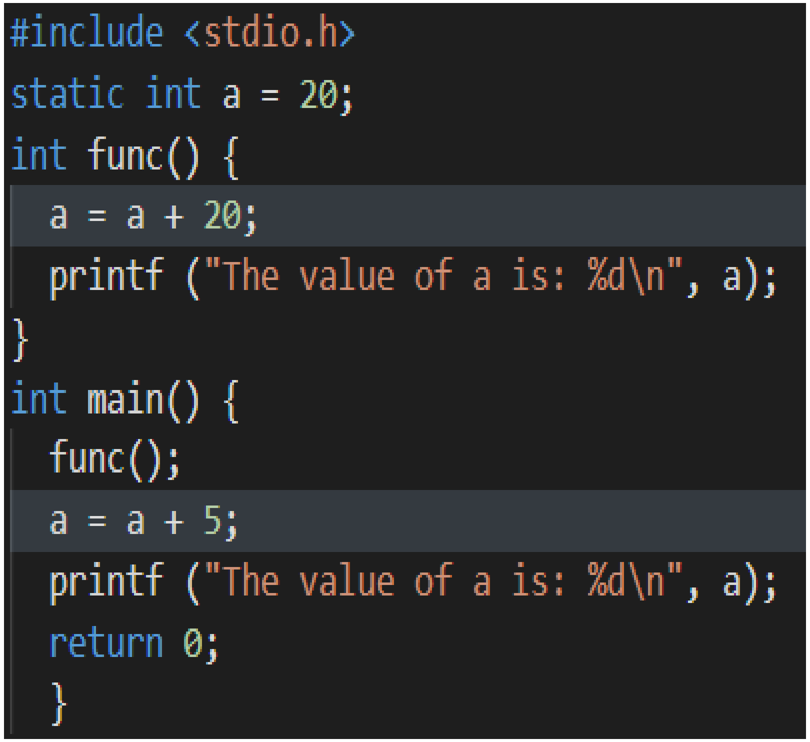
Output

```
The value of a is: 8
The value of b is: 7.500000
```

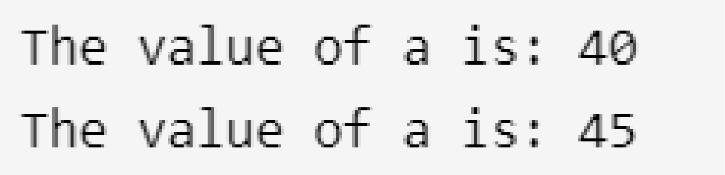
Scope of variables

3. **File scope**

- These variables are usually declared outside of all of the functions and blocks, at the top of the program and can be accessed from any portion of the program.
- The *global* *static* *variable* is accessible by all the functions in the same source file as the variable. This variable has a File Scope.



```
#include <stdio.h>
static int a = 20;
int func() {
  a = a + 20;
  printf ("The value of a is: %d\n", a);
}
int main() {
  func();
  a = a + 5;
  printf ("The value of a is: %d\n", a);
  return 0;
  }
```

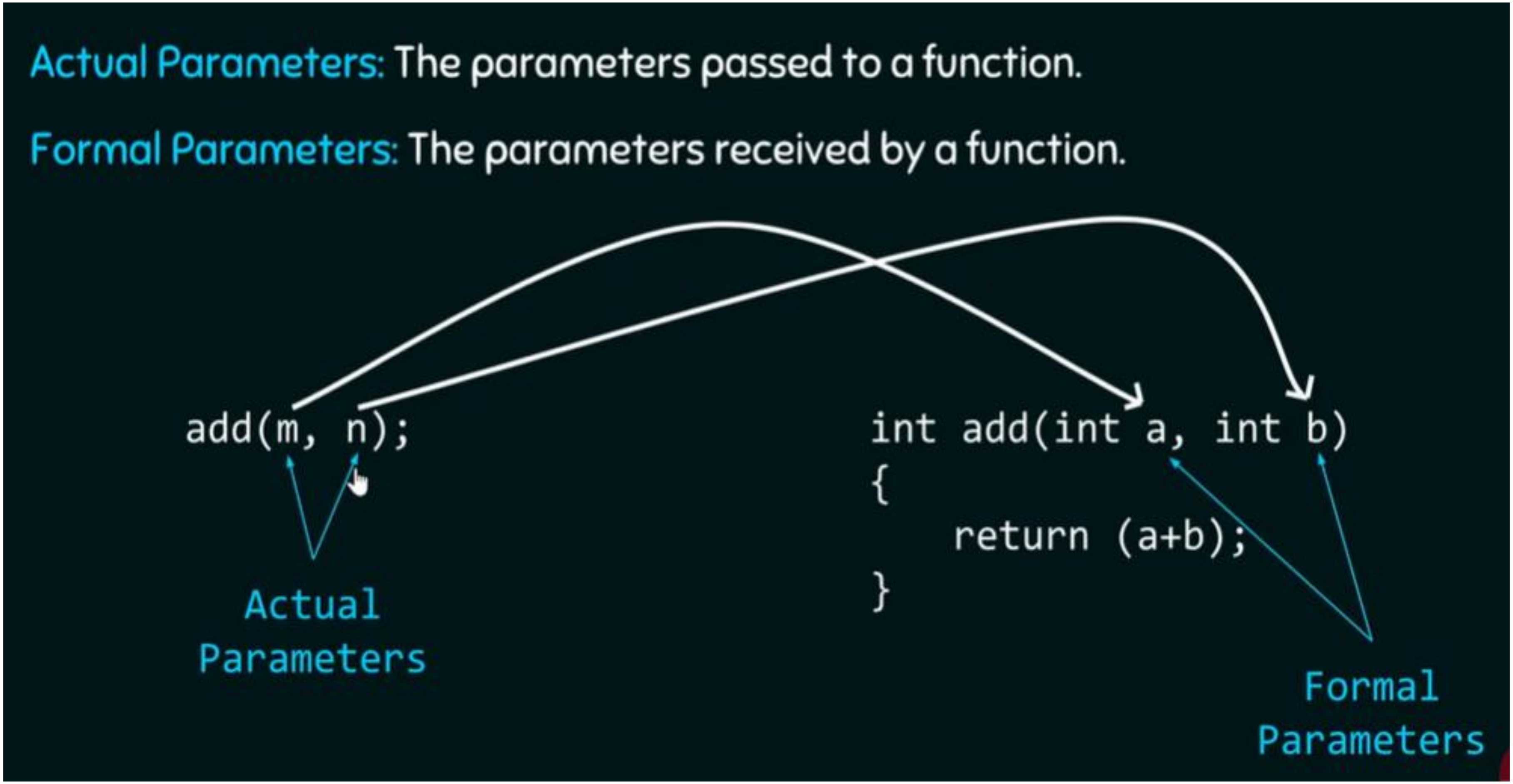


```
The value of a is: 40
The value of a is: 45
```

Actual arguments and Formal arguments :-

- When the function is called, the values that are passed in the call are called as actual parameters.
- The formal parameters are written in the function prototype and function header of the definition .
- These are called as dummy parameters which are assigned the values from the arguments when the function is called.

Actual arguments and Formal arguments :-



Actual Parameters: The parameters passed to a function.

Formal Parameters: The parameters received by a function.

```
add(m, n);
```

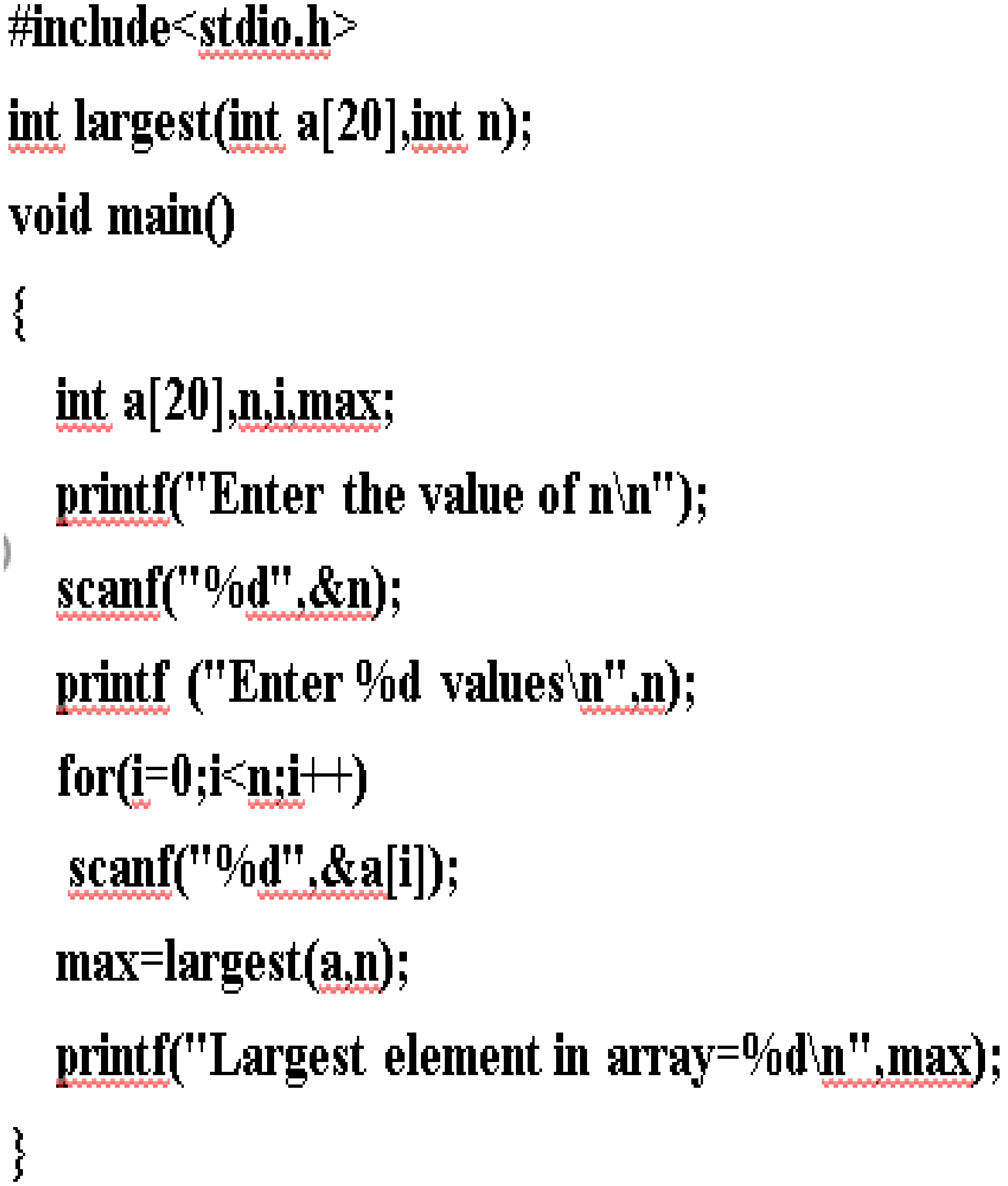
```
int add(int a, int b)
{
    return (a+b);
}
```

C program to demonstrate actual arguments and formal arguments

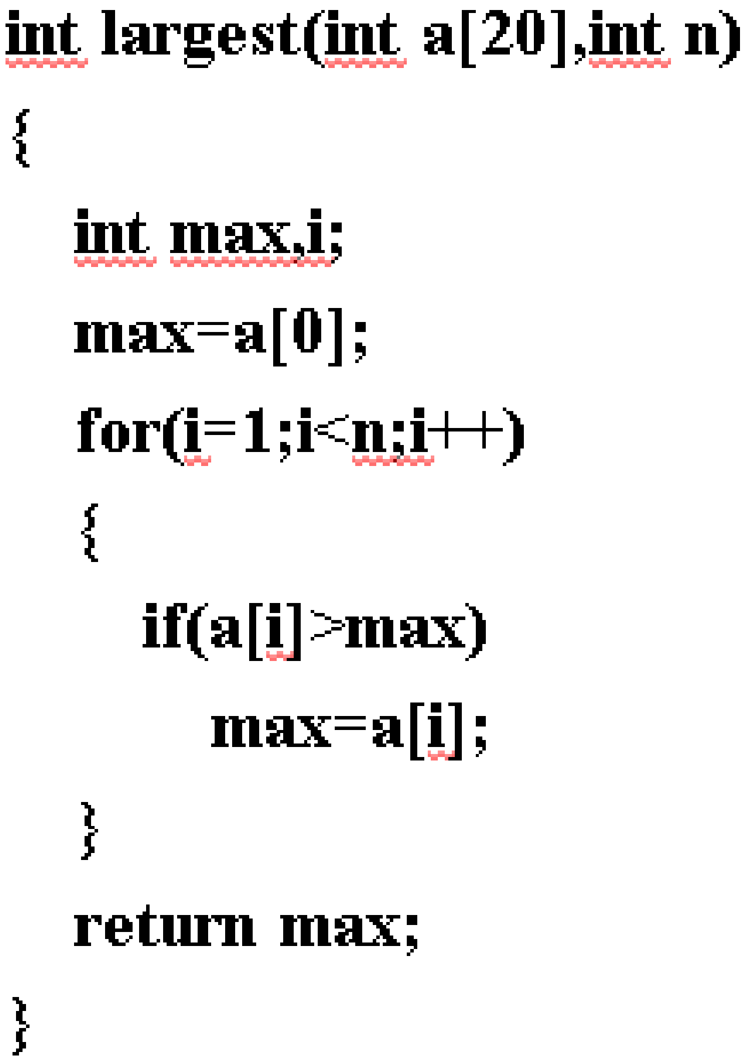
```
#include<stdio.h>
int perimeter(int x,int y);
void main()
{
            int l,b,p;
            printf("Enter length and breadth\n");
             scanf("%d%d",&l,&b);
             p=perimeter(l,b);   // function call with actual parameters
             printf("Perimeter of Rectangle=%d\n",p);
}
int perimeter(int x,int y) // int x, int y are formal parameters
{
            int per;     //int per is a local variable
            per=2*(x+y);
            return per;
}
```

- **Passing Arrays to functions** :- Array elements or an entire array can be passed to a function such a mechanism is called a s passing array to the function.

program to demonstrate passing array to the functions



```
#include<stdio.h>
int largest(int a[20],int n);
void main()
{
   int a[20],n,i,max;
   printf("Enter the value of n\n");
   scanf("%d",&n);
   printf ("Enter %d values\n",n);
   for(i=0;i<n;i++)
    scanf("%d",&a[i]);
   max=largest(a,n);
   printf("Largest element in array=%d\n",max);
}
```



```
int largest(int a[20],int n)
{
   int max,i;
   max=a[0];
   for(i=1;i<n;i++)
   {
      if(a[i]>max)
         max=a[i];
   }
   return max;
}
```