C Pointers

In Short:

1. What is a Pointer?

A pointer is a variable that stores the memory address of another variable. Instead of holding a value like regular variables, pointers point to the location in memory where a value resides.

---

2. Declaration of Pointers

To declare a pointer, use the * operator.

type *pointer_name;

- type specifies the data type of the variable the pointer will point to.
- * indicates that the variable is a pointer.

Example:

int *p; // p is a pointer to an integer

float *q; // q is a pointer to a float

---

3. How Pointers Work

Consider a variable x with a value of 10. It is stored in a memory address, say 0x100.

- The **address-of operator** (&) gives the memory address of a variable.
- The **dereference operator** (*) accesses the value stored at a memory address.

```
#include <stdio.h>

int main() {
   int x = 10;
   int *p = &x;  // p holds the address of x
printf("Value of x: %d\n", x);     // Outputs 10
   printf("Address of x: %p\n", &x);  // Outputs the memory address of x
   printf("Value of x using pointer: %d\n", *p); // Outputs 10
   return 0;
}
```

Output:

Value of x: 10

Address of x: 0x100

Value of x using pointer: 10

---

4. Pointer Initialization

Pointers must be initialized before use; otherwise, they contain garbage values (random addresses).

Ways to Initialize Pointers:

1. Assign the address of an existing variable.

```
int a = 5;
int *p = &a;  // p points to a
```

2. Use NULL for an uninitialized pointer.

```
int *p = NULL;  // p points to nothing
```

---

5. Pointer Dereferencing

Dereferencing means accessing the value stored at the address the pointer is pointing to.

```
#include <stdio.h>
int main() {
   int a = 5;
   int *p = &a;  // p holds the address of a

   printf("Value of a: %d\n", *p);  // Dereferences p to get the value of a
   *p = 10;  // Modifies the value of a through the pointer
   printf("Updated value of a: %d\n", a);
   return 0;
}
```

Output:

Value of a: 5

Updated value of a: 10

---

6. Null Pointers

A null pointer is a pointer that does not point to any memory location.

```
int *p = NULL;
```

Purpose:

- Indicate an invalid or uninitialized pointer.
- Useful for error checking.

```
if (p == NULL) {
   printf("Pointer is null.\n");
}
```

---

7. Pointer Arithmetic

Pointers can perform arithmetic to traverse memory locations. The arithmetic depends on the size of the data type the pointer points to.

Operation	Description
p++ or p--	Moves the pointer to the next or previous memory location.
p + n or p - n	Moves the pointer n locations forward or backward.

Example:

```
#include <stdio.h>
int main() {
   int arr[] = {10, 20, 30};
   int *p = arr;  // Points to the first element of the array

   printf("Value at p: %d\n", *p);      // Outputs 10
   printf("Value at p+1: %d\n", *(p+1)); // Outputs 20
   printf("Value at p+2: %d\n", *(p+2)); // Outputs 30
   return 0;
}
```

Output:

```
Value at p: 10
Value at p+1: 20
Value at p+2: 30
```

---

8. Dynamic Memory Allocation with Pointers

C uses pointers for dynamic memory allocation via functions like malloc, calloc, realloc, and free.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int *p = (int *)malloc(sizeof(int) * 5);  // Allocates memory for 5 integers

   for (int i = 0; i < 5; i++) {
      p[i] = i + 1;  // Assign values to allocated memory
   }

   for (int i = 0; i < 5; i++) {
      printf("%d ", p[i]);  // Access values using pointer
   }

   free(p);  // Frees the allocated memory
   return 0;
}
```

Output:

```
1 2 3 4 5
```

---

9. Pointers and Functions

Pointers can be used to pass variables by reference to a function, allowing the function to modify the original value.

Example:

```
#include <stdio.h>
void increment(int *p) {
   (*p)++;
}

int main() {
   int a = 10;
```

```
    increment(&a);  // Pass address of a
    printf("Value of a: %d\n", a);  // Outputs 11
    return 0;
}
```

---

10. Pointers and Arrays

An array name is essentially a pointer to the first element of the array.

Example:

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3};
    int *p = arr;  // Points to the first element of arr

    for (int i = 0; i < 3; i++) {
        printf("%d ", *(p + i));  // Access array elements using pointer
    }
    return 0;
}
```

Output:

```
1 2 3
```

---

11. Pointers and Strings

Pointers can manipulate strings (character arrays).

Example:

```
#include <stdio.h>
int main() {
    char str[] = "Hello";
    char *p = str;

    while (*p != '\0') {
        printf("%c", *p);  // Prints each character
```

```
    p++;
  }
  return 0;
}
```

Output:

Hello

---

12. Advantages of Pointers

1. **Efficient Memory Usage**: Pointers allow efficient handling of data and memory.
2. **Dynamic Memory Allocation**: Enable allocating and freeing memory at runtime.
3. **Data Structures**: Essential for linked lists, trees, and other structures.
4. **Pass-by-Reference**: Modify variables directly via function calls.

---

13. Common Pitfalls

1. **Dereferencing Null Pointers**: Leads to segmentation faults.
2. **Memory Leaks**: Forgetting to free dynamically allocated memory.
3. **Pointer Arithmetic Errors**: Can lead to undefined behavior.
4. **Dangling Pointers**: Using a pointer after the memory it points to is freed.

In detail:

Pointer is a variable that stores the memory address of another variable

Instead of holding a direct value, it holds the address where the value is stored in memory. There are **2 important operators** that we will use in pointers concepts i.e.

- **Dereferencing operator**(*) used to declare pointer variable and access the value stored in the address.
- **Address operator(&)** used to returns the address of a variable or to access the address of a variable to a pointer.

Consider the following example to define a pointer which stores the address of an integer.

1. **int** n = 10;
2. **int*** p = &n; // Variable p of type pointer is pointing to the address of the variable n of type in teger.

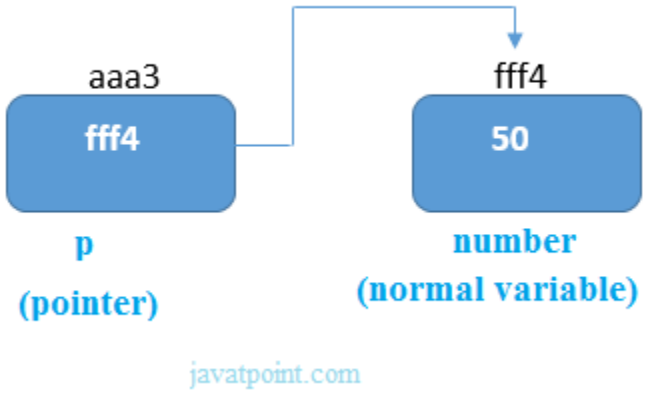
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. **int** *a;//pointer to int
2. **char** *c;//pointer to char

Pointer Example:

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. **int** *p;
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
7. printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointe r therefore if we print *p, we will get the value stored at the address contained by p.
8. **return** 0;
9. }

Output

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

Pointer to array

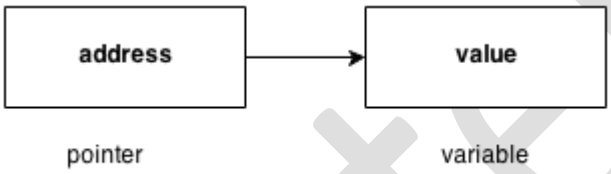
1. **int** arr[10];
2. **int** *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr .

Pointer to a function

1. **void** show (**int**);
2. **void**(*p)(**int**) = &display; // Pointer p is pointing to the address of a function

Pointer to structure

1. **struct** st {
2. **int** i;
3. **float** f;
4. }ref;
5. **struct** st *p = &ref;



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. printf("value of number is %d, address of number is %u",number,&number);
5. return 0;
6. }
```

Output

value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

*int *p=NULL;*

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
1. #include<stdio.h>
2. int main(){
3. int a=10,b=20,*p1=&a,*p2=&b;
4.
5. printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
6. *p1=*p1+*p2;
7. *p2=*p1-*p2;
8. *p1=*p1-*p2;
9. printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
10.
11. return 0;
12. }
```

Output

*Before swap: *p1=10 *p2=20*

*After swap: *p1=20 *p2=10*

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
(), []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

Here,we must notice that,

- (): This operator is a bracket operator used to declare and define the function.
- []: This operator is an array subscript operator
- * : This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

How to read the pointer: int (*p)[10].

To read the pointer, we must see that () and [] have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to ().

Inside the bracket (), pointer operator * and pointer name (identifier) p have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to p, and the second priority goes to *.

Assign the 3rd priority to [] since the data type has the last precedence. Therefore the pointer will look like following.

- char -> 4
- * -> 2
- p -> 1
- [10] -> 3

The pointer will be read as p is a pointer to an array of integers of size 10.

Example

How to read the following pointer?

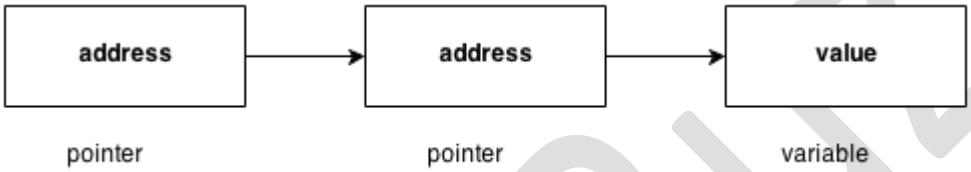
1. **int** (*p)(**int** (*)[2], **int** (*)**void**))

Explanation

This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the second parameter as the pointer to a function which parameter is void and return type is the integer.

C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



The syntax of declaring a double pointer is given below.

1. **int** **p; // pointer to a pointer which is pointing to an integer.

Consider the following example.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int a = 10;
5.     int *p;
6.     int **pp;
7.     p = &a; // pointer p is pointing to the address of a
8.     pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
9.     printf("address of a: %x\n",p); // Address of a will be printed
10.    printf("address of p: %x\n",pp); // Address of p will be printed
11.    printf("value stored at p: %d\n",*p); // value stoted at the address contained by p i.e. 10 w
       ill be printed
12.    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the po
       inter stoyred at pp
13. }
```

Output

address of a: d26a8734

address of p: d26a8738

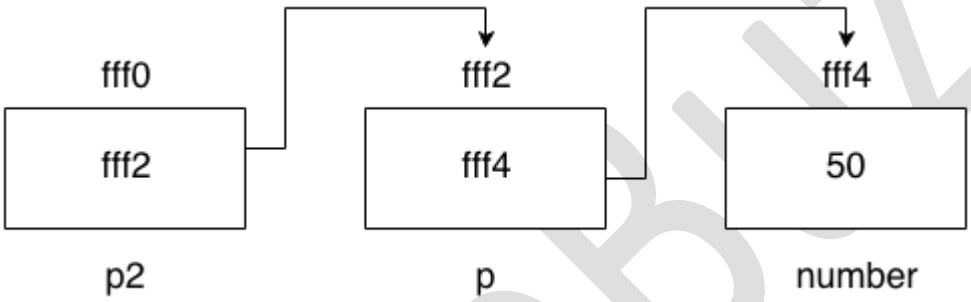
value stored at p: 10

value stored at pp: 10

C double pointer example:

Let's see an example where one pointer points to the address of another pointer.

Backward Skip 10sPlay VideoForward Skip 10s



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. int **p2;//pointer to pointer
6. p=&number;//stores the address of number variable
7. p2=&p;
8. printf("Address of number variable is %x \n",&number);
9. printf("Address of p variable is %x \n",p);
10. printf("Value of *p variable is %d \n",*p);
11. printf("Address of p2 variable is %x \n",p2);
12. printf("Value of **p2 variable is %d \n",*p);
13. return 0;
```

```
14. }
```

Output

Address of number variable is fff4

Address of p variable is fff4

*Value of *p variable is 50*

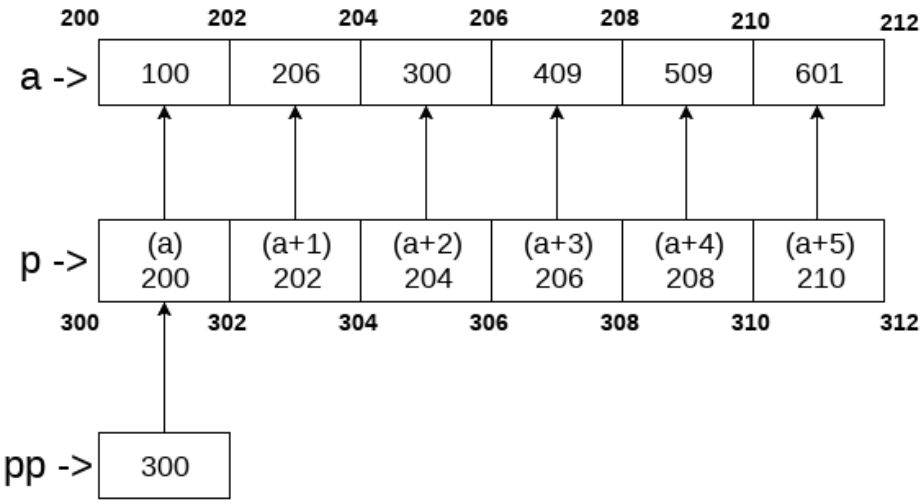
Address of p2 variable is fff2

*Value of **p variable is 50*

Q. What will be the output of the following program?

```
1.  #include<stdio.h>
2.  void main ()
3.  {
4.      int a[10] = {100, 206, 300, 409, 509, 601}; //Line 1
5.      int *p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line 2
6.      int **pp = p; //Line 3
7.      pp++; // Line 4
8.      printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 5
9.      *pp++; // Line 6
10.     printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 7
11.     ++*pp; // Line 8
12.     printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 9
13.     ++**pp; // Line 10
14.     printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 11
15. }
```

Explanation:



To access a[0] ⟶ a[0] = * (a) = *p[0] = **(p+0) = **(pp+0) = 100

In the above question, the pointer arithmetic is used with the double pointer. An array of 6 elements is defined which is pointed by an array of pointer p. The pointer array p is pointed by a double pointer pp. However, the above image gives you a brief idea about how the memory is being allocated to the array a and the pointer array p. The elements of p are the pointers that are pointing to every element of the array a. Since we know that the array name contains the base address of the array hence, it will work as a pointer and can the value can be traversed by using *(a), *(a+1), etc. As shown in the image, a[0] can be accessed in the following ways.

- a[0]: it is the simplest way to access the first element of the array
- *(a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
- *p[0]: if a[0] is to be accessed by using a pointer p to it, then we can use indirection operator (*) on the first element of the pointer array p, i.e., *p[0].
- **(pp): as pp stores the base address of the pointer array, *pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. **p will give the actual value of the first element of the integer array.

Coming to the program, Line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210. Let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array, i.e., 300. Line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

Line number 5 contains an expression which prints three values, i.e., pp - p, *pp - a, **pp. Let's calculate them each one of them.

- pp = 302, p = 300 => pp-p = (302-300)/2 => pp-p = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, a = 200 => *pp - a = 202 - 200 = 2/2 = 1, i.e., 1 will be printed.

- pp = 302, *pp = 202, *(*pp) = 206, i.e., 206 will be printed.

Therefore as the result of line 5, The output 1, 1, 206 will be printed on the console. On line 6, *pp++ is written. Here, we must notice that two unary operators * and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression *pp++ can be rewritten as (*(pp++)). Since, pp = 302 which will now become, 304. *pp will give 204.

On line 7, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, a = 200 => *pp-a = (204 - 200)/2 = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, *(*pp) = 300, i.e., 300 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. On line 8, ++*pp is written. According to the rule of associativity, this can be rewritten as, (++(*(pp))). Since, pp = 304, *pp = 204, the value of *pp = *(p[2]) = 206 which will now point to a[3].

On line 9, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- pp = 304, *pp = 206, *(*pp) = 409, i.e., 409 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. On line 10, ++**pp is writen. according to the rule of associativity, this can be rewritten as, (++(*(*(pp)))). pp = 304, *pp = 206, **pp = 409, ++**pp => *pp = *pp + 1 = 410. In other words, a[3] = 410.

On line 11, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- On line 8, **pp = 410.

Therefore as the result of line 9, the output 2, 3, 410 will be printed on the console.

At last, the output of the complete program will be given as:

Output

1 1 206

2 2 300

2 3 409

2 3 410

Pointer Arithmetic in C:

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1. new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+1;
```

```
8. printf("After increment: Address of p variable is %u \n",p); // in our case, p will get increment
   ed by 4 bytes.
9. return 0;
10. }
```

Output

Address of p variable is 3214864300

After increment: Address of p variable is 3214864304

Traversing an array by using pointer

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int arr[5] = {1, 2, 3, 4, 5};
5.     int *p = arr;
6.     int i;
7.     printf("printing array elements...\n");
8.     for(i = 0; i< 5; i++)
9.     {
10.        printf("%d  ",*(p+i));
11.    }
12. }
```

Output

printing array elements...

1 2 3 4 5

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

```
1. new_address= current_address - i * size_of(data type)
```

Test it Now

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
1. #include <stdio.h>
2. void main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-1;
8. printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immidia
   te previous location.
9. }
```

Output

Address of p variable is 3214864300

After decrement: Address of p variable is 3214864296

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

```
1. new_address= current_address + (number * size_of(data type))
```

32-bit

For 32-bit int variable, it will add 2 * number.

64-bit

For 64-bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3;  //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u \n",p);
```

```
9. return 0;
10. }
```

Output

Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

```
1. new_address= current_address - (number * size_of(data type))
```

32-bit

For 32-bit int variable, it will subtract 2 * number.

64-bit

For 64-bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-3; //subtracting 3 from pointer variable
8. printf("After subtracting 3: Address of p variable is %u \n",p);
9. return 0;
10. }
```

Output

Address of p variable is 3214864300

After subtracting 3: Address of p variable is 3214864288

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

Consider the following example to subtract one pointer from an another.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i = 100;
5.     int *p = &i;
6.     int *temp;
7.     temp = p;
8.     p = p + 3;
9.     printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
10. }
```

Output

Pointer Subtraction: 1030585080 - 1030585068 = 3

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```
1. #include<stdio.h>
2. int addition ();
3. int main ()
4. {
5.     int result;
6.     int (*ptr)();
7.     ptr = &addition;
8.     result = (*ptr)();
9.     printf("The sum is %d",result);
10. }
11. int addition()
12. {
13.     int a, b;
14.     printf("Enter two numbers?");
15.     scanf("%d %d",&a,&b);
16.     return a+b;
17. }
```

Output

Enter two numbers?10 15

The sum is 25

Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

```
1. #include<stdio.h>
2. int show();
3. int showadd(int);
4. int (*arr[3])();
```

```
5.  int (*(*ptr)[3])();
6.
7.  int main ()
8.  {
9.      int result1;
10.     arr[0] = show;
11.     arr[1] = showadd;
12.     ptr = &arr;
13.     result1 = (**ptr)();
14.     printf("printing the value returned by show : %d",result1);
15.     (*(*ptr+1))(result1);
16. }
17. int show()
18. {
19.     int a = 65;
20.     return a++;
21. }
22. int showadd(int b)
23. {
24.     printf("\nAdding 90 to the value returned by show: %d",b+90);
25. }
```

Output

printing the value returned by show : 65

Adding 90 to the value returned by show: 155

sizeof() operator in C:

The **sizeof()** operator is commonly used in C. It determines the size of the expression or the data type specified in the number of char-sized storage units. The **sizeof()** operator contains a single operand which can be either an expression or a data typecast where the cast is data type enclosed within parenthesis. The data type cannot only be primitive data types such as integer or floating data types, but it can also be pointer data types and compound data types such as unions and structs.

Need of sizeof() operator

Mainly, programs know the storage size of the primitive data types. Though the storage size of the data type is constant, it varies when implemented in different platforms. For example, we dynamically allocate the array space by using **sizeof()** operator:

```
1. int *ptr=malloc(10*sizeof(int));
```

In the above example, we use the sizeof() operator, which is applied to the cast of type int. We use **malloc()** function to allocate the memory and returns the pointer which is pointing to this allocated memory. The memory space is equal to the number of bytes occupied by the int data type and multiplied by 10.

Note:
The output can vary on different machines such as on 32-bit operating system will show different output, and the 64-bit operating system will show the different outputs of the same data types.

The **sizeof()** operator behaves differently according to the type of the operand.

- **Operand is a data type**
- **Operand is an expression**

When operand is a data type.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int x=89;    // variable declaration.
5.     printf("size of the variable x is %d", sizeof(x));  // Displaying the size of ?x? variable.
6.     printf("\nsize of the integer data type is %d",sizeof(int)); //Displaying the size of integer da
   ta type.
7.     printf("\nsize of the character data type is %d",sizeof(char)); //Displaying the size of charac
   ter data type.
8.
9.     printf("\nsize of the floating data type is %d",sizeof(float)); //Displaying the size of floating
   data type.
10. return 0;
11. }
```

In the above code, we are printing the size of different data types such as int, char, float with the help of **sizeof()** operator.

Output :

```
size of the variable x is 4
size of the integer data type is 4
size of the character data type is 1
size of the floating data type is 4

...Program finished with exit code 0
Press ENTER to exit console.
```

When operand is an expression

```
1. #include <stdio.h>
2. int main()
3. {
4.   double i=78.0; //variable initialization.
5.   float j=6.78; //variable initialization.
6.   printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of the expression (i+j)
     .
7.   return 0;
8. }
```

In the above code, we have created two variables 'i' and 'j' of type double and float respectively, and then we print the size of the expression by using **sizeof(i+j)** operator.

Output

size of (i+j) expression is : 8

Handling Arrays and Structures

The ***sizeof() operator*** is highly helpful when working with arrays and structures in addition to the above usage cases. ***Contiguous blocks*** of memory are known as ***arrays***, and understanding their size is crucial for a few tasks.

For example:

```
1. #include <stdio.h>
2.
3. int main() {
4.     int arr[] = {1, 2, 3, 4, 5};
5.     int arrSize = sizeof(arr) / sizeof(arr[0]);
6.
7.     printf("Size of the array arr is: %d\n", sizeof(arr));
8.     printf("Number of elements in arr is: %d\n", arrSize);
```

```
9.
10.     return 0;
11. }
```

Output

Size of the array arr is: 20

Number of elements in arr is: 5

Sizeof(arr) returns the array's overall size in bytes, whereas ***sizeof(arr[0])*** returns the array's smallest element's size. The number of items in the array is determined by dividing the overall size by the size of a ***single element (arrSize)***. By using this technique, the code will continue to be ***flexible*** in the face of changing array sizes.

Similarly, you can use the ***sizeof() operator*** to figure out the size of structures:

```
1.  #include <stdio.h>
2.
3.  struct Person {
4.      char name[30];
5.      int age;
6.      float salary;
7.  };
8.
9.  int main() {
10.     struct Person p;
11.
12.     printf("Size of the structure Person is: %d bytes\n", sizeof(p));
13.
14.     return 0;
15. }
```

Output

Size of the structure Person is: 40 bytes

Allocation of dynamic memory and pointer arithmetic

Other applications of the ***sizeof() operator*** include ***pointer arithmetic*** and **dynamic memory allocation**. Knowing the size of data types becomes essential when working with ***arrays*** and ***pointers*** for correct memory allocation and element access.

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

```c
#include <stdlib.h>

int main() {
    int *ptr;
    int numElements = 5;

    ptr = (int*)malloc(numElements * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numElements; i++) {
        ptr[i] = i + 1;
    }

    printf("Dynamic array elements: ");
    for (int i = 0; i < numElements; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr); // Release allocated memory.

    return 0;
}
```

Output

Dynamic array elements: 1 2 3 4 5

Explanation:

In this example, a size ***numElements integer*** array has a memory that is dynamically allocated. ***numElements * sizeof(int)*** bytes represent the total amount of memory allocated. By doing this, the array is guaranteed to have enough room to accommodate the desired amount of integers.

Sizeof() for Unions

Unions and the ***sizeof() operator*** are compatible. ***Unions*** are comparable to ***structures,*** except only one member can be active at once, and all its members share memory.

```
1.  #include <stdio.h>
2.
3.  union Data {
4.      int i;
5.      float f;
6.      char str[20];
7.  };
8.
9.  int main() {
10.     union Data data;
11.
12.     printf("Size of the union Data is: %d bytes\n", sizeof(data));
13.
14.     return 0;
15. }
```

Output

Size of the union Data is: 20 bytes

The ***sizeof() operator*** is extremely important since it's essential for ***memory management***, ***portability***, and ***effective data handling***. The ***sizeof() operator*** is crucial in C for the reasons listed in the list below:

Memory Allocation: When working with ***arrays*** and ***dynamic memory allocation***, the ***sizeof() operator*** is frequently used in memory allocation. Knowing the size of ***data types*** when allocating memory for arrays or structures guarantees that the correct amount of memory is reserved, reducing ***memory overflows*** and improving memory utilization.

Portability: Since C is a ***popular programming language***, code frequently has to operate on several systems with differing architectures and ***data type sizes***. As it specifies the size of data types at compile-time, the ***sizeof() operator*** aids in designing portable code by enabling programs to adapt automatically to various platforms.

Pointer Arithmetic: When dealing with pointers, the ***sizeof() operator*** aids in figuring out ***memory offsets***, allowing accurate movement within ***data structures, arrays***, and other memory regions. It is extremely helpful when iterating across arrays or dynamically allocated memory.

Handling Binary Data: The ***sizeof() operator*** guarantees that the right amount of data is read or written when working with binary data or files, eliminating mistakes brought on by inaccurate data size assumptions.

Unions and Structures: The ***sizeof() operator*** is essential when managing ***structures*** and ***unions***, especially when utilizing them to build complicated data structures. ***Memory allocation*** and access become effective and error-free when you are aware of the size of structures and unions.

Safe Buffer Management: The ***sizeof() operator*** helps make sure that the buffer is big enough to hold the data being processed while working with character ***arrays (strings)***, preventing ***buffer overflows*** and ***potential security flaws***.

Data Serialization and Deserialization: The ***sizeof() operator*** guarantees that the right amount of data is handled, maintaining ***data integrity*** throughout ***data transfer*** or storage, in situations where data needs to be serialized (converted to a byte stream) or deserialized (retrieved from a byte stream).

Code Improvement: Knowing the size of various data formats might occasionally aid in ***code optimization***. For instance, it enables the compiler to more effectively align data structures, reducing memory waste and enhancing cache performance.

Sizeof() Operator Requirement in C

The ***sizeof() operator*** is a key component in C programming due to its need in different elements of memory management and data processing. Understanding ***data type*** sizes is essential for ***effectively allocating memory***, especially when working with arrays and dynamic memory allocation. By ensuring that the appropriate amount of memory is reserved, this information helps to avoid memory overflows and optimize memory use. The ***sizeof() operator*** is also essential for creating ***portable code***, which may execute without ***error*** on several systems with differing architectures and data type sizes.

The program can adapt to many platforms without the need for manual modifications since it supplies the size of data types at compile-time. Additionally, the ***sizeof() operator*** makes it possible to navigate precisely around data structures and arrays while working with pointers, facilitating safe and effective pointer arithmetic. Another application for the ***sizeof() operator*** is handling ***unions*** and ***structures***. It ensures precise memory allocation and access within intricate ***data structures***, preventing mistakes and inefficiencies. The ***sizeof() operator*** is a basic tool that enables C programmers to develop effective, portable, and resilient code while optimizing performance and data integrity. It ensures ***safe buffer management*** and makes data serialization and deserialization easier.

Conclusion:

In summary, the ***C sizeof() operator*** is a useful tool for calculating the size of many sorts of objects, including ***data types, expressions, arrays, structures, unions***, and more. As it offers the size of data types at compile-time, catering to multiple platforms and settings, it enables programmers to create portable and flexible code. Developers may effectively handle ***memory allocation, pointer arithmetic***, and ***dynamic memory allocation*** in their programs by being aware of the storage needs of various data types.

When working with ***arrays*** and ***structures***, the ***sizeof() operator*** is very helpful since it ensures proper ***memory allocation*** and makes it simple to retrieve elements. Additionally, it

facilitates ***pointer arithmetic***, making it simpler to move between memory regions. However, because of operator precedence, programmers should be cautious when utilizing complicated expressions with ***sizeof() operator***.

Overall, learning the ***sizeof() operator*** equips C programmers to create stable and adaptable software solutions by enabling them to write efficient, dependable, and platform-independent code.