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. Means, in 32-bit system pointer size is 4 Byte and in 64-bit system pointer size is 8 Byte.
- The pointer in c language can be declared using * (*asterisk symbol*). It is also known as *indirection pointer* used to dereference a pointer.

- **Pointer:**
  - Syntax:

```
int main()
{
        int *ptr;          // A pointer variable is declared
}
```

Explanation:

ptr **:** It gives address of the assigned variable.
***ptr** **:** It gives the value of assigned variable.
&ptr **:** It give the address of the pointer variable i.e. address of itself.

  - **Example:**
    - *Input:*

```
int main()
{
        int a    = 87;
        float b  = 4.5;

        int      *p1 = &a;      // referencing variable 'a' in pointer 'p1'.
        float    *p2;
        p2 = &b;                // referencing variable 'b' in pointer 'p2'.

        printf("Value of a = %d, *p1 = %d, *(&a) = %d\n", a, *p1, *(&a));
        printf("Value of b = %f, *p2 = %f, *(&b) = %f\n", b, *p2, *(&b));

        printf("Value of &a = %p, p1 = %p\n", &a, p1);
        printf("Value of &b = %p, p2 = %p\n", &b, p2);

        printf("Value of &p1 = %p\n", &p1);
        printf("Value of &p2 = %p\n", &p2);
}
```

- *Output:*

```
Value of a = 87, *p1 = 87, *(&a) = 87
Value of b = 4.500000, *p2 = 4.500000, *(&b) = 4.500000

Value of &a = 0x7ffdf05734f8, p1 = 0x7ffdf05734f8
Value of &b = 0x7ffdf05734fc, p2 = 0x7ffdf05734fc

Value of &p1 = 0x7ffdf0573500
Value of &p2 = 0x7ffdf0573508
```

- **Void Pointers:**
  - A void pointer is a pointer that has no associated data type with it.
  - A void pointer can hold address of any type and can be type-casted to any type.
  - Void Pointer Example:

```
int main()
{
        int a     = 10;
        char b  = 'x';

        void *p = &a;          // void pointer holds address of int 'a'
        p = &b;                // void pointer holds address of char 'b'
}
```

  - Important Points:
    - Void pointers cannot be dereferenced. It can however be done using typecasting the void pointer.
    - Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.
  - Void pointers in C are used to implement generic functions in C.
  - **Generic Function Code:**
    - *Input:*

```
/* Main Function */
int main()
{
        int a = 5, b = 10;
        float n = 2.4, m = 3.7;
        char x = 'V', y = 'S';

        fun('I', &a, &b);
```

```
        fun('C', &x, &y);
        fun('f', &n, &m);
}

/* Called Function */
void fun(char ch, void *ptr1, void *prt2)
{
        switch(ch)
        {
                case 'I':
                case 'i':
                {
                        int *p1 = (int*)ptr1;
                        int *p2 = (int*)prt2;
                        printf("%d\n", (*p1 + *p2));
                        break;
                }
                case 'C':
                case 'c':
                {
                        char *p1 = (char*)ptr1;
                        char *p2 = (char*)prt2;
                        printf("%c%c\n", *p1 , *p2);
                        break;
                }
                case 'F':
                case 'f':
                {
                        float *p1 = (float*)ptr1;
                        float *p2 = (float*)prt2;
                        printf("%f\n", (*p1 + *p2));
                        break;
                }
        }
}
```

- *Output:*

```
15
VS
6.100000
```

- **Null Pointers:**
  - NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.
  - Example:
    - *Input:*

```
int main()
{
        // Null Pointer
        int *ptr = NULL;
        printf("The value of ptr is %p", ptr);

        return 0;
}
```

    - *Output:*

```
The value of ptr is(nil)
```

  - NULL is defined as ***((void*)0)***, we can think of NULL as a special pointer and its size would be equal to any pointer.
  - It is used initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
  - Always check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.
  - It also used to pass a null pointer to a function argument when we don't want to pass any valid memory address.
  - Important Points:
    - **NULL vs Uninitialized pointer –** An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
    - **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type.

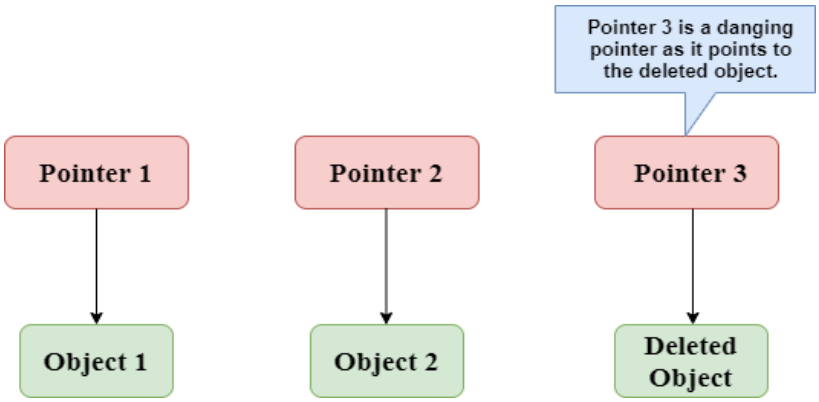
- **Wild Pointers:**
  - A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.
  - The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

- Example:

```
int main()
{
        int *p;  /* wild pointer */
        int x = 10;

        // p is not a wild pointer now
        p = &x;
}
```

- **Dangling Pointer:**
  - A pointer pointing to a memory location that has been deleted (or freed) or programmer fails to initialize the pointer with a valid address then this type of pointer is called dangling pointer.



  - There are three different ways where Pointer acts as dangling pointer:
    - **De-allocation of memory:**

```
// Deallocating a memory pointed by ptr causes dangling pointer.
int main()
{
        int *ptr = (int *)malloc(sizeof(int));

        // After below free call, ptr becomes a dangling pointer.
        free(ptr);
        // No more a dangling pointer.
        ptr = NULL;
}
```

- **Function Call:**

```
// The pointer pointing to local variable becomes dangling when
// local variable is not static.
int *fun()
{
        // x is local variable and goes out of scope after an execution
        // of fun() is over.
        int x = 5;

        return &x;
}

int main()
{
        int *p = fun();
        fflush(stdin);

        // p points to something which is not valid anymore.
        printf("%d", *p);
}
```

Output:

```
A garbage Address
```

- **Variable goes out of scope:**

```
void main()
{
        int *ptr;
        .....
        {
                int ch;
                ptr = &ch;
        }
        .....
        // Here ptr is dangling pointer.
}
```

- **Pointer Arithmetic:**
  - All types of arithmetic operations are not possible with pointers. The only valid operations that can be performed are as-
    - Addition of an integer to a pointer and increment operation.
    - Subtraction of an integer from a pointer and decrement operation.
    - Subtraction of a pointer from another pointer of same type.
  - Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative to the size of base type of pointer.
  - **Example:**
    - *Input:*

```
int main()
{
        Int     x;
        char    y;
        double z;
        int     *ptr1 = &x;
        char    *ptr2 = &y;
        double *ptr3 = &z;

        //Input Data
        printf("ptr1      :  %u\n", ptr1);
        printf("ptr2      :  %u\n", ptr2);
        printf("ptr3      :  %u\n", ptr3);

        //Case 1: Incrementing to 1
        ptr1++;
        ptr2++;
        ptr3++;
        printf("ptr1++    :  %u\n", ptr1);
        printf("ptr2++    :  %u\n", ptr2);
        printf("ptr3++    :  %u\n", ptr3);

        //Case 2: Incrementing to 3
        ptr1 = ptr1 + 3;
        ptr2 = ptr2 + 3;
        ptr3 = ptr3 + 3;
        printf("ptr1 + 3  :  %u\n", ptr1);
        printf("ptr2 + 3  :  %u\n", ptr2);
        printf("ptr3 + 3  :  %u\n", ptr3);
```

```
        //Case 3: Decrementing to 2
        ptr1 = ptr1 - 2;
        ptr2 = ptr2 - 2;
        ptr3 = ptr3 - 2;
        printf("ptr1 - 2   :  %u\n", ptr1);
        printf("ptr2 - 2   :  %u\n", ptr2);
        printf("ptr3 - 2   :  %u\n", ptr3);
}
```

- *Output:*

```
ptr1       :  2862701260
ptr2       :  2862701259
ptr3       :  2862701264

ptr1++     :  2862701264
ptr2++     :  2862701260
ptr3++     :  2862701272

ptr1 + 3   :  2862701276
ptr2 + 3   :  2862701263
ptr3 + 3   :  2862701296

ptr1 - 2   :  2862701268
ptr2 - 2   :  2862701261
ptr3 - 2   :  2862701280
```

- **Increment / Decrement Operators:**
  - The precedence level of * operator and increment/decrement operators, is same and their associativity is from right to left.
  - **Example:**
    - *Input:*

```
int main()
{
        int val = 5;
        int *ptr = &val;
```

```
        printf("Input Value:\n");
        printf("*ptr   :  %d    ::    ptr    :  %u \n", *ptr, ptr);

        *ptr++;
        printf("Case 1: *ptr++ or *(ptr++) \n");
        printf("*ptr   :  %d    ::    ptr    :  %u\n", *ptr, ptr);

        *++ptr;
        printf("Case 2: *++ptr or *(++ptr) \n");
        printf("*ptr   :  %d    ::    ptr    :  %u\n", *ptr, ptr);

        ++*ptr;
        printf("Case 3: ++*ptr or ++(*ptr) \n");
        printf("*ptr   :  %d    ::    ptr    :  %u\n", *ptr, ptr);

        (*ptr)++;
        printf("Case 4: (*ptr)++ \n");
        printf("*ptr   :  %d    ::    ptr    :  %u\n", *ptr, ptr);
}
```

- *Output:*

```
Input Value:
*ptr   :  5    ::    ptr    :  3237942432

Case 1: *ptr++ or *(ptr++)
*ptr   :  5    ::    ptr    :  3237942436

Case 2: *++ptr or *(++ptr)
*ptr   :  5    ::    ptr    :  3237942436

Case 3: ++*ptr or ++(*ptr)
*ptr   :  6    ::    ptr    :  3237942432

Case 4: (*ptr)++
*ptr   :  6    ::    ptr    :  3237942432
```

- **Pointer to Pointer:**
  - We know that pointer is a variable that can contain memory address. This pointer variable takes some space in memory and hence it also has an address. We can store the address of a pointer variable in some other variable, which is known as a *pointer to pointer* variable.
  - Similarly, we can have a pointer to pointer to pointer variable and this concept can be extended to any limit, but in practice only pointer to pointer is used.
  - Pointer to pointer is generally used while passing pointer variables to functions.
  - **Syntax:**

```
data_type      **pptr;
```

  - **Example:**
    - *Input:*

```
int main()
{
        int x       =     5;
        int *ptr    =     &x;
        int **p2p   =     &ptr;

        printf("Value of [x = %d] , [*ptr = %d] , [*(&a) = %d] , [**p2p = %d] ,  [**(&ptr) = %d]\n", x, *ptr, *(&x), **p2p, **(&ptr));

        printf("Value of [&x = %p] , [ptr = %p], [*p2p = %p]\n", &x, ptr, *p2p);

        printf("Value of [&ptr = %p] , [p2p = %p]\n", &ptr, p2p);

        printf("Value of [&p2p = %p]\n", &p2p);
}
```

    - *Ouput:*

```
Value of [x = 5] , [*ptr = 5] , [*(&a) = 5] , [**p2p = 5] , [**(&ptr) = 5]

Value of [&x = 0x7ffc20977b4c] , [ptr = 0x7ffc20977b4c], [*p2p = 0x7ffc20977b4c]

Value of [&ptr = 0x7ffc20977b50] , [p2p = 0x7ffc20977b50]

Value of [&p2p = 0x7ffc20977b58]
```

- **Pointers and Arrays:**
  - When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e. address of the first element of the array is also allocated by the compiler.
  - Suppose we declare an array *arr*, → int arr[5] = { 1, 2, 3, 4, 5 };
  - Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

  - Here variable *arr* will give the base address, which is a constant pointer pointing to the first element of the array, *arr[0]*. Hence *arr* contains the address of *arr[0]* i.e *1000*. In short, *arr* has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.
  - *arr* is equal to *&arr[0]* by default.
  - We can also declare a pointer of type *int* to point to the array *arr*.

```
int main()
{
        int arr[5] = { 1, 2, 3, 4, 5 };
        int *p;
        p = arr;
        // or,
        p = &arr[0];    //both the statements are equivalent.
}
```

  - Now we can access every element of the array *arr* using *p++* to move from one element to another.
  - You cannot decrement a pointer once incremented. *p--* won't work.

- **Pointers to Array:**
  - As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements.

- **Example:**

```
int main()
{
        int i;
        int a[5] = {1, 2, 3, 4, 5};

         int *p = a;     // same as int*p = &a[0]
        for (i = 0; i < 5; i++)
        {
                printf("%d", *p);
                p++;
        }
}
```

- In the above program, the pointer **p* will print all the values stored in the array one by one. We can also use the Base address (*a* in above case) to act as a pointer and print all the values.

Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

printf("%d", *a); ⟶ **will print value of a[0] only**

a++; ⟶ **Compile time error, we cannot change base address of the array.**

- The generalized form for using pointer with an array is:

```
*(a+i)
```

- And it is same as:

```
a[i]
```

- **Pointers to Multidimensional Array:**
  - A multidimensional array is of form, *a[i][j]*.
  - We know that, name of the array gives its base address. In *a[i][j]*, *a* will give the base address of this array, even *a + 0 + 0* will also give the base address, that is the address of *a[0][0]* element.
  - Here is the generalized form for using pointer with multidimensional arrays.

```
*(*(a + i) + j)
```

  - And which is same as,

```
a[i][j]
```

- **Pointers and Characters Strings:**
  - Pointer can also be used to create strings.
  - Pointer variables of *char* type are treated as string.

```
char *str = "Hello";
```

  - The above code creates a string and stores its address in the pointer variable *str*. The pointer *str* now points to the first character of the string "Hello". Another important thing to note here is that the string created using *char* pointer can be assigned a value at *runtime*.

```
char *str;
str = "hello";      //this is Legal
```

  - The content of the string can be printed using *printf()* and *puts()*.

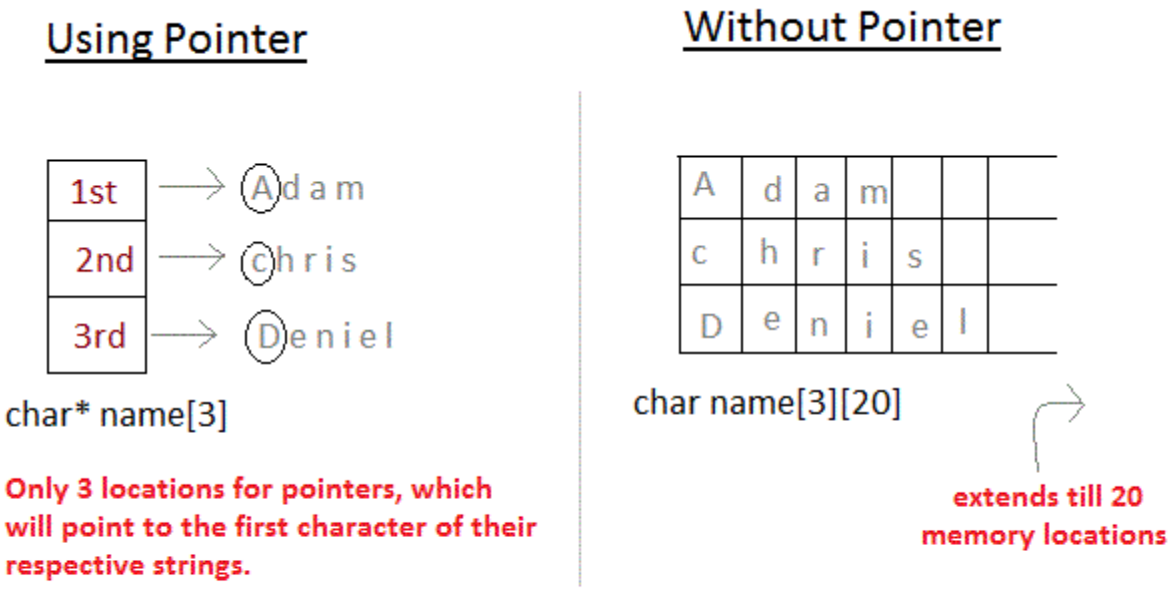
```
printf("%s", str);
puts(str);
```

  - Here *str* is pointer to the string, it is also name of the string. Therefore, we do not need to use indirection operator *.

- **Array of Pointers:**
  - Pointers are very helpful in handling character array with rows of varying length.

```
// Declaration of array using pointers
char *name[3] = { "Adam", "chris", "Deniel" };

//Now lets see same array without using pointer
char name[3][20] = { "Adam", "chris", "Deniel" };
```



- In the second approach memory wastage is more, hence it is prefered to use pointer in such cases.
- When we say memory wastage, it doesn't means that the strings will start occupying less space, means no characters will take the same space, but when we define array of characters, a contiguous memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

- **Pointers as Function Argument:**
  - Pointer as a function parameter is used to hold addresses of arguments passed during function call.
  - This is also known as **call by reference**.
  - When a function is called by reference any change made to the reference variable will affect the original variable.
  - **Example:** Swapping two numbers using pointer.
    - *Input:*

```
int main()
{
        int m = 10, n = 20;
        printf("m = %d\n", m);
```

```
        printf("n = %d\n\n", n);

        swap(&m, &n);    //passing address of m and n to the swap function

        printf("After Swapping:\n\n");
        printf("m = %d\n", m);
        printf("n = %d", n);
}

/*
pointer 'a' and 'b' holds and
points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
        int temp;
        temp = *a;
        *a = *b;
        *b = temp;
}
```

- *Output:*

```
m = 10
n = 20
After Swapping:
m = 20
n = 10
```

- **Function returning Pointer variables:**
  - A function can also *return* a pointer to the calling function.
  - In this case we have to be careful, because local variables of function doesn't live outside the function. They have scope only inside the function.
  - Hence if we return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.
  - **Example:**
    - *Input:*

```
int main()
{
        int a = 15;
        int b = 92;
```

```
        int *p;
        p = larger(&a, &b);
        printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{
        if(*x > *y)
                return x;
        else
                return y;
}
```

- *Output*

```
92 is larger
```

- Safe ways to return a valid pointers:-
  - Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.
  - Or, use **static local variables** inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available through out the program.

• Function Pointer / Pointer to Function:

- A function pointer, also called a **subroutine pointer** or **procedure pointer**, is a pointer that points to a function.
- As opposed to referencing a data value, a function pointer points to executable code within memory.
- Dereferencing the function pointer yields the referenced function, which can be invoked and passed arguments just as in a normal function call. Such an invocation is also known as an "indirect" call, because the function is being invoked *indirectly* through a variable instead of *directly* through a fixed identifier or address.
- Function pointers can be used to simplify code by providing a simple way to select a function to execute based on run-time values.
- **Syntax:**

```
type (*pointer-name)(parameter);
```

- Here is an example:-

```
int (*sum)();   //legal declaration of pointer to function
int *sum();     //This is not a declaration of pointer to function
```

- A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

- Here, *s* is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```
s (10, 20);
```

- **Example:**
  - *Input:*

```
int main( )
{
        int (*fp)(int, int);
        fp = sum;
        int s = fp(10, 15);
        printf("Sum is %d", s);
}

int sum(int x, int y)
{
        return x+y;
}
```

  - *Output:*

```
25
```

- Complicated Function Pointer example:
  - There are lot of complex function pointer examples around like shown below:

```
        void *(*foo) (int*);
```

  - It appears complex but it is very simple. In this case (*foo) is a pointer to the function, whose argument is of int* type and return type is void*.
