

POINTERS:

Definition:

- ☐ C Pointer is a variable that stores/points the address of the another variable.
- ☐ C Pointer is used to allocate memory dynamically i.e. at run time.
- ☐ The variable might be any of the data type such as int, float, char, double, short etc.
- ☐ Syntax : data_type *var_name;

Example : int *p; char *p;

Where, * is used to denote that “p” is pointer variable and not a normal variable.

Key points to remember about pointers in C:

- ☐ Normal variable stores the value whereas pointer variable stores the address of the variable.
- ☐ The content of the C pointer always be a whole number i.e. address.
- ☐ Always C pointer is initialized to null, i.e. int *p = null.
- ☐ The value of null pointer is 0.
- ☐ & symbol is used to get the address of the variable.

- ☐ * symbol is used to get the value of the variable that the pointer is pointing to.
- ☐ If pointer is assigned to NULL, it means it is pointing to nothing.
- ☐ The size of any pointer is 2 byte (for 16 bit compiler).
- ☐ No two pointer variables should have the same name.
- ☐ But a pointer variable and a non-pointer variable can have the same name.

4.4.1 Pointer –Initialization:

Assigning value to pointer:

It is not necessary to assign value to pointer. Only zero (0) and NULL can be assigned to a pointer no other number can be assigned to a pointer. Consider the following examples; int *p=0; int *p=NULL; The above two assignments are valid.
int *p=1000; This statement is invalid.

Assigning variable to a pointer:

```
int x; *p;  
p = &x;
```

This is nothing but a pointer variable p is assigned the address of the variable x. The address of the variables will be different every time the program is executed.

Reading value through pointer:

```
int x=123; *p;  
p = &x;
```

Here the pointer variable p is assigned the address of variable x.

printf(“%d”, *p); will display value of x 123. This is reading value through pointer

printf(“%d”, p); will display the address of the variable x.

printf(“%d”, &p); will display the address of the pointer variable p.

printf(“%d”,x); will display the value of x 123.

printf(“%d”, &x); will display the address of the variable x.

Note: It is always a good practice to assign pointer to a variable rather than 0 or NULL.

Pointer Assignments:

We can use a pointer on the right-hand side of an assignment to assign its value to another variable.

Example:

```
int main()
{
int var=50;
int *p1, *p2;
p1=&var;
p2=p1;
}
```

Pointer Expression & Pointer Arithmetic

- ☐ C allows pointer to perform the following arithmetic operations:
- ☐ A pointer can be incremented / decremented.
- ☐ Any integer can be added to or subtracted from the pointer.

A pointer can be incremented / decremented.

In 16 bit machine, size of all types[data type] of pointer always 2 bytes.

Eg:

```
int a;
int *p;
p++;
```

Each time that a pointer p is incremented, the pointer p will points to the memory location of the next element of its base type. Each time that a pointer p is decremented, the pointer p will points to the memory location of the previous element of its base type.

```
int a,*p1, *p2, *p3;
p1=&a;
p2=p1++;
p3=++p1;
printf(“Address of p where it points to %u”, p1); 1000
printf(“After incrementing Address of p where it points to %u”, p1); 1002
printf(“After assigning and incrementing p %u”, p2); 1000
printf(“After incrementing and assigning p %u”, p3); 1002
```

In 32 bit machine, size of all types of pointer is always 4 bytes.

The pointer variable p refers to the base address of the variable a. We can increment the pointer variable,

p++ or ++p

This statement moves the pointer to the next memory address. let p be an integer pointer with a current value of 2,000 (that is, it contains the address 2,000). Assuming 32-bit integers, after the expression

p++

the contents of p will be 2,004, not 2,001! Each time p is incremented, it will point to the next integer. The same is true of decrements. For example,

p –

will cause p to have the value 1,996, assuming that it previously was 2,000. Here is why: Each time that a pointer is incremented, it will point to the memory location of the next element of its base type. Each time it is decremented, it will point to the location of the previous element of its base type.

Any integer can be added to or subtracted from the pointer.

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;
sum=sum+*p1;
z= 5* - *p2/p1;
*p2= *p2 + 10;
```

C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

*/*Program to illustrate the pointer expression and pointer arithmetic*/*

```
#include< stdio.h >
#include<conio.h>
void main()
{
int ptr1,ptr2;
int a,b,x,y,z;
a=30;b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 -6;
y=6*- *ptr1/ *ptr2 +30;
printf("\nAddress of a + %u",ptr1);
printf("\nAddress of b %u", ptr2);
printf("\na=%d, b=%d", a, b);
printf("\nx=%d,y=%d", x, y);
ptr1=ptr1 + 70;
ptr2= ptr2;
printf("\na=%d, b=%d", a, b);
}
```

/ Sum of two integers using pointers*/*

```
#include <stdio.h>
int main()
{
    int first, second, *p, *q, sum;
    printf("Enter two integers to add\n");
    scanf("%d%d", &first, &second);
    p = &first;
    q = &second;
    sum = *p + *q;
    printf("Sum of entered numbers = %d\n",sum);
    return 0;
}
```

Pointers to Pointers in C:

In the c programming language, we have pointers to store the address of variables of any datatype. A pointer variable can store the address of a normal variable. C programming language also provides a pointer variable to store the address of another pointer variable. This type of pointer variable is called a pointer to pointer variable. Sometimes we also call it a double pointer.

syntax for creating pointer to pointer:

```
datatype **pointerName ;
```

- To store the address of normal variable we use single pointer variable
- To store the address of single pointer variable we use double pointer variable
- To store the address of double pointer variable we use triple pointer variable
- Similarly the same for remaining pointer variables also...

Example Program

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

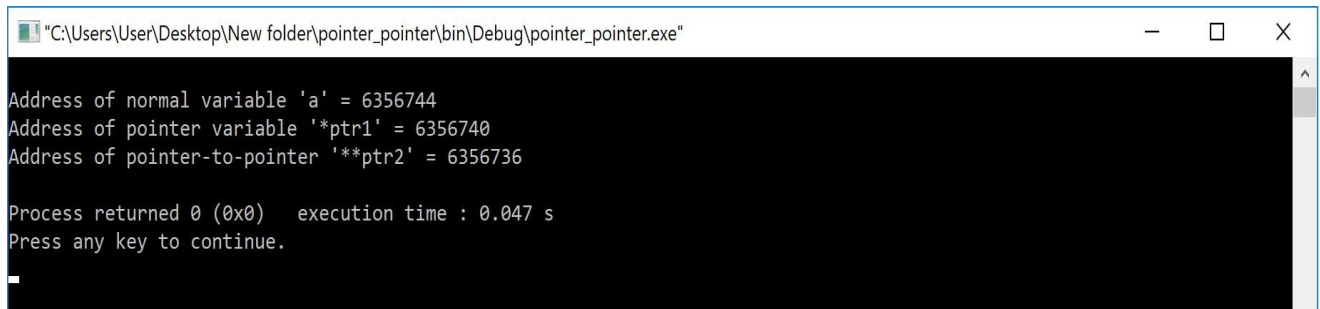
int main()
{
    int a ;
    int *ptr1 ;
    int **ptr2 ;
    int ***ptr3 ;

    ptr1 = &a ;
    ptr2 = &ptr1 ;
```

```
ptr3 = &ptr2 ;

printf("\nAddress of normal variable 'a' = %u\n", ptr1) ;
printf("Address of pointer variable '*ptr1' = %u\n", ptr2) ;
printf("Address of pointer-to-pointer '**ptr2' = %u\n", ptr3) ;
return 0;
}
```

Output:



```
"C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe"
Address of normal variable 'a' = 6356744
Address of pointer variable '*ptr1' = 6356740
Address of pointer-to-pointer '**ptr2' = 6356736

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
_
```

4.4.3 Pointers and Arrays

Array name is a constant pointer that points to the base address of the array[i.e the first element of the array]. Elements of the array are stored in contiguous memory locations. They can be efficiently accessed by using pointers.

Pointer variable can be assigned to an array. The address of each element is increased by one factor depending upon the type of data type. The factor depends on the type of pointer variable defined. If it is integer the factor is increased by 2. Consider the following example:

```
int x[5]={ 11,22,33,44,55}, *p;
p = x; //p=&x; // p = &x[0];
```

Remember, earlier the pointer variable is assigned with address (&) operator. When working with array the pointer variable can be assigned as above or as shown below:

Therefore the address operator is required only when assigning the array with element. Assume the address on x[0] is 1000 then the address of other elements will be as follows

```
x[1] = 1002
x[2] = 1004
x[3] = 1006
x[4] = 1008
```

The address of each element increase by factor of 2. Since the size of the integer is 2 bytes the memory address is increased by 2 bytes, therefore if it is float it will be increase 4 bytes, and for double by 8 bytes. This uniform increase is called scale factor.

```
p = &x[0];
```

Now the value of pointer variable p is 1000 which is the address of array element x[0].

To find the address of the array element `x[1]` just write the following statement.

`p = p + 1;`

Now the value of the pointer variable `p` is 1002 not 1001 because since `p` is pointer variable the increment of will increase to the scale factor of the variable, since it is integer it increases by 2. The `p = p + 1;` can be written using increment or decrement operator `++p;` The values in the array element can be read using increment or decrement operator in the pointer variable using scale factor.

Consider the above example.

```
printf("%d", *(p+0)); will display value of array element x[0] which is 11.
printf("%d", *(p+1)); will display value of array element x[1] which is 22.
printf("%d", *(p+2)); will display value of array element x[2] which is 33.
printf("%d", *(p+3)); will display value of array element x[3] which is 44.
printf("%d", *(p+4)); will display value of array element x[4] which is 55.
```

*/*Displaying the values and address of the elements in the array*/*

```
#include<stdio.h>
void main()
{
int a[6]={ 10, 20, 30, 40, 50, 60};
int *p;
int i;
p=a;
for(i=0;i<6;i++)
{
printf("%d", *p); //value of elements of array
printf("%u",p); //Address of array
}
getch();
}
```

/ Sum of elements in the Array*/*

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10];
int i,sum=0;
int *ptr;
printf("Enter 10 elements:n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
ptr = a; /* a=&a[0] */
for(i=0;i<10;i++)
{
sum = sum + *ptr; /*p=content pointed by 'ptr'
ptr++;
```

```
}  
printf("The sum of array elements is %d",sum);  
}
```

```
/*Sort the elements of array using pointers*/
```

```
#include<stdio.h>  
int main()  
{  
    int i,j, temp1,temp2;  
    int arr[8]={5,3,0,2,12,1,33,2};  
    int *ptr;  
    for(i=0;i<7;i++){  
        for(j=0;j<7-i;j++){  
            if(*(arr+j)>*(arr+j+1)){  
                ptr=arr+j;  
                temp1=*ptr++;  
                temp2=*ptr;  
                *ptr--=temp1;  
                *ptr=temp2;  
            }  
        }  
    }  
    for(i=0;i<8;i++){  
        printf(" %d",arr[i]);  
    }  
}
```

Pointers Arithmetic Operations in C

Pointer variables are used to store the address of variables. Address of any variable is an unsigned integer value i.e., it is a numerical value. So we can perform arithmetic operations on pointer values. But when we perform arithmetic operations on pointer variable, the result depends on the amount of memory required by the variable to which the pointer is pointing

In the c programming language, we can perform the following arithmetic operations on pointers...

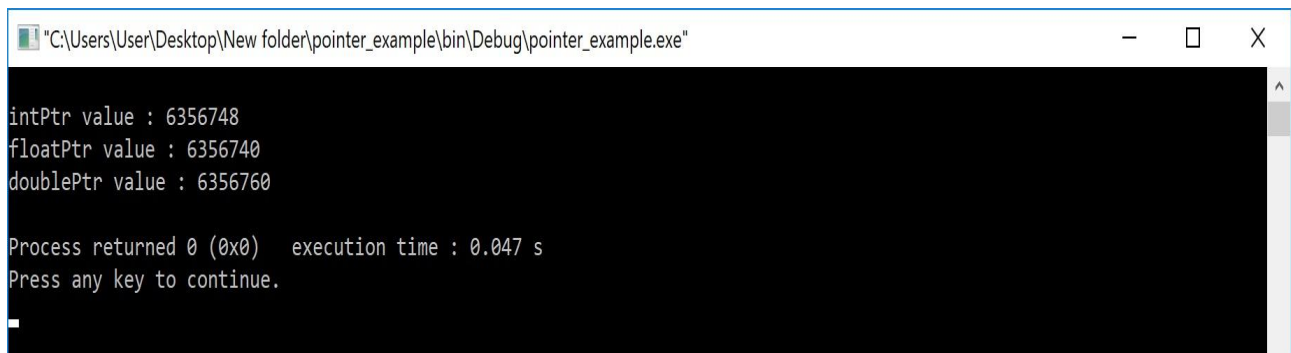
1. Addition
2. Subtraction
3. Increment
4. Decrement
5. Comparison

Addition Operation on Pointer

Example Program

```
void main()  
{  
    int a, *intPtr ;  
    float b, *floatPtr ;  
    double c, *doublePtr ;
```

```
clrscr() ;  
  
intPtr = &a ; // Asume address of a is 1000  
floatPtr = &b ; // Asume address of b is 2000  
doublePtr = &c ; // Asume address of c is 3000  
  
intPtr = intPtr + 3 ; // intPtr = 1000 + ( 3 * 2 )  
floatPtr = floatPtr + 2 ; // floatPtr = 2000 + ( 2 * 4 )  
doublePtr = doublePtr + 5 ; // doublePtr = 3000 + ( 5 * 6 )  
  
printf("intPtr value : %u\n", intPtr) ;  
printf("floatPtr value : %u\n", floatPtr) ;  
printf("doublePtr value : %u", doublePtr) ;  
  
getch() ;  
}
```



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"  
  
intPtr value : 6356748  
floatPtr value : 6356740  
doublePtr value : 6356760  
  
Process returned 0 (0x0)   execution time : 0.047 s  
Press any key to continue.  
-
```

Subtraction Operation on Pointer

Example Program

```
void main()  
{  
    int a, *intPtr ;  
    float b, *floatPtr ;  
    double c, *doublePtr ;  
    clrscr() ;  
    intPtr = &a ; // Asume address of a is 1000  
    floatPtr = &b ; // Asume address of b is 2000  
    doublePtr = &c ; // Asume address of c is 3000
```

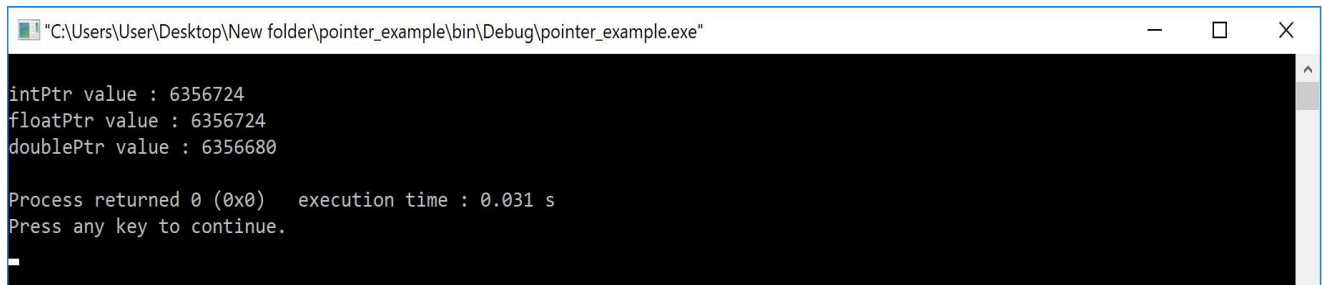


```
intPtr = intPtr - 3 ; // intPtr = 1000 - ( 3 * 2 )
floatPtr = floatPtr - 2 ; // floatPtr = 2000 - ( 2 * 4 )
doublePtr = doublePtr - 5 ; // doublePtr = 3000 - ( 5 * 6 )

printf("intPtr value : %u\n", intPtr) ;
printf("floatPtr value : %u\n", floatPtr) ;
printf("doublePtr value : %u", doublePtr) ;

getch() ;
}
```

Output:



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"
intPtr value : 6356724
floatPtr value : 6356724
doublePtr value : 6356680

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
_
```

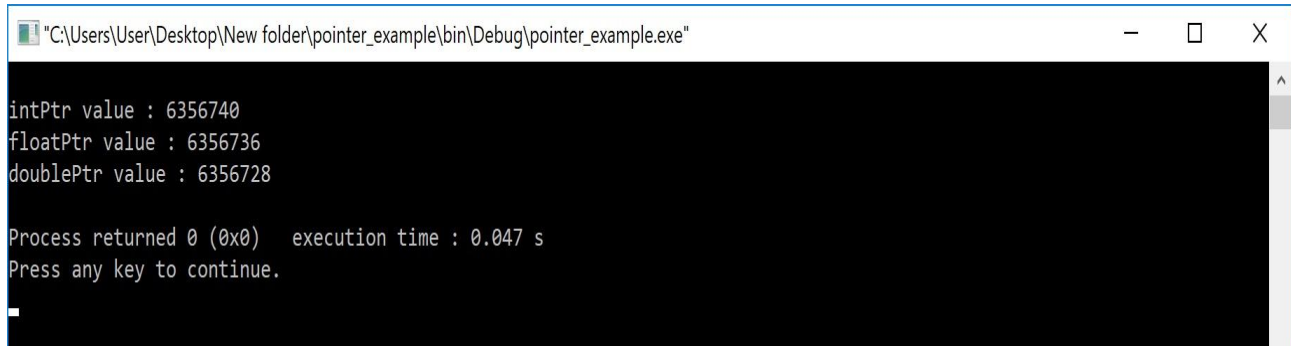
Increment & Decrement Operation on Pointer Example Program

```
void main()
{
    int a, *intPtr ;
    float b, *floatPtr ;
    double c, *doublePtr ;
    clrscr() ;

    intPtr = &a ; // Asume address of a is 1000
    floatPtr = &b ; // Asume address of b is 2000
    doublePtr = &c ; // Asume address of c is 3000

    intPtr++ ; // intPtr = 1000 + 2
    floatPtr++ ; // floatPtr = 2000 + 4
    doublePtr++ ; // doublePtr = 3000 + 6
}
```

```
printf("intPtr value : %u\n", intPtr) ;  
printf("floatPtr value : %u\n", floatPtr) ;  
printf("doublePtr value : %u", doublePtr) ;  
  
getch() ;  
}
```



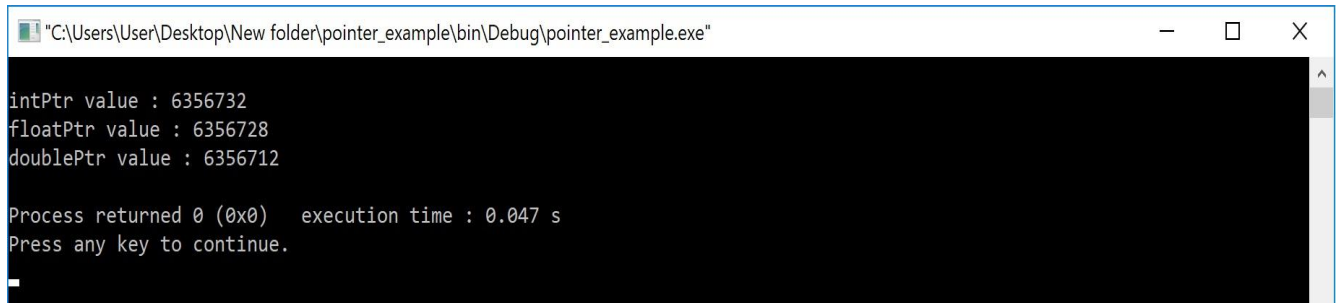
The decrement operation on pointer variable is calculated as follows...

Example Program

```
void main()  
{  
    int a, *intPtr ;  
    float b, *floatPtr ;  
    double c, *doublePtr ;  
    clrscr() ;  
    intPtr = &a ; // Asume address of a is 1000  
    floatPtr = &b ; // Asume address of b is 2000  
    doublePtr = &c ; // Asume address of c is 3000  
  
    intPtr-- ; // intPtr = 1000 - 2  
    floatPtr-- ; // floatPtr = 2000 - 4  
    doublePtr-- ; // doublePtr = 3000 - 6  
  
    printf("intPtr value : %u\n", intPtr) ;  
    printf("floatPtr value : %u\n", floatPtr) ;  
    printf("doublePtr value : %u", doublePtr) ;
```

```
    getch() ;  
}
```

Output:



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"  
  
intPtr value : 6356732  
floatPtr value : 6356728  
doublePtr value : 6356712  
  
Process returned 0 (0x0)   execution time : 0.047 s  
Press any key to continue.  
_
```

Dynamic Memory Allocation in C:

In C programming language, when we declare variables memory is allocated in space called stack. The memory allocated in the stack is fixed at the time of compilation and remains until the end of the program execution. When we create an array, we must specify the size at the time of the declaration itself and it can not be changed during the program execution. This is a major problem when we do not know the number of values to be stored in an array. To solve this we use the concept of Dynamic Memory Allocation. The dynamic memory allocation allocates memory from heap storage. Dynamic memory allocation is defined as follow...

Allocation of memory during the program execution is called dynamic memory allocation.

Dynamic memory allocation is the process of allocating the memory manually at the time of program execution.

We use pre-defined or standard library functions to allocate memory dynamically. There are FOUR standard library functions that are defined in the header file known as "stdlib.h". They are as follows...

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

malloc()

malloc() is the standard library function used to allocate a memory block of specified number of bytes and returns void pointer. The void pointer can be casted to any datatype. If malloc() function unable to allocate memory due to any reason it returns NULL pointer

```
#include <stdlib.h>  
#include<stdio.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("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for(i=0;i<n;++i){
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }

    printf("Sum=%d\n",sum);
    free(ptr);
    return 0;
}
```

calloc()

calloc() is the standard library function used to allocate multiple memory blocks of the specified number of bytes and initializes them to ZERO. calloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns a NULL pointer. Generally, calloc() is used to allocate memory for array and structure. calloc() function takes two arguments and they are 1. The number of blocks to be allocated, 2. Size of each block in bytes

```
#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));
    if(ptr==NULL){
        printf("Error! memory not allocated.");
        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;
}
```

```
}
```

realloc()

realloc() is the standard library function used to modify the size of memory blocks that were previously allocated using malloc() or calloc(). realloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns NULL pointer.

Example Program for *realloc()*.

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

int main () {

    char *title;

    title = (char *) malloc(15);

    strcpy(title, "c programming");
    printf("Before modification : String = %s, Address = %u\n", title, title);

    title = (char*) realloc(title, 30);

    strcpy(title,"C Programming Language");
    printf("After modification : String = %s, Address = %u\n", title, title);

    return(0);

}
```

free()

free() is the standard library function used to deallocate memory block that was previously allocated using malloc() or calloc(). free() function returns void pointer. When free() function is used with memory allocated that was created using calloc(), all the blocks are get deallocated.

Example Program for *free()*.

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

```
int main () {  
  
    char *title;  
  
    title = (char *) malloc(15);  
  
    strcpy(title, "c programming");  
    printf("Before modification : String = %s, Address = %u\n", title, title);  
  
    title = (char*) realloc(title, 30);  
  
    strcpy(title,"C Programming Language");  
    printf("After modification : String = %s, Address = %u\n", title, title);  
  
    free(title);  
  
    return(0);  
  
}
```