Unit 4
Introduction to functions and pointers

Mohd Adil
Assistant Professor
Dayananda Sagar College of Engineering

Basics of function

Definition:

Function is basically a set of statements that takes inputs, perform some computation and produces output.

Syntax:

```
Return_type function_name(set_of_inputs);
```

- Return_type=> the type of output returned by the function.
- Function_name => Name of the function.
- Set_of_inputs => inputs provided to the function

Why functions

- There are two important reasons of why we are using functions:
- 1. Reusability
- Once the function is defined, it can be reused over and overagain.
- 2. Abstraction
- If you are just using the function in your program, then you don't have to worry about how it works inside!
- Example: scanf function

Function Declaration

- As we already know, when we declare a variable, we declare its properties to the compiler.

- For example: int var;
- Properties:
- 1. Name of variable: var
- 2. Type of variable: int

Function Declaration

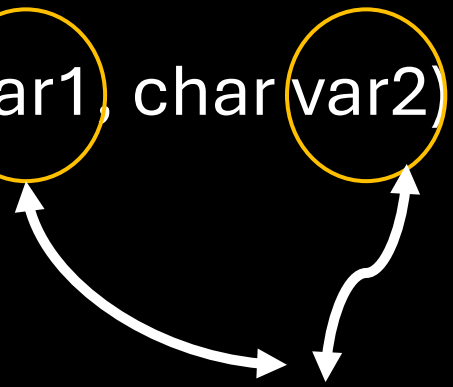
- Similarly, function declaration (also called function prototype) means declaring the properties of a function to the compiler.
- For example: int fun(int, char);
- Properties: note function prototype end with

1. Name of function : fun semicolon
2. Return Type of function: int
3. Number of parameters: 2
4. Type of parameter 1: int
5. Type of parameter 2:char

Function Declaration points.....

- It is not necessary to put the name of the parameters in function prototype.

For example: int fun(int var1, char var2);



Not necessary to mention these names

Classification of Functions

Functions are classified into **4 types** based on:

- Whether they take **arguments (input)**
- Whether they **return a value (output)**

1. No Arguments, No Return Value
2. With Arguments, No Return Value
3. No Arguments, With Return Value
4. With Arguments, With Return Value

Function Definition

- What is function definition ?
- Function definition consists of block of code which can perform some specific task.

For example:

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

How function works

```
int add(int, int);
int main(){
int m = 20, n = 30, sum;
sum = add(m, n);

printf("sum is %d", sum);
}
int add(int a, int b){
return ( a + b );
}
```

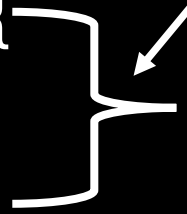
Don't Forget to mention the prototype of a function.
Recall: There is no need to mention names of the parameters.

This is the way you call a function.
NOTE: while calling a function, you should not mention the return type of the function Also you should not mention the data types of the arguments

How function works

```
int add(int, int);
int main(){
int m = 20, n = 30, sum;
sum = add(m, n);
printf("sum is %d", sum);
}

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



This is the way how you define a function.
NOTE: it is important to mention both data type and name of parameters

What is the difference between an argument and a parameter?

Parameter: is a variable in the declaration and definition of the function.

Argument: is the actual value of the parameter that gets passed to the function.

NOTE: Parameter is also called as Formal Parameter and Argument is also called as Actual Parameter.

```
int add(int, int);
int main(){
int m = 20, n = 30, sum;
sum = add(m, n);
printf("sum is %d", sum);
}

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

Arguments or
Actual parameters

Parameters or
formal parameters

Points to be noted

1. We can make any no. of function
2. We can call the function any no. of times
3. We can call any function within any other function
4. We can define the function in any order
5. We can’t define a function within another function (including main)

IS IT ALWAYS NECESSARY TO DECLARE THE FUNCTION BEFORE USING IT?

- Not Necessary but it is preferred to declare the function before using it.

- Why is it not necessary?

```
#include <stdio.h>
char fun (){
return ‘a’;
}
int main (){
char c = fun ();
printf("character is: %c", c) ;
}
```

What happens when we use the function before defining it?

```
#include<stdio.h>
int main(){
char c=fun();
printf(“character is %c”,c);
}
char fun()
{
return ‘a’;
}
```

Defining Scope

Scope=Lifetime

The area under which a variable is applicable or alive.

Strict Definition: a block or a region where a variable is declared, defined and used and when a block or a region ends, variable is automatically destroyed.

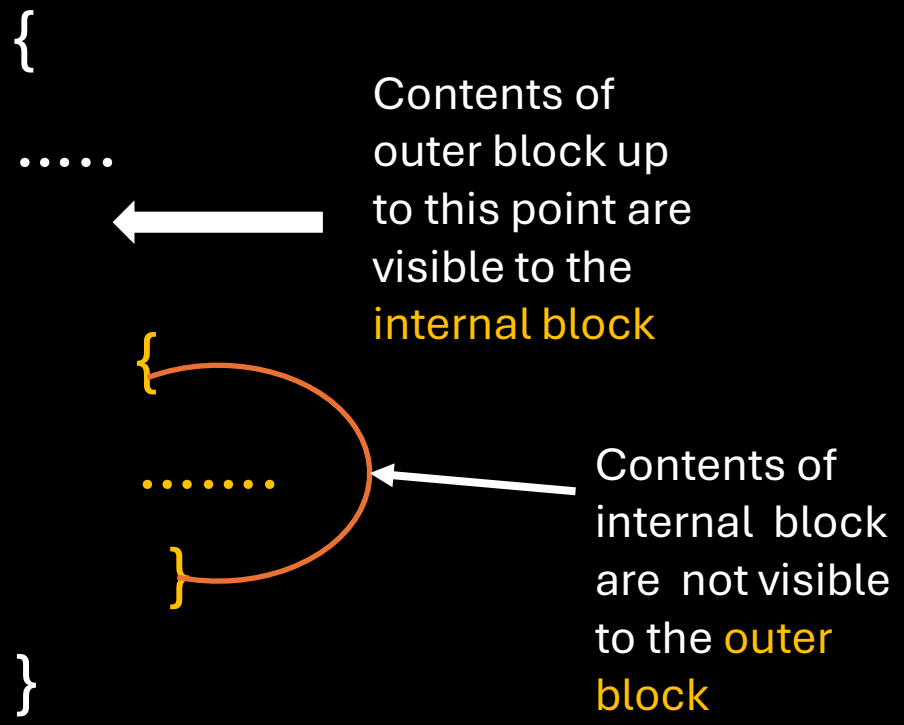
Defining Scope

```
#include <stdio.h>
Int main() {
Int var=34;
printf(“%d”, var);
return 0;
}

Int fun(){
printf(“%d”, var );
}
```

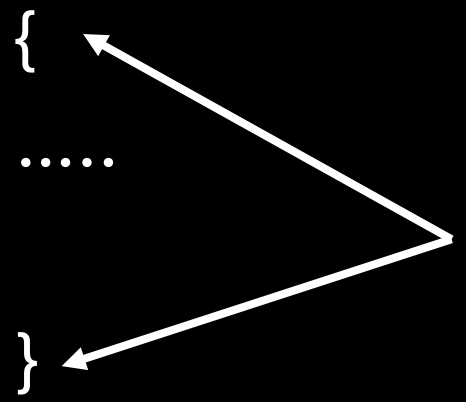
- Scope of this variable is within main() function only. Therefore called LOCAL to main() function
- Error: ‘var’ undeclared (first use in this function
- Trying to access variable ‘var’ Outside main() function

Basic principle of scoping

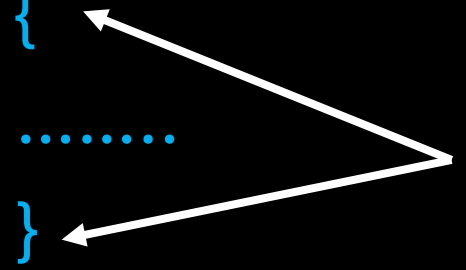


Contents of outer block up to this point are visible to the internal block

Contents of internal block are not visible to the outer block



Contents of this block is not visible to any block outside to this block



Contents of this block is not visible to any block outside to this block

```c
#include <stdio.h>

int main() {
    int var = 3;
    int var = 4;
    printf("%d\n", var);
    printf("%d", var);
    return 0;
}
```

```
error: redefinition of 'var'
```

```c
#include <stdio.h>

int main() {
    int var = 3;
    {
        int var = 4;
        printf("%d\n", var);
    }
    printf("%d", var);
    return 0;
}
```

Global variable

```
#include <stdio.h>
Int fun();
Int var=10;
Int main() {
Int var=34;
printf(“%d”, var);
return 0;
}

Int fun(){
printf(“%d”, var );
}
```

- This variable is outside of all functions
  Therefore called a Global variable
- Output: 3
- Output:10
- It will access the global variable

Static variable

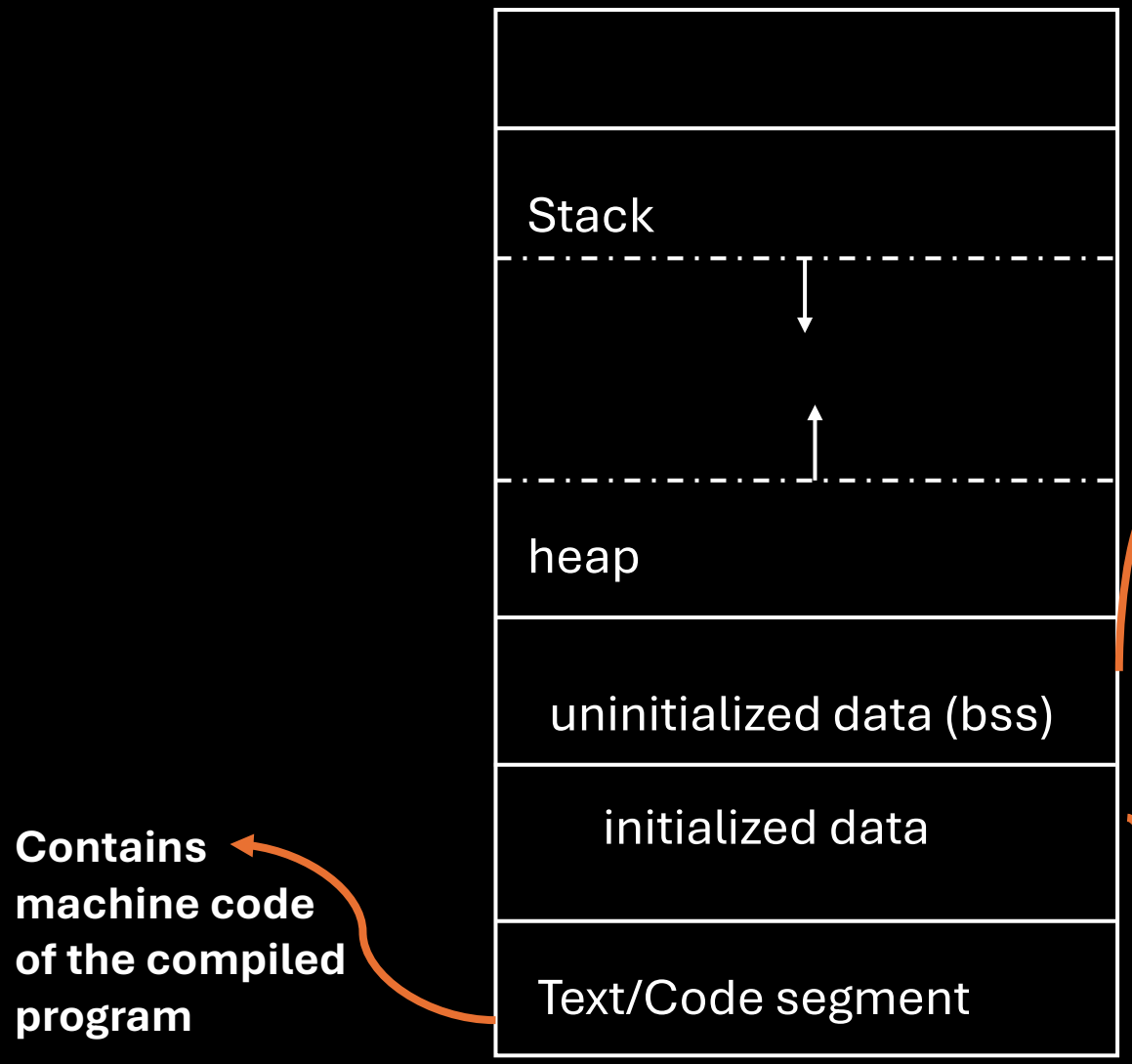
1. Static variable remains in memory even if it is declared within a block on the other hand automatic variable is destroyed after the completion of function in which it was declared.
2. Static variable if declared outside the scope of any function will act like global variable but only within the file in which it is declared.
3. You can only assign a constant literal (or value) to a static variable.

Memory layout of c program

Two memory segments:

1. Text/code segment
2. Data segment
   - a) Initialized
     - I) Read only
     - ii) Read write
   - b) Uninitialized

     (bss – Block started by Symbol)
   - c) Stack
   - d) Heap



Questions 1

```
#include<stdio.h>
int fun1(int n){
int count=0;
while(n){
count++;
num >> =1;
}
return count;
}
```

```
int main(){
printf("%d",fun1(435));
return 0;
}
```

The value returned by fun1(435) is_____

435 = 110110011

011011001	iter 1
001101100	iter 2
000110110	iter 3
000011011	iter 4
000001101	iter 5
000000110	iter 6
000000011	iter 7
000000001	iter 8
000000000	iter 9

Questions 2

```
#include<stdio.h>
void f1(int a , int b){
int c;
c=a;
a=b;
b=c;
}
```

```
int main(){
Int a=4, b=5, c=6
f1(a,b);
printf(“%d”, c-a-b);
return 0;
}
```

Questions 3

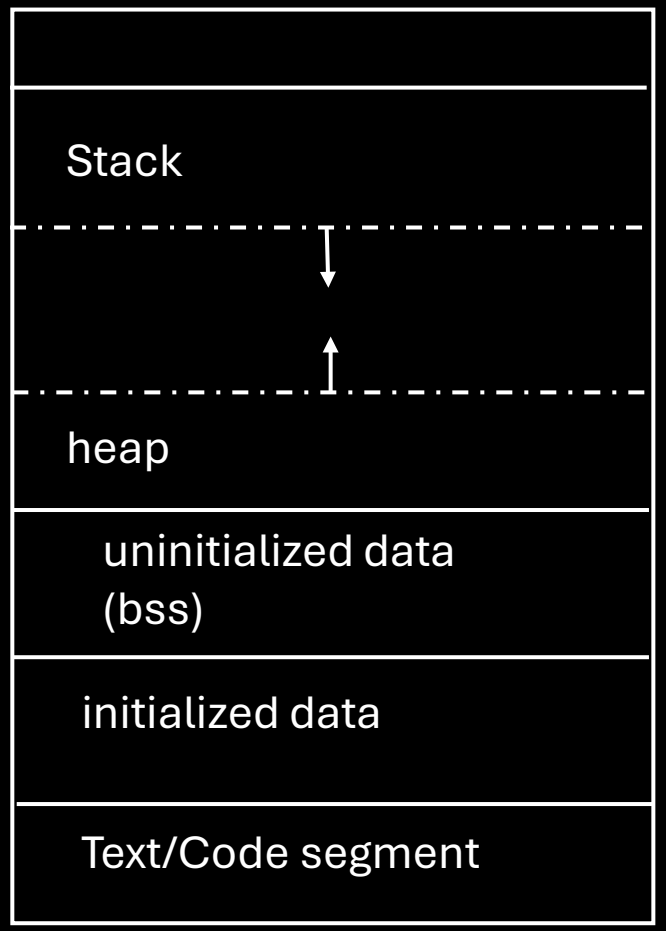
```
#include<stdio.h>
Int fun(){
static int num=16;
return num --;
}
```

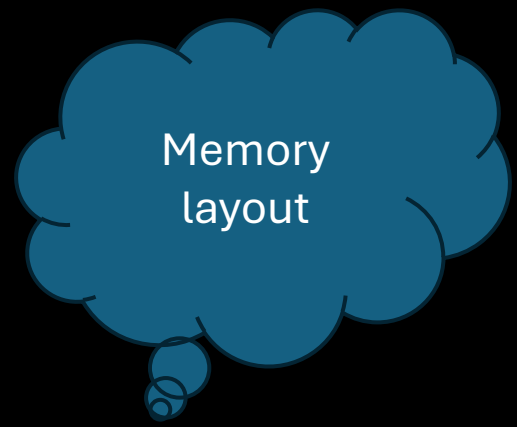
```
int main(){
for(fun();fun();fun())
printf(“%d”, fun());
return 0;
}
```

Static function

- In c, functions are global by default.
- This means if we want to access the function outside from the file where it is declared, we can access it easily.
- Now if we want to restrict this access, then we make our function static by simply putting a keyword static in front of the function.
- Static functions are restricted to the files where they are declared.
- Reuse of the same function in another file is possible

Static scoping and dynamic scoping

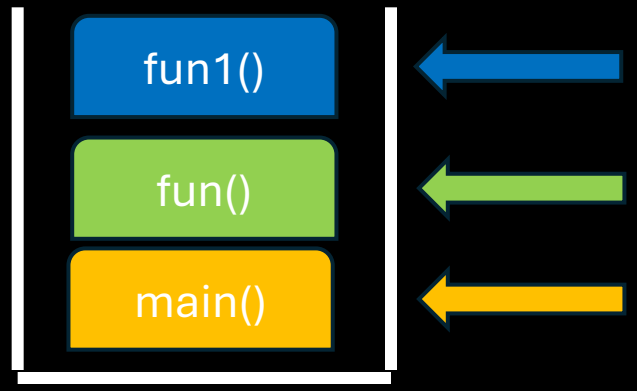




Static scoping and dynamic scoping

- Stack is a container (or memory segment) which holds some data.
- Data is retrieved in Last In First Out (LIFO) fashion.
- Two operations: push and pop.

```
main(){
      fun();
}
fun() { fun1();}
fun1(){return }
```

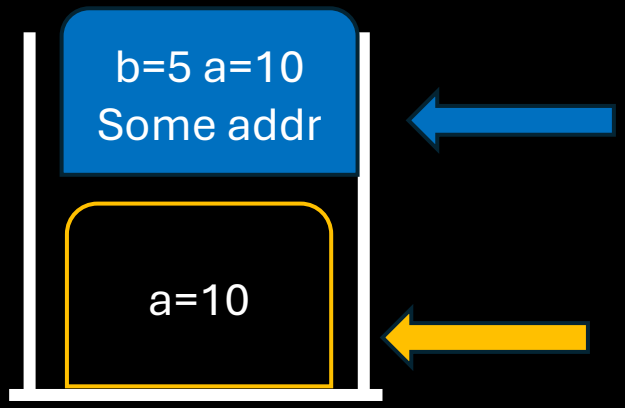


stack (or call stack)

Static scoping and dynamic scoping

- Activation Record is a portion of a stack which is generally composed of:
- Local of the callee
- Return address to the caller
- Parameter of the callee

```
 int main(){
int a = 10;
a = fun1(a);
printf("%d",a);
}
int fun1(int a)
int b = 5;
return b;
```



Why scoping

- Scoping is necessary if you want to reuse variable names

Example:

```
int fun1(){
int a = 10;
}
int fun2()
{
int a = 40;
```

What is static scoping

In Static scoping (or lexical scoping), definition of a variable is resolved by searching its containing block or function.

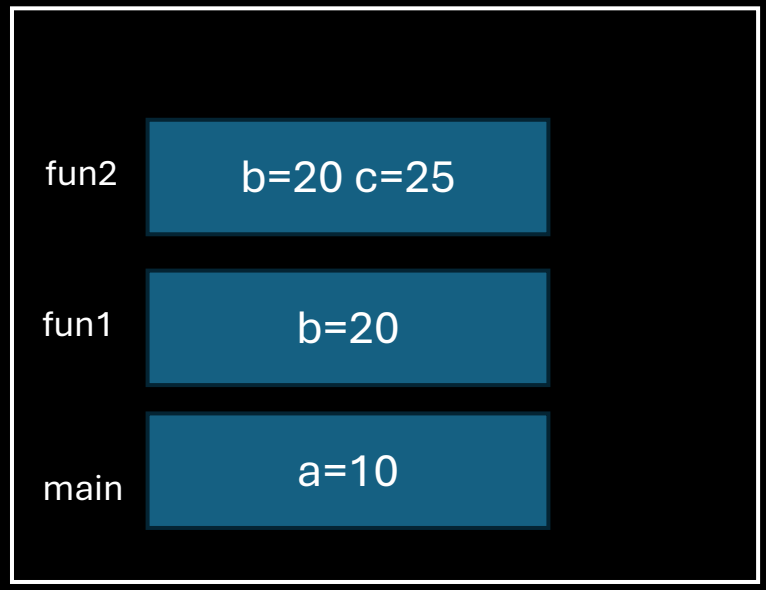
If that fails, then searching the outer containing block and so on.

```
int a = 10, b = 20;
int fun(){
int a = 5;
{
int c;
c = b/a;
printf("%d",c);
}
}
```

static scoping example

```
int fun1(int);
int fun2(int);
int a = 5;
int main(){
int a = 10;
a = fun1(a);
printf("%d", a);
}
int fun1(int b){
b = b+10;
b = fun2(b);
return b;
}
```

```
int fun2(int b){
int c;
c=a+b;
return c;
}
```



Call stack

Initialized data segment

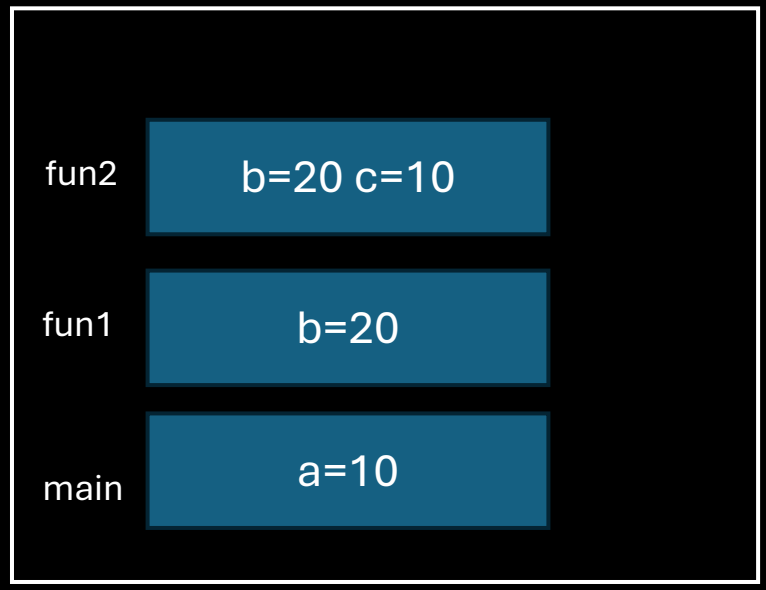
What is dynamic scoping

- In dynamic scoping, definition of a variable is resolved by searching its containing block and if not found, then searching its calling function and if still not found then the function which called that calling function will be searched and so on.

dynamic scoping example

```
int fun1(int);
int fun2(int);
int a = 5;
int main(){
int a = 10;
a = fun1(a);
printf("%d", a);
}
int fun1(int b){
b = b+10;
b = fun2(b);
return b;
}
```

```
int fun2(int b){
int c;
c=a+b;
return c;
}
```



Call stack

Initialized data segment

some important point on scoping

1. In most of the programming languages, static scoping is followed instead of dynamic scoping
2. Languages, including Algol, Pascal, C, Haskell, Scheme etc. are statically scoped
3. Some languages, including SNOBOL, APL, Lisp etc. are dynamically scoped
4. As C follows static scoping therefore it is not possible to see programmatically, how dynamic scoping works in C.

Recursion function

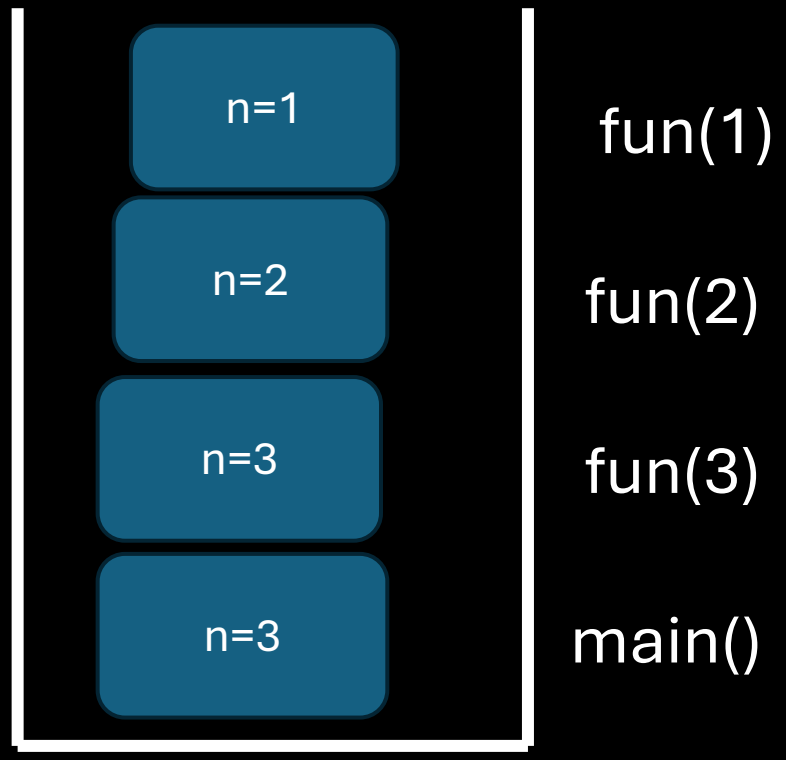
Recursion is a process in which a function calls itself directly or indirectly.

For example,

```
int fun(){
..
fun();
}
```

Recursion function example

```
int fun ( n ){
If(n==1)
return 1;
else
   return 1 + fun(n-1);
}
int main() {
Int n=3;
printf(“%d”,fun(n));
return 0;
}
```



Recursion function example

```
int fun ( n ){
If(n==1)
return 1;
else
   return 1 + fun(n-1);
}
int main() {
Int n=3;
printf(“%d”,fun(n));
return 0;
}
```

fun(3)

return 1 +fun(2)

return 2

return 1

How to write recursion function

1. Divide the problem into smaller sub-problems.
2. Specify the base condition to stop the recursion.

Problem: Calculate the factorial of a number.

Factorial of 5:5 x 4 x 3 x 2 x 1 = 120

Basic structure of recursion

```
int fun ( n ){
If(n==1){

return 1;
}
else{

   return n *fun(n-1);
}
}
```

base condition

recursive call

1. Divide the problem into smaller sub-problems.

Calculate Fact(4)

Fact(1) = 1

Fact(2) = 2 * 1 = 2 * Fact(1)

Fact(3) = 3 * 2 * 1 =3 * Fact(2)

Fact(4) =4* 3 * 2 * 1 = 4 * Fact(3)

Fact(n)= n*fact(n-1)

Base condition is the one which doesn't require to call the same function again and it helps in stopping the recursion.

Recursion

Advantage

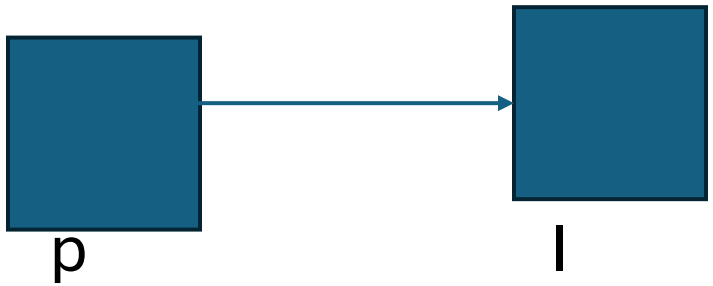
- Every recursive program can be written without recursion (using loops instead)
- but recursion looks cleaner and needs fewer lines of code

Disadvantage

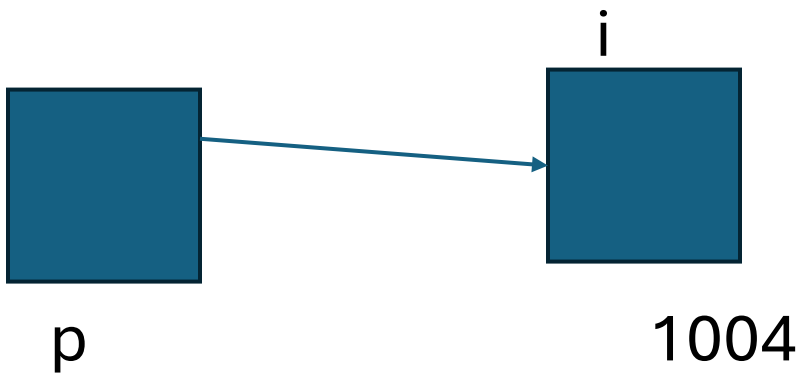
- Recursive programs require more space than iterative programs.

Introduction to pointers

- Pointer is a special variable that is capable of storing some address.



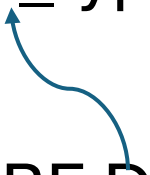
- It points to a memory location where the first byte is stored



Declaring and initializing pointers

General syntax for declaring pointer variables:

data_type *pointer_name



HERE DATA TYPE REFERS TO THE TYPE OF THE VALUE THAT THEPOINTER WILL POINT TO.

Declaring and initializing pointers

For example:

int *ptr ← points to integer value

char *ptr ← points to character value

float *ptr ← points to float value

Declaring and initializing pointers

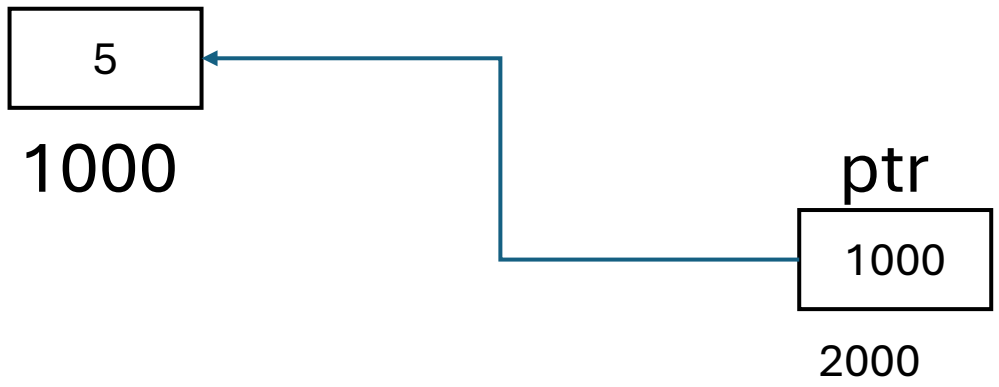
NEED OF ADDRESS OF OPERATOR

- Simply declaring a pointer is not enough.
- It is important to initialize pointer before use.
- One way to initialize a pointer is to assign address of some variable.

int x=5

int *ptr

ptr= &x



Value of operator

Value of operator/indirection operator/dereference operator is an operator that is used to access the value stored at the location pointed by the pointer.

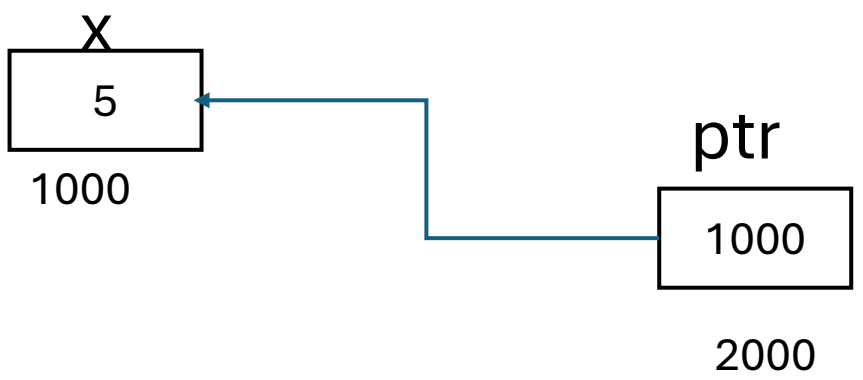
Int x=5;

Int *ptr;

ptr=&x;

printf(“%d”,*ptr);

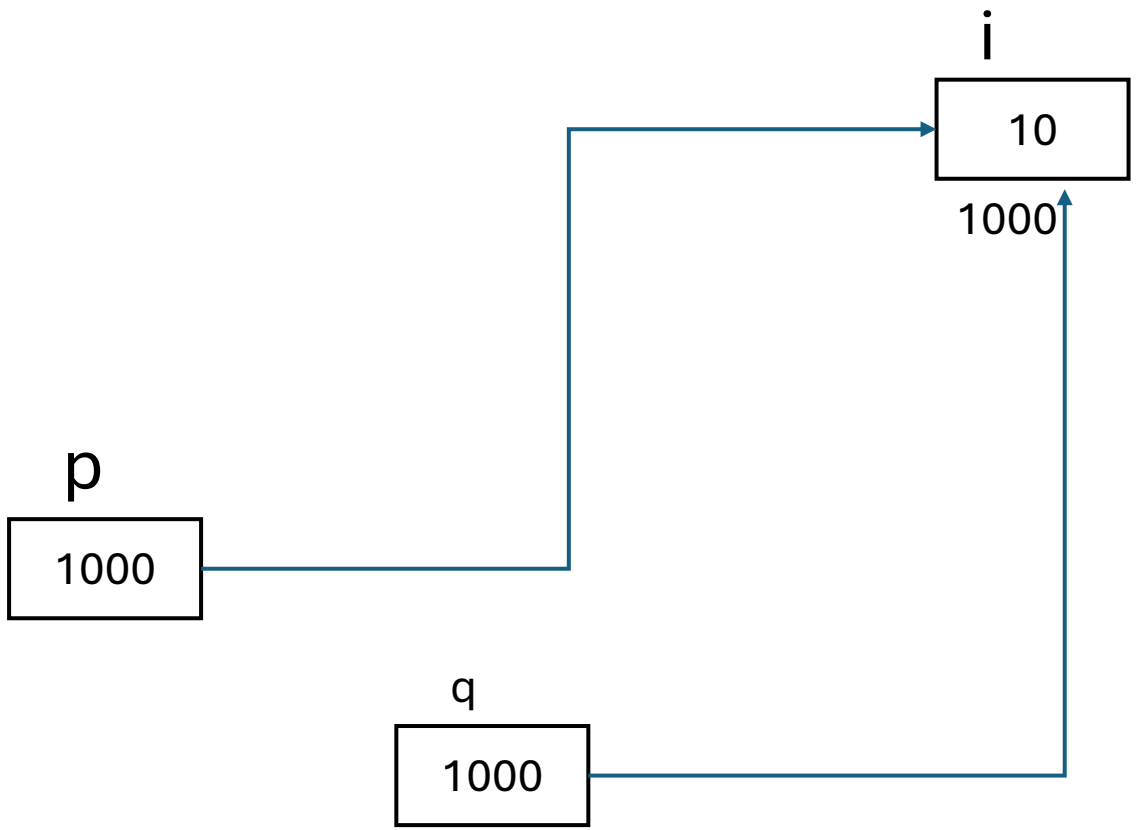
Value of operator



Pointer Assignment

```
int i=10;
int *p,*q;
p=&i;
q=p;
```

```
printf("%d",*p,*q);
```

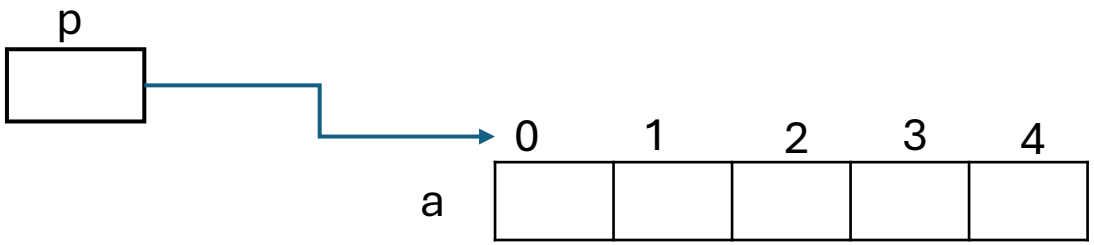


Returning pointers

```
int main(){
int a[]={1,2,3,4,5};
int n=sizeof(a)/sizeof(a[0]);
int *mid=findmid(a,n);
printf(“%d”,*mid);
return 0;
}
```

```
int *findmid(int a[],int n){
return &a[n/2];
}
```

Pointer Arithmetic (Addition)



p=&a[0]

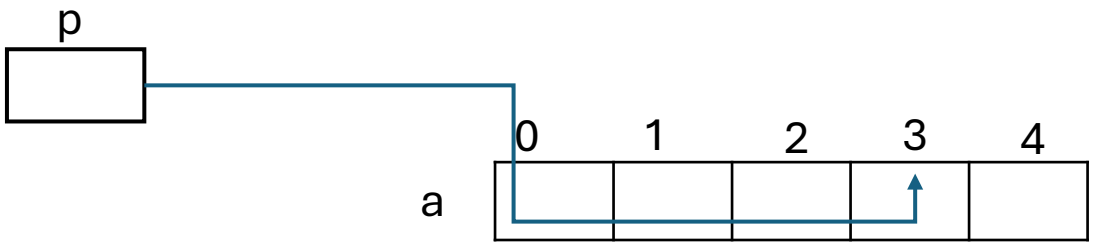
If we add some integer to the pointer means moving the pointer positions in forward direction

p=p+3

Initially, if p points to a[0], then

p=p+3 ≡ &a[0+3]

Pointer Arithmetic (subtraction)



p=&a[3]

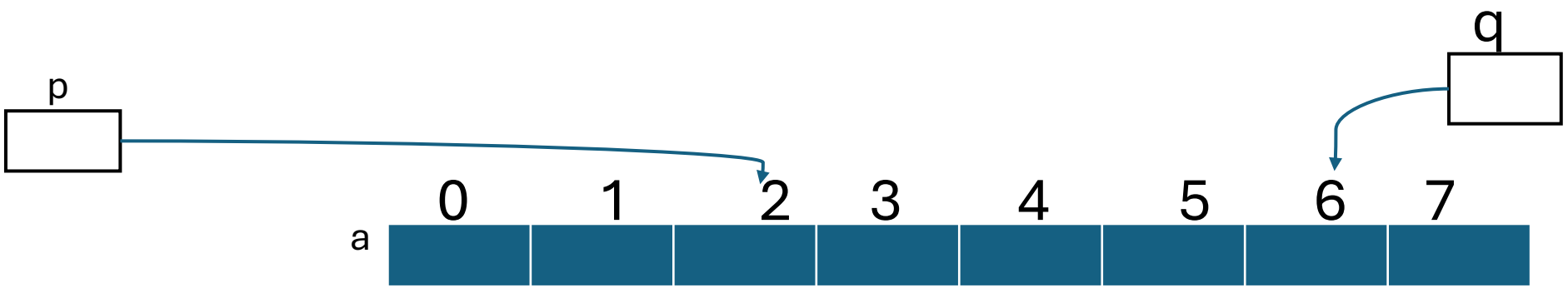
If we subtract some integer to the pointer means moving the pointer positions in backward direction

p=p-3

Initially, if p points to a[3], then

p=p-3 ≡ &a[3-3]

Pointer Arithmetic (subtraction)
Subtracting one pointer from another



q - p = 4

p - q = -4

Pointer Arithmetic Post and pre increment

```
Int main(){
Int  a[]={5,16,7,89,45,32,23,10};
Int *p=&a[0];
printf(“%d”,*(p++));
printf(“%d”,*(p));
return 0;
}
```

```
Int main(){
Int  a[]={5,16,7,89,45,32,23,10};
Int *p=&a[0];
printf(“%d”,*(++p));
printf(“%d”,*(p));
return 0;
}
```

Pointer Arithmetic Post and pre decrement

```
Int main(){
Int  a[]={5,16,7,89,45,32,23,10};
Int *p=&a[0];
printf(“%d”,*(p--));
printf(“%d”,*(p));
return 0;
}
```

```
Int main(){
Int  a[]={5,16,7,89,45,32,23,10};
Int *p=&a[0];
printf(“%d”,*(--p));
printf(“%d”,*(p--));
return 0;
}
```

Pointer Arithmetic comparing the pointers

- Use relational operators(< , > , <= , >=) and equality operators (== , !=) to compare pointers.
- Only possible when both pointers point to same array
- Output depends upon the relative positions of both the pointers.

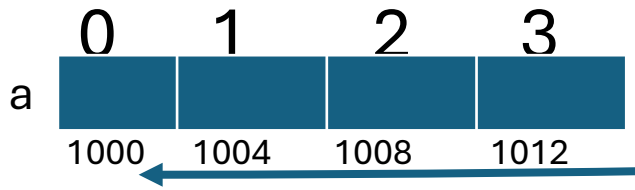
```
int main(){
int a[]={1,2,3,4,5,6};
int *p=&a[3];
int *q=&a[5];
printf("%d\n",p<=q);
printf("%d\n",p>=q);
q=&a[3];
printf("%d\n",p==q);
return 0;
}
```

Array name as pointer

```
int main(){
int a[]={11,22,36,5,2};
int sum=0, *p;
for(p=&a[0];p<=&a[4];p++)
sum+=*p;
printf(“sum is %d”,sum);
}
```

```
int main(){
int a[]={11,22,36,5,2};
int sum=0, *p;
for(p=a;p<=a+4;p++)
sum+=*p;
printf(“sum is %d”,sum);
}
```

Passing array as an argument to a function

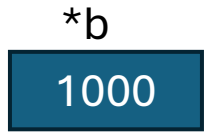



```
int main(){
int a[]={1,2,3,4};
int len= sizeof(a)/sizeof(a[0]);
printf("%d",add(a,len);
return 0;
}
```

We are not passing the whole array. We are just passing the base address of the array

```
int add(int b[], int len){
int sum=0,i;
for(i=0;i<len;i++)
        sum+=b[i];
return sum;
}
```

We can also write *b



Call by value and call by reference

- Actual parameters: The parameters passed to a function.
- Formal parameters: The parameters received by a function.

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

Actual parameters

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

formal parameters

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;
  }
```

Call by reference

- Here both actual and formal parameters refers to same memory location. Therefore, any changes made to the formal parameters will get reflected to actual parameters
- Here instead of passing values, we pass addresses.

```
int x=10, y=20;
fun(&x , &y);
printf("x= %d, y=%d", x , y);
```

```
int fun( int *a ,int *b){
*a=20;
 *b=10;
  }
```

Undefined behaviours

- Performing arithmetic on pointers which are not pointing to array element leads to undefined behavior

```
Int main(){
Int i=10;
Int *p=&I;
return(“%d”,*(p+3));
return 0;
}
```