```
    printf("Value of b = %f\n",*(float *)vp);
    return 0;
}
```

Output:
Value of a = 3
Value of a = 12
Value of b = 3.400000

```
/*P9.27 Pointer arithmetic in void pointers*/
#include<stdio.h>
int main(void)
{
        int i;
        float a[4]={1.2,2.5,3.6,4.6};
        void *vp;
        vp=a;
        for(i=0; i<4; i++)
        {
                printf("%.1f\t", *(float *)vp);
                (float *)vp=(float *)vp+1 ;      /*Can't write vp=vp+1*/
        }
        printf("\n");
        return 0;
}
```

Output:
```
1.2   2.5   3.6   4.6
```

`void` pointers are generally used to pass pointers to functions which have to perform same operations on different data types.

9.21 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 allocate or deallocate memory during the execution of program. In many applications it is not possible to predict how much memory would be needed by the program at run time. For example suppose we declare an array of integers of size 200.

```
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 runtime. Now two types of problems may occur. The first case is that the number of values to be stored is less than the size of array, so there will be wastage of memory. For example if we have to store only 50 values in the above array, then space for 150 values(600 bytes) is wasted. In second case our program fails if we want to store more values than the size of array, for example if there is need to store 205 values in the above array.

To overcome these problems we should be able to allocate and deallocate memory at run time. 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 using libraray functions that are declared `stdlib.h`. These functions allocate memory from a memory area called heap and deallocate this memory whenever not required, so that it can be used again for some other purpose.

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

9.21.1 malloc()

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

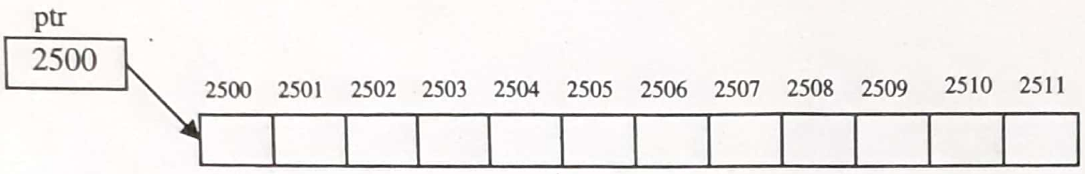
This function is used to allocate memory dynamically. The argument `size` specifies the number of bytes to be allocated. The type `size_t` is defined in `stdlib.h` and other headers as `unsigned int`. On success, `malloc()` returns a pointer to the first byte of allocated memory. The returned pointer 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 allocated. The expression `(datatype *)` is used to typecast the pointer returned by `malloc()`. For example-

```
int *ptr;
ptr=(int *)malloc(12);
```

This allocates 12 contiguous bytes of memory space and the address of first byte is stored in the pointer variable `ptr`.



This space can hold 3 integers. The allocated memory contains garbage value. We can use `sizeof` operator to make the program portable and more readable.

```
ptr=(int *)malloc(3*sizeof(int));
```

This allocates the memory space to hold three integer values.

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 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.

```
/*P9.28 Program to understand dynamic allocation of memory*/
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
        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");
                exit(1);
        }
        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));
        return 0;
}
```

The function `malloc()` returns a `void` pointer and we have studied that a `void` pointer can be assigned to any type of pointer without typecasting. But we have used typecasting because it is a good practice to do so and moreover it ensures compatibility with C++.

9.21.2 calloc()

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

The `calloc()` function is used to allocate multiple blocks of memory. It is similar to `malloc()` function except for two differences. The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. For example-

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

This allocates 5 blocks of memory, each block contains 4 bytes and the starting address is stored in the pointer variable `ptr`, which is of type `int`. An equivalent `malloc()` call would be-

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

The other difference between `calloc()` and `malloc()` is that the memory allocated by `malloc()` contains garbage value while the memory allocated by `calloc()` is initialized to zero. But this initialization by `calloc()` is not very reliable, so it is better to explicitly initialize the elements whenever there is need to do so.

Like `malloc()`, `calloc()` also returns NULL if there is not sufficient memory available in the heap.

9.21.3 realloc()

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

We may want to increase or decrease the memory allocated by `malloc()` or `calloc()`. 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. This function takes two arguments, first is a pointer to the block of memory that was previously allocated by `malloc()` or `calloc()` and second one is the new size for that block. For example-

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

This statement allocates the memory of the specified size and the starting address of this memory block is stored in the pointer variable `ptr`. If we want to change the size of this memory block, then we can 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`. The `newsize` may be smaller or larger than the old size. If the `newsize` is larger, then the old data is not lost and the newly allocated bytes are uninitialized. The starting address contained in `ptr` may change if there is not sufficient memory at the old address to store all the bytes consecutively. This function moves the contents of old block into the new block and the data of the old block is not lost. On failure, `realloc()` returns NULL and in this case the old memory is not deallocated and it remains unchanged.

If `ptr` is a null pointer, `realloc()` behaves like `malloc()` function. If `ptr` is not a pointer returned by `malloc()`, `calloc()` or `realloc()`, the behaviour is undefined.

```
/*P9.29 realloc() function*/
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
        int i,*ptr;
        ptr = (int *)malloc(5 *sizeof(int));
        if(ptr==NULL)
        {
                printf("Memory not available\n");
                exit(1);
        }
        for(i=0; i<5; i++)
                *(ptr+i)=i*2;

        ptr = (int *)realloc(ptr,9*sizeof(int));  /*Allocate memory for 4 more integers*/
        if(ptr==NULL)
        {
            printf("Memory not available\n");
            exit(1);
        }
        for(i=5; i<9; i++)
            *(ptr+i)=i*10;

        for(i=0; i<9; i++)
            printf("%d ",*(ptr+i));
        return 0;
}
```

Output:

```
0 2 4 6 8 50 60 70 80
```

9.21.4 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 the heap again and can be used for some other purpose.

```
free(ptr);
```

Here `ptr` is a pointer variable that contains the base address of a memory block created by `malloc()` or `calloc()`. Once a memory location is freed it should not be used. We should not try to free any memory location that was not allocated by `malloc()`, `calloc()` or `realloc()`.

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. We won't get any errors if we don't free the dynamically allocated memory, but this would lead to memory leak i.e. memory is slowly leaking away and can be reused only after the termination of program. For example consider this function-

```
void func()
{
        int *ptr;
        ptr = (int*)malloc(10*sizeof(int));
        ..................
}
```

Here we have allocated memory for 10 integers through `malloc()`, so each time this function is called, space for 10 integers would be reserved. We know that the local variables vanish when the function terminates, and since `ptr` is a local pointer variable, it will be finished automatically at the termination of function. But the space allocated dynamically is not deallocated automatically, so that space remains there and can't be used, leading to memory leaks. We should free the memory space by putting a call to `free()` at the end of the function.

Since the memory space allocated dynamically is not released after the termination of function, it is valid to return a pointer to dynamically allocated memory. For example-

```
int *func()
{
        int *ptr;
        ptr=(int*)malloc(10*sizeof(int));
        ..............
        return ptr;
}
```

Here we have allocated memory through `malloc()` in `func()`, and returned a pointer to this memory. Now the calling function receives the starting address of this memory, so it can use this memory. Note that now the call to function `free()` should be placed in the calling function when it has finished working with this memory. Here `func()` is declared as a function returning pointer. Recall that it is not valid to return address of a local variable since it vanishes after the termination of function.

9.22 Dynamic Arrays

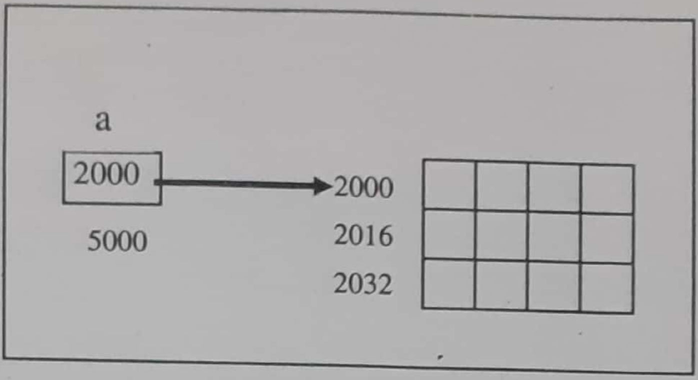
The memory allocated by `malloc()`, `calloc()` and `realloc()` is always made up of contiguous bytes. In C there is equivalence between pointer notation and subscript notation i.e. we can apply subscripts to a pointer variable. So we can access the dynamically allocated memory through subscript notation also. We can utilize these features to create dynamic arrays whose size can vary during run time. Now we'll rewrite the program P9.28 using subscript notation.

```
/*P9.30 Program to access dynamically allocated memory as a 1d array*/
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
        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");
                exit(1);
        }
        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]);
        return 0;
}
```

In this way we can simulate a 1-D array for which size is entered at execution time. Now we'll see how to create a dynamically allocated 2-D array. In the next program we have used a pointer to an array to dynamically allocate a 2-D array. Here the number of columns is fixed while the number of rows can be entered at run time.

```
/*P9.31 Program to dynamically allocate a 2-D array using pointer to an array*/
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
        int i,j,rows;
        int (*a)[4];
        printf("Enter number of rows : ");
        scanf("%d",&rows);
        a=(int(*)[4])malloc(rows*4*sizeof(int));
        for(i=0; i<rows; i++)
                for(j=0; j<4; j++)
                {
                        printf("Enter a[%d][%d] : ",i,j);
                        scanf("%d",&a[i][j]);
                }
        printf("The matrix is :\n");
        for(i=0; i<rows; i++)
        {
                for(j=0; j<4; j++)
                        printf("%5d",a[i][j]);
                printf("\n");
        }
        free(a);
        return 0;
}
```

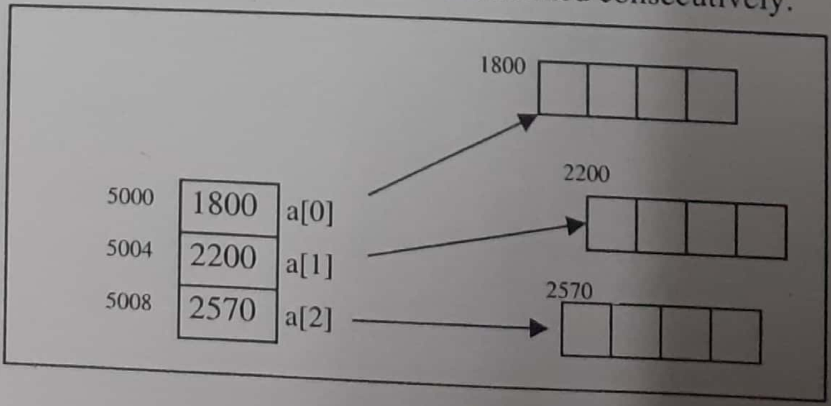
Suppose the number of rows entered is 3. The following figure shows how the dynamically allocated memory is accessed using pointer to an array. Since there are 3 rows and 4 columns, we'll allocate 48 bytes through `malloc()`, and the address returned by `malloc()` is assigned to a. The return value of `malloc()` is cast appropriately.



Now we'll allocate a 2-D array using array of pointers. Here the number of rows is fixed while the number of columns can be entered at run time.

```
/*P9.32 Program to dynamically allocate a 2-D array using array of pointers*/
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
        int *a[3],i,j,cols;
        printf("Enter number of columns : ");
        scanf("%d",&cols);
        /*Initialize each pointer in array by address of dynamically allocated memory*/
        for(i=0; i<3; i++)
                a[i]=(int *)malloc(cols*sizeof(int));
        for(i=0; i<3; i++)
                for(j=0; j<cols; j++)
                {
                        printf("Enter value for a[%d][%d] : ",i,j);
                        scanf("%d",&a[i][j]);
                }
        printf("The matrix is :\n");
        for(i=0; i<3; i++)
        {
                for(j=0; j<cols; j++)
                        printf("%5d",a[i][j]);
                printf("\n");
        }
        for(i=0; i<3; i++)
                free(a[i]);
        return 0;
}
```

Suppose the number of columns entered is 4. This figure shows how to dynamically allocate a 2-D array using array of pointers. In this case the rows are not allocated consecutively.



1800
2200
2570
5000
5004
5008
1800
2200
2570
a[0]
a[1]
a[2]