Overview of C Programming languages

Data types in C

C languages contains four basic data types:

int , float , char and double

There are three qualifiers that can be applied to **int** s

short , long and unsigned

The actual maximum sizes implied by short int , long int or int vary from machine to machine. An unsigned integer is an integer that is always positive and follows the arithmetic laws of modulo 2**n, where n is the number of bits in an integer

A variable declaration in C specifies two things. First , it specifies the amount of storage that must be set aside for objects declared with that type. Second, it specifies how data represented by strings of bits are to be interpreted.

Pointers in C

C allows the programmer to reference the location of objects as well as object (that is , the contents of those locations) themselves. For example, if x is declared as integer, &x refers to the location that has been set aside to contain x. &x is called a pointer.

It is possible to declare a variable whose data type is a pointer and whose possible values are memory locations. For example, the declarations

```
int *pi;
float *pf;
char *pc;
```

declare three pointer variables. **pi** is a pointer to an integer number, **pf** is a pointer to a float number and **pc** is a pointer to a character. The asterisk indicates that the values of the variables being declared are pointers to values of the type specified in the declaration.
The notation *pi in C refers to the integer at the location referenced by the pointer pi. The statement

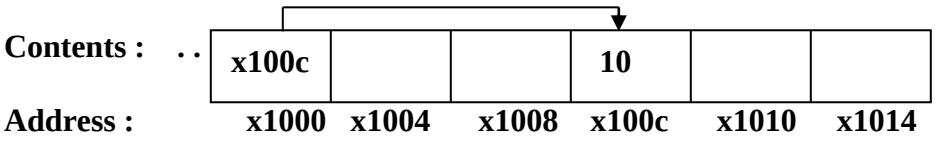
```
x = *pi  ;
```

assigns the value of the integer to the integer variable x.

A data item is stored in memory in one or more adjacent storage locations depending upon its type. The address of a data item is the address of its first storage location. This address can be stored in another data item and manipulated in a program.

The address of a data item is called a **pointer** to the data item, and a variable that holds an address is called a **pointer variable**.

Thus , if **ip** is a pointer variable that contains the address of **i**, an **int**, this situation can be shown below.



&i	i	ip	*ip
x100c	10	x100c	10

For each type of object that can be declared in C, a corresponding type of pointer can be declared. To indicate that a variable contains a pointer , an asterisk is included before the name of the object in the type declaration. Thus,

type *identifier ;

Note that the declaration allocates space for the named pointer variable, but not for what it points to.

For example , to declare **cp** to be a pointer to an object of type **char**, **ip** a pointer to an object of type **int**, and **dp** a pointer to an object of type **double**,

```
char   *cp;
int    *ip;
double *dp;
```

The **asterisk** indicates that the values of the variables being declared are pointers to values of the type specified in the declaration rather than objects of that type.

The notation ***ip** in **C** refers to the integer at the location referenced by the pointer **pi**.

A pointer value may be assigned to another pointer of the same type. For example , in the program fragment

```
int  i , j ,  * ip  ;
i = 123;
ip = &i ;
j = *ip ;
*ip = 0 ;
```

After assigning **123** to variable **i**, next statement assigns the address of variable **i** to **ip**, next statement assigns the **value** at address **ip**, that is , **123** to **j**, and finally last statement assigns **0** to **i** since ***ip** is the same as **i**.

Note that the two statements

```
ip = &i;
j = *ip ;
```

Are equivalent to the single assignment

```
j = *(&i) ;
```

or to the assignment

```
j = i ;
```

That is, the address operator & is the inverse of the de-referencing operator *.

The form of an initialization of a pointer variable is

```
type   *identifier  =  initializer;
```

The initializer must either evaluate to an address of previously defined data of appropriate type or it can be the NULL pointer. For example the declaration

```
float  *fp  = NULL ;
short  s ;
short  *sp = &s ;
char  c[10] ;
char  *cp = &c[4];
```

initialize **fp** to **NULL** and initialize **sp** to the address of **s** and initialize **cp** to the address of the fifth element of the array **c**.

The following is a simple program that illustrates declaration, initialization, assignment , and de-referencing of pointers.

```
#include  <stdio.h>
int main(void)
{
  int  i , j = 1 ;
  int  *jp1 ,  *jp2 = &j ;        /* jp2 points to j */

  jp1 = jp2 ;               /* jp1 also points to j * /
  i = *jp1 ;                /*  i gets the value of  j  */
  *jp2 = *jp1 + i ;         /*  i  is added to j */
  printf("i = %d ,  j = %d , *jp1 =  %d , *jp2 = %d \n",
          i ,  j ,  *jp1 ,  *jp2);
  Return 0;
}
```

This program prints:

i = 1, j = 2 , *jp1 = 2 , *jp2 = 2

If **pi** is the pointer to an integer, then **pi+1** is the pointer to the integer immediately following the integer ***pi** in memory, **pi – 1** is the pointer to the integer immediately preceding ***pi , pi + 2** is the pointer to the second integer following ***pi**, and so on.

In C we know that parameters are passed by using call by value principle. In a call by value, values of the arguments are used to initialize parameters of the called function, but the addresses of the arguments are not provided to the called function. Therefore, any changes in the value of parameter in the called function is not reflected in the actual variables in the calling function.

For example, consider the following program segment and function.

```
x = 3;
printf("x = %d\n",x);
funct(x);
printf("x = %d\n", x);
..
..
void funct(int  y)
{
  ++ y;
  printf("y = %d\n", y);
}
```

as an output program generate following

```
x = 3
y = 4
x = 3
```

If we wish to use **funct** to modify the value of x, we must pass the address of **x** as follows:

```
x = 3;
printf("x = %d\n",x);
funct(&x);
printf("x = %d\n", x);
..
..
void funct(int *y)
{
  ++ (*y);
  printf("y = %d\n", *y);
}
```

program generates output as

```
x = 3
y = 4
x = 4
```

Important note : Pointers are the mechanism used in C to allow a called function to modify variables in a calling function.

To change the values of variables in the calling environment, other languages provide the "call-by-reference" mechanism. In this section we show how to use of addresses of variables as arguments to functions can produce the effect of call-by-reference.

Similar program can be done using referencing the addresses**(call by reference** principle). This is a way of passing addresses(references) of variables to a function that then allows the body of the function to make changes to the values of variables in the calling environment.

```
#include <stdio.h>
void exchange(int *, int *);
int main(void)
{
int i ,  j ;
  printf("Enter two numbers for j and j  \n");
  scanf("%d %d", &i, &j);
  printf("Main   : i = %d  j = %d \n", i, j);
  exchange( &i , &j);
  printf("Main   : i = %d  j = %d \n", i, j);
  return 0;
}
void exchange(int *m  , int  *n)
{
int t;
  t = *m , *m =*n ,  *n = t ;
 printf("Exchange : i = %d   j = %d \n", m , n );
}
```

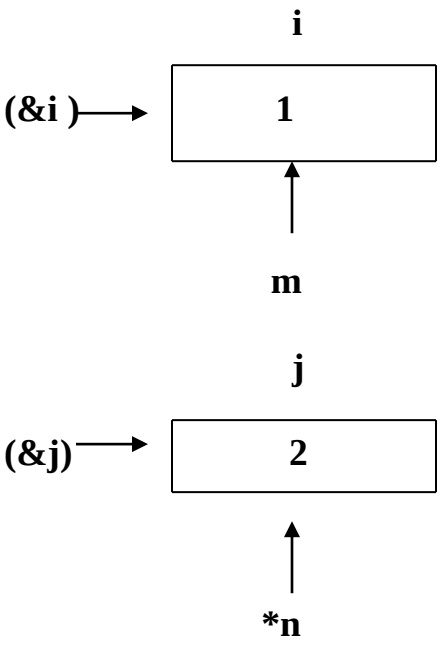
if given numbers are 1 and 2 then output will be

```
Main        :  i  =  1        j  =  2
Exchange    :  i  =  2        j  =  1
Main        :  i =   2        j  =  1
```

In this program when **exchange(&i, &j)** is called, the address of the **i** and **j** are used in the formal parameters such that ***m** points to **i** and ***n** points to **j.**

We can trace what happens as the function is executed.

t = *m **1**
***m = *n**
***m = t**



Upon completion, the variable **t** is discarded and the variables **i** and *j* have had their values changed to those shown as below

If **pi** is a pointer to an integer, then

pi + 1

is the pointer to the integer immediately following the integer *pi in memory,

pi – 1

is the pointer to the integer immediately preceding *pi,

pi+2

is the pointer to the second integer following *pi, and so on.

For example, suppose that a particular machine uses byte addressing, an integer requires four bytes, and the value of **pi** happens to be 100. (that is , **pi** points to the integer ***pi** at location 100). Then the value of **pi-1** is 96, the value of **pi+1** is 104 and the value of **pi+2** is 108. The value of ***(pi-1)** is the contents of the four bytes 96, 97 , 98 and 99 interpreted as an integer; the value of ***(pi+1)** is the contents of bytes 104, 105 and 106 and 107 interpreted as an integer; and the value of ***(pi+2)** is the integer at bytes 108, 109, 110 and 111.

Similarly, if the value of the variable **pc** is 100(pc is a pointer to a character) and a character is one byte long, **pc-1** refers to location 99, **pc+1** to location 101, and **pc+2** to location 102. Thus the result of pointer arithmetic in C depends on the base type of the pointer.

The difference between *pi + 1 , which refers to 1 added to the integer *pi, and *(pi+1), which refers to the integer following(at the next address) the integer at location pi.

Arrays in C

One dimensional array

A finite ordered set of homogeneous elements is called as array. Finite means that there is a specific number of elements in the array. Ordered means it uses a successive memory locations. Homogeneous means that all the elements in the array must be of the same type.

Two basic operations are:

Extraction(determine the location):using index or subscript element location will be determined.

a[i]

Storing(assignment)

a[i]=x;

Lower bound : 0
Upper bound : high-1
Range :number of elements in the array(upper-lower + 1)

```
#define NUMELTS 100
int a[NUMELTS];
for(i=0;i<NUMELTS;i++) a[i]=0;
```

Arrays as Parameters.

Every parameter of a C function must be declared within the function. However, the range of an one dimensional array parameter is only specified in the main program. This is because in C new storage is not allocated for an array parameter. Rather, the parameter refers to the original array that was allocated in the calling program.

For example :
In the main program we might have written

```
#include <stdio.h>
float avg(float a[], int); /* prototype */
#define RANGE  10

int main()
{
float a[RANGE];int i;
for( i=0;i<RANGE;i++)
scanf("%f",&a[i]);
printf("Average=%f",avg(a,RANGE));
return 0;
}

float avg(float a[], int size)
{float sum=0;int i;
 for(i=0;i<size;i++)
sum+=a[i];
return(sum/size);
}
```

Note that if the array range is needed in the function, it must be passed separately. Since an array variable in C is a pointer, array parameters are passed **by reference** rather than by value. So only based address of the array is passed.

Passing an array **by reference** rather than **by value** is more efficient in both time and space.

Alternative solution can be given by using pointer variables.

```
#include <stdio.h>
float avg(float *, int);  /* prototype */
int main()
{
#define RANGE  3
float a[RANGE];
for(int i=0;i<RANGE;i++)
scanf("%f",&a[i]);
printf("Average=%f",avg(a,RANGE));
return 0;
}
float avg(float *a, int size)
{float sum=0;
 for(int i=0;i<size;i++)
sum+=*(a+i);
return(sum/size);
}
```

Accessing arrays with pointers

Pointer Manipulation

Arrays in C are quite loosely implemented. Not only can they be accessed via an index, they may also just as easily be accessed via a pointer variable.

Accessing a cell via a pointer

It was explained before that an element of an array is accessed through an index. Since each cell has an address, it should be possible to access any cell through an address reference.

Array Position	month [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
	0	1	3	2	6	5	4	8	7	14	3	0
Address	430	434	438	442	446	450	454	458	462	466	470	474

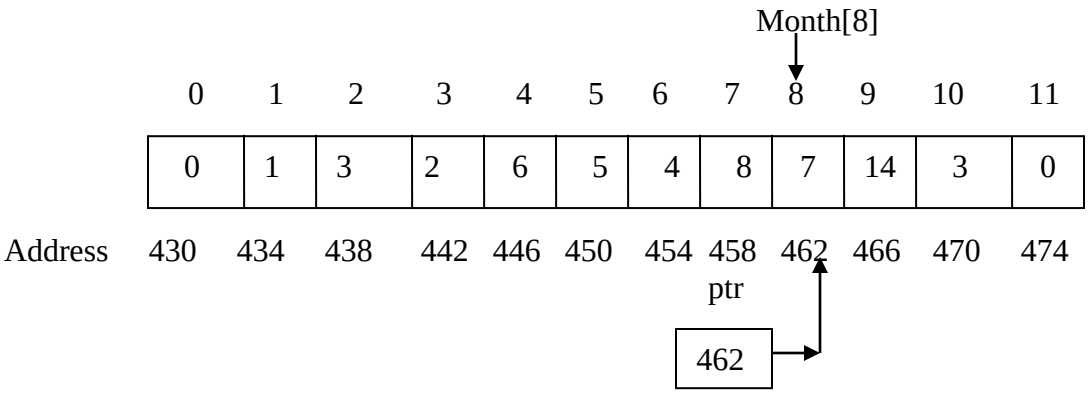
Suppose you wish to access the cell at position 8: To access this cell a pointer, a pointer variable will be needed besides the array declaration:

```
int month[12];
int * ptr;
```

To make *ptr* point to the cell at position 8(i.e., month[8]) is a simple matter of assigning the address of *month[8]* to *ptr*;

```
ptr = &month[8];
```

Now *ptr*, assigned the address 462 , points to the cell at that address ,as shown in figure



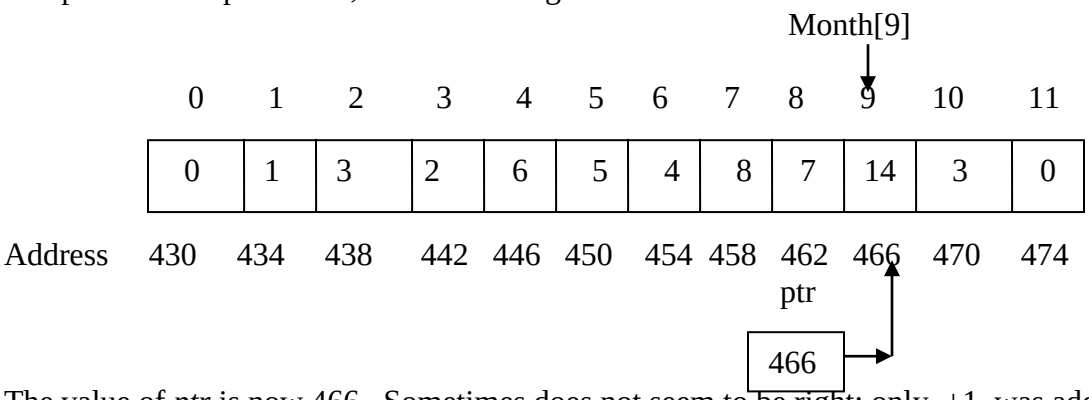
To display the content of this cell via the pointer, simply use the dereferencing operator *, which makes the pointer go to the cell to which the operator points:

```
printf("element at position 8 is %d \n",*ptr);
```

Moving to another cell via a pointer
In the above example, to move the pointer to the next cell, simply increment the pointer by 1:

ptr=ptr+1; or ptr++;

The pointer now position 9, as shown in figure



The value of *ptr* is now 466 . Sometimes does not seem to be right: only +1 was added to it: it should now be 463. The explanation is that pointer arithmetic works a little differently. In the following expression:

ptr=ptr+1;

the 1 tells C to move the pointer to the next cell. It does not mean to move to the next address, which is 463. Since a cell occupies 4 bytes, the content of the pointer is actually incremented by 4.

Base address
In the example showed that the statement ***ptr=&month[0]*** sets the pointer to the address of the first element. This address is called *base address* of the array. The fallowing statement accomplish the same thing:

ptr = month;

This implies that the name of the array, *month*, also holds the base address of that array.

Pointer Initialization
The most widely used applications of pointer initialization at declaration time are as follows.

- A pointer variable set to the base of an array
- A pointer variable set to zero

```
int month[12]={0};
int *ptr = month;
int *listptr=NULL;
```

Here *ptr* points to the array month as soon as it is declared, and *listptr* is set to zero. A pointer set to zero means that it points to nothing.

Dereferencing and indexing at the same time

The elements of an array may be referenced to by using more complex pointer notations than shown so far in this section. For example, consider the following three statements that display the tenth element of the array month via a pointer reference:

ptr = &month[8];
ptr=ptr+1;
printf("%f",*ptr);

The second step can be merged with the third one:

ptr =&month[8];
printf("%f",*(ptr+1));

and it will display 14

A pointer can be made to move back and forth through the array. The following segment illustrate this.

```
int month[12] = { 0,1,3,2,6,5,4,8,7,14,3,0 };
int *ptr, elm;
ptr = month;          /*  ptr points to the element  month[0] */
printf("*ptr = %d\n", *ptr);
ptr = month + 3;      /*  ptr points to the element month[3]  */
printf("*ptr = %d\n", *ptr);
elm = *(month + 3);       /*  same  as  elm = month[3]    */
printf("elm= %d\n",elm);
       ptr = month;
elm = *(ptr + 3);     /*  same  as  elm = month[3]    */
printf("*ptr = %d\n", *ptr);
ptr--;
elm = *(ptr + 5);     /*  same  as  elm = month[4]    */
printf("elm = %d\n", elm);
```

	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	3	2	6	5	4	8	7	14	3	0
Address	430	434	438	442	446	450	454	458	462	466	470	474

Example

```
int  list[]={11 , 91 , 66 , 99 , 77};
int  *ptr = &list[3];

for(   ; ptr>list;  ptr--)
   printf("%d",*ptr);
```

output of the following program is

99 66 91

Example:
Following program list the total sum of rainy days of each month. Program first read the monthly rainy days for each month.

```
#include <stdio.h>
int main()
{
int month[12];
int *ptr, i , sum;
printf("enter the monthly rainy days this year ");
for(ptr=&month[0];ptr<=month[11];ptr++)
scanf("%d",ptr);
sum=0;
for(ptr=&month[0], i=0;ptr<=month[11];ptr++, i++)
{
printf("In month %d , %d  rainy days \n", i+1, *ptr);
sum+=*ptr;
}
printf("There were %d rainy days \n",sum);

}
```

In this solution a pointer will run through the array, so index variable is not needed and has been replaced with a pointer variable declaration.
Inside the for loop, the pointer *ptr* runs from the first cell to the twelfth cell of the array.

```
for(ptr=&month[0];ptr<=month[11];ptr++)
```

In the fallowing statement *&month[0]* is the address of the cell at position 0; *&month[11]* is the address of the cell at position 11(the twelfth cell); and the ++ operator tells C to move the pointer to the next cell.
To access the element at the cell pointed to by *ptr*, it is necessary to use the dereferencing operator(argument *ptr* to the *scanf()*)

Passing Array addresses to functions

It is known that an array is sent by reference, meaning that the base address of the array is copied to the parameter.

Consider the following function that simply sums up the elements of an array.

```
int  sum (int  list[], int size)
{
        int  i , sum=0;
        for(i=0; i<size ; ++i)
                sum+=list[i];
        return sum;
}
```

Since an array is nothing more than a pointer, it should be possible to pass this function *sum()* not only an array, but also a pointer to any one cell from an array. The following program illustrates this

```
main()
{
 int ar[10]={2,-2,8,1,0,-8,6,3,9,12};
 int s;
 s=sum(ar,10);       /*   s is the sum of ar[0]+… + ar[9] */
 s=sum(ar,5);        /*   s is the sum of ar[0]+… + ar[4] */
 s=sum(&ar[0],8); /*   s is the sum of ar[0]+… + ar[7] */
 s=sum(&ar[2],4); /*   s is the sum of ar[2]+… + ar[5] */
 s=sum(ar+2,8);    /*   s is the sum of ar[2]+… + ar[9] */
}
```

It uses the function *sum()* above to calculate the sum of selected subsets from the array *ar*.

More Pointer Examples using one dimensional array

Example 1

```
#include <stdio.h>
void main()
{
        float arr[]={8,5,-7,12,0,3};
        float *ap=&arr[4];
        for(;ap>arr;ap--)
        printf("%f\n",*ap);
}
```

Output will be

0
12
-7
5

Example 2

```
#include <stdio.h>
float sum(float *ptr, int size)
{       float isum = 0;
        for (int i = 0; i < size; i++)
        {
                isum += *++ptr;
                //isum += ++*ptr;
                /*  see what happens when you replace with the following statements
                isum+=(*ptr)++;
                isum+=--*ptr;
                */
                printf("%d \t%f\t %f\n", i, *ptr, isum);
        }
        return isum;
}
void main()
{       float arr[] = { 8,5,-7,12,0,3 };
        printf("SUM=%f\n", sum(arr, 5));

}
```

Two dimensional arrays.

Following declaration defines a new array containing three elements. Each of these elements is itself an array containing five integers.

int a[3][5];
Such an array is called a **two-dimensional array.**

	column 0	**column 1**	**column 2**	**column 3**	**column 4**
row 0 →	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 1 →	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 2 →	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Two-dimensional arrays

Two dimensional arrays logically might be represented as matrix. But physically represented as follows.