

Pointers:

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

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

```
int n = 10;  
int* p = &n; // Variable p of type pointer is pointing to the address of the variable n  
              of type integer.
```

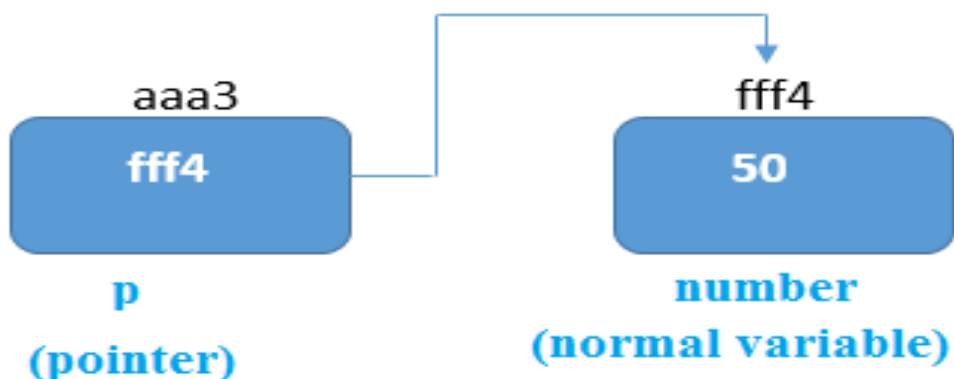
Declaring a pointer:

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

```
int *a;//pointer to int  
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.

Example of pointer:

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

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array:

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

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.

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

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:

```
#include<stdio.h>
int main(){
int a=10,b=20,*p1=&a,*p2=&b;
printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
return 0;
}
```

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.

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.

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.

syntax of declaring a double pointer:

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

Example of double pointer:

```
#include<stdio.h>
void main ()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a; // pointer p is pointing to the address of a
    pp = &p; // pointer pp is a double pointer pointing to the
              address of pointer p
    printf("address of a: %x\n",p); // Address of a will be printed
    printf("address of p: %x\n",pp); // Address of p will be printed
    printf("value stored at p: %d\n",*p); // value stored at the address
                                          contained by p i.e. 10 will be printed
    printf("value stored at pp: %d\n",**pp); // value stored at the address
                                          contained by the pointer stored at pp
}
```

Output:

```
address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10
```

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. $\text{new_address} = \text{current_address} + i * \text{sizeof}(\text{data type})$

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.

Example of incrementing pointer variable on 64-bit architecture:

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
```

```

p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p will get
incremented by 4 bytes.
return 0;
}

```

Output:

```

Address of p variable is 3214864300
After increment: Address of p variable is 3214864304

```

Traversing an array by using pointer:

```

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

```

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. $\text{new_address} = \text{current_address} - i * \text{sizeof}(\text{data type})$

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.

Example of decrementing pointer variable on 64-bit OS:

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

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. $\text{new_address} = \text{current_address} + (\text{number} * \text{sizeof}(\text{data type}))$

32-bit

For 32-bit int variable, it will add $2 * \text{number}$.

64-bit

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

Example of adding value to pointer variable on 64-bit architecture:

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3; //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

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., $4 \times 3 = 12$ 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., $2 \times 3 = 6$. 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. $\text{new_address} = \text{current_address} - (\text{number} * \text{sizeof}(\text{data type}))$

32-bit

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

64-bit

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

Example of subtracting value from the pointer variable on 64-bit architecture:

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

Output:

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

Illegal arithmetic with pointers:

There are various operations which cannot 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

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:

```
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:

```
#include <stdio.h>

int main()

{

int x=89; // variable declaration.
```

```
printf("size of the variable x is %d", sizeof(x)); // Displaying the size of ?  
x? variable.
```

```
printf("\nsize of the integer data type is %d",sizeof(int)); //Displaying the  
size of integer data type.
```

```
printf("\nsize of the character data type is %d",sizeof(char)); //Displaying  
the size of character data type.
```

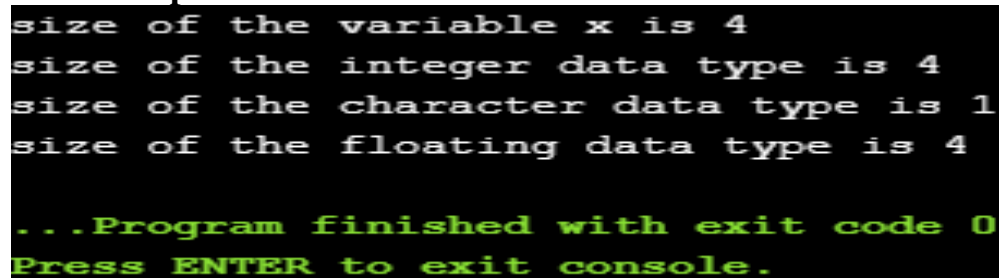
```
printf("\nsize of the floating data type is %d",sizeof(float)); //Displaying t  
he size of floating data type.
```

```
return 0;
```

```
}
```

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:

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

```
int main()
```

```
{
```

```
double i=78.0; //variable initialization.
```

```
float j=6.78; //variable initialization.
```

```
printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of  
the expression (i+j).
```

```
return 0;
```

```
}
```

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

Dynamic memory allocation in C:

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Difference between static memory allocation and dynamic memory allocation:

static memory allocation

- 1.Memory is allocated at compile time.
- 2.Memory can't be increased while executing program.
- 3.Used in array.

dynamic memory allocation

- 1.Memory is allocated at run time.
- 2.Memory can be increased while executing program.
- 3.Used in linked list.

malloc()

allocates single block of requested memory.

calloc()

allocates multiple block of requested memory.

realloc()

reallocates the memory occupied by malloc() or calloc() functions.

free()

frees the dynamically allocated memory.

malloc() function in C:

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function:

1. ptr=(cast-type*)malloc(byte-size)

Example of malloc() function:

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.   int n,i,*ptr,sum=0;
5.   printf("Enter number of elements: ");
6.   scanf("%d",&n);
7.   ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
8.   if(ptr==NULL)
9.   {
10.    printf("Sorry! unable to allocate memory");
11.    exit(0);
12.  }
13.  printf("Enter elements of array: ");
14.  for(i=0;i<n;++i)
15.  {
16.    scanf("%d",ptr+i);
17.    sum+=*(ptr+i);
18.  }
19.  printf("Sum=%d",sum);
20.  free(ptr);
21. return 0;
22. }
```

Output:

Enter elements of array: 3

Enter elements of array:

10

10

10

Sum=30

calloc() function in C:

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Let's see the example of calloc() function.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.   int n,i,*ptr,sum=0;
5.   printf("Enter number of elements: ");
6.   scanf("%d",&n);
7.   ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
8.   if(ptr==NULL)
9.   {
10.    printf("Sorry! unable to allocate memory");
11.    exit(0);
12.  }
13.  printf("Enter elements of array: ");
14.  for(i=0;i<n;++i)
15.  {
16.    scanf("%d",ptr+i);
17.    sum+=*(ptr+i);
18.  }
19.  printf("Sum=%d",sum);
20.  free(ptr);
```



```
21.return 0;  
22.}
```

Output:

```
Enter elements of array: 3  
Enter elements of array:  
10  
10  
10  
Sum=30
```

realloc() function in C:

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

The syntax of realloc() function:

```
1. ptr=realloc(ptr, new-size)
```

free() function in C:

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

The syntax of free() function:

```
free(ptr)
```