

DATA TYPES AND LOOPS

Data Type

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. It specifies the type of data that the variable can store like integer, character, floating, double, etc. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

Classification of Data Types

Types	Description
Primitive Data Types	Arithmetic types can be further classified into integer and floating data types.
Void Types	The data type has no value or operator and it does not provide a result to its caller. But void comes under Primitive data types.
User Defined DataTypes	It is mainly used to assign names to integral constants, which make a program easy to read and maintain
Derived types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.

Size and Range of Data Types

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu

long long int	8	$-(2^{63})$ to $(2^{63})-1$	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Primary Data Types

- Integer
- Character
- Floating Point
- Double Floating Point
- Void

Integer Types

The integer data type in C is used to store the whole numbers without decimal values. Octal values, hexadecimal values, and decimal values can be stored in int data type in C. We can determine the size of the int data type by using the [sizeof operator](#) in C. Unsigned int data type in C is used to store the data values from zero to positive numbers but it can't store negative values like signed int. Unsigned int is larger in size than signed int and it uses “%u” as a format specifier in C programming language. Below is the programming implementation of the int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 2 bytes or 4 bytes
- **Format Specifier:** %d

Character Types

Character data type allows its variable to store only a single character. The storage size of the character is 1. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte
- **Format Specifier:** %c

Floating Point Types

In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- **Range:** $1.2\text{E}-38$ to $3.4\text{E}+38$
- **Size:** 4 bytes
- **Format Specifier:** %f

Double Types

A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C. Double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points. Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory as occupied by the floating-point type. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes
- **Format Specifier:** %lf

Void data Types

The void data type in C is used to specify that no value is present. It does not provide a result value to its caller. It has no values and no operations. It is used to represent nothing. Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

User-Defined DataTypes

The data types that are defined by the user are called the derived datatype or user-defined derived data type.

These types include:

- Class
- Structure
- Union
- Enumeration
- Typedef defined DataType

Class

- The building block of C++ that leads to Object-Oriented programming is a **Class**. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Structure

- A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.
- We can use this data type to store data of different attributes of different data types. For example, If we want to store data on multiple patients such as patient name, age, and blood group.
- 'struct' keyword is used to create a structure.

Union

- Like Structures union is a user defined data type. In union, all members share the same memory location.
- Size of a union is taken according the size of largest member in union.
- Unions can be useful in many situations where we want to use the same memory for two or more members. For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data

Enumeration

- Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.
- Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.
- If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.
- We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

- The value assigned to enum names must be some integral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.
- All enum constants must be unique in their scope. For example, the following program fails in compilation.

Typedef

- C++ allows you to define explicitly new data type names by using the keyword typedef. Using typedef does not actually create a new data class, rather it defines a name for an existing type. This can increase the portability(the ability of a program to be used across different types of machines; i.e., mini, mainframe, micro, etc; without much changes into the code)of a program as only the typedef statements would have to be changed. Using typedef one can also aid in self-documenting code by allowing descriptive names for the standard data types.

Gate Questions

Output of the following Code

```
int main()
{
    void *vptr, v;
    v = 0;
    vptr = &v;
    printf("%v", *vptr);
    getchar();
    return 0;
}
```

- (A) 0
- (B) Compiler Error
- (C) Garbage Value

Output of the following Code

```
int main()
{
    void *vptr, v;
    v = 0;
    vptr = &v;
    printf("%v", *vptr);
    getchar();
    return 0;
}
```

- (A) 0
- (B) Compiler Error
- (C) Garbage Value

Answer: (B)

Explanation: void is not a valid type for declaring variables. void * is valid though.

Output of the following Code

```
#include <stdio.h>
int main()
{
    if (sizeof(int) > -1)
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

- (A) Yes
- (B) No
- (C) Compiler Error
- (D) Runtime Error

Output of the following Code

```
#include <stdio.h>
int main()
{
    if (sizeof(int) > -1)
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

- (A) Yes
- (B) No
- (C) Compiler Error
- (D) Runtime Error

Answer: (B)

Explanation: In C, when an integer value is compared with an unsigned int, the int is promoted to unsigned. Negative numbers are stored in 2's complement form and unsigned value of the 2's complement form is much higher than the sizeof int.

LOOPS

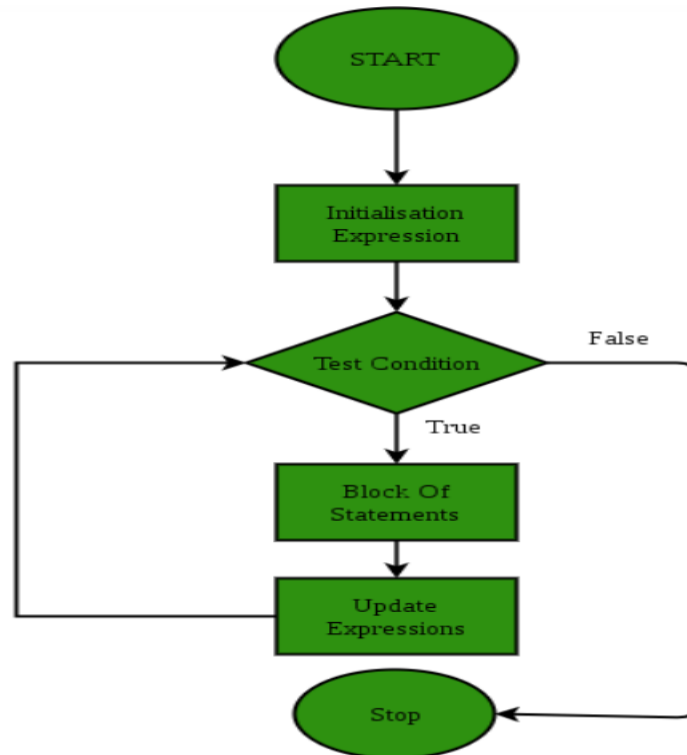
Loops in programming are used to repeat a block of code until the specified condition is met. A loop statement allows programmers to execute a statement or group of statements multiple times without repetition of code.

There are mainly two types of loops in C Programming:

- 1.Entry Controlled loops: In Entry controlled loops the test condition is checked before entering the main body of the loop. For Loop and While Loop is Entry-controlled loops.
- 2.Exit Controlled loops: In Exit controlled loops the test condition is evaluated at the end of the loop body. The loop body will execute at least once, irrespective of whether the condition is true or false. do-while Loop is Exit Controlled loop.

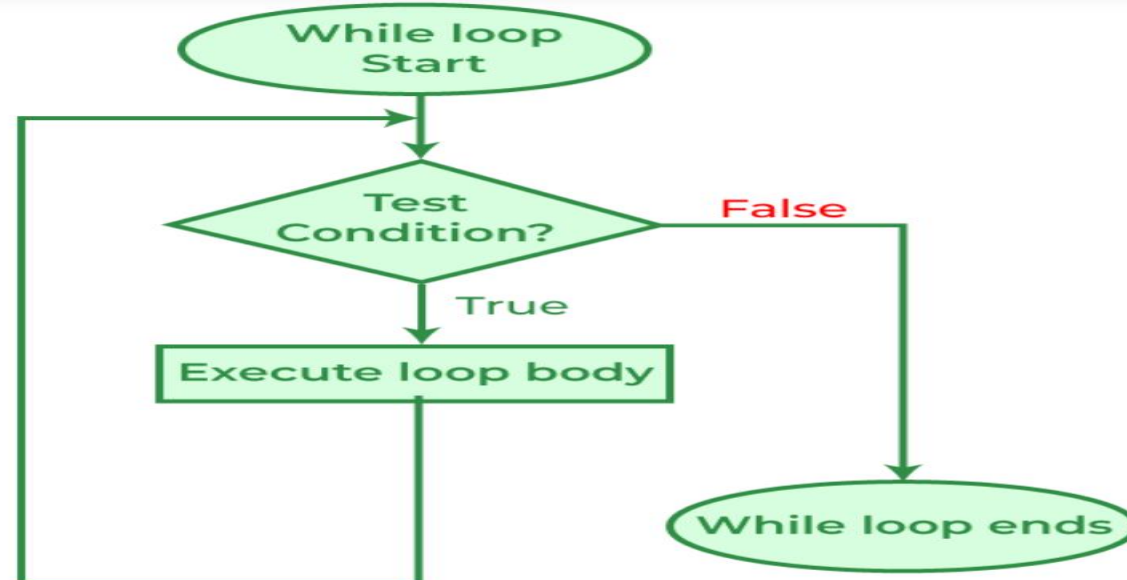
For Loop

- for loop in C programming is a repetition control structure that allows programmers to write a loop that will be executed a specific number of times. for loop enables programmers to perform n number of steps together in a single



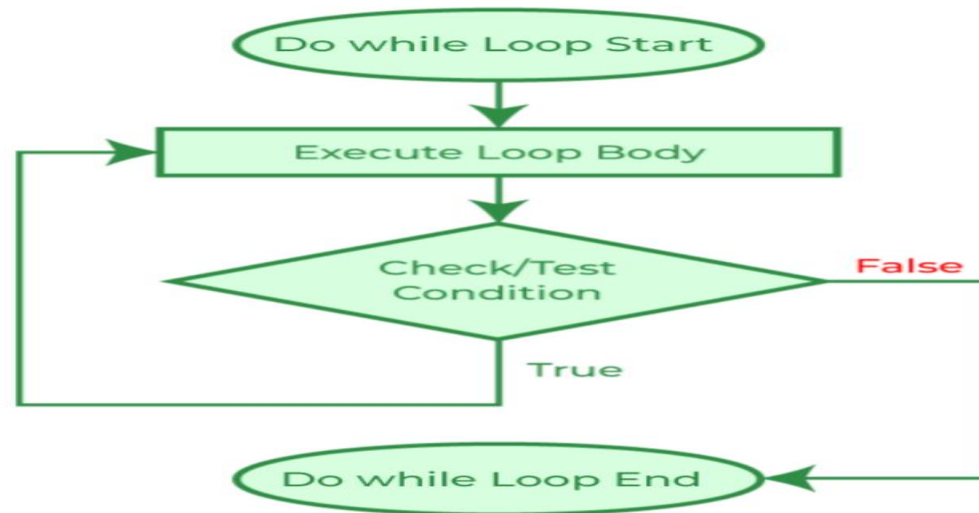
While Loop

- While loop does not depend upon the number of iterations. In for loop the number of iterations was previously known to us but in the While loop, the execution is terminated on the basis of the test condition. If the test condition will become false then it will break from the while loop else body will be executed.



Do-while Loop

- The do-while loop is similar to a while loop but the only difference lies in the do-while loop test condition which is tested at the end of the body. In the do-while loop, the loop body will **execute at least once** irrespective of the test condition.



Loop control Statements

Name	Description
break statement	the break statement is used to terminate the switch and loop statement. It transfers the execution to the statement immediately following the loop or switch.
continue statement	continue statement skips the remainder body and immediately resets its condition before reiterating it.
goto statement	goto statement transfers the control to the labeled statement.

Infinite Loop

An infinite loop is executed when the test expression never becomes false and the body of the loop is executed repeatedly. A program is stuck in an Infinite loop when the condition is always true. Mostly this is an error that can be resolved by using Loop Control statements.

Gate Questions

Output of the following Code

```
#include <stdio.h>
int i;
int main()
{
    if (i);
    else
        printf("Else");
    return 0;
}
```

- (A) if block is executed.
- (B) else block is executed.
- (C) It is unpredictable as i is not initialized.
- (D) Error: misplaced else

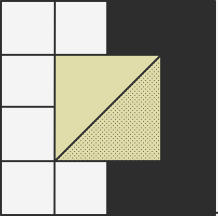
Output of the following Code

```
#include <stdio.h>
int i;
int main()
{
    if (i);
    else
        printf("Else");
    return 0;
}
```

- (A) IF BLOCK IS executed.
- (B) else block is executed.
- (C) It is unpredictable as i is not initialized.
- (D) Error: misplaced else

Answer: (B)

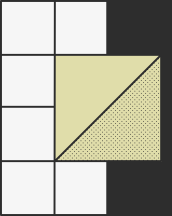
Explanation: Since i is defined globally, it is initialized with default value 0. The Else block is executed as the expression within if evaluates to FALSE. Please note that the empty block is equivalent to a semi-colon(;). So the statements **if (i);** and **if (i) {}** are equivalent.



Conditional Statements

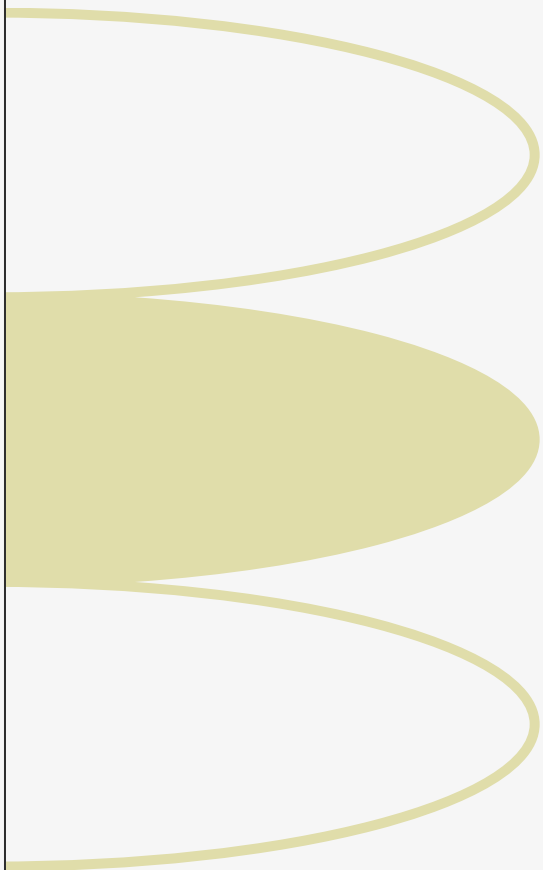
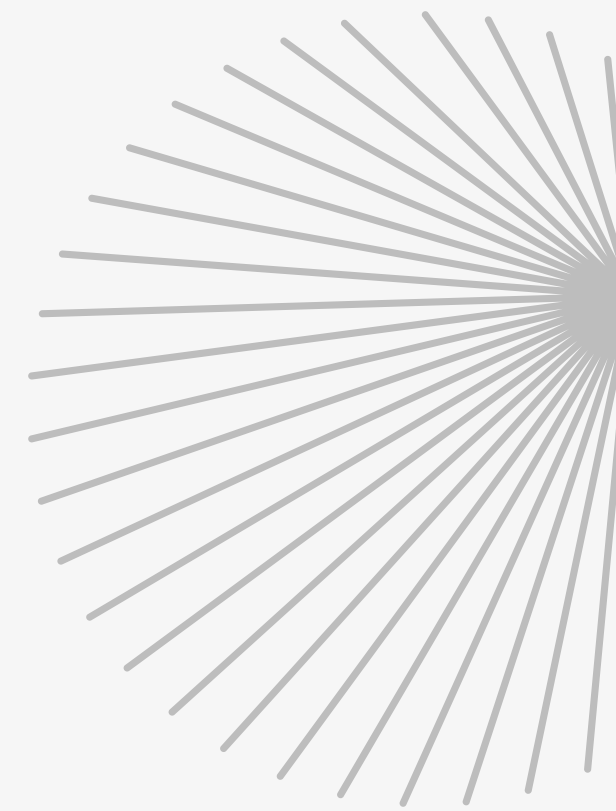
February 23, 2023
Tharun A





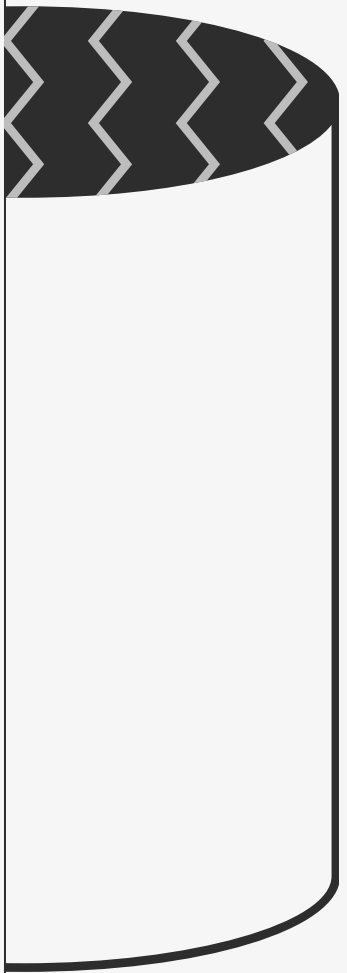
WHAT? and WHY?

Conditional statements help you to make a decision based on certain conditions. These conditions are specified by a set of conditional statements having boolean expressions which are evaluated to a boolean value of true or false.





Relational Operators

- 
- Less than: $a < b$
 - Less than or equal to: $a \leq b$
 - Greater than: $a > b$
 - Greater than or equal to: $a \geq b$
 - Equal to: $a == b$
 - Not Equal to: $a != b$

IF Statement

```
// if (condition)
//   instruction;

#include<stdio.h>
int main()
{
    int num1=1;
    int num2=2;
    if(num1<num2)           //test-condition
    {
        printf("num1 is smaller than
num2});
    return 0;
}
```

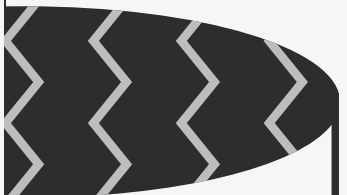
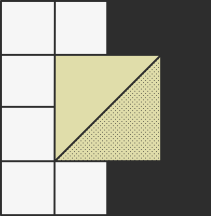
IF-ELSE Statement

```

// if (test-expression)
// {
//     True block of statements
// }
// Else
// {
//     False block of statements
// }
// Statements;

#include<stdio.h>
int main()
{
    int num=19;
    if(num<10)
    {
        printf("The value is less than 10");
    }
    else
    {
        printf("The value is greater than
10");
    }
    return 0;
}
```

Conditional expressions



```
#include <stdio.h>
int main() {
    int y;
    int x = 2;
    y = (x >= 6) ? 6 : x;
    printf("y =%d ", y);
    return 0;}
Output :0;
}
```

Example



```
y = (x >= 6) ? 6 : x;
```

```
if (x >= 6) y = 6;  
else y=x;
```

```
if (x >= 6) y = 6;  
if (x < 6) y = x;
```

Switch case

```
#include <stdio.h>
int main() {
    int num = 8;
    switch (num) {
        case 7:
            printf("Value is 7");
            break;
        case 8:
            printf("Value is 8");
            break;
        case 9:
            printf("Value is 9");
            break;
        default:
            printf("Out of range");
            break;
    }
    return 0;
}
```

Nested IF-ELSE Statement

```
#include<stdio.h>
int main()
{
    int num=1;
    if(num<10)
    {
        if(num==1)
        {
            printf("The value is:%d\n",num);
        }
        else
        {
            printf("The value is greater than 1");
        }
    }
    else
    {
        printf("The value is greater than 10");
    }
    return 0;
}
```


Nested ELSE-IF Statement

```
#include<stdio.h>
int main()
{
    int marks=83;
    if(marks>75){
        printf("First class");
    }
    else if(marks>65){
        printf("Second class");
    }
    else if(marks>55){
        printf("Third class");
    }
    else{
        printf("Fourth class");
    }
    return 0;
}

return 0;
```

Question



```
// Assuming a and b are positive numbers
while(a!=b){
    if(a > b) a -= b;
    else b -= a;
}
cout<<a;
```

Question

```
#include "stdio.h"
int main()
{
    int i;
    if(i=0,2,3)
        printf("World ");
    else
        printf("Hello ");
    printf("%d\n",i);
}
```

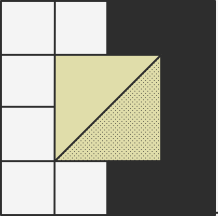
- a) Hello 3
- b) Hello 0
- c) World 0
- d) World 3

Question



// What's the "condition" so that the following code snippet prints both HelloWorld !

```
if "condition"  
    printf ("Hello");  
else  
    printf ("World");
```

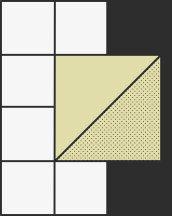


Storage Classifiers

February 23, 2023

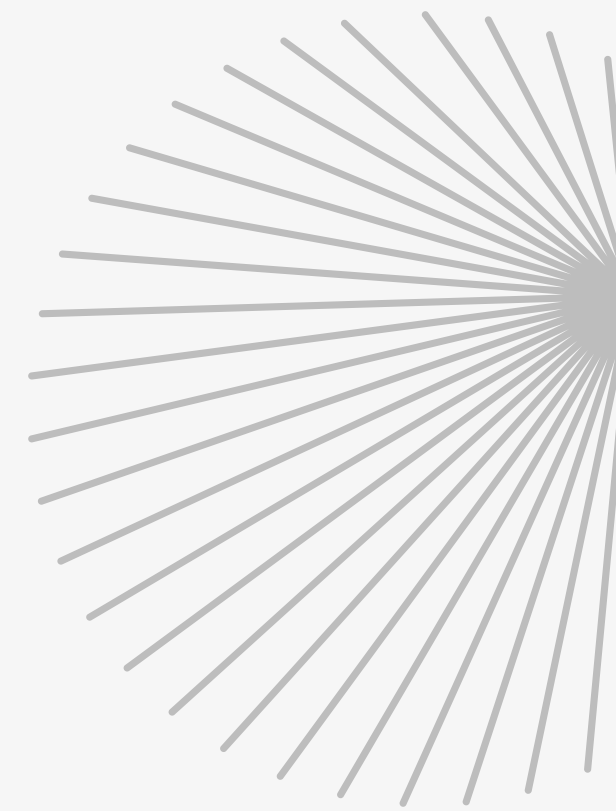
Tharun A





WHAT? and WHY?

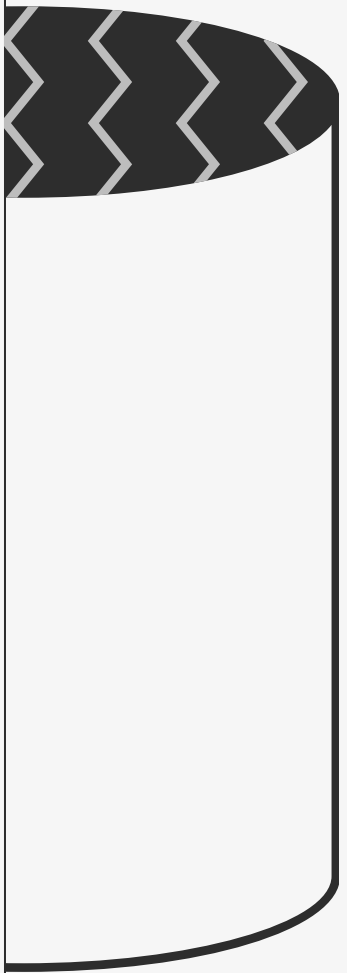
Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.





auto

- **Storage:** RAM
- **Default value:** Garbage value
- **Scope:** Local
- **Lifetime:** Within function

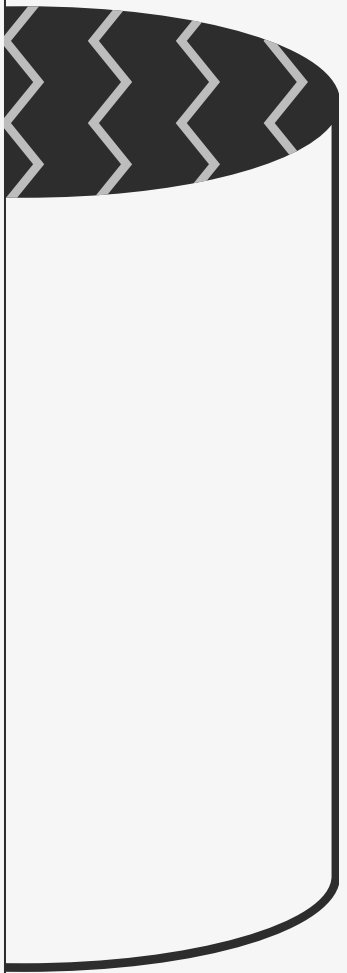


```
#include <stdio.h>
int main()
{
    int a; //auto
    char b;
    float c;
    printf("%d %c %f",a,b,c); // printing
    initial default value of automatic
    variables a, b, and c.
    return 0;
}
```



static

- **Storage:** RAM
- **Default value:** 0
- **Scope:** Local
- **Lifetime:** Till the end of the main program



```
#include<stdio.h>
void sum()
{
    static int a = 10;
    static int b = 24;
    printf("%d %d \n",a,b);
    a++;
    b++;
}
void main(){
    int i;
    for(i = 0; i< 3; i++) {
        sum(); // The static variables holds their
               // value between multiple function calls.
    }
}
```


extern

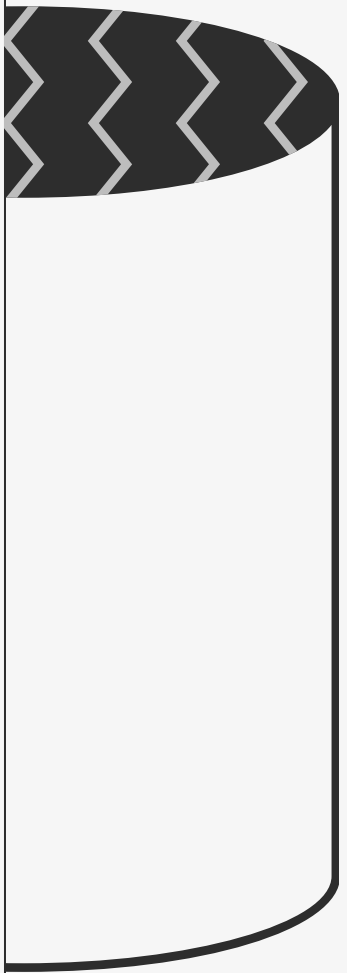
- **Storage:** RAM
- **Default value:** 0
- **Scope:** Global
- **Lifetime:** Till the end of the main program

```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined
globally, the memory will not be allocated
to a
printf("%d",a);
}
```



register

- **Storage:** Register
- **Default value:** Garbage value
- **Scope:** Local
- **Lifetime:** Within function



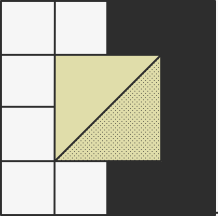
```
#include <stdio.h>
int main()
{
    register int a; // variable a is allocated
                    // memory in the CPU register. The initial
                    // default value of a is 0.
    printf("%d",a);
}
```

Question

```
● ● ●  
  
#include <stdio.h>  
int a, b, c = 0;  
void prtFun (void);  
int main ()  
{  
    static int a = 1; /* line 1 */  
    prtFun();  
    a += 1;  
    prtFun();  
    printf ( "\n %d %d " , a, b) ;  
}  
  
void prtFun (void)  
{  
    static int a = 2; /* line 2 */  
    int b = 1;  
    a += ++b;  
    printf ( " \n %d %d " , a, b);  
}
```

Question

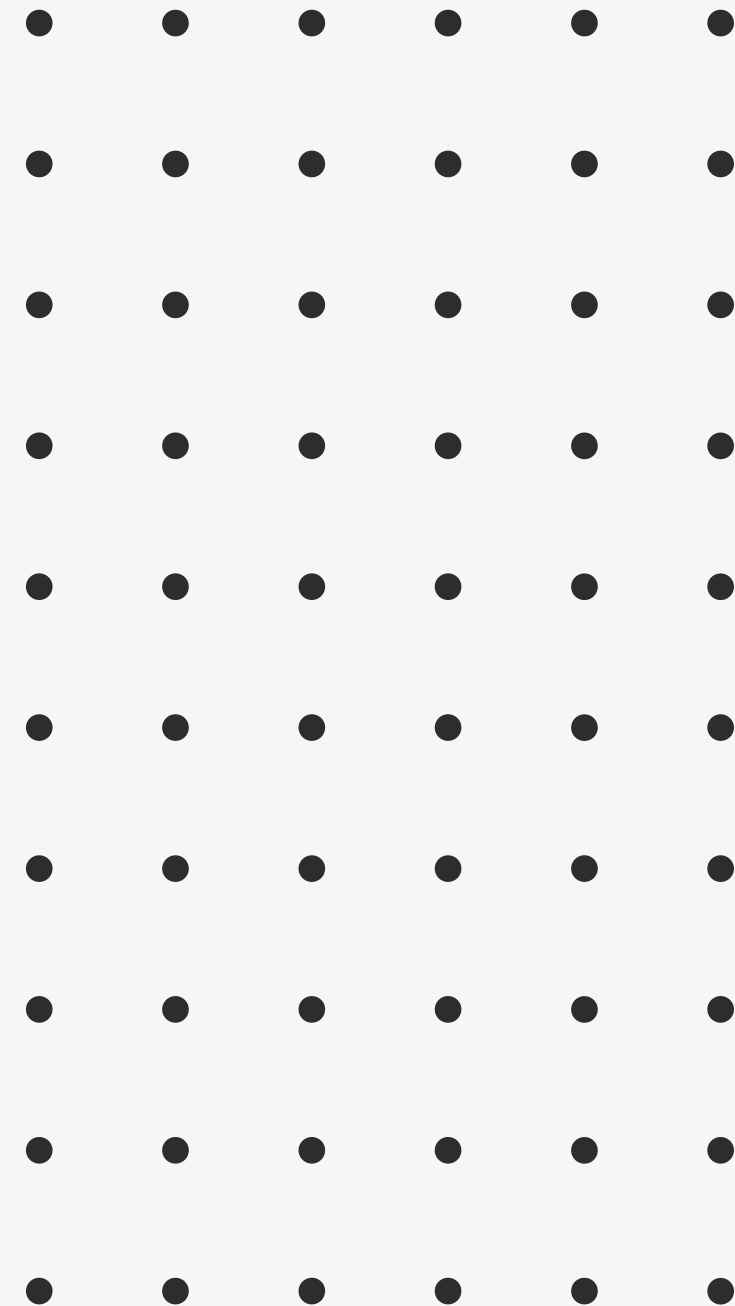
```
● ● ●  
  
#include <stdio.h>  
int a, b, c = 0;  
void prtFun (void);  
int main ()  
{  
    auto int a = 1; /* line 1 */  
    prtFun();  
    a += 1;  
    prtFun();  
    printf ( "\n %d %d " , a, b) ;  
}  
  
void prtFun (void)  
{  
    register int a = 2; /* line 2 */  
    int b = 1;  
    a += ++b;  
    printf ( " \n %d %d " , a, b);  
}
```



Thank you for listening

February 23, 2023

Tharun A



Functions

A **function** is a block of code that performs a particular task.

Benefits of Using Functions

- It provides modularity to your program's structure.
- 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.

FUNCTIONS

```
graph TD; FUNCTIONS --> BuiltIn[Built-In Functions]; FUNCTIONS --> UserDefined[User-Defined Functions]; BuiltIn --- List["scanf()  
printf()  
getc()  
putc()"]; UserDefined --- Def["A series of Instructions that  
are to be executed more than once"];
```

Built-In Functions

`scanf()`
`printf()`
`getc()`
`putc()`

User-Defined Functions

A series of Instructions that
are to be executed more
than once

Elements of User-Defined Function

- Function Declaration
- Function calling
- Function Definition

USER DEFINED FUNCTION :

SYNTAX :

```
retu_datatype func_name(arguments)
{
    Body of the function
    statements;
    return;
}
```

```
#include <stdio.h>
main()
{
    :
    :
    address()
    :
    :
}

address()
{
    :
    :
    :
}
```

call the function from main() :

syntax :

```
func_name(arguments );
```

How Function Works

- Once a function is called the control passes to the called function.
- The working of calling function is temporarily stopped.
- When the execution of called function is completed then the control return back to the calling function and execute the next statement.

main()

{

.....

.....

func1();

.....

.....

}

data_type func1()

{

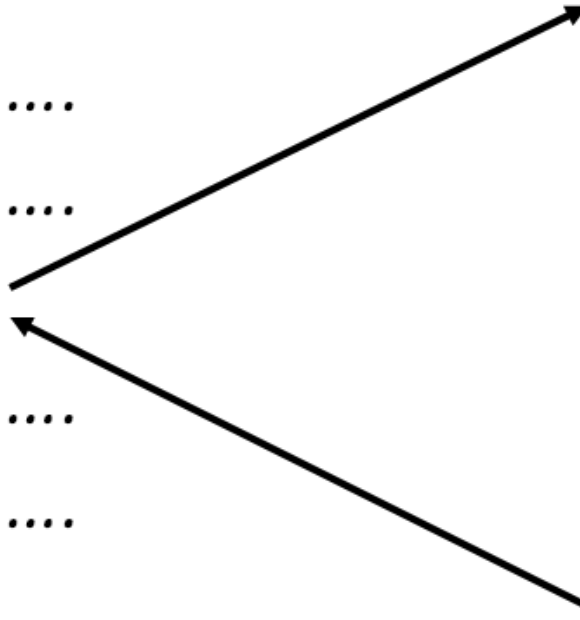
.....

.....

.....

.....

}



Calling function

Called function

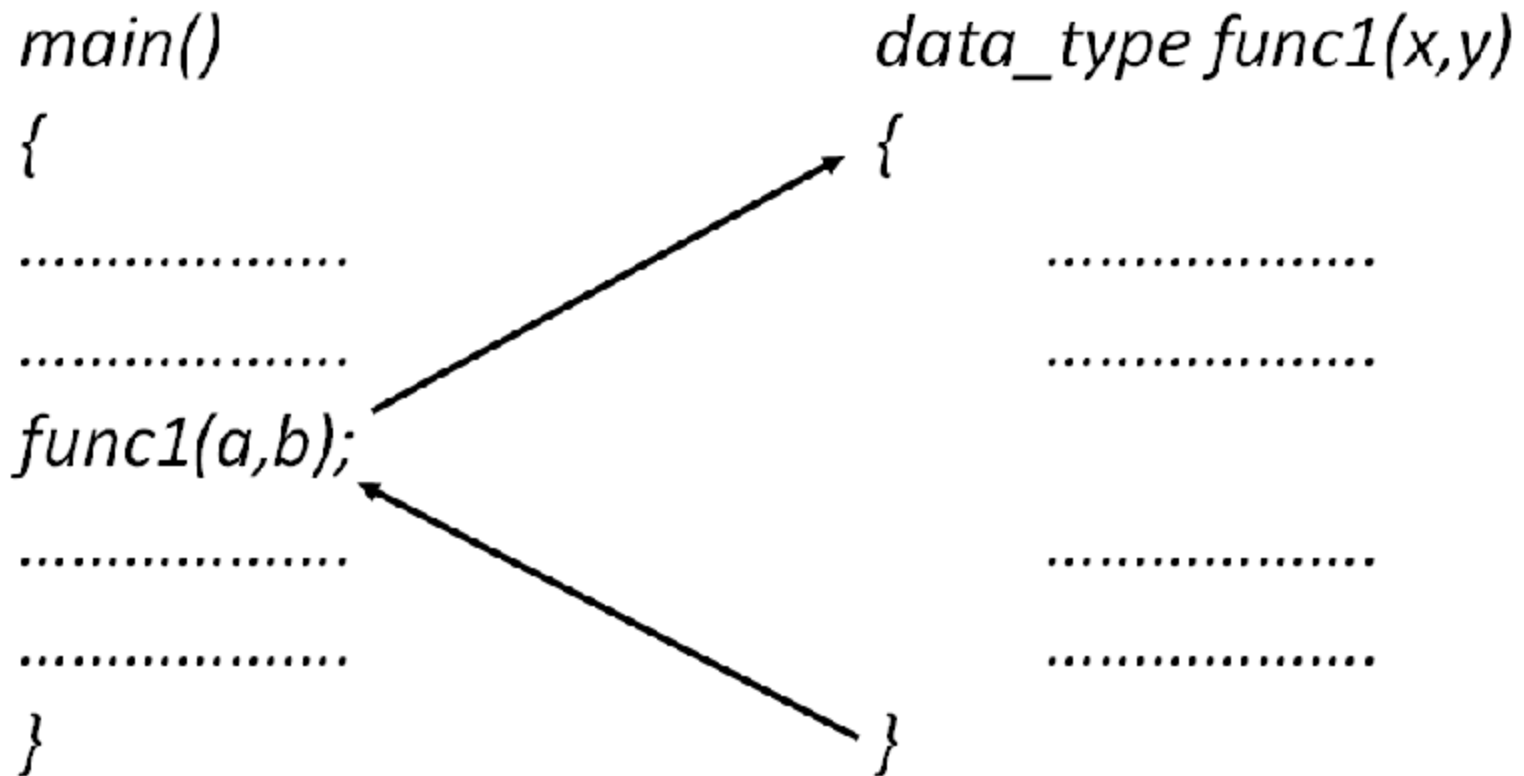
Parameters

- Actual Parameter

These are the parameters transferred from the calling function to the called function.

- Formal Parameter

These are the parameters which is used in the called function.



Calling function

Called function

a,b are the Actual parameters

x,y are the Formal parameters

return Statement

- The ***return*** statement may or may not send some values to the calling function.
- ***Syntax:***
 - return; (or)
 - return(expression);

Function Prototypes

1. Function with no arguments and no return values.
2. Function with arguments and no return values.
3. Function with arguments and return values.
4. Function with no arguments and with return values.

1. Function with no arguments and no return values

- Here no data transfer take place between the calling function and the called function.
- These functions act independently, i.e. they get input and display output in the same block.

Function with no arguments and no return values

The diagram illustrates the relationship between a function call and its definition. On the left, the `main()` function body contains a call to `func1();`. On the right, the `data_type func1()` function definition is shown. A dashed arrow points from the `func1();` call in `main()` to the opening curly brace of the `func1()` definition. Another dashed arrow points from the closing curly brace of the `func1()` definition back to the `func1();` call, indicating the return path.

```
main()  
{  
.....  
.....  
func1();  
.....  
}  
  
data_type func1()  
{  
.....  
.....  
.....  
.....  
}
```

Example

```
#include <stdio.h>
#include<conio.h>
void add();
void main()    //calling function
{
    clrscr()
    add();
    getch();
}
void add()    //called function
{
    int a,b,c;
    printf("\nEnter two number:");
    scanf("%d%d",&a,&b);
    c=a+b;
    printf("\nSum is:%d",c);
}
```

Output

Enter two number:3

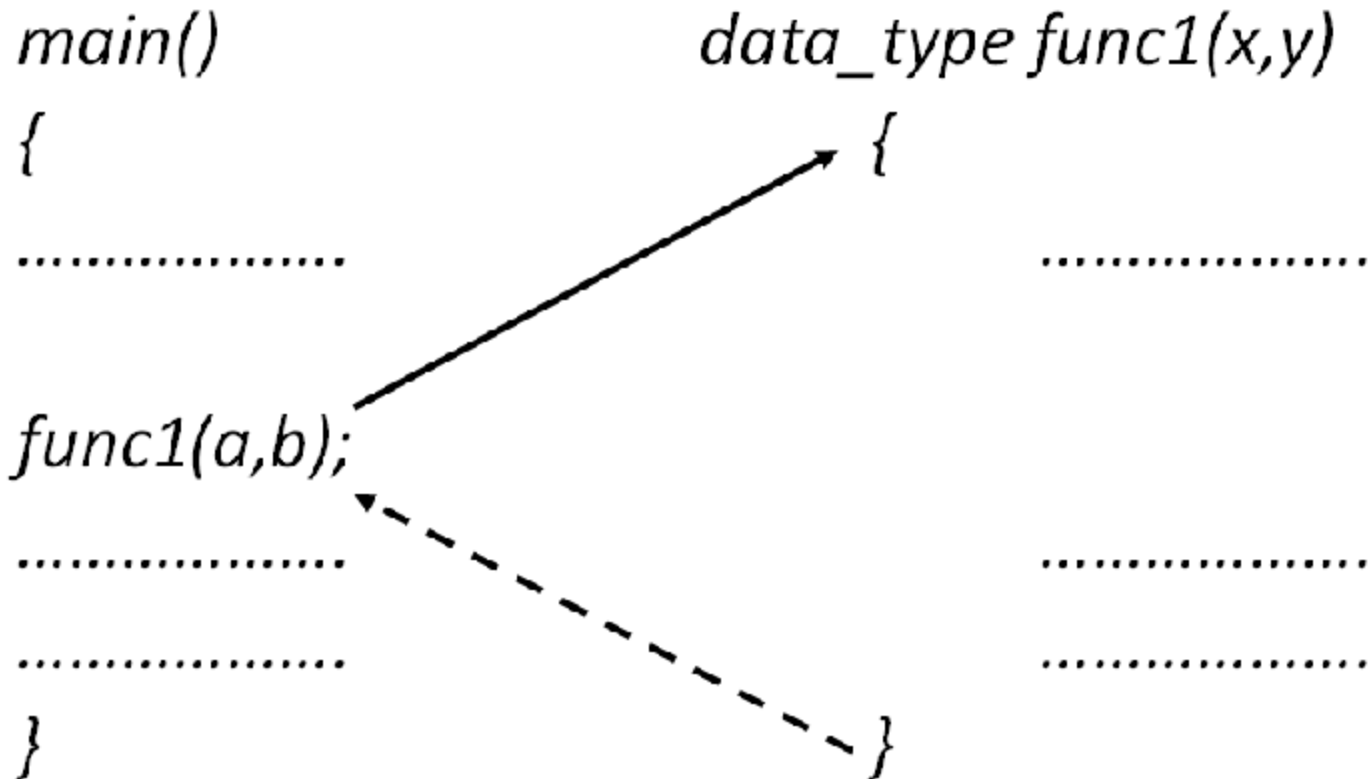
4

Sum is:7

2. Function with arguments and no return values

- Here data transfer take place between the calling function and the called function.
- It is a **one way data communication**, i.e. the called program receives data from calling program but it does not return any value to the calling program.

Function with arguments and no return values



Example

```
#include <stdio.h>
#include<conio.h>
void add(int,int);
void main()
{
    int a,b;
    clrscr();
    printf("\nEnter two number:");
    scanf("%d%d",&a,&b);
    add(a,b);
}
void add(int x,int y) //function with arguments
{
    int z;
    z=x+y;
    printf("\nSum is:%d",z);
}
```

Output

Enter two number:2

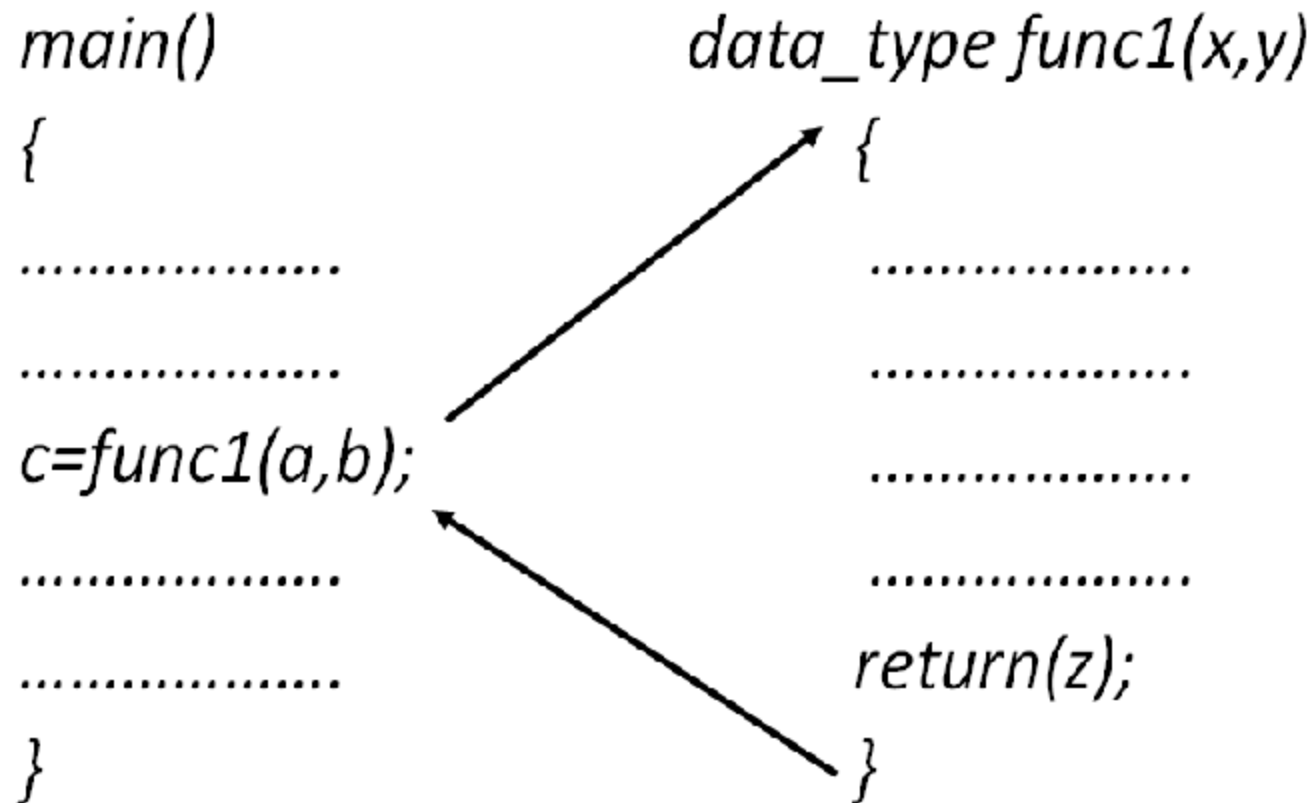
4

Sum is:6

3. Function with arguments and return values

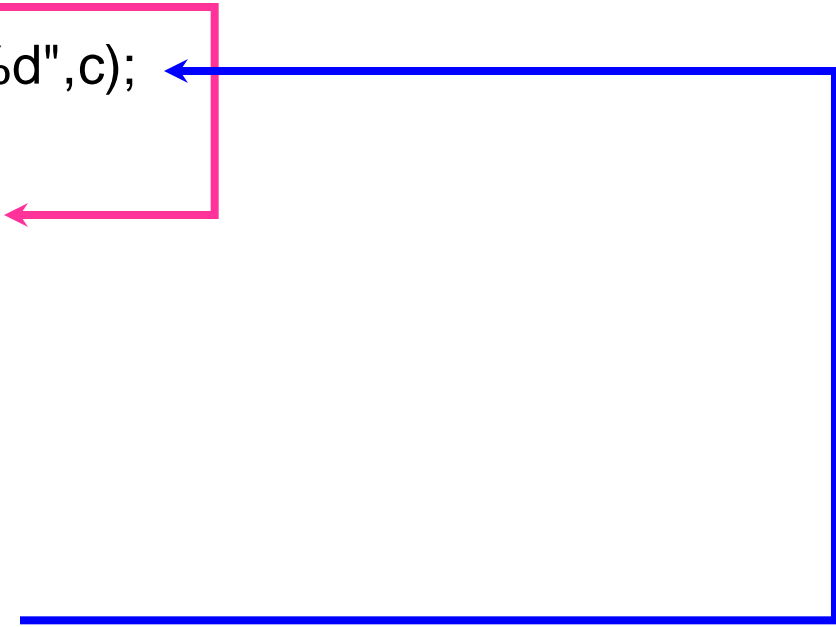
- Here data transfer take place between the calling function and the called function as well as between called function and calling function .
- It is a **two way data communication**, i.e. the called program receives data from calling program and it return some value to the calling program.

Function with arguments and return values



Example

```
#include <stdio.h>
#include <conio.h>
int add(int,int);
int main()
{
    int a,b,c;
    clrscr();
    printf("\nEnter two number:");
    scanf("%d%d",&a,&b);
    c=add(a,b);
    printf("\nSum is:%d",c);
}
int add(int x,int y)
{
    int z;
    z=x+y;
    return(z);
}
```



The diagram illustrates the execution flow between the `main` function and the `add` function. A pink line originates from the `c=add(a,b);` statement in the `main` function, moves right, then down, then left, ending with an arrow pointing to the `int add(int x,int y)` function definition. A blue line originates from the `printf("\nSum is:%d",c);` statement in the `main` function, moves right, then down, then left, ending with an arrow pointing to the `return(z);` statement in the `add` function.

Output

Enter two number:6

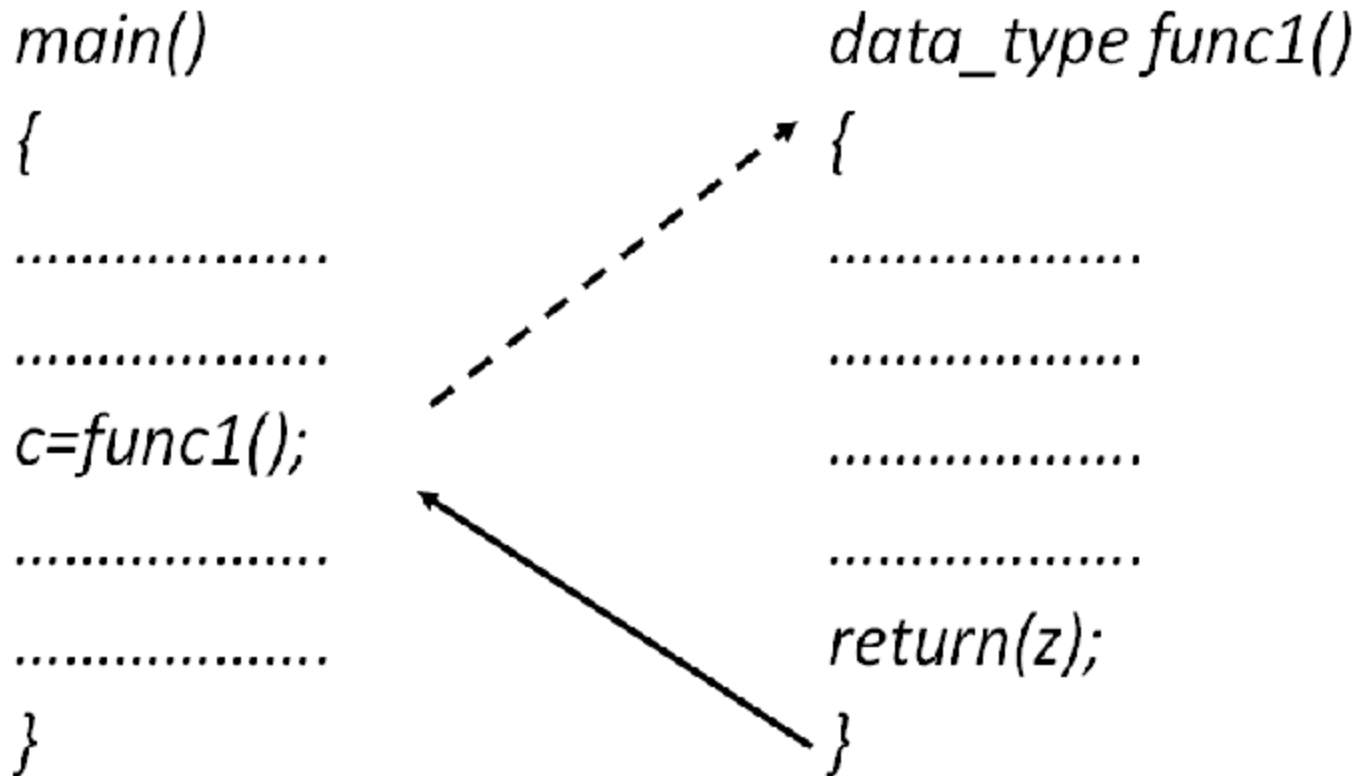
7

Sum is:13

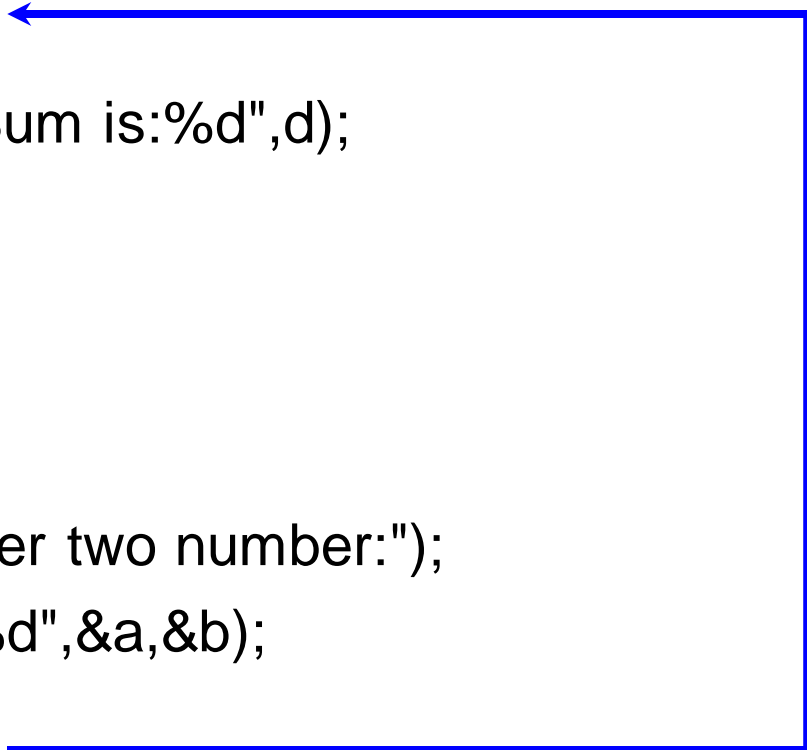
4. Function with no arguments and with return values

- Here data transfer take place between the called function and the calling function.
- It is a **one way data communication**, i.e. the called program does not receives data from calling program but it return some value to the calling program.

Function with no arguments and with return values



```
#include <stdio.h>
#include<conio.h>
int add();
int main()
{
    int d;
    d=add();
    printf("\nSum is:%d",d);
}
int add()
{
    int a,b,c;
    printf("\nEnter two number:");
    scanf("%d%d",&a,&b);
    c=a+b;
    return(c);
}
```



Output

Enter two number:5

8

Sum is:13

return **expression**

- ⊕ **The return statement is used to return from a function.**
- ⊕ **It causes execution to return to the point at which the call to the function was made.**
- ⊕ **The return statement can have a value with it, which it returns to the program.**

Parameter Passing Methods

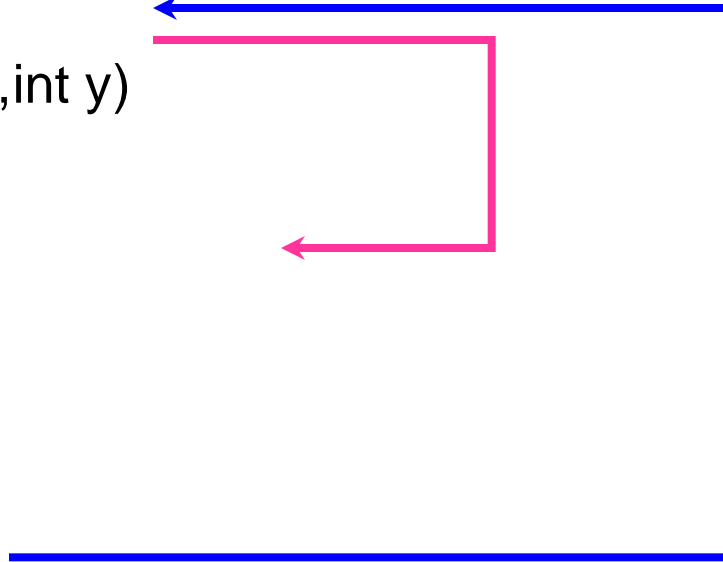
- Call by value
- Call by reference

Call by value

- Actual argument passed to the formal argument.
- Any changes to the formal argument does not affect the actual argument.

Example

```
#include <stdio.h>
#include <conio.h>
int add(int,int);
int main()
{
    int a,b,c;
    clrscr();
    printf("\nEnter two number:");
    scanf("%d%d",&a,&b);
    c=add(a,b);
    printf("\nSum is:%d",c);
}
int add(int x,int y)
{
    int z;
    z=x+y;
    return(z);
}
```



Output

Enter two number:6

7

Sum is:13

Call by reference

- Instead of passing value, the address of the argument will be passed.
- Any changes to the formal argument will affect the actual argument.

Example

```
#include <stdio.h>
#include <conio.h>
void swap(int*,int*);
void main()
{
    int x,y;
    printf("\nEnter value of x:");
    scanf("%d",&x);
    printf("\nEnter value of y:");
    scanf("%d",&y);
```

```
    swap(&x,&y);
    printf("\nx=%d,y=%d",x,y);
}
void swap(int *a,int *b)
{
    int c;
    c=*a;
    *a=*b;
    *b=c;
    printf("\nx=%d,y=%d",*a,*b);
}
```

Output

Enter value of x:5

Enter value of y:6

x=6,y=5

Library Function

- It is pre-defined function.
- The library function provides functions like mathematical, string manipulation etc,.

Example

sqrt(x):

It is used to find the square root of **x**

Example: **sqrt(36)** is 6

abs(x):

It is used to find the absolute value of **x**

Example: **abs(-36)** is 36

pow(x,y):

It is used to find the value of **x^y**

Example: **pow(5,2)** is 25

ceil(x):

It is used to find the smallest integer greater than or equal to **x**

Example: **ceil(7.7)** is 8

rand():

It is used to generate a random number.

sin(x):

It is used to find the sine value of **x**

Example: **sin(30)** is 0.5

cos(x):

It is used to find the cosine value of **x**

Example: **cos(30)** is 0.86

tan(x):

It is used to find the tan value of **x**

Example: **tan(30)** is 0.577

toascii(x):

It is used to find the ASCII value of **x**

Example: **toascii(a)** is 97

toupper(x):

It is used to convert lowercase character to uppercase.

Example: **toupper('a')** is A

toupper(97) is A

tolower(x):

It is used to convert uppercase character to lowercase.

Example: **tolower('A')** is a

Why are Pointers Used ?

- **To return more than one value from a function**
- **To pass arrays & strings more conveniently from one function to another**
- **To manipulate arrays more easily by moving pointers to them, Instead of moving the arrays themselves**
- **To allocate memory at run time and access it (Dynamic Memory Allocation)**
- **To create complex data structures such as Linked List, Where one data structure must contain references to other data structures**

Pointers

Pointer is the variable which stores the address of the another variable

Declaration of pointer :

syntax : datatype *pointername;

Example :

int *ptr;

char *pt;

& → Returns the memory address of the operand

*** → It is the complement of &. It returns the value contained in the memory location pointed to by the pointer variable's value**

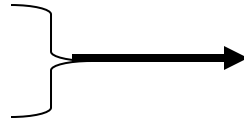
Assigning data to the pointer variable

syntax :

pointervariablename=&variablename;

For Example :

```
int *a,b=10;  
a=&b;  
int *p,quantity=20;  
p=&quantity;
```



Variable	Value	Address
Quantity	20	500
P	500	5048

For Example :

```
#include<stdio.h>  
void main()  
{  
int val=100;  
printf("%u\n",&val);  
printf("%d\n",val);  
printf("%d\n",*(&val));  
}
```

Pointer Arithmetic

✿ Addition and subtraction are the only operations that can be performed on pointers.

✿ Take a look at the following example :

```
int var, *ptr_var;  
ptr_var = &var;  
var = 500;  
ptr_var++ ;
```

✿ Let var be an integer type variable having the value 500 and stored at the address 1000.

✿ Then ptr_var has the value 1000 stored in it. Since integers are 2 bytes long, after the expression “ptr_var++;” ptr_var will have the value as 1002 and not 1001.

HINTS

- ✿ Each time a pointer is incremented, it points to the memory location of the next element of its base type.
- ✿ Each time it is decremented it points to the location of the previous element.
- ✿ All other pointers will increase or decrease depending on the length of the data type they are pointing to.
- ✿ Two pointers can be compared in a relational expression provided both the pointers are pointing to variables of the same type.

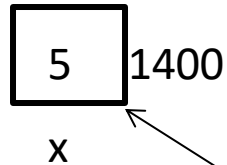
Advantages:

- ☺ A pointer enables us to access a variable that is defined outside the function.
- ☺ Pointers are more efficient in handling the data tables.
- ☺ Pointers reduce the length and complexity of a program.
- ☺ They increase the execution speed.
- ☺ The use of a pointer array to character strings results in saving of data storage space in memory.
- ☺ The function pointer can be used to call a function
- ☺ Pointer arrays give a convenient method for storing strings
- ☺ Many of the 'C' Built-in functions that work with strings use Pointers
- ☺ It provides a way of accessing a variable without referring to the variable directly

Passing pointers to function

```
Int main()  
{  
.....  
.....  
foo(&x);  
}
```

int x=5;



```
void foo(int *a)  
{  
  
int m;  
m=*a;  
m=m+1;  
*a=m;  
}
```

a



*a

&x=1400

Dynamic Memory Allocation

- `malloc`
 - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space
- `calloc`
 - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- `free`
 - Frees previously allocated space.
- `realloc`
 - Modifies the size of previously allocated space.

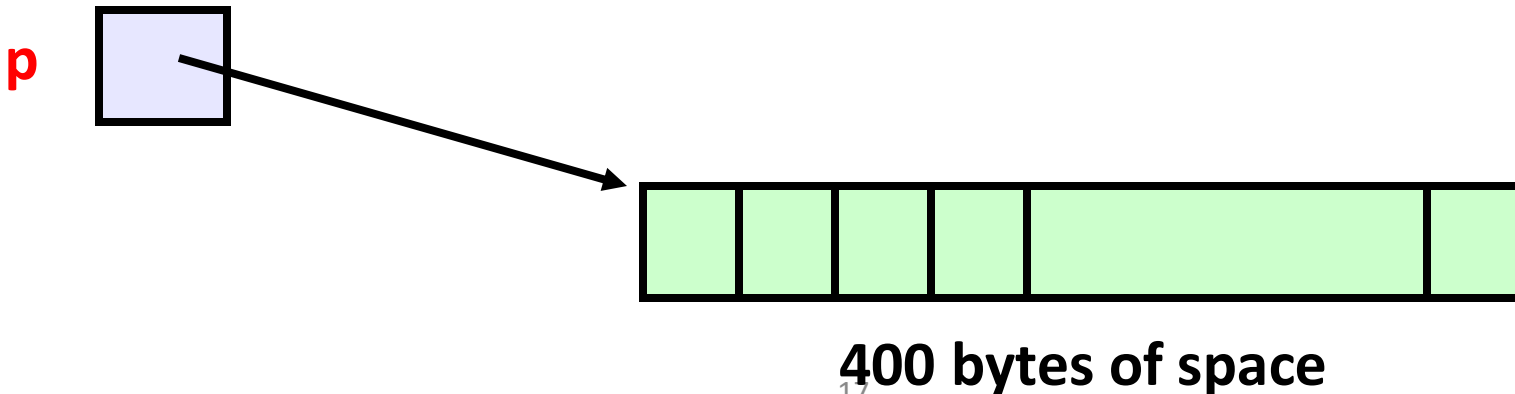
1.Allocating a Block of Memory

- A block of memory can be allocated using the function `malloc`
 - Reserves a block of memory of specified size and returns a pointer of type `void`
 - The return pointer can be type-casted to any pointer type
- General format:
`type *p;`
`p = (type *) malloc (byte_size);`

Example

```
p = (int *) malloc(100 * sizeof(int));
```

- A memory space equivalent to 100 times the size of an int bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer p of type int



Contd.

- `cptr = (char *) malloc (20);`

Allocates 20 bytes of space for the pointer `cptr` of type `char`

Always use sizeof operator to find number of bytes for a data type, as it can vary from machine to machine

Points to Note

- `malloc` always allocates a block of contiguous bytes
 - The allocation can fail if sufficient contiguous memory space is not available
 - If it fails, `malloc` returns `NULL`

```
if ((p = (int *) malloc(100 * sizeof(int))) == NULL)
{
    printf ("\n Memory cannot be allocated");
    exit();
}
```

Using the malloc'd Array

- Once the memory is allocated, it can be used with pointers, or with array notation
- Example:

```
int *p, n, i;  
scanf("%d", &n);  
p = (int *) malloc (n * sizeof(int));  
for (i=0; i<n; ++i)  
    scanf("%d", &p[i]);
```

The n integers allocated can be accessed as `*p`, `*(p+1)`, `*(p+2)`, ..., `*(p+n-1)` or just as `p[0]`, `p[1]`, `p[2]`, ..., `p[n-1]`

2. Releasing the Allocated Space: `free`

- An allocated block can be returned to the system for future use by using the `free` function
- General syntax:
`free (ptr);`
where `ptr` is a pointer to a memory block which has been previously created using `malloc`
- Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

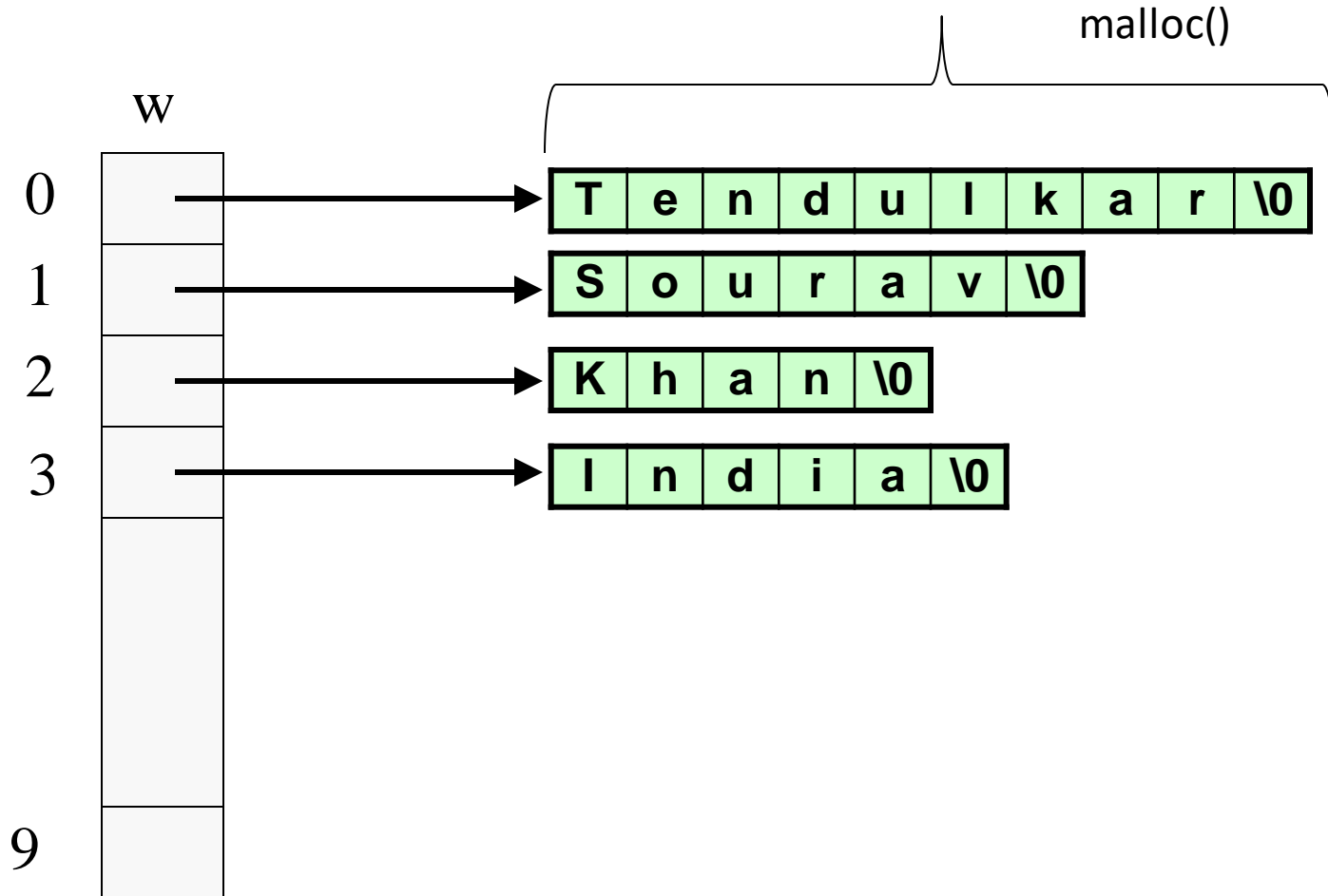
Static array of pointers

```
#define N 20
#define M 10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d] = %s \n",i,w[i]);
    return 0;
}
```

Output

```
4
Tendulkar
Sourav
Khan
India
w[0] = Tendulkar
w[1] = Sourav
w[2] = Khan
w[3] = India
```

How it will look like



Pointers to Pointers

- Pointers are also variables (storing addresses), so they have a memory location, so they also have an address
- Pointer to pointer – stores the address of a pointer variable

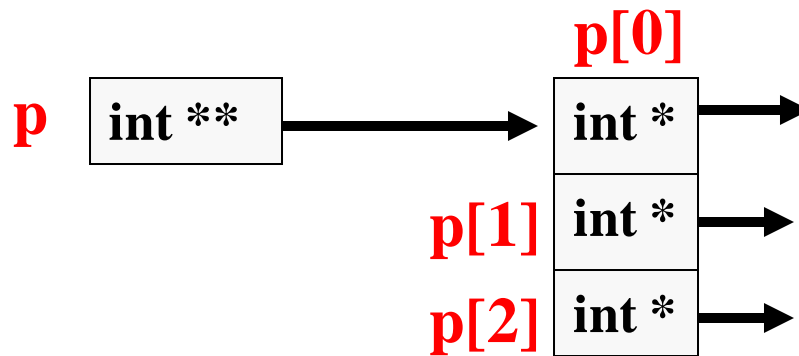
```
int x = 10, *p, **q;  
p = &x;  
q = &p;  
printf("%d %d %d", x, *p, *(*q));
```

will print 10 10 10 (since *q = p)

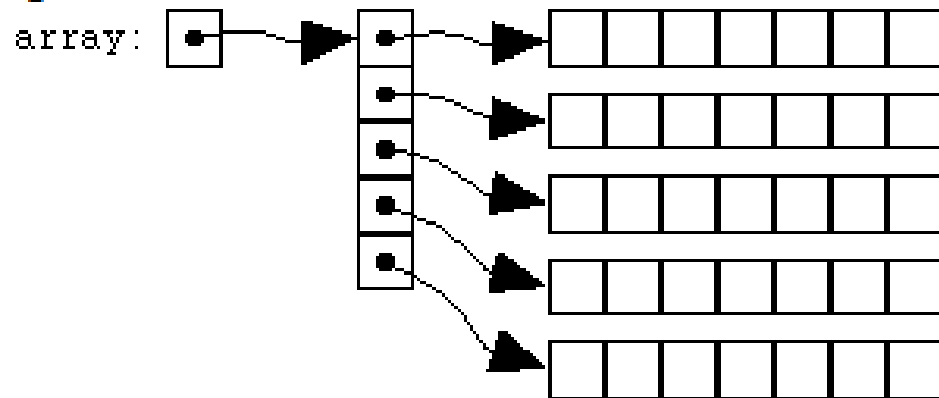
Allocating Pointer to Pointer

```
int **p;
```

```
p = (int **) malloc(3 * sizeof(int *));
```



2D array

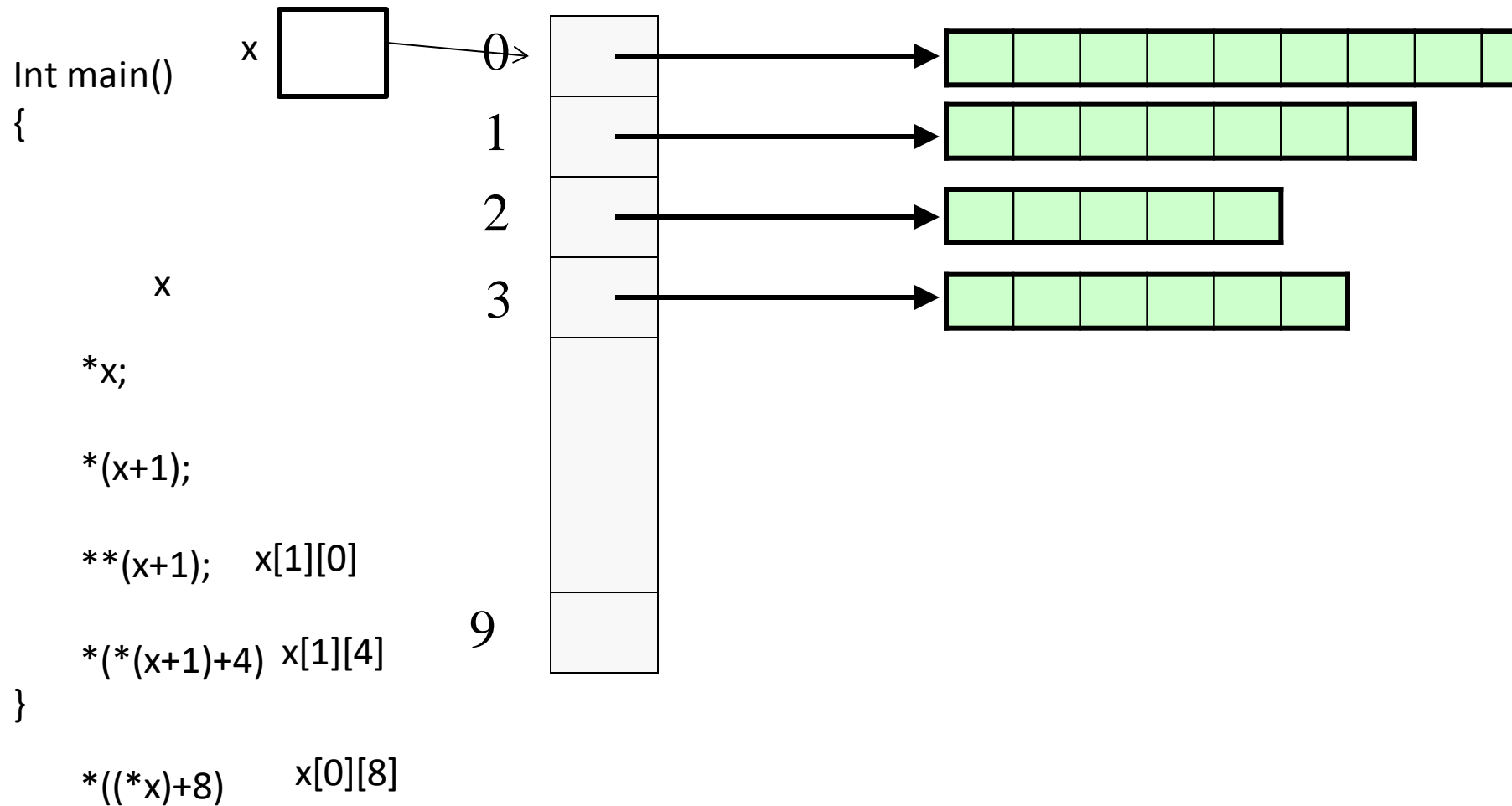


```
#include <stdlib.h>
```

```
int main()
```

```
{  
    int **array;  
    array = (int**) malloc(nrows * sizeof(int *));  
  
    for(i = 0; i < nrows; i++)  
    {  
        array[i] = (int*)malloc(ncolumns * sizeof(int));  
    }  
  
}
```

2D array



3. Array Allocation with calloc

prototype: `void * calloc(size_t num, size_t esize)`

`size_t` is a special type used to indicate sizes, generally an unsigned int

`num` is the number of elements to be allocated in the array

`esize` is the size of the elements to be allocated

generally use `sizeof` and type to get correct value

an amount of memory of size `num*esize` allocated on heap

`calloc` returns the address of the first byte of this memory

generally we cast the result to the appropriate type

if not enough memory is available, `calloc` returns `NULL`

4. Increasing Memory Size with realloc

prototype: `void * realloc(void * ptr, size_t esize)`

`ptr` is a pointer to a piece of memory previously dynamically allocated

`esize` is new size to allocate (no effect if `esize` is smaller than the size of the memory block `ptr` points to already)

program allocates memory of size `esize`,

then it copies the contents of the memory at `ptr` to the first part of the new piece of memory,

finally, the old piece of memory is freed up

QUESTIONS

IN C, PARAMETERS ARE ALWAYS?

- (a) Pass by value
- (b) Pass by reference
- (c) Non pointer variables are passed by value and pointers are passed by reference.
- (d) Passed by value result

ANSWER

Pass by value only

```
#include <stdio.h>

void function2(int *param) {
    printf("address param is pointing to %d\n", param);
    param = NULL;
}

int main(void) {
    int variable = 111;
    int *ptr = &variable;

    function2(ptr);
    printf("address ptr is pointing to %d\n", ptr);
    return 0;
}
```

```
address param is pointing to -1846583468
address ptr    is pointing to -1846583468
```

2) IDENTIFY THE OUTPUT

```
char c[] = "GATE2011";
```

```
char *p =c;
```

```
printf("%s", p + p[3] - p[1]) ;
```

- (A) GATE2011
- (B) 2011
- (C) E2011
- (D) ATE2011

ANSWER (B)

$$\text{'E'} - \text{'A'} = 4$$

Base + 4 leads to “2011”
starting

3) IDENTIFY THE CORRECT OPTION

```
int f(int n)
{
    static int i = 1;
    if(n >= 5) return n;
    n = n+i;
    i++;
    return f(n);
}
```

(a) 5 (b) 6 (c) 7 (d) 8

(C) 7

Execution of f(1)

i = 1

n = 2

i = 2

Call f(2)

i = 2

n = 4

i = 3

Call f(4)

i = 3

n = 7

i = 4

— Call f(7) which gives 7