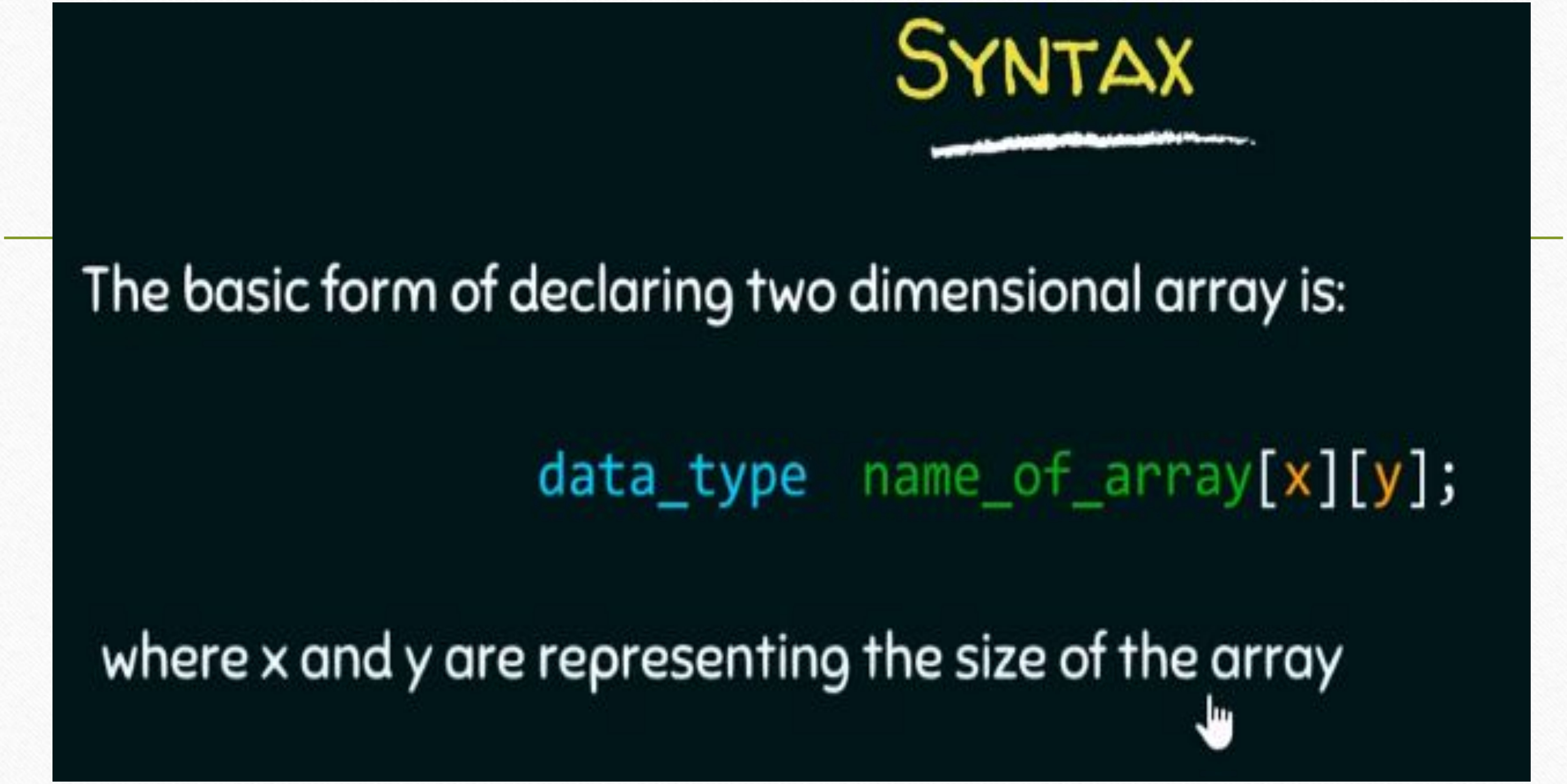
2-D arrays

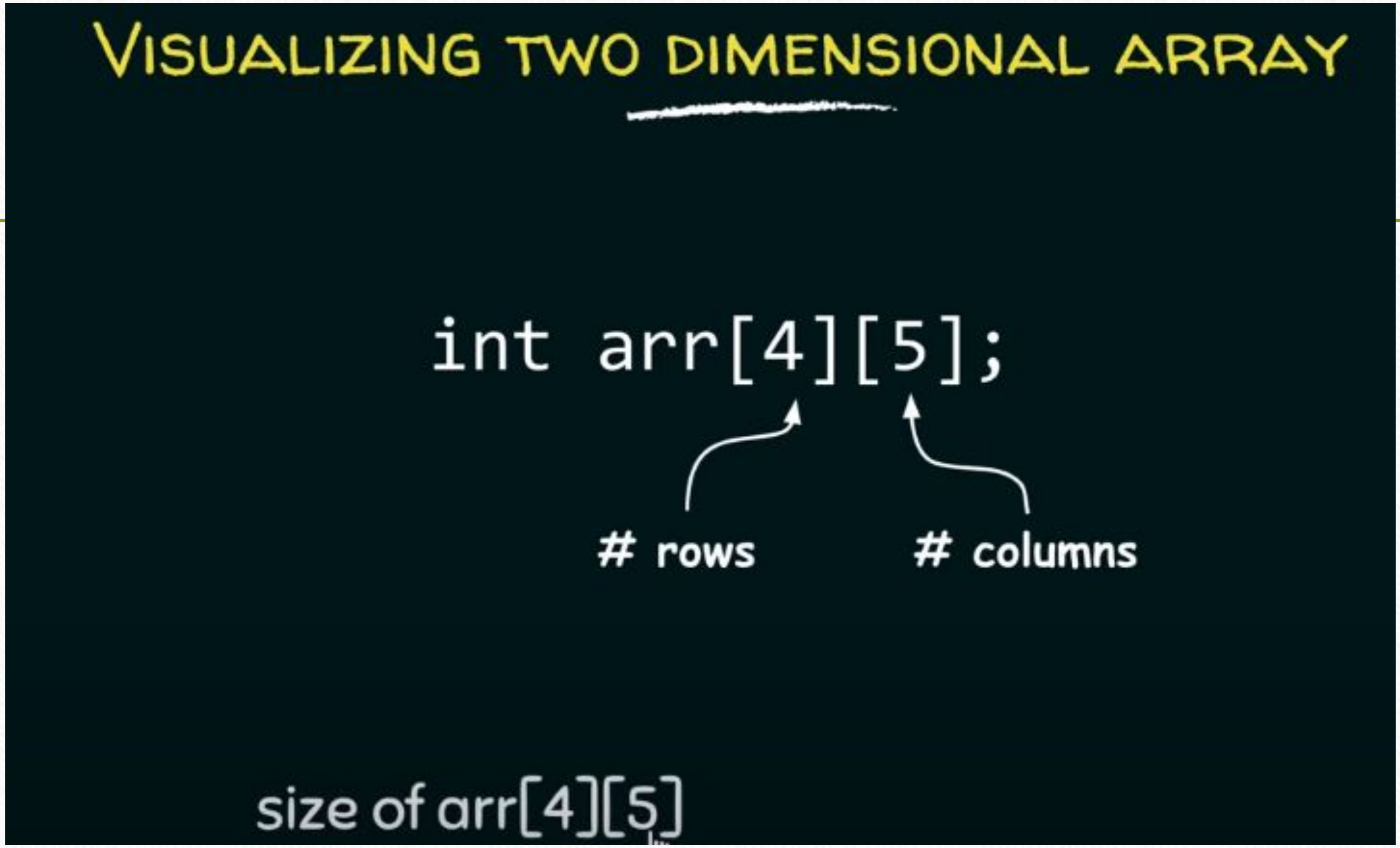


Syntax

The basic form of declaring two dimensional array is:

```
data_type  name_of_array[x][y];
```

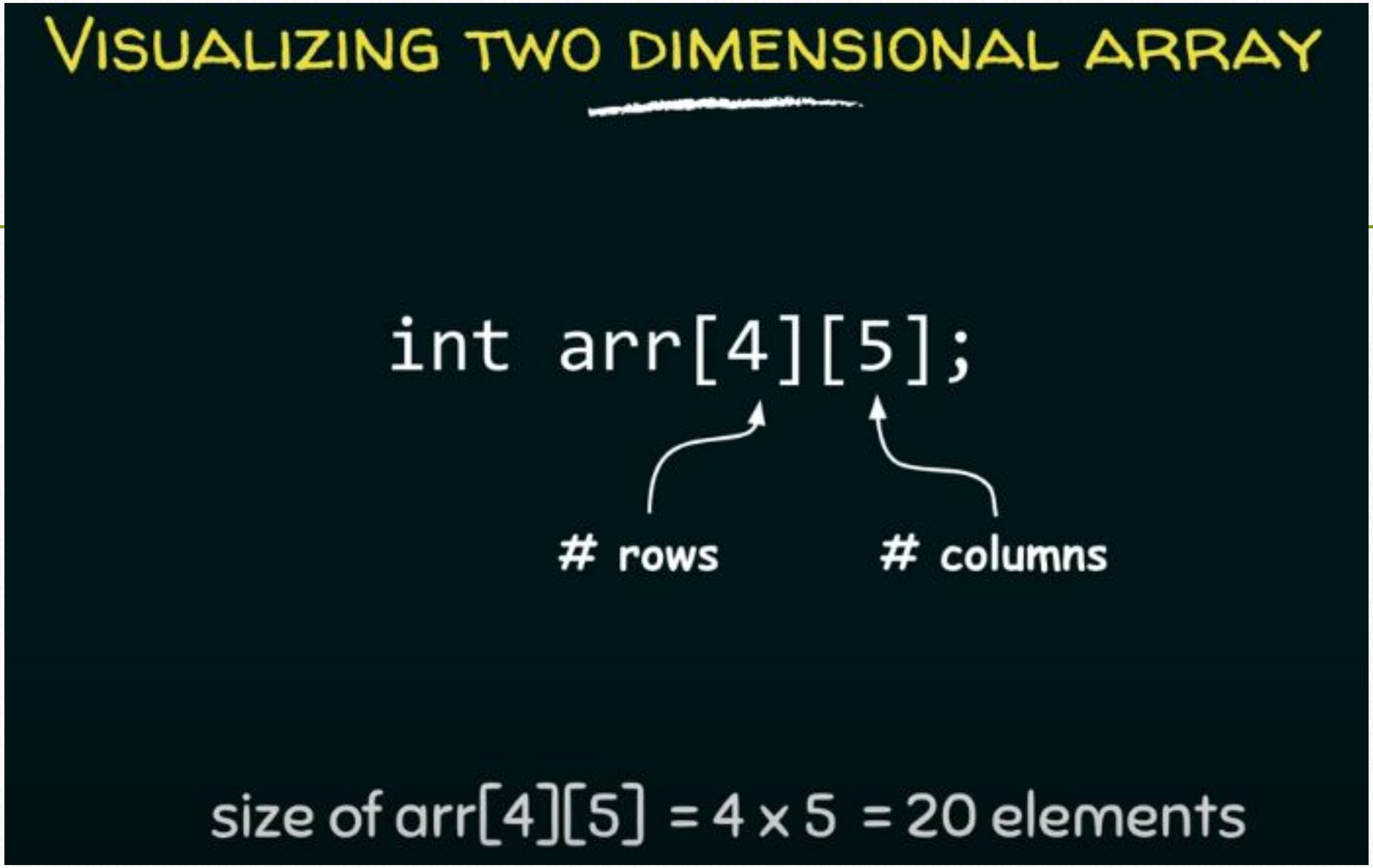
where x and y are representing the size of the array



VISUALIZING TWO DIMENSIONAL ARRAY

```
int arr[4][5];
```

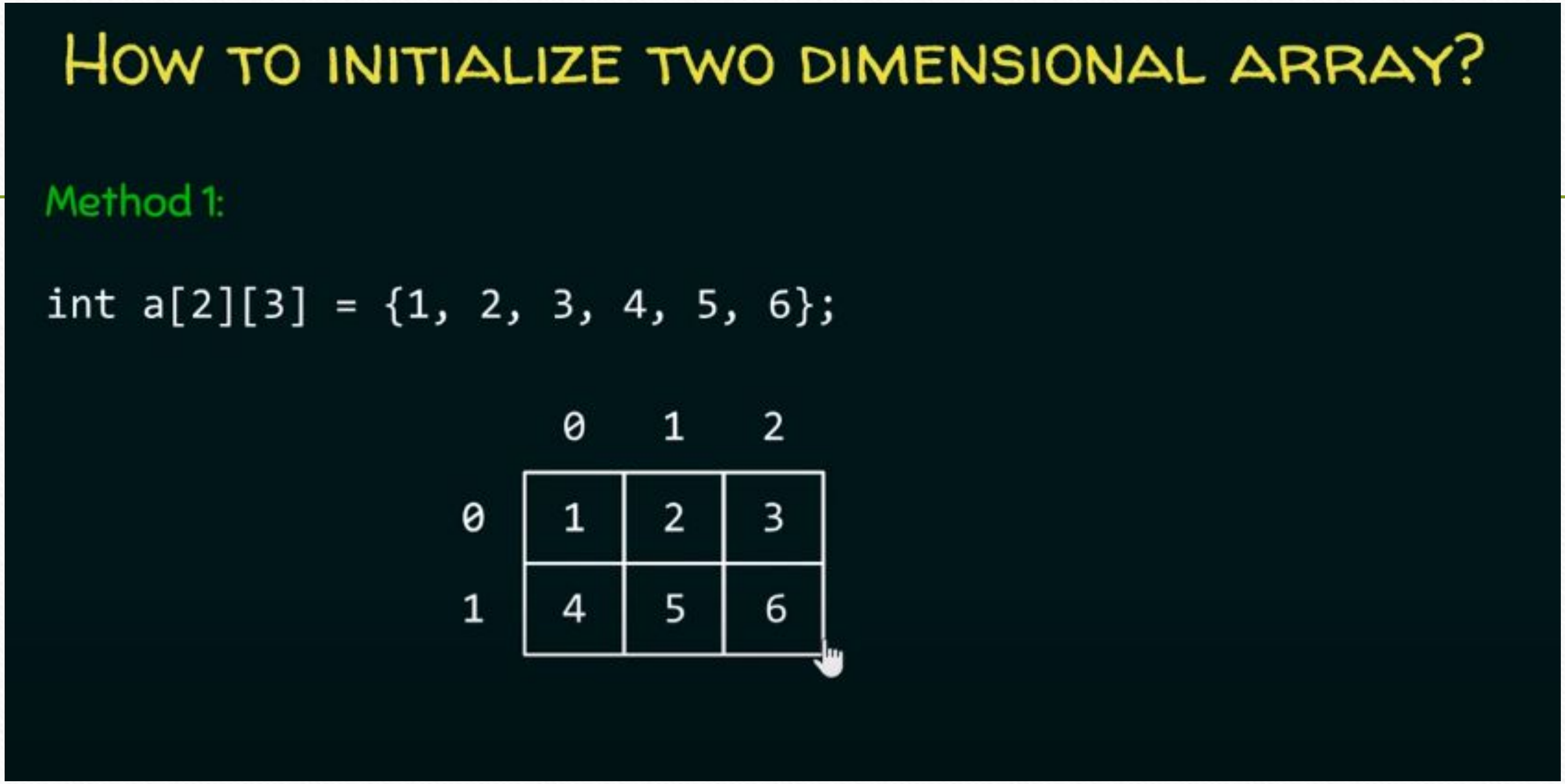
size of arr[4][5]



VISUALIZING TWO DIMENSIONAL ARRAY

```
int arr[4][5];
```

size of arr[4][5] = 4 x 5 = 20 elements

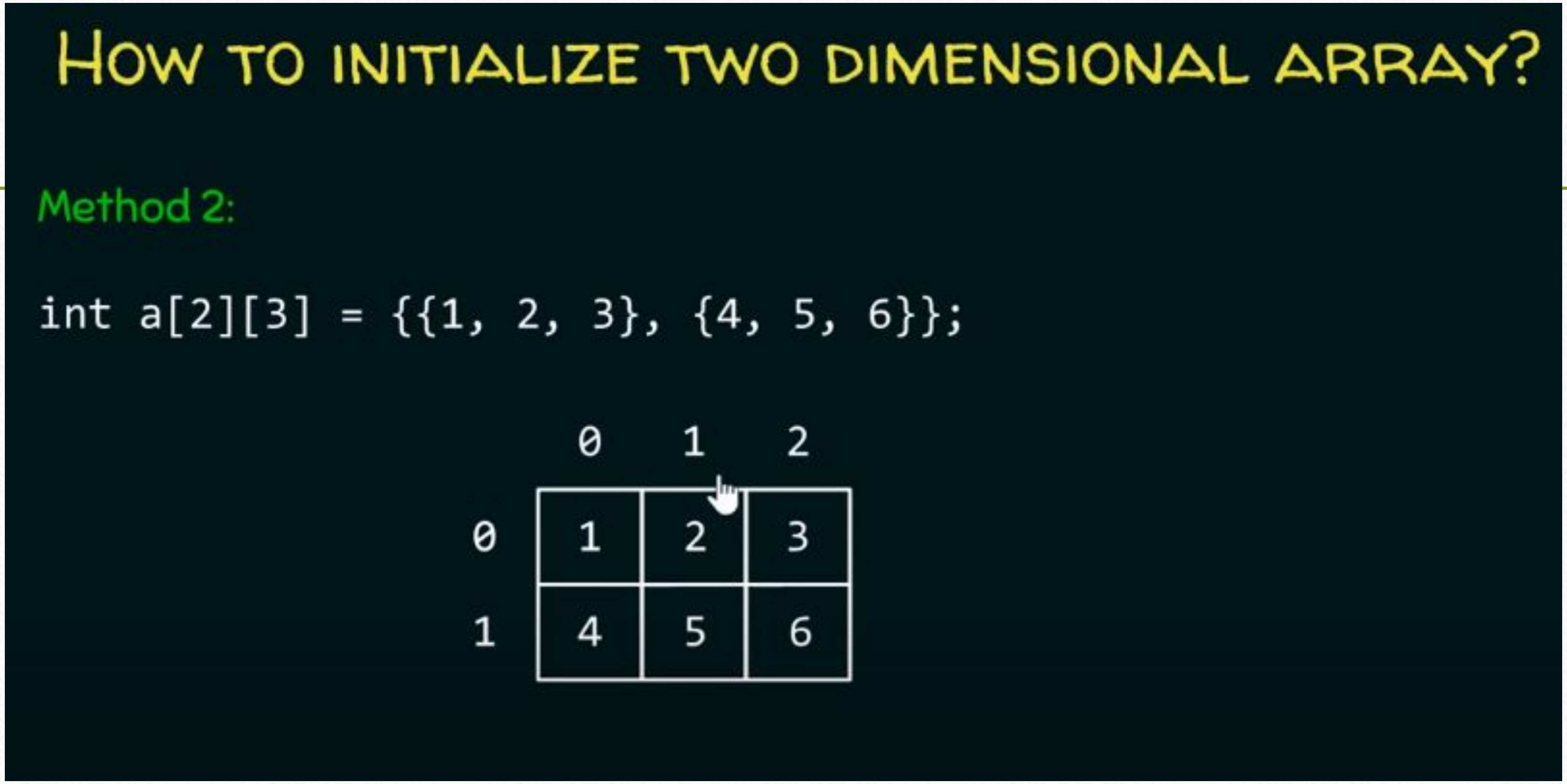


How to initialize two dimensional array?

Method 1:

```
int a[2][3] = {1, 2, 3, 4, 5, 6};
```

	0	1	2
0	1	2	3
1	4	5	6

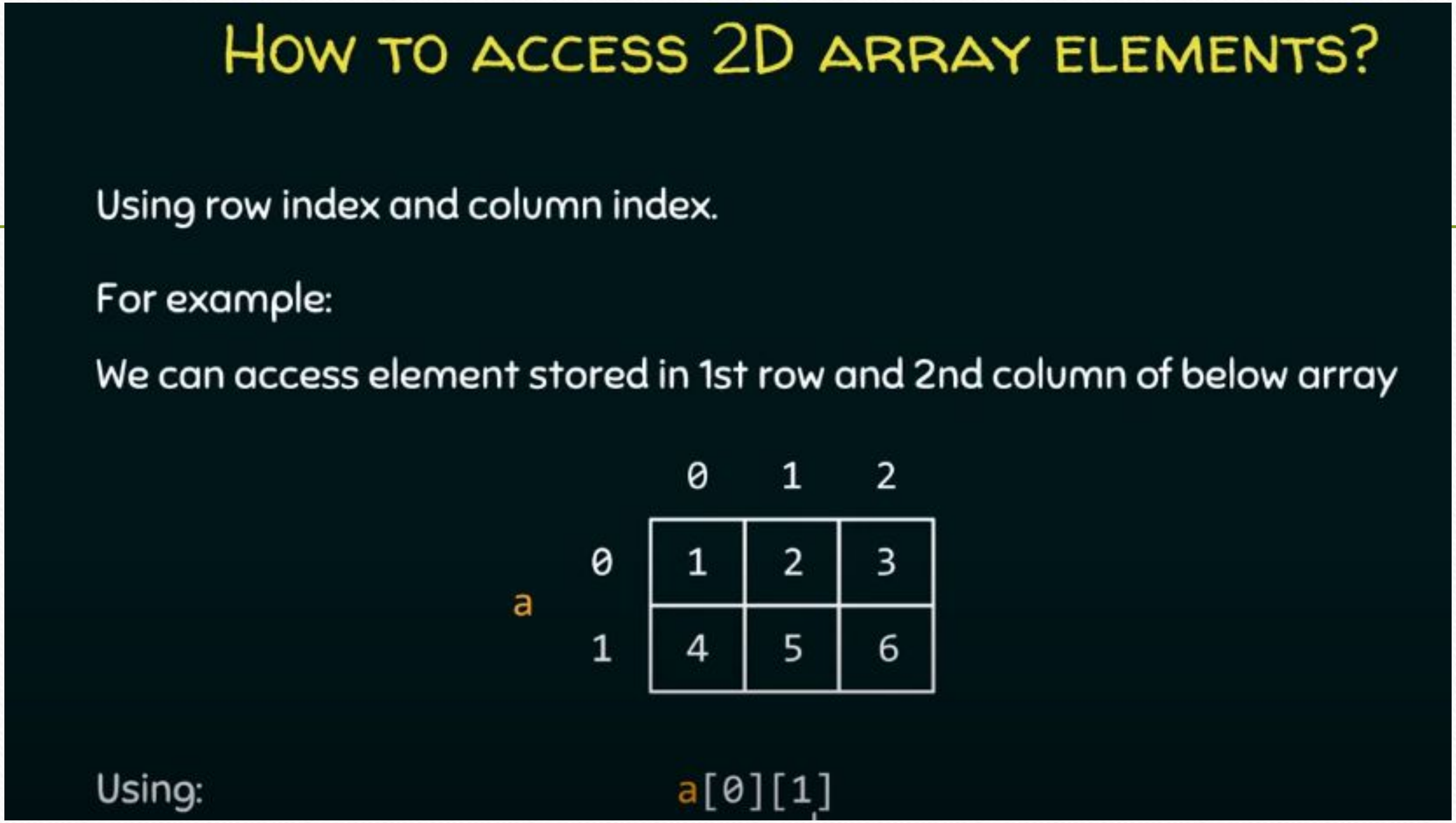


HOW TO INITIALIZE TWO DIMENSIONAL ARRAY?

Method 2:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

	0	1	2
0	1	2	3
1	4	5	6



How to access 2D array elements?

Using row index and column index.

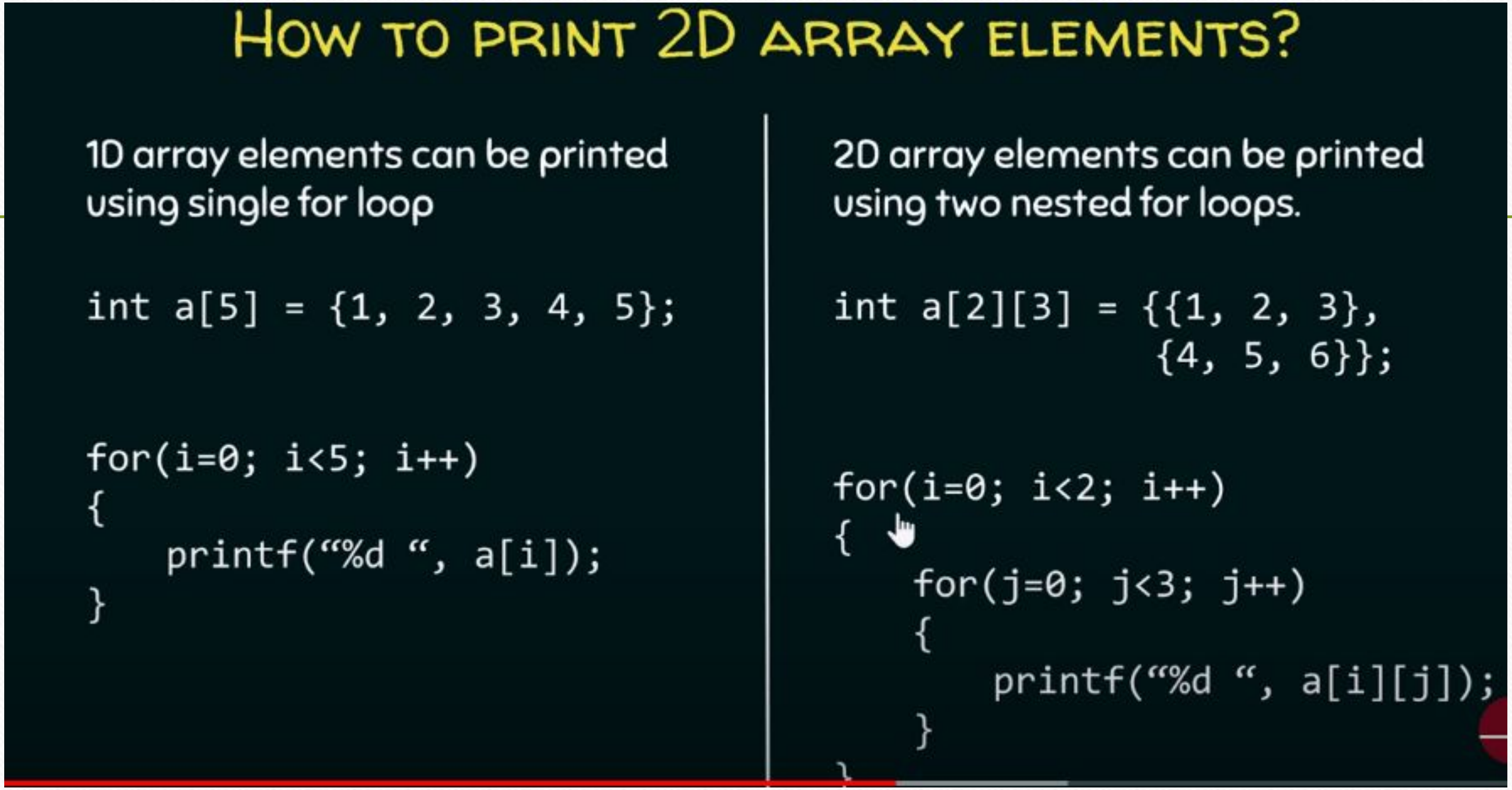
For example:

We can access element stored in 1st row and 2nd column of below array

		0	1	2
a	0	1	2	3
	1	4	5	6

Using:

```
a[0][1]
```



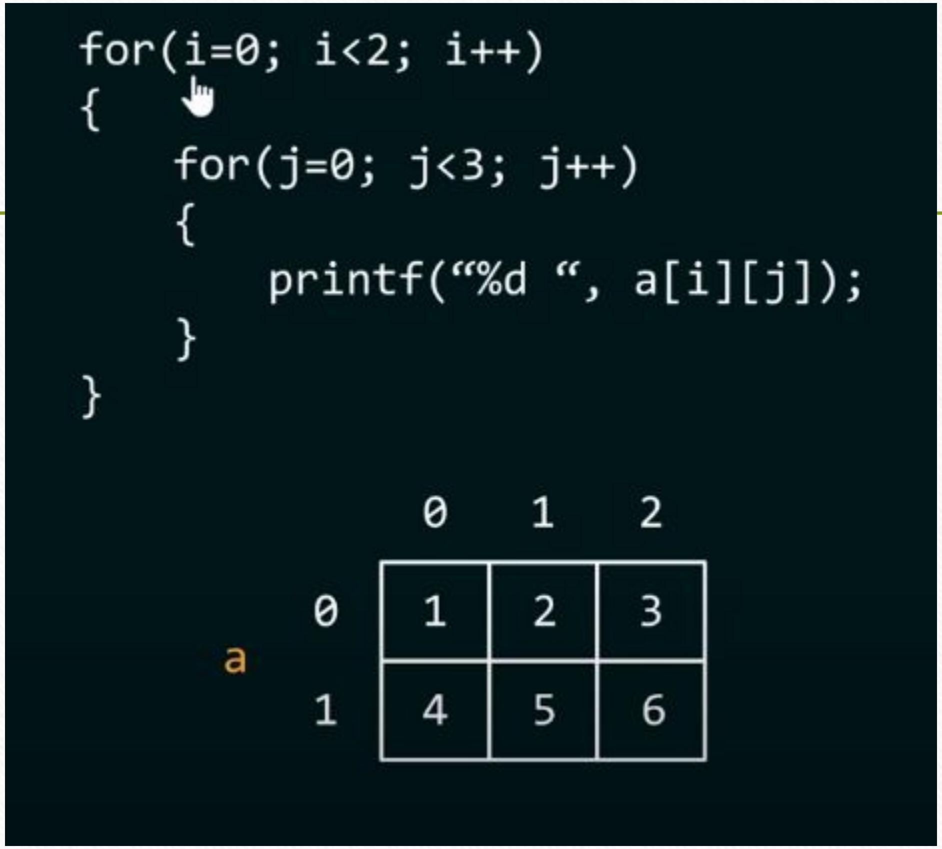
How to print 2D array elements?

1D array elements can be printed using single for loop

```
int a[5] = {1, 2, 3, 4, 5};


for(i=0; i<5; i++)
{
    printf(“%d “, a[i]);
}
```

2D array elements can be printed using two nested for loops.

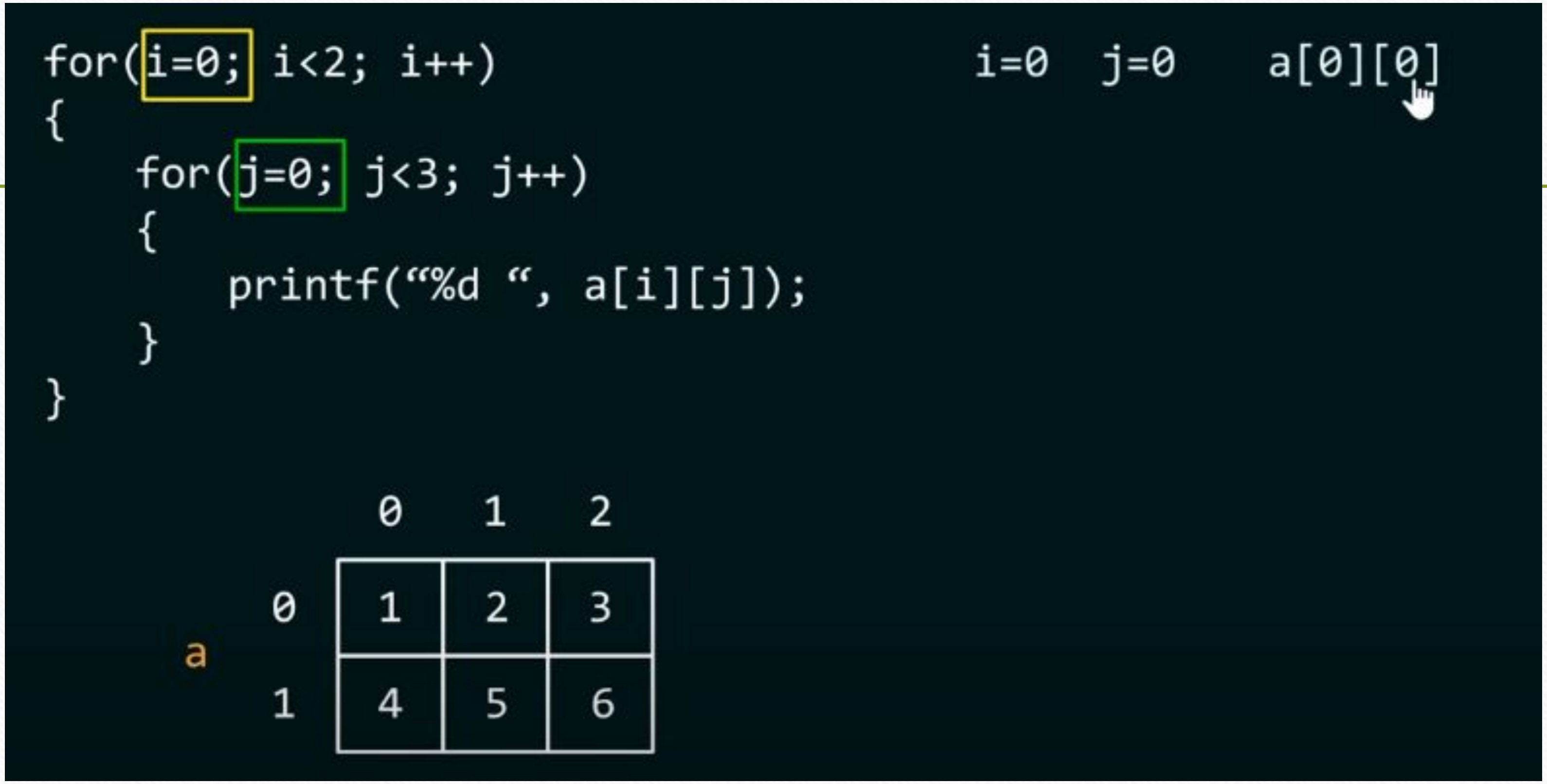
```
int a[2][3] = {{1, 2, 3},
                {4, 5, 6}};


for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```



```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

a

	0	1	2
0	1	2	3
1	4	5	6

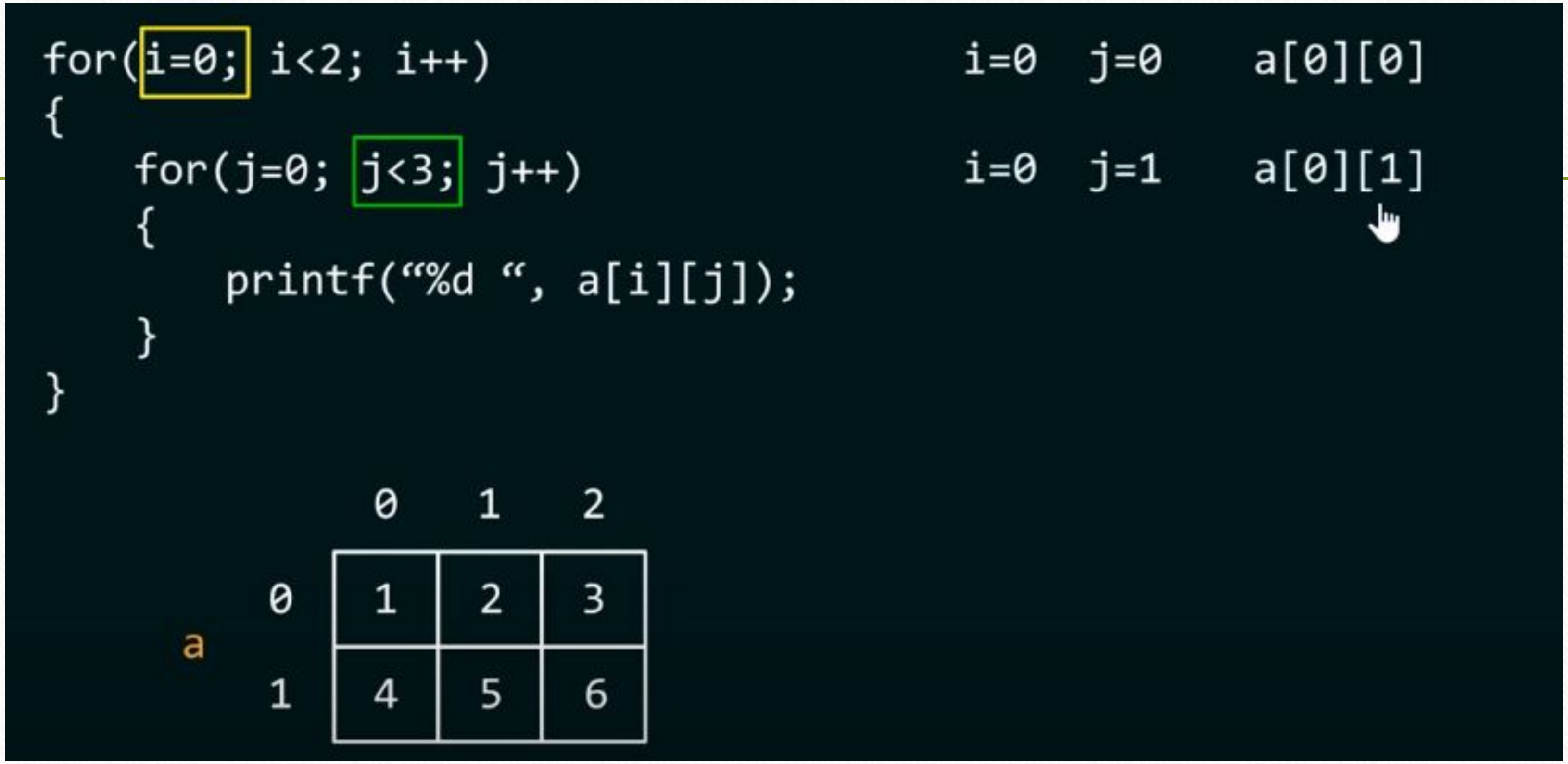


```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

```
i=0   j=0    a[0][0]
```

a

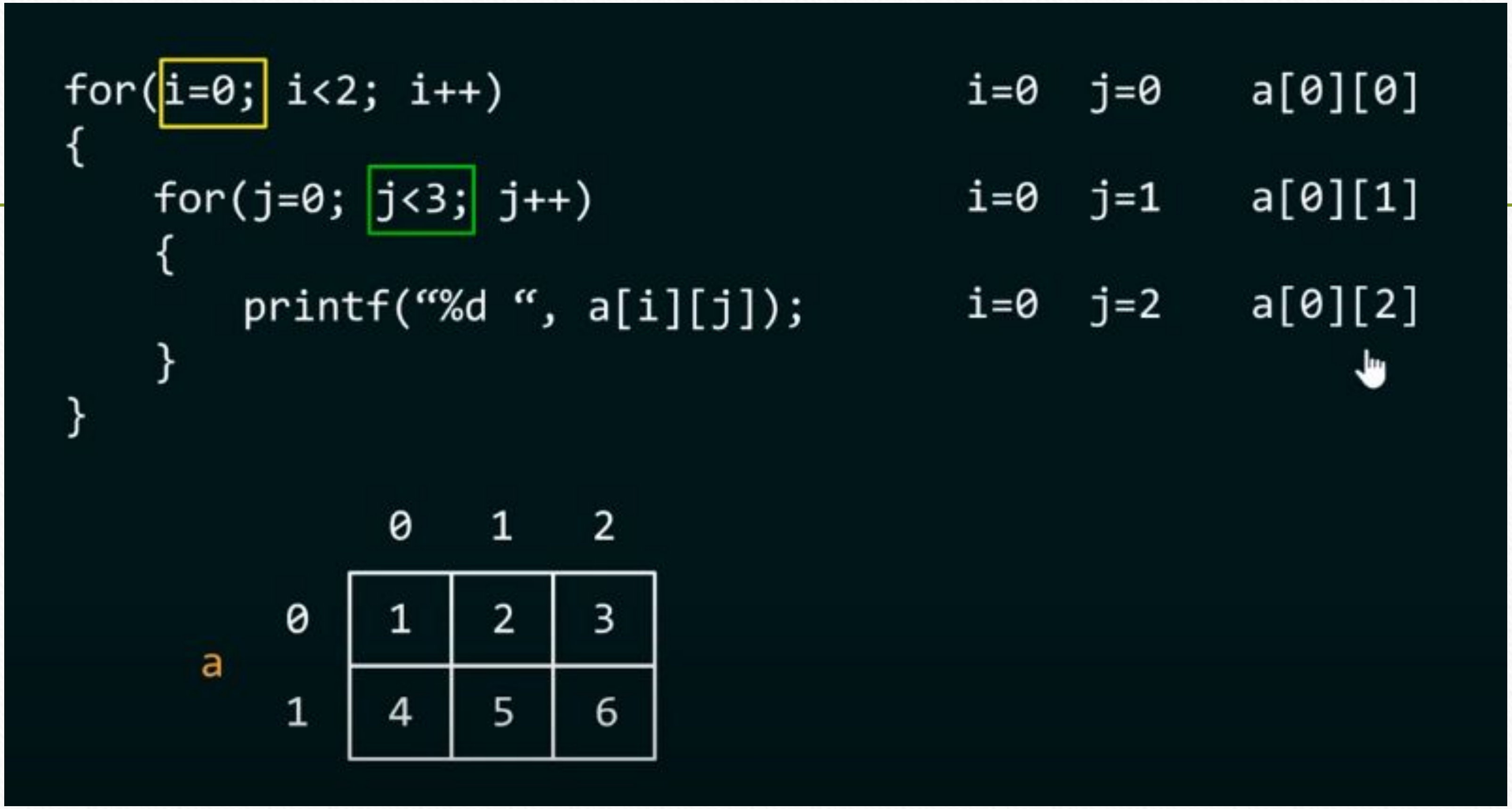
	0	1	2
0	1	2	3
1	4	5	6



```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

```
i=0  j=0    a[0][0]

i=0  j=1    a[0][1]
```

		0	1	2
a	0	1	2	3
	1	4	5	6

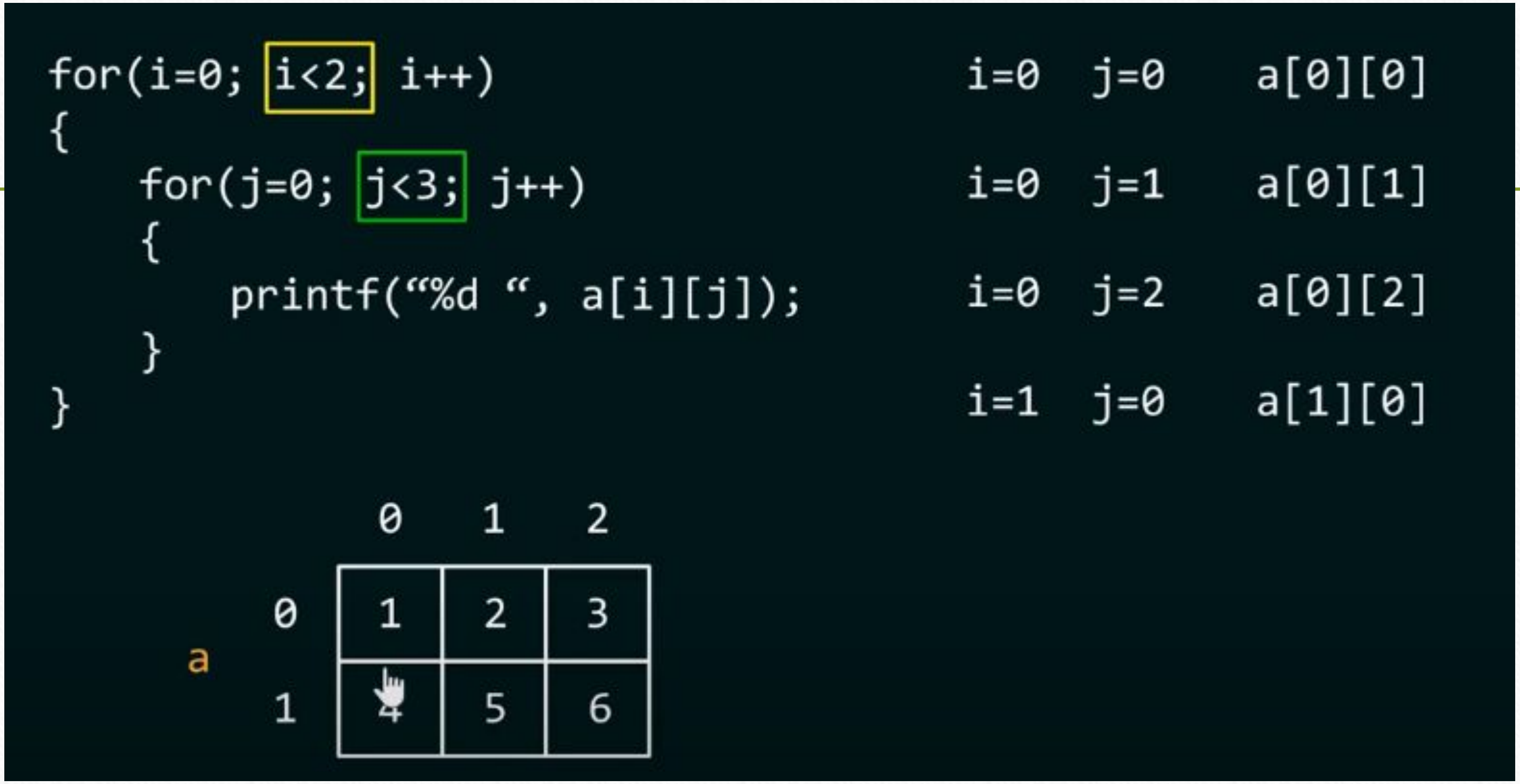


```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

```
i=0  j=0   a[0][0]

i=0  j=1   a[0][1]

i=0  j=2   a[0][2]
```

a

	0	1	2
0	1	2	3
1	4	5	6

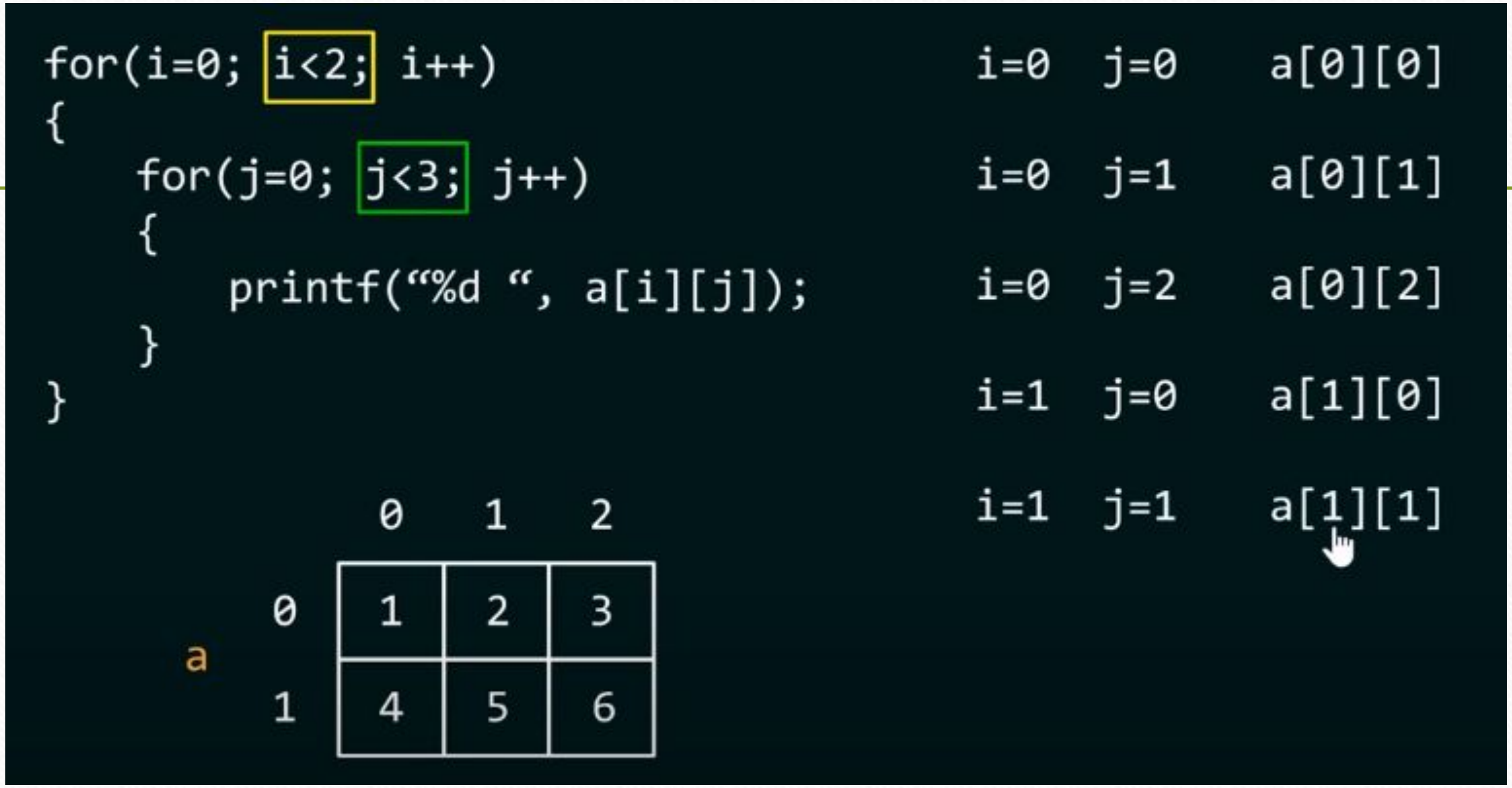


```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

```
i=0   j=0    a[0][0]

i=0   j=1    a[0][1]

i=0   j=2    a[0][2]

i=1   j=0    a[1][0]
```

a

	0	1	2
0	1	2	3
1	4	5	6

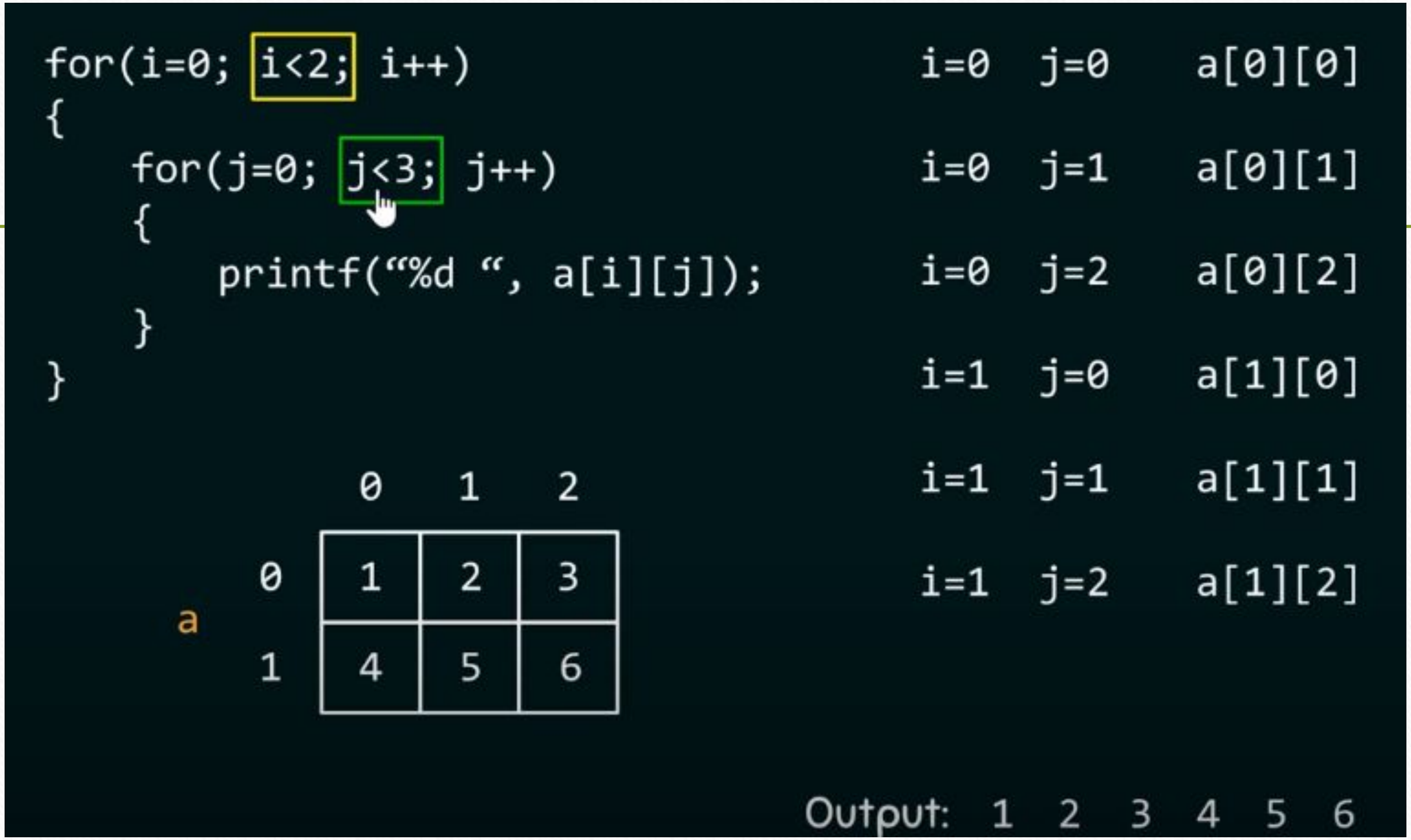


```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

```
i=0  j=0   a[0][0]

i=0  j=1   a[0][1]

i=0  j=2   a[0][2]

i=1  j=0   a[1][0]

i=1  j=1   a[1][1]
```

a

	0	1	2
0	1	2	3
1	4	5	6

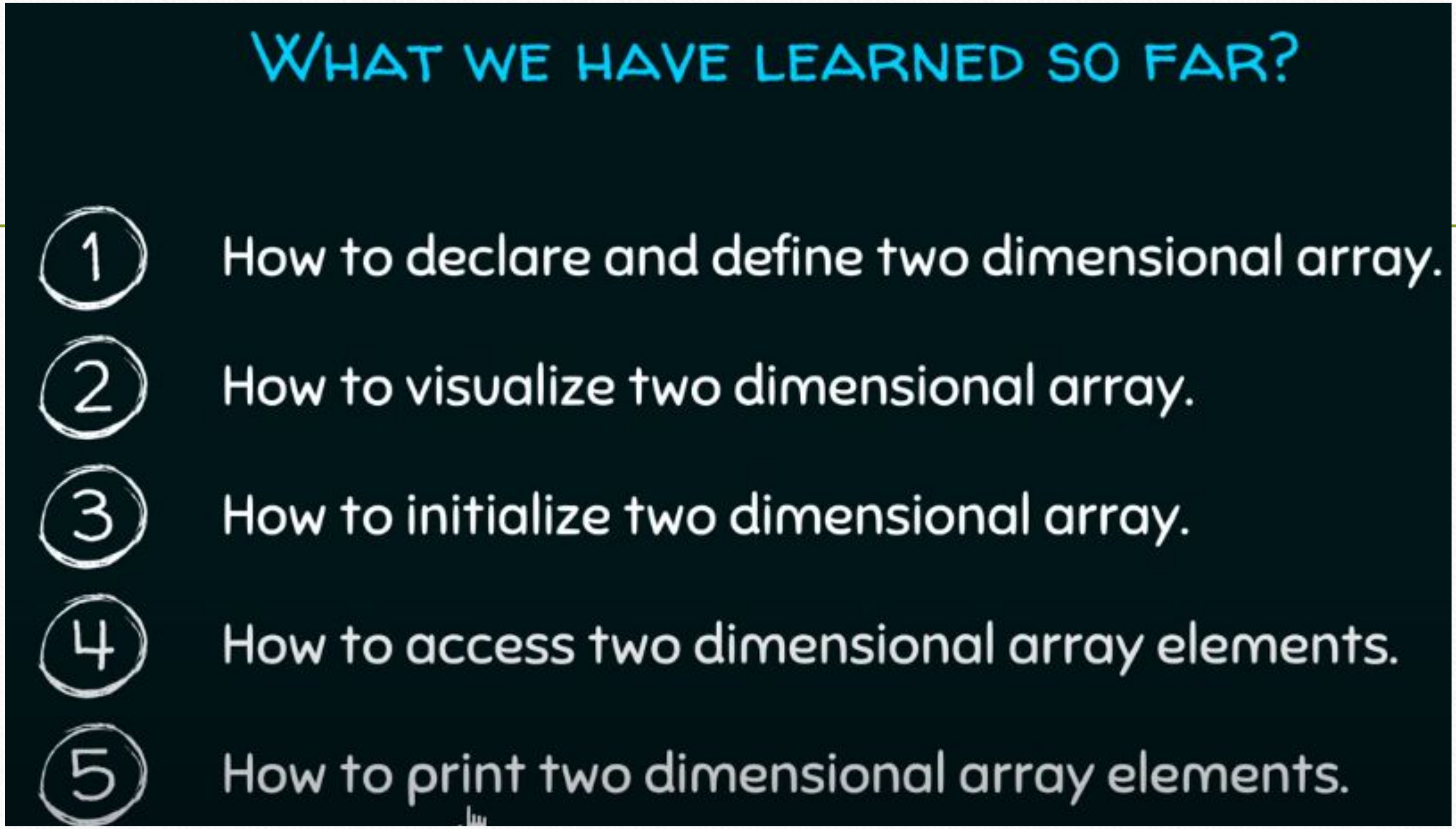


```
for(i=0; i<2; i++)
{
    for(j=0; j<3; j++)
    {
        printf(“%d “, a[i][j]);
    }
}
```

		0	1	2
a	0	1	2	3
	1	4	5	6

i=0	j=0	a[0][0]
i=0	j=1	a[0][1]
i=0	j=2	a[0][2]
i=1	j=0	a[1][0]
i=1	j=1	a[1][1]
i=1	j=2	a[1][2]

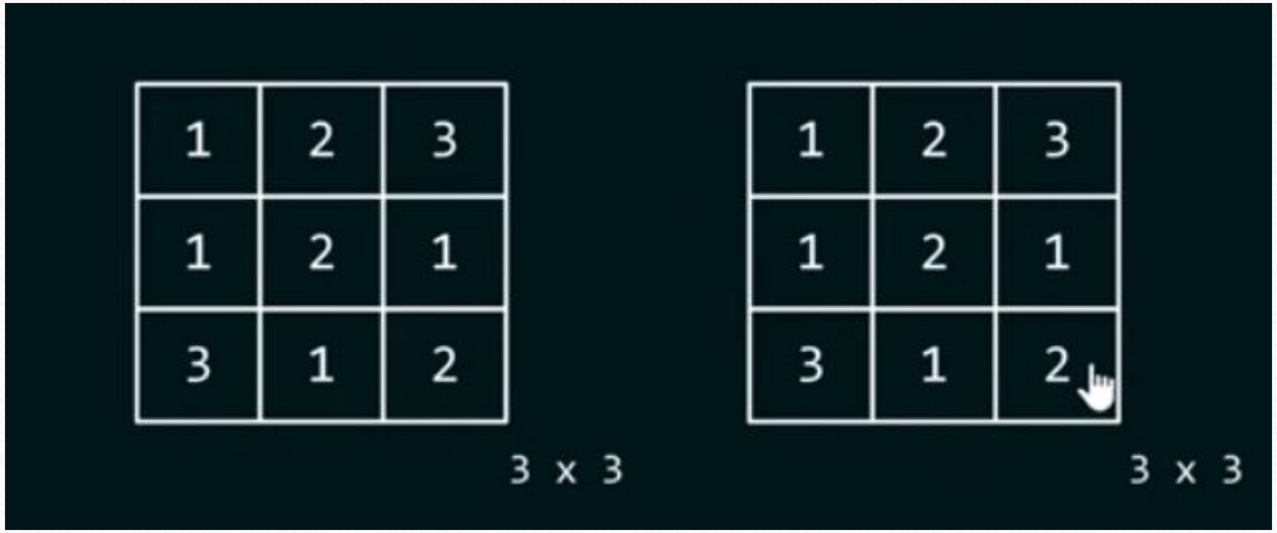
Output: 1 2 3 4 5 6



WHAT WE HAVE LEARNED SO FAR?

1. How to declare and define two dimensional array.
2. How to visualize two dimensional array.
3. How to initialize two dimensional array.
4. How to access two dimensional array elements.
5. How to print two dimensional array elements.

Matrix Multiplication

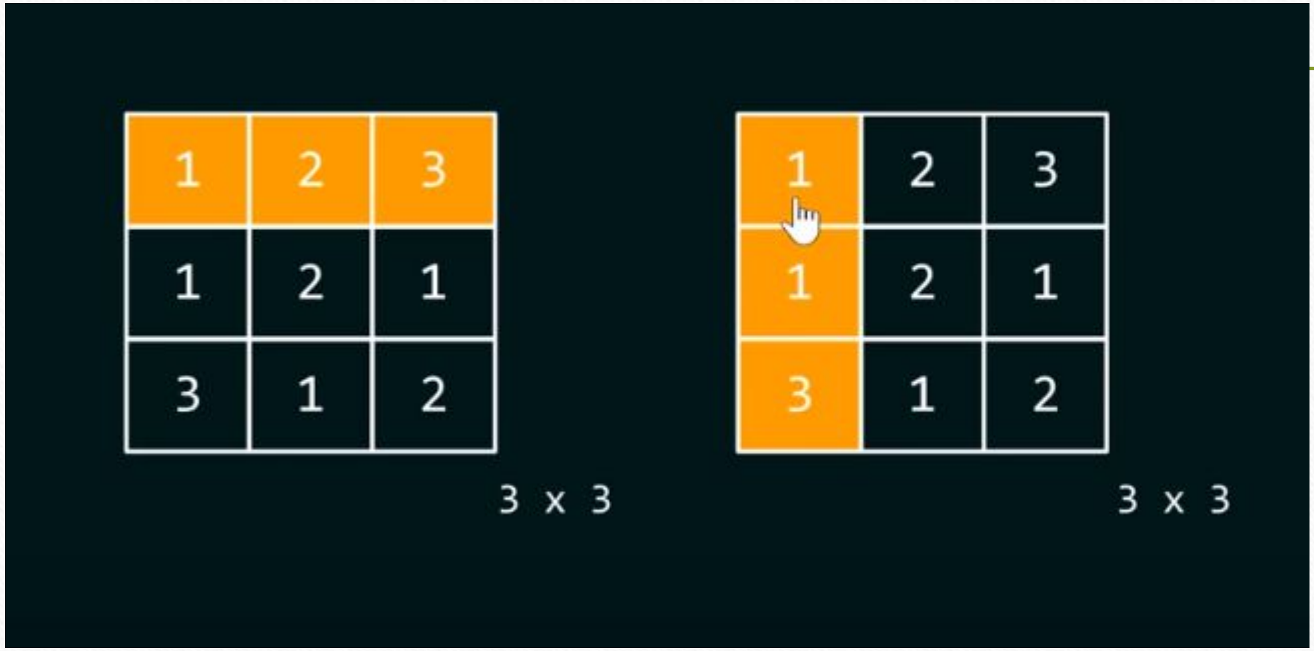


1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

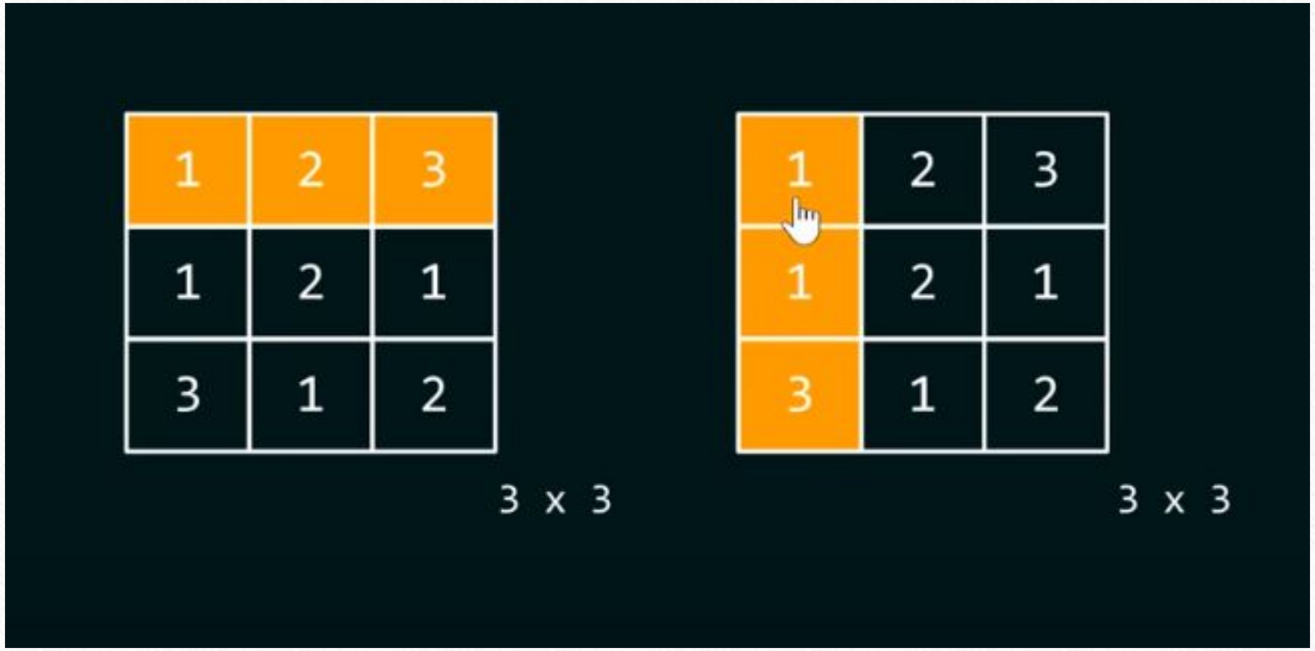


1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

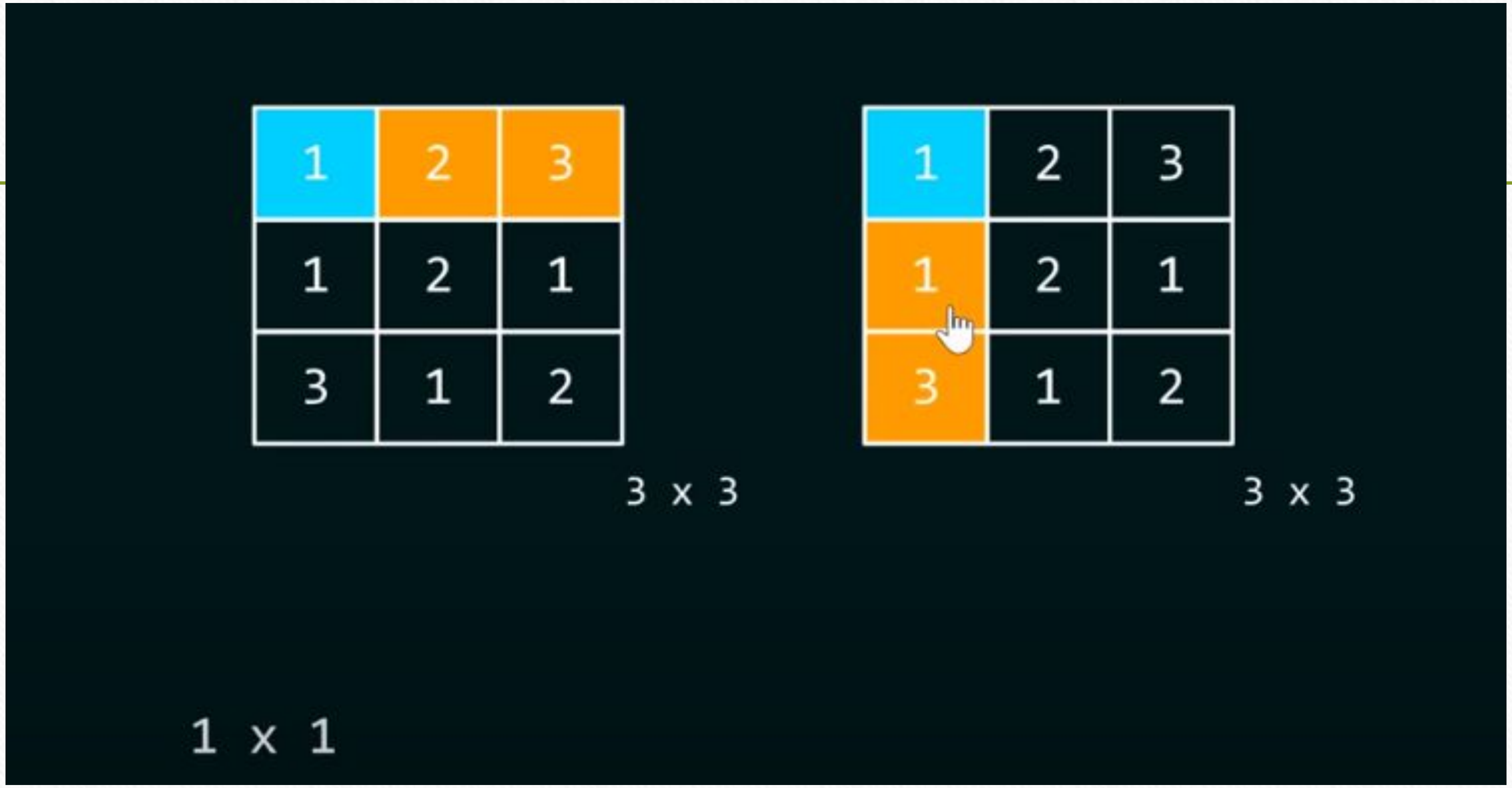


1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3



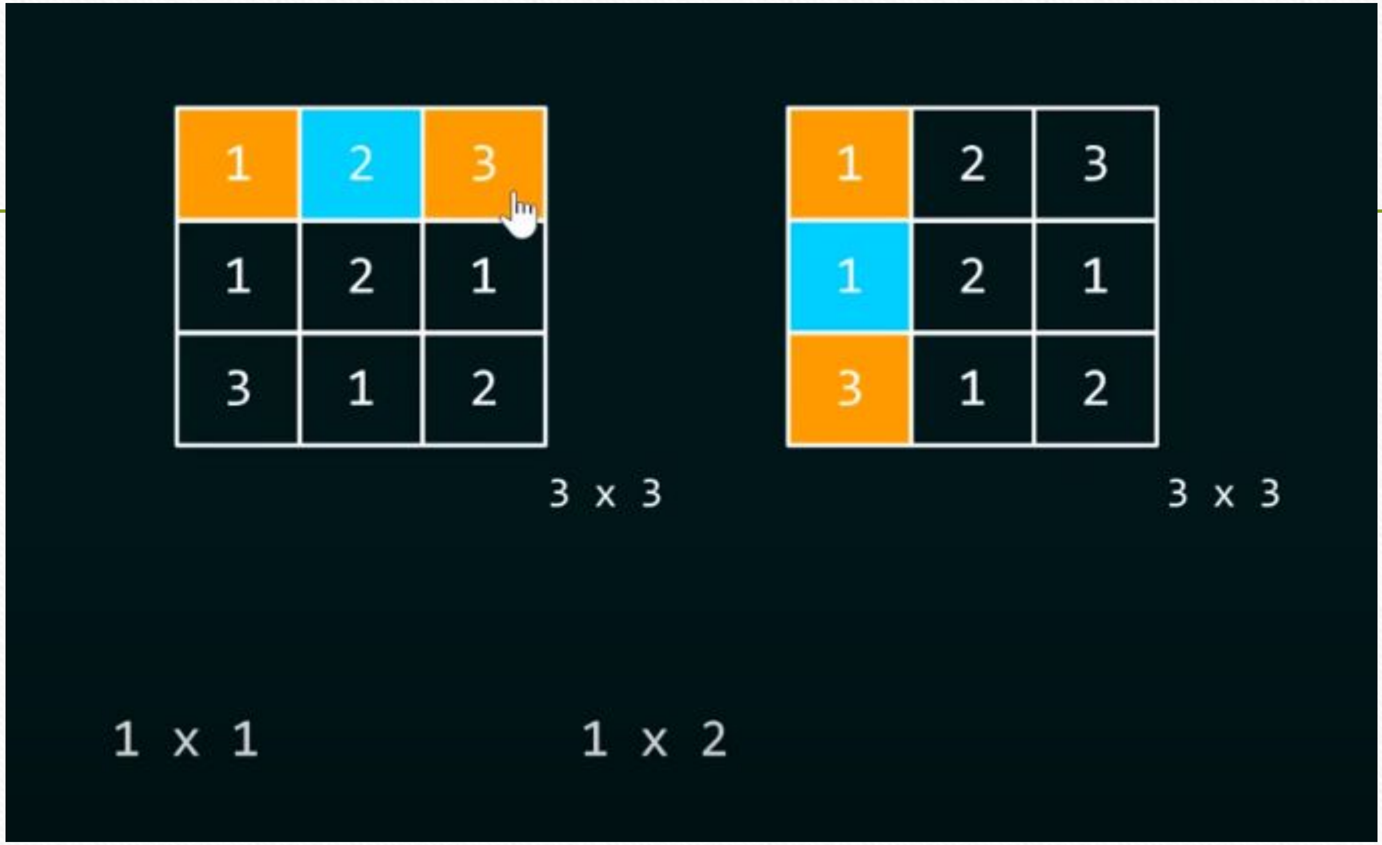
1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

1 x 1



1	2	3
1	2	1
3	1	2

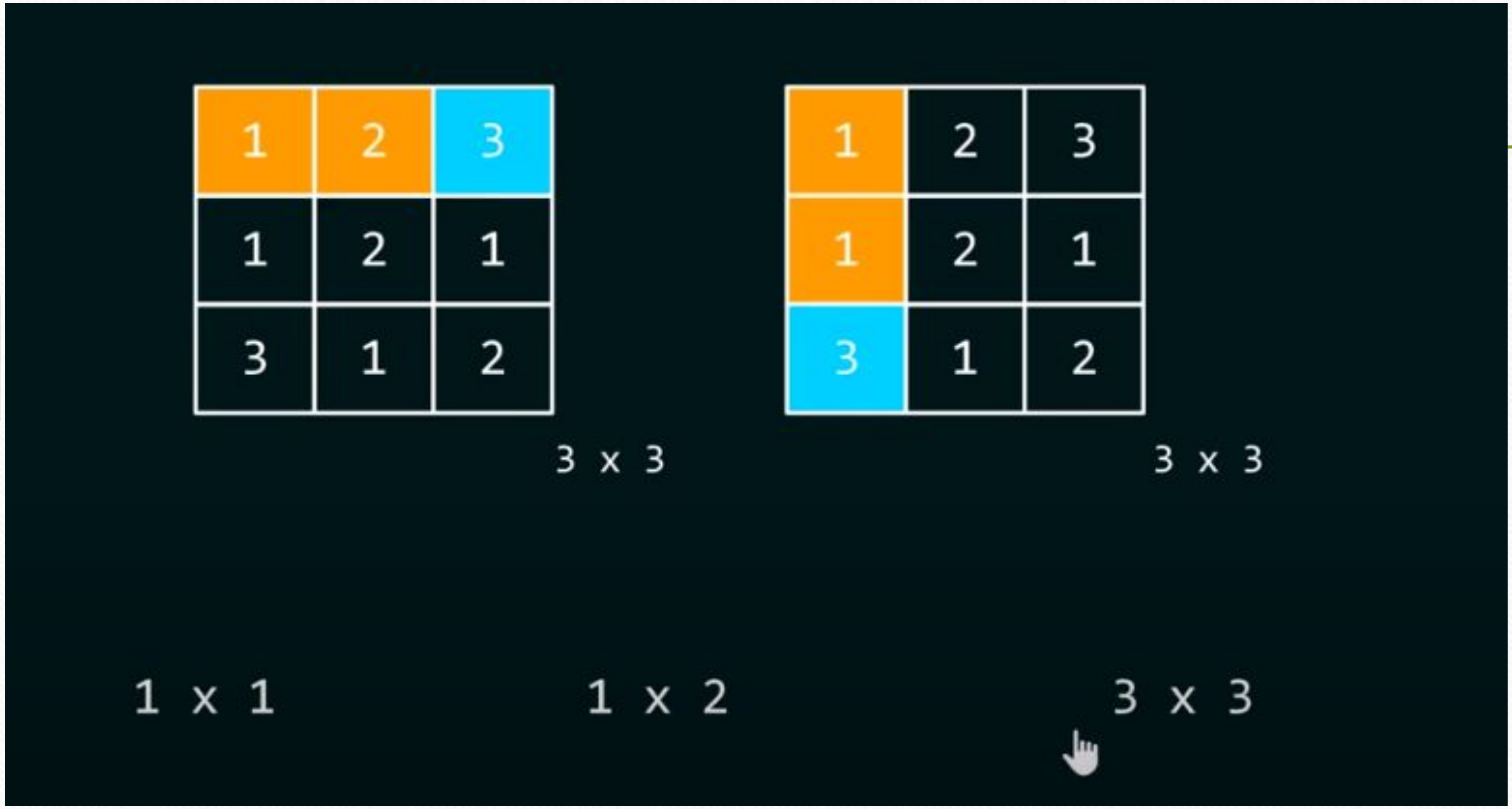
3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

1 x 1

1 x 2



1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

1 x 1

1 x 2

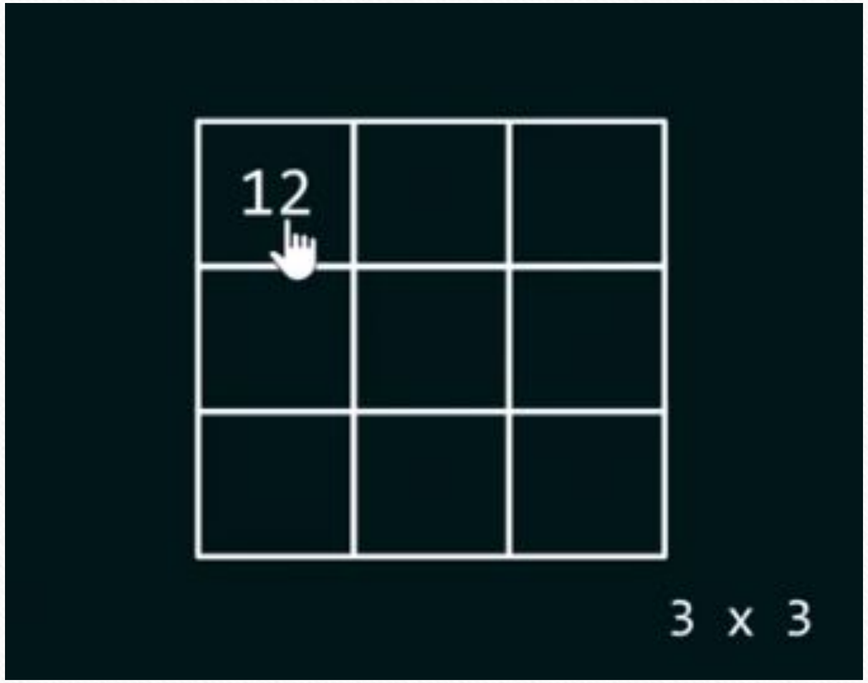
3 x 3

mul[0][0]=

a[0][0]*b[0][0]+

a[0][1]*b[1][0]+

a[0][2]*b[2][0]



12		

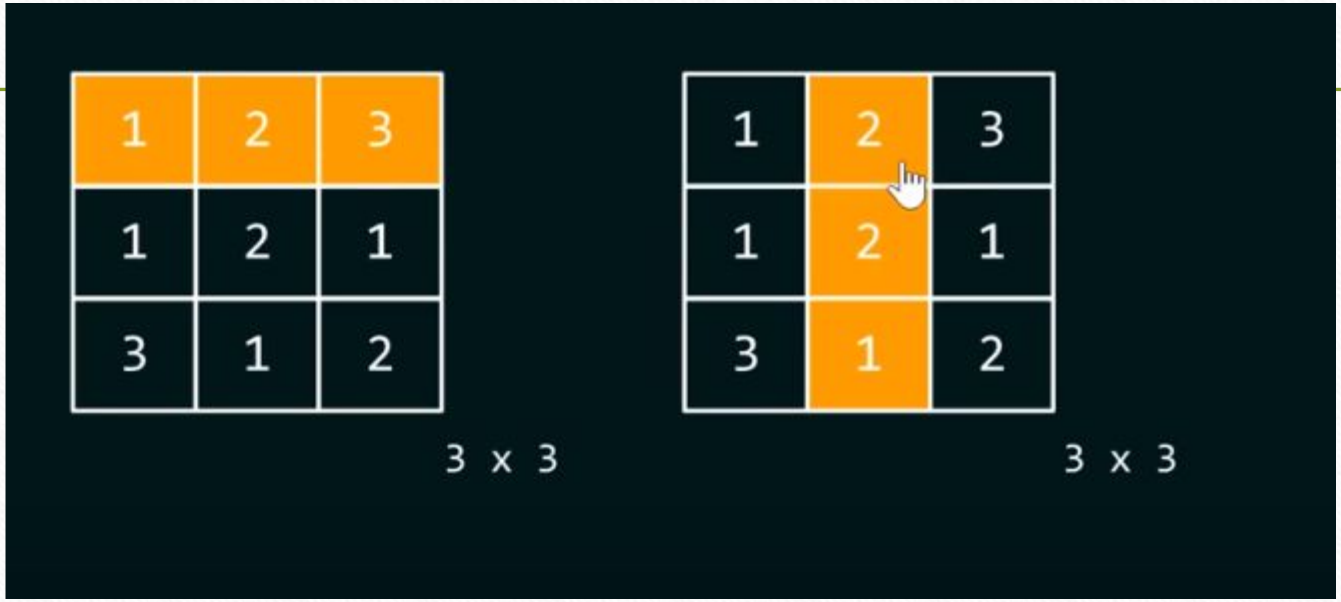
3 x 3

mul[0][0]=

a[0][0]*b[0][0]+

a[0][1]*b[1][0]+

a[0][2]*b[2][0]

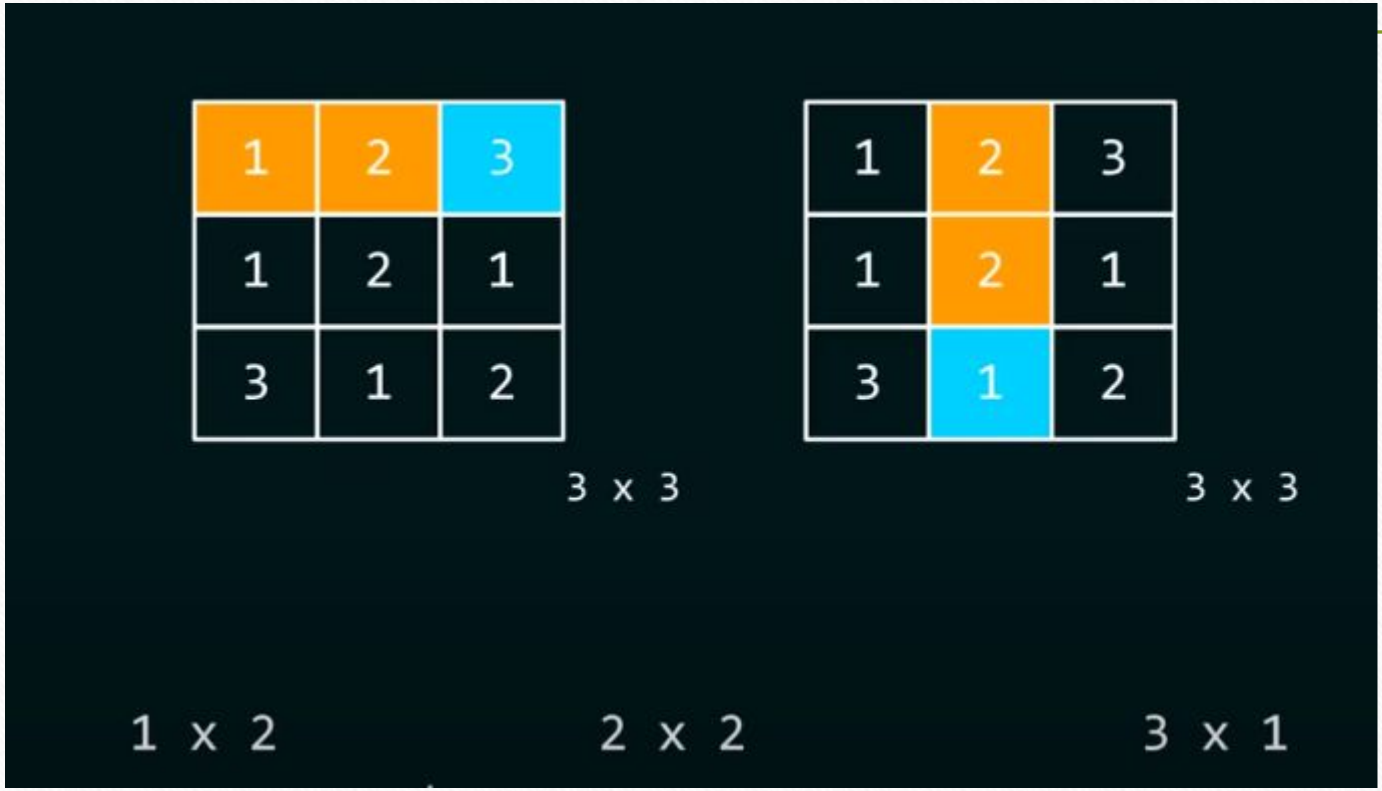


1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3



1	2	3
1	2	1
3	1	2

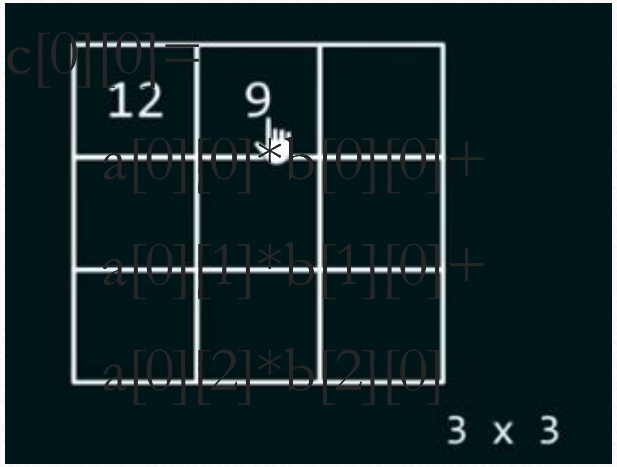
1	2	3
1	2	1
3	1	2

mul[0][1]=

a[0][0]*b[0][1]+

a[0][1]*b[1][1]+

a[0][2]*b[2][1]



c[0][0]=

a[0][0]*b[0][0]+

a[0][1]*b[1][0]+

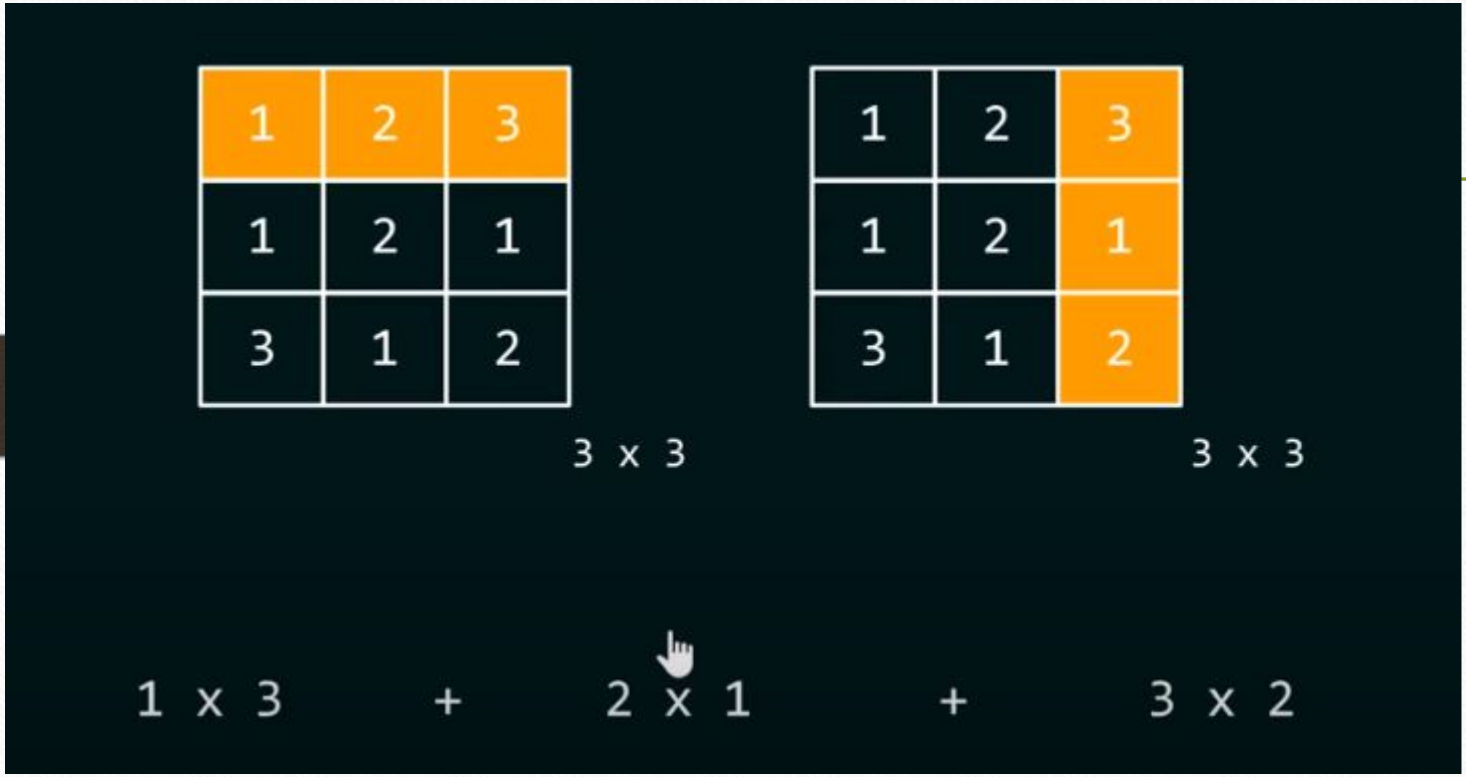
a[0][2]*b[2][0]

mul[0][1]=

a[0][0]*b[0][1]+

a[0][1]*b[1][1]+

a[0][2]*b[2][1]



1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

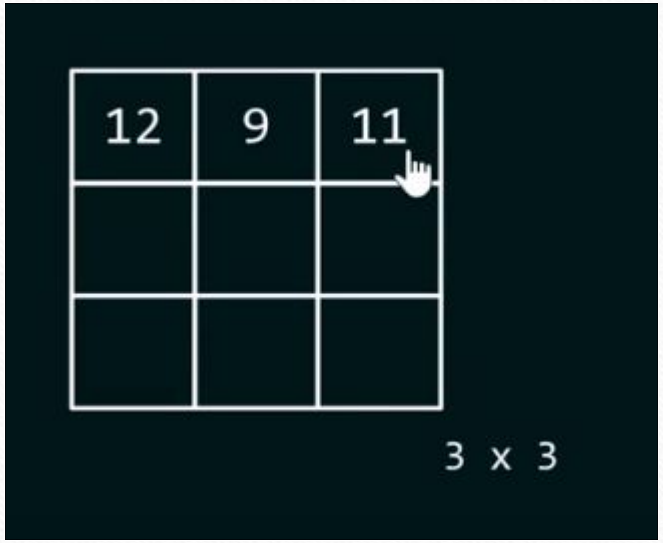
1 x 3 + 2 x 1 + 3 x 2

mul[0][2]=

a[0][0]*b[0][2]+

a[0][1]*b[1][2]+

a[0][2]*b[2][2]



12	9	11

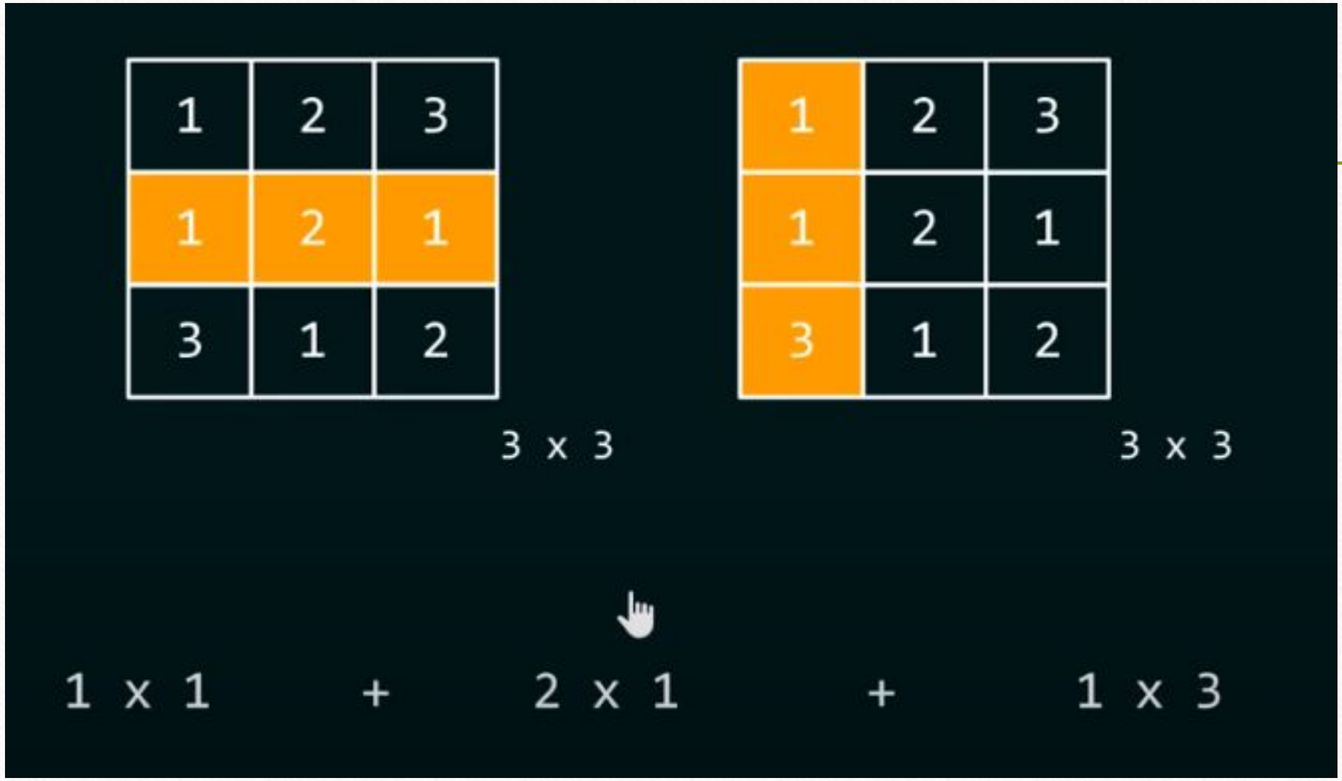
3 x 3

mul[0][2]=

a[0][0]*b[0][2]+

a[0][1]*b[1][2]+

a[0][2]*b[2][2]



1	2	3
1	2	1
3	1	2

3 x 3

1	2	3
1	2	1
3	1	2

3 x 3

1 x 1 + 2 x 1 + 1 x 3

mul[1][0]=

a[1][0]*b[0][0]+

a[1][1]*b[1][0]+

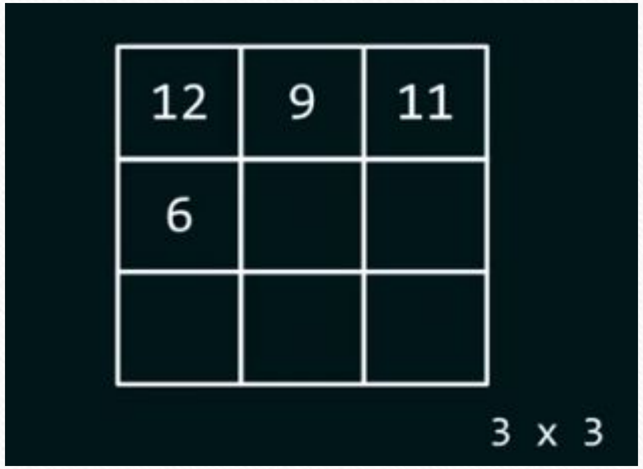
a[1][2]*b[2][0]

mul[i][j]=

a[i][0]*b[0][j]+

a[i][1]*b[1][j]+

a[i][2]*b[2][j]



12	9	11
6		

3 x 3

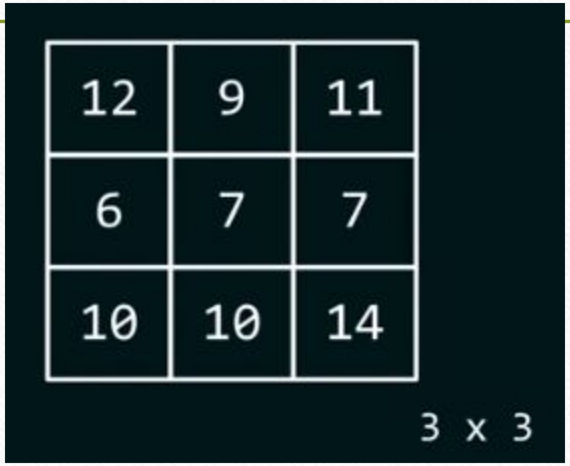
mul[i][j]=

a[i][k]*b[k][j]+

a[i][k]*b[k][j]+

a[i][k]*b[k][j]

mul[i][j]=mul[i][j]+ a[i][k]* b[k][j]



12	9	11
6	7	7
10	10	14

3 x 3

b) C-Program to perform multiplication of two matrices.

```
#include<stdio.h>
int main()
{
    int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
    printf("enter the number of row=");
    scanf("%d",&r);
    printf("enter the number of column=");
    scanf("%d",&c);
    printf("enter the first matrix element=\n");
    for(i=0;i<r;i++)
        for(j=0;j<c;j++)
            scanf("%d",&a[i][j]);
```

```
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
        for(j=0;j<c;j++)
                scanf("%d",&b[i][j]);
printf("multiply of  the matrix=\n");
for(i=0;i<r;i++)
{
        for(j=0;j<c;j++)
        {       mul[i][j]=0;
                for(k=0;k<c;k++)
                {
                        mul[i][j]+=a[i][k]*b[k][j];
                }
        }
}
```

```
//for printing result
for(i=0;i<r;i++)
{
    for(j=0;j<c;j++)
    {
        printf("%d\t",mul[i][j]);
    }
    printf("\n");
}
return 0;
}
```

C-Program to find transpose of the given matrices.

```
#include <stdio.h>
 int main()
{
       int m, n, i, j, matrix[10][10], transpose[10][10];
       printf("Enter the number of rows and columns of a matrix\n");
       scanf("%d%d", &m, &n);
       printf("Enter elements of the matrix\n");
       for (i = 0; i < m; i++)
              for (j = 0; j < n; j++)
                     scanf("%d", &matrix[i][j]);
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
            transpose[j][i] = matrix[i][j];
printf("Transpose of the matrix:\n");
for (i = 0; i < n; i++)
{
        for (j = 0; j < m; j++)
            printf("%d\t", transpose[i][j]);
        printf("\n");
}
return 0;
}
```

C-Program to find row sum and column sum and sum of all elements in a matrix.

```
#include <stdio.h>


int main()
{
      int m,n , sum=0;      //Row Column Declaration
      printf("Enter the number of rows and column\n");
      scanf("%d %d",&m,&n);    //Row Column Initialization
      int arr[m][n];   //Matrix Declaration
```

```
printf("Enter the elements of the matrix\n");
for(int i=0;i<m;i++)      //Matrix Initialization
    for(int j=0;j<n;j++)
    {     scanf("%d",&arr[i][j]);
        sum= sum+arr[i][j]
    }
printf("\nElements in the matrix are \n");
for(int i=0;i<m;i++)
{         //Print Matrix
    for(int j=0;j<n;j++)
        printf("%d ",arr[i][j]);
    printf("\n");
}
```

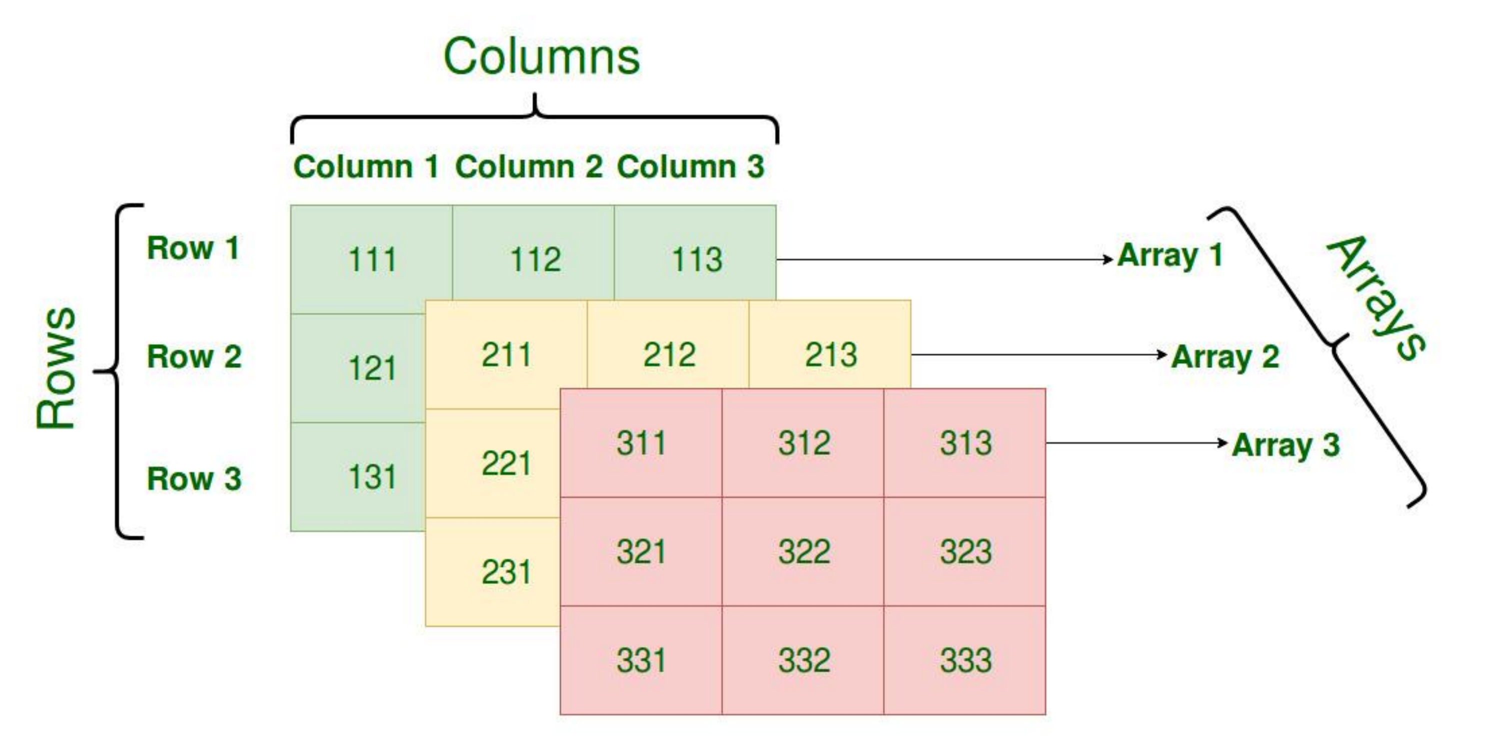
```
printf("\nRow Sum....\n");

for(inti=0;i<m;i++)
{
        int rsum=0;
        for(int j=0;j<n;j++)
        {
            rsum=rsum+arr[i][j];
        }
        printf("\nSum of all the elements in row %d is %d\n",i,rsum);
}
```

```
printf("\nColumn Sum....\n");
for(int i=0;i<m;i++)
{
        int csum=0;
        for(int j=0; j<n; j++)
        {
            csum=csum+arr[j][i];
        }
        printf("\nSum of all the elements in column %d is %d\n",i,csum);
}
printf("Sum of all elements=%d", sum);
return 0;
}
```

Multidimensional Arrays

int table [3] [3] [3]

Plane Rows Column

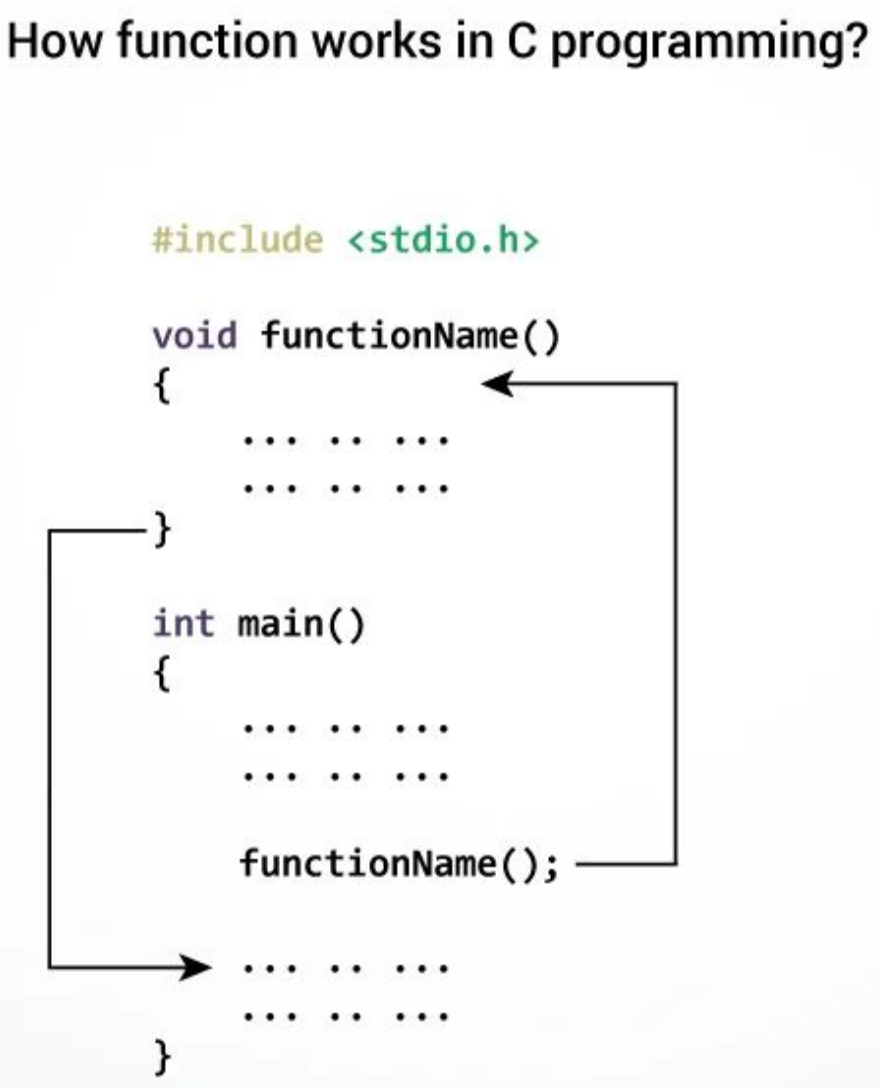


Functions

- **A Brief Overview**
- **Defining a Function**
- **User Defined Functions**
- **Function Prototypes**
- **Passing Arguments to a Function**
- **Scope-global and local; Recursion.**

Functions in C

A function in C is an independent module that will be called to do a specific task. A **called function** receives control from the **calling function**. When the called function completes its task it returns the control back to called function



How function works in C programming?

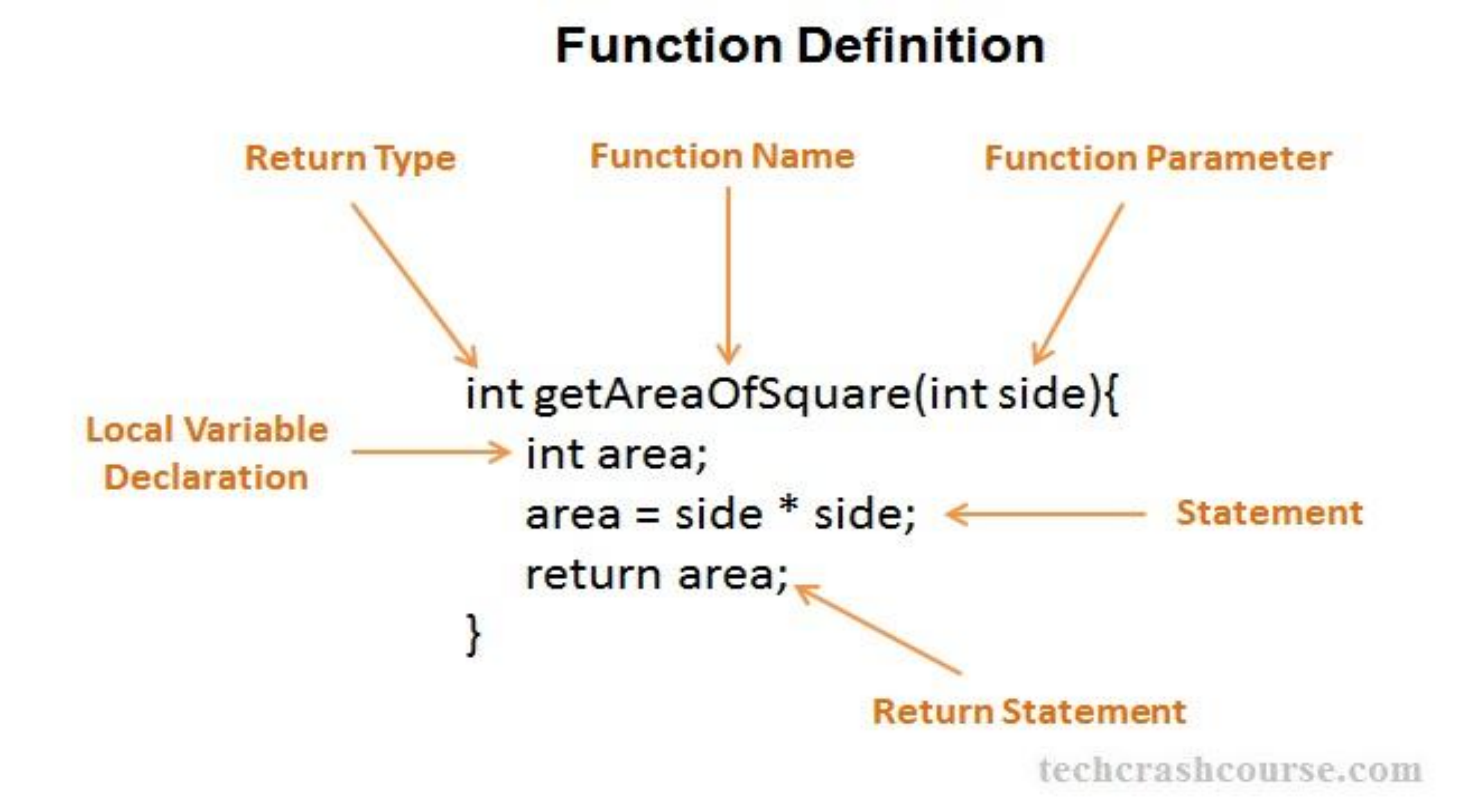
```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Types of functions

- C functions can be classified into two categories:
  - Library functions
  - User-defined functions
- **Library functions** are those functions that are already defined in the C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.
- **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

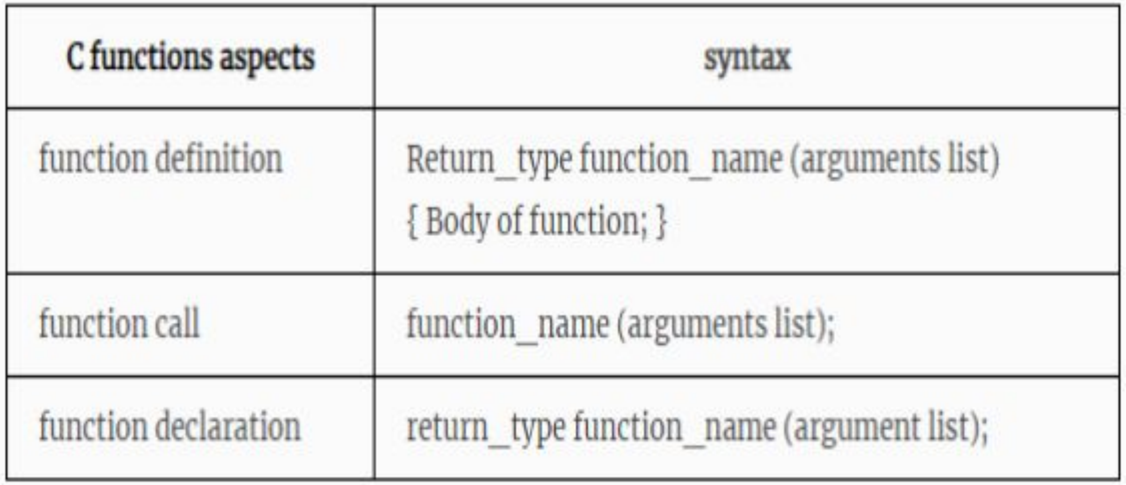
Benefits/Need of Using Functions:

- It provides **modularity** to your program's structure.
- A large C program can easily be tracked when it is divided into functions.
- It makes your **code reusable**. You just have to call the function by its name to use it, wherever required.
- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
- It makes the program more readable and easy to understand.
- It is used to avoid rewriting same logic/code again and again in a program



Function Definition

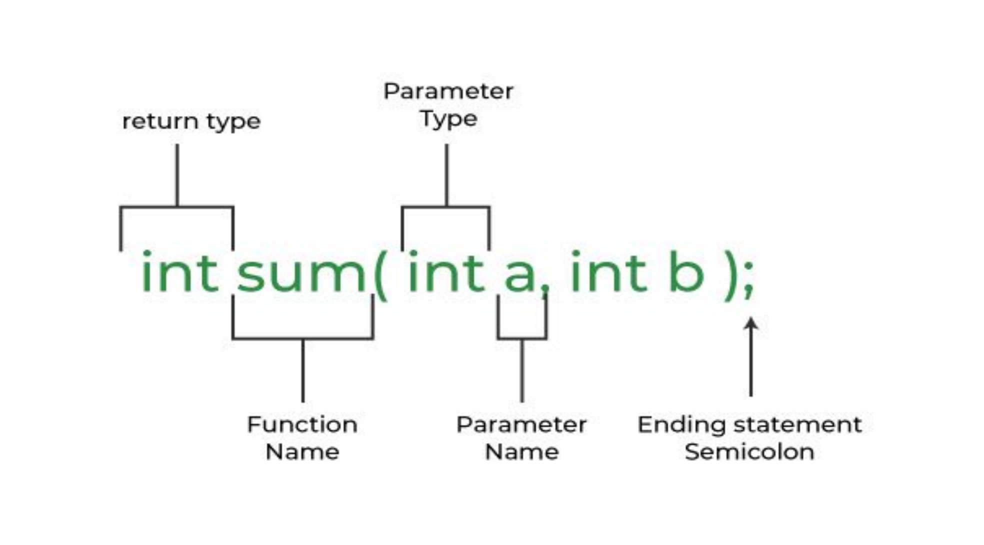
```
int getAreaOfSquare(int side){
  int area;
  area = side * side;
  return area;
}
```



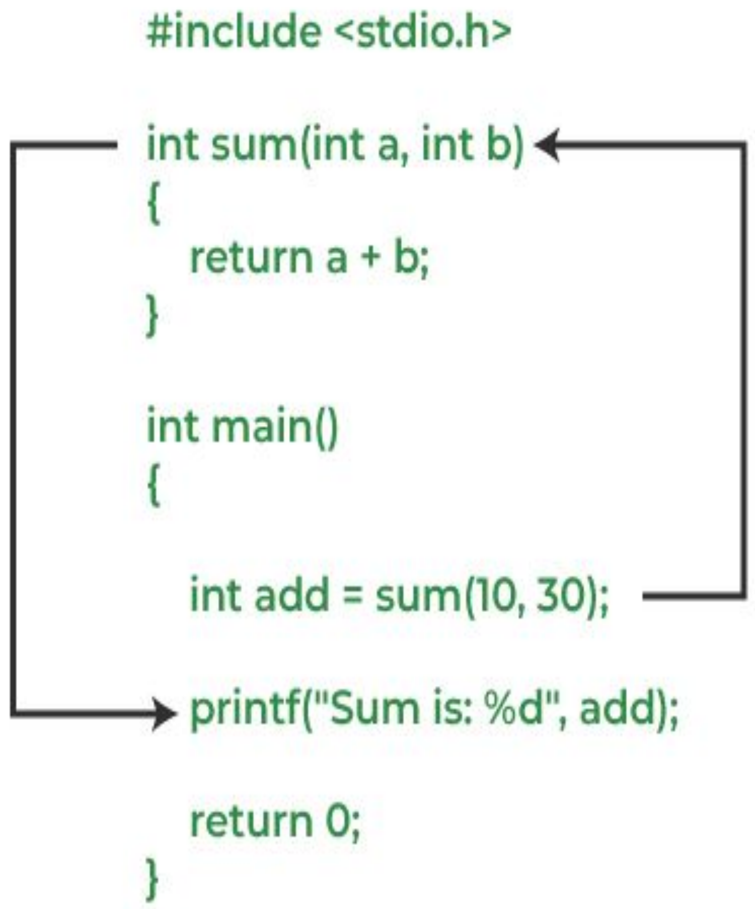
C functions aspects	syntax
function definition	Return_type function_name (arguments list) { Body of function; }
function call	function_name (arguments list);
function declaration	return_type function_name (argument list);

Function Definition & Function Call

Function Declaration



```
int sum( int a, int b );
```



```
#include <stdio.h>

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

int main()
{

   int add = sum(10, 30);

   printf("Sum is: %d", add);

   return 0;
}
```

Categories of Functions

There can be 4 different types of user-defined functions, they are:

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
    add();
    return 0;
}
```

```
void add()
{
    int a=2,b=3,sum;
    sum=a+b;
    printf(“sum=%d”,sum);
    return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>

void add();

int main()

{

      add();

      return 0;

}
```

```
void add()
{
      int a=2,b=3,sum;
      sum=a+b;
      printf(“sum=%d”,sum);
      return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add(); function declaration
int main()
{
    add();
    return 0;
}
```

```
void add()
{
    int a=2,b=3,sum;
    sum=a+b;
    printf(“sum=%d”,sum);
    return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>

void add();

int main()

{

    add();

    return 0;

}
```

```
void add()
{
    int a=2,b=3,sum;
    sum=a+b;
    printf(“sum=%d”,sum);
    return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
      add(); function call
      return 0;
}
```

```
void add()
{
      int a=2,b=3,sum;
      sum=a+b;
      printf("sum=%d",sum);
      return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
      add();
      return 0;
}
```

```
void add()
{
      int a=2,b=3,sum;
      sum=a+b;
      printf(“sum=%d”,sum);
      return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>

void add();

int main()

{

      add();

      return 0;

}
```

```
void add()
{
      int a=2,b=3,sum;
      sum=a+b;
      printf(“sum=%d”,sum);
      return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
    add();
    return 0;
}
```

```
void add()
{
    int a=2,b=3,sum;
    sum=a+b;
    printf(“sum=%d”,sum);
    return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
Int main()
{
      add();
      return 0;
}
```

```
void add()
{
      int a=2,b=3,sum;
      sum=a+b;
      printf("sum=%d",sum); 5
      return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
	add();
	return 0;
}
```

```
void add()
{
	int a=2,b=3,sum;
	sum=a+b;
	printf(“sum=%d”,sum);
	return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
    add();
    return 0;
}
```

```
void add()
{
    int a=2,b=3,sum;
    sum=a+b;
    printf(“sum=%d”,sum);
    return;
}
```

Function with no parameters and no return type

```
#include<stdio.h>
void add();
int main()
{
    add();
    return 0;
}
```

```
void add()
{
    int a=2,b=3,sum;
    sum=a+b;
    printf(“sum=%d”,sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x,int y);
int main()
{
        int a=2,b=3;
        add(a,b);
        return 0;
}
```

```
void add(int x, int y)
{
        int sum;
        sum=x+y;
        printf(“sum=%d”,sum);
        return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
        int a=2,b=3;
        add(a,b);
        return 0;
}
```

```
void add(int x, int y)
{
        int sum;
        sum=x+y;
        printf("sum=%d",sum);
        return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf(“sum=%d”,sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf(“sum=%d”,sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf(“sum=%d”,sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
      int a=2,b=3;
      add(a,b);
      return 0;
}
```

```
void add(int x, int y)
{
      int sum;
      sum=x+y;
      printf(“sum=%d”,sum);
      return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int,int);
int main()
{
      int a=2,b=3;
      add(a,b);
      return 0;
}
```

```
void add(int x, int y)
{
      int sum;
      sum=x+y;
      printf(“sum=%d”,sum);
      return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf(“sum=%d”,sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf(“sum=%d”,sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x,int y);
int main()
{
      int a=2,b=3;
      add(a,b);
      return 0;
}
```

```
void add(int x, int y)
{
      int sum;
      sum=x+y;
      printf("sum=%d",sum);
      return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x ,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf("sum=%d",sum);
    return;
}
```

Function with parameters and no return type

```
#include<stdio.h>
void add(int x,int y);
int main()
{
    int a=2,b=3;
    add(a,b);
    return 0;
}
```

```
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf(“sum=%d”,sum);
    return;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

int add();

int main()

{

    int sum;

    sum=add();

    printf(“sum=%d”,sum);

    return 0;

}
```

```
int add()
{
    int res,x=2,y=3;
    res=x+y;
    return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

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

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

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

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>
Int add();
Int main()
{
    int sum;
    sum=add();
    printf(“sum=%d”,sum);
    return 0;
}
```

```
int add()
{
    int res,x=2,y=3;
    res=x+y;
    return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

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

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with no parameters and with return type

```
#include<stdio.h>

Int add();

Int main()

{

      int sum;

      sum=add();

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add()
{
      int res,x=2,y=3;
      res=x+y;
      return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

      int sum,a=2,b=3;

      sum=add(a,b);

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add(int x, int y)
{
      int res;
      res=x+y;
      return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

      int sum,a=2,b=3;

      sum=add(a,b);

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add(int x, int y)
{
      int res;
      res=x+y;
      return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>
int add(int,int);
int main()
{
    int sum,a=2,b=3;
    sum=add(a,b);
    printf(“sum=%d”,sum);
    return 0;
}
```

```
int add(int x, int y)
{
    int res;
    res=x+y;
    return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

     int sum,a=2,b=3;

     sum=add(a,b);

     printf(“sum=%d”,sum);

     return 0;

}
```

```
int add(int x, int y)
{
     int res;
     res=x+y;
     return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

      int sum,a=2,b=3;

      sum=add(a,b);

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add(int x, int y)
{
      int res;
      res=x+y;
      return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

    int sum,a=2,b=3;

    sum=add(a,b);

    printf(“sum=%d”,sum);

    return 0;

}
```

```
int add(int x, int y)
{
    int res;
    res=x+y;
    return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

       int sum,a=2,b=3;

       sum=add(a,b);

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

       return 0;

}
```

```
int add(int x, int y)
{
       int res;
       res=x+y;
       return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

    int sum,a=2,b=3;

    sum=add(a,b);

    printf(“sum=%d”,sum);

    return 0;

}
```

```
int add(int x, int y)
{
    int res;
    res=x+y;
    return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>
int add(int,int);
int main()
{
      int sum,a=2,b=3;
      sum=add(a,b);
      printf(“sum=%d”,sum);
      return 0;
}
```

```
int add(int x, int y)
{
      int res;
      res=x+y;
      return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>

int add(int,int);

int main()

{

      int sum,a=2,b=3;

      sum=add(a,b);

      printf(“sum=%d”,sum);

      return 0;

}
```

```
int add(int x, int y)
{
      int res;
      res=x+y;
      return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>
int add(int,int);
int main()
{
    int sum,a=2,b=3;
    sum=add(a,b);
    printf(“sum=%d”,sum);
    return 0;
}
```

```
int add(int x, int y)
{
    int res;
    res=x+y;
    return res;
}
```

Function with parameters and with return type

```
#include<stdio.h>
int add(int,int);
int main()
{
    int sum,a=2,b=3;
    sum=add(a,b);
    printf(“sum=%d”,sum);
    return 0;
}
```

```
int add(int x, int y)
{
    int res;
    res=x+y;
    return res;
}
```

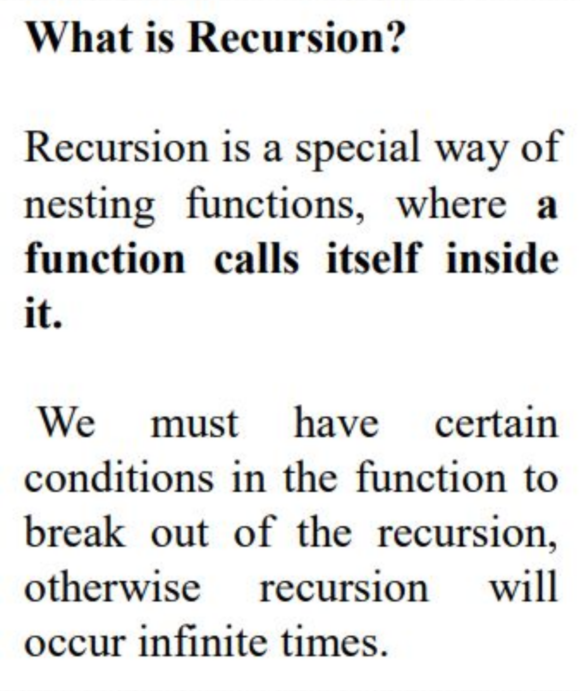
Simple Examples

- Write a program in C to find the square of any number using the function.

```
#include <stdio.h>
double square(double num)
{
    return (num * num);
}
```

```
int main()
{
     int num;
    double n;
    printf("Input any number for square : ");
    scanf("%d", &num);
    n = square(num);
    printf("The square of %d is : %.2f\n",
num, n);
     return 0;
}
```

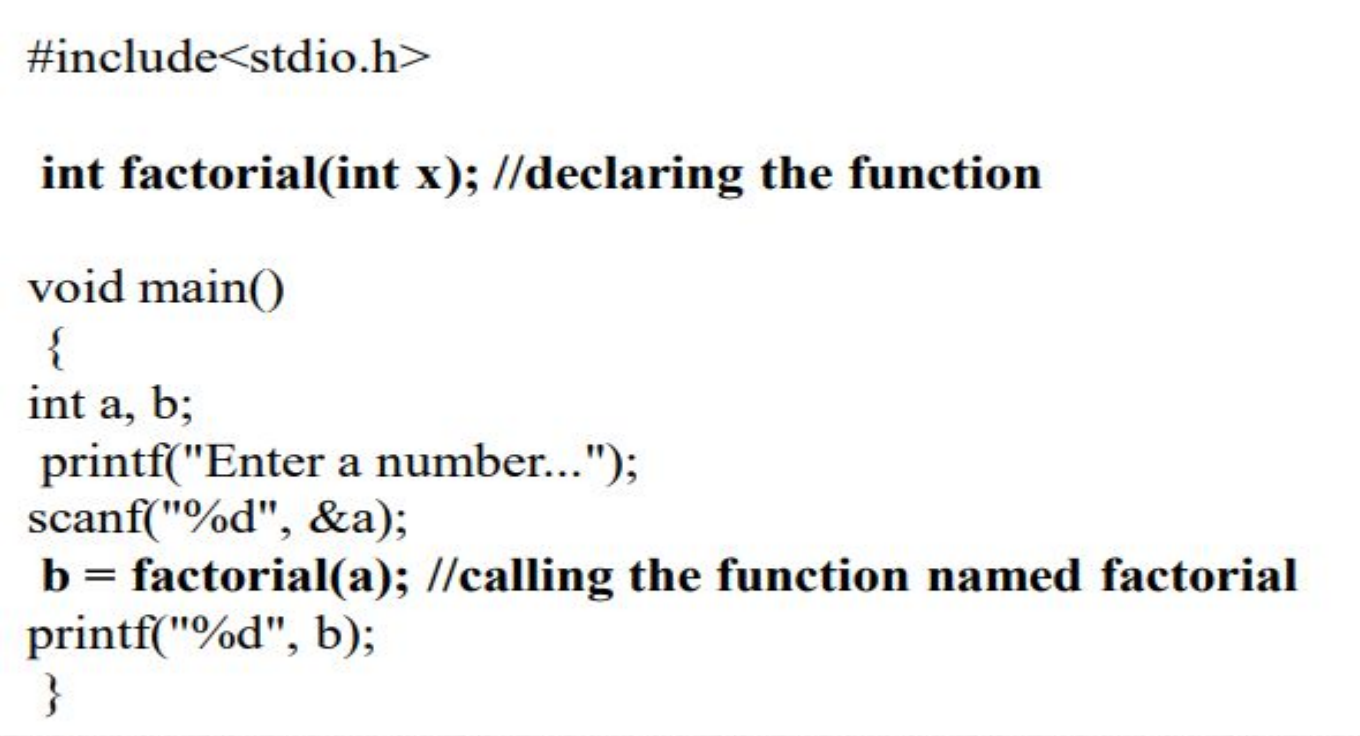
Recursion



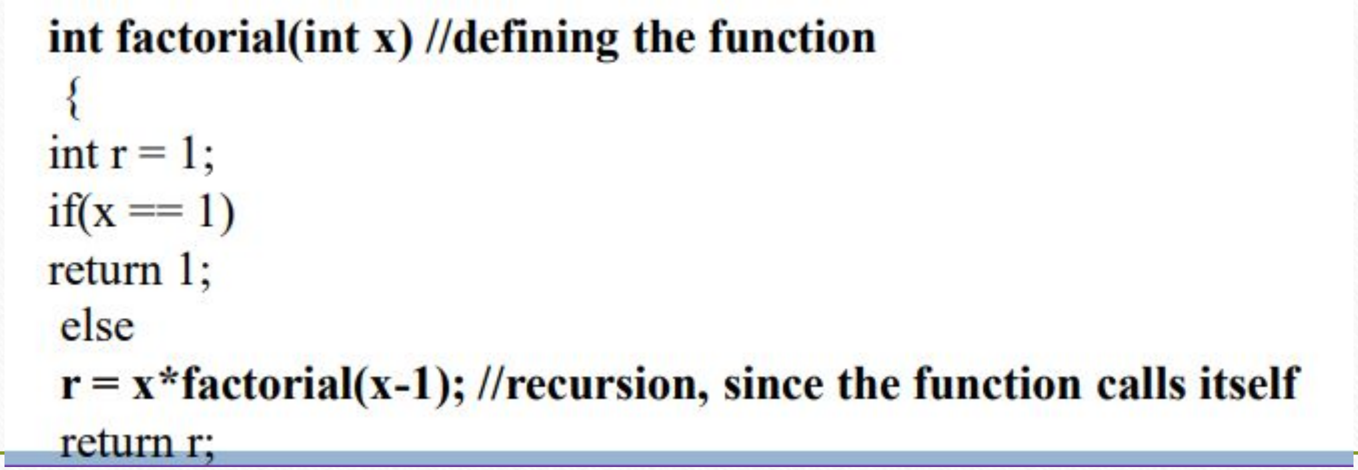
What is Recursion?

Recursion is a special way of nesting functions, where **a function calls itself inside it.**

We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.



```
#include<stdio.h>

 int factorial(int x); //declaring the function

void main()
 {
int a, b;
 printf("Enter a number...");
scanf("%d", &a);
 b = factorial(a); //calling the function named factorial
printf("%d", b);
 }
```



```
 int factorial(int x) //defining the function
  {
 int r = 1;
 if(x == 1)
 return 1;
  else
  r = x*factorial(x-1); //recursion, since the function calls itself
  return r;
```

```
#include<stdio.h>
int factorial(int x);
    //declaring the function
void main()
{
    int a, b;
    printf("Enter a number...");
    scanf("%d", &a);
    b = factorial(a);
    //calling the function named factorial
    printf("%d", b);
}
```

```
int factorial(int x)
//defining the function
{
    if(x == 1)
        return 1;
    else
        return (x*factorial(x-1));
}
```

The scope, visibility and lifetime of variables:

- Automatic variables
- Static Variables
- Register Variables and
- External Variables.

Scope

- Scope determines the region of program in which a defined object is visible.
  - Global: Any object defined in the global area of a program is visible from its definition until the end of the program.
  - Local: Variables defined within a block have local scope . They are visible from the point of declaration until the end of the block.

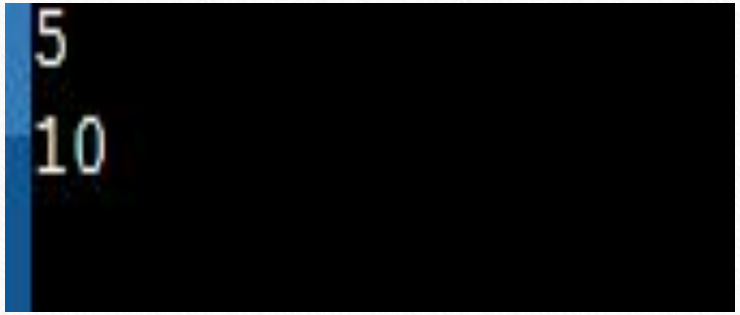
```
#include<stdio.h>
int a=10;
void fun(int  a)
{
      printf(“%d”,a);
}
void fun()
{
      printf(“%d”,a);
}
```

```
void main( )
{
      fun(5);
      fun();
}
```

```
#include<stdio.h>
int a=10;
void fun(int  a)
{
      printf(“%d”,a);
}
void fun()
{
      printf(“%d”,a);
}
```

```
void main( )
{
      fun(5);
      fun();
}
```

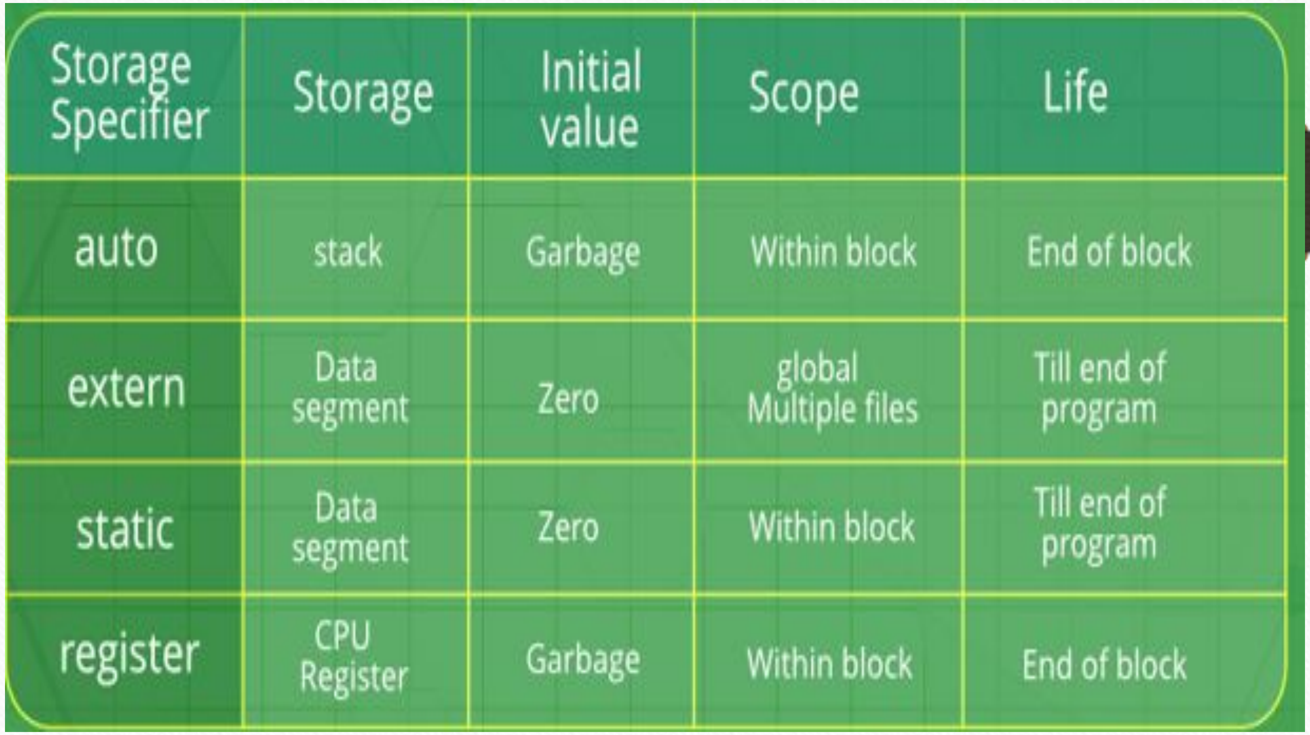
OUTPUT:



```
5
10
```

Storage classes

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.



Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

auto

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language.

Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

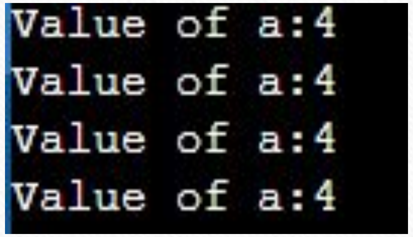
```
#include <stdio.h>
int main( )
{
    auto int j = 1;
    {
        auto int j= 2;
        {
            auto int j = 3;
            printf ( " %d ", j);
        }
        printf ( "\t %d ",j);
    }
    printf( "%d\n", j);
}
```

Static Variables

- Static variables: Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.
- Syntax:

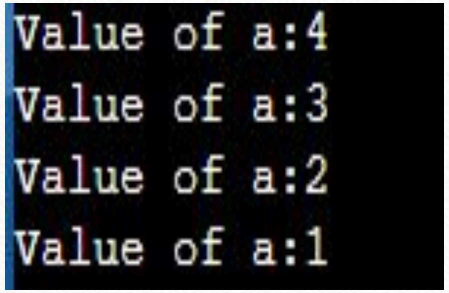
```
static data_type var_name = var_value;
```

```
#include <stdio.h>
void decrement()
{
   int a=5;
   a--;
   printf("Value of  a:%d\n", a);
}
int main()
{
   int i , n=4;
   for(i=0; i<n; i++)
      decrement();
   return 0;
}
```



```
Value of a:4
Value of a:4
Value of a:4
Value of a:4
```

```
#include <stdio.h>
void decrement()
{
     static int a=5;
    a--;
    printf("Value of  a:%d\n", a);
}
int main()
{
    int i , n=4;
    for(i=0; i<n; i++)
        decrement();
    return 0;
}
```

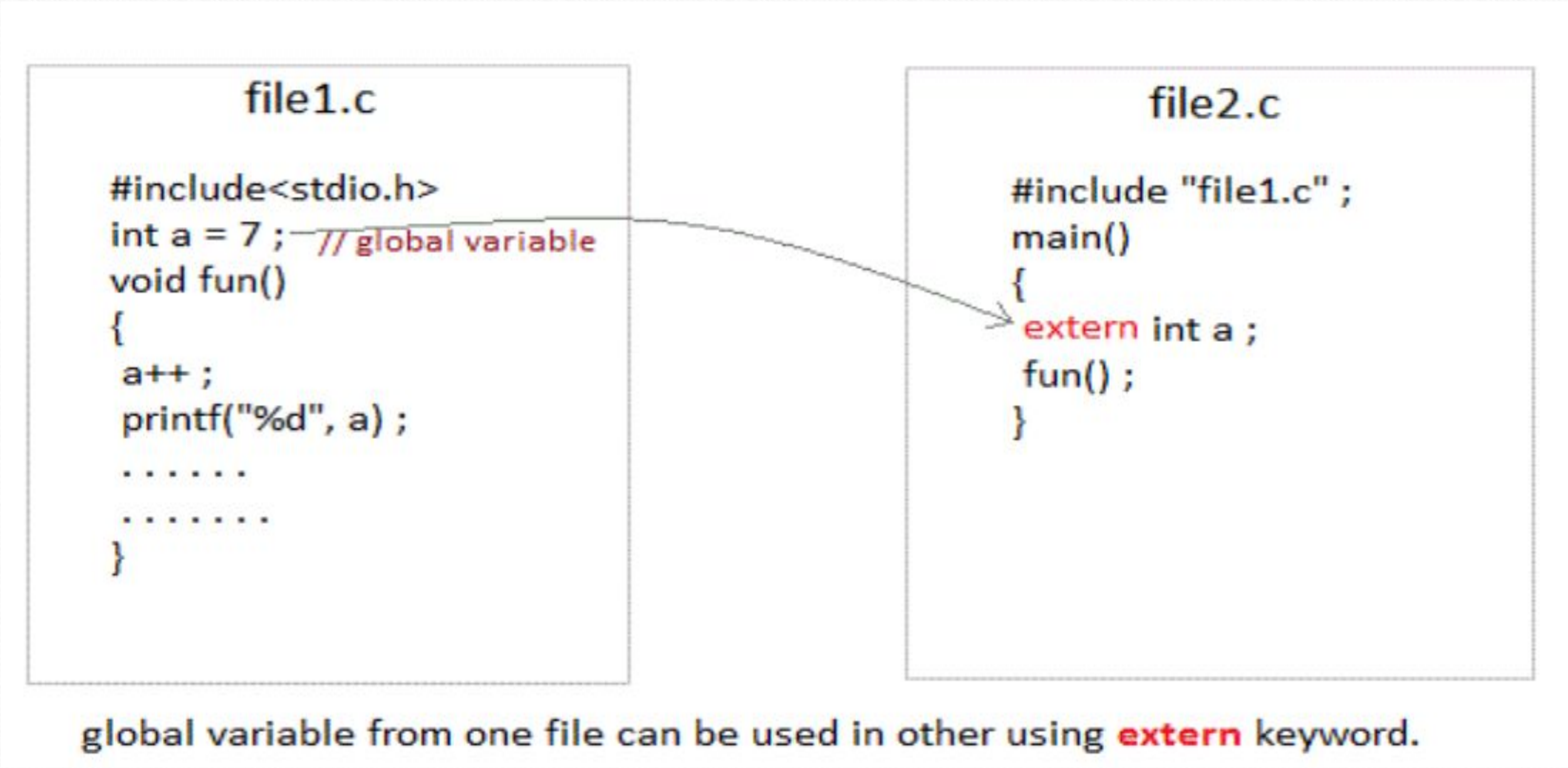


```
Value of a:4
Value of a:3
Value of a:2
Value of a:1
```

extern

Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword extern is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.



file1.c

```
#include<stdio.h>
int a = 7 ; // global variable
void fun()
{
 a++ ;
 printf("%d", a) ;
 . . . . . .
 . . . . . . .
}
```

file2.c

```
#include "file1.c" ;
main()
{
 extern int a ;
 fun() ;
}
```

global variable from one file can be used in other using **extern** keyword.

Register variables

The scope of the register variables is local to the block in which the variables are defined. The variables which are used more number of times in a program and are declared as register variables for faster access.

Example: loop counter variables.

```
register int y=6;
```