Unit 7
Pointers

Pointer: Introduction

Pointer is a variable that hold the address of another variable. All the pointer operations are done through two operators: '*' (star) and '&' (ampersand). '&' (also known *address of operator*) is a unary operator that returns a memory address of a variable where as '*' (also known as *value at address* operator) returns value stored at a memory location stored in a pointer.

Features of Pointer:

- It saves the memory space
- Execution time is faster because data are manipulated with the address.
- It efficiently handles the string data.
- It is used with data structure so it is useful for representing two dimensional and multi-dimensional arrays.
- It can allocate memory dynamically i.e. it can assign and release the memory space at run time.

Declaration of pointer:

Syntax:

```
data_type *ptr_variable;
```

Example:

```
int *ptr;
```

It declares the variable ***ptr*** as a pointer variable that points to an integer data type i.e. it points to the variable that holds the integer value.

Initialization of pointer:

Syntax:

```
ptr_variable = &variable;
```

Example:

```
ptr  = &x;
```

Address of (&) Operator:

It is possible to obtain the address of a variable by using the address of operator (&). The following program demonstrates the use of the operator:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=10;
clrscr();
printf("\n Address of the number =  %u", &num);
printf("\n Value of number is = %d", num);
getch();
}
```

Output:

Address of the number = 65524
Value of number is = 10

Here the expression **&num** returns the address of the variable **num**. **&** is "address of" operator. Hence it is displayed in monitor by using **%u**, an unsigned integer.

Void pointer:

Let us consider following declaration.

```
float num =10.678;
int *ptr;
ptr = &num;
```

The statement `ptr = &num` results an error during compilation of the program. This can be overcome by `void` pointer.

A void pointer is a general purpose pointer that can hold address of a variable of any data types i.e. this pointer do not have any type associated with this and can hold the address of any type of variables. It can be illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=10;
float x = 3.14;
void *ptr;
ptr = &x;
printf("\n %f", *(float *)ptr);
ptr = &num;
printf("\n %d", *(int *)ptr);
getch();
}
```

NULL Pointer:

A null pointer is a pointer, which does not point anywhere. A null pointer is a pointer of any type (int, float, char, etc) that doesn't point to any memory location. The integer value zero is used to represent a null pointer.

For example:

```
int *np = 0;
```

It is different from a macro NULL. A NULL is a macro defined in the header file `"stdio.h"` that is used to represent a null pointer in the source code. A NULL macro has a value zero associated with it.

```
int *np = NULL;
```

Assigning NULL makes np as a null pointer.
A null pointer is assigned with an integer value that cannot be used to point to a memory location but an uninitialized pointer can be used to point to a memory location.

WAP to perform basic Arithmetic operation of any two numbers, using pointer variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,sum,sub,prod;
int *x,*y;
clrscr();
printf("\n Input any two numbers:- ");
scanf("%d%d",&a,&b);
x=&a;
y=&b;
sum = *x + *y;
sub = *x - *y;
prod = *x * *y;
printf("\n Sum = %d, Subtraction = %d and Product = %d", sum,sub,prod);
getch();
}
```

WAP to find factorial value of a given number, by using pointer variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num,i;
long fact = 1;
int *ptr;
clrscr();
printf("\n Input a number:- ");
scanf("%d",&num);
ptr = &num;
for(i=1;i<=*ptr;i++)
   fact*=i;
printf("\n Factorial = %ld", fact);
getch();
}
```

Pointer and Array

There is a strong relationship between arrays and pointers. Any operations that can be achieved by array can also be done with pointers because an array name by itself is an address, or pointer value.

Let us consider one dimensional array named as "num[10]" then

- The address of the first array element can be expressed as either ***&num[0]*** or ***num***.
- Similarly, the address of the second array element be ***&num[1]*** or ***(num+1)*** and so on.
- In general, the address of array element ***(i+1)*** will expressed as ***&num[i]*** or as ***(num+i)***

Handling of 1D array:

There are varieties of ways to access the elements of a 1-D array.
The array elements can be accessed by using the indirection operator directly to the address of a particular element. The following program illustrates the process:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, num[50], sum = 0;
printf("\n How many numbers:- ");
scanf("%d",&n);
printf("\n Input %d numbers:- ",n);
for(i=0;i<n;i++)
   scanf("%d",&num[i]);
for(i=0;i<n;i++)
   sum = sum + *(&num[i]);
printf("\n Sum of the series = %5d", sum);
getch();
}
```

The operator `&num[i]` returns the address of the i^{th} element and `*(&num[i])` returns the value at that address.

Another way to access the array elements is to assign the base address to a pointer variable and access the elements through this pointer variable. It can be shown as in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, num[50], sum = 0,*ptr;
ptr = num;                          // assigns the address of first element
```

```
printf("\n How many numbers:- ");
scanf("%d",&n);
printf("\n Input %d numbers:- ",n);
for(i=0;i<n;i++)
   scanf("%d",(num+i));
for(i=0;i<n;i++)
   sum = sum + *(num+i);
printf("\n Sum of the series = %5d", sum);
getch();
}
```

The relation of pointer and array is presented in the following sizes:

Address of array elements	
Array elements	Equivalent pointer notation
&num[0]	Num
&num[1]	(num+1)
……	……
&num[i]	(num+i)

Value of array elements	
Array elements	Equivalent pointer notation
num[0]	*num
num[1]	*(num+1)
……	……
num[i]	*(num+i)

Example:

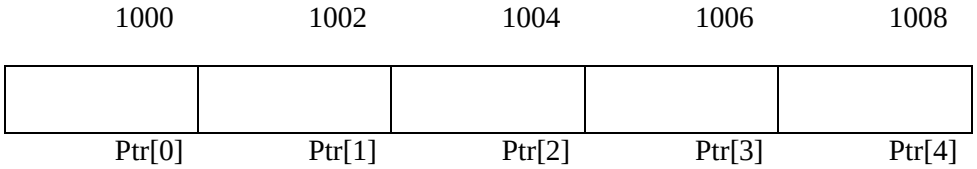
```
#include<stdio.h>
#include<conio.h>
void main()
{
int *num[10],i;
clrscr();
printf("\n Input any 10 number:- ");
for(i=0;i<10;i++)
   scanf("%d",(num+i));
printf("\n The numbers are:\n");
for(i=0;i<10;i++)
   printf("%5d",*(num+i));
getch();
}
```

Handling one dimensional array with array of pointer:

The elements of an array can be accessed by array of a pointer. An array of pointer can be declared as:

```
data type *pointer_variable[size];
int *ptr[5];
```

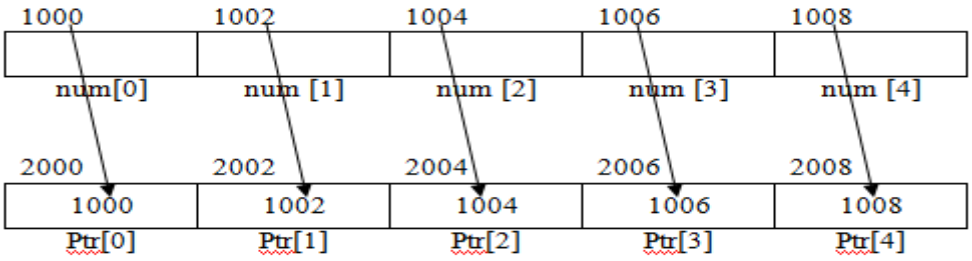
It can be shown as:



It can be demonstrated in following program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, num[50], *ptr[50], sum = 0;
printf("\n How many numbers:- ");
scanf("%d",&n);
for(i=0;i<n;i++)
   ptr[i] = &num[i];      // Assigns the address of elements to pointer variable.
printf("\n Input %d numbers:- ",n);
for(i=0;i<n;i++)
   scanf("%d",ptr[i]);
   // scanf("%d", &num[i]) or scanf("%d", (num+i));
for(i=0;i<n;i++)
   sum = sum + *num[i];
printf("\n Sum of the series = %5d", sum);
getch();
}
```

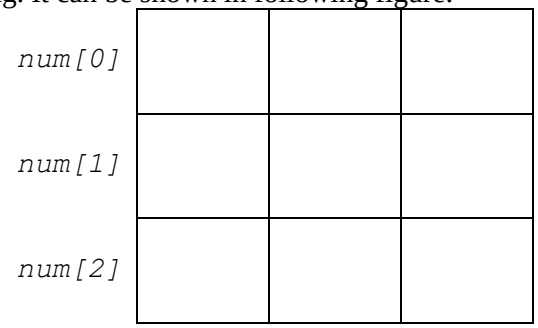
The memory arrangement:



Two dimensional arrays with pointer:

We can refer elements of a two dimensional array using pointer. In 2-D array, each row can be thought as a separate 1-D array.

For example: `int num[3][3]` can be though as three 1-D arrays named: `num[0]`, `num[1]` and `num[2]` each of 3 elements long. It can be shown in following figure.



We know that the name of 1-D array gives the base address. Thus `num[0]` gives the base address of 0th row (In above case it is 1000). `num[1]`gives the base address of 1st row (1006 in above case) and `num[2]`gives the base address of

2nd row (1012 in above case) i.e. *num[i]* gives the base address of i^{th} row. The following program shows the output of the base address of three 1-D array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, num[][3]={1, 2, 3, 4, 5, 6, 7, 8,9};
for(i=0;i<3;i++)
   printf("\n The base address of %d th row is %u", i, num[i]);
getch();
}
```

Output:

```
The base address of 0th row is 1000
The base address of 1th row is 1006
The base address of 2th row is 1012
```

Now, the question is how to access a particular element of a 2-D array using pointer? Let us consider that we have to access the element *num[2][1]*. As discussed in earlier, *num[2]* would be the base address of 2nd 1-D array (Say 1012), then ***1012+1*** i.e. ***num[2]+1*** would give the address of 1014 and the value at this address can be obtained by an expression ****(num[2]+1)***. Similarly

**(num[0]+0)* would give the value at address 1000
**(num[0]+1)* would give the value at address 1002
**(num[0]+2)* would give the value at address 1004
**(num[1]+0)* would give the value at address 1006
**(num[1]+1)* would give the value at address 1008
**(num[1]+2)* would give the value at address 1010
**(num[2]+0)* would give the value at address 1012
**(num[2]+1)* would give the value at address 1014
**(num[2]+2)* would give the value at address 1016

From above expression it can be concluded that the value of *ith* row and *jth* column can be obtained by the expression **(num[i]+j).*

The following program demonstrated how to receive and print the element of 2-D array with pointer.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, j, r, c, num[5][5];
printf("\n Input size of a matrix:- ");
scanf("%d%d", &r, &c);

printf("\n Input elements of %d x %d matrix", r, c);
for(i=0;i<r;i++)
   {
   for(j=0;i<c;j++)
      scanf("%d", &(num[i]+j));
   }

printf("\n The elements of %d x %d" matrix are:\n, r, c);
for(i=0;i<r;i++)
   {
   for(j=0;i<c;j++)
      printf("%5d", *(num[i]+j));
   printf("\n");
   }
```

```
getch();
}
```

Returning multiple values from function using pointer

Using a call by reference intelligently we can make a function return more than one value at a time, which is not possible ordinarily. This is shown in the program given below.

```
#include<stdio.h>
float areaperi(int, float *, float *);
main( )
{
int radius ;
float area, perimeter ;
printf ( "\n Enter radius of a circle " ) ;
scanf ( "%d", &radius ) ;
areaperi ( radius, &area, &perimeter ) ;
printf ( "Area = %f", area ) ;
printf ( "\n Perimeter = %f", perimeter ) ;
}

float areaperi ( int r, float *a, float *p )
{
*a = 3.14 * r * r ;
*p = 2 * 3.14 * r ;
}
```

And here is the output...

Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000

Pointer arithmetic

Pointer arithmetic is one of the powerful features of C language. All the operation cannot be operated on pointer variables. We can:

- Compare pointers
- Increment / decrement
- Add and subtract integer values from them (i.e. an integer can be added or subtracted from pointer.
- Subtract two pointers of same type.

The following operations are not possible on the pointer variables:

- Addition of two pointers
- Multiplication of a pointer with a constant
- Multiplication of pointers
- Division of a pointer with a constant
- Division of two pointers.

a) Address assignment:

A pointer variable can be assigned the address of an ordinary variable. It is illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=10, *ptr;
ptr = &num;
printf("\n The address of variable num is %u", ptr);
getch();
}
```

b) Assignment of one pointer to another:

A pointer variable can be assigned the content of another variable provided with both pointers points to the same data type. It is illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num[] = {10,20,30,40, 50}, *ptr1, *ptr2;
ptr1 = &num[0];
ptr2 = ptr1;
getch();
}
```

c) Addition of an integer constant to a pointer:

An integer constant value (x) can be added to a pointer variable. It is illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=10, *ptr;
ptr = &num;
printf("\n The value of pointer before operation is = %u", ptr);
ptr = ptr + 2;
printf("\n The value of pointer after operation is = %u", ptr);
getch();
}
```

Output:

```
The value of pointer before operation is = 1000
The value of pointer after operation is = 1004
```

d) Subtraction of an integer constant from a pointer:

An integer constant value (x) can be subtracted from a pointer variable. It is illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=10, *ptr;
ptr = &num;
printf("\n The value of pointer before operation is = %u", ptr);
ptr = ptr - 2;
printf("\n The value of pointer after operation is = %u", ptr);
getch();
}
```

Output:

```
The value of pointer before operation is = 1200
The value of pointer after operation is = 1196
```

e) Subtraction of one pointer from another:

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements. It is illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num[] = {10,20,30,40, 50}, *ptr1, *ptr2;
```

```
ptr1 = &num[0];
ptr2 = &num[4];
printf("\n The difference of first and last elements is = %u", ptr2-ptr1);
getch();
}
```

Output:

```
The difference of first and last elements is = 4
```

f) Comparison of two pointer variables:

Two pointer variables can be compared provided both variables to point to elements of the same data type using relation operator. It is illustrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num[] = {10,20,30,40, 50}, *ptr1, *ptr2;
ptr1 = &num[0];
ptr2 = &num[4];
if(ptr1==ptr2)
   printf("\n They points to the same elements");
else
   printf("\n They doesn't points to the same elements");
getch();
}
```

Pointer for strings

There are so many ways to print a string by the concept of pointer.

The simple way is the concept of pointer variable. In this method to print each character of a string variable *text* constant we can use the expression ***(text+i)**. It can be shown in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char text[]="Computer Science";
int i;
printf("\n The given string is :- ");
for(i=0;i<17;i++)
   pritf("%c",*(text+i));
getch();
}
```

Here, *text + i* returns the address of the i^{th} element and ***(text+i)** returns the value of the i^{th} element. But this method is not efficient because we cannot edit the string using pointer constant.

It can be eliminated by another method called pointer variable. It can be shown in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char text[]="Computer Science",*ptr;
int i;
ptr=text; //Assigning the base address
printf("\n The given string is :- ");
for(i=0;i<17;i++)
   pritf("%c",*(text+i));
getch();
}
```

Array of pointers to strings

There is a drawback to store an array of strings, in that the sub arrays that hold the string must all be the same length. So, that memory space is wasted when strings are shorter than the sub arrays.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char *weeks[7]= {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday"};
int i;
printf("\n The days in a week are :- ");
for(i=0;i<7;i++)
   puts(weeks[i]);
getch();
}
```

Double indirection

C is very powerful programming language that can also support a mechanism to create pointer of pointer. It is also known as chain of pointer. It can be demonstrated in following program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num = 10, *ptr1, **ptr2, ***ptr3;
clrscr();
ptr1 = &num;
ptr2 = &ptr1;
ptr3 = &ptr2;
printf("\n The value = %d, %d, %d", num, *ptr1, **ptr2, ***ptr3);
getch();
}
```

In above program *num* is an ordinary variable and holds a value *10*. *ptr1* is a pointer variable that is holding address of *num* i.e. *ptr1* is a pointer of *num. ptr2* is another pointer variable for pointing to a ordinary variable pointer *ptr1*. Similarly *ptr2* is another pointer variable that points to another pointer variable *ptr1*.

Dynamic Memory allocation

There are two ways to allocate memory for a program. They are:

a) Compile time / design time / Static memory allocation
b) Run time / dynamic memory allocation

Compile time memory allocation:

In compile time memory allocation, a fixed number of bytes are pre-reserved. Array is compile time memory allocation. The main drawbacks of this memory allocation are:

- If the user's data is less than reserved bytes, there is wastage of memory bytes and the unused bytes cannot be assigned to variable of other programs.
- If the user's data exceeds the reserved bytes, then memory overflow occurs there.

Dynamic memory allocation (DMA)

The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array we have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required. It is an extendable memory allocation method. The memory bytes reserved during the run time can be released when the program is not using it.

The process of allocating memory at run time i.e. the ability of calculating and assigning the memory space required by the variable in a program is known as dynamic memory allocation. It provides flexibility in adding, deleting or rearranging data items at run time i.e. this technique allows us to allocate additional memory space or to release unwanted space at run time, thus optimizing the use of storage space.

There are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution and they are: *calloc(), malloc(), realloc()* and *free()* and their function prototype are in *stdlib.h.*

calloc()

The calloc () takes two arguments, the first argument is the number of items and the second argument is size of each item for which the memory is to be allocated. It also initializes the allocated memory to zero.

Syntax: *calloc(number of items, size of each item)*

Example: *calloc(n, sizeof(int));*

Program to demonstrate dynamic memory allocation to the size of an integer array:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int *size, *ptr, num;
clrscr();
printf("\n How many numbers:- ");
scanf("%d",&num);
size=(int *)calloc(num, sizeof(int));
printf("\n Input %d numbers:- ",num);
for(ptr=size; ptr <size+num;ptr++)
   scanf("%d",ptr);
printf("\n The contents are:- ");
for(ptr=size;ptr<size+num;ptr++)
   printf("%5d",*ptr);
free(size);
getch();
}
```

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form. The malloc() function takes one argument that specifies the total memory to be allocated. So to allocate memory for n integer, the argument is passed as n*sizeof(int). It does not initialize the allocated memory and is faster than calloc(). The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of malloc()

ptr =(cast-type*)malloc(byte-size)

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

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

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
```

```
int n,i,*ptr,sum=0;
printf("\n 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",sum);
free(ptr);
return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient, then, we can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, ptr is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
int *ptr,i,n1,n2;
printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
     printf("%u\t", ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=(int *)realloc(ptr,n2);
for(i=0;i<n2;++i)
     printf("%u\t", ptr+i);
getch();
}
```

free()

Dynamically allocated memory with either *calloc()* or *malloc()* does not get return on its own. The programmer must use free() explicitly to release space.

Syntax of free()

```
free(ptr);
```

This statement causes the space in memory pointer by ptr to be de-allocated.