POINTERS

Introduction:

Address in C:

Whenever a variable is declaring in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the `&` symbol. If `var` is the name of the variable, then `&var` will give it's address.

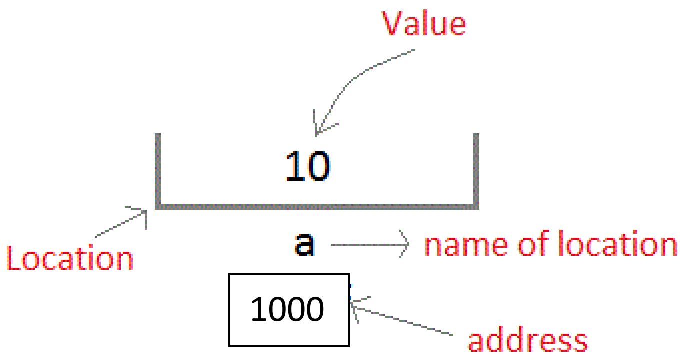
```
scanf("%d", &var);
```

This is used to store the user inputted value to the address of the variable `var`.

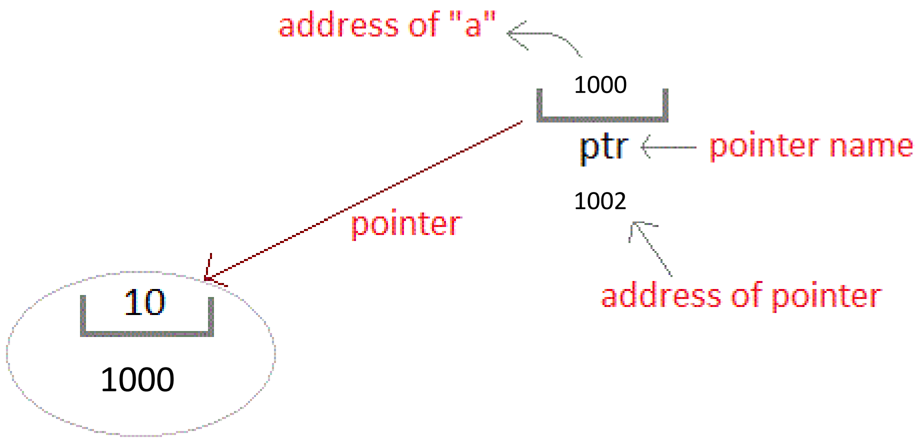
Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location `1000` for a variable `a`.

```
int a = 10;
```



We can access the value 10 either by using the variable name a or by using its address 1000. The variables which are used to hold memory addresses of another variable are called **Pointer variables**. A pointer variable is therefore nothing but a variable which holds an address of some other variable. And the value of a pointer variable gets stored in another memory location.



Declaring Pointer Variables:

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 Pointer Variables

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
}
```

Accessing a Pointer Variable:

A normal variable contains the value of any type like int, char, float etc, while a pointer variable contains the memory address of another variable.

Steps:

1. Declare a normal variable, assign the value
2. Declare a pointer variable with the same type as the normal variable
3. Initialize the pointer variable with the address of normal variable
4. Access the value of the variable by using asterisk (*****) - it is known as **dereference operator**

Example:

Here, we have declared a normal integer variable num and pointer variable ptr, ptr is being initialized with the address of num and finally getting the value of num using pointer variable ptr.

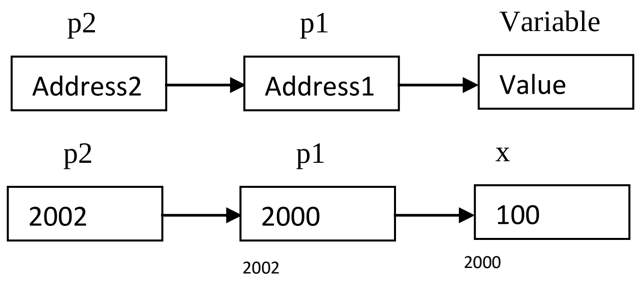
```
#include <stdio.h>

void main()
{
        int num = 100;      //normal variable
        int *ptr;           //pointer variable
        ptr = &num;         //pointer initialization
        printf("value of num = %d\n", *ptr);          //pritning the value
        getch();
}
```

Output

```
value of num = 100
```

Chain of Pointers:

It is possible to create a pointer to another pointer, by creating a chain of pointer.



Here, the pointer variable **p2** contains the address of the pointer variable **p1**, which contains the address of another variable **x**. This is known as ***multiple indirection*** or ***chain of pointers***. A variable that is a pointer to a pointer must be declared using additional indirection operator.

For example,

```
int x=100;
int *p1=&x;
int **p2=&p1;
```

In the above example, variable p2 holds the address of p1 and p1 holds the address of x.

Pointer Expressions:

Like other variables, pointer variables can be used in expressions. For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid:

y=*p1 * *p2;

y=x+ *p1;

sum=sum+*p1;

z=5* - *p2/ *p1;

In the above statement there should be a blank space between / and *. The following is wrong:

z=5* - *p2 /*p1;

In the above statement /* will be considered as a beginning of comment line.
Increment/Decrement and Relational operators also can be used with pointers.

p1++;

It will increments the address of pointer to next address location.

p1>p2

p1= =p2

p1!=p2

Pointer Increments and Scale Factor:

Consider an expression,

p1++;

The above expression will point to the next value of its type. For example, if p1 is an integer pointer with an initial value 2000, then after the above operation (p1++), the value of p1 will be 2002 but not 2001. So when increment a pointer, its value is incremented by the length of the data type that it points to.

Rules of Pointer Operations:

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from other.

7. When two pointers point to the same objects of the same data types, they can compared using relational operators
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variable cannot be added
10. A value cannot be assigned to an arbitrary address. i.e x= &10;

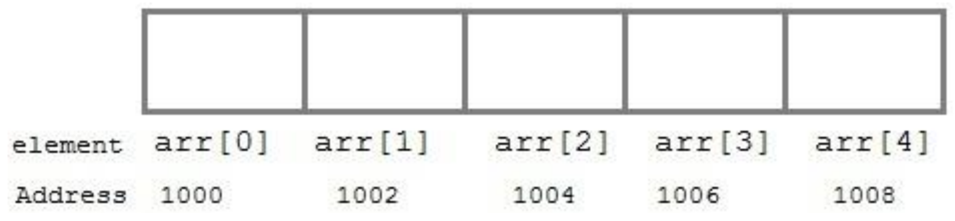
Pointers and Arrays:

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:



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.

Pointers 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);
```

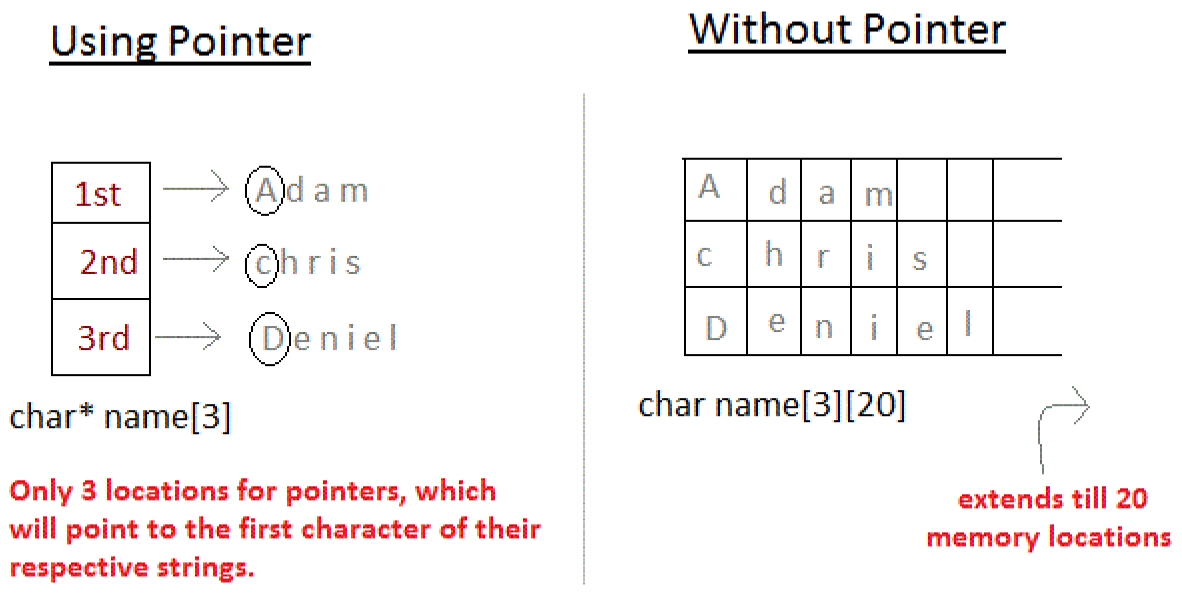
Notice that `str` is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator `*`.

```
//Program for character pointer
#include<stdio.h>
void main()
{
  char *s;
  char str[10];
  clrscr();
  printf("\nEnter a string");
  scanf("%s",str);
  s=str;   //Assign
  printf("\nString in pointer variable %s",s);
  getch();
}
```

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
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.
Without Pointer
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 preferred 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 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.

Example:

```
//Program for array of pointers
#include<stdio.h>
void main()
{
  char *names[3]={"Anita","Bala","Chitra"};
  int i,j;
  clrscr();
  printf("\nStrings are...\n");
  for(i=0;i<3;i++)
   printf("%s\n",names[i]);  //  *(names[i]+j)
  getch();
}
```

Pointers as Function Arguments:

It is possible to pass pointer variable as an argument to a function. This is called pointers as function arguments.

Example:

```
void returnMulValues(int *x, float *y, char *c)
{
    *x=10;
    *y=20.5;
    *c='x';
}
void main()
{
    int Value1=1;
    float Value2=2.5;
    char ch='a';
    printf("Value of Value1: %d\n",Value1);
    printf("Value of Value2: %f\n",Value2);
    printf("Value of ch: %c\n",ch);

     returnMulValues(&Value1,&Value2,&ch);  //function call (or) call by reference

    printf("Value of Value1: %d\n",Value1);
    printf("Value of Value2: %f\n",Value2);
    printf("Value of ch: %c\n",ch);
    getch();
}
```

Output:

```
    Value of Value1: 1
    Value of Value2: 2.5
    Value of ch: a


    Value of Value1: 10
    Value of Value2: 20.5
    Value of ch: x
```

Pointers and Structures:

To access members of structure using the structure variable, we used the dot (.) operator. But when we have a pointer of structure type, we use arrow -> to access structure members.

```
//Program for Pointer with structure
#include <stdio.h>
struct student
{
   char name[20];
   int number;
   int rank;
};
void main()
{
   struct student s1;
   struct student *ptr;
   ptr = &s1;
   printf("\nEnter student name, roll no & rank\n");
   scanf("%s%d%d",s1.name,&s1.number,&s1.rank);
   printf("\nStudent Details are...\n");
   printf("NAME: %s\n", ptr->name);
   printf("NUMBER: %d\n", ptr->number);
   printf("RANK: %d", ptr->rank);
   getch();
}
```

Output:

```
Student Details are…
NAME: Kumar
NUMBER: 35
RANK: 1
```

Dynamic Memory Allocation:

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.

malloc() function:

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=(data-type*)malloc(size);
```

```
 #include<stdio.h>
 #include<stdlib.h>
 void 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);
getch();
}
```

Output:

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

calloc() function:

The calloc() function allocates multiple block of requested memory. It initially initializes 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>
void 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);
```

```
        getch();
    }
```

Output:

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

realloc() function:

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. It just changes the memory size and also deletes the previously stored data.

Syntax of realloc() function:

```
        ptr=realloc(ptr, new-size);
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int *ptr, i , n1, n2;
   printf("Enter size: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("\nEnter %d elements: ",n1);
   for(i = 0; i < n1; ++i)
         scanf("%d",ptr+i);

   printf("\nEnter the new size: ");
   scanf("%d", &n2);

   // rellocating the memory
   ptr = realloc(ptr, n2 * sizeof(int));
   printf("\nNow enter %d elements : ",n2);
   for(i=0; i < n2; ++i)
         scanf("%d",ptr+i);
   printf("\nAll elements in array are\n");
   for(i=0;i<n2;i++)
     printf("%d\n",*(ptr+i));
   free(ptr);
   getch();
}
```

Output:

```
Enter size: 2
Enter 2 elements:
3
4
```

```
Enter the new size: 4
Now enter 4 elements:
5
6
7
8
All elements in array are
5
6
7
8
```

free() function:

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

Syntax of free() function

```
    free(ptr);
```