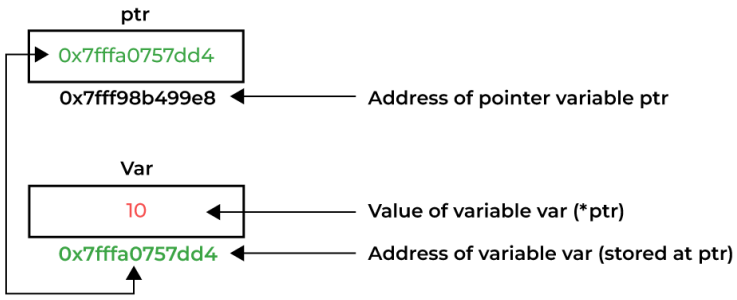
Pointers

By Abdallah Ghazy

❖ What is a Pointer in C?

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

As the pointers in C store the memory addresses, their size is independent of the type of data they are pointing to. This size of pointers in C only depends on the system architecture.



❖ Pointer Declaration

To declare a pointer, use the dereferencing operator (*) followed by the data type.

Syntax

The general form of a pointer variable declaration is –

```
type *var-name;
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents **a memory address**.

Pointer Initialization

After declaring a pointer variable, you need to initialize it with the address of another variable using the address of (&) operator. This process is known as referencing a pointer.

Syntax

The following is the syntax to initialize a pointer variable –

```
pointer_variable = &variable;
```

Example

Here is an example of pointer initialization –

```
int x = 10;
int *ptr = &x;
```

❖ Referencing and Dereferencing Pointers

A pointer references a location in memory. Obtaining the value stored at that location is known as dereferencing the pointer.

- **The & Operator** – It is also known as the "Address-of operator". It is used for Referencing which means taking the address of an existing variable (using **&**) to set a pointer variable.
- **The * Operator** – It is also known as the "dereference operator". **Dereferencing** a pointer is carried out using the *** operator** to get the value from the memory address that is pointed by the pointer.

❖ Why We Need Pointers in C

- Dynamic Memory Allocation:

○ Allocate memory at runtime for creating flexible data structures.

1. Dynamic Memory Allocation

c نسخ الكود

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

int main() {
    int *arr;
    int size = 5;

    // Allocate memory for an array of integers
    arr = (int *)malloc(size * sizeof(int));

    // Use the allocated memory
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
        printf("%d ", arr[i]);
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

- Parameter Passing:

- Pass parameters by reference to functions for efficiency, especially with large data structures.

2. Parameter Passing

c نسخ الكود

```c
#include <stdio.h>

void increment(int *value) {
    (*value)++;
}

int main() {
    int num = 10;

    // Pass the address of num to the function
    increment(&num);

    printf("Incremented value: %d\n", num); // Output: 11

    return 0;
}
```

- Manipulating Hardware:

- Access and control hardware directly through memory-mapped registers.

3. Manipulating Hardware

c نسخ الكود

```c
#include <stdint.h>

#define GPIO_BASE_ADDRESS 0x40020000
#define GPIO_ODR_OFFSET   0x14

volatile uint32_t *GPIO_ODR = (uint32_t *)(GPIO_BASE_ADDRESS + GPIO_ODR_OFFSET);

int main() {
    // Set a specific bit in the GPIO output data register
    *GPIO_ODR |= (1 << 5); // Example: Set bit 5

    return 0;
}
```

- Arrays and Strings:

- Implement and efficiently manipulate arrays and strings using pointers.

4. Arrays and Strings

c نسخ الكود

```c
#include <stdio.h>

int main() {
    char str[] = "Hello, World!";
    char *ptr = str;

    // Print the string using a pointer
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }

    printf("\n");

    return 0;
}
```

- Peripheral Register Access:

- Configure and interact with peripheral register addresses directly.

```c
#include <stdint.h>

#define PERIPHERAL_BASE_ADDRESS 0x40021000
#define PERIPHERAL_REG_OFFSET   0x10

volatile uint32_t *PERIPHERAL_REG = (uint32_t *)(PERIPHERAL_BASE_ADDRESS + PERIPHERAL_REG_OFFSET);

int main() {
    // Write a value to the peripheral register
    *PERIPHERAL_REG = 0xA5A5A5A5;

    return 0;
}
```

- Read/Write Operations:

- Perform read/write operations in SRAM, FLASH, and other memory locations.

```c
#include <stdint.h>

#define SRAM_BASE_ADDRESS  0x20000000
#define FLASH_BASE_ADDRESS 0x08000000

volatile uint32_t *sram_ptr = (uint32_t *)SRAM_BASE_ADDRESS;
volatile uint32_t *flash_ptr = (uint32_t *)FLASH_BASE_ADDRESS;

int main() {
    // Write a value to SRAM
    *sram_ptr = 0x12345678;

    // Read a value from FLASH
    uint32_t flash_value = *flash_ptr;

    return 0;
}
```

- Data Structures:

- Essential for creating linked lists, trees, graphs, and other complex structures.

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

// Define a node structure for a linked list
typedef struct Node {
    int data;
    struct Node *next;
} Node;

int main() {
    // Create the first node
    Node *head = (Node *)malloc(sizeof(Node));
    head->data = 1;
    head->next = NULL;

    // Create the second node
    Node *second = (Node *)malloc(sizeof(Node));
    second->data = 2;
    second->next = NULL;

    // Link the first node to the second node
    head->next = second;

    // Print the linked list
    Node *current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");

    // Free the allocated memory
    free(second);
    free(head);

    return 0;
}
```

- Memory Efficiency:

  - Enable creation of memory-efficient data structures compared to using arrays.

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

// Define a structure for a tree node
typedef struct TreeNode {
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

TreeNode* createNode(int data) {
    TreeNode *node = (TreeNode *)malloc(sizeof(TreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

int main() {
    // Create tree nodes
    TreeNode *root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);

    // Print the root node data
    printf("Root node data: %d\n", root->data);

    // Free the allocated memory
    free(root->left);
    free(root->right);
    free(root);

    return 0;
}
```

Conclusion

Pointers enhance the power and flexibility of the C language, making it suitable for low-level programming and efficient memory management.

Access and Manipulate Values using Pointer

The value of the variable which is pointed by a pointer can be accessed and manipulated by using the pointer variable. You need to use the asterisk (*) sign with the pointer variable to access and manipulate the variable's value.

text.c

```c
#include <stdio.h>

int main() {
   int x = 10;

   // Pointer declaration and initialization
   int * ptr = & x;

   // Printing the current value
   printf("Value of x = %d\n", * ptr);

   // Changing the value
   * ptr = 20;

   // Printing the updated value
   printf("Value of x = %d\n", * ptr);

   return 0;
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Applicatior
Value of x = 10
Value of x = 20
```

Addition and Subtraction of Variables Using Pointers in C

text.c

```c
#include <stdio.h>

int main() {
    // Define variables
    int x = 10;
    int y = 5;
    int sum, difference;

    // Define pointers and assign the addresses of the variables to them
    int *ptrX = &x;
    int *ptrY = &y;

    // Add values using pointers
    sum = *ptrX + *ptrY;
    printf("Addition result: %d + %d = %d\n", *ptrX, *ptrY, sum);

    // Subtract values using pointers
    difference = *ptrX - *ptrY;
    printf("Subtraction result: %d - %d = %d\n", *ptrX, *ptrY, difference);

    return 0;
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Deskt
Addition result: 10 + 5 = 15
Subtraction result: 10 - 5 = 5
```

Size of a Pointer Variable

The memory (or, size) occupied by a pointer variable does not depend on the type of the variable it is pointing to. The size of a pointer depends on the system architecture.

- 64-bit Machine:

○ Pointer size: 8 bytes (64 bits)

- 32-bit Machine:

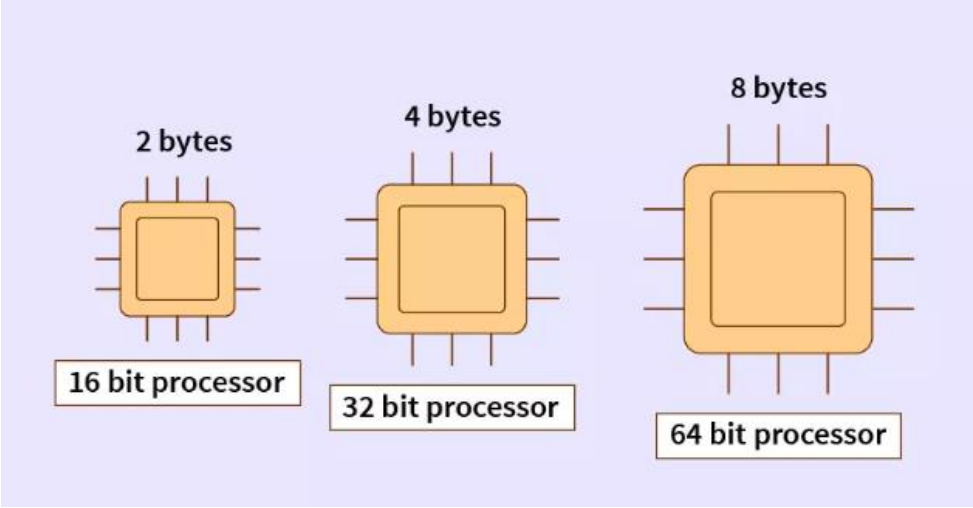
○ Pointer size: 4 bytes (32 bits)

- Uniformity Across Data Types:

○ The size of a pointer is the same regardless of the data type it points to (e.g., int *, float *, char *, etc.).

```c
#include <stdio.h>

int main() {
  int x = 10;
  float y = 1.3f;
  char z = 'p';

  // Pointer declaration and initialization
  int * ptr_x = & x;
  float * ptr_y = & y;
  char * ptr_z = & z;

  // Printing the size of pointer variables
  printf("Size of integer pointer : %lu\n", sizeof(ptr_x));
  printf("Size of float pointer : %lu\n", sizeof(ptr_y));
  printf("Size of char pointer : %lu\n", sizeof(ptr_z));

  return 0;
}
```

```
Size of integer pointer : 4
Size of float pointer : 4
Size of char pointer : 4
```



Example

```c
#include <stdio.h>

int main() {
    int x = 3;
    int *px = &x;
    printf("sizeof(x): %d ,sizeof(px): %d",sizeof(x),sizeof(px));

    return 0;
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdal
sizeof(x): 4 ,sizeof(px): 4
```

Example

```c
#include <stdio.h>

int main() {
    int x = 345678;
    short *px = &x;
    printf("sizeof(x): %d ,sizeof(px): %d , x = %d\n",sizeof(x),sizeof(px),x);
    *px = 2;
    printf("sizeof(x): %d ,sizeof(px): %d , x = %d",sizeof(x),sizeof(px),x);

    return 0;
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\D
sizeof(x): 4 ,sizeof(px): 4 , x = 345678
sizeof(x): 4 ,sizeof(px): 4 , x = 327682
```

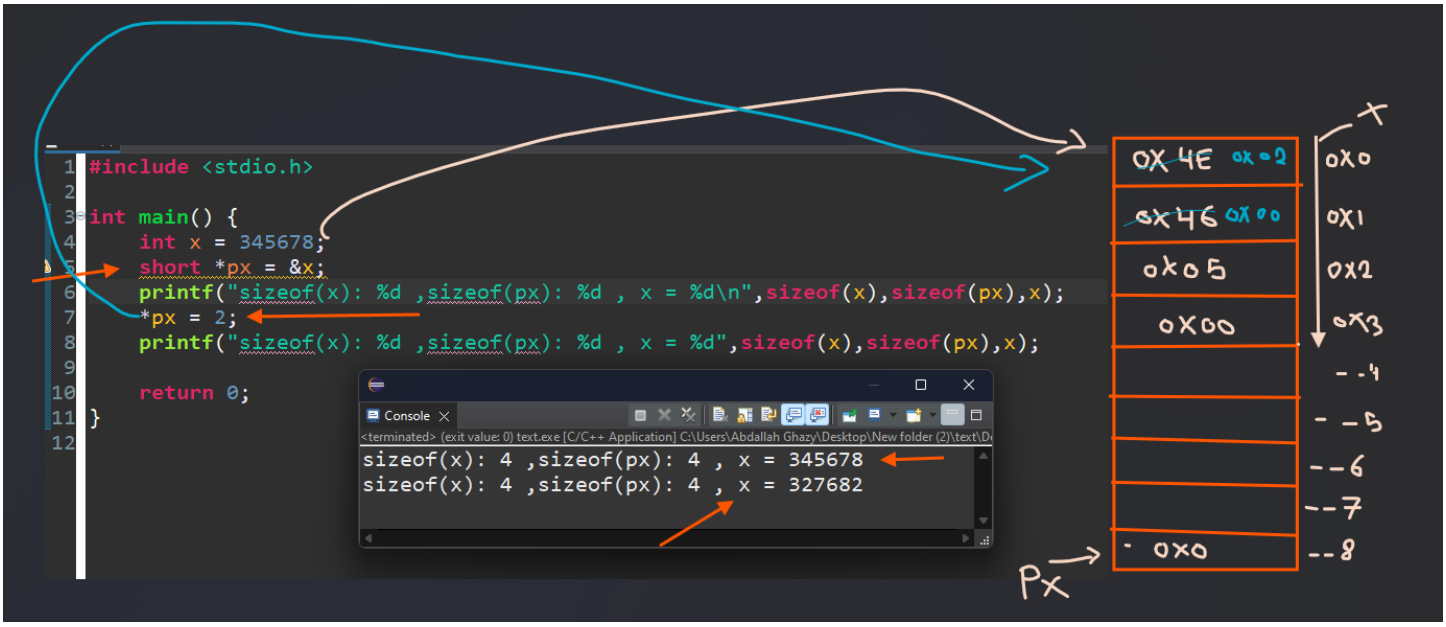
Example

text.c

```c
#include <stdio.h>

int main() {
    char *px;
    int *py;

    if (sizeof(px) == sizeof(py)){
        printf("sizeof(px) == sizeof(py) ");
    }
    else
        printf("sizeof(px) != sizeof(py)");

    return 0;
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desl
sizeof(px) == sizeof(py)
```

Example



&(x) : 0x61FF18 <Hex Integer> | New Renderings...

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0061FEB0	BC	FE	61	00	75	87	96	76	1C	93	9B	76	FC	FE	61	00
0061FEC0	3D	7C	96	76	08	00	00	00	7D	81	95	76	63	81	95	76
0061FED0	C8	AC	EC	EC	E0	12	40	00	E0	12	40	00	00	00	00	00
0061FEE0	90	19	40	00	D0	FE	61	00	08	FF	61	00	CC	FF	61	00
0061FEF0	70	E1	95	76	4C	8D	17	9A	FE	FF	FF	FF	6E	14	40	00
0061FF00	6D	82	95	76	90	19	40	00	54	FF	61	00	EB	19	40	00
0061FF10	90	19	40	00	08	16	7A	00	4E	46	05	00	00	B0	34	00
0061FF20	80	00	40	00	10	FF	61	00	54	FF	61	00	88	12	40	00
0061FF30	01	00	00	00	08	16	7A	00	80	1D	7A	00	FD	FF	FF	FF
0061FF40	02	00	00	00	00	00	00	00	54	FF	61	00	CD	88	95	76
0061FF50	00	B0	34	00	84	FF	61	00	F5	12	40	00	01	00	00	00
0061FF60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0061FF70	00	00	00	00	00	00	00	00	A9	7B	1B	76	00	B0	34	00
0061FF80	90	7B	1B	76	DC	FF	61	00	0B	C1	AF	77	00	B0	34	00
0061FF90	7B	1B	E3	B9	00	00	00	00	00	00	00	00	00	B0	34	00
0061FFA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0061FFB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0061FFC0	00	00	00	00	90	FF	61	00	00	00	00	00	E4	FF	61	00

&(x) : 0x61FF18 <Hex Integer> | New Renderings...

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0061FEB0	57	A2	96	76	11	00	00	00	F8	FE	61	00	48	7C	97	76
0061FEC0	01	00	00	00	20	96	9B	76	26	7C	97	76	CC	AC	EC	EC
0061FED0	E0	12	40	00	E0	12	40	00	00	B0	34	00	29	00	00	00
0061FEE0	CC	FE	61	00	D0	FE	61	00	CC	FF	61	00	70	E1	95	76
0061FEF0	24	B7	17	9A	FE	FF	FF	FF	28	FF	61	00	A2	14	40	00
0061FF00	64	50	40	00	04	00	00	00	04	00	00	00	4E	46	05	00
0061FF10	90	19	40	00	08	16	7A	00	02	00	05	00	18	FF	61	00
0061FF20	80	00	40	00	10	FF	61	00	54	FF	61	00	88	12	40	00
0061FF30	01	00	00	00	08	16	7A	00	80	1D	7A	00	FD	FF	FF	FF
0061FF40	02	00	00	00	00	00	00	00	54	FF	61	00	CD	88	95	76
0061FF50	00	B0	34	00	84	FF	61	00	F5	12	40	00	01	00	00	00
0061FF60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0061FF70	00	00	00	00	00	00	00	00	A9	7B	1B	76	00	B0	34	00
0061FF80	90	7B	1B	76	DC	FF	61	00	0B	C1	AF	77	00	B0	34	00
0061FF90	7B	1B	E3	B9	00	00	00	00	00	00	00	00	00	B0	34	00
0061FFA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0061FFB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0061FFC0	00	00	00	00	90	FF	61	00	00	00	00	00	E4	FF	61	00

Pointer Casting in C

Pointer casting is the process of converting a pointer of one data type to another. This can be useful in situations where you need to treat a block of memory as a different data type or interface with external libraries or hardware that use different data types.

Example 1 of Pointer Casting

Here is an example of pointer casting in C:

text.c

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr = &x;
    char *cptr = (char*)ptr;

    printf("The value of x is %d\n", x);
    printf("The value of ptr is %p\n", ptr);
    printf("The value of cptr is %p\n", cptr);

    *cptr = 20;

    printf("The value of x is now %d\n", x);
    printf("The value stored at the memory address pointed to by ptr is now %d\n", *ptr);

    return 0;
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\Debug\text.exe (7/20/24, 1:20 PM)
The value of x is 10
The value of ptr is 0061FF14
The value of cptr is 0061FF14
The value of x is now 20
The value stored at the memory address pointed to by ptr is now 20
```

Example 2 of Pointer Casting

*text.c

```c
#include <stdio.h>

int main() {
    // Example 1: Casting from int* to char*

    int num = 65;
    int *intPtr = &num;         // Pointer to int
    char *charPtr = (char*) intPtr; // Cast int* to char*

    printf("Value pointed to by intPtr: %d\n", *intPtr);
    printf("Value pointed to by charPtr: %c\n", *charPtr); // Interprets the same memory as char

    // Example 2: Casting from void* to specific pointer type

    void *voidPtr;              // Void pointer can point to any data type
    int value = 123;
    voidPtr = &value;           // Assign address of int to void pointer

    int *intPtrFromVoid = (int*) voidPtr; // Cast void* to int*

    printf("Value pointed to by intPtrFromVoid: %d\n", *intPtrFromVoid);

    return 0;
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Deskt
Value pointed to by intPtr: 65
Value pointed to by charPtr: A
Value pointed to by intPtrFromVoid: 123
```

Important Considerations

Alignment and Undefined Behavior: Pointer casting can lead to undefined behavior if memory alignment is not respected. For example, casting an integer pointer to a character pointer and modifying the value can result in partial updates if the sizes of the data types are different.

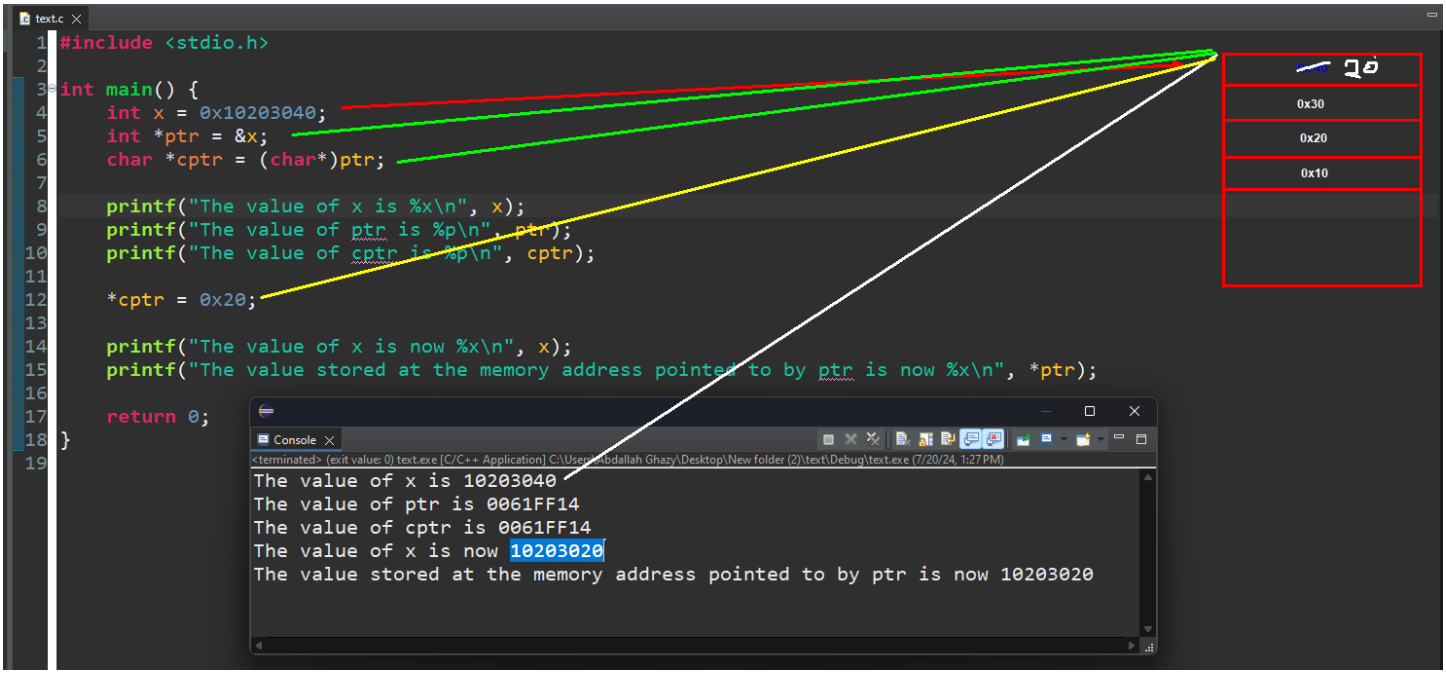
Bounds Checking: Ensure that pointer casting does not lead to access beyond the allocated memory bounds, which can cause memory corruption or program crashes.

Use Cases: Pointer casting is often used in low-level programming, such as interfacing with hardware or optimizing memory usage in system programming.

Conclusion

Pointer casting is a powerful tool in C that allows for flexible memory manipulation and interfacing with various data types. However, it should be used with caution to avoid undefined behavior and ensure proper memory alignment and access within bounds.

Example

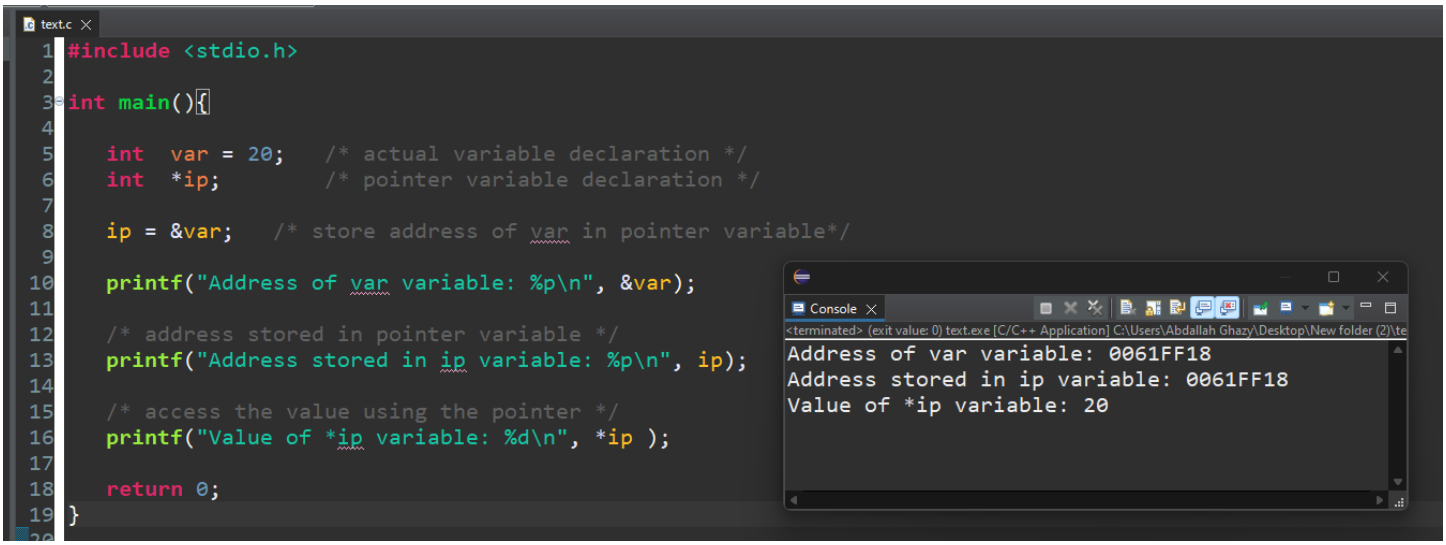


Example

Examples of C Pointers

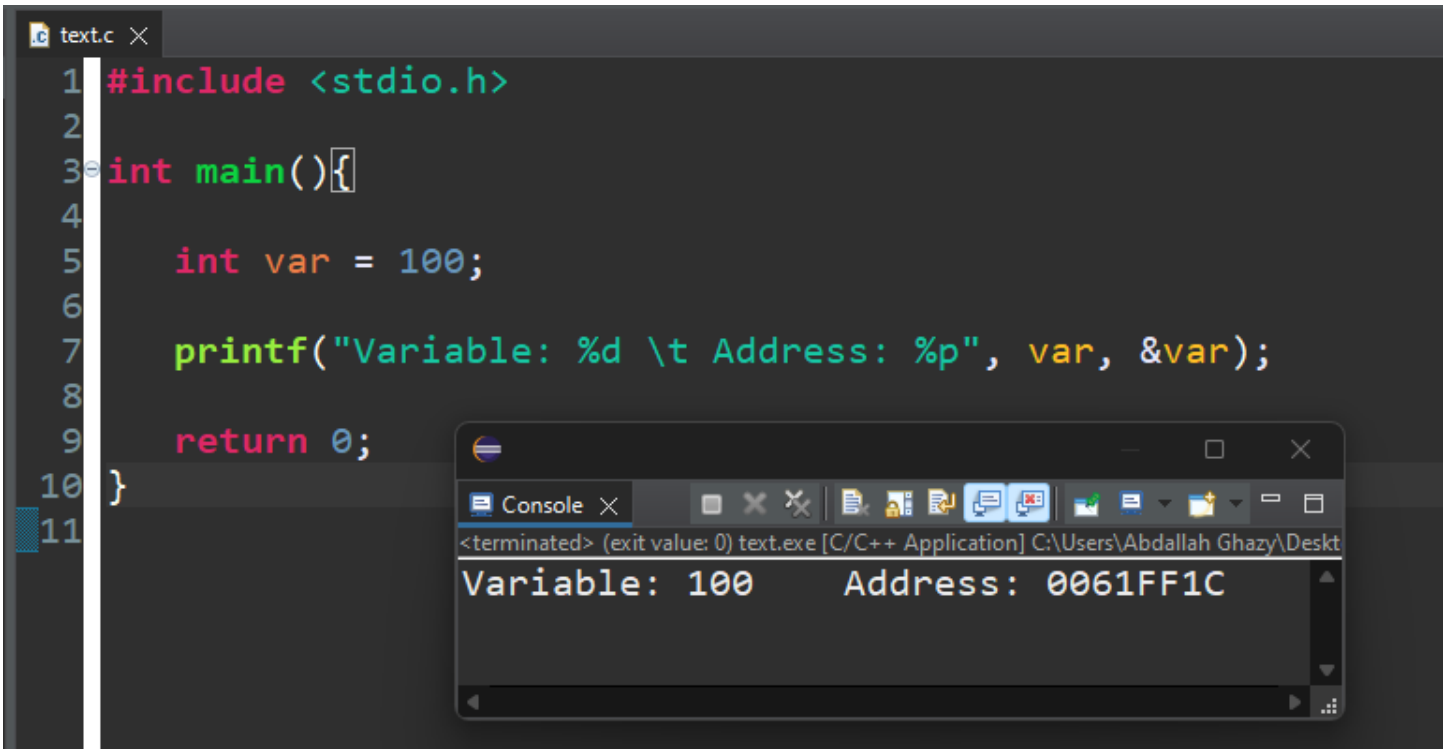
Example 1: Using Pointers in C

The following example shows how you can use the & and * operators to carry out pointer-related opeartions in C –



Example: Print Value and Address of an Integer

We will declare an int variable and display its value and address –



Example: Integer Pointer

In this example, the address of var is stored in the intptr variable with & operator

text.c

```c
#include <stdio.h>

int main(){

   int var = 100;
   int *intptr = &var;

   printf("Variable: %d \nAddress of Variable: %p \n\n", var, &var);
   printf("intptr: %p \nAddress of intptr: %p \n\n", intptr, &intptr);

   return 0;
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New fol
Variable: 100
Address of Variable: 0061FF1C

intptr: 0061FF1C
Address of intptr: 0061FF18
```

Example

text.c

```c
#include <stdio.h>

int main() {
    // Define a float variable
    float floatValue = 123.456;

    // Define a float pointer and assign it the address of floatValue
    float *floatPtr = &floatValue;

    // Print the value of floatValue using floatPtr
    printf("Value of floatValue: %.3f\n", *floatPtr);

    // Cast the float pointer to another type, such as int*
    int *intPtr = (int *)floatPtr;

    // Print the value pointed to by intPtr
    // Note: This will print an unexpected value because intPtr reads the memory as int, not as float
    printf("Value interpreted as int: %d\n", *intPtr);

    return 0;
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Deskt
Value of floatValue: 123.456
Value interpreted as int: 1123477881
```

Example

Now let's take an example of a float variable and find its address –

We can see that the address of this variable (any type of variable for that matter) is an integer. So, if we try to store it in a pointer variable of "float" type, see what happens –

```
float var1 = 10.55;
int *intptr = &var1;
```

The compiler doesn't accept this, and reports the following **error** –

```
initialization of 'int *' from incompatible pointer type
'float *' [-Wincompatible-pointer-types]
```

Note: The type of a variable and the type of its pointer must be same.

Pointer Arithmetic

C pointers arithmetic operations are different from the general arithmetic operations.

The following are some of the important pointer arithmetic operations in C:

- Increment and Decrement of a Pointer
- Addition and Subtraction of Integer to Pointer
- Subtraction of Pointers
- Comparison of Pointers

Pointer Arithmetic in C

Pointer arithmetic is the process of manipulating pointers to access data in memory. This technique allows you to move pointers to different memory locations based on the size of the data type being pointed to. It is particularly useful for iterating over arrays, creating data structures, and performing other memory operations.

Rules Governing Pointer Arithmetic

- Base Type Alignment:

- Pointer arithmetic is always performed relative to the base type of the pointer. The arithmetic takes into account the size of the data type being pointed to.

- Incrementing a Pointer:

- When a pointer is incremented (e.g., ptr++), it points to the next element of its base type.
- The address incremented by the pointer is **ptr + 1 * sizeof(base_type).**

- Decrementing a Pointer:

- When a pointer is decremented (e.g., ptr--), it points to the previous element of its base type.
- The address decremented by the pointer is **ptr - 1 * sizeof(base_type).**

- Adding/Subtracting an Integer:

- When an integer is added to or subtracted from a pointer, the resulting pointer is moved by that integer times the size of the base type.
- For example, ptr + n results in ptr pointing to **ptr + n * sizeof(base_type).**

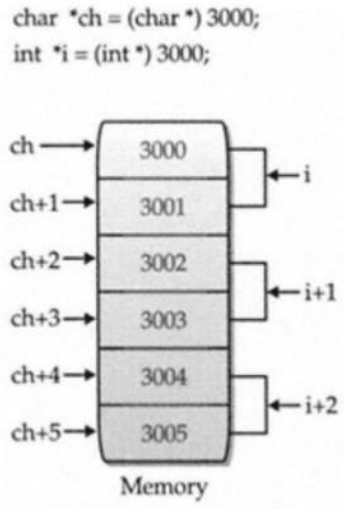
```c
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("Array elements using pointer arithmetic:\n");

    // Iterate over the array using pointer arithmetic
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));
    }

    // Increment pointer to point to the next element
    ptr++;
    printf("Pointer now points to: %d\n", *ptr); // Output: 20

    // Decrement pointer to point to the previous element
    ptr--;
    printf("Pointer now points to: %d\n", *ptr); // Output: 10

    // Add an integer to a pointer
    int *new_ptr = ptr + 3;
    printf("Pointer after adding 3: %d\n", *new_ptr); // Output: 40

    // Subtract an integer from a pointer
    new_ptr = ptr + 4;
    printf("Pointer after subtracting 4: %d\n", *(new_ptr - 4)); // Output: 10

    return 0;
}
```

```
Element 4: 50
Pointer now points to: 20
Pointer now points to: 10
Pointer after adding 3: 40
Pointer after subtracting 4: 10
```

Note The compiler reserve 8 bytes (for 64 bits machine) of memory for the definition.

It is worth to remark that the compiler will always reserve 8 bytes independently from the pointer datatype (which could be char*, int* etc etc).

- In other words, the pointer datatype doesn't control the memory size of the pointer variable.
- The pointer datatype identifies the behavior of the operations carried out on the pointer variable. (Read/Write)



Pointer Size is depend only on the machine

&PX

4 bytes in 32 machine or 8 bytes in 64 machine

Increment and Decrement of a Pointer

We know that "++" and "--" are used as the increment and decrement operators in C. They are unary operators, used in prefix or postfix manner with numeric variable operands, and they increment or decrement the value of the variable by one.

Assume that an integer variable "x" is created at address 1000 in the memory, with 10 as its value. Then, "x++" makes the value of "x" as 11.

```
int x = 10;    // created at address 1000

x++;           // x becomes 11
```

What happens if we declare "y" as pointer to "x" and increment "y" by 1 (with "y++")? Assume that the address of "y" itself is 2000.

```
int x = 10;    // created at address 1000

// "y" is created at address 2000
// it holds 1000 (address of "x")
int *y = &x ;

y++;           // y becomes 1004
```

Since the variable "y" stores 1000 (the address of "x"), we expect it to become 1001 because of the "++" operator, but it increments by 4, which is the size of "int" variable.

The is why because, if the address of "x" is 1000, then it occupies 4 bytes: 1000, 1001, 1002 and 1003. Hence, the next integer can be put only in 1004 and not before it. Hence "y" (the pointer to "x") becomes 1004 when incremented.

Example of Incrementing a Pointer

text.c

```
#include <stdio.h>

int main(){

   int x = 10;
   int *y = &x;

   printf("Value of y before increment: %d\n", y);

   y++;

   printf("Value of y after increment: %d", y);
}
```

Console

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\
Value of y before increment: 6422296
Value of y after increment: 6422300
```

Example of Decrementing a Pointer

```c
#include <stdio.h>

int main(){

   double x = 10;
   double *y = &x;

   printf("value of y before decrement: %ld\n", y);

   y--;

   printf("value of y after decrement: %ld", y);
}
```

```
value of y before decrement: 6422288
value of y after decrement: 6422280
```

When an array is declared, the elements are stored in adjacent memory locations. In case of "int" array, each array subscript is placed apart by 4 bytes, as the following figure shows –

1000	1004	1008	1012	1016	1020	1024	1028	1032	1036
10	20	30	40	50	60	70	80	90	100

Hence, if a variable stores the address of 0th element of the array, then the "increment" takes it to the 1st element.

```c
#include <stdio.h>

int main(){

   int a[]= {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
   int len = sizeof(a)/sizeof(int);
   int *x = a;
   int i = 0;

   for(i = 0; i < len; i++){
      printf("Address of subscript %d = %d Value = %d\n", i, x, *x);
      x++;
   }

   return 0;
}
```

```
Address of subscript 0 = 6422252 Value = 10
Address of subscript 1 = 6422256 Value = 20
Address of subscript 2 = 6422260 Value = 30
Address of subscript 3 = 6422264 Value = 40
Address of subscript 4 = 6422268 Value = 50
Address of subscript 5 = 6422272 Value = 60
Address of subscript 6 = 6422276 Value = 70
Address of subscript 7 = 6422280 Value = 80
Address of subscript 8 = 6422284 Value = 90
Address of subscript 9 = 6422288 Value = 100
```

Addition and Subtraction of Integer to Pointer

n integer value can be added and subtracted to a pointer. When an integer is added to a pointer, the pointer points to the next memory address. Similarly, when an integer is subtracted from a pointer, the pointer points to the previous memory location.

Addition and subtraction of an integer to a pointer does not add and subtract that value to the pointer, multiplication with the size of the data type is added or subtracted to the pointer.

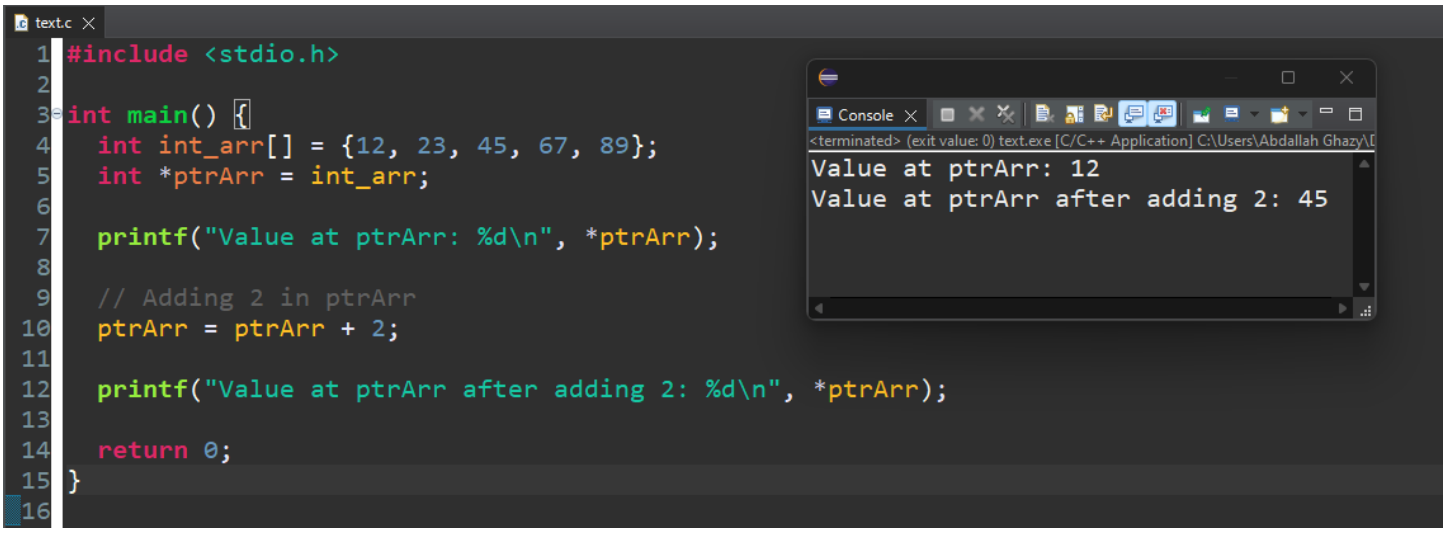
For example, there is an integer pointer variable ptr and it is pointing to an address 123400, if you add 1 to the ptr (ptr+1), it will point to the address 123404 (size of an integer is 4).

Let's evaluate it,

```
ptr = 123400
ptr = ptr + 1
ptr = ptr + sizeof(int)*1
ptr = 123400 + 4
ptr = 123404
```

Example of Adding Value to a Pointer

In the following example, we are declaring an array and pointer to an array. Initializing the pointer with the first element of the array and then adding an integer value (2) to the pointer to get the third element of the array.



Subtraction of Pointers

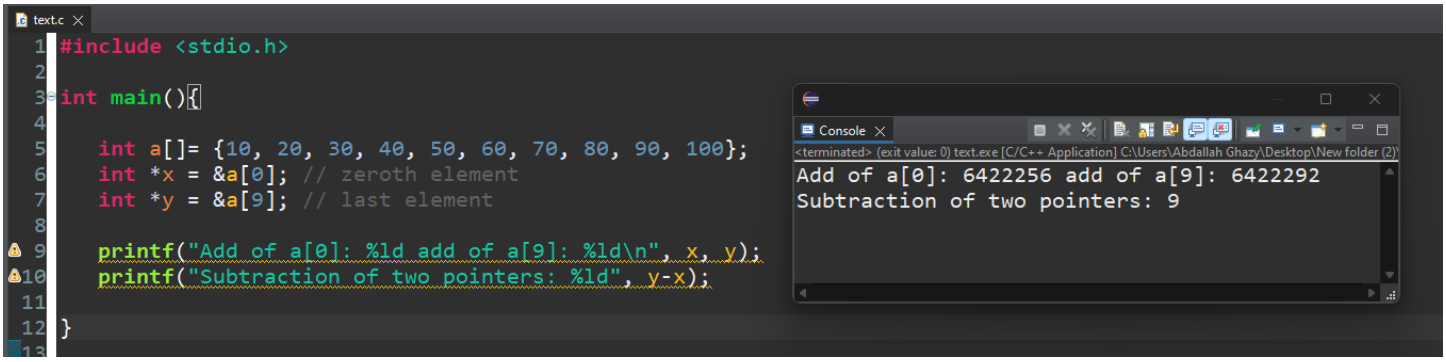
We are familiar with the "+" and "−" operators when they are used with regular numeric operands. However, when you use these operators with pointers, they behave in a little different way.

Since pointers are fairly large integers (especially in modern 64-bit systems), addition of two pointers is meaningless. When we add a 1 to a pointer, it points to the next location where an integer may be stored. Obviously, when we add a pointer (itself a large integer), the location it points may not be in the memory layout.

However, subtraction of two pointers is realistic. It returns the number of data types that can fit in the two pointers.

Example of Subtracting Two Pointers

Let us take the array in the previous example and perform the subtraction of pointers of a[0] and a[9]

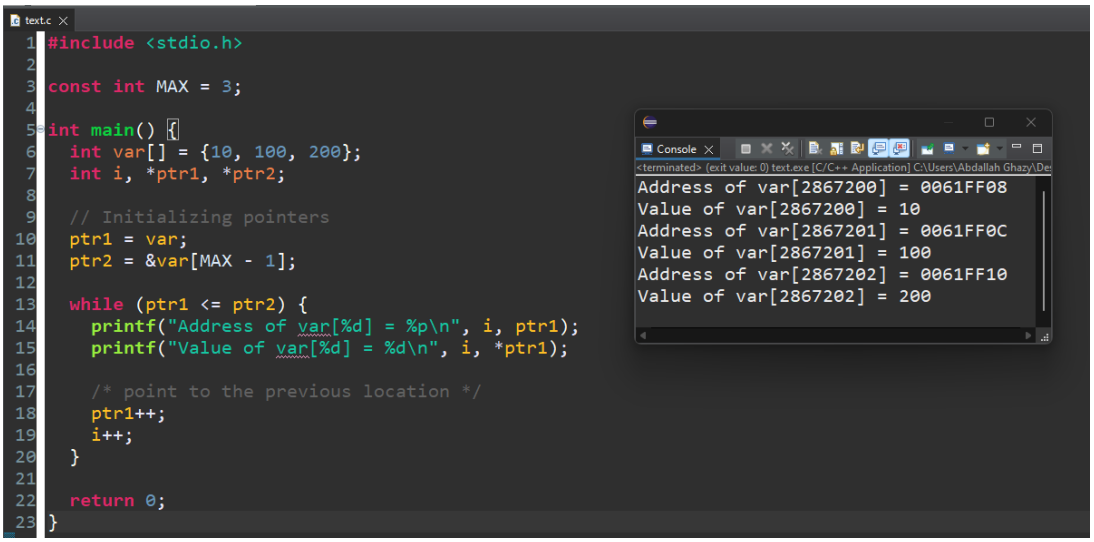


Comparison of Pointers

Pointers may be compared by using relational operators such as "==", "<", and ">". If "p1" and "p2" point to variables that are related to each other (such as elements of the same array), then "p1" and "p2" can be meaningfully compared.

Example of Comparing Pointers

In the following example, we are declaring two pointers and initializing them with the first and last elements of the array respectively. We will keep incrementing the first variable pointer as long as the address to which it points is either less than or equal to the address of the last element of the array, which is "&var[MAX – 1]" (i.e., the second pointer).



Pointer to Array

An array name is a constant pointer to the first element of the array. Therefore, in this declaration,

```
int balance[5];
```

balance is a pointer to &balance[0], which is the address of the first element of the array.

Example

```c
#include <stdio.h>

int main(){

   /* an array with 5 elements */
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *ptr;
   int i;

   ptr = balance;

   /* output each array element's value */
   printf("Array values using pointer: \n");

   for(i = 0; i < 5; i++){
      printf("*(ptr + %d): %f\n",  i, *(ptr + i));
   }

   printf("\nArray values using balance as address:\n");

   for(i = 0; i < 5; i++){
      printf("*(balance + %d): %f\n",  i, *(balance + i));
   }

   return 0;
}
```

```
Array values using pointer:
*(ptr + 0): 1000.000000
*(ptr + 1): 2.000000
*(ptr + 2): 3.400000
*(ptr + 3): 17.000000
*(ptr + 4): 50.000000

Array values using balance as address:
*(balance + 0): 1000.000000
*(balance + 1): 2.000000
*(balance + 2): 3.400000
*(balance + 3): 17.000000
*(balance + 4): 50.000000
```

In this code, we have a pointer ptr that points to the address of the first element of an integer array called balance.

```c
#include <stdio.h>

int main(){

   int *ptr;
   int balance[5] = {1, 2, 3, 4, 5};

   ptr = balance;

   printf("Pointer 'ptr' points to the address: %d", ptr);
   printf("\nAddress of the first element: %d", balance);
   printf("\nAddress of the first element: %d", &balance[0]);

   return 0;
}
```

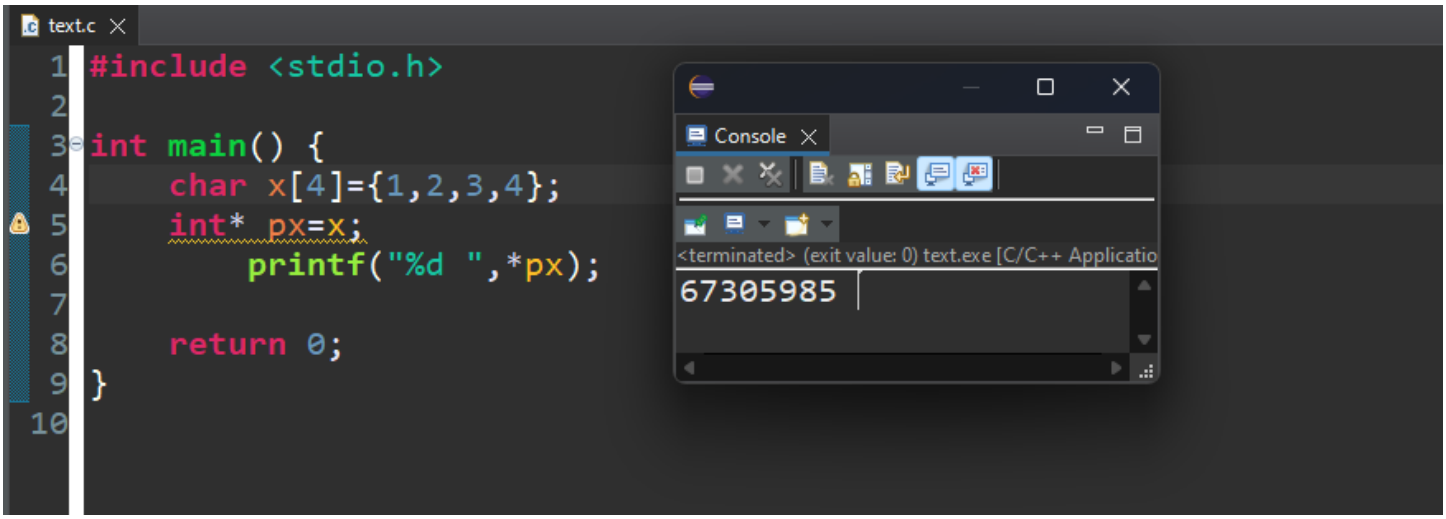
```
Pointer 'ptr' points to the address: 6422280
Address of the first element: 6422280
Address of the first element: 6422280
```

Array Names as Constant Pointers

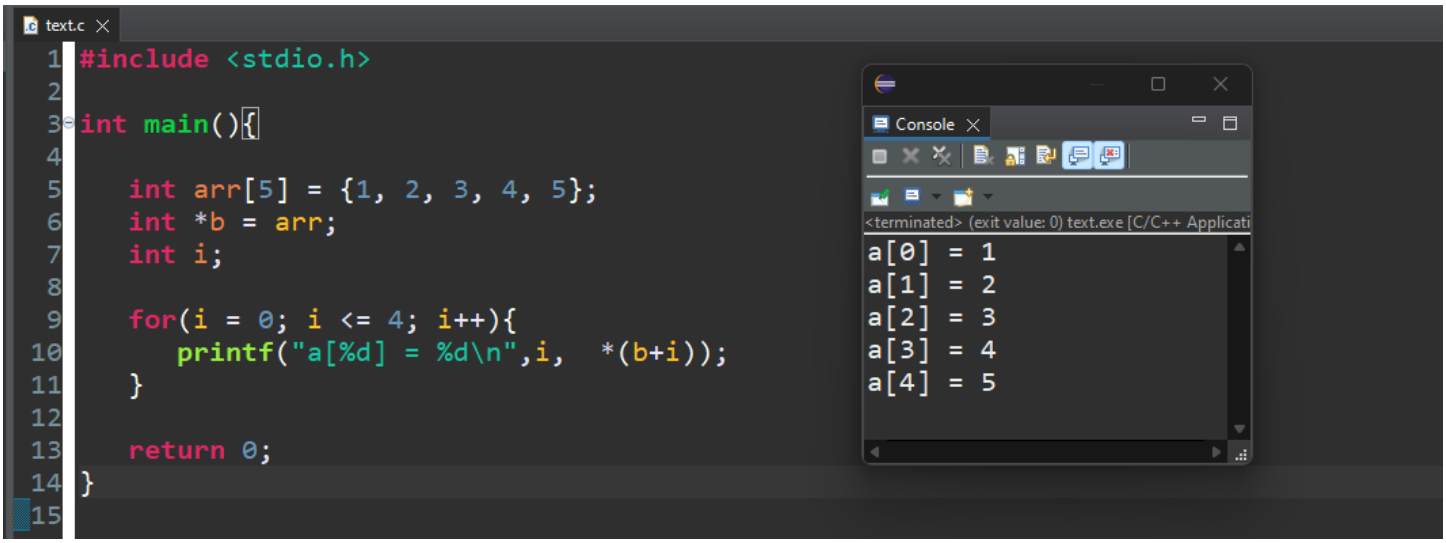
it is legal to use array names as constant pointers and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in "ptr", you can access the array elements using *ptr, *(ptr + 1), *(ptr + 2), and so on.

Example



Example: Traversing an Array using the Dereference Operator



Points to Note

- **"&arr[0]"** is equivalent to **"b"** and **"arr[0]"** to **"*b".**
- Similarly, **"&arr[1]"** is equivalent to **"b + 1"** and **"arr[1]"** is equivalent to **"*(b + 1)".**
- Also, **"&arr[2]"** is equivalent to **"b + 2"** and **"arr[2]"** is equivalent to **"*(b+2)".**
- In general, **"&arr[i]"** is equivalent to **"b + I"** and **"arr[i]"** is equivalent to **"*(b+i)".**

Pointer to Multidimensional Arrays

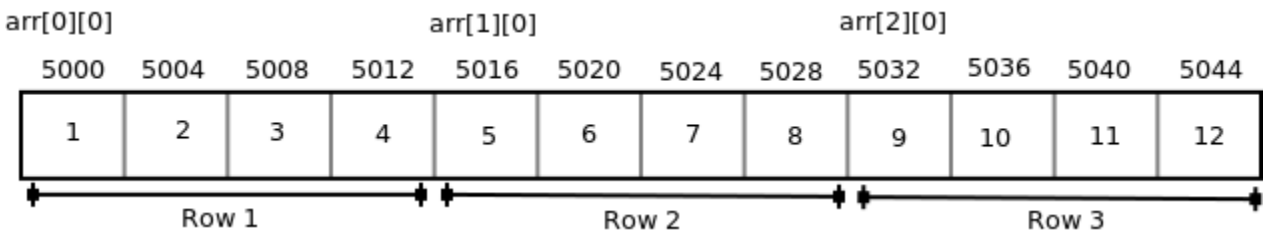
1. Pointers and Two-Dimensional Arrays

In a two-dimensional array, we can access each element by using two subscripts, where the first subscript represents the row number and the second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element arr[i][j] of the array using the pointer expression ***(*(arr + i) + j)**. Now we'll see how this expression can be derived.

Let us take a two dimensional array arr[3][4]:

```
int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

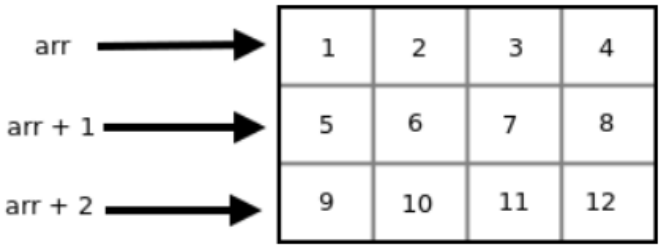
	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, we can say that 2-D dimensional arrays that are placed one after another. So here arr is an array of 3 elements where each element is a 1-D array of 4 integers.

We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000. Since arr is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression **arr + 1** will represent the address 5016 and expression arr + 2 will represent address 5032.

So we can say that arr points to the 0th 1-D array, arr + 1 points to the 1st 1-D array and arr + 2 points to the 2nd 1-D array.



arr - Points to 0th element of arr - Points to 0th 1-D array - 5000
arr + 1 - Points to 1th element of arr - Points to 1nd 1-D array - 5016
arr + 2 - Points to 2th element of arr - Points to 2nd 1-D array - 5032

Since arr + i points to ith element of arr, on dereferencing it will get ith element of arr which is of course a 1-D array. Thus the expression *(arr + i) gives us the base address of ith 1-D array.

We know, the pointer expression *(arr + i) is equivalent to the subscript expression arr[i]. So is same as arr[i] gives us the base address of ith 1-D array.

To access an individual element of our 2-D array, we should be able to access any jth element of ith 1-D array.

Since the base type of *(arr + i) is int and it contains the address of 0th element of ith 1-D array, we can get the addresses of subsequent elements in the ith 1-D array by adding integer values to *(arr + i).

For example : *(arr + i) + 1 will represent the address of 1st element of 1stelement of ith 1-D array and *(arr+i)+2 will represent the address of 2nd element of ith 1-D array.

Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array.

On dereferencing this expression, we can get the jth element of the ith 1-D array.

Pointers and Three-Dimensional Arrays

```
int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} };
```

In a three-dimensional array we can access each element by using three subscripts. Let us take a 3-D array- We can consider a three-dimensional array to be an array of 2-D array i.e each element of a 3-D array is considered to be a 2-D array. The 3-D array arr can be considered as an array consisting of two elements where each element is a 2-D array. The name of the array arr is a pointer to the 0th 2-D array.

arr	**Points to 0^{th} 2-D array.**
arr + i	**Points to i^{th} 2-D array.**
***(arr + i)**	**Gives base address of i^{th} 2-D array, so points to 0^{th} element of i^{th} 2-D array, each element of 2-D array is a 1-D array, so it points to 0^{th} 1-D array of i^{th} 2-D array.**
***(arr + i) + j**	**Points to j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j)**	**Gives base address of j^{th} 1-D array of i^{th} 2-D array so it points to 0^{th} element of j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j) + k**	**Reprents the value of j^{th} element of i^{th} 1-D array.**
***(*(arr + i) + j) + k)**	**Gives the value of k^{th} element of j^{th} 1-D array of i^{th} 2-D array.**

Thus the pointer expression *(*(*(arr + i) + j) + k) is equivalent to the subscript expression arr[i][j][k]. We know the expression *(arr + i) is equivalent to arr[i] and the expression *(*(arr + i) + j) is equivalent arr[i][j]. So we can say that arr[i] represents the base address of ith 2-D array and arr[i][j] represents the base address of the jth 1-D array.

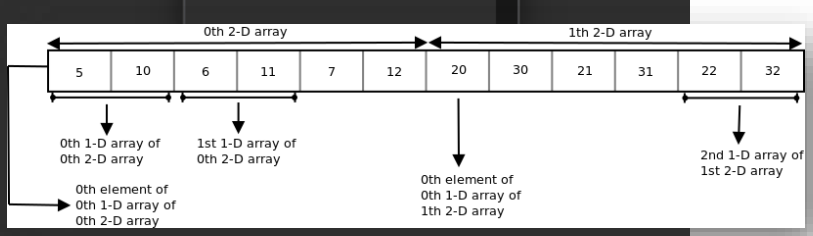
text.c | a.h | b.h | t.h

```c
// C program to print the elements of 3-D
// array using pointer notation
#include<stdio.h>
int main()
{
int arr[2][3][2] = {
                    {
                        {5, 10},
                        {6, 11},
                        {7, 12},
                    },
                    {
                        {20, 30},
                        {21, 31},
                        {22, 32},
                    }
                    };
int i, j, k;
for (i = 0; i < 2; i++)
{
    for (j = 0; j < 3; j++)
    {
    for (k = 0; k < 2; k++)
        printf("%d\t", *(*(*(arr + i) + j) +k));
    printf("\n");
    }
}

return 0;
}
```

Console

<terminated> (exit value: 0) text.exe [C/C++ Appl

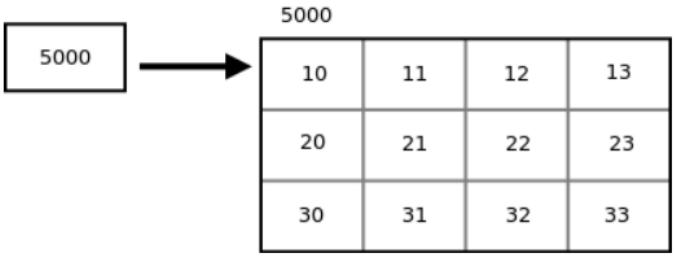
```
5       10
6       11
7       12
20      30
21      31
22      32
```



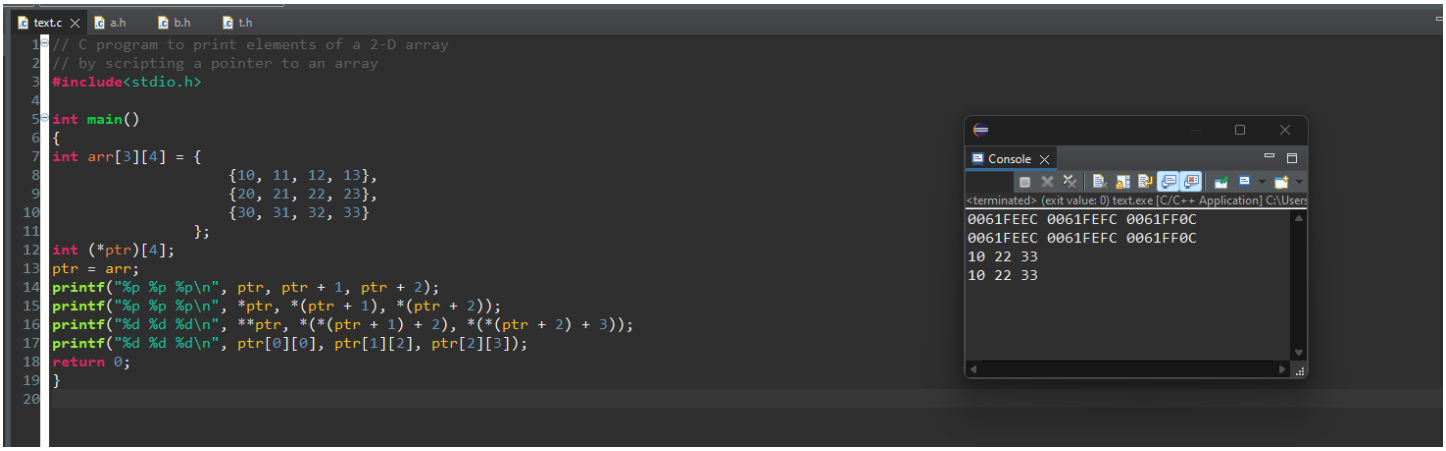
Subscripting Pointer to an Array

Suppose arr is a 2-D array with 3 rows and 4 columns and ptr is a pointer to an array of 4 integers, and ptr contains the base address of array arr.

```
int arr[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33}};
int (*ptr)[4];
ptr = arr;
```



Since ptr is a pointer to the first row 2-D array i.e. array of 4 integers, ptr + i will point to ith row. On dereferencing ptr + i, we get base address of ith row. To access the address of jth element of ith row we can add j to the pointer expression *(ptr + i). So the pointer expression *(ptr + i) + j gives the address of jth element of ith row and the pointer expression *(*(ptr + i)+j) gives the value of the jth element of ith row.



Array of Pointers in C

What is an Array of Pointers?

Just like an integer array holds a collection of integer variables, an array of pointers would hold variables of pointer type. It means each variable in an array of pointers is a pointer that points to another address.

Example of Creating an Array of Pointers

```c
#include <stdio.h>

int main() {
  // Declaring integers
  int var1 = 1;
  int var2 = 2;
  int var3 = 3;

  // Declaring an array of pointers to integers
  int *ptr[3];

  // Initializing each element of
  // array of pointers with the addresses of
  // integer variables
  ptr[0] = &var1;
  ptr[1] = &var2;
  ptr[2] = &var3;

  // Accessing values
  for (int i = 0; i < 3; i++) {
    printf("Value at ptr[%d] = %d\n", i, *ptr[i]);
  }

  return 0;
}
```

```
Value at ptr[0] = 1
Value at ptr[1] = 2
Value at ptr[2] = 3
```

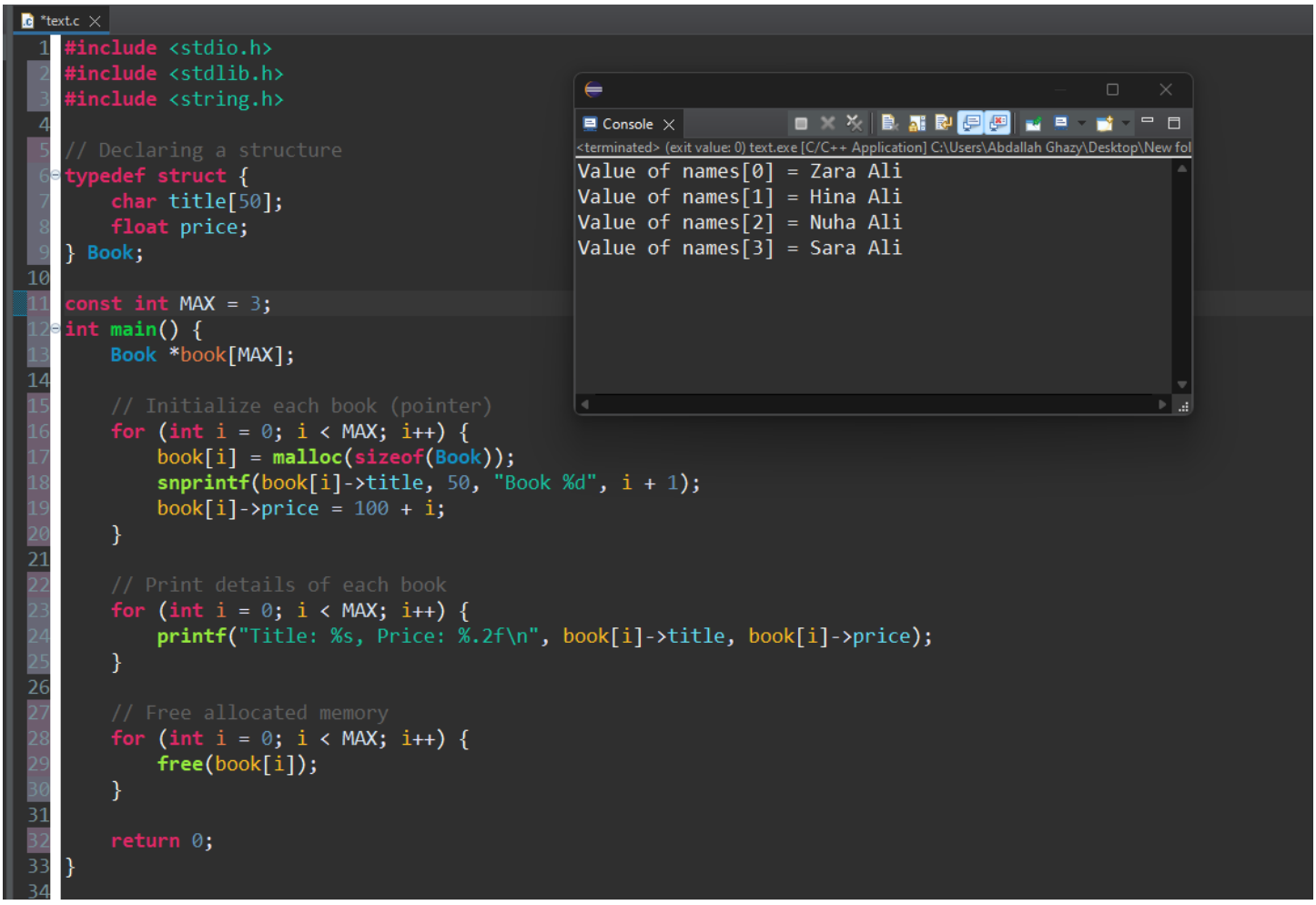
An Array of Pointers to Characters

```c
#include <stdio.h>

const int MAX = 4;

int main(){

   char *names[] = {
      "Zara Ali",
      "Hina Ali",
      "Nuha Ali",
      "Sara Ali"
   };

   int i = 0;

   for(i = 0; i < MAX; i++){
      printf("Value of names[%d] = %s\n", i, names[i]);
   }

   return 0;
}
```

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

An Array of Pointers to Structures

When you have a list of structures and want to manage it using a pointer. You can declare an array of structures to access and manipulate the list of structures.



Pointer to structure

Syntax: Defining and Declaring a Structure

This is how you will define a new derived data type using the "**struct**" keyword –

```
struct type {
   type var1;
   type var2;
   type var3;
   ...
   ...
};
```

You can then **declare** a variable of this derived data type as following –

```
struct type var;
```

You can then declare a pointer variable and store the address of **var**. To declare a variable as a pointer, it must be prefixed by "*"; and to obtain the address of a variable, we use the "**&**" operator.

```
struct type *ptr = &var;
```

Accessing the Elements of a Structure

To access the elements of a structure with pointer, we use a special operator called the indirection operator (→) .

Here, we define a user-defined struct type called book. We declare a book variable and a pointer.

```
struct book{
   char title[10];
   double price;
   int pages;
};
struct book b1 = {"Learn C", 675.50, 325},
struct book *strptr;
```

To store the address, use the **&** operator.

```
strptr = &b1;
```

Using the Indirection Operator

In C programming, we use the **indirection** operator ("→") with struct pointers. It is also called the "struct dereference operator". It helps to access the elements of a struct variable to which the pointer references to.

To access an individual element in a struct, the indirection operator is used as follows –

```
strptr -> title;
strptr -> price;
strptr -> pages;
```

The struct pointer uses the indirection operator or the dereference operator to fetch the values of the struct elements of a struct variable. The dot operator (".") is used to fetch the values with reference to the struct variable. Hence,

```
b1.title is the same as strpr -> title
b1.price is the same as strptr -> price
b1.pages is the same as strptr -> pages
```

Example: Pointers to Structures

text.c

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

struct book{
   char title[10];
   double price;
   int pages;
};

int main(){

   struct book b1 = {"Learn C", 675.50, 325};
   struct book *strptr;
   strptr = &b1;

   printf("Title: %s\n", strptr -> title);
   printf("Price: %lf\n", strptr -> price);
   printf("No of Pages: %d\n", strptr -> pages);

   return 0;
}
```

Console

<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\

```
Title: Learn C
Price: 675.500000
No of Pages: 325
```

Points to Note

- The dot operator (.) is used to access the struct elements via the struct variable.
- To access the elements via its pointer, we must use the indirection operator (→).

Example dump the memory using a pointer in C

```c
#include <stdio.h>

struct Sdata {
    unsigned char data1;
    unsigned int data2;
    unsigned char data3;
    unsigned short data4;
} data1;

void print_memory_range(unsigned char* addr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%p\t%02X\n", addr, *addr);
        addr++;
    }
}

int main() {
    data1.data1 = 0x11;
    data1.data2 = 0xFFFFEEEE;
    data1.data3 = 0x22;
    data1.data4 = 0xABCD;

    print_memory_range((unsigned char*)&data1, sizeof(data1));

    return 0;
}
```

```
00407070        11
00407071        00
00407072        00
00407073        00
00407074        EE
00407075        EE
00407076        FF
00407077        FF
00407078        22
00407079        00
0040707A        CD
0040707B        AB
```

Example

```c
#include <stdio.h>

struct Sdata {
    unsigned char data1;
    unsigned int data2;
    unsigned char data3;
    unsigned short data4;
} data1;

void print_memory_range(unsigned char* addr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%p\t%02X\n", addr, *addr);
        addr++;
    }
}

int main() {
    data1.data1 = 0x11;
    data1.data2 = 0xFFFFEEEE;
    data1.data3 = 0x22;
    data1.data4 = 0xABCD;

    print_memory_range((unsigned char*)&data1, sizeof(data1));

    struct Sdata* PStruct = &data1;

    PStruct -> data1 = 0XBB;

    printf("%p\t%02X\n",  &data1.data1,  data1.data1);

    return 0;
}
```

```
00407070        11
00407071        00
00407072        00
00407073        00
00407074        EE
00407075        EE
00407076        FF
00407077        FF
00407078        22
00407079        00
0040707A        CD
0040707B        AB
00407070        BB
```

What is the difference between using a pointer of type struct GStruct* and a pointer of type int* in the context of accessing members within a structure in C? How does each affect type safety, code readability, and ease of accessing structure members?

Pointer of Type struct GStruct*:

- **Memory**: Knows exactly how the members are organized within the structure.
- **Access**: The -> operator can be used for direct access to members, using knowledge of the internal offsets between members.
- **Type Safety**: The compiler ensures that access to members conforms to the structure definition.

Pointer of Type int*:

- **Memory**: Points to memory as int, so you need to manually calculate offsets based on the size of int.
- **Access**: Requires calculating offsets to determine the location of each member.
- **Type Safety**: Does not know about the structure of the members, which can lead to errors if offsets are not calculated correctly.

Using a pointer of type struct GStruct* makes the code clearer and safer, while a pointer of type int* requires additional calculations and can be less safe.

Pointers and Functions in C

Pointers are integral to functions in C due to their efficiency and versatility.

They enable fast data transfer and allow functions to have multiple outputs. Here are the two main features of using pointers with functions:

- Fast Data Transfer:

  - When passing large data structures (like arrays or structs) to functions, using pointers avoids copying the entire data structure, thus saving time and memory.

- Multiple Outputs:

  - Pointers allow functions to modify multiple variables, effectively enabling a function to return more than one value.

Examples Demonstrating Efficient Use of Pointers with Functions

- Fast Data Transfer

  - Passing large arrays to functions without copying the entire array:

text.c

```c
#include <stdio.h>

// Function to print an array using pointers
void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int array[] = {1, 2, 3, 4, 5};
    int size = sizeof(array) / sizeof(array[0]);

    // Passing array to function
    printArray(array, size);

    return 0;
}
```

Console

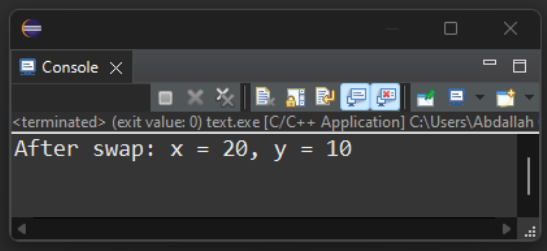
```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah
1 2 3 4 5
```

- Multiple Outputs

  - Using pointers to allow a function to modify multiple variables:

text.c

```c
#include <stdio.h>

// Function to swap two integers using pointers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;

    printf("Before swap: x = %d, y = %d\n", x, y);

    // Passing addresses of x and y to the function
    swap(&x, &y);

    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}
```



- Interfacing with Hardware

  - Pointers are essential for interfacing with hardware, where you need to access specific memory addresses:

text.c

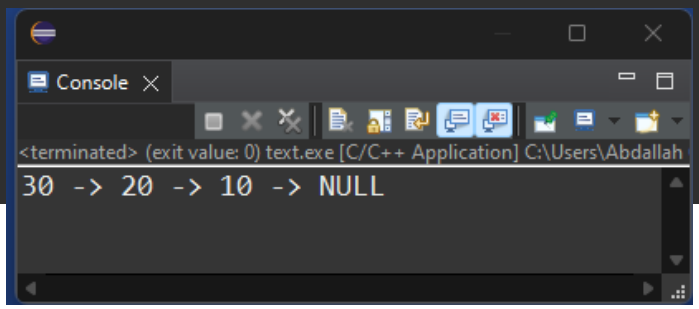
```c
#include <stdint.h>
#include <stdio.h>

#define GPIO_BASE_ADDRESS 0x40020000
#define GPIO_ODR_OFFSET   0x14

volatile uint32_t *GPIO_ODR = (uint32_t *)(GPIO_BASE_ADDRESS + GPIO_ODR_OFFSET);

void setGPIOPinHigh() {
    *GPIO_ODR |= (1 << 5); // Set bit 5
}

int main() {
    setGPIOPinHigh();
    printf("GPIO pin set high.\n");

    return 0;
}
```

In this example, setGPIOPinHigh modifies a specific hardware register through its address, showing the utility of pointers in low-level programming.

- Complex Data Structures

- Pointers enable functions to create and manipulate complex data structures like linked lists and trees:

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

// Define a node structure for a linked list
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Function to insert a new node at the beginning of the list
void insertAtBeginning(Node **head, int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}

// Function to print the linked list
void printList(Node *head) {
    Node *temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    Node *head = NULL;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 30);

    printList(head);

    // Free the allocated memory
    Node *temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```



Passing Pointers to Functions in C

A pointer In C is a variable that stores the address of another variable. It acts as a reference to the original variable. A pointer can be passed to a function, just like any other argument is passed.

A function in C can be called in two ways –

- Call by Value
- Call by Reference

To call a function by reference, you need to define it to receive the pointer to a variable in the calling function. Here is the syntax that you would use to call a function by reference –

```
type function_name(type *var1, type *var2, ...)
```

Advantages of Passing Pointers to Functions

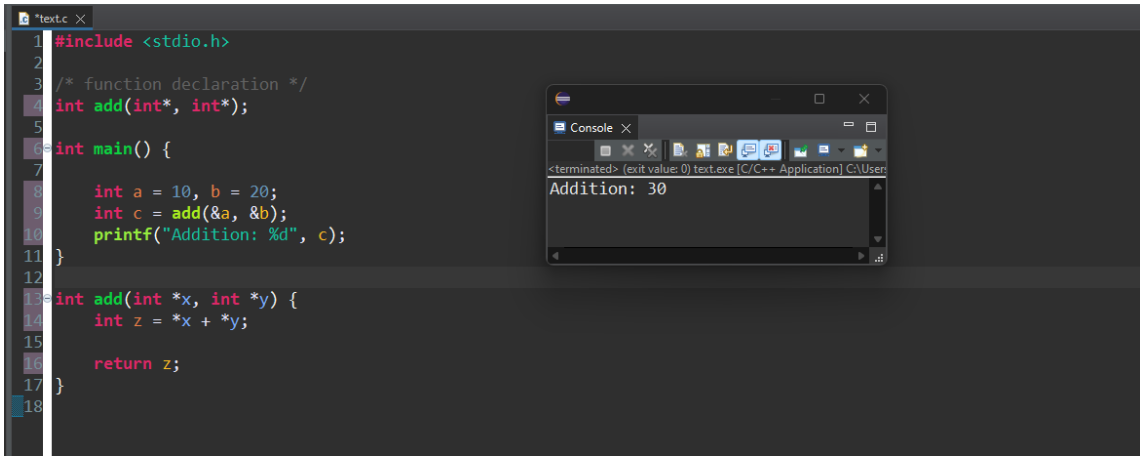
It overcomes the limitation of pass by value. Changes to the value inside the called function are done directly at the address stored in the pointer. Hence, we can manipulate the variables in one scope from another.

It also overcomes the limitation of a function that it can return only one expression. By passing pointers, the effect of processing of a function takes place directly at the address. Secondly, more than one values can be returned if we return a pointer of an array or struct variable.

- In this chapter, we shall see how to –

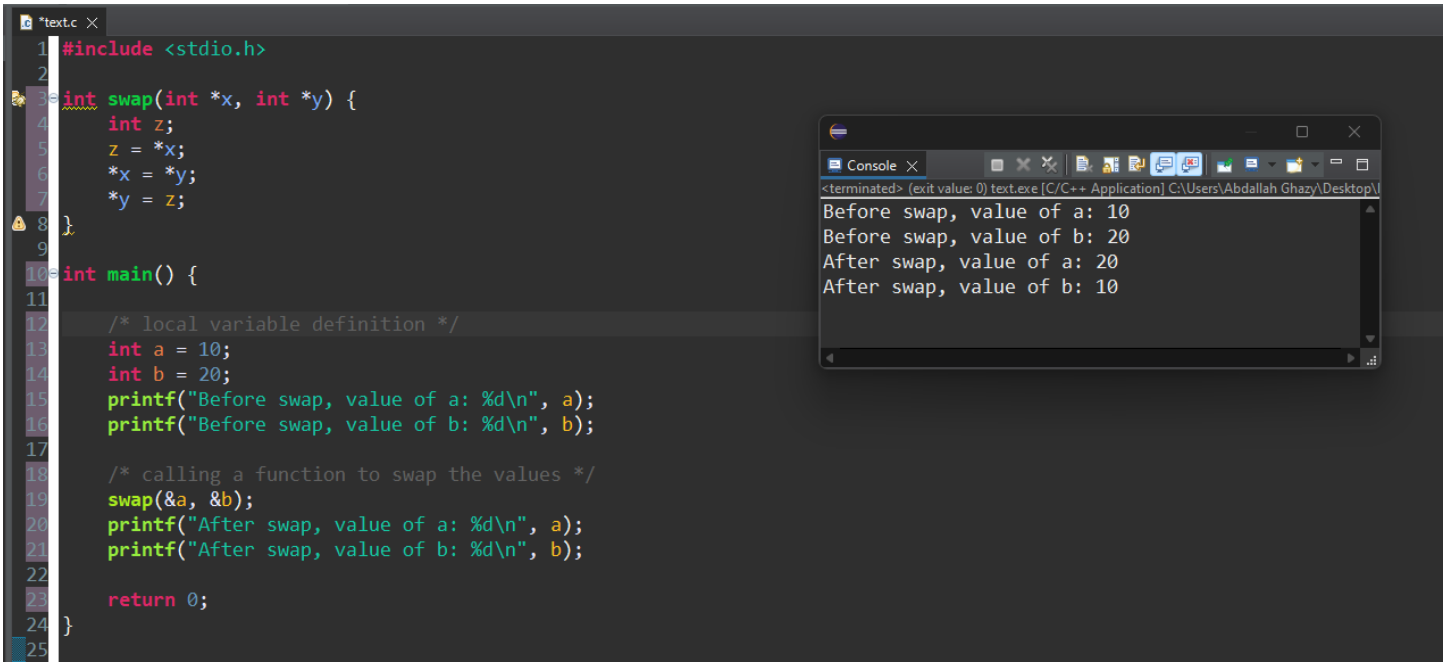
- Pass pointers to int variables
- Pass pointers to array
- Pass pointer to structure

- Example of Passing Pointers to Functions



- Swap Values by Passing Pointers

One of the most cited applications of passing a pointer to a function is how we can swap the values of two variables.

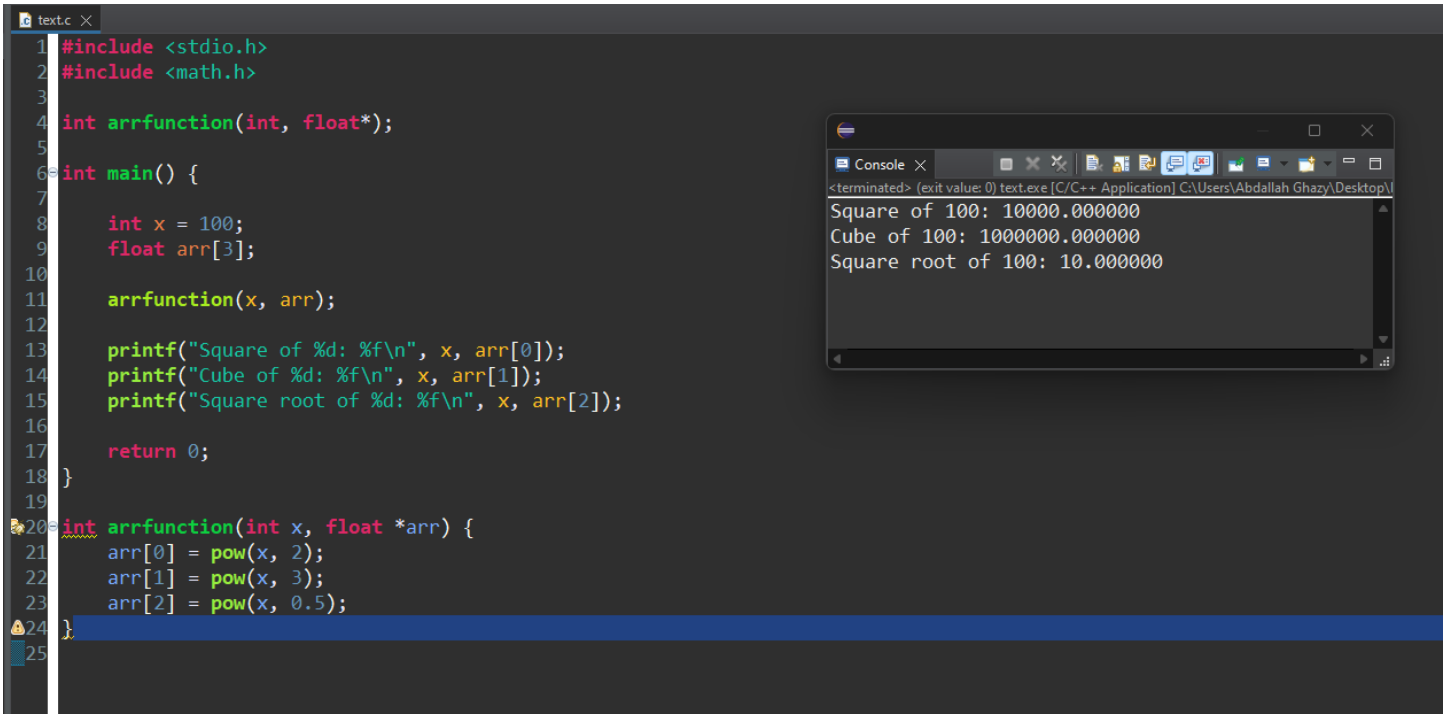


```c
#include <stdio.h>

int swap(int *x, int *y) {
    int z;
    z = *x;
    *x = *y;
    *y = z;
}

int main() {

    /* local variable definition */
    int a = 10;
    int b = 20;
    printf("Before swap, value of a: %d\n", a);
    printf("Before swap, value of b: %d\n", b);

    /* calling a function to swap the values */
    swap(&a, &b);
    printf("After swap, value of a: %d\n", a);
    printf("After swap, value of b: %d\n", b);

    return 0;
}
```

```
Before swap, value of a: 10
Before swap, value of b: 20
After swap, value of a: 20
After swap, value of b: 10
```

- Passing an Array Pointer to a Function

In C programming, the name of an array acts the address of the first element of the array; in other words, it becomes a pointer to the array.



```c
#include <stdio.h>
#include <math.h>

int arrfunction(int, float*);

int main() {

    int x = 100;
    float arr[3];

    arrfunction(x, arr);

    printf("Square of %d: %f\n", x, arr[0]);
    printf("Cube of %d: %f\n", x, arr[1]);
    printf("Square root of %d: %f\n", x, arr[2]);

    return 0;
}

int arrfunction(int x, float *arr) {
    arr[0] = pow(x, 2);
    arr[1] = pow(x, 3);
    arr[2] = pow(x, 0.5);
}
```

```
Square of 100: 10000.000000
Cube of 100: 1000000.000000
Square root of 100: 10.000000
```

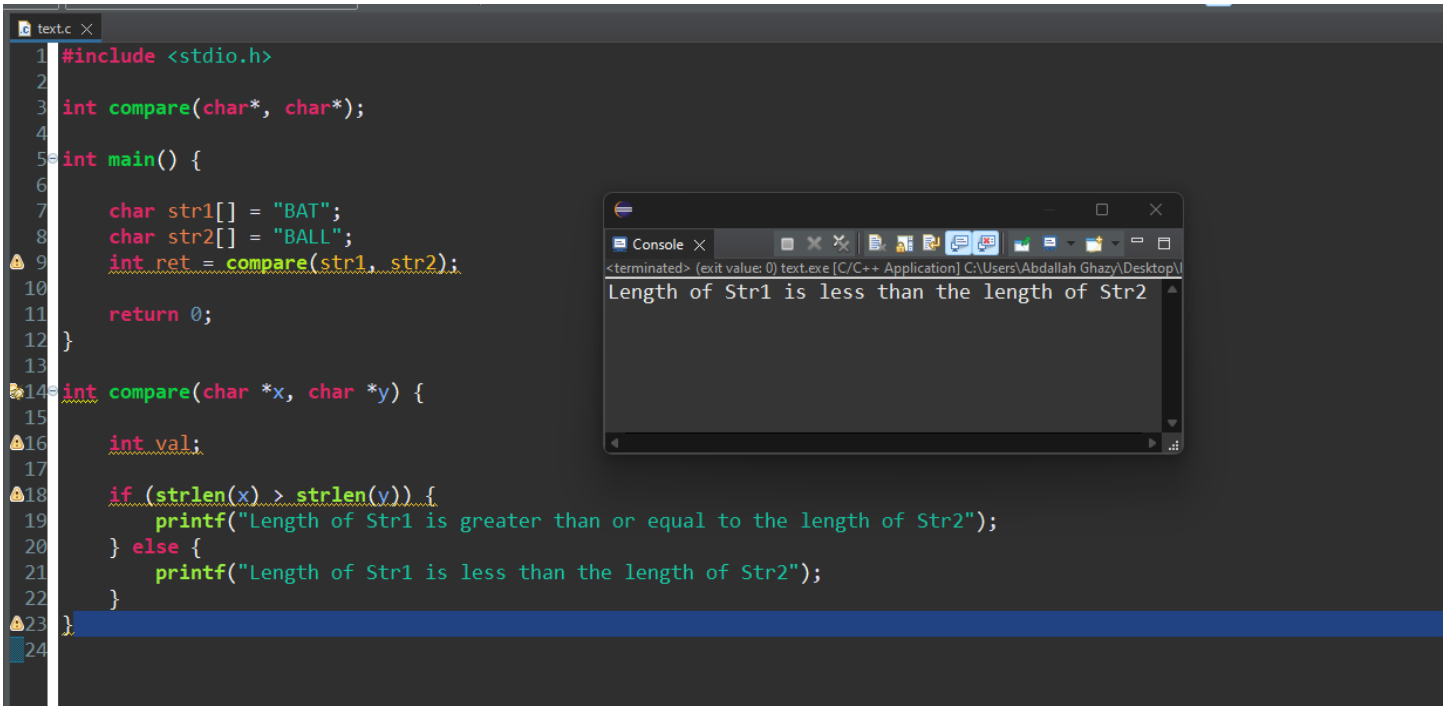
Method 2 is completely equivalent to Method 1 which means that:

`void Sort(int values[], int nValues)`	is equivelant to	`void Sort(int* values, int nValues)`

Programmer is free to choose which notation is sutable, because both methods gives the same behaviour.

- Passing String Pointers to a Function

Let us have a look at another example, where we will pass string pointers to a function.



- Passing Struct Pointer to a Function

In C programming, a structure is a heterogenous data type containing elements of different data types. Let's see how we can pass a struct pointer to a function.

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

struct rectangle{
   float len, brd;
   double area;
};

int area(struct rectangle *);

int main(){

   struct rectangle s;
   printf("Input length and breadth of a rectangle");
   scanf("%f %f", &s.len, &s.brd);
   area(&s);

   return 0;

}

int area(struct rectangle *r){

   r->area = (double)(r->len*r->brd);
   printf("Length: %f \n Breadth: %f \n Area: %lf\n", r->len, r->brd, r->area);

   return 0;
```

```
Input length and breadth of a rectangle
10.5 20.5
Length: 10.500000
Breadth: 20.500000
Area: 215.250000
```

Return a Pointer from a Function in C

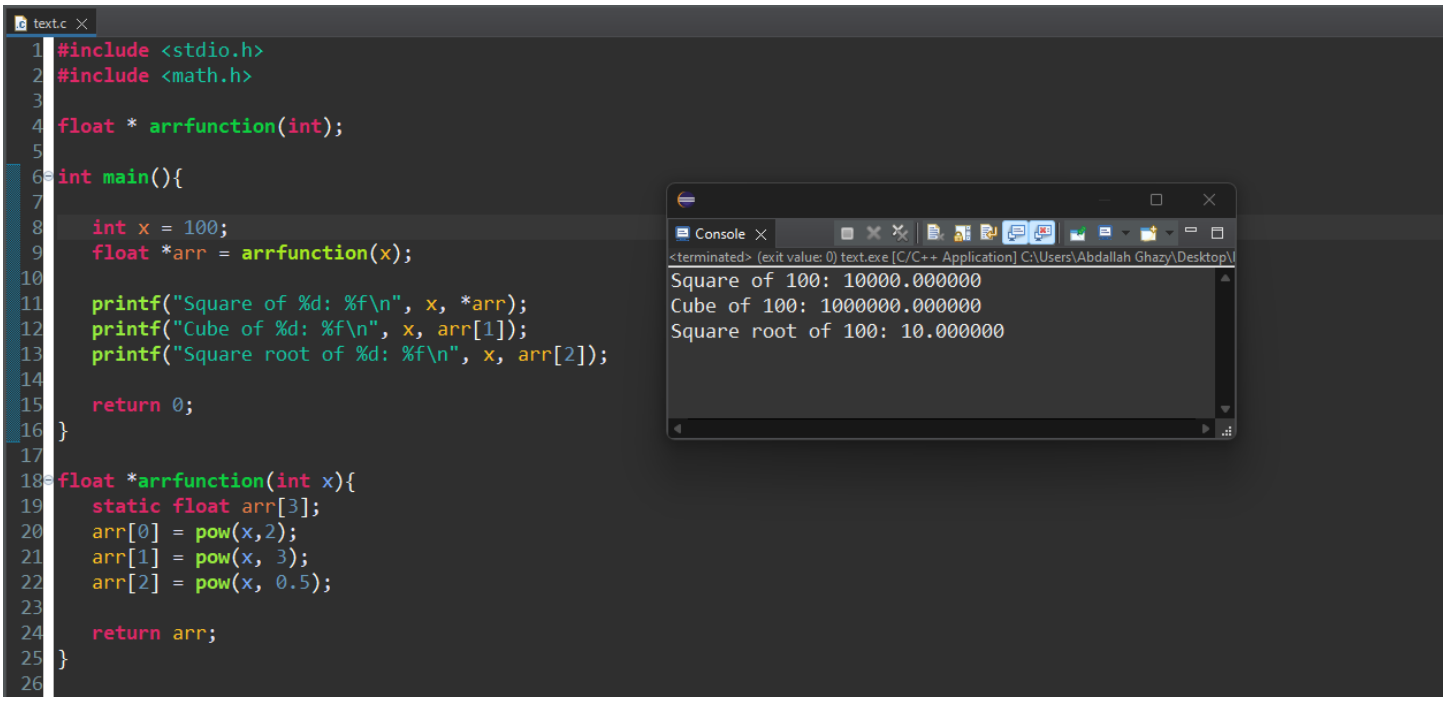
In C programming, a function can be defined to have more than one argument, but it can return only one expression to the calling function.

A function can return a single value that may be any type of variable, either of a primary type (such as int, float, char, etc.), a pointer to a variable of primary or user-defined type, or a pointer to any variables.

- Return a Static Array from a Function in C

If a function has a local variable or a local array, then returning a pointer of the local variable is not acceptable because it points to a variable that no longer exists. Note that a local variable ceases to exist as soon as the scope of the function is over.

Example 1



Example 2

Now consider the following function which will generate 10 random numbers. They are stored in a static array and return their pointer to the main() function. The array is then traversed in the main() function as follows –

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

/* function to generate and return random numbers */
int *getRandom() {
   static int  r[10];
   srand((unsigned)time(NULL));    /* set the seed */

   for(int i = 0; i < 10; ++i){
      r[i] = rand();
   }

   return r;
}

int main(){

   int *p;     /* a pointer to an int */
   p = getRandom();

   for(int i = 0; i < 10; i++) {
      printf("*(p + %d): %d\n", i, *(p + i));
   }

   return 0;
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\
*(p + 0): 253
*(p + 1): 21669
*(p + 2): 32627
*(p + 3): 8997
*(p + 4): 19307
*(p + 5): 22902
*(p + 6): 6124
*(p + 7): 27444
*(p + 8): 11278
*(p + 9): 17754
```

Return a String from a Function in C

Using the same approach, you can pass and return a string to a function.

Example

Inside the called function, we use the malloc() function to allocate the memory. The passed string is concatenated with the local string before returning.

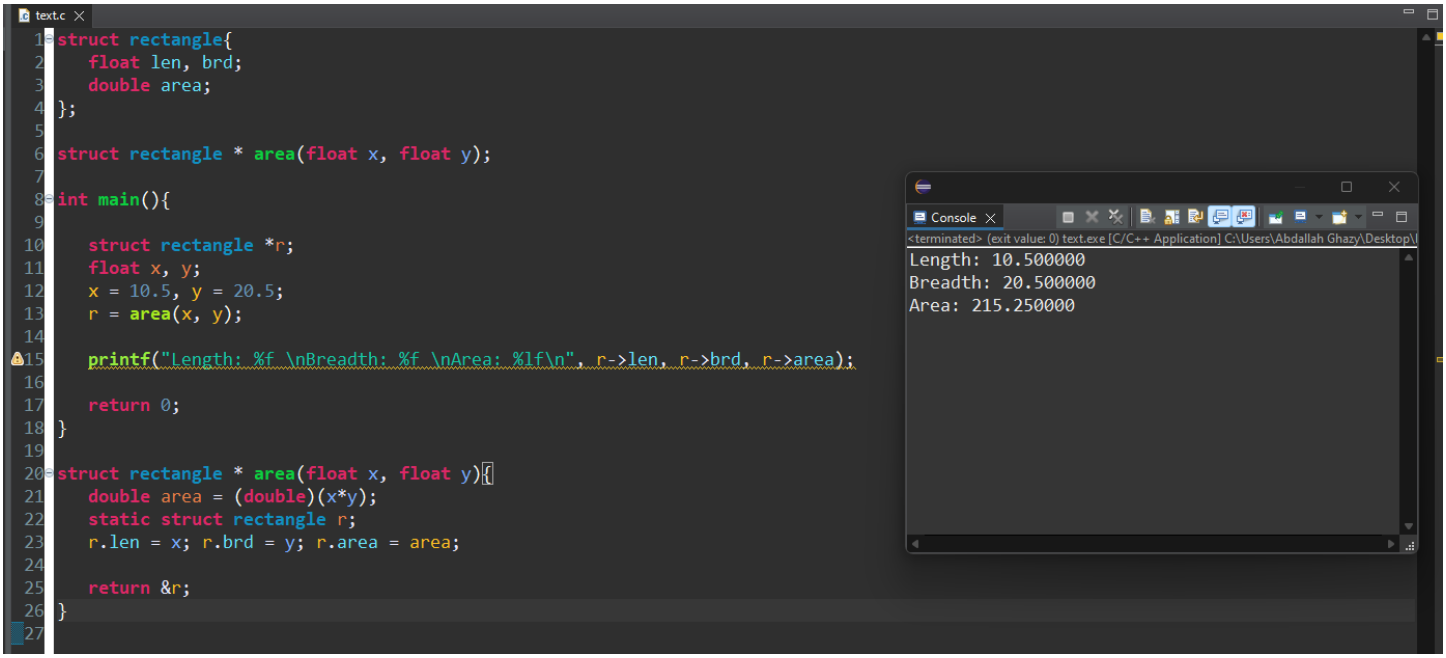
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *hellomsg(char *);

int main(){

   char *name = "TutorialsPoint";
   char *arr = hellomsg(name);
   printf("%s\n", arr);

   return 0;
}

char *hellomsg(char *x){
   char *arr = (char *)malloc(50*sizeof(char));
   strcpy(arr, "Hello ");
   strcat(arr, x);

   return arr;
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\
Hello TutorialsPoint
```

Return a Struct Pointer from a Function in C

The following example shows how you can return the pointer to a variable of struct type.

Here, the area() function has two call-by-value arguments. The main() function reads the length and breadth from the user and passes them to the area() function, which populates a struct variable and passes its reference (pointer) back to the main() function.



Function Pointers in C

What is Function Pointer in C ?

A pointer in C is a variable that stores the address of another variable. Similarly, a variable that stores the address of a function is called a function pointer or a pointer to a function. Function pointers can be useful when you want to call a function dynamically. The mechanism of callback functions in C is dependent on the function pointers.

Function Pointers point to code like normal pointers. In functions pointers, the function's name can be used to get function's address. A function can also be passed as an argument and can be returned from a function.

Declaring a Function Pointer

You should have a function whose function pointer you are going to declare. To declare a function pointer in C, write a declarative statement with the return type, pointer name, and parameter types of the function it points to.

Declaration Syntax

The following is the syntax to declare a function pointer:

```
function_return_type(*Pointer_name)(function argument list)
```

Example

Here is a simple hello() function in C –

```
void hello(){
   printf("Hello World");
}
```

We declare a pointer to this function as follows –

```
void (*ptr)() = &hello;
```

We can now call the function with the help of this function pointer "**(*ptr)();**".

Function Pointers in C

Function pointers are a powerful feature in C that allow you to store the address of a function in a pointer variable. This enables dynamic function calls, passing functions as arguments to other functions, and creating callback mechanisms.

Key Points about Function Pointers

- Function Address:
  - Every function has an address in memory, which is its entry point. This address can be stored in a pointer variable.
- Calling a Function through a Pointer:
  - Once you have a pointer to a function, you can call the function using that pointer.
- Passing Functions as Arguments:
  - Function pointers allow functions to be passed as arguments to other functions, enabling flexible and reusable code.

Syntax

Declaring a Function Pointer

c نسخ الكود

```c
return_type (*pointer_name)(parameter_types);
```

Assigning a Function to a Pointer

c نسخ الكود

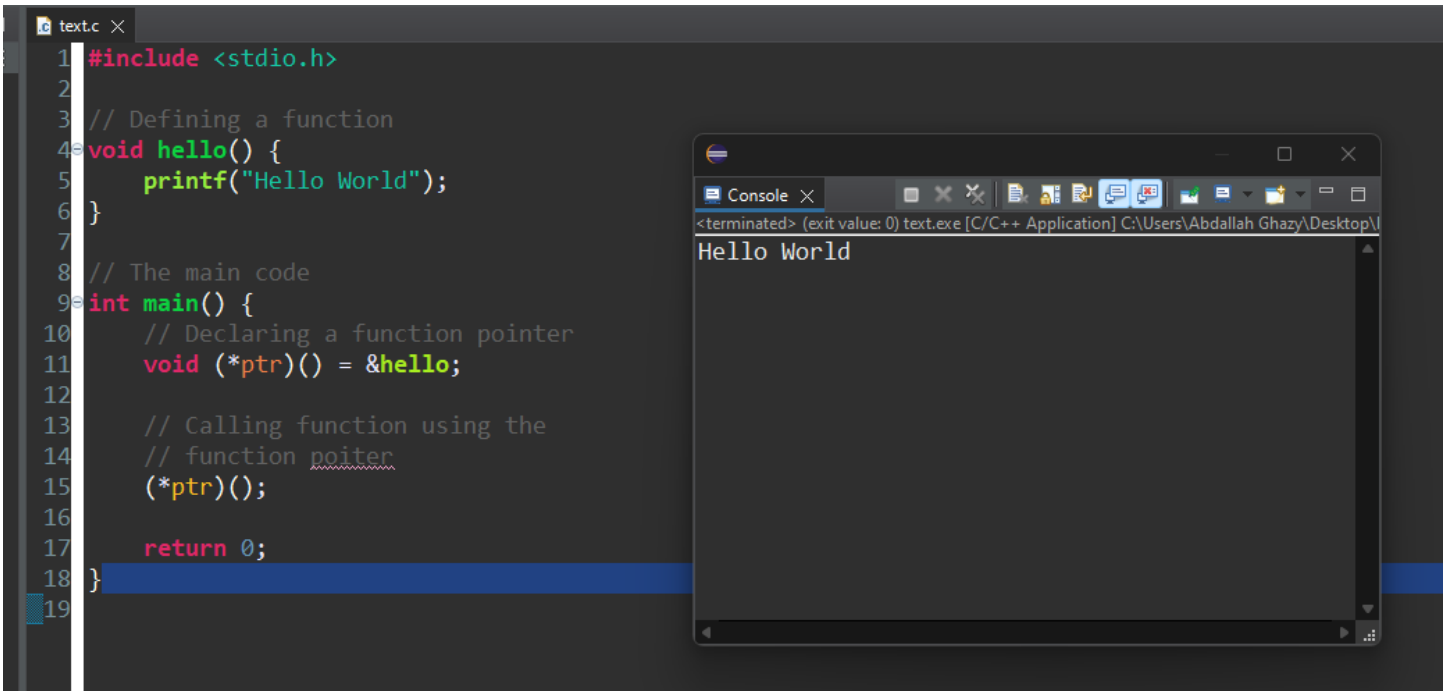
```c
pointer_name = function_name;
```

Calling a Function through a Pointer

c نسخ الكود

```c
(*pointer_name)(arguments);
// Or simply
pointer_name(arguments);
```

Function Pointer Example



Note: Unlike normal pointers which are data pointers, a function pointer points to code. We can use the name of the function as its address (as in case of an array). Hence, the pointer to the function hello() can also be declared as follows -

```
void (*ptr)() = hello;
```

Function Pointer with Arguments

A function pointer can also be declared for the function having arguments. During the declaration of the function, you need to provide the specific data types as the parameters list.

Understanding Function Pointer with Arguments

Suppose we have a function called **addition()** with two arguments –

```
int addition (int a, int b){

   return a + b;
}
```

To declare a function pointer for the above function, we use two arguments –

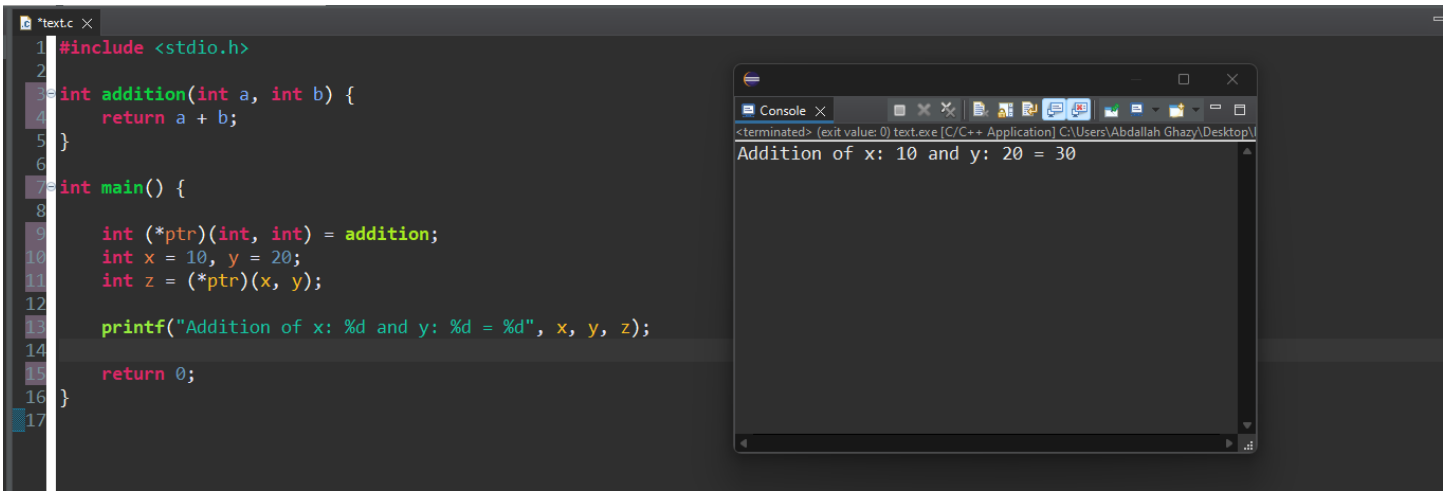
```
int (*ptr)(int, int) = addition;
```

We can then call the function through its pointer, by passing the required arguments –

```
int z = (*ptr)(x, y);
```

Try the complete code as below –

Example of Function Pointer with Arguments



Pointer to Function with Pointer Arguments

We can also declare a function pointer when the host function itself as pointer arguments.

Understanding Pointer to Function with Pointer Arguments

- We have a swap() function that interchanges the values of "x" and "y" with the help of their pointers –

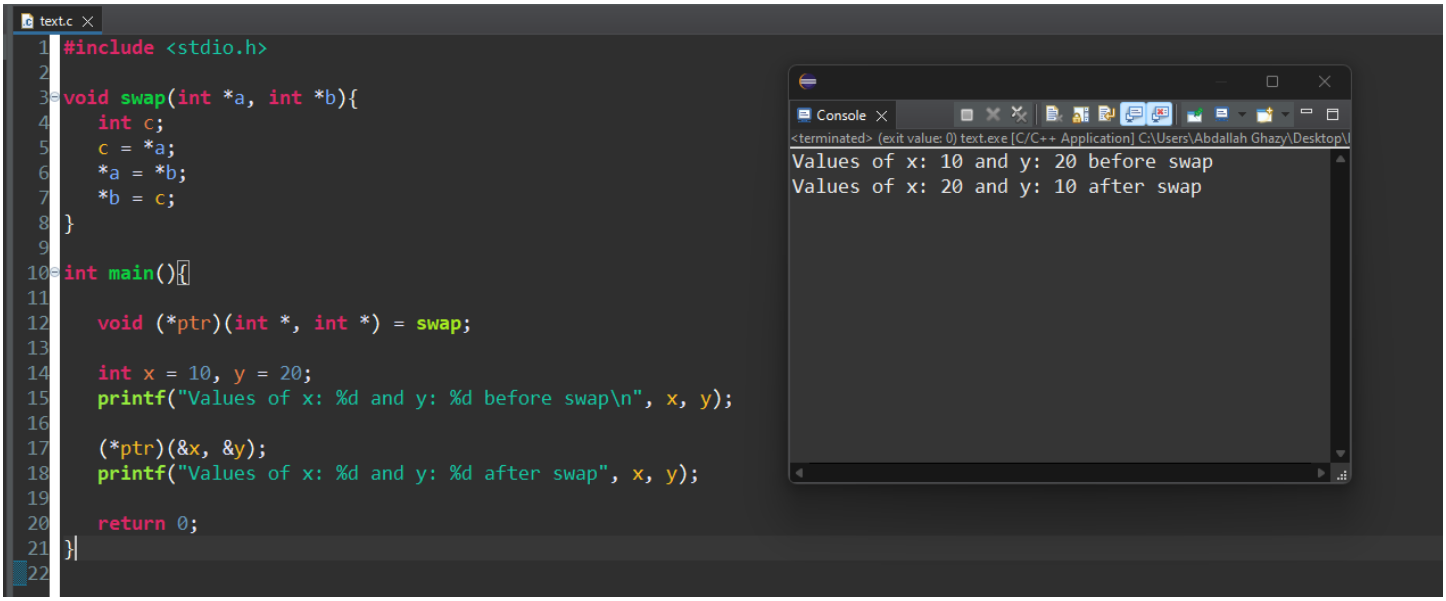
```
void swap(int *a, int *b){
   int c;
   c = *a;
   *a = *b;
   *b = c;
}
```

By following the syntax of declaring a function pointer, it can be declared as follows –

```
void (*ptr)(int *, int *) = swap;
```

To swap the values of "x" and "y", pass their pointers to the above function pointer –

```
(*ptr)(&x, &y);
```



Benefits of Using Function Pointers

- Dynamic Function Calls:

- Function pointers allow you to call different functions dynamically based on runtime conditions.

- Callback Mechanisms:

- They enable callback mechanisms where a function can be passed as an argument and called later, which is useful in event-driven programming.

- Flexibility and Reusability:

- They enhance code flexibility and reusability by allowing functions to be passed as arguments, making it easy to implement strategies and handlers.

- Simplifying Complex Programs:

- They simplify complex programs by allowing you to replace if-else or switch-case statements with more elegant and manageable solutions.

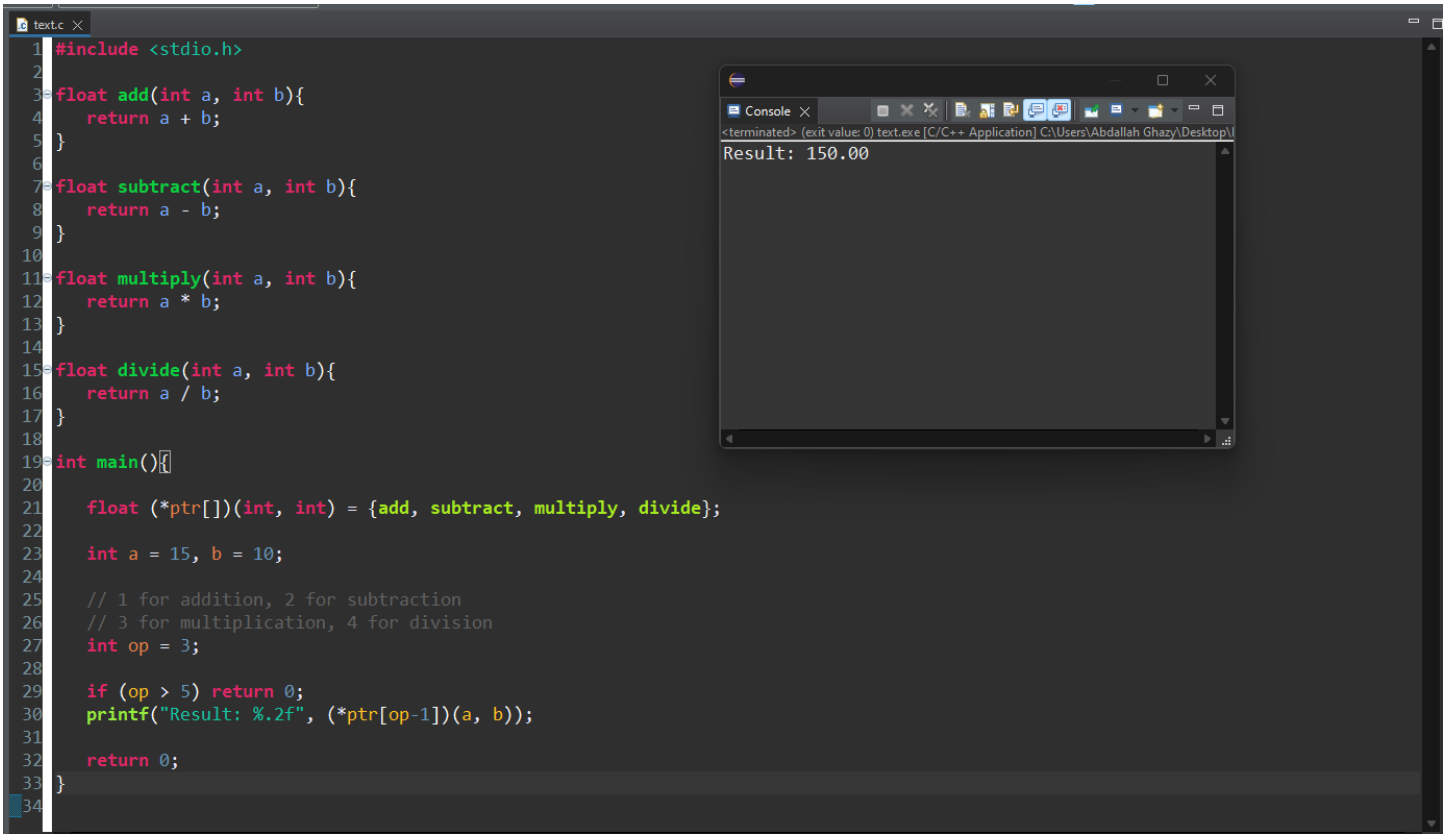
Array of Function Pointers

You can also declare an array of function pointers as per the following syntax:

```
type (*ptr[])(args) = {fun1, fun2, ...};
```

Example of Array of Function Pointers

We can use the property of dynamically calling the function through the pointers instead of if-else or switch-case statements. Take a look at the following example –



Character Pointers and Functions in C

What is a Character Pointer in C?

A character pointer stores the address of a character type or address of the first character of a character array (string). Character pointers are very useful when you are working to manipulate the strings.

There is no string data type in C. An array of "char" type is considered as a string. Hence, a pointer of a char type array represents a string. This char pointer can then be passed as an argument to a function for processing the string.

Declaring a Character Pointer

A character pointer points to a character or a character array. Thus, to declare a character pointer, use the following syntax:

```
char *pointer_name;
```

Initializing a Character Pointer

After declaring a character pointer, you need to initialize it with the address of a character variable. If there is a character array, you can simply initialize the character pointer by providing the name of the character array or the address of the first elements of it.

Character Pointer of Character

The following is the syntax to initialize a character pointer of a character type:

```
char *pointer_name = &char_variable;
```

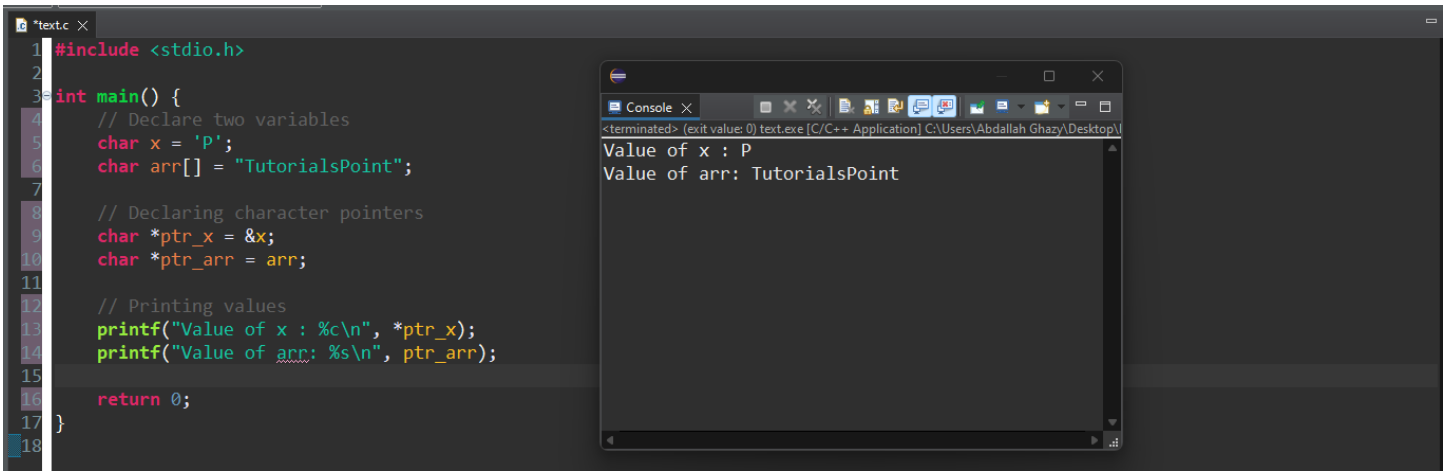
Character Pointer of Character Array

The following is the syntax to initialize a character pointer of a character array (string):

```
char *pointer_name = char_array;
/*or*/
char *pointer_name = &char_array[0];
```

Character Pointer Example

In the following example, we have two variables character and character array. We are taking two pointer variables to store the addresses of the character and character array, and then printing the values of the variables using the character pointers.



Understanding Character Pointer

- A string is declared as an array as follows –

```
char arr[] = "Hello";
```

The string is a NULL terminated array of characters. The last element in the above array is a NULL character (\0).

Declare a pointer of char type and assign it the address of the character at the 0th position –

```
char *ptr = &arr[0];
```

Remember that the name of the array itself is the address of 0th element.

```
char *ptr = arr;
```

A string may be declared using a pointer instead of an array variable (no square brackets).

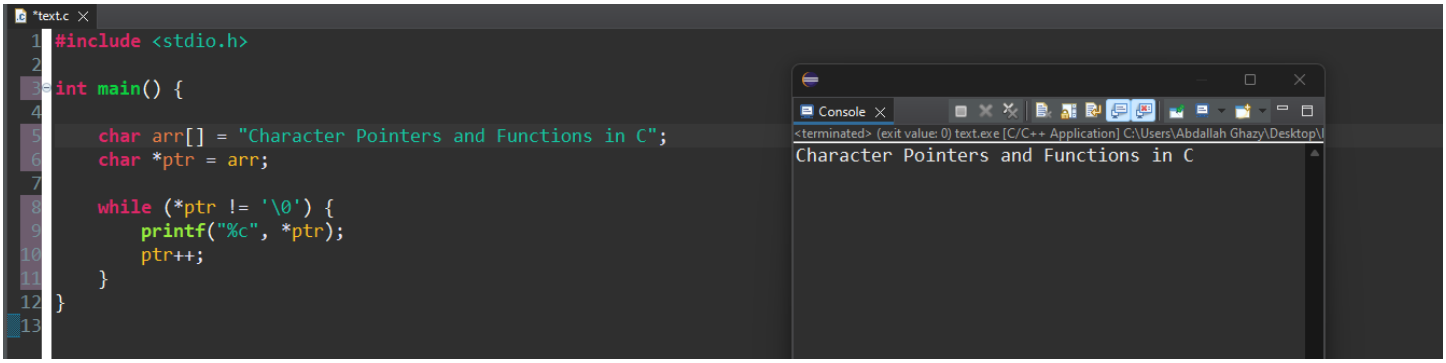
```
char *ptr = "Hello";
```

- This causes the string to be stored in the memory, and its address stored in ptr. We can traverse the string by incrementing the ptr.

```
while(*ptr != '\0'){
    printf("%c", *ptr);
    ptr++;
}
```

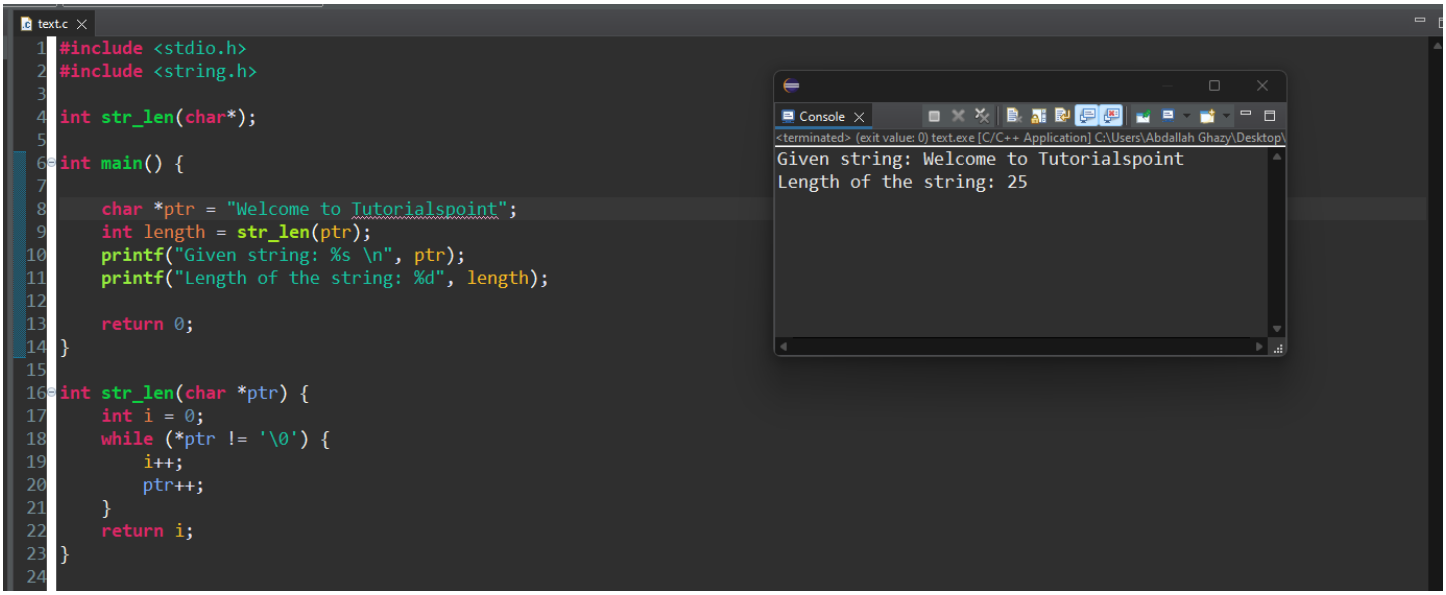
Accessing Character Array

If you print a character array using the %s format specifier, you can do it by using the name of the character pointer. But if you want to access each character of the character array, you have to use an asterisk (*) before the character pointer name and then increment it.



Example

Effectively, the strlen() function computes the string length as per the user-defined function str_len() as shown below –



Pointer with Unknown Type (void*)

void Pointer in C

A void pointer in C is a type of pointer that is not associated with any data type. A void pointer can hold an address of any type and can be typecasted to any type. They are also called general-purpose or generic pointers.

In C programming, the function malloc() and calloc() return "void *" or generic pointers.

```
void *ptr;
```

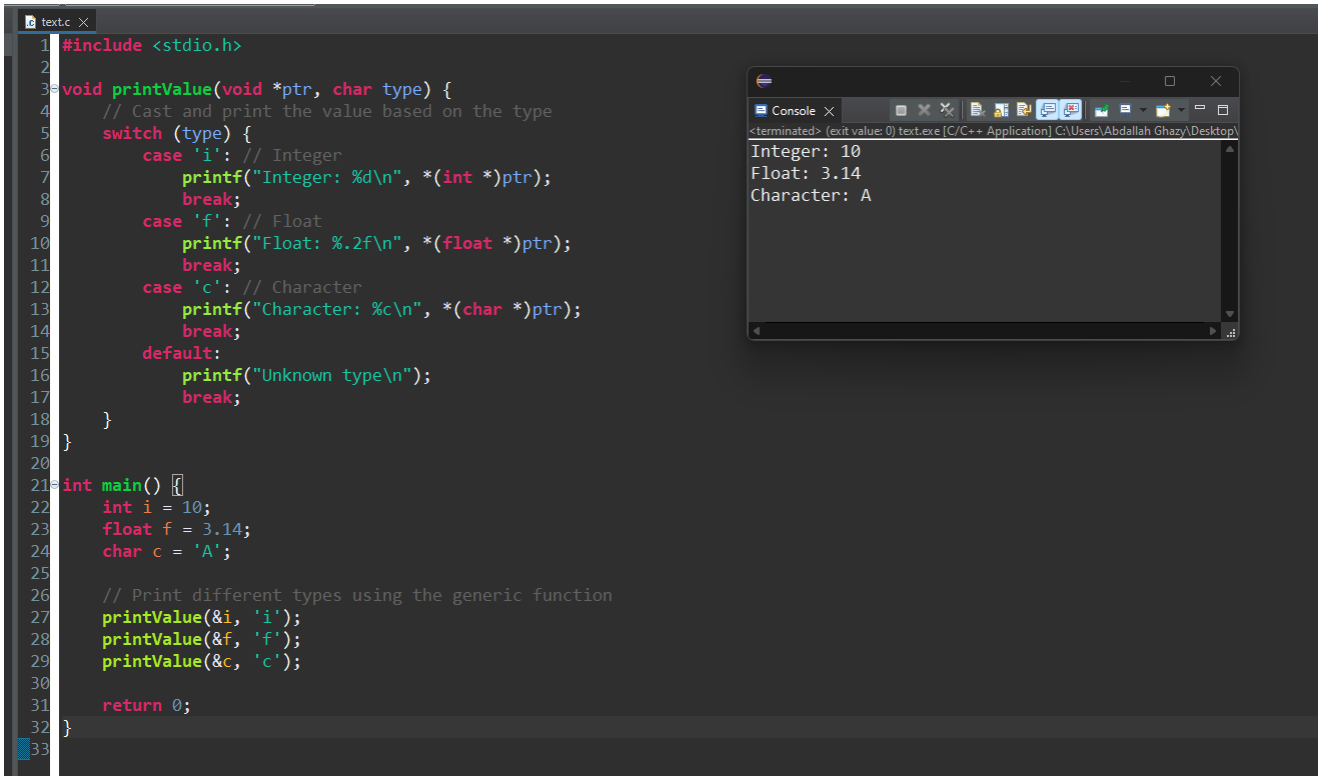
Characteristics of Void Pointers

- Type-Agnostic:

○ A void* pointer can hold the address of any data type but does not know the type of data it points to. This is useful for writing generic functions or for dynamic memory allocation.

- Type Casting Required:

○ Since a void* does not have information about the type of data it points to, you must cast it to the appropriate type before dereferencing it. This is crucial for correct data manipulation and access.



Key Points

- Cannot be Dereferenced Directly:

- You cannot directly dereference a void* pointer because the compiler does not know what type of data it points to. You must cast it to a specific type first.

- Useful for Generic Functions:

- void* pointers are often used in functions that need to operate on different data types without knowing the exact type at compile time.

- Memory Allocation Functions:

- Functions like malloc, calloc, and realloc return void* pointers because they can be used to allocate memory for any data type.

Characteristics of void* Pointers

- Generic Pointer:

- A void* pointer can be assigned to any other type of pointer and vice versa. This makes void* a versatile tool for working with pointers of different types.

- Base Type Unknown:

- A void* pointer is used when the type of data it points to is not known. This is useful in functions that need to handle different data types generically.

- Type Conversion:

- While a void* can be assigned to and from any other pointer type, it must be explicitly cast to the appropriate type before dereferencing. This is because the void* type does not provide information about the size or type of the data being pointed to.

- Raw Memory:

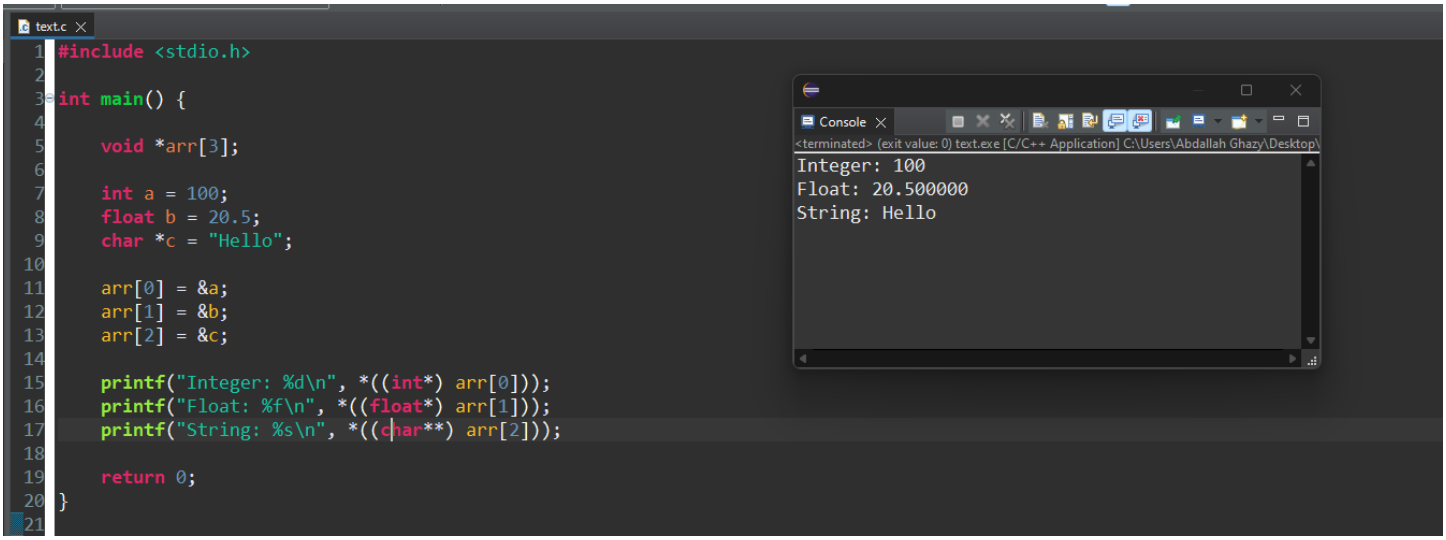
- void* pointers are commonly used to refer to raw memory blocks, such as those allocated by functions like malloc, calloc, and realloc.

An Array of void Pointers

- We can declare an array of void pointers and store pointers to different data types.

A void pointer is a pointer that can hold the memory address of any data type in C. Hence, an array of void pointers is an array that can store memory addresses, but these addresses can point to variables of different data types.

Example



Application of void Pointers

The malloc() function is available as a library function in the header file stdlib.h. It dynamically allocates a block of memory during the runtime of a program. Normal declaration of variables causes the memory to be allocated at the compile time.

void *malloc(size_t size);

Void pointers are used to implement generic functions. The dynamic allocation functions malloc() and calloc() return "void *" type and this feature allows these functions to be used to allocate memory of any data type.

The most common application of void pointers is in the implementation of data structures such as linked lists, trees, and queues, i.e., dynamic data structures.

Limitations of void Pointer

- Pointer arithmetic is not possible with void pointer due to its concrete size.
- It can't be used as dereferenced.
- A void pointer cannot work with increment or decrement operators because it doesn't have a specific type.

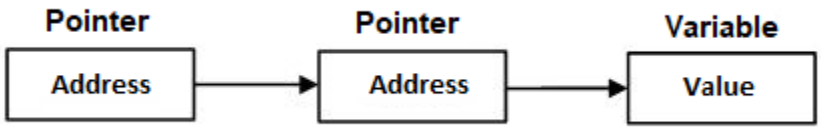
Pointer to Pointer (Double Pointer) in C

What is a Double Pointer in C?

A pointer to pointer which is also known as a double pointer in C is used to store the address of another pointer.

A "pointer to a pointer" is a form of multiple indirections or a chain of pointers.

Normally, a pointer contains the address of a variable. When we define a "pointer to a pointer", the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below

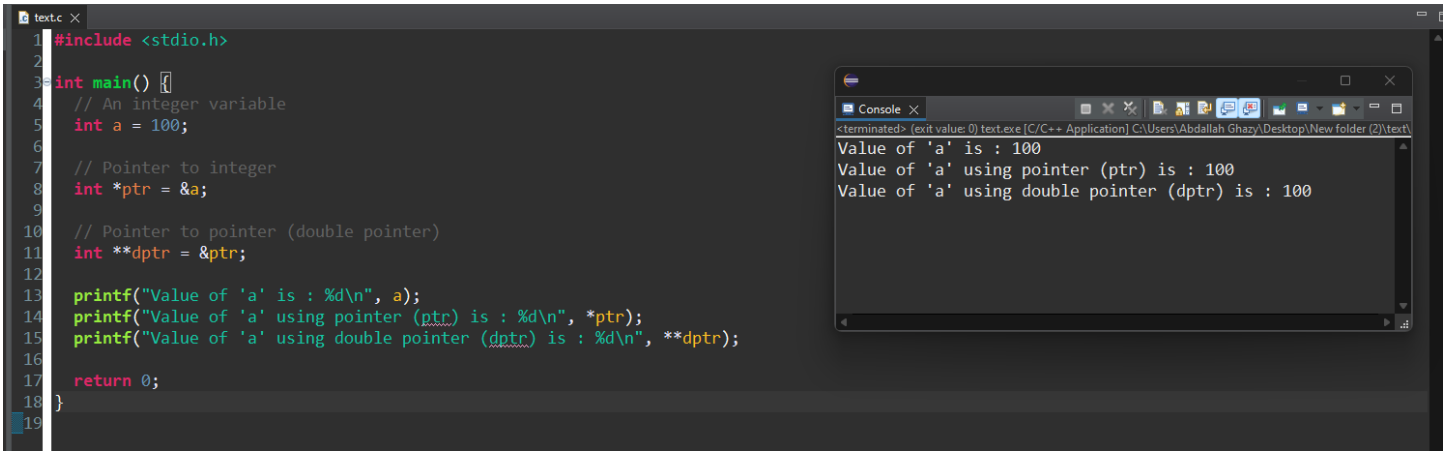


Declaration of Pointer to a Pointer

The declaration of a pointer to pointer (double pointer) is similar to the declaration of a pointer, the only difference is that you need to use an additional asterisk (*) before the pointer variable name.

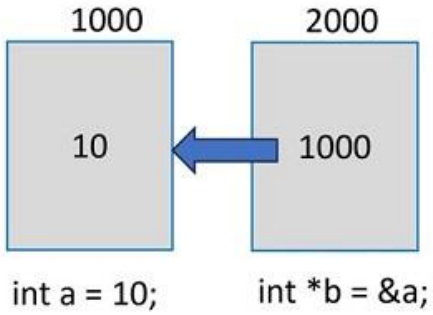
```
int **var;
```

Example of Pointer to Pointer (Double Pointer)



How Does a Normal Pointer Work in C ?

ssume that an integer variable "a" is located at an arbitrary address 1000. Its pointer variable is "b" and the compiler allocates it the address 2000. The following image presents a visual depiction –



Let us declare a pointer to **int** type and store the address of an **int** variable in it.

```
int a = 10;
int *b = &a;
```

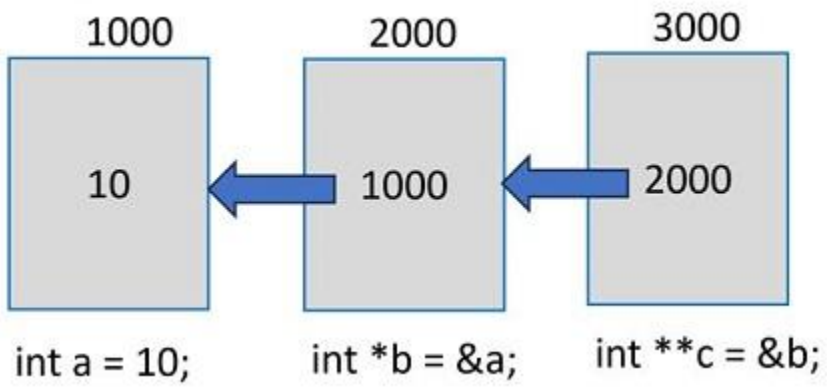
The **dereference operator** fetches the value via the pointer.

```
printf("a: %d \nPointer to 'a' is 'b': %d \nValue at 'b': %d", a, b, *b);
```

How Does a Double Pointer Work?

Let us now declare a pointer that can store the address of "b", which itself is a pointer to int type written as "int *".

Let's assume that the compiler also allocates it the address 3000.



Hence, **"c"** is a pointer to a pointer to **int,** and should be declared as **"int **"**.

```
int **c = &b;
printf("b: %d \nPointer to 'b' is 'c': %d \nValue at b: %d\n", b, c, *c);
```

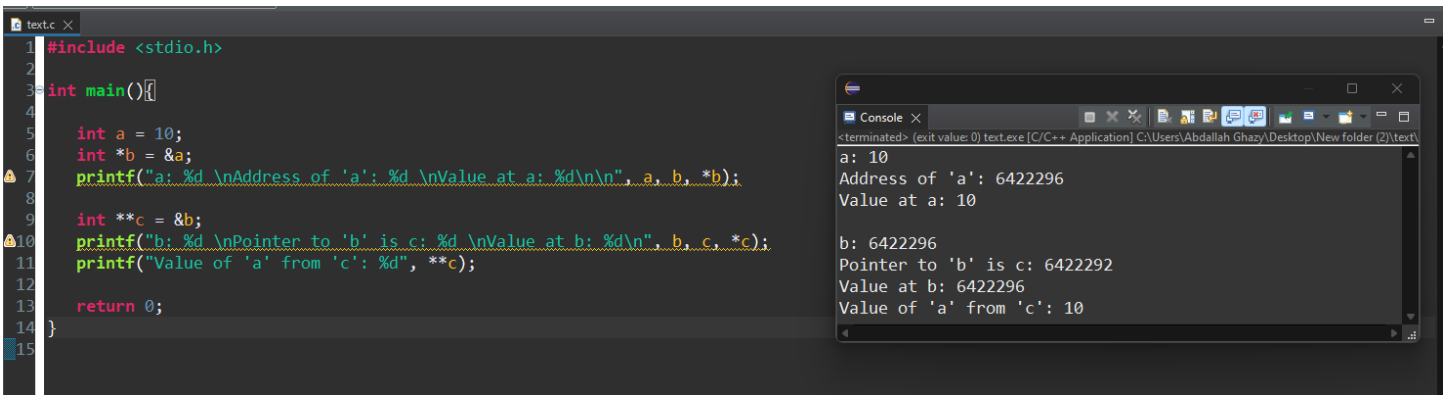
You get the value of "b" (which is the address of "a"), the value of "c" (which is the address of "b:), and the dereferenced value from "c" (which is the address of "a") –

```
b: 6422036
Pointer to b is c: 6422024
Value at b: 6422036
```

Here, "c" is a double pointer. The first asterisk in its declaration points to "b" and the second asterisk in turn points to "a". So, we can use the double reference pointer to obtain the value of "a" from "c".

```
printf("Value of 'a' from 'c': %d", **c);
```

This should display the value of 'a' as 10.



Multilevel Pointers in C (Is a Triple Pointer Possible?)

Theoretically, there is no limit to how many asterisks can appear in a pointer declaration.

If you do need to have a pointer to "c" (in the above example), it will be a "pointer to a pointer to a pointer" and may be declared as –

```
int ***d = &c;
```

Mostly, double pointers are used to refer to a two–dimensional array or an array of strings.

Pointer vs Array in C

Arrays and Pointers are two important language constructs in C, associated with each other in many ways. In many cases, the tasks that you perform with a pointer can also be performed with the help of an array.

However, there are certain conceptual differences between arrays and pointers.

Arrays in C

In a C program, an array is an indexed collection of elements of similar type, stored in adjacent memory locations.

❖ sizeof Operator:

- **sizeof(array)** returns the total size of the array in bytes, which is the size of one element multiplied by the number of elements.
- sizeof(pointer) returns the size of the pointer itself, typically 4 bytes on a 32-bit system and 8 bytes on a 64-bit system.

❖ & Operator:

&array is equivalent to **&array[0]**, which returns the address of the **first element** in the array.

&pointer returns the address of the pointer variable itself, not the address it points to.

❖ String Literal Initialization:

char array[] = "abc" initializes the array with the characters 'a', 'b', 'c', and the null terminator '\0'.

char *pointer = "abc" sets the pointer to the address of the string literal "abc" in memory, which might be in **read-only memory.ROM**

❖ Assignment:

You can assign a value to a pointer variable, but you cannot assign a value to an array variable directly. For example, **pointer = array** is legal, but **array = pointer** is not.

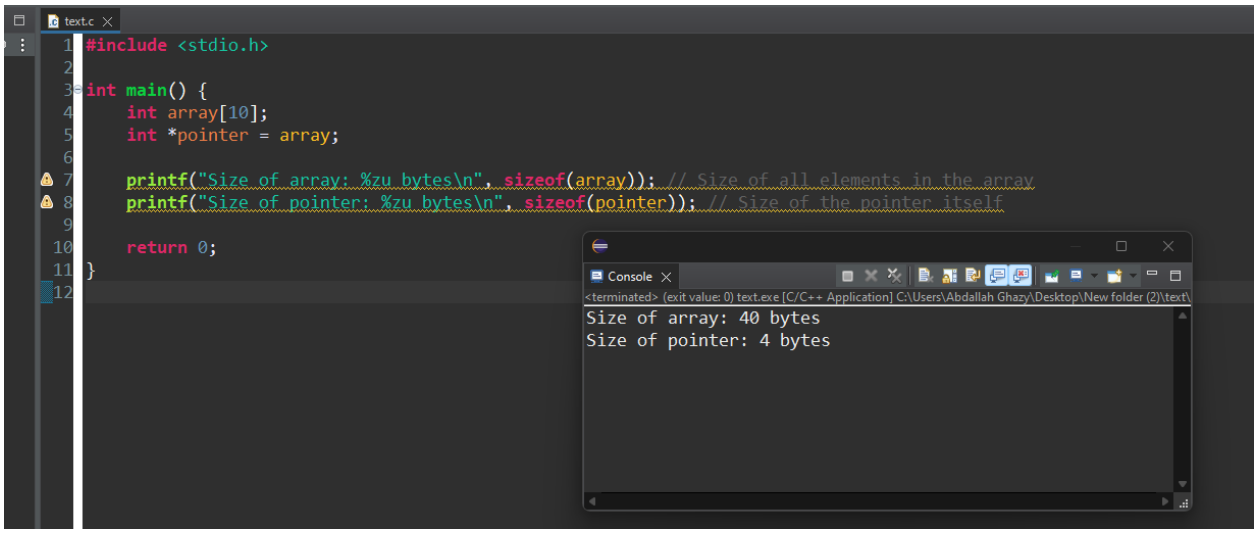
❖ Pointer Arithmetic:

Pointer arithmetic is allowed and can be used to navigate through an array. For example, pointer++ increments the pointer to point to the next element.

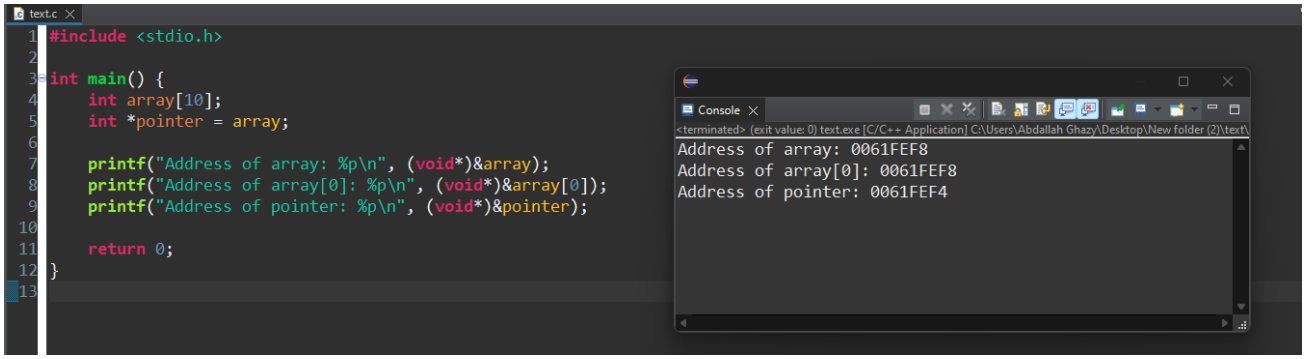
Array variables cannot be incremented or decremented. For example, **array++ is illegal**.

Examples Demonstrating Differences

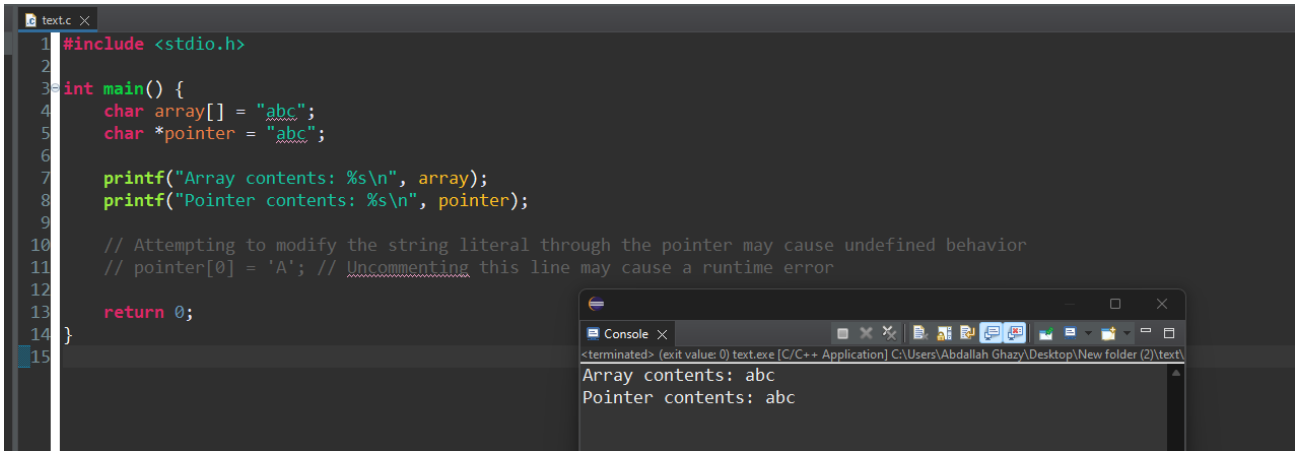
- sizeof Operator



- & Operator



- String Literal Initialization



- Assignment

```c
#include <stdio.h>

int main() {
    int array[10];
    int *pointer = array; // Legal: Assigning array to pointer

    // array = pointer; // Illegal: Array variable cannot be assigned a value

    return 0;
}
```

- Pointer Arithmetic

```c
#include <stdio.h>

int main() {
    int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *pointer = array;

    // Incrementing the pointer to navigate through the array
    for (int i = 0; i < 10; i++) {
        printf("%d ", *pointer);
        pointer++;
    }
    printf("\n");

    // Incrementing the array variable is illegal
    // array++; // This line will cause a compilation error

    return 0;
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\
0 1 2 3 4 5 6 7 8 9
```

NULL and Unassigned Pointers

NULL Pointer in C

A NULL pointer in C is a pointer that doesn't point to any of the memory locations. The NULL constant is defined in the header files stdio.h, stddef.h as well as stdlib.h.

A pointer is initialized to NULL to avoid the unpredicted behavior of a program or to prevent segmentation fault errors.

Declare and Initialize a NULL Pointer

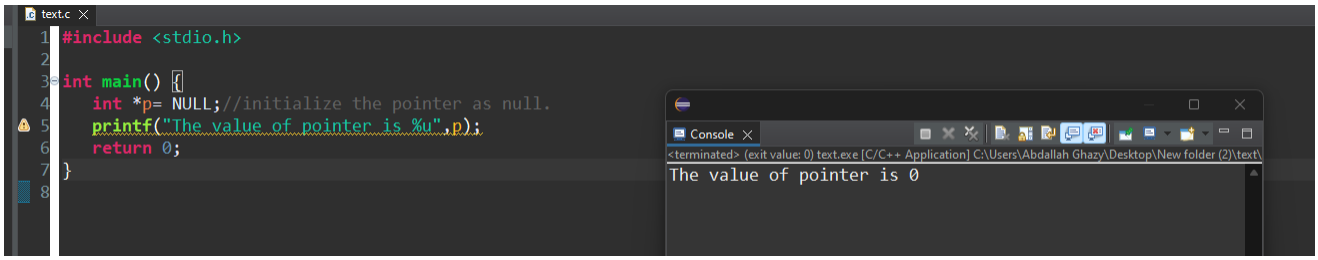
This is how you would declare and initialize a NULL pointer –

```
type *ptr = NULL;
```

Or, you can use this syntax too –

```
type *ptr = 0;
```

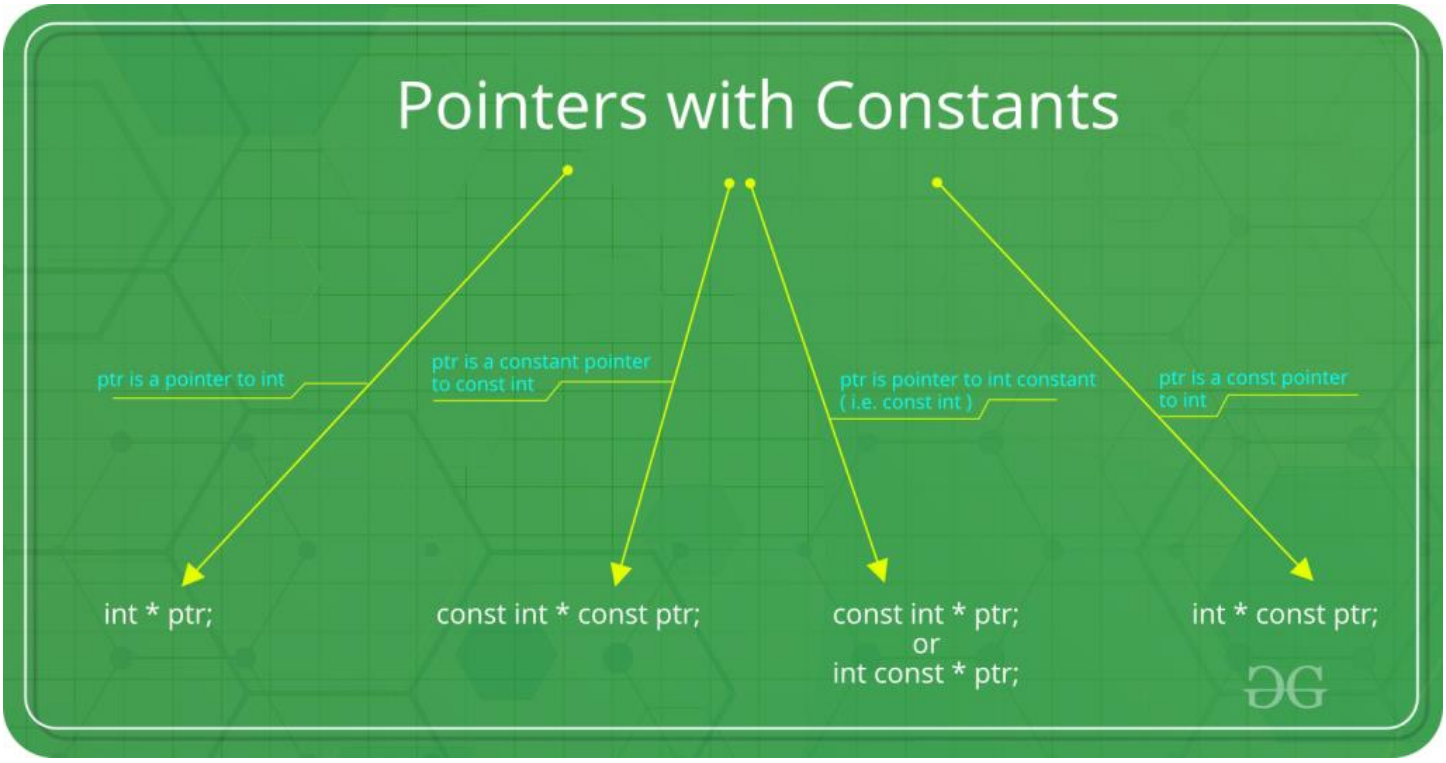
Example of a NULL Pointer

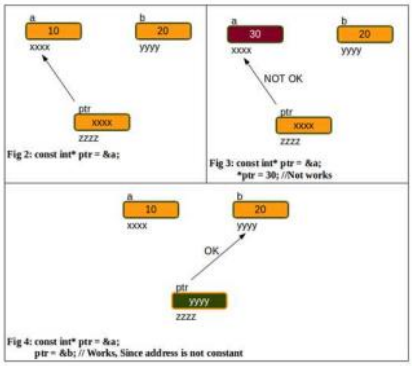


Applications of NULL Pointer

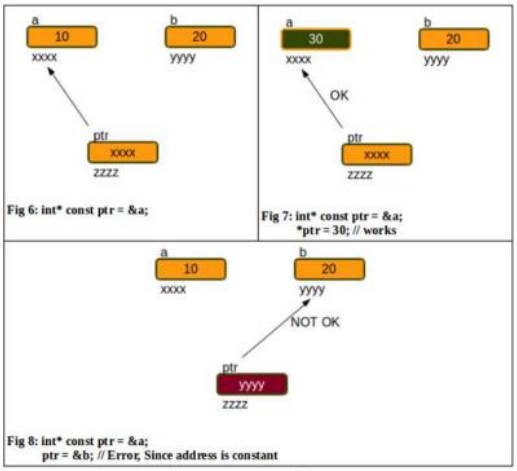
- To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
- To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- To check for a null pointer before accessing any pointer variable so that we can perform error handling in pointer-related code. For example, dereference a pointer variable only if it's not NULL.

pointers with constant






```
int* ptr = &a;
```



constant

```
int const *ptr = &a;
const int *ptr = &a;
```

Value is constant

```
int* const ptr = &a;
```

Pointer is constant

int const*

int const* is pointer to constant integer This means that the variable being declared is a pointer, pointing to a constant integer.

Effectively, this implies that the pointer is pointing to a value that shouldn't be changed.

Const qualifier doesn't affect the pointer in this scenario so the pointer is allowed to point to some other address. The first const keyword can go either side of data type, hence int const* is equivalent to const int*.

```c
#include <stdio.h>

int main(){
    const int q = 5;
    int const* p = &q;

    //Compilation error
    *p = 7;

    const int q2 = 7;

    //Valid
    p = &q2;

    return 0;
}
```

int *const

int *const is a constant pointer to integer This means that the variable being declared is a constant pointer pointing to an integer.

Effectively, this implies that the pointer shouldn't point to some other address.

Const qualifier doesn't affect the value of integer in this scenario so the value being stored in the address is allowed to change.

```c
#include <stdio.h>

int main(){
    int q = 5;
    int *const p = &q;

    //Valid
    *p = 7;

    const int q2 = 7;

    //Compilation error
    p = &q2;

    return 0;
}
```

const int* const

const int* const is a constant pointer to constant integer This means that the variable being declared is a constant pointer pointing to a constant integer.

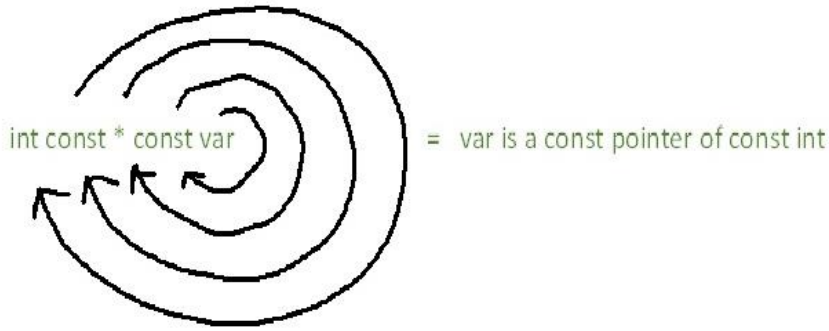
Effectively, this implies that a constant pointer is pointing to a constant value.

Hence, neither the pointer should point to a new address nor the value being pointed to should be changed. The first const keyword can go either side of data type, hence const int* const is equivalent to int const* const.

```c
#include <stdio.h>

int main(){
    const int q = 5;

    //Valid
    const int* const p = &q;

    //Compilation error
    *p = 7;

    const int q2 = 7;

    //Compilation error
    p = &q2;

    return 0;
}
```

emory Map

One way to remember the syntax (according to Bjarne Stroustrup) is the spiral rule- The rule says, start from the name of the variable and move clockwise to the next pointer or type. Repeat until expression ends.

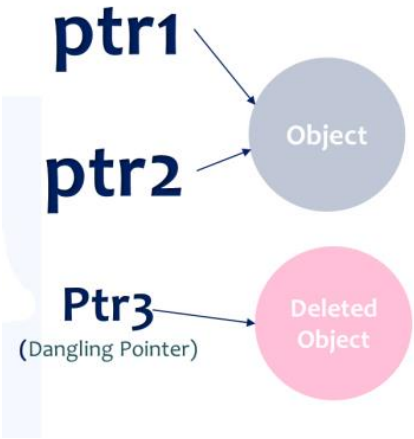


The rule can also be seen as decoding the syntax from right to left.

Hence,

Dangling Pointers in C

Dangling pointers in C is used to describe the behavior of a pointer when its target (the variable it is pointing to) has been deallocated or is no longer accessible. In other words, a dangling pointer in C is a pointer that doesn't point to a valid variable of the appropriate type.



Why Do We Get Dangling Pointers in C?

Working with a dangling pointer can lead to unpredicted behavior in a C program and sometimes it may result in the program crashing. The situation of dangling pointers can occur due to the following reasons –

- De-allocation of memory
- Accessing an out-of-bounds memory location
- When a variable goes out of scope

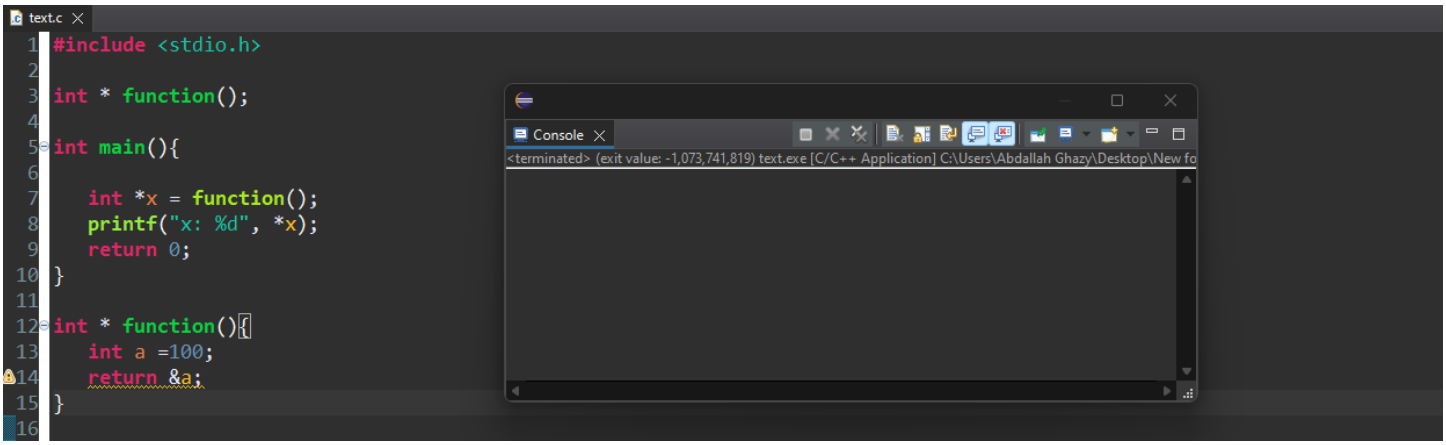
- De-allocation of Memory

```c
#include <stdio.h>

int main(){

    int *x = (int *) malloc(sizeof(int));
    *x = 100;
    printf("x: %d\n", *x);

    free (x);
    printf("x: %d\n", *x);
}
```

```
Console
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\
x: 100
x: 11605624
```

- Accessing an Out-of-Bounds Memory Location

We know that a function can return a pointer. If it returns a pointer to any local variable inside the function, it results in a dangling pointer in the outer scope, as the location it points to is no longer valid.



When compiled, the following **warning** is displayed at the "return &a" statement in the function –

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

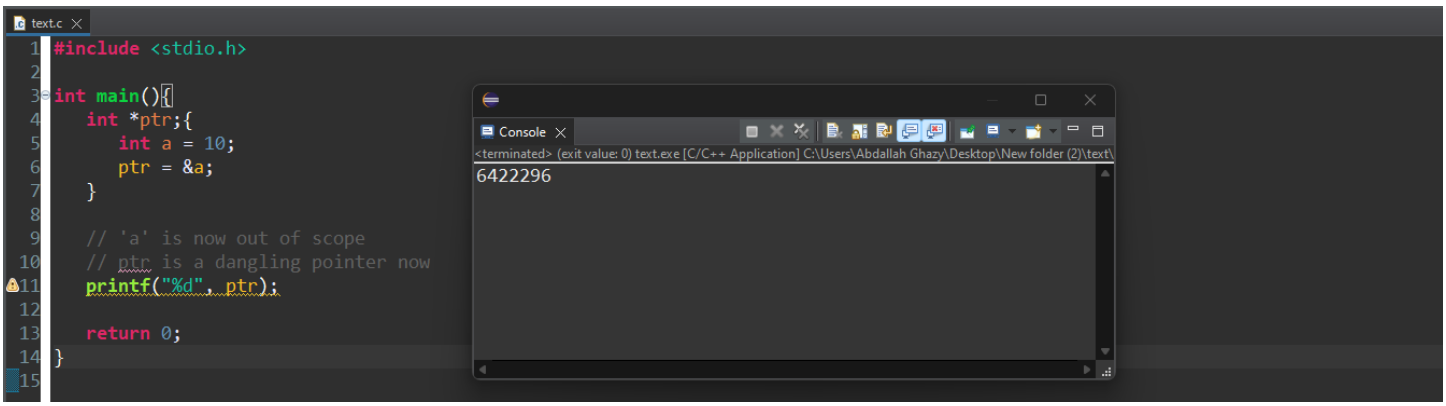
If you run the program despite the warning, you get the following **error** –

```
Segmentation fault (core dumped)
```

- When a Variable Goes Out of Scope

The same reason applies when a variable declared in an inner block is accessed outside it. In the following example, we have a variable inside a block and its address is stored in a pointer variable.

Example



How to Fix Dangling Pointers?

C doesn't have the feature of automatic garbage collection, so we need to carefully manage the dynamically allocated memory.

To fix the issue of dangling pointers or to avoid them altogether, you need to apply proper memory management and try to avoid situations where you may end up getting dangling pointers.

Here are some general guidelines that you can follow to avoid dangling pointers –

Always ensure that pointers are set to NULL after the memory is deallocated. It will clearly signify that the pointer is no longer pointing to a valid memory location.

Avoid accessing a variable or a memory location that has gone out of scope.

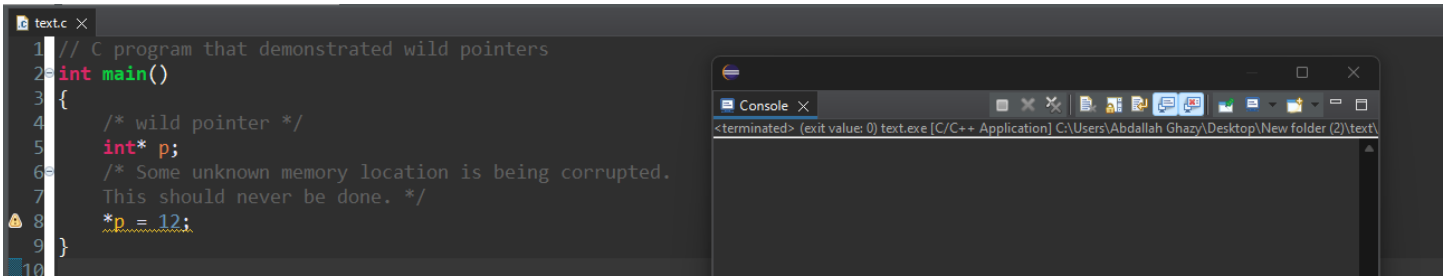
Do not return pointers to local variables because such local variables will go out of scope when the function returns.

Wild Pointer

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave unexpectedly.

Example of Wild Pointers

In the below code, p is a wild pointer.

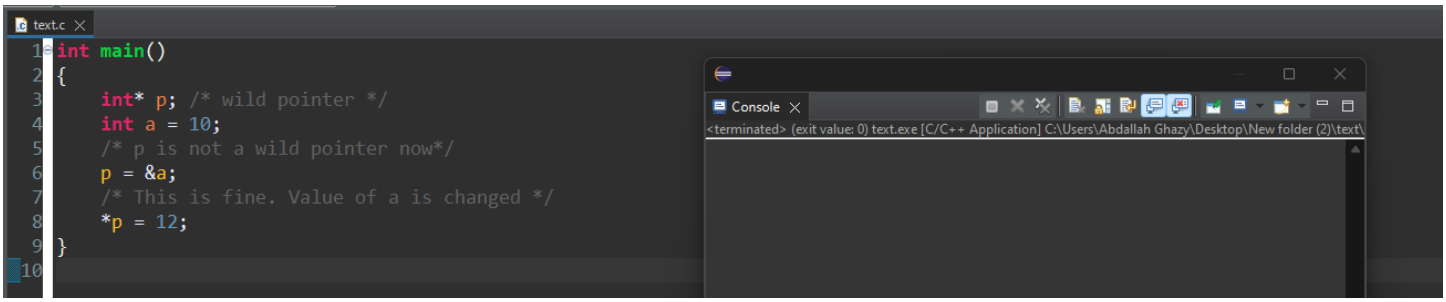


How can we avoid wild pointers?

- If a pointer points to a known variable then it's not a wild pointer.

Example

In the below program, p is a wild pointer till this points to a.



Complex Pointer

A Complex pointers contains of [] , * ,(), data type, identifier. These operators have different associativity and precedence.

() & [] have the highest precedence & associativity from left to right followed by * & Identifier having precedence 2 & associativity from right to left and Data Type having the least priority.

Operator	Precedence	Associative
(),[]	1	Left to Right
*,Identifier	2	Right to Left
Data Type	3	–

Different Terms From Table –

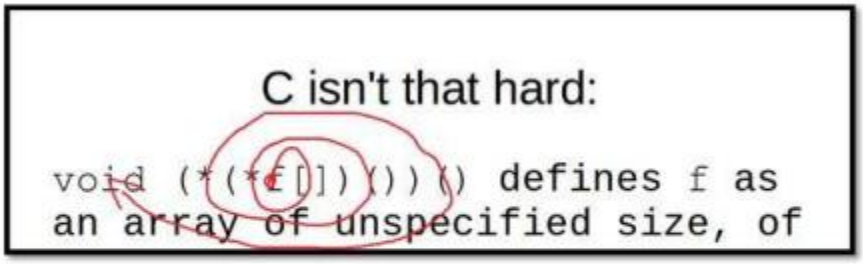
()	Bracket operator **OR** function operator.
[]	Array subscription operator
*****	Pointer operator
Identifier	**Name of Pointer Variable**
Data type	Type of Pointer

Here are some legal and illegal examples:

```
int i;                  an int
int *p;                 an int pointer (ptr to an int)
int a[];                an array of ints
int f();                a function returning an int
int **pp;               a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];            a pointer to an array of ints
int (*pf)();            a pointer to a function returning an int
int *ap[];              an array of int pointers (array of ptrs to ints)
int aa[][];             an array of arrays of ints
int af[]();             an array of functions returning an int (ILLEGAL)
int *fp();              a function returning an int pointer
int fa()[];             a function returning an array of ints (ILLEGAL)
int ff()();             a function returning a function returning an int
                                (ILLEGAL)
int ***ppp;             a pointer to a pointer to an int pointer
int (**ppa)[];          a pointer to a pointer to an array of ints
int (**ppf)();          a pointer to a pointer to a function returning an int
int *(*pap)[];          a pointer to an array of int pointers
int (*paa)[][];         a pointer to an array of arrays of ints
int (*paf)[]();         a pointer to a an array of functions returning an int
                                (ILLEGAL)
int *(*pfp)();          a pointer to a function returning an int pointer
int (*pfa)()[];         a pointer to a function returning an array of ints
```

```
                                (ILLEGAL)
int (*pff)()();          a pointer to a function returning a function
                                returning an int (ILLEGAL)
int **app[];             an array of pointers to int pointers
int (*apa[])[];          an array of pointers to arrays of ints
int (*apf[])();          an array of pointers to functions returning an int
int *aap[][];            an array of arrays of int pointers
int aaa[][][];           an array of arrays of arrays of ints
int aaf[][]();           an array of arrays of functions returning an int
                                (ILLEGAL)
int *afp[]();            an array of functions returning int pointers (ILLEGAL)
int afa[]()[];           an array of functions returning an array of ints
                                (ILLEGAL)
int aff[]()();           an array of functions returning functions
                                returning an int (ILLEGAL)
int **fpp();             a function returning a pointer to an int pointer
int (*fpa())[];          a function returning a pointer to an array of ints
int (*fpf())();          a function returning a pointer to a function
                                returning an int
int *fap()[];            a function returning an array of int pointers (ILLEGAL)
int faa()[][];           a function returning an array of arrays of ints
                                (ILLEGAL)
int faf()[]();           a function returning an array of functions
                                returning an int (ILLEGAL)
int *ffp()();            a function returning a function
                                returning an int pointer (ILLEGAL)
```

How to Read C complex pointer



f as array of pointer to function returning pointer to function returning void

```
    int *a[10];
Applying rule:


    int *a[10];       "a is"
         ^

    int *a[10];       "a is an array"
          ^^^^

    int *a[10];       "a is an array of pointers"
        ^

    int *a[10];       "a is an array of pointers to `int`".
    ^^^
```

void (*(*f[]) ()) ();

by applying the above rules:

```
   void ( *(*f[]) () ) ();       "f is"
          ^

   void ( *(*f[]) () ) ();       "f is an array"
           ^^

   void ( *(*f[]) () ) ();       "f is an array of pointers"
         ^

   void ( *(*f[]) () ) ();       "f is an array of pointers to function"
             ^^

   void ( *(*f[]) () ) ();    "f is an array of pointers to function returning pointer"
        ^


   void ( *(*f[]) () ) ();   "f is an array of pointers to function returning pointer to function"
                ^^

   void ( *(*f[]) () ) ();   "f is an array of pointers to function returning pointer to function returning `void`"
   ^^^^
```

```
void ( *( *f[] ) () ) ();
```

Another Example

```
int * (* (*fp1) (int) ) [10];
```

This can be interpreted as follows:

1. Start from the variable name ---------- `fp1`
2. Nothing to right but `)` so go left to find `*` ---------- is a pointer
3. Jump out of parentheses and encounter (`int`) --------- to a function that takes an `int` as argument
4. Go left, find `*` ---------- and returns a pointer
5. Jump put of parentheses, go right and hit `[10]` -------- to an array of 10
6. Go left find `*` ---------- pointers to
7. Go left again, find `int` ---------- `int`s.

Another Example

```
int *( *( *arr[5])())();
```

1. Start from the variable name ---------- arr
2. Go right, find array subscript ---------- is an array of 5
3. Go left, find * ---------- pointers
4. Jump out of parentheses, go right to find () ------ to functions
5. Go left, encounter * ---------- that return pointers
6. Jump out, go right, find () ---------- to functions
7. Go left, find * ---------- that return pointers
8. Continue left, find int ---------- to ints.

Dereference Pointer in C

The dereference operator is used to access and manipulate the value stored in the variable pointed by the pointer. The dereference or indirection operator (*) acts as a unary operator, and it needs a pointer variable as its operand.

Syntax

```
*pointer_variable;
```

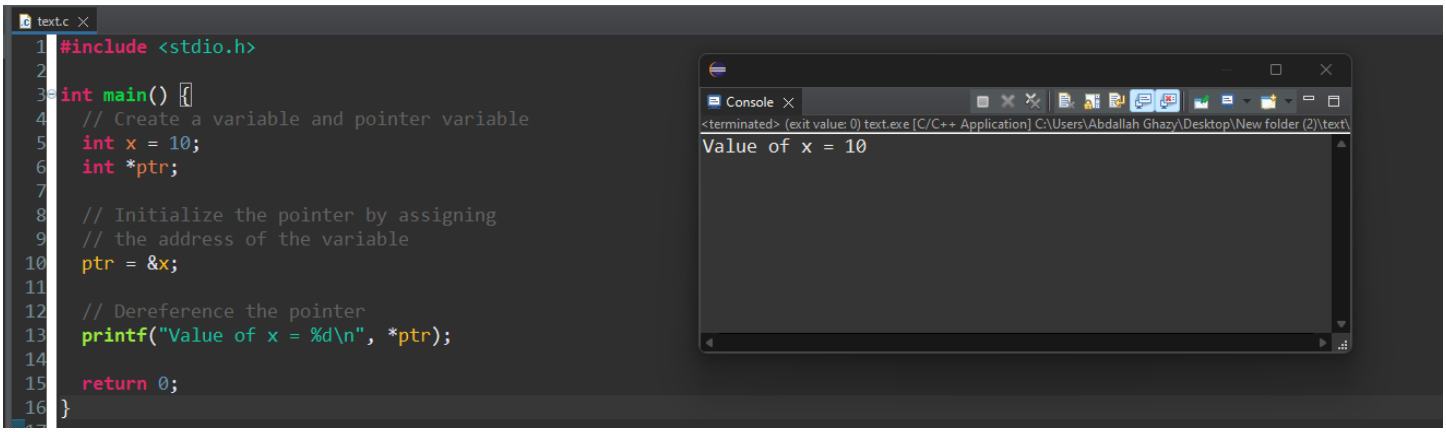
With the help of the above syntax (dereference pointer), you can get and update the value of any variable that is pointing by the pointer.

How to Dereference a Pointer?

To dereference a pointer, you need to follow the below-given steps:

- ✓ Create a variable and declare a pointer variable.
- ✓ Initialize the pointer by assigning the address of the variable.
- ✓ Now, you can dereference the pointer to get or update the value of the variable.

Example



Near, Far, and Huge Pointers in C

Concepts like near pointers, far pointers, and huge pointers were used in the C programming language to handle segmented memory models. However, these concepts are no longer relevant in modern computing environments with improved CPU architecture.

The idea of near, far, and huge pointers was implemented in 16-bit Intel architectures, in the days of the MS DOS operating system.

Near Pointer

The "near" keyword in C is used to declare a pointer that can only access memory within the current data segment. A near pointer on a 16-bit machine is a pointer that can store only 16-bit addresses.

A near pointer can only access data of a small size of about 64 kb in a given period, which is its main disadvantage. The size of a near pointer is 2 bytes.

Syntax of Near Pointer

```
<data type> near <pointer definition>
<data type> near <function definition>
```

The following statement declares a near pointer for the variable "ptr" –

```
char near *ptr;
```

Far Pointer

A far pointer is a 32-bit pointer that can access information that is outside the computer memory in a given segment.

To use this pointer, one must allocate the "sector register" to store data addresses in the segment and also another sector register must be stored within the most recent sector.

A far pointer stores both the offset and segment addresses to which the pointer is differencing. When the pointer is incremented or decremented, only the offset part is changing. The size of the far pointer is 4 bytes.

Syntax of Far Pointer

```
<data type> far <pointer definition>
<data type> far <function definition>
```

The following statements declares a far pointer for the variable "ptr" –

```
char far *s;
```

Huge Pointer

A huge pointer has the same size of 32-bit as that of a far pointer. A huge pointer can also access bits that are located outside the sector.

A far pointer is fixed and hence that part of the sector in which they are located cannot be modified in any way; however huge pointers can be modified.

In a huge pointer, both the offset and segment address is changed. That is why we can jump from one segment to another using a huge pointer. As they compare the absolute addresses, you can perform the relational operation on it. The size of a huge pointer is 4 bytes.

Syntax of Huge Pointer

Below is the syntax to declare a huge pointer –

```
</>

data_type huge* pointer_name;
```

Pointers to Remember

Remember the following points while working with near, far, and huge pointers –

A near pointer can only store till the first 64kB addresses, while a far pointer can store the address of any memory location in the RAM. A huge pointer can move between multiple memory segments.

A near pointer can only store addresses in a single register. On the other hand, a far pointer uses two registers to store segment and offset addresses. The size of a near pointer is 2 bytes, while the size of far and huge pointers is 4 bytes.

Two far pointer values can point to the same location, while in the case of huge pointers, it is not possible.

The near, far, and huge pointers were used to manage memory access based on segment registers in segmented memory architectures. Modern systems use flat memory models where memory is addressed as a single contiguous space. Modern C compilers provide better memory management techniques that don't rely on segmentation concepts.

interview questions

What is a pointer in C?

A pointer is a special variable in C language meant just to store address of any other variable or function. Pointer variables unlike ordinary variables cannot be operated with all the arithmetic operations such as '*','%' operators. It follows a special arithmetic called as pointer arithmetic.

A pointer is declared as:

int *ap;

int a = 5;

In the above two statements an integer a was declared and initialized to 5. A pointer to an integer with name ap was declared. Next before ap is used

ap=&a;

This operation would initialize the declared pointer to int. The pointer ap is now said to point to a.

Operations on a pointer:

Dereferencing operator ' * ': This operator gives the value at the address pointed by the pointer .

For example after the above C statements if we give

printf("%d",*ap);

Actual value of a that is 5 would be printed. That is because ap points to a.

Addition operator ' + ': Pointer arithmetic is different from ordinary arithmetic.

ap=ap+1; Above expression would not increment the value of ap by one, but would increment it by the number of bytes of the data type it is pointing to. Here ap is pointing to an integer variable hence ap is incremented by 2 or 4 bytes depending upon the compiler.

What are the advantages of using pointers?

Advantages:

1. Pointers allow us to pass values to functions using call by reference. This is useful when large sized arrays are passed as arguments to functions. A function can return more than one value by using call by reference.

2. Dynamic allocation of memory is possible with the help of pointers.

3. We can resize data structures. For instance, if an array's memory is fixed, it cannot be resized. But in case of an array whose memory is created out of malloc can be resized.

4. Pointers point to physical memory and allow quicker access to data.

What is the equivalent pointer expression for referring an element a[i][j][k][l], in a four dimensional array?

Consider a multidimensional array a[w][x][y][z].

In this array, a[i] gives address of a[i][0][0][0] and a[i]+j gives the address of a[i][j][0][0]

Similarly, a[i][j] gives address of a[i][j][0][0] and a[i][j]+k gives the address of a[i][j][k][0]

a[i][j][k] gives address of a[i][j][k][0] and a[i][j][k]+l gives address of a[i][j][k][l]

Hence a[i][j][k][l] can be accessed using pointers as *(a[i][j][k]+l)

where * stands for value at address and a[i][j][k]+l gives the address location of a[i][j][k][l].

Declare an array of three function pointers where each function receives two integers and returns float.

```
// Declaration of a function pointer type

typedef float (*FuncPtr)(int, int);

// Declaration of an array of three function pointers

FuncPtr funcArray[3];
```

Explain the variable assignment in the declaration

int *(*p[10])(char *, char *)

It is an array of function pointers that returns an integer pointer. Each function has two arguments which in turn are pointers to character type variable. p[0], p[1],....., p[9] are function pointers.

return type : integer pointer. p[10] : array of function pointers char * : arguments passed to the function

What is the value of sizeof(a) /sizeof(char *)

- in a code snippet:

○ char *a[4]={"sridhar","raghava","shashi","srikanth"};

Explanation: Here a[4] is an array which holds the address of strings. Strings are character arrays themselves. Memory required to store an address is 4 bits. So memory required to store 4 addresses is equal to 4*4=16 bits. char *; is a pointer variable which stores the address of a char variable. So sizeof(char *) is 4 bits. Therefore sizeof(a) /sizeof(char *) = 16/4 = 4 bytes.

What are the differences between the C statements below:

char *str = "Hello";

char arr[] = "Hello";

char *str="Hello";

"Hello" is an anonymous string present in the memory. 'str' is a pointer variable that holds the address of this string.

char arr[]="Hello";

This statement assigns space for six characters: 'H' 'e' 'l' 'l' 'o' '\0' . 'arr' is the variable name assigned to this array of characters.

- str[4] and arr[4] also have different meanings.

str[4]: adds 4 to the value of 'str' and points to the address same as value of str + 4.

arr[4]: points to the fourth element in array named 'arr'.

Whether following statements get complied or not? Explain each statement.

```
arr++;

*(arr + 1) = 's';

printf("%s",arr);
```

arr++;

'arr' is variable name of an array. A variable name can not be incremented or decremented. Hence arr++ is an invalid statement and would result in a compilation error.

***(arr+1) ='s';**

'arr' is the name of a character array that holds string "Hello". Usually, name of an array points to its base address. Hence value of arr is same as &arr[0].

arr+1 is address of the next element: &arr[1]

Character 's' is assigned to the second element in array 'arr', thereby string changes from "Hello" to "Hsllo".

printf("%s",arr);

This statement prints the string stored in character array 'arr'.

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"