Welcome to C-programming Class

Introduction to Pointer in C

A pointer in C is a variable that stores the address of another variable.

Instead of holding a value directly, a pointer points to the location in memory where the value is stored.

Introduction to Pointer in C

Basic Syntax

```
int a = 10;
int *p;
p = &a;   // p stores address of a
```

&a → gives address of variable a

p → stores that address

*p → gives value at that address

Memory Representation

Step 1: Normal Variable

Variable	Value	Address
a	10	1000

Step 2: Pointer Variable

Variable	Value	Address
a	10	1000
p	1000	2000

Basic Syntax

```
int a = 10;
int *p;
p = &a;   // p stores address of a
```

&a → gives address of variable a

p → stores that address

*p → gives value at that address

Memory Representation of Pointers

Step 1: Normal Variable

Variable	Value	Address
a	10	1000

Step 2: Pointer Variable

Variable	Value	Address
a	10	1000
p	1000	2000

Explanation:

a is stored at address 1000

Pointer p is stored at address 2000

p contains 1000 → address of a

Basic Syntax

```
int a = 10;
int *p;
p = &a;   // p stores address of a
```

&a → gives address of variable a
p → stores that address
*p → gives value at that address

Memory Representation of Pointers

Explanation:
a is stored at address 1000
Pointer p is stored at address 2000
p contains 1000 → address of a

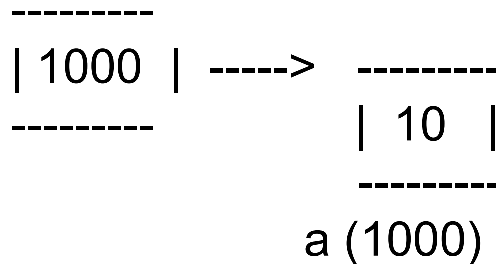
Step 3: Access Using Pointer
*p = 10

*p means:
“Go to address stored in p (1000) and get the value”

So, *p = 10

Memory Representation of Pointers

Diagram View



Key Points

Pointer stores address, not value

& → Address operator

* → Value at address (dereferencing)

Used for efficient memory handling, functions, arrays, and dynamic memory

Passing Array to Function

Simple Example

```
#include <stdio.h>
int main() {
    int a = 10;
    int *p = &a;
    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", &a);
    printf("Pointer p = %p\n", p);
    printf("Value using pointer = %d", *p);
    return 0;
}
```

Passing Array to Function

Simple Example

```
#include <stdio.h>
int main() {
    int a = 10;
    int *p = &a;
    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", &a);
    printf("Pointer p = %p\n", p);
    printf("Value using pointer = %d", *p);
    return 0;
}
```

```
Value of a = 10
Address of a = 0x7ffee4b3a9ac
Pointer p = 0x7ffee4b3a9ac
Value using pointer = 10
```

Passing Array to Function

Key Points
Pointer stores address, not value
& → Address operator
* → Value at address (dereferencing)

Used for efficient memory handling, functions, arrays, and dynamic memory

Simple example of pointers

Value of a = 10
Address of a = 0x7ffee4b3a9ac
Pointer p = 0x7ffee4b3a9ac
Value using pointer = 10

Explanation

Value of a = 10 → direct value
Address of a → memory location of a
Pointer p → same address (since p = &a)
*p = 10 → value stored at that address

Strings in C

Array of characters ending with \0

```
char name[] = "AIML";
```

Memory:

A I M L \0

Inputting Strings

```
char name[20];
scanf("%s", name);
```

Better method:

```
fgets(name, 20, stdin);
```

Character Library Functions

Include:

```
#include <ctype.h>
```

Examples:

```
toupper('a');  // 'A'
isdigit('5');  // 1
```

String Handling Functions

Include:

#include <string.h>

Function	Work
strlen()	Length
strcpy()	Copy
strcat()	Concatenate
strcmp()	Compare

Example:-

```
#include <stdio.h>
#include <string.h>

int main() {
    char a[20] = "AI";
    char b[] = "ML";
     strcat(a, b);
    printf("%s", a);   // AIML
}
```

Pointer Arithmetic in C

Pointer arithmetic means performing operations on pointers to navigate memory.

Key Concept

When we do:

- p++ → pointer moves to next element (not next byte)
- Movement depends on data type size

Pointer Arithmatic in C

Key Concept

- p++ → pointer moves to next element (not next byte)
- Movement depends on data type size

Formula:

- New Address = Current Address + (n × size of data type)

Pointer Arithmatic in C

Increment (p++)

Explanation:

Moves pointer to next memory location of same type.

Example:

```
int a=10;
int*p=&a;
printf("%p\n", p);
p++;
printf("%p\n", p);
```

```
rt@rt:~/C_Prog_Pointer$ nano p1.c
rt@rt:~/C_Prog_Pointer$ gcc p1.c -o p1
rt@rt:~/C_Prog_Pointer$ ./p1
0x7ffc5ae646fc
0x7ffc5ae64700
```

Hexadecimal	Decimal
0x7ffc5ae646fc	140721833527036
0x7ffc5ae64700	40721833527040

p++ → pointer moves to next element (not next byte)

Pointer Arithmatic in C

Increment (p++)

Explanation:

Moves pointer to next memory location of same type.

Example:

```
int a=10;
int*p=&a;
printf("%p\n", p);
p++;
printf("%p\n", p);
```

Standard sizes of `int` on common architectures

Architecture / Machine Type	Size of `int`
16-bit systems (old DOS, Turbo C)	2 bytes
32-bit systems (x86)	4 bytes
64-bit systems (Linux, Windows, macOS)	4 bytes

```
rt@rt:~/C_Prog_Pointer$ nano p1.c
rt@rt:~/C_Prog_Pointer$ gcc p1.c -o p1
rt@rt:~/C_Prog_Pointer$ ./p1
0x7ffc5ae646fc
0x7ffc5ae64700
```

Hexadecimal	Decimal
0x7ffc5ae646fc	140721833527036
0x7ffc5ae64700	40721833527040

p++ → pointer moves to next element (not next byte)

Pointer Arithmatic in C

Memory Understanding:

If p = 1000 and int = 4 bytes

After p++:

p = 1000 + 4 = 1004

Pointer Arithmatic in C

Decrement (p--)

Explanation: Moves pointer to previous memory location.

Example:

```
int arr[3] = {10, 20, 30};

int *p = &arr[2];

p--;  //Moves pointer to previous memory location.

printf("%d", *p); // value before 30 I.e 20
```

Pointer Arithmatic in C

Decrement (p--)

Explanation: Moves pointer to previous memory location.

Example:

```
int arr[3] = {10, 20, 30};

int *p = &arr[2];

p--; // Moves pointer to previous memory location.

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

Output will be 20

Addition (p + n)

Explanation:

Moves pointer forward by n elements.

Example:

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

int *p = arr;

printf("%d", *(p + 3));
```

Output will be 4

Subtraction (p - n)

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

int *p = &arr[4];

printf("%d", *(p - 2));
```

Subtraction (p - n)

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

int *p = &arr[4];

printf("%d", *(p - 2));
```

Output: 3

Pointer Difference (p1 - p2)

```
#include <stdio.h>
int main()
{
int arr[5] = {1,2,3,4,5};
int *p1 = &arr[4];
int *p2 = &arr[1];
printf("%ld", p1 - p2);
return 0;
}
```

Pointer Difference (p1 - p2)

```
#include <stdio.h>

int main()

{

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

int *p1 = &arr[4];

int *p2 = &arr[1];

printf("%ld", p1 - p2);

return 0;

}
```

Output

3

Explanation

arr[4] - arr[1]
= 4th index - 1st index
= 4 - 1
= 3 elements
Pointer subtraction gives the number of elements between two pointers, not bytes.

Pointer Difference (p1 - p2)

Memory Illustration

Index	Value	Address Example
Arr[0]	1	100
Arr[1]	2	104
Arr[2]	3	108
Arr[3]	4	112
Arr[4]	5	116

p1 = &arr[4] → 116
p2 = &arr[1] → 104

(116 - 104) / sizeof(int)
= 12 / 4
= 3

Pointer Incremental Effect

Data Type	Size	Increment Effect
char	1 byte	+1
int	4 bytes	+4
float	4 bytes	+4
double	8 bytes	+8

Pointer Arithmatic

```
char *c;

int *i;

c++;

i++;
```

Pointer Arithmatic

```
char *c;
int *i;
c++;  // +1 byte
i++;  // +4 bytes
```

Memory Visualization

Array Example:

```
int arr[3] = {10,20,30};
```

Address	Value
1000	10
??	20
??	30

Memory Visualization

Array Example:

int arr[3] = {10,20,30};

Address	Value
1000	10
??	20
??	30

If p = arr

p → 1000

p+1 → 1004

P+2 → 1008

GATE-Level Questions

Question 1

```
int arr[] = {10, 20, 30, 40};
int *p = arr;
printf("%d", *(p + 2));
```

GATE-Level Questions

Question 1

```
int arr[] = {10, 20, 30, 40};
int *p = arr;
printf("%d", *(p + 2));
```

Output is 30

GATE-Level Questions

Question 2

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

int *p = arr + 3;

printf("%d", *(p - 2));
```

GATE-Level Questions

Question 2

```
int arr[] = {1,2,3,4,5};
int *p = arr + 3;
printf("%d", *(p - 2));
```

Output is 2

GATE-Level Questions

Question 3

```
#include <stdio.h>
int main() {
    int arr[10];
    int *p = &arr[5];
    int *q = &arr[2];
    printf("%d", p - q);
    return 0;
}
```

GATE-Level Questions

Explanation

Pointer subtraction gives the number of elements between two pointers, not the number of bytes.

p = &arr[5]
q = &arr[2]

p - q = 5 - 2 = 3
So the result printed is:
3

Index	Address Example
Arr[0]	100
Arr[1]	104
Arr[2]	108
Arr[3]	112
Arr[4]	116
Arr[5]	120

GATE-Level Questions

Explanation

Pointer subtraction gives the number of elements between two pointers, not the number of bytes.

p = &arr[5]
q = &arr[2]

p - q = 5 - 2 = 3
So the result printed is:
3

p = 120

q = 108

(120 − 108) / 4 = 12 / 4 = 3

GATE-Level Questions

Question 4

```
int arr[] = {5,10,15};

int *p = arr;

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

GATE-Level Questions

Question 4

```
int arr[] = {5,10,15};
int *p = arr;
printf("%d %d", *p, *p++);
```

Output is 5 5

Explanation:

*p → 5

*p++ → first use value, then increment pointer

GATE-Level Questions

Question 5

If int *p = 1000, what will be value of p + 3?

GATE-Level Questions

Question 5

If int *p = 1000, what will be value of p + 3?

Answer:

$1000 + (3 \times 4) = 1012$

Increment (p++)

Explanation:

Moves pointer to next memory location of same type.

GATE EXAM QUESTIONS

Question 6

```
int arr[] = {10, 20, 30, 40};

int *p = arr;

printf("%d", *(p++ + 2));
```

GATE EXAM QUESTIONS

Question 6

```
int arr[] = {10, 20, 30, 40};
int *p = arr;
printf("%d", *(p++ + 2));
```

Answer:

30

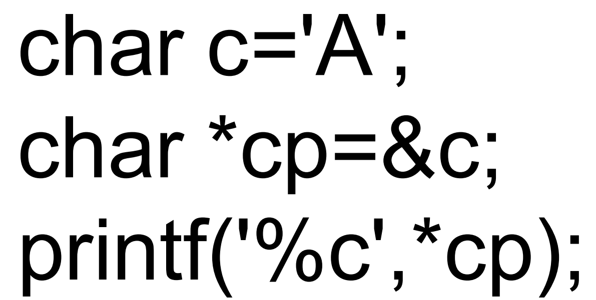
Explanation:

p++ used after evaluation

(p + 2) → points to 30

Pointers

```
float x=3.14;
float *fp=&x;
printf('%f',*fp);
```



char c='A';
char *cp=&c;
printf('%c',*cp);

pointer to pointer

Pointer storing address of another pointer

Syntax: int **pp

```
int a=5;
int *p=&a;
int **pp=&p;
printf('%d',**pp);
```

pointer to pointer

Pointer storing address of another pointer
Syntax: int **pp

```
#include <stdio.h>
int main()
{
int a = 5;        // integer variable
int *p = &a;      // pointer p stores address of a
int **pp = &p;    // pointer to pointer storing address of p
printf("%d", **pp);  //
return 0;
}
```

pointer to pointer

Pointer storing address of another pointer

Syntax: int **pp

```
#include <stdio.h>
int main()
{
int a = 5;        // integer variable
int *p = &a;      // pointer p stores address of a
int **pp = &p;    // pointer to pointer storing address of p
printf("%d", **pp);  // **pp → value of a
return 0;
}
```

Generic Pointer (void pointer)

void * can store address of any data type
Must be typecast before dereferencing

A void pointer (void *) is a generic pointer that can store the address of any data type (int, float, char, etc.).

However, the compiler does not know what type of data is stored at that address.

Because of this, you cannot directly access the value using *ptr.

Generic Pointer (void pointer) Example

```
#include <stdio.h>
int main() {
    int a = 10;
    void *p = &a;
    printf("%d", *p);   // ERROR
}
```

Why Error?

p is a void pointer, so the compiler does not know whether the data is int, float, char, etc.

Generic Pointer (void pointer) Example

```
#include <stdio.h>
int main() {
    int a = 10;
    void *p = &a;
   printf("%d", *(int*)p);
}
```

```
#include <stdio.h>
int main() {
    int a = 10;
    void *p = &a;
    printf("%d", *p);   // ERROR
}
```

Why not Error?

(int*)p means convert void pointer to int pointer. Then dereferencing becomes possible.

Generic Pointer (void pointer) Memory Visualization

Variable a = 10

Memory Address
1000 → 10
void *p = 1000 But compiler does not know:

int (4 bytes)
float (4 bytes)
char (1 byte)
So we must tell it:
(int*)p

Generic Pointer (void pointer)NET / GATE

What will be the output?

```
int a = 5;
void *p = &a;
printf("%d", *(int*)p);
```

Answer ??

Generic Pointer (void pointer)NET / GATE

What will be the output?

```
int a = 5;
void *p = &a;
printf("%d", *(int*)p);
```

Answer 5

because the void pointer is typecast to int pointer before dereferencing.

Difference Between (*p)++ and *p++

Case 1: (*p)++

Here value pointed by pointer is incremented.

```
#include <stdio.h>
int main() {
    int a = 5;
    int *p = &a;
    (*p)++;
    printf("%d", a);
}
```

Difference Between (*p)++ and *p++

Case 1: (*p)++
Here value pointed by pointer is incremented.

```
#include <stdio.h>
int main() {
    int a = 5;
    int *p = &a;
    (*p)++;
    printf("%d", a);
}
```

Output 6

Difference Between (*p)++ and

Case 1: (*p)++
Here value pointed by pointer is incremented.

```
#include <stdio.h>
int main() {
    int a = 5;
    int *p = &a;
    (*p)++;
    printf("%d", a);
}
```

Output 6

Explanation

p → address of a

*p → value of a

Operation performed:

(*p)++ → increase value stored at address

Difference Between (*p)++ and

Case 1: (*p)++
Here value pointed by pointer is incremented.

```
#include <stdio.h>
int main() {
    int a = 5;
    int *p = &a;
    (*p)++;
    printf("%d", a);
}
```

Output 6

Explanation

p → address of a

*p → value of a

Operation performed:

(*p)++ → increase value stored at address

Memory Diagram

Address	Value
1000	5 ← a

p = 1000

After operation:

Address	Value
1000	6

Difference Between (*p)++ and *p++

Here pointer moves to next location.

Operator precedence matters:

*p++ = *(p++)

So pointer increments first.

Priority	Operator	Meaning
1	++ -- (postfix)	highest
2	* (dereference)	next
3	prefix ++ --	next

Difference Between (*p)++ and *p++

Here pointer moves to next location.
Operator precedence matters:
*p++ = *(p++) So pointer increments first

```
#include <stdio.h>
int main() {
    int arr[3] = {10,20,30};
    int *p = arr;
    printf("%d\n", *p);
    p++;
    printf("%d\n", *p);
}
```

GATE EXAM:- Difference Between (*p)++ and *p++

Here pointer moves to next location.
Operator precedence matters:
*p++ = *(p++) So pointer increments first

```
#include <stdio.h>
int main() {
    int arr[3] = {10,20,30};
    int *p = arr;
    printf("%d\n", *p);
    p++;
    printf("%d\n", *p);
}
```

Output
10
20

GATE:- Difference Between (*p)++ and *p++

Here pointer moves to next location.
Operator precedence matters:
*p++ = *(p++) So pointer increments first

```
#include <stdio.h>
int main() {
    int arr[3] = {10,20,30};
    int *p = arr;
    printf("%d\n", *p);
    p++;
    printf("%d\n", *p);
}
```

Output
10
20

Memory Representation

Address	Value
1000	10
1004	20
1008	30

Pointer movement:
p = 1000 → value 10
p++ → 1004 → value 20

GATE EXAM QUESTION

```
int a=10;
int *p=&a;
(*p)++;
printf("%d",a);
```

GATE EXAM QUESTION

```
int a=10;
int *p=&a;
(*p)++;
printf("%d",a);
```

ANS:- 11

GATE EXAM QUESTION

Difference Between int *p[5] and int (*p)[5]

Case 1: int *p[5]

This means:

p is an array of 5 pointers to int
Structure
p[0] → int address
p[1] → int address
p[2] → int address
p[3] → int address
p[4] → int address

GATE EXAM QUESTION

Case 1: int *p[5] // Array of pointers
This means:
p is an array of 5 pointers to int

Structure
p[0] → int address
p[1] → int address
p[2] → int address
p[3] → int address
p[4] → int address
Example

```
int a=10,b=20,c=30;

int *p[3] = {&a,&b,&c}; // array of pointers

printf("%d", *p[1]);
```

Output:
20
Memory Diagram
p[0] → address of a
p[1] → address of b
p[2] → address of c

GATE EXAM QUESTION

Case 1: int *p[5]
This means:
p is an array of 5 pointers to int

Structure
p[0] → int address
p[1] → int address
p[2] → int address
p[3] → int address
p[4] → int address
Example

```
int a=10,b=20,c=30;
int *p[3] = {&a,&b,&c};
printf("%d", *p[1]);
```

Output:
20
Memory Diagram
p[0] → address of a
p[1] → address of b
p[2] → address of c

GATE EXAM QUESTION

Case 2: int (*p)[5] // Pointer to array

This means:
p is a pointer to an array of 5 integers
Example

```
int arr[5] = {1,2,3,4,5};
int (*p)[5] = &arr;

printf("%d", (*p)[2]);
```

Output:
3

Function Pointer

A function pointer stores address of a function.

Syntax

```
return_type (*pointer_name) (parameters);
```

Function Pointer

A function pointer stores address of a function.

Syntax

return_type (*pointer_name) (parameters);

```
#include <stdio.h>       // Header file required for printf()

// Function definition that takes two integers and returns their sum
int add(int a, int b)
{
    return a + b;       // Returns the addition of a and b
}

int main()
{
    int (*fp)(int, int);   // Declare a function pointer named fp
                        // fp points to a function that:
                        // 1) takes two int arguments
                        // 2) returns an int

    fp = add;           // Assign the address of function 'add' to fp
                        // Function name itself represents its address
                        // So fp now points to function add()

    printf("%d", fp(3,4)); // Call the function through the pointer fp
                        // This is equivalent to calling add(3,4)
                        // add(3,4) returns 7, which is printed

    return 0;              // End of program
}
```

Function Pointer

A function pointer stores address of a function.

Syntax

return_type (*pointer_name) (parameters);

```
#include <stdio.h>
int add(int a,int b)
{
    return a+b;
}
int main()
{
    int (*fp)(int,int);
    fp = add;
    printf("%d", fp(3,4));
}
```

Example 2 (Another Style)

```
printf("%d", (*fp)(3,4));
```

Both are correct

Function Pointer

```
int fun(int x)
{
return x*x;
}

int main()
{
int (*fp)(int)=fun;
printf("%d",fp(4));
}
```

God Bless you all

Thanks …