NAGARJUNA COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MODULE – 5

printf(" STRINGS & POINTERS ");

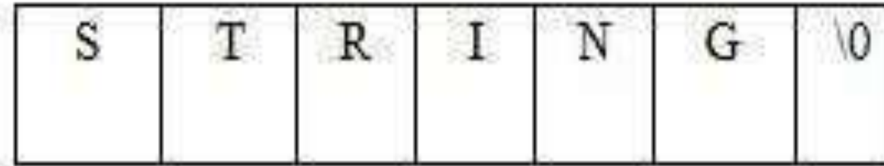
Strings

- **In C programming, array of characters or collection of characters is called a string. A string always recognized in double quotes. A string is terminated by a null character /0.**

  For example:

  “String”

- **Here, “String” is a string. When, compiler encounters strings, it appends a null character /0 at the end of string.**



String declaration :-

In C, a string is a one dimensional character array, terminated by null.

Syntax:

char string_name[size];

The size determines the number of characters in the string name.

Example:

1. **char name[20];**
2. **char city[15];**

A null character is specified as '\0' for the reason the size of the string must be equal to maximum number of characters plus one.

String initialization :-

Character arrays may be initialized when they are declared. C permits character array to be initialized in either of the form

Static char city[10]=”New Delhi”; (or)

Static char city[10]={‘N’,’e’,’w’,‘ ’,’D’,’e’,’l’,’h’,’i’};

The city is 10 elements long i.e. the string “New Delhi” contains 9 characters and one element space is provided for the null terminator.

Example :-

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main ()
{
         char str1[20];
         charstr2[20];
         printf(“Enter First Name”);
         scanf(“%s”,&str1);
         printf(“Enter last Name” );
         scanf(“%s” ,&str2);
        puts(str1);
        puts(str2);
}
```

String Handling :-

These String functions are:

- **strlen()**
- **strupr()**
- **strlwr()**
- **strcmp()**
- **strcat()**
- **strcpy()**
- **strrev()**

printf(“continue”);

1. strcat( ): Used to join 2 strings.

Syntax:

Strcat(string1, string2);

String1 and string2 are 2 character arrays. String2 is appended to string1. By removing null character at the end of string1 and placing string2 from there. So string1 size should be large and string2 remains unchanged.

Example: strcat(s1,”good”);

2. strcmp( ):It compares 2 strings identified by the arguments and has a value 0 if they are equal. If not, it has the numeric difference between the first non-matching characters in the string.

Syntax:

strcmp(string1,string2);

Where string1 and string2 may be string variables or string constants.

Example: 1.strcmp(name1,name2);

2. strcmp(name1,”john”);

printf(“continue”);

3. strcpy( ):It works like string assignment operator.

Syntax:

Strcpy(string1,string2);

Assigns the contents of string2 to string1. string2 may be a character array variable or a string constant.

Example: strcpy(city,” Hyderabad”);

Will assign “Hyderabad” to city.

4. Strlen( ): It counts and returns the number of characters in a string excluding the null terminator.

Syntax:

Strlen(string);

Example: n=strlen(“hello”);

Returns 5 to n.

printf(“continue”);

5. Strlwr( ): It converts any uppercase letters in the string to the lowercase letters.

Syntax:

Strlwr(string);

Example: strlwr(“COMPUTER”);

It returns computer

6. Strupr( ): It converts any lowercase letters in the string to the uppercase letters.

Syntax:

Strupr(string);

Example: strlwr(“computer”);

It returns COMPUTER

Strlen :-

```
size_t strlen(const char *str);
```

The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed.

The strlen() function is defined in <string.h> header file

printf(“continue”);

```
#include <stdio.h>
#include<string.h>
int main()
{
    char a*20+=”Program”
    char b*20+=,‘P’,’r’,’o’,’g’,’r’,’a’,’m’,’\0’-;
    char c[20];
    printf(“Enter string: “);
    gets(c);
    printf(“Length of string a = %d \n”, strlen(a));
    printf(“ length of a string of string b = %d \n”, strlen(b));
    printf(“Length of string c= %d \n”, strlen(c));
    return 0;
}
```

Example on all manipulation function

```
1.  #include <stdio.h>
2.  #include<conio.h>
3.  #include<string.h>
4.  #include<ctype.h>
5.  #include<stdlib.h>
6.  int length(char str[]);
7.  void reverse(char str[]);
8.  int palindrome(char str[]);
9.  void copy(char str1[], char str2[]);
10. int compare(char str1[], char str2[]);
11. void concat(char str1[], char str2[]);
12. void search(char str1[], char str2[]);
13. void count(char str1[]);
14. void main()
15. {
16.     char a[100], b[100];
17.     int result, option;
18.     do
19.     {
20.         printf("\n1.Length of a string");
21.         printf("\n2.Reverse the Given String");
22.         printf("\n3.Check for Palindrome");
23.         printf("\n4.Copy");
24.         printf("\n5.String Comparison");
25.         printf("\n6.String Concatenation");
26.         printf("\n7.String Searching");
27.         printf("\n8.Counting of Words,Characters &
27.         Special Characters");
```

printf(“continue”);

```
28.  printf("\n9.Quit");
29.  printf("\n\nEnter Your Choice:");
30.  scanf("%d", &option);
31. flushall();
32. switch (option)
33. {
34.       case 1:
•          printf("\nEnter a String:");
•          gets(a);
•          result = length(a);
•          printf("\nLength of %s=%d", a, result);
•          printf("\nPress a Character");
•          getch();
•          break;
35.       case 2:
•          printf("\nEnter a String:");
•          gets(a);
•          reverse(a);
•          printf("\nResult=%s", a);
•          printf("\nPress a Character");
•          getch();
•          break;
```

printf(“continue”);

36. case 3:

```
printf("\n Enter a String:");
gets(a);
result = palindrome(a);
if (result == 0)
printf("\nNot a palindrome");
else
printf("\nA palindrome");
printf("\nPress a Character");
getch();
break;
```

37. case 4:

```
printf("\nEnter a String:");
gets(a);
copy(b, a);
printf("\nResult=%s", b);
printf("\nPress a Character");
getch();
break;
```

printf(“continue”);

38. case 5:

- printf("\nEnter 1st string:");
- gets(a);
- printf("\nEnter 2nd string:");
- gets(b);
- result = compare(a, b);
- if (result == 0)
- printf("\nboth are same");
- else if (result > 0)
- printf("\n1st>2nd");
- else
- printf("\n1st<2nd");
- printf("\nPress a Character");

- getch();
- break;

39. case 6:

- printf("\nEnter 1st string:");
- gets(a);
- printf("\nEnter 2nd string:");
- gets(b);
- concat(a, b);
- printf("\nresult=%s", a);
- printf("\nPress a Character");
- getch();
- break;

printf(“continue”);

```
40.     case 7:
•         printf("\nEnter 1st string:");
•         gets(a);
•         printf("\nEnter 2nd string:");
•         gets(b);
•         search(a, b);
•         printf("\nPress a Character");
•         getch();
•         break;
41.     case 8:
•         printf("\nEnter a string:");
•         gets(a);
•         count(a);
•         printf("\nPress a Character");
•         getch();
•         break;
42.     default:
•         printf("\nInvalid Choice:");
•         break;
43.     }
44.     } while (option != 9);
45. }
46.  int length(char str[]) {
47. int i = 0;
48. while (str[i] != '\0’)
49.  i++;
```

printf(“continue”);

```
50.   return (i);
51.   }
52.      void reverse(char str[])
53.     {
54.           int i, j;
55.           char temp;
56.           i = j = 0;
57.           while (str[j] != '\0’)
58.           j++;
59.           j--;
60.          while (i < j) {
61.          temp = str[i];
62.              str[i] = str[j];
63.              str[j] = temp;
64.              i++;
65.              j--;
66.        }
67.   }
68.   int palindrome(char str[]) {
69.   int i, j;
70.   i = j = 0;
71.   while (str[j] != '\0’)
72.   j++;
73.   while (i < j) {
74.   if (str[i] != str[j - 1])
75.   return (0);
```

printf(“continue”);

```
76.      i++;
77.      j--;
78.   }
79    return (1);
80.  }
81.  void copy(char str2[], char str1[]) {
82.  int i = 0;
83.  while (str1[i] != '\0') {
84.  str2[i] = str1[i];
85.  i++;
86.  }
87.  str2[i] = '\0’;
88.  }
```

```
89.      int compare(char str1[], char str2[]) {
90.      int i;
91.      i = 0;
92.      while (str1[i] != '\0') {
93.      if (str1[i] > str2[i])
94.      return (1);
95.      if (str1[i] < str2[i])
96.      return (-1);
97.       i++;
98.    }
99.    return (0);
100.  }
```

printf("continue");

```
101.	void concat(char str1[], char str2[]) {
102.	int i, j;
103.	i = 0;
104.	while (str1[i] != '\0’)
105.	i++;
106.	for (j = 0; str2[j] != '\0'; i++, j++)
107.	str1[i] = str2[j];
108.	str1[i] = '\0’;
109.	}
110.	void search(char str1[], char str2[]) {
111.	int i, j, lena, lenb;
112.	for (lena = 0; str1[lena] != '\0'; lena++);
113.	for (lenb = 0; str2[lenb] != '\0'; lenb++);
114.	for (i = 0; i <= lena - lenb + 1; i++) {
115.	for (j = 0; str1[i + j] == str2[j] && str2[j] != '\0'; j++);
116.	if (str2[j] == '\0’)
117.	printf("\nString found at location : %d", i + 1);
118.	}
119.	}
120.	void count(char str[]) {
121.	int words = 0, characters = 0, spchar = 0, i;
122.	for (i = 0; str[i] != '\0'; i++) {
123.	if (isalnum(str[i]) && (i == 0 || !isalnum(str[i - 1])))
124.	words++;
125.	characters++;
126.	if (!isalnum(str[i]) && !isspace(str[i]))
127.	spchar++;
128.	}
```

printf(“continue”);

```
129.   printf("\nNo of characters = %d", characters);
130.   printf("\nNo of special characters = %d", spchar);
131.   printf("\nNo of words = %d", words);
132.   }
```

Pointer :-

Pointer Definition and syntax

A pointer is a variable whose value is the address of another variable,

i.e., direct address of the memory location.

Syntax:

Data_type *variable_name;

- **Asterisk is called as Indirection Operator. It is also called as Value at**

  Address Operator

- **It Indicates Variable declared is of Pointer type.**

pointer_name must follow the rules ofidentifier.

Example of pointers:-

```
int       *ip;
double *dp;
float *fp;
char *ch;

    /* pointer to an integer*/
/* pointer to a double */
/* pointer to a float */
/* pointer to a character */
```

Diff Between pointer and normal variable:

```
 int*ptr;
int ptr;
```

Pointer concept :-

```
int i;
int*j;
j=&i;
```

Variable Name ➔	I	j
Value of Variable ➔	3	65524
Address of Location ➔	65524	65522

Simple example on pointers :-

```
#include <stdio.h>
 int main()
{
int *ptr, i; i = 11;
/* address of i     is assigned to ptr */


ptr = &i;
/* show i's value using ptr variable */


printf("Value of i : %d", *ptr); return 0;
}
```

OUTPUT:

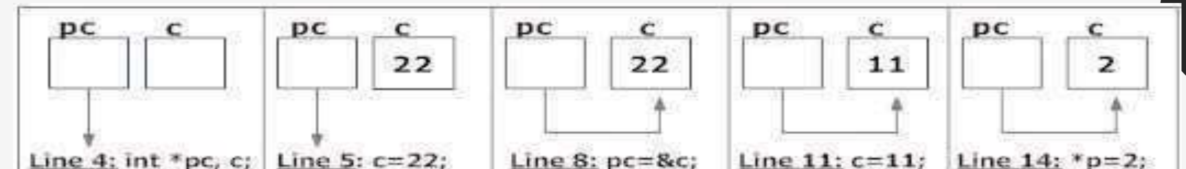
Value of i is 11.

Program on reference and de reference :-

```
#include <stdio.h>
int main()
{
  int* pc; int c; c=22;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  pc=&c;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  c=11;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  *pc=2;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  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



Pointer arithmetic :-

- **Pointer is a variable that points to a memory location. Memory addresses are numeric value that ranges from zero to maximum memory size in bytes.**
- **These addresses can be manipulated like simple variables. You can increment, decrement, calculate or compare these addresses manually.**
- **C language provides a set of operators to perform arithmetic and comparison of memory addresses.**
- **Pointer arithmetic and comparison in C is supported by following operators**
- **Increment and decrement ++ and --**
- **Addition and Subtraction + and -**
- **Comparison <, >, <=, >=, ==, !=**

printf("continue");

Pointer increment and decrement

- **Increment operator when used with a pointer variable returns next address pointed by the pointer. The next address returned is the sum of current pointed address and size of pointer datatype**
- **Or in simple terms, incrementing a pointer will cause the pointer to point to a memory location skipping Nbytes from current pointed memory location. Where N is size of pointer data type.**
- **Similarly, decrement operator returns the previous address pointed by the pointer. The returned address is the difference of current pointed address and size of pointer data type.**

Pointer addition and subtraction :-

- **Pointer increment operation increments pointer by one. Causing it to point to a memory location skipping N bytes (where N is size of pointer data type).**
- **We know that increment operation is equivalent to addition by one. Suppose an integer pointer int * ptr. Now, ptr++ is equivalent to ptr= ptr + 1. Similarly, you can add or subtract any integer value to a pointer.**
- **Adding K to a pointer causes it to point to a memory location skipping K * N bytes. Where K is a constant integer and N is size of pointer data type.**
- **Let us revise the above program to print array using pointer.**

Example :-

```
#include <stdio.h>
#define SIZE 5 int main()
{
    int arr[SIZE] = {10, 20, 30, 40, 50};
    int *ptr;
    int count;
    ptr = &arr[0]; // ptr points to arr[0] count = 0;
    printf("Accessing array elements using pointer \n");
   while(count < SIZE)
  {
        printf("arr[%d] = %d \n", count, *(ptr + count));
        count++;
   }
   return 0;
}
```

Pointer comparison:-

- **In C, you can compare two pointers using relational operator. You can perform six different type of pointer**

comparison <, >, <=, >=, == and !=.

- ***Note: Pointer comparison compares two pointer addresses to which they point to, instead of comparing their values.***

- **Pointer comparisons are less used when compared to pointer arithmetic. However, I frequently use pointer comparison when dealing with arrays.**
- **Pointer comparisons are useful, If you want to check if two pointer points to same location.**

Example :-

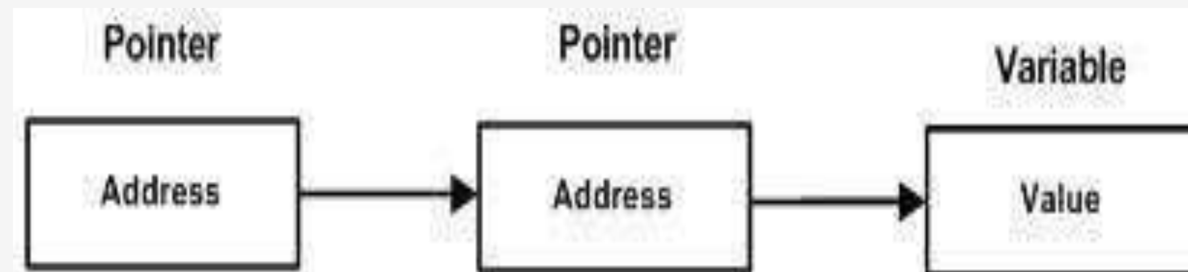
```
int main()
{
    int num = 10;
    int *ptr1 = &num;
    int *ptr2 = &num;
    if(ptr1 == ptr2)
    {
        // ptr1 points to num
        // ptr2 also points to num
        // Both pointers points to same memory location
        // Do some task
    }
    return 0;
}
```

Pointer to pointer

- **Pointers are used to store the address of other variables of similar data type. 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;
```

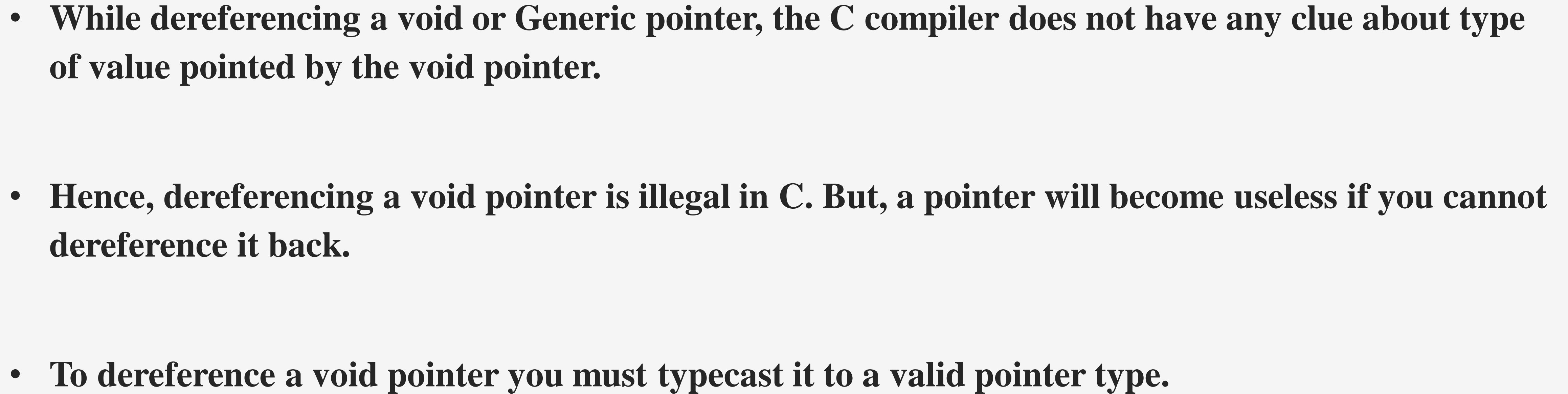
Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:



Simple program :-

- #include <stdio.h> int main()
- {
- int a = 10;
- int *p1; int **p2;
- //this can store the address of variable
- /*this can store 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 address stored by p2 = %u\n", *p2);
- printf("Value at 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;
- }

printf(“continue”);



- While dereferencing a void or Generic pointer, the C compiler does not have any clue about type of value pointed by the void pointer.
- Hence, dereferencing a void pointer is illegal in C. But, a pointer will become useless if you cannot dereference it back.
- To dereference a void pointer you must typecast it to a valid pointer type.

Array of pointer :-

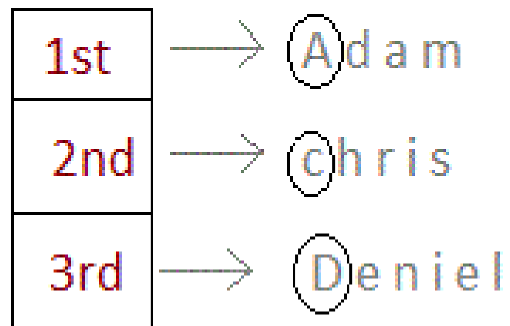
- **In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.**

- **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.**
- **An array of pointers in C: it sets each pointer in one array to point to an integer in another and then prints the values of the integers by dereferencing the pointers (printing the value in memory that the pointers point to).**

EXAMPLE :-

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

Using Pointer



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

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;
}
```

Dynamic memory :-

- **Dynamically allocate memory in your C program using standard library functions:**

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

- **To allocate memory dynamically, library functions are**
- **malloc()**
- **calloc()**
- **realloc()**
- **free()**

are used. These functions are defined in the <stdlib.h> header file.

1. malloc() :-

- The name "malloc" stands for memory allocation.
- The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example:

```
ptr = (float*) malloc(100 * sizeof(float));
```

- The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.
- The expression results in a NULL pointer if the memory cannot be allocated.

2. calloc() :-

- **The name "calloc" stands for contiguous allocation.**
- **The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.**

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

- **The above statement allocates contiguous space in memory for 25 elements of type float.**

3. free() :-

- **Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.**

Syntax of free()

```
free(ptr);
```

- **This statement frees the space allocated in the memory pointed by ptr.**

Example :-

```
#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));
    // if memory cannot be allocated
if(ptr == NULL)
{
    printf("Error! memory not allocated.");
    exit(0);
}
printf("Enter elements: ");
for(i = 0; i < n; ++i)
{
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}
printf("Sum = %d", sum);
// deallocating the memory free(ptr);
return 0;
}
```