Session 13

Wednesday, 30 July 2025 9:17 AM

Pointers in C :

C KEYWORDS OR RESERVED WORDS

auto	break	case	char
const	continue	default	do
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
double	else	enum	extern
float	for	goto	if

① 1.imp — Storage classes ← auto, static, register, extern → read

② switch → case → default

③ struct, union, sizeof

const, typedef

pointers in C :-

int x = 10;

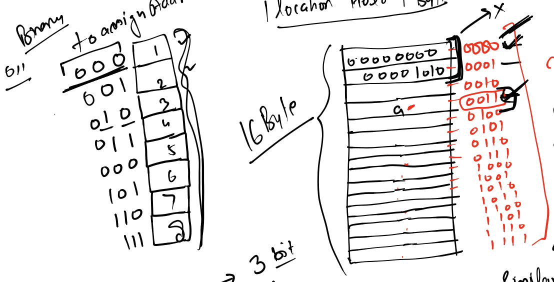
↳ int * p = &x;

float *

Why

Constant pointer → pointer to Constant → Void pointer → pointer to array → pointer to pointer

1 location Hold 1 Byte



int x = 10; 00000000 01010 — 2 Byte

int * p = &x ;

→ pointer is a Variable which hold an address. (Starting address)

Char 2 = 'a' ;

Char * C = &2 ;

• pointer can hold the address of Similar Kind of Data Type.

printf(" %d ", x); → 10

printf(" %c ", 2); → a

printf(" %d ", p); → 000

printf(" %d ", C); → 3

→ Dereference operator :-

printf(" %d ", *p); → 10

printf(" %d ", *C); → a

8 Memory → 3 bit

9—16 → 4 bit

17 — 32 → 5 bit

5 bit Address → then How many Unique address we Represent → (32)

6 → 64

64 memory Location How many bits are required to represent the of one location.

6 bit

64 → $\log_2(64)$ → (6)

$\log_2 2^6 \Rightarrow 6 \log_2 2 \Rightarrow 6$

n memory location then we $\log_2(n)$ bits to represents

Q Why we store a similar pointer to Hold the address)

Ans if we Use pointer variable to access the Value of Variable then How memory location can be accessed.

→ 2 — location

1 — location

require address uniquely

Let's delve into each type of pointer with illustrative C code examples.

1. Basic Pointer (Type-Specific Pointers)

A pointer that holds the memory address of a variable of a specific data type.

C

```
#include <stdio.h>
int main() {
    int num = 10;
    int *ptr_num; // Declares an integer pointer
ptr_num = &num; // Stores the address of 'num' in 'ptr_num'
printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", &num);
    printf("Value of ptr_num (address it holds): %p\n", ptr_num);
    printf("Value pointed to by ptr_num: %d\n", *ptr_num); // Dereferencing the pointer
*ptr_num = 20; // Modifying the value through the pointer
    printf("New value of num: %d\n", num);
char ch = 'A';
    char *ptr_ch = &ch; // Declares and initializes a character pointer
printf("\nValue of ch: %c\n", ch);
    printf("Value pointed to by ptr_ch: %c\n", *ptr_ch);
return 0;
}
```

2. Null Pointer

A pointer that points to no valid memory location. It's often used to indicate that a pointer is not currently pointing to anything, or to initialize pointers to a known "safe" state.

C

```
#include <stdio.h>
int main() {
    int *ptr_null = NULL; // Declares and initializes a null pointer
if (ptr_null == NULL) {
        printf("ptr_null is a null pointer.\n");
    } else {
        printf("ptr_null is not a null pointer.\n");
    }
// Attempting to dereference a null pointer leads to undefined behavior/segmentation fault
    // if (*ptr_null == 0) { // DO NOT uncomment this line, it will crash!
    //     printf("This would crash if uncommented.\n");
    // }
// Null pointers are useful for error checking, e.g., after malloc
    int *dynamic_array = (int *)malloc(5 * sizeof(int));
    if (dynamic_array == NULL) {
        printf("Memory allocation failed!\n");
    } else {
        printf("Memory allocated successfully.\n");
        free(dynamic_array); // Free the allocated memory
    }
return 0;
}
```

3. Void Pointer (Generic Pointer)

A pointer that can hold the address of any data type. It's "generic" because it doesn't know the type of data it's pointing to. Requires typecasting before dereferencing.

C

```
#include <stdio.h>
int main() {
    int num = 100;
    float pi = 3.14f;
    void *ptr_void; // Declares a void pointer
ptr_void = &num; // ptr_void now holds the address of an int
    printf("Value of num using void pointer: %d\n", *(int *)ptr_void); // Typecast to int* before
dereferencing
ptr_void = &pi; // ptr_void now holds the address of a float
    printf("Value of pi using void pointer: %f\n", *(float *)ptr_void); // Typecast to float* before
dereferencing
// Void pointers are commonly used with memory allocation functions like malloc
    int *arr = (int *)malloc(5 * sizeof(int)); // malloc returns void*, typecasted to int*
    if (arr != NULL) {
        arr[0] = 10;
        printf("First element of allocated array: %d\n", arr[0]);
        free(arr);
    }
return 0;
}
```

4. Dangling Pointer

A pointer that points to a memory location that has been deallocated (freed). Accessing a dangling pointer leads to undefined behavior.

C

```
#include <stdio.h>
#include <stdlib.h> // For malloc and free
int main() {
    int *ptr_dangling = (int *)malloc(sizeof(int)); // Allocate memory
    if (ptr_dangling == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
*ptr_dangling = 50;
    printf("Value before free: %d\n", *ptr_dangling);
```

```
   printf("Value before free: %d\n", *ptr_dangling);
free(ptr_dangling); // Deallocate the memory. ptr_dangling is now dangling.
// At this point, ptr_dangling points to freed memory.
   // Attempting to access *ptr_dangling is undefined behavior.
   // It might print the old value, print garbage, or crash the program.
   printf("Value after free (DANGEROUS ACCESS): %d\n", *ptr_dangling); // DO NOT RELY ON THIS
OUTPUT
ptr_dangling = NULL; // Good practice: set to NULL after freeing to avoid dangling
   printf("Pointer set to NULL after free: %p\n", (void *)ptr_dangling);
return 0;
}
```

5. Wild Pointer

An uninitialized pointer. It contains an unpredictable, garbage value and points to an arbitrary memory location. Dereferencing it is very risky.

C

```
#include <stdio.h>
int main() {
   int *ptr_wild; // Declares a pointer but doesn't initialize it. It's a wild pointer.
// Attempting to dereference a wild pointer is highly dangerous and can crash your program
   // or corrupt memory.
   // *ptr_wild = 100; // DO NOT UNCOMMENT THIS LINE IN PRODUCTION CODE
printf("Wild pointer value (uninitialized, unpredictable): %p\n", (void *)ptr_wild);
// To make it safe, initialize it:
   int x = 5;
   ptr_wild = &x; // Now it's a valid pointer to x
   printf("Initialized pointer value: %p\n", (void *)ptr_wild);
   printf("Value pointed to by initialized pointer: %d\n", *ptr_wild);
return 0;
}
```

6. Double Pointer (Pointer to Pointer)

A pointer that stores the address of another pointer.

C

```
#include <stdio.h>
int main() {
   int num = 77;
   int *ptr_num = &num;    // ptr_num stores the address of 'num'
   int **ptr_to_ptr = &ptr_num; // ptr_to_ptr stores the address of 'ptr_num'
printf("Value of num: %d\n", num);
   printf("Address of num: %p\n", &num);
   printf("Value of ptr_num (address of num): %p\n", ptr_num);
   printf("Address of ptr_num: %p\n", &ptr_num);
   printf("Value of ptr_to_ptr (address of ptr_num): %p\n", ptr_to_ptr);
   printf("Address of ptr_to_ptr: %p\n", &ptr_to_ptr);
// Dereferencing a double pointer:
   printf("\nValue pointed to by ptr_num: %d\n", *ptr_num); // *ptr_num gets the value of num
   printf("Value pointed to by ptr_to_ptr (indirectly num): %d\n", **ptr_to_ptr); // **ptr_to_ptr gets the
value of num
// Modifying num through double pointer
   **ptr_to_ptr = 88;
   printf("New value of num: %d\n", num);
return 0;
}
```

7. Function Pointer

A pointer that stores the memory address of a function. This allows you to call functions indirectly.

C

```
#include <stdio.h>
// A simple function
int add(int a, int b) {
   return a + b;
}
// Another function
int subtract(int a, int b) {
   return a - b;
}
int main() {
   // Declare a function pointer that points to a function
   // taking two ints and returning an int.
   int (*op_ptr)(int, int);
// Assign the address of the 'add' function to the function pointer
   op_ptr = &add; // '&' is optional but good practice for clarity
// Call the 'add' function using the function pointer
   int result_add = op_ptr(10, 5);
   printf("Result of addition: %d\n", result_add);
// Assign the address of the 'subtract' function
   op_ptr = subtract; // 'subtract' itself decays to its address
// Call the 'subtract' function using the same function pointer
   int result_subtract = op_ptr(10, 5);
   printf("Result of subtraction: %d\n", result_subtract);
// Function pointers in an array (for a simple menu system, for example)
   int (*operations[2])(int, int);
   operations[0] = add;
   operations[1] = subtract;
printf("Result from array [0] (add): %d\n", operations[0](20, 10));
   printf("Result from array [1] (subtract): %d\n", operations[1](20, 10));
return 0;
}
```

8. Array Pointer

A pointer that points to an entire array. Its type explicitly includes the size of the array it points to. This is different from a pointer to the first element of an array.

C

```
#include <stdio.h>
int main() {
   int arr[5] = {10, 20, 30, 40, 50};
// Pointer to the first element of the array (most common)
   int *ptr_first_element = arr; // 'arr' decays to a pointer to its first element
   printf("Using pointer to first element: %d\n", *ptr_first_element);
   printf("Sizeof ptr_first_element: %zu bytes\n", sizeof(ptr_first_element)); // Size of the pointer itself
(e.g., 8 bytes on 64-bit)
// Pointer to the entire array
   // The syntax: 'int (*ptr_array)[5]' means ptr_array is a pointer to an array of 5 integers.
   int (*ptr_array)[5] = &arr; // &arr points to the whole array
printf("\nUsing pointer to the entire array:\n");
   printf("Value of the first element using ptr_array: %d\n", (*ptr_array)[0]);
   printf("Value of the third element using ptr_array: %d\n", (*ptr_array)[2]);
printf("Sizeof ptr_array: %zu bytes\n", sizeof(ptr_array)); // Size of the pointer itself
```

```
// You can also iterate through the array using ptr_array
    printf("Iterating through array using ptr_array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", (*ptr_array)[i]);
    }
    printf("\n");
return 0;
}
```

H/W

Const → Keyword

~~10~~ 20
x

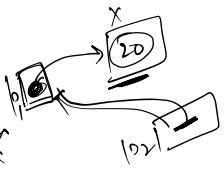
int x = 10;
print (x) → 10 ;
x=20;
print (x) → 20;

→ I want that the value will Not be Changed in future;

Pointer to pointer

```
int x = 20;
int *p1 = &x;
int **p2 = &p1;
```

printf(x); → 20
printf(p1); → address of x
printf(p2); → address of p1
printf(*p1); → 20
printf(*p2) → address of x
printf(**p2); → 20



Const → Keyword Comes into picture;

int const x = 10;