

POINTERS

The & and * operators.

Consider the declaration,

```
int i = 3;
```

i location name

3 value at location

6485 location no.(address)

This declaration tells the C compiler to:

- (a) Reserve space in memory to hold the integer value.
- (b) Associate the name I with this memory location.
- (c) Store the value 3 at this location.

```
main()
{
    int i=3;
    printf("\nAddress of i=%u",&i);
    printf("\nValue of i=%u",i);
}
```

Output:

Address of i = 6485

Value of i = 3

The ‘&’ operator used in this statement in C’s ‘address of’ operator.

The **&i** returns address of the variable i , which in this case happens to be 6485

```
main()
{
    int i=3;
    printf("\nAddress of i=%u",&i);
    printf("\nValue of i=%u",i);
    printf("\nValue of i=%u",*(&i));
}
```

Output:

Address of i = 6485

Value of i = 3

Value of i = 3

i

j

3

6485

6485

3276

```
int i;
```

```
int *j;
```

```
main()
{
    int i=3;
    int *j;
    j=&i;
    printf("\nAddress of i=%u",&i);
    printf("\nAddress of i=%u", j);
    printf("\n Address of j=%u",&j);
    printf("\n Value of j=%d", j );
    printf("\nValue of i=%d", i);
    printf("\nValue of i=%d",*(&i));
    printf("\nValue of i=%d", *j);

}
```

Output:

Address of i= 6485

Address of i= 6485

Address of j= 3276

Value of j= 6485

Value of i= 3

Value of i= 3

Value of i= 3

i

3

6485

```
int i;
```

```
int *j;
```

```
int **k;
```

j

6485

3276

k

3276

7234

```
main()
{
    int i=3;
    int *j;
    j=&i;

    printf("\nAddress of i=%u",&i);
    printf("\nAddress of i=%u", j);
    printf("\n Address of i=%u", *k);
    printf("\n Address of j=%u", &j );
    printf("\nAddress of j=%u", k);
    printf("\n Address of k=%u",(&k));

    printf("\nValue of j=%u", j);
    printf("\nValue of k=%u", k);
    printf("\n Value of i=%d",i);
    printf("\nValue of i=%d",*(&i));
    printf("\nValue of i=%d", *j);
    printf("\nValue of i=%d", **k);

}
```

Address of i= 6485

Address of i= 6485

Address of i= 6485

Address of j= 3276

Address of j= 3276

Address of k= 7234

Value of j= 6485

Value of k= 3276

Value of i= 3

Value of i= 3

Value of i= 3

Value of i= 3

c

65

1004

l

Bin eq of 54

2008

2009

a

Binary

Equivalent

of

3.14

7006

7007

7008

7009

cc

1004

1962

ii

2008

7602

aa

7006

9118

```
main()
{
    char c, *cc;
    int i, *ii ;
    float a, *aa ;
    c='A' ;
    i=54;
    a= 3.14;
    cc=&c;
    ii =&i;
    aa= &a ;
    printf("\nAddress of cc=%u",cc);
    printf("\nAddress of ii=%u", ii);
    printf("\n Address of aa=%u", aa);
    printf("\n Value of c=%c", *cc );
    printf("\nValue of i=%d", *ii);
    printf("\n Value of a=%f",*aa);
}
```

Output

Address of cc= 1004

Address of ii= 2008

Address of aa=7006

Value of c= A

Value of i=54

Value of a=3.14

```
main()
{
int *p,i ;
p=&i;
*p=10;
printf(“%p%p”,p, &i);
printf(“\n %d%d\n”,*p,i);
return 0;
}
```

Main Memory

Address	Memory	variable
0023FF66		p
0023FF70		i

Soon after declaration

Address	Memory	variable
0023FF66	0023FF70	p
0023FF70		i

After executing $p=&i$

Address	Memory	variable
0023FF66	0023FF70	p
0023FF70	10	i

*After executing $*p=10$*

```
main()
{
    int i=3,*x;
    float j=1.5,*y;
    char k='c',*z;
    printf("\n Value of i = %d",i);
    printf("\n Value of j = %d",j);
    printf("\n Value of k = %d",k);
    x= &i;
    y= &j;
    z= &k;
    printf("\n Original Value of x = %d",x);
    printf("\n Original Value of y= %d",y);
    printf("\n Original Value of z= %d",z);
    x++;
    y++;
    z++;
    printf("\n New Value of x = %d",x);
    printf("\n New Value of y= %d",y);
    printf("\n New Value of z= %d",z);
}
```

Consider the i,j and k are stored in memory at address 1002,2004 and 5006.

Output:

Value of i =3

Value of j = 1.5

Value of k = c

Original Value of x = 1002

Original Value of y= 2004

Original Value of z= 5006

New Value of x =1004

New Value of y= 2008

New Value of z= 5007

Pointer Arithmetic

```
int a=5,*pi=&a;
```

```
float b=2.2,*pf=&b;
```

```
char c='x',*pc=&c;
```

Suppose the address of variable a,b and c are 1000,4000,5000 respectively.

```
pi++ or ++pi;
```

```
pi=pi-3;
```

```
pi=pi+5;
```

```
pi-- or --pi;
```

```
pf++ or ++pf;
```

```
pf=pf-3;
```

```
pf=pf+5;
```

```
pf-- or --pf;
```

```
pc++ or ++pc;
```

```
pc=pc-3;
```

```
pc=pc+5;
```

```
pc-- or --pc;
```

pi++ or pi++;	pi=1000+2=1002
pi=pi-3;	pi=1002-3*2=996
pi=pi+5;	pi=996+5*2=1006
pi-- or pi--;	pi=1006-2=1004

pf++ or ++pf;	pf=4000+4=4004
pf=pf-3;	pf=4004-3*4=3992
pf=pf+5;	pf=3992+5*4=4012
pf-- or pf--;	pf=4012-4=4008

pc++ or pc++;	pc=5000+1=5001
pc=pc-3;	pc=5001-3=4998
pc=pc+5;	pc=4998+5=5003
pc-- or pc--;	pc=5003-1=5002

```
main()
{
    int a=5;
    int *p;
    p=&a;
    printf("Value of p=Address of a=%u",p);
    printf("Value of p=%u",++p);
    printf("Value of p=%u",p++);
    printf("Value of p=%u",--p);
    printf("Value of p=%u",p--);
    printf("Value of p=%u",p);
}
```

Output:

Value of p=address of a=1000

Value of p=1002

Value of p=1002

Value of p=1002

Value of p=1002

Value of p=1000

Precedence of Increment/Decrement Operator

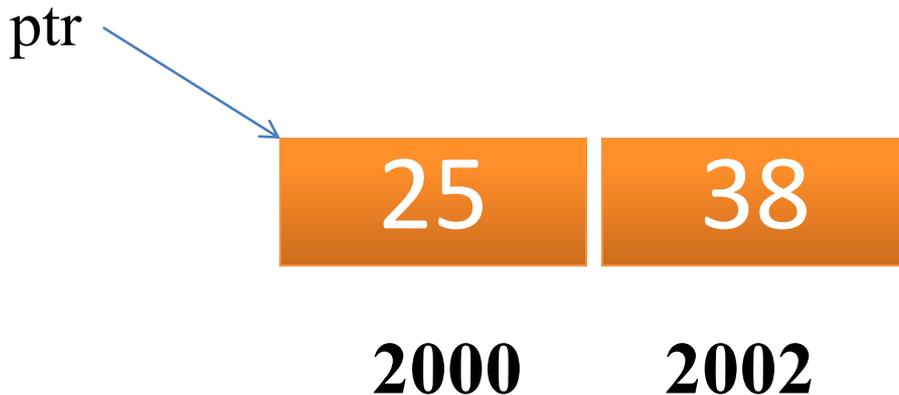
(i) $x = *ptr++;$

(ii) $x = *++ptr;$

(iii) $x = ++*ptr;$

(iv) $x = (*ptr)++;$

Given value and addresses of ptr, find the values of x given above.



(i) `x=*ptr++;` It is also equivalent to `*(ptr++)`
`x=*ptr;`
`ptr=ptr+1;`
Value of `x=25`, Address contained in `ptr=2002`, `*ptr=38`

(ii) `x=*++ptr;` It is also equivalent to `*(++ptr)`
`ptr=ptr+1;`
`x=*ptr;`
Value of `x=25`, Address contained in `ptr=2002`, `*ptr=38`

(iii) `x=++*ptr;` It is also equivalent to `++(*ptr)`
`*ptr=*ptr+1;`
`x=*ptr;`
Value of `x=26`, Address contained in `ptr=2000`, `*ptr=26`

(iv) `x=(*ptr)++;`
`x=*ptr;`
`*ptr=*ptr+1;`
Value of `x=25`, Address contained in `ptr=2000`, `*ptr=26`

```
int main()
{
    int *p, i=10;
    p=&i;
    i++;
    printf(“%d %d\n”,i,*p);
    i=i+10;
    printf(“%d %d\n”,i,*p);
    *p=*p+10;
    printf(“%d %d\n”,i,*p);
    (*p)++;
    printf(“%d %d\n”,i,*p);
    printf(“Enter a value for i\n”);
    scanf(“%d”,p);
    printf(“%d %d”,i,*p);
    return 0;
}
```

Output:

11 11

21 21

31 31

32 32

Enter a value for i

56

56 56

```

int main()
{
    int *p, i=10, j=90;
    p=&j;
    printf("Addresses of i and j are \n");
    printf("%u %u\n", &i, &j);
    i++;
    printf("%d %d\n", i, *p);
    i=i+10;
    printf("%d %d\n", i, *p);
    *p=*p+10;
    printf("%d %d\n", i, *p);
    *p++;
    printf("%d %d\n", i, *p);
    return 0;
}

```

Address	Memory	Variable
0023FF68		p
0023FF6C	90	j
0023FF70	10	i

Output

Address of i and j are:

0023FF70 0023FF6C

11 90

21 90

21 100

21 21

Pointers and One Dimensional Arrays

5000	5002	5004	5006	5008
1	2	3	4	5
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]

```
int arr[5] = {1,2,3,4,5};
```

Here 5000 is the address of first element.

Since each element (type int) takes 2 bytes,so address of next element is 5002, and so on.

The address of the first element of array is known as the **base address**.

The relationship between pointer and array

1. Element of an array are stored in consecutive memory locations.
2. The name of an array is a constant pointer that points to the first element of the array, i.e it stores the address of the first element , also known as the base address of array.
3. According to pointer arithmetic, when a pointer variable is incremented,it points to the next location of its base type.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
5	10	15	20	25

1.

2000	2002	2004	2006	2008
------	------	------	------	------

```
main()
{
int arr[5] = {5,10,15,20,25};
int i ;
for(i=0;i<5;i++)
{
printf("Value of arr[%d]=%d\t", i, arr[i]);
printf("Address of arr[%d]=%u\n", i, &arr[i]);
}
}
```

Output

Value of arr[0]=5	Address of arr[0]=2000
Value of arr[1]=10	Address of arr[0]=2002
Value of arr[2]=15	Address of arr[0]=2004
Value of arr[3]=20	Address of arr[0]=2006
Value of arr[4]=25	Address of arr[0]=2008

(arr+i) denotes the address of &arr[i]

(arr+1) denotes & arr[1] i.e address of arr[1]

arr	→	points to 0 th element	→	&arr[0]	→	2000
arr+1	→	points to 1st element	→	&arr[1]	→	2002
arr+2	→	points to 2nd element	→	&arr[2]	→	2004
arr+3	→	points to 3rd element	→	&arr[3]	→	2006
arr+4	→	points to 4 th element	→	&arr[4]	→	2008

*arr or *(arr+0) denotes the 0th element of array.

*(arr+i) → arr[i]

*arr	→	Value of 0 th element	→	arr[0]	→	5
*(arr+1)	→	Value of 1st element	→	arr[1]	→	10
*(arr+2)	→	Value of 2nd element	→	arr[2]	→	15
*(arr+3)	→	Value of 3rd element	→	arr[3]	→	20
*(arr+4)	→	Value of 4 th element	→	arr[4]	→	25

```
main()
{
int arr[5] = {5,10,15,20,25};
int i ;
for(i=0;i<5;i++)
{
    printf("Value of arr[%d]=%d\t", i, * (arr+i));
    printf("Address of arr[%d]=%u\n", i, arr+i);
}
```

The previous program can also be written as
above

arr[i] is equivalent to *(arr+i)

*(arr+i) is same as *(i+arr)

*(i+arr) is equivalent to i[arr]

arr[i] is equivalent to i[arr]

```
main()
{
int arr[5] = {5,10,15,20,25};
int i=0 ;
for(i=0;i<5;i++)
{
    printf("Value of arr[%d]=", i);
    printf("%d\t",arr[i]);
    printf("%d\t",*(arr+i));
    printf("%d\t",*(i+arr));
    printf("%d\t",i[arr]);
    printf("Address of arr[%d]= %u\n", i,&arr[i]);
}
```

Output

Value of arr[0]=5 5 5 5

Address of arr[0]=2000

Value of arr[1]=10 10 10 10

Address of arr[1]=2002

Value of arr[2]=15 15 15 15

Address of arr[2]=2004

Value of arr[3]=20 20 20 20

Address of arr[3]=2006

Value of arr[4]=25 25 25 25

Address of arr[4]=2008

Subscripting Pointer Variables

ptr

2000

4500

arr[0]

arr[1]

arr[2]

arr[3]

arr[4]

5

10

15

20

25

2000

2002

2004

2006

2008

```
int *ptr;
```

```
ptr=arr; /*we can also write ptr=&arr[0];*/
```

The name of an array is a constant pointer hence it always points to the 0th element of the array.

```
arr = &num;    /* Illegal*/
```

```
arr ++;       /* Illegal*/
```

```
arr = arr-1;  /* Illegal*/
```

But since ptr is a pointer variable, all these operations are valid for it.

```
ptr = &num;    /* Now ptr points to variable num*/
```

```
ptr ++;       /* ptr points to next location*/
```

```
ptr = ptr-1;  /* ptr points to previous location*/
```

```
main()
```

```
{
```

```
int arr[5] = {5,10,15,20,25};
```

```
int i, *p;
```

```
p=arr;
```

```
for(i=0;i<5;i++)
```

```
{
```

```
printf("Address of arr[%d]= %u %u %u %u\n", i, &arr[i], arr+i, p+i, &p[i]);
```

```
printf("Value of arr[%d]=%d %d %d %d", i, arr[i], *(arr+i), *(p+i), p[i]);
```

```
}
```

Output

Address of arr[0]=2000	2000	2000	2000
------------------------	------	------	------

Value of arr[0]= 5	5	5	5
--------------------	---	---	---

Address of arr[0]=2002	2002	2002	2002
------------------------	------	------	------

Value of arr[1]= 10	10	10	10
---------------------	----	----	----

Address of arr[0]=2004	2004	2004	2004
------------------------	------	------	------

Value of arr[2]= 15	15	15	15
---------------------	----	----	----

Address of arr[0]=2006	2006	2006	2006
------------------------	------	------	------

Value of arr[3]= 20	20	20	20
---------------------	----	----	----

Address of arr[0]=2008	2008	2008	2008
------------------------	------	------	------

Value of arr[4]= 25	25	25	25
---------------------	----	----	----

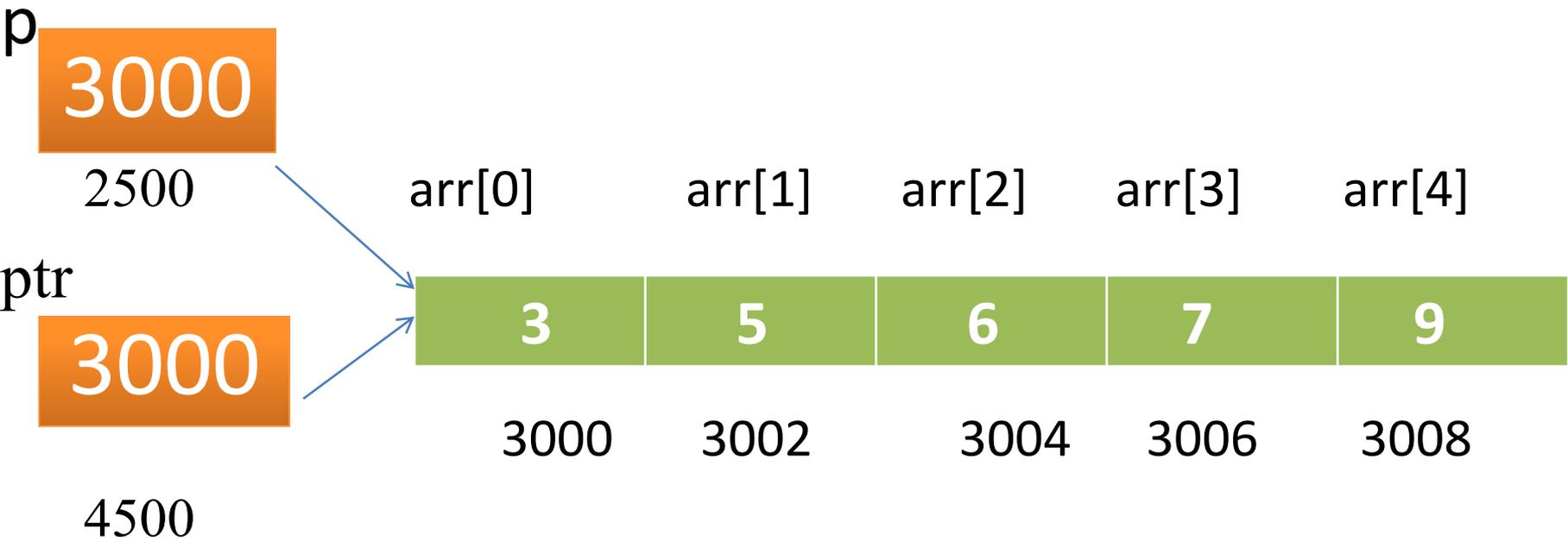
Pointer to an Array

- We declare a pointer to an array as:

```
int (*ptr)[10];
```

- Here ptr is pointer that can point to an array of 10 integers.
- It is necessary to enclose the pointer name inside parentheses.
- Here the type of ptr is ‘pointer to an array of 10 integers’.
- Pointer that points to the 0th element of array and pointer that points to the whole array are totally different.

- Here p is a pointer that points to the 0th element of array arr
- While ptr is a pointer that points to the whole array arr.
- The base type of p is 'int' while the base type of ptr is 'an array of 5 integers'.
- When we write ptr++ ,the the pointer is shifted by 10 bytes.



Program to understand **pointer to an integer** and **pointer to an array of integers**

main()

```
{  
    int *p;  
    int (*ptr)[5];  
    int arr[5];  
    p=arr;  
    ptr=arr;  
    printf("p=%u, ptr=%u\n",p,ptr);  
    p++;  
    ptr++;  
    printf("p=%u, ptr=%u\n",p,ptr);  
}
```

Output

p=3000,ptr=3000

P=3002,ptr=3010

```
main()
```

```
{  
  int arr[5]={3,5,6,7,9}  
  int *p=arr;  
  int (*ptr)[5]=arr;  
  printf("p=%u, ptr=%u\n",p,ptr);  
  printf("**p=%d, *ptr=%u\n",*p,*ptr);  
  printf("sizeof(p)=%u, sizeof(*p)=%u\n",sizeof(p), sizeof(*p));  
  printf("sizeof(ptr)=%u, sizeof(*ptr)=%u\n",sizeof(ptr), sizeof(*ptr));  
  
}
```

Output

p=3000,ptr=3000

*p=3,*ptr=3000

sizeof(p)=2, sizeof(*p)=2

sizeof(ptr)=2, sizeof(*ptr)=10

Pointer And Two Dimensional Arrays

In a 2-D array we can access each element by using two subscripts i.e the first represents the row no. and the second represents the column no.

We can access any element $\text{arr}[i][j]$ of this array using the pointer expression **$*(*(\text{arr}+i)+j)$** .

Let us consider a 2-D array $\text{arr}[3][4]$

```
int arr[3][4]={{10,11,12,13},{20,21,22,23},{30,31,32,33}};
```

	Col 0	Col 1	Col 2	Col 3
Row 0	10	11	12	13
Row 1	20	21	22	23
Row 2	30	31	32	33

Actually in 2-D array elements are stored in row major order i.e rows are placed next to each other.

Each row can be considered as 1-D array ,so a two dimensional array can be considered as a collection of 1-D array that are placed one after the another.

In other words we can say that 2-D array is an array of arrays.

arr[0][0]				arr[1][0]				arr[2][0]			
10	11	12	13	20	21	22	23	30	31	32	33
5000	5002	5004	5006	5008	5010	5012	5014	5016	5018	5020	5022

So here arr is an array of 3 elements where each element is a 1-D array of integers.

The name of the array is a constant pointer that points to 0th element of an array.

In case of 2-D arrays , 0th element is a 1-D array, so the name of a 2-D array represents a pointer to a 1-D array.

- For ex: in the above case ,arr is a pointer to 0th 1-D array and contains address 5000.
- Since arr is a ‘pointer to an array of 4 integers’, so acco. to pointer arithmetic, the expression (arr+1) will represent the address 5008 and expression (arr+2) will represent address 5016.
- So we can say arr points to the 0th 1-D array,(arr+1) points to the 1st 1-D array and (arr+2) points to the 2nd 1-D array.

arr →	10	11	12	13
arr+1 →	20	21	22	23
arr+2 →	30	31	32	33

arr- Points to 0th element of arr- Points to 0th 1-D array -5000

arr+1- Points to 1st element of arr- Points to 1st 1-D array -5008

arr+2- Points to 2nd element of arr- Points to 1st 1-D array -5016

In general we can write :

arr+i Points to ith element of arr → points to ith 1-D array .

***(arr+0) –arr[0]** Base address of 0th 1-D array → Points to 0th element of 0th 1-D array
--5000

***(arr+1)** -- Base address of 1st 1-D array → Points to 0th element of 1st 1-D array --
5008

***(arr+2)** -- Base address of 2nd 1-D array → Points to 0th element of 2nd 1-D array –
5016

***(arr+i) –arr[i]** Base address of ith 1-D array → Points to 0th element of ith 1-D array

Both (arr+i) and *(arr+i) are pointers, but their base type are different.

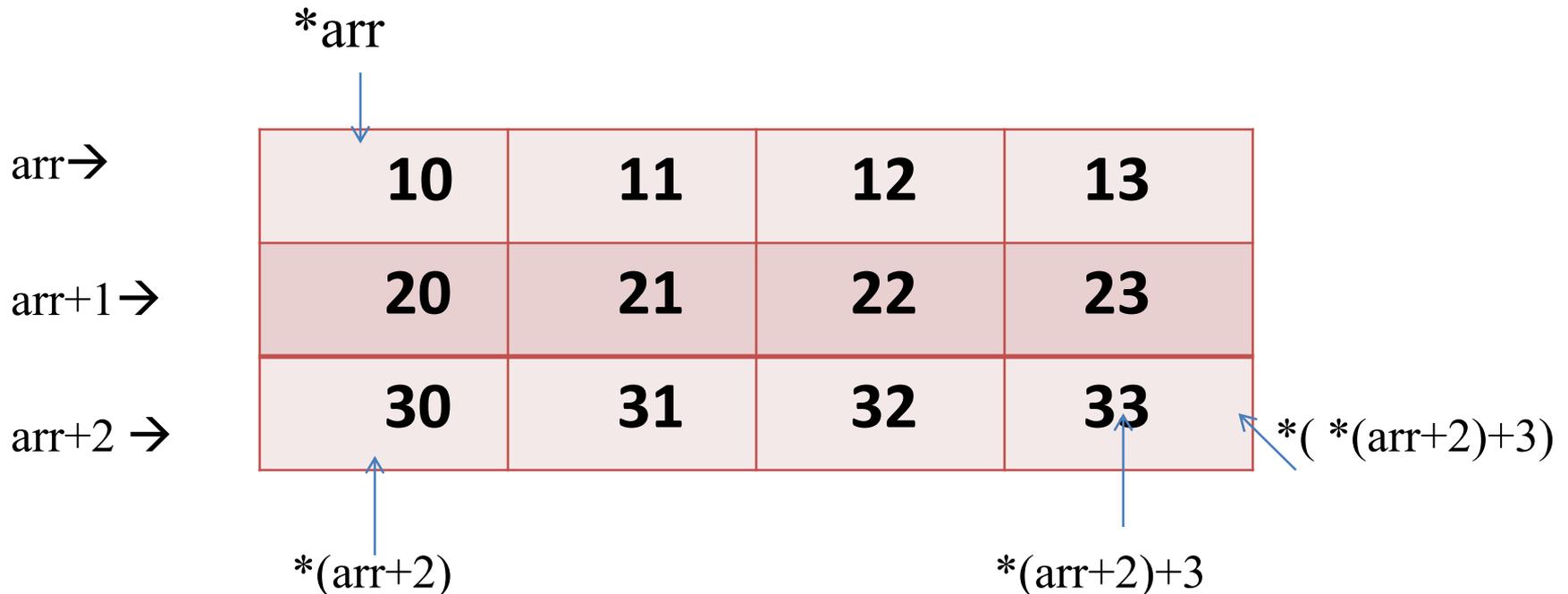
The base type of (arr+i) is ‘an array of 4 ints’.

While the base type of *(arr+i) is ‘int’ and it contains the address of 0th element of ith 1-D array, so we can get the addresses of subsequent elements in the ith 1-D array by adding integers values to *(arr+i).

For Ex: $*(arr+i)+1$ will represent the address of 1st element of i th 1-D array and $*(arr+i)+2$ will represent the address of 2nd element of i th 1-D array.

$*(arr+i)+j$ will represent the address of j th element of i th 1-D array.

- arr Points to 0th 1-D array
- $*arr$ Points to 0th element of 0th 1-D array
- $(arr+i)$ Points to i th 1-D array
- $*(arr+i)$ Points to 0th element of i th 1-D array
- $*(arr+i) + j$ Points to j th element of i th 1-D array
- $*(*(arr+i) + j)$ Represents the value of j th element of i th 1-D array



```
main()
{
    int arr[3][4]=        {{10,11,12,13} ,
                           {20,21,22,23},
                           {30,31,32,33}};

    int i,j;
    for(i=0;i<3;i++)
    {
        printf("address of %dth array =%u %u",i ,arr[i],*(arr+i));
    for(j=0;j<4;j++)

        printf("%d %d",arr[i][j],*(*(arr+i)+j));
    }
}
```

Output

Address of 0th 1-D array=5000 5000

10 10 11 11 12 12 13 13

Address of 1th 1-D array=5008 5008

20 20 21 21 22 22 23 23

Address of 0th 1-D array=5016 5016

30 30 31 31 32 32 33 33

Subscripting Pointer To An Array

```
int arr[3][4]= { {10,11,12,13} ,  
                {20,21,22,23},  
                {30,31,32,33} };
```

```
int (*ptr)[4];
```

```
ptr=arr;
```

```

main()
{
    int i, arr[3][4]=        {{10,11,12,13} ,
                              {20,21,22,23},
                              {30,31,32,33}};

    int (*ptr)[4];
    ptr=arr;

    printf(“%u %u %u”,ptr ,ptr+1,ptr+2);
    printf(“%u %u %u”,*ptr ,*(ptr+1),*(ptr+2));
    printf(“%d %d %d”,**ptr ,*(*(ptr+1)+2),*(*(ptr+2)+3));
    printf(“%d %d %d”,ptr[0] [0],ptr[1][2],ptr[2][3]);
}

```

output

5000 5008 5016

5000 5008 5016

10 22 33

10 22 33

Pointer and Three Dimensional Arrays

```
int arr[2][3][2]=      {      {      {5,10} ,  
                        {      {6,11},  
                        {      {7,12}  
                        },  
                        {      {20,30} ,  
                        {      {21,31},  
                        {      {22,32};  
                        }  
};
```

We can consider a three dimensional array to be an array of 2-D arrays i.e each element of a 3-D array is considered to be a 2- D array.

The 3-D array arr can be considered as an array consisting of two elements where each element is a 2-D array.

The name of the array arr is a pointer to the 0th element of the array,so arr points to the 0th 2-D array

<code>arr</code>	Points to 0 th 2-D array
<code>(arr+i)</code>	Points to i th 2-D array
<code>*(arr+i)</code>	Gives base address of ith 2-D array, so points to 0 th element ith 2-D array, each element of 2-D array is 1-D array
<code>*(arr+i) +j</code>	Points to j th 1-D array of ith 2-D array
<code>*(*(arr+i) +j)</code>	Gives base address of jth 1-D array of ith 2-D array, so it points to 0 th element jth 1-D array of ith 2-D array
<code>*(*(arr+i) +j)+k</code>	Points to kth element of jth 1-D array of ith 2-D array.
<code>*(***(arr+i)+j)+k)</code>	Gives the value of kth element of jth 1-D array of ith 2-D array.

```
int arr[2][3][2]=    {    {    {5,10} ,
                    {6,11},
                    {7,12}
                    },
                    {    {20,30} ,
                    {21,31},
                    {22,32};
                    }
};
```

```
int i,j, k;
for(i=0;i<2;i++)
    for(j=0;j<3;j++)
        {
            for(k=0;k<2;k++)
                printf("%d \t", *((*(arr+i)+j)+k));
            printf("\n");
        }
}
```

Output:

5 10

6 11

7 12

20 30

21 31

22 32

Array of Pointers

We can declare an array that contains pointers as its elements.

Every element of this array is a pointer variable that can hold address of any variable of appropriate type.

Syntax:

```
datatype *arrayname[size]
```

```
int *arrp[10];          /*this is known as array of pointers*/
```

Program for understanding array of pointers

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

```
main()
```

```
{
```

```
int *pa[3];
```

```
int i , a=5, b=10, c=15;
```

```
pa[0]=&a;
```

```
pa[1]=&b;
```

```
pa[2]=&c;
```

```
for(i=0;i<3;i++)
```

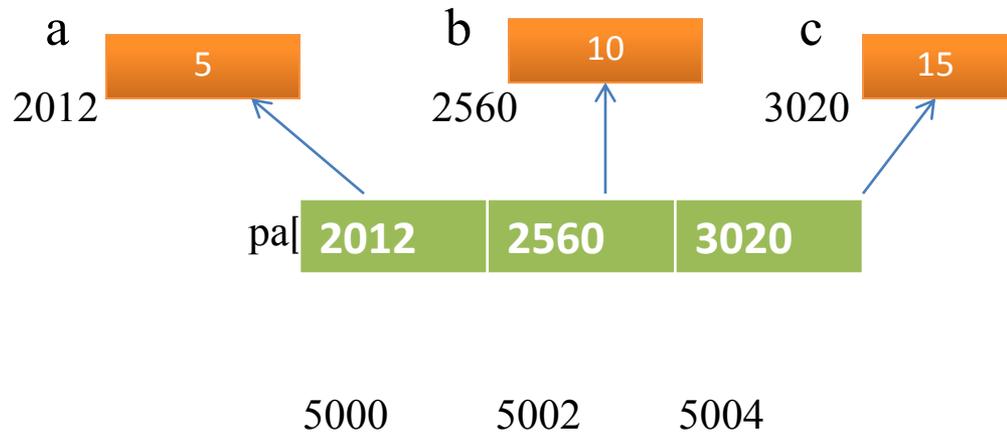
```
{
```

```
printf("pa[%d]=%u\t",i,pa[i]);
```

```
printf("*pa[%d]=%d\t",i, *pa[i]);
```

```
}
```

```
}
```



Output

```
pa[0] =2012
```

```
*pa[0] = 5
```

```
pa[1] =2560
```

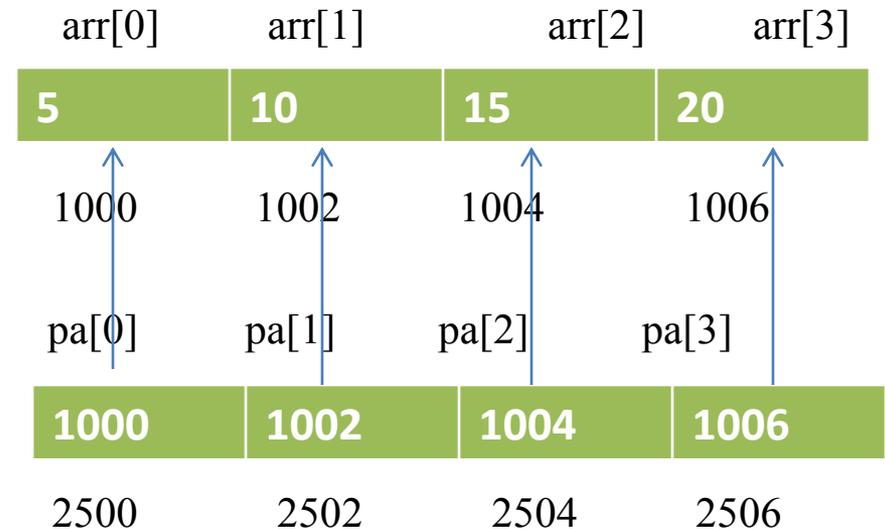
```
*pa[1] =10
```

```
pa[2] =3020
```

```
*pa[2] = 15
```

In the above program pa is declared as an array of pointers. Every element of this array is a pointer to an integer.

```
#include<stdio.h>
main()
{
    int i , arr[4]={5,10,15,20};
    int *pa[4];
    for(i=0; i<4;i++)
    pa[i]=&arr[i];
    for(i=0; i<4;i++)
    {
        printf("pa[%d]=%u\t",i,pa[i]);
        printf("*pa[%d]=%d\t",i, *pa[i]);
    }
}
```



Output

```
pa[0] =1000    *pa[0] = 5
pa[1] =1002    *pa[1] =10
pa[2] =1004    *pa[2] = 15
pa[3] =1006    *pa[3] = 20
```

Type Conversion

If the types of two operands in an assignment expression are different ,then the type of the right hand side operand is converted to the type of left hand operand.

```
main()
```

```
{
```

```
    char c1,c2;
```

```
    int i1,i2;
```

```
    c1='H' ;
```

```
    i1= 80.56; /* Demotion :float is converted to int ,only 80 is assigned to i1 */
```

```
    f1=12.6;
```

```
    c2=i1;    /* Demotion : int converted to char */
```

```
    i2=f1; /* Demotion :float is converted to int */
```

```
/* Now c2 has character with ASCII value 80, i2 is assigned value 12 */
```

```
    printf("c2=%c, i2=%d\n",c2,i2);
```

```
    f2=i1; /* Promotion : int is converted to float */
```

```
    i2=c1; /* Promotion : char is converted to int */
```

```
/* Now i2 contains ASCII value of character 'H' which is 72 */
```

```
    printf("f2=%.2f, i2=%d\n", f2, i2);
```

Output

```
c2=P, i2=12
```

```
f2=80.00, i2=72
```

Explicit type Conversion or Typecasting

```
float z;  
int x=20,y=3;  
z=x/y;
```

The value of z will be 6.0 instead of 6.66.

The syntax of cast operator is:

```
(datatype) expression  
z=(float)x/y;
```

```
main()
```

```
{
```

```
    int x=5,y=2;
```

```
    float p,q;
```

```
    p= x/y;
```

```
    printf("p=%f\n",p);
```

```
    q=(float)x/y;
```

```
    printf("q=%f\n",q);
```

```
}
```

Output

```
p=2.000000
```

```
q=2.500000
```

Dynamic Memory Allocation

The memory allocation that we have done till now was static memory allocation.

The memory that could be used by the program was fixed i.e we could not increase or decrease the size of the memory during the execution of the program .

In many applications it is not possible to predict how much memory would be needed by the program at run time.

```
int emp_no[200];
```

In an array it is must to specify the size of array while declaring so the size of this array will be fixed during run time.

Now 2 types of problem may occur:

One case can be if 50 nos of memory are required rest 150 are wasted. Similarly if 205 is the required memory and there are only 200 memory spaces then we will run shortage of memory.

To overcome this problem we use the following:

- The process of allocating memory at the time of execution is called dynamic memory allocation.
- The allocation and release of this memory space can be done with the help of some built-in functions whose prototypes are found in alloc.h and stdlib.h header files.
- These functions take memory from memory area called heap and release this memory whenever not required ,so that it can be used again for some other purposes.
- `malloc()`

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory through pointers.

Declaration: `void *malloc(size_t size);`

- This function is used to allocate memory dynamically. The argument `size` specifies the number of bytes to be allocated.
- The type `size_t` is defined in `stdlib.h` as unsigned int.

`malloc()` returns a pointer to the first byte of allocated memory.

- The returned pointer is of type `void` which can be type cast to appropriate type of pointer. It is generally used as:

```
ptr=(datatype *) malloc(specified size);
```

- Here `ptr` is a pointer of type `datatype`, and `specified size` is the size in bytes required to be reserved in memory. The expression `(datatype*)` is used to typecast the pointer returned by `malloc()`.
- For Ex:

```
int *ptr;
```

```
ptr=(int *) malloc (10);
```

`ptr`



This allocates 10 contiguous bytes of memory space and address of first byte is stored in the pointer variable ptr. This space can hold 5 integers.

```
ptr=(int *) malloc ( 5* sizeof(int));
```

If there is not sufficient memory available in heap then malloc() returns NULL. So we should always check the value returned by malloc().

```
ptr=(float *) malloc ( 10* sizeof(float));  
if(ptr == NULL)  
printf(“sufficient memory is not available”);
```

Unlike memory allocated for variables and arrays, dynamically allocated memory has no name associated with it. So it can be accessed only through pointers.

We have a pointer which points to the first byte of the allocated memory and we can access the subsequent bytes using pointer arithmetic.

Program to understand dynamic memory allocation of memory.

```
#include<stdio.h>
#include<alloc.h>
main()
{
int *p, n, i;
printf("Enter the number of integers to be entered :");
scanf("%d",&n);
p=(int *)malloc(n*sizeof(int));
if(p==NULL)
{
    printf("memory not available\n");

}
for(i=0;i<n;i++)
{
    printf("Enter an integer");
    scanf("%d",p+i);
}
for(i=0;i<n;i++)
printf("%d\t",*(p+i));
}
```

calloc()

Declaration: `void *calloc(size_t n, size_t size);`

The `calloc` function is used to allocate multiple blocks of memory.

It has two differences from `malloc()`:

1. The first one is that it takes two arguments.
2. The second one specifies the size of each block.
3. Memory allocated by `malloc()` contains garbage value while memory allocated by `calloc()` is initialized to zero.

For Ex:

```
ptr= (int *) calloc (5 , sizeof (int))
```

This allocates 5 blocks of memory, each block containing 2 bytes and the starting address is stored in the pointer variable `ptr`, which is of type `int`.

An equivalent `malloc()` would be :

```
ptr=(int *)malloc(5*sizeof(int));
```

Here we have to do the calculation by ourselves but in `calloc()` function does the calculation for us.

realloc()

Declaration: `void * realloc(void *ptr, size_t newsize);`

The function `realloc()` is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

`ptr=(int *)malloc(5*sizeof(int));`if we want to change the size of the memory block then we use `realloc()` as:

`ptr=(int *) realloc (ptr , newsize);` This statement allocates the memory space of `newsize` bytes, and the starting address of this memory block is stored in the pointer variable `ptr`.

Program to understand dynamic memory allocation of memory.

```
#include<stdio.h>
#include<alloc.h>
main()
{
int *ptr, i;
ptr=(int *)malloc(5*sizeof(int));
    if(ptr==NULL)
    {
        printf("memory not available\n");
        exit(1);
    }
printf("Enter 5 integers");
for(i=0;i<5;i++)
{
    scanf("%d",ptr+i);
    ptr=(int *) realloc(ptr,9*sizeof(int)); /* Allocate memory for 4 more integers */
}
}
```

```
if(ptr==NULL)
{
    printf("memory not available\n");
    exit(1);
}
printf("Enter 4 more integers");
for(i=5;i<9;i++)
{
    scanf("%d",ptr+i);
for(i=0;i<9;i++)
printf("%d",*(ptr+i));

}
```

free()

Declaration: `void free(void *p)`

The dynamically allocated memory is not automatically released

It will exist till the end of program.

If we have finished working with the memory allocated dynamically, it is our responsibility to release that memory so that it can be reused.

The function `free()` is used to release the memory space allocated dynamically.

The memory released by `free()` is made available to heap again and can be used for some other purposes.

For Ex: `free(ptr);`

When the program terminates all the memory is released automatically by the operating system, but it is a good practice to free whatever has been allocated dynamically.