UNIT-2/POINTERS

C 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.

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

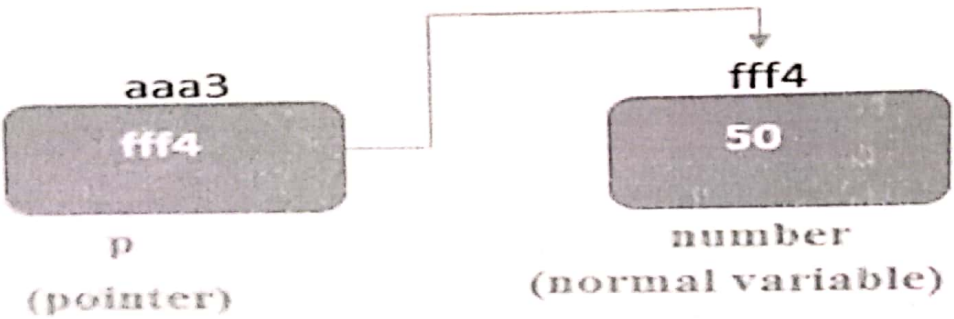
Declaring a pointer

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

```
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 t
   o dereference a pointer 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

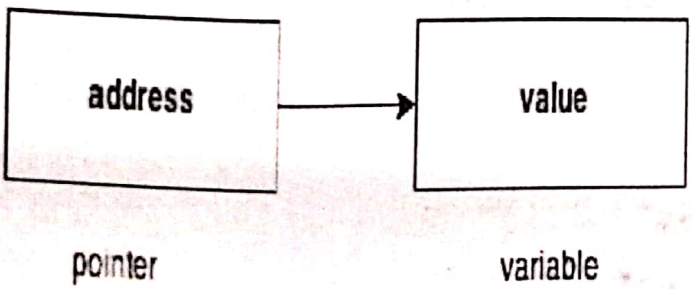
1. int arr[10];
2. int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.

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,&numb
   er);
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.
```

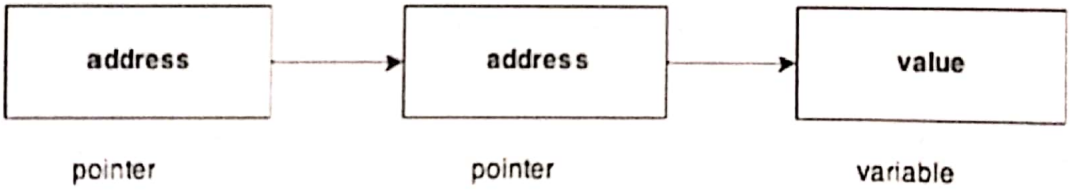
```
11. return 0;
12.     }
```

Output

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

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 p
   ointer 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 will be printed
12.          printf("value stored at pp: %d\n",**pp); // value stored at the
   address contained by the pointer stoyred at pp
13.      }
```

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. 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 c
   ase, p will get incremented 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)

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 n
   ow point to the immidiate 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,

Address2 -
Address1 = (Subtraction of two addresses)/size of data type which po inter 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); }
```

Output

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

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
```

Static Memory Allocation

In static memory allocation whenever the program executes it fixes the size that the program is going to take, and it can't be changed further. So, the exact memory requirements must be known before. Allocation and deallocation of memory will be done by the compiler automatically. When everything is done at compile time (or) before run time, it is called static memory allocation.

Key Features:

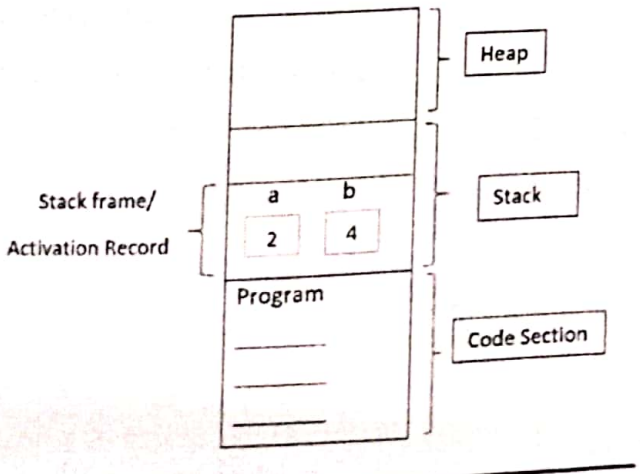
- Allocation and deallocation are done by the compiler.
- It uses a data structures stack for static memory allocation.
- Variables get allocated permanently.
- No reusability.
- Execution is faster than dynamic memory allocation.
- Memory is allocated before runtime.
- It is less efficient.

```
// C++ program to illustrate the

// concept of memory allocation
```

```
#include <iostream>

using namespace std;

 // Driver Code

void main()

{int a; // 2 byte

long b; // 4 bytes

}
```

Explanation:

- The above piece of code declared 2 variables. Here the assumption is that **int takes 2 bytes** and **long takes 4 bytes** of memory. How much memory is taken by variables is again depending upon the compiler.
- These variables will be stored in the stack section. For every function in the program it will take some part of the stack section it is known as the Activation record (or) stack frame and it will be deleted by the compiler when it is not in use.
- Below is the image to illustrate the same:



Below is the C program to illustrate the Static Memory Allocation:

- C

C program to implement

// static memory allocation

```
#include <stdio.h>

#include <stdlib.h>

  // Driver code

int main()

{   int size;

    printf("Enter limit of the text: \n");

    scanf("%d", &size);

    char str[size];

    printf("Enter some text: \n");

    scanf(" ");

    gets(str);

    printf("Inputted text is: %s\n", str);
```

```
return 0;}
```

Advantages:

- Simple usage.
- Allocation and deallocation are done by the compiler.
- Efficient execution time.
- It uses stack data structures.

Disadvantages:

- Memory wastage problem.
- Exact memory requirements must be known.
- Memory can't be resized once after initialization.

Dynamic memory allocation in C

In Dynamic memory allocation size initialization and allocation are done by the programmer. It is managed and served with pointers that point to the newly allocated memory space in an area which we call the heap. Heap memory is unorganized and it is treated as a resource when you require the use of it if not release it. When everything is done during run time or execution time it is known as Dynamic memory allocation.

Key Features:

- Dynamic allocated at runtime
- We can also reallocate memory size if needed.
- Dynamic Allocation is done at run time.
- No memory wastage

There are some functions available in the stdlib.h header which will help to allocate memory dynamically.

- **malloc()**: The simplest function that allocates memory at runtime is called malloc(). There is a need to specify the number of bytes of memory that are required to be allocated as the argument returns the address of the first byte of memory that is allocated because you get an address returned, a pointer is the only place to put it.

Syntax:
int *p = (int*)malloc(No of values*size(int));

The argument to malloc() above clearly indicates that sufficient bytes for accommodating the number of values of type int should be made available. Also notice the cast (int*), which converts the address returned by the function to the type pointer to int. malloc() function returns a pointer with the value **NULL**.

- **calloc()**: The calloc() function offers a couple of advantages over malloc(). It allocates memory as a number of elements of a given size. It initializes the memory that is allocated so that all bytes are zero. calloc() function requires two argument values:
    - The number of data items for which space is required.
    - Size of each data item.

It is very similar to using malloc() but the big plus is that you know the memory area will be initialized to zero.

Syntax:
int *p = (int*)calloc(Number of data items, sizeof(int));

- **realloc()**: The **realloc() function** enables you to reuse or extend the memory that you previously allocated using malloc() or calloc(). A pointer containing an address that was previously returned by a call to malloc(), calloc(). The size in bytes of the new memory that needs to be allocated. It allocates the memory specified by the

second argument and transfers the contents of the previously allocated memory referenced by the pointer passed as the first argument to the newly allocated memory.

Syntax:

int *np = (type cast) realloc (previous pointer type, new number of elements * sizeof(int));

- . **free()**: When memory is allocated dynamically it should always be released when it is no longer required. Memory allocated on the heap will be automatically released when the program ends but is always better to explicitly release the memory when done with it, even if it's just before exiting from the program. A memory leak occurs memory is allocated dynamically and reference to it is not retained, due to which unable to release the memory.

Syntax:

free(pointer);

```
// C program to illustrate the concept

// of memory allocation

#include <iostream>

using namespace std;

 // Driver Code

void main()
{ int* p; // 2 bytes  P = (int*)malloc(5 * sizeof(int));}
```

Examples:

- In the above piece of code, a pointer p is declared. Assume that pointer p will take 2 bytes of memory and again it depends upon the compiler.
- This pointer will store in the stack section and will point to the array address of the first index which is allocated in the heap. Heap memory cannot be used directly but with the help of the pointer, it can be accessed.

.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

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 is given below:

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

Let's see the 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 is given below:

- 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.

Let's see the syntax of realloc() function.

- ptr=realloc(ptr, new-size)

```
2. #include <stdio.h>
3. #include <stdlib.h>
4. int main() {
5.    int n = 4, i, *p, s = 0;
6.    p = (int*) calloc(n, sizeof(int));
```

```
7.    if(p == NULL) {
8.       printf("
   Error! memory not allocated.");
9.       exit(0);
10.    }
11.  printf("
   Enter elements of array : ");
12.    for(i = 0; i < n; ++i) {
13.       scanf("%d", p + i);
14.       s += *(p + i);
15.    }
16.    printf("
   Sum : %d", s);
17.    p = (int*) realloc(p, 6);
18.    printf("
   Enter elements of array : ");
19.    for(i = 0; i < n; ++i) {
20.       scanf("%d", p + i);
21.       s += *(p + i);
22.    }
23.    printf("
   Sum : %d", s);
24.    return 0;
25. }
```

```
26.       Output
27. Enter elements of array : 3 34 28 8
28. Sum : 73
29. Enter elements of array : 3 28 33 8 10 15
30. Sum : 145
```

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.

Let's see the syntax of free() function.

- free(ptr)

example:

```
#include <stdio.h>
    int main() {
int* ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL){
  *(ptr + 2) = 50;
      printf("Value of the 2nd integer is %d",*(ptr + 2));
}
    free(ptr);
}
```

Output of the above free in C example:

Value of the 2nd integer is 50