Pointer in C programming

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

Address in C

If you have a variable `var` in your program, `&var` will give you its address in the memory.

We have used address numerous times while using the `scanf()` function.

```
scanf("%d", &var);
```

Here, the value entered by the user is stored in the address of `var` variable. Let's take a working example.

```
#include <stdio.h>
int main()
{
  int var = 5;
  printf("var: %d\n", var);

  // Notice the use of & before var
  printf("address of var: %p", &var);
  return 0;
}
```

Output

```
var: 5
address of var: 268677.
```

C Pointers

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
int *p1;
int * p2;
```

Let's take another example of declaring pointers.

```
int* p1, p2;
```

Here, we have declared a pointer `p1` and a normal variable `p2`.

Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;
c = 5;
pc = &c;
```

Here, 5 is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);   // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

> Note: In the above example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c`;
>
> By the way, `*` is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

Changing Value Pointed by Pointers

Let's take an example.

```
int* pc, c;
c = 5;
pc = &c;
c = 1;
printf("%d", c);    // Output: 1
printf("%d", *pc);  // Ouptut: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed the value of `c` to 1. Since `pc` and the address of `c` is the same, `*pc` gives us 1.

Let's take another example.

```
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);  // Ouptut: 1
printf("%d", c);    // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed `*pc` to 1 using `*pc = 1;`. Since `pc` and the address of `c` is the same, `c` will be equal to 1.

Let's take one more example.

```
int* pc, c, d;
c = 5;
d = -15;

pc = &c; printf("%d", *pc); // Output: 5
pc = &d; printf("%d", *pc); // Ouptut: -15
```

Initially, the address of `c` is assigned to the `pc` pointer using `pc = &c;`. Since `c` is 5, `*pc` gives us 5.

Then, the address of `d` is assigned to the `pc` pointer using `pc = &d;`. Since `d` is -15, `*pc` gives us -15.

Example: Working of Pointers

Let's take a working example.

```
#include <stdio.h>
int main()
{
   int* pc, c;

   c = 22;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c);  // 22

   pc = &c;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 22

   c = 11;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 11

   *pc = 2;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c); // 2
   return 0;
}
```

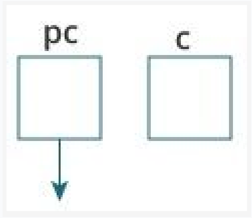
Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
```

```
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

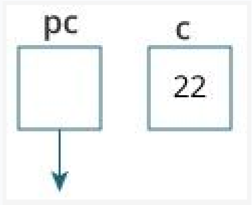
Explanation of the program

1. `int* pc, c;`

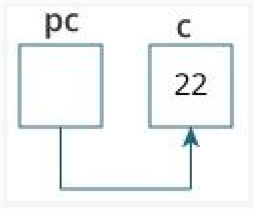
   Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created. Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`

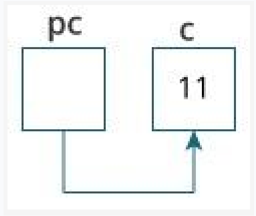
   This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`

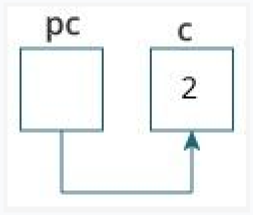
   This assigns the address of variable `c` to the pointer `pc`.

4. `c = 11;`

   This assigns 11 to variable `c`.

5. `*pc = 2;`

   This change the value at the memory location pointed by the pointer `pc` to 2.

Common mistakes when working with pointers

Suppose, you want pointer `pc` to point to the address of `c`. Then,

```
int c, *pc;

// pc is address but c is not
pc = c; // Error

// &c is address but *pc is not
*pc = &c; // Error

// both &c and pc are addresses
pc = &c;

// both c and *pc values
*pc = c;
```

Here's an example of pointer syntax beginners often find confusing.

```
#include <stdio.h>
int main() {
   int c = 5;
   int *p = &c;

   printf("%d", *p);  // 5
   return 0;
}
```

Why didn't we get an error when using `int *p = &c;`?

It's because

```
int *p = &c;
```

is equivalent to

```
int *p:
p = &c;
```

In both cases, we are creating a pointer `p` (not `*p`) and assigning `&c` to it.

To avoid this confusion, we can use the statement like this:

```
int* p = &c;
```

In **static memory allocation**, **memory** is **allocated** while writing the **C** program. Actually, user requested **memory** will be **allocated** at compile time. In **dynamic memory allocation**, **memory** is **allocated** while executing the program. That means at run time.

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()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

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.

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
       printf("Sorry! unable to allocate memory");
       exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
       scanf("%d",ptr+i);
       sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

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.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

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)
```

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)
```

ADDRESS OPERATORS AND POINTERS

The "**Address** Of" **Operator** denoted by the ampersand character (&), & is a unary **operator**, which returns the **address** of a variable. After declaration of a **pointer** variable, we need to initialize the **pointer** with the valid memory **address**; to get the memory **address** of a variable **Address** Of" (&) **Operator** is used.

Pointer address operator in C Programming

1. Pointer address operator is denoted by '&' symbol
2. When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

lets take an example –

```
&n  -  It gives an address on variable n
```

Working of address operator

```
#include<stdio.h>
void main()
{
int n = 10;
printf("\nValue of n is : %d",n);
printf("\nValue of &n is : %u",&n);
}
```

Output :

```
Value of  n is : 10
Value of &n is : 1002
```

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with `%u`

```
printf("\nValue of &n is : %u",&n);
```

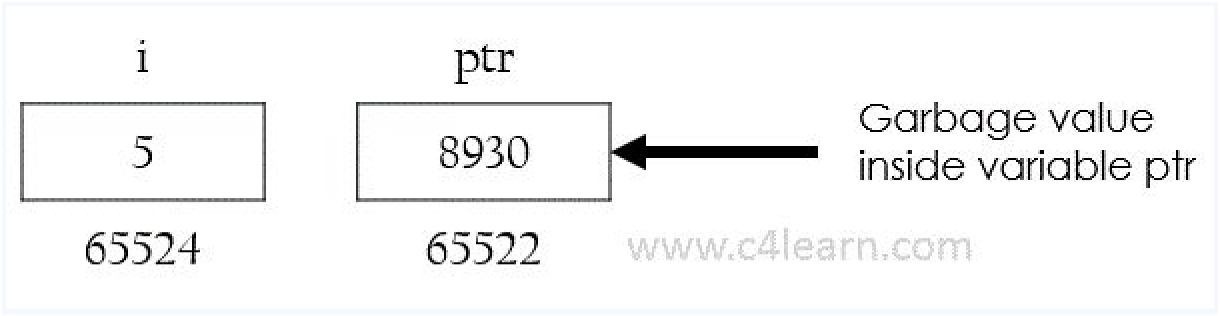
Understanding address operator

Consider the following program –

```
#include<stdio.h>
int main()
{
int i = 5;
int *ptr;

ptr = &i;

printf("\nAddress of i      : %u",&i);
printf("\nValue of ptr is : %u",ptr);

return(0);
}
```

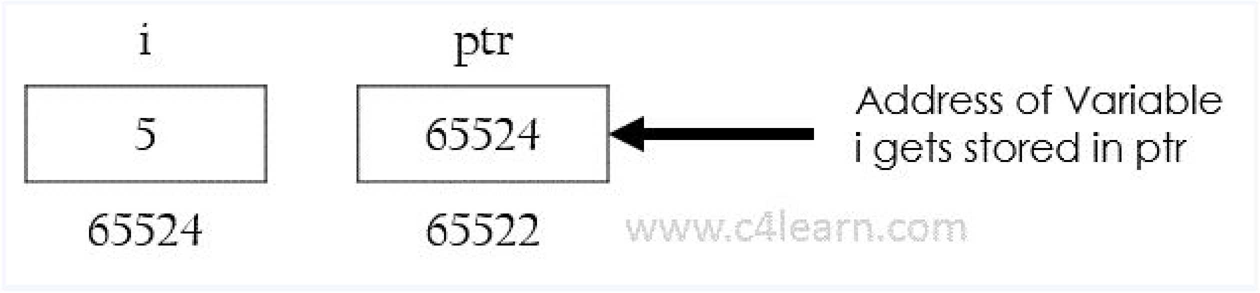
After declaration memory map will be like this –

```
int i = 5;
int *ptr;
```



after Assigning the address of variable to pointer , i.e after the execution of this statement –

```
ptr = &i;
```



Invalid Use of pointer address operator

Address of literals

In C programming using address operator over literal will throw an error. We cannot use address operator on the literal to get the address of the literal.

`&75`

Only variables have an address associated with them, constant entity does not have corresponding address. Similarly we cannot use address operator over character literal –

`&('a')`

Character ‘a’ is literal, so we cannot use address operator.

Address of expressions

`(a+b)` will evaluate addition of values present in variables and output of `(a+b)` is nothing but Literal, so we cannot use Address operator

`&(a+b)`

Conclusion

Thus we have learnt –

1. About address operator
2. How Address Operator is used to access the address of Variable
3. Different illegal or wrong ways of using address operator
4. Visual Understanding about Address operator

Declaring, Initializing and using a pointer variable in C

Declaration of C Pointer variable

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. `void` type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip     // pointer to integer variable
float *fp;      // pointer to float variable
double *dp;     // pointer to double variable
char *cp;       // pointer to char variable
```

Initialization of C Pointer variable

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator** `&` is used to determine the address of a variable. The `&` (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>

void main()
{
    int a = 10;
    int *ptr;       //pointer declaration
    ptr = &a;       //pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>

void main()
{
    float a;
    int *ptr;
    ptr = &a;       // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a `NULL` value to your pointer variable. A pointer which is assigned a `NULL` value is called a **NULL pointer**.

```
#include <stdio.h>

int main()
{
    int *ptr = NULL;
    return 0;
}
```

Using the pointer or Dereferencing of Pointer

Once a pointer has been assigned the address of a variable, to access the value of the variable, pointer is **dereferenced**, using the **indirection operator** or **dereferencing operator** `*`.

```
#include <stdio.h>

int main()
{
    int a, *p;  // declaring the variable and pointer
    a = 10;
    p = &a;     // initializing the pointer

    printf("%d", *p);    //this will print the value of 'a'

    printf("%d", *&a);   //this will also print the value of 'a'

    printf("%u", &a);    //this will print the address of 'a'

    printf("%u", p);     //this will also print the address of 'a'

    printf("%u", &p);    //this will print the address of 'p'

    return 0;
}
```

Points to remember while using pointers

1. While declaring/initializing the pointer variable, `*` indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand `&`.
3. The pointer variable stores the address of a variable. The declaration `int *a` doesn't mean that `a` is going to contain an integer value. It means that `a` is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, `*` is used. Here, the `*` can be read as **'value at'**.

Time for an Example!

Let's take a simple code example,

```
#include <stdio.h>

int main()
{
    int i = 10;     // normal integer variable storing value 10
    int *a;     // since '*' is used, hence its a pointer variable

    /*
        '&' returns the address of the variable 'i'
        which is stored in the pointer variable 'a'
    */
    a = &i;

    /*
```

```
        below, address of variable 'i', which is stored
        by a pointer variable 'a' is displayed
    */
   printf("Address of variable i is %u\n", a);

    /*
        below, '*a' is read as 'value at a'
        which is 10
    */
   printf("Value at the address, which is stored by pointer variable a is
%d\n", *a);

   return 0;
}
```

```
Address of variable i is 2686728 (The address may vary)
Value at an address, which is stored by pointer variable a is 10
```

Pointer to a Pointer in C(Double Pointer)

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator(`*`) which stores and points to the address of a pointer variable i.e, `int *`. If we want to store the address of this (double pointer) variable `p1`, then the syntax would become:

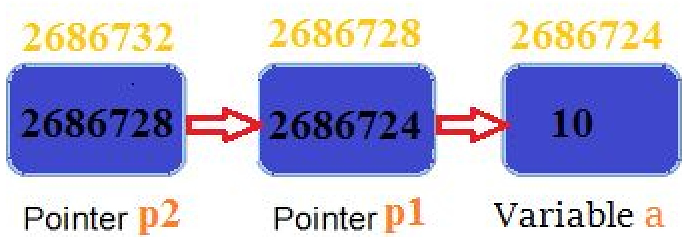
```
int ***p2
```

Simple program to represent Pointer to a Pointer

```c
#include <stdio.h>

int main() {

    int  a = 10;
    int  *p1;        //this can store the address of variable a
    int  **p2;
    /*
        this can store the address of pointer variable p1 only.
        It cannot store the address of variable 'a'
    */

    p1 = &a;
    p2 = &p1;

    printf("Address of a = %u\n", &a);
    printf("Address of p1 = %u\n", &p1);
    printf("Address of p2 = %u\n\n", &p2);

    // below print statement will give the address of 'a'
    printf("Value at the address stored by p2 = %u\n", *p2);

    printf("Value at the address stored by p1 = %d\n\n", *p1);

    printf("Value of **p2 = %d\n", **p2); //read this *(*p2)

    /*
        This is not allowed, it will give a compile time error-
        p2 = &a;
        printf("%u", p2);
    */
    return 0;
}
```

```
Address of a = 2686724
Address of p1 = 2686728
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
Value of **p2 = 10
```

Explanation of the above program



- `p1` pointer variable can only hold the address of the variable `a` (i.e Number of indirection operator(*)-1 variable). Similarly, `p2` variable can only hold the address of variable `p1`. It cannot hold the address of variable `a`.
- `*p2` gives us the value at an address stored by the `p2` pointer. `p2` stores the address of `p1` pointer and value at the address of `p1` is the address of variable `a`. Thus, `*p2` prints address of `a`.
- `**p2` can be read as `*(*p2)`. Hence, it gives us the value stored at the address `*p2`. From above statement, you know `*p2` means the address of variable a. Hence, the value at the address `*p2` is 10. Thus, `**p2` prints `10`.
-

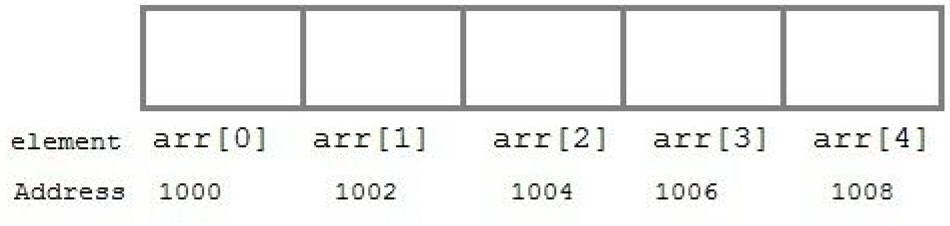
Pointer and Arrays in C

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:



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 *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.

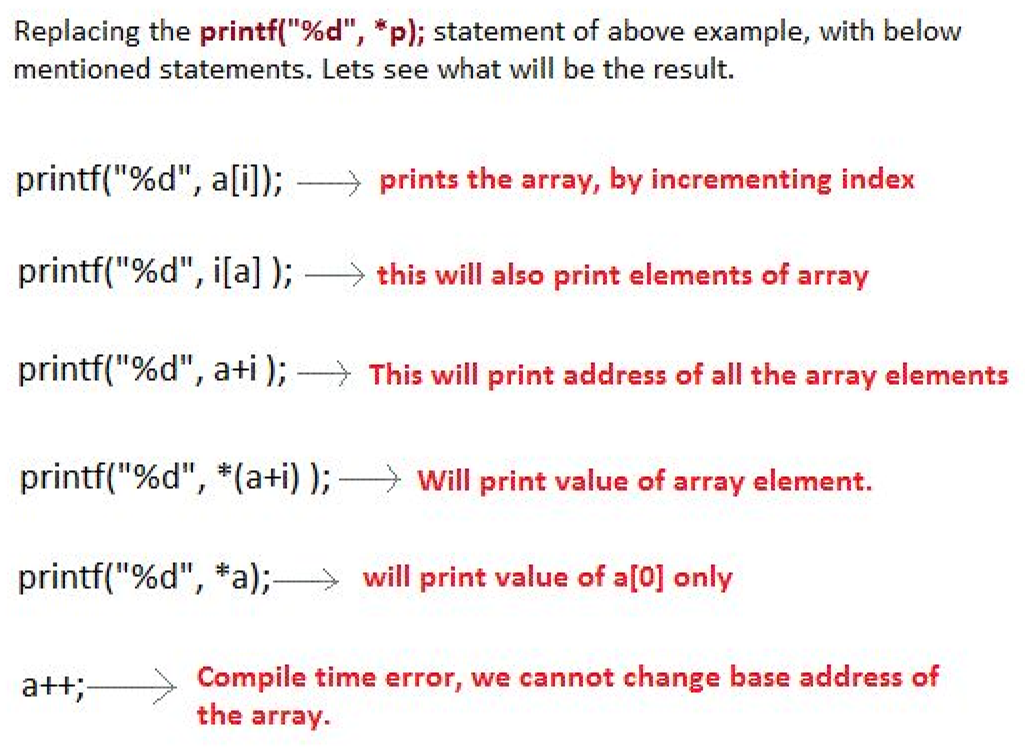
NOTE: You cannot decrement a pointer once incremented. `p--` won't work.

Pointer 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. Lets have an example,

```
#include <stdio.h>

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++;
    }

    return 0;
}
```

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,

`*(a+i)`

is same as:

`a[i]`

Pointer to Multidimensional Array

A multidimensional array is of form, `a[i][j]`. Lets see how we can make a pointer point to such an array. As we know now, 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)
```

which is same as,

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

Pointer and Character 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()`.

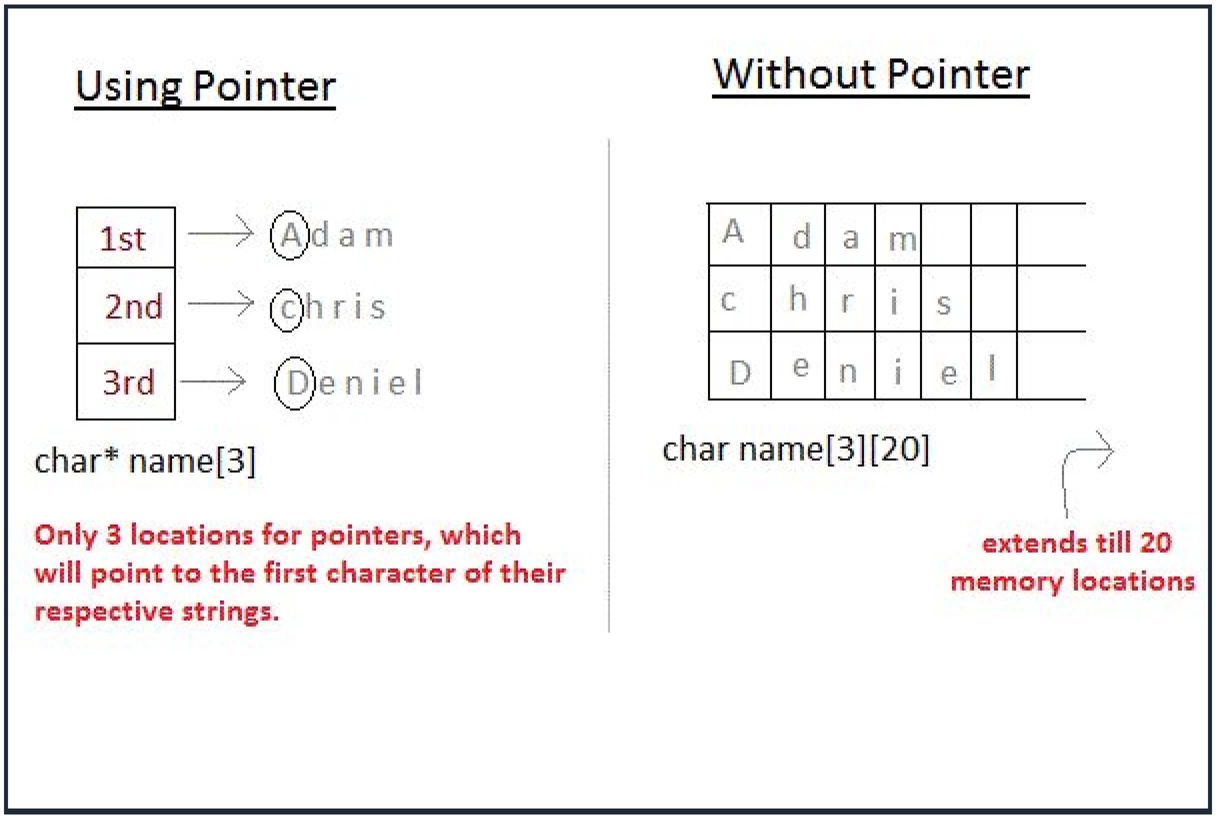
```
printf("%s", str);
puts(str);
```

Notice that `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

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

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


Using Pointer
Without Pointer
1st
2nd
3rd
Adam
chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

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, no, characters will take the same space, but when we define array of characters, a contiguos 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.