Multi-Dimensional Array

A multi-dimensional array is an array with more than one level or dimension. It might be 2-Dimensional and 3-Dimensional and so on.

Consider the example of storing marks of 5 students in some particular subject A. We can define,

```
int A_marks[ ] = {90,95,99,100,89};
```

Consider the example of storing marks of 5 students in 3 different subjects A, B and C. We can say,

Version-1:

```
int A_marks[ ] = {90,95,99,100,89};

int B_marks[ ] = {95,90, 91,99, 94};

 int C_marks[ ] = {91,92,93,95,100};
```

Version-2:

```
int student_1_marks[ ] = {90, 95, 91};

int student_2_marks[ ] = {95, 90, 92};

int student_3_marks[ ] = {99, 91, 93};

int student_4_marks[ ] = {100, 99, 95};

int student_5_marks[ ] = {89, 94, 100};
```

Now, rather than storing marks this way in One - Dimensional Array, we can use Two - Dimensional array as we have two dimensions involved in this particular example: student and subject marks

Two – Dimensional Array

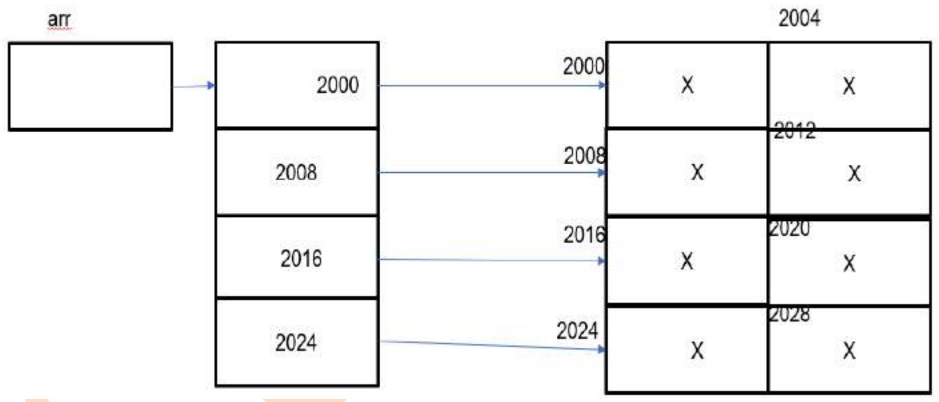
It is treated as an array of arrays. Every element of the array must be of same type as arrays are homogeneous. Hence, every element of the array must be an array itself in 2D array.

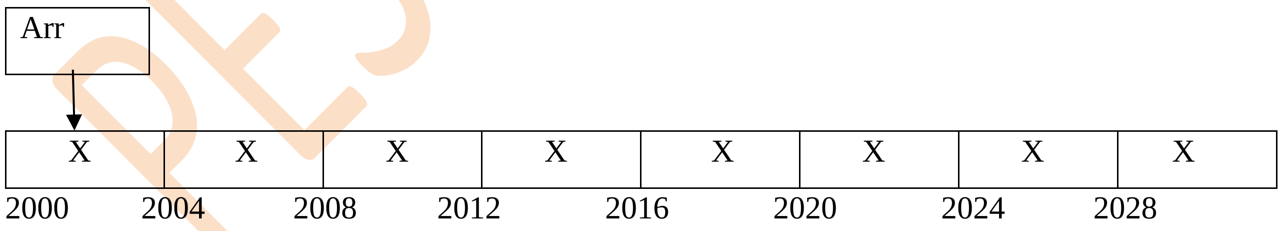
Let us consider the above example of storing 5 students marks in three different subjects. int marks[5][3] = {{90, 95, 91}, {95, 90, 92}, {99, 91, 93}, {100, 99, 95}, {89, 94, 100} };

Declaration of a 2D Array

data_type array_name[size_1][size_2];

int arr[4][2]; // Allocate 8 contiguous memory locations





If the size of the integer is 4 bytes ,8 contigeous memory allocated

Initialization of a 2D Array: Compiler knows the array size based the array elements and allocates memory

data_type array_name[size_1][size_2] = {elements separated by comma};

int arr[][] = {11,22,33,44,55,66}; // Error. Column size is compulsory

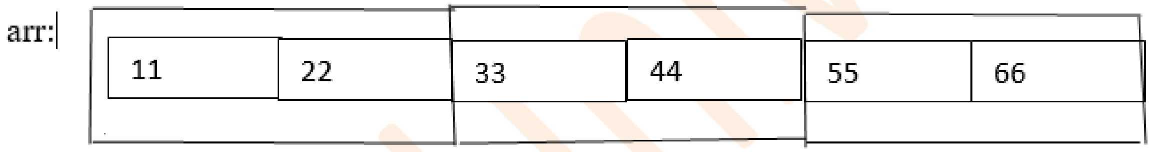
Without knowing the number of columns in each row – it is impossible to locate an element in 2D array.

int arr[3][2] = {{11,22},{33,44},{55,66}};

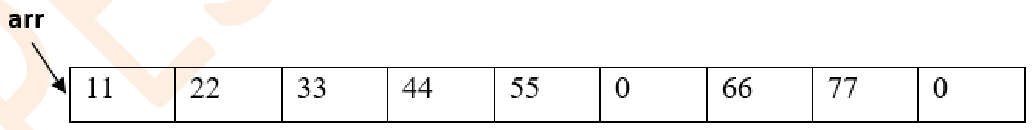
int arr[3][2] = {11,22,33,44,55,66};

// Allocate 6 contiguous memory locations and assign the values

Int arr[][2] = {11,22,33,44,55,66};



int arr[][3] = {{11,22,33},{44,55},{66,77}}; **//Partial initialization**

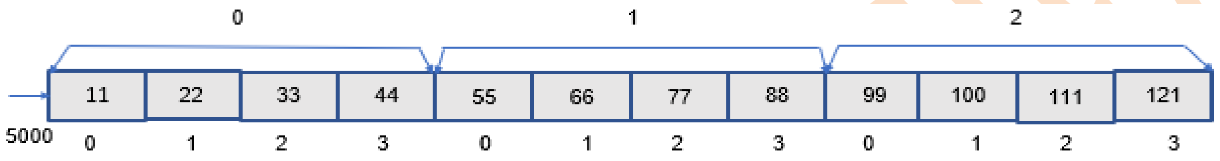


Internal Representation of a 2D Array

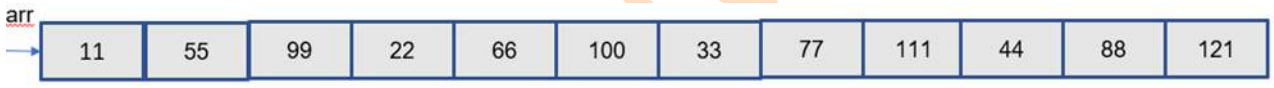
As two - dimensional array itself is an array, elements are stored in contiguous memory locations. And since elements are stored in contiguous memory locations, there are two possibilities.

int arr[3][4] = {{11, 22, 33, 44}, {55, 66, 77, 88}, {99, 100, 111, 121}};

Row major Ordering: All elements of one row are stored followed by all elements of the next row and so on.



Column Major Ordering: All elements of one column are stored followed by all elements of the next column and so on.



Generally, systems support Row Major Ordering.

<u>Address of an element in a 2D Array</u>

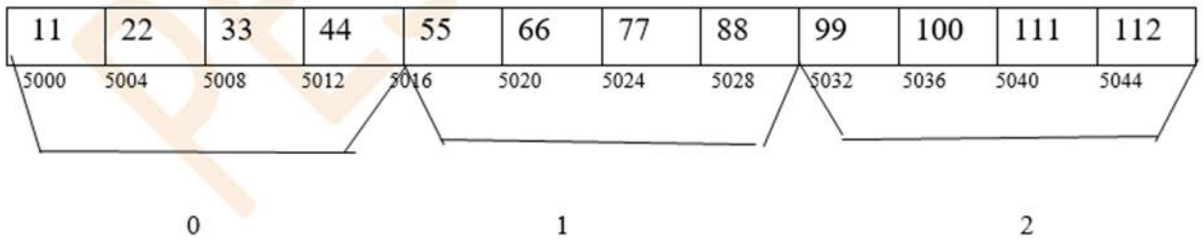
Address of A[i][j] = Base_Address +((i * No. of columns in every row) + j)*size of every element;

Consider, int arr[3][4] = {{11, 22, 33, 44}, {55, 66, 77, 88}, {99, 100, 111, 121}};

If the size of integer is 4 bytes in the system and the base address is 5000, then address of arr[1][2] can be found by 5000+((1*4)+2)*4 = 5000+6*4 = 5000+24 = 5024.

Consider, int arr[3][4] = {11, 22, 33, 44, 55, 66, 77, 88, 99, 100, 111, 121};

If the size of integer is 4 bytes in the system and the base address is 5000, then address of arr[1][5] can be found by 5000+((1*4)+5)*4 = 5000+9*4 = 5000+36 = 5036.



Code to Read and display 2D Array

```
int a[100][100];

int n; int m;

printf("enter row num and column numbers\n");

 scanf("%d %d",&n,&m);

printf("Enter %d elements",n*m);

for(int i = 0;i<n;i++)   // n – row index          // m – column index

{

        for(int j= 0;j<m;j++)

        {

                scanf("%d",&a[i][j]);

        }

}
```

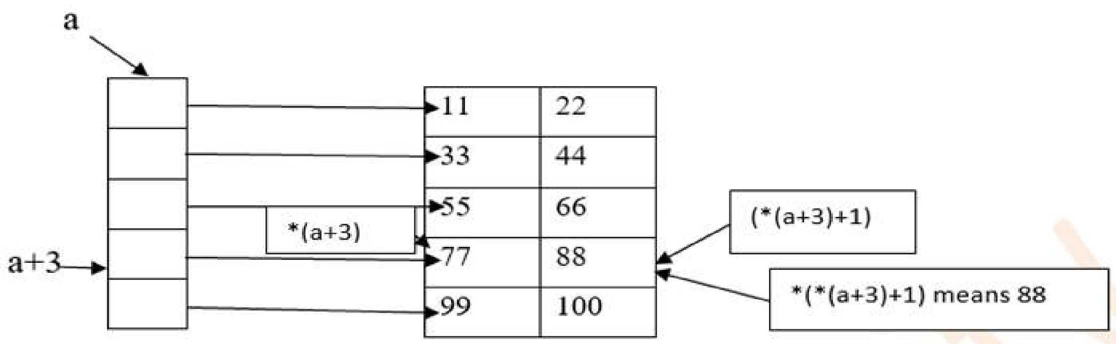
```
printf("entered elements are\n");

for(int i = 0;i<n;i++)

{

        for(int j= 0;j<m;j++)

        {

                printf("%d\t",a[i][j]);

        }

        printf("\n");

}
```

Two-Dimensional Array and Pointers

Consider int a[5][2] = {{11,22}, {33,44}, {55,66}, {77,88}, {99,100}};

```
        printf("Using array, accessing the elements of the array");

        printf("%d\n",a[3][2]);          // 99   array notation

        printf("%d\n",a[3][1]);          // 88    array notation

        printf("%d\n",*(*(a+3)+1)); // 88       // pointer notation
```

Array name is a pointer to a row. The expression a + 3 is a pointer to the third row. *(a + 3) becomes a pointer to the zeroth element of that row. Then +1 becomes a pointer to the nextelement in that row. To get the element in that location, dereference the pointer.



In General, (a + i) points to ith 1-D array

//int *p1 = a;// incompatible type. Warning during compilation

// If above statement is fine, what happens when p1[7] is accessed and p1[2][1] is accessed

int (*p)[2] = a;// **p is a pointer to an array of 2 integers**

Since subscript([ ]) have higher precedence than indirection(*), it is necessary to have parentheses for indirection operator and pointer name. Here the type of p is 'pointer to an array of 2 integers'.

```
printf("using pointer accessing the elements of the array")

printf("%d\n",p[3][1]);          // 88     array notation

printf("%d\n",*(*(p+3)+1)); // 88      pointer notation
```

Note : The pointer that points to the 0th element of array and the pointer that points to the whole array are totally different.

Consider the program to illustrate the size of pointer to an array of items.

```
int arr[4][3]={33,44,11,55,88,22,33,66,99,11,80,9};

//assigning array to a pointer

int *p=arr;                    //gives warning

//printf("%d",p[7]);  //looks fine,not a good idea.

//printf("\n%d\n",p[2][1]);            //error..Column size not available to p.

//create a pointer to an array of integers

int (*p)[3]=arr;

printf("%d\n",p[2][1]);

printf("%d\n",*(*(p+2)+1));

printf("%d\n",sizeof(p));      //size of pointer

printf("%d %d\n",sizeof(*p),sizeof(arr));        //size of the array pointing to 0th row and size of the entire 2D array

}
```

Passing 2D Array to a function

Let us try to read and display 2D array using functions. Client code is as below.

```
int main()

{

int a[100][100];

int m1,n1;

printf("enter the order of a\n");

scanf("%d%d",&m1,&n1); // user entered 3 5

printf("enter the elements of a\n");

read(a,m1,n1);

printf("elements of A are\n");

display(a,m1,n1);

}
```

Consider the following cases while passing 2D array to functions

Case1: Declaration and definition of the function with no column size

```
        void read(int[][],int m,int n);       //Compiletime error

        void display(int[][],int m,int n);
```

Case 2: Declaration and definition of the function with column size not same as what is given in the declaration of the array in the client code

```
void read(int[][3],int m,int n);     // warning but code works
void display(int[][3],int m,int n);
```

Case 3: Declaration and definition of the function with column size same as what is given in the declaration of the array in the client code

```
void read(int[][100],int m,int n);     // fine.
void display(int[][100],int m,int n);
```

```
// But can we say void read(int[][n],int m,int n ) ? Think !!!!
```

Case 4: If the order of parameters is changed as below in declaration and definition of the function, we need to change the client code. Changing the client code is not a good programmers habit. Because interface cannot be changed. Implementation can change. So refer to case 5.

```
void read(int m,int n,int a[][n]);
void display(int m,int n,int a[][n]);
```

Case 5: Forward declaration

```
void read(int n;int[][n],int m,int n);
void display(int n; int[][n],int m,int n);
```

```
void read(int n;int a[][n],int m,int n)
{
int i,j;
```

```
        for(i = 0;i<m;i++)
                for(j = 0;j < n;j++)
                {
                        printf("enter the element");
                        scanf("%d",&a[i][j]);
                }
}
void display(int n; int a[][n],int m,int n)
{
        int i,j;
        for(i = 0;i<m;i++)
        {
                for(j = 0;j < n;j++)
                {
                        printf("%d\t",a[i][j]);
                }
                printf("\n");
        }
}
```

Case 6: Array degenerates to a pointer at run time. So how about using pointer to an array as a parameter.

```
                    void read(int n; int(*)[n],int m,int n);

                    void display(int n; int(*)[n],int m,int n);

void read(int n; int (*a)[n],int m,int n)

{

          int i,j;

          for(i = 0;i<m;i++)

                    for(j = 0;j < n;j++)

                    {

                              printf("enter the element");

                              scanf("%d",&a[i][j]);

                    }

}

void display(int n; int (*a)[n],int m,int n)

{

          int i,j;

          for(i = 0;i<m;i++)

          {

                    for(j = 0;j < n;j++)
```

```
                {
                        printf("%d\t",a[i][j]);
                }
                printf("\n");
        }
}
```

Let us write a function to add, subtract and multiply two matrices. Display appropriate message when these two matrices are not compatible for these operations.

```
void read(int (*a1)[],int,int); void display( int (*a1)[],int,int);
void multiply(int n1; int n2; int (*a)[n1],int (*b)[n2],int m1,int n1,int n2,int (*c)[n2]);
 void subtract(int n; int (*a)[n],int (*b)[n],int m, int n, int (*c)[n]);
void add(int n; int (*a)[n],int (*b)[n],int m, int n, int (*c)[n]);
 int main()
{
        int a[100][100];
        int b[100][100];
        int c[100][100];
        int m1,n1;
 int m2,n2;

printf("enter the order of a\n");

scanf("%d%d",&m1,&n1);

 printf("enter the elements of a\n");

read(a,m1,n1);

printf("enter the order of b\n");

scanf("%d%d",&m2,&n2);

printf("enter the elements of b\n");

 read(b,m2,n2);

if (m1 == m2 && n1 == n2)

{

            printf("elements of A are\n");

            display(a,m1,n1);

            printf("elements of B are\n");

            display(b,m2,n2);

            printf("Addition of two matrices\n");

            add(a,b,m1,n1,c);

            display(c,m2,n2);

            printf("Subtraction of two matrices\n");

            subtract(a,b,m1,n1,c);
```

```
        display(c,m2,n2);
}
else
{
        printf("Two matrices are not compatible for additiona nd subtraction\n");
}

if (n1==m2)
{
        printf("Multiplication of two matrices\n");
        multiply(a,b,m1,n1,n2,c);
        display(c,m1,n2);
}
else
{
        printf("Two matrices are not compatible for multiplication too\n");
}
return 0;
}
```

```
 void add(int n; int (*a)[n],int (*b)[n],int m, int n, int (*c)[n])

{

        for(int i = 0;i<m;i++)

        {

                for(int j= 0;j<n;j++)

                {

                        c[i][j] = a[i][j]+b[i][j];

                }

        }

}

void subtract(int n; int (*a)[n],int (*b)[n],int m, int n, int (*c)[n])

{

        for(int i = 0;i<m;i++)

        {

                for(int j= 0;j<n;j++)

                        {

                                c[i][j] = a[i][j] - b[i][j];

                        }

        }
```

```
}

void multiply(int n1; int n2; int (*a)[n1],int (*b)[n2],int m1,int n1,int n2,int (*c)[n2])

{


        for(int i = 0;i<m1;i++)

        {

                for(int j= 0;j<n2;j++)

                {

                        c[i][j] = 0;

                }

}

for(int i=0;i<m1;i++)

        {

                for(int j=0;j<n2;j++)

                {

                int sum=0;

                for(int k=0;k<n1;k++)

                {

                        sum=sum+a[i][k]*b[k][j];

                        }
```

```
                    c[i][j]=sum;
                }
            }
        }
```

Let us write a program to take n names from the user and print it. Each name can have maximum of 20 characters.

```
#include<stdio.h>
 int main()
 {
        // Think about the commented code? Is it correct?
        /*
        char name1[20]; //Can store one name with maximum of 20 characters including '\0'
        // So need n different variables to store n names
        // But can we store n names under the same variable name???? --- Yes. Use 2D array
        */

        char names[200][20]; int n;
        printf("How many names you want to store?\n");
        scanf("%d", &n);
        int i;
        printf("enter %d names\n", n); for(i=0;i<n;i++)

        {

                scanf("%[^\n]s",names[i]);

        }

        printf("U entered below names\n"); for(int i=0;i<n;i++)

        {

                printf("%s\n",names[i]);

        }

        return 0;

}
```

Structures

Structure is a user-defined data type in C language which allows us to combine data of different types together. It helps to construct a complex data type which is more meaningful. It is often convenient to have a single name with which to refer to a **collection of related items of different types**. Structures provide a way of storing many different values in variables of potentially different types under the same name. Structures are generally useful whenever a lot of data needs to be grouped together. For instance, they can be used to hold records from a database or to store information about contacts in an address book.

Why Structures?

Need of Structure is discussed through below mentioned programs.

Let us try to interactively store the Roll_number, Total_marks and Name of 20 students.

```
char names[20][15];
int marks[20];
int roll_num[20];
for(int i=0;i<20;i++)
{       printf("Enter Roll_number, marks and name\n");
        scanf("%d",&roll_num[i]);
        scanf("%d",&marks[i]);
        scanf("%[^\n]s",names[i]);
}
```

Display the entries and check whether this code works.

By using separate arrays for each of the details of students, nowhere we saying all these three are related. So, we need Structures.

Introduction to Structures

A structure creates a data type that can be used to group items of possibly different types into a single type. Grouping of attributes into a single entity is known as Structure. Creating a new type decides the binary layout of the type. Order of fields and the total size of a variable of that type is decided when the new type is created.

Characteristics/properties of structures:

- A user defined data type/non-primary/secondary
- Contains one or more components – Generally known as data members. These are named ones.
- Collection of homogeneous or heterogeneous data members
- Size of a structure depends on implementation. Memory allocation would be at least equal to the sum of the sizes of all the data members in a structure. Offset is decided at compile time.
- Compatible structures may be assigned to each other.

To define a new type/entity, use the **keyword - struct**

The format for declaring a structure is as below:

```
struct Tag
{
        data_type member1;
        data_type member2;
        …..
        data_typememeber;
};                              // Don't forget semicolon here
```

where Tag is the name of the entire structure and Members are the elements within the struct.

Example: User defined type Student entity is created.

```
struct Student
{       int roll_no;
        char name[20];
        int marks;          };      //No memory allocation for declaration/description of the structure.
```

Accessing members of a structure

We can access the members of a structure only when we create instance variables of that type. If struct Student is the type, we can create the instance variable as:

struct student s1;   // s1 is the instance variable of type struct Student. Now memory gets allocated.

struct student* s2;   // s2 is the instance variable of type struct student*. Now memory gets allocated. s2 is pointer to structure

Operators used for accessing the members of a structure.

1. Dot operator (.)
2. Arrow operator (->)

Any member of a structure can be accessed using the structure variable as:

structure_variable_name.member_name

Suppose, s1 is the structure variable name and we want to access roll_no member of s1. Then, it can be accessed as:

s1.roll_no

Any member of a structure can be accessed using the pointer to a structure as:
pointer_variable->member_name

Suppose, s2 is the pointer to structure variable and we want to access roll_no member of s2. Then, it can be accessed as:

s2->roll_no

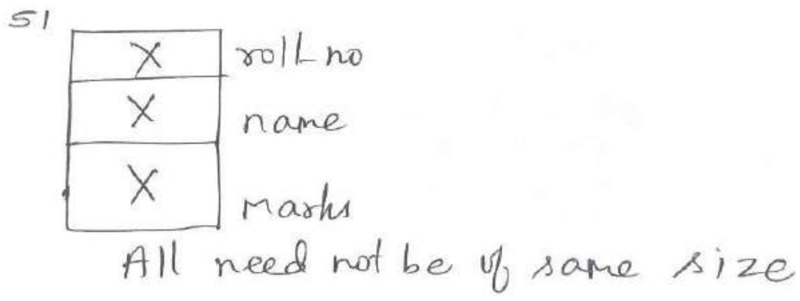
Declaration and Initialization

```
struct student                  // template declaration which describes how student entity looks like
{
        int roll_no;                    // these three are members of the structure
        char name[20];
        int marks;
};
```

Given the above user defined type, look at below client codes

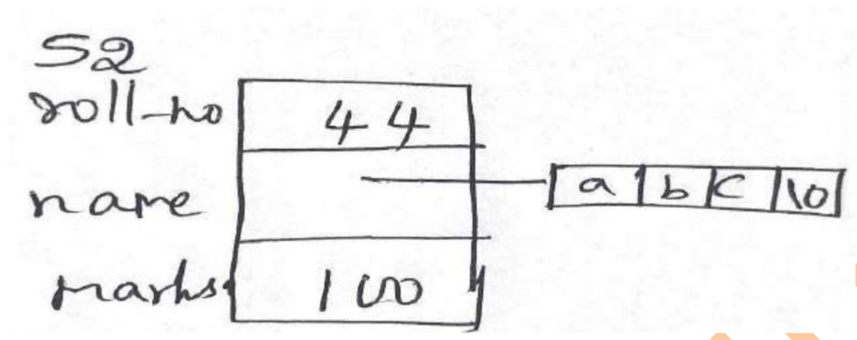
Version 1:

```
        struct student s1;                      // Declaration

        printf("%s\n",s1.name);
        printf("%d\n",s1.roll_no);
        printf("%d\n",s1.marks);

        // some undefined value gets printed.
```



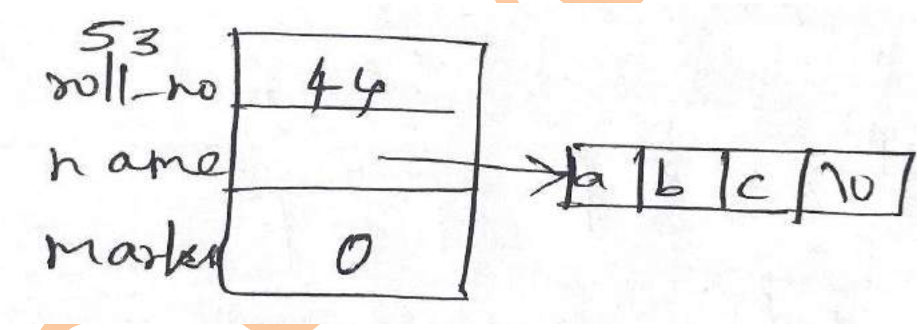
Version 2:

```
struct student s2 = {44, "abc", 100};                    // Initialization
printf("%d %d %s", s2.marks, s2.roll_no, s2.name);          // 100 44 abc
```



Version 3:

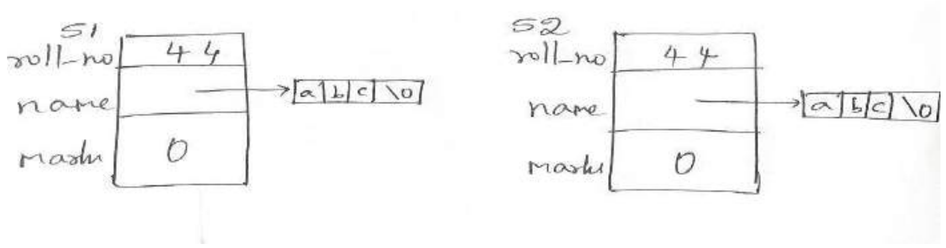
```
struct student s3 = {44, "abc"};         // Partial Initialization
// Explicitly few members are initialized. others are initialized to default value
printf("%d %d %s", s3.marks, s3.roll_no, s3.name);          // 0 44  abc
```



Version 4: Designated Initializers in C

Specify the name of a field to initialize with '.member_name =' or 'member_name:' before the element value. others are initialized to default value.

```
struct student s1 = {.name = "abc", .roll_no = 44};
struct student s2 = {name : "abc", roll_no : 44};
printf("%d %d %s\n", s1.roll_no, s1.marks, s1.name);        // 44 0 abc
printf("%d %d %s", s2.roll_no, s2.marks, s2.name);          // 44 0 abc
```



Memory Allocation for Structure Variables

Every data type in C has alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

More details, Refer to below links.

https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/

https://www.geeksforgeeks.org/data-structure-alignment/

What is the minimum number of bytes allocated for the structure variable in each of these cases? – Sum of the sizes of all the data members. Size of data members is implementation specific.

What is the maximum number of bytes allocated for the structure variable in each of these cases? **Implementation Specific.**

Below programs are run on Windows7 machine and 32 bit GCC compiler

```
struct test
{       int i;
        char j;
};
struct test1
{       char j;
        int i;
};
struct test2
{      char  k;
        char


    j; int i;
};
struct test3
{       int i;
        char k;
        int j;
};
int main()
{
        printf("size of the structure is %lu\n",sizeof(struct test));    //8bytes         4+4
        struct test t;
        printf("size of the structure is %lu\n",sizeof(t));              //    8bytes     4+4
        printf("size of the structure is %lu\n",sizeof(struct test1));  //8bytes        4+4
        printf("size of the structure is %lu\n",sizeof(struct test2));  //8bytes        4+4
        printf("size of the structure is %lu\n",sizeof(struct test3));  //12bytes       4+4+4
        return 0;
}
```

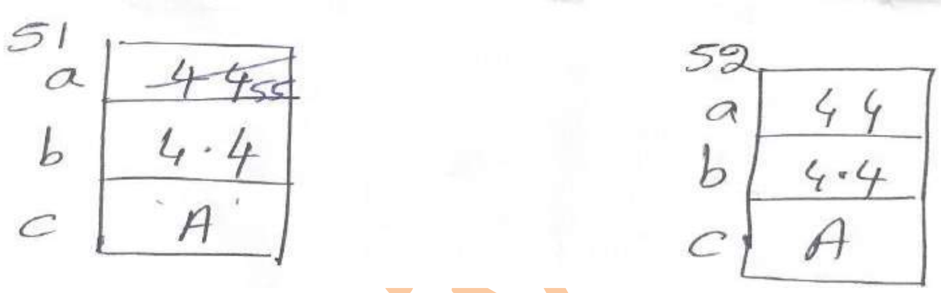
Comparison of Structures

```
        struct testing
        {       int a;   float b;        char c;
        };
        int main()
        {       struct testing s1 = {44, 4.4, 'A'};
                struct testing s2 = {44, 4.4, ‘A’};
                //printf(“%d”, s1== s2); // Compiletime Error.
                // == operator cannot be applied on structures for comparing
                // Think about s1- s2
                // Write a function to check for equality of every member of the structure
                printf(“%d”,is_equal(s1,s2));          // 1      // all fields are equal
                struct testing s3 = {33, 3.3, ‘A’};
                printf(“%d”,is_equal(s1,s3));          // 0      //first condition itself fails
                return 0;
        }
        int is_equal(struct testing s1, struct testing s2)
        {           return (s1.a == s2.a && s1.b == s2.b && s1.c == s2.c);   }
```

Member - wise copy

Structures of same type are assignment compatible. When you assign one structure variable to another structure variable of same type, member- wise copy happens. Both of them do not point to the same memory location. The compiler generates the code for member – wise copy.

```
struct testing
{       int a;
        float b;
        char c;
};
int main()
{       struct testing s1 = {44,4.4, 'A'};
        struct testing s2 = s1;
        s1.a = 55;
        printf("%d\n",s1.a);
        printf("%d\n",s2.a);
        return 0;
}
```



Few points to think:

- Structures can be assigned even if the structure contains an array. Is it True?
- If the structure contains pointer, how member-wise copy happens?

Parameter passing and return

Consider the Player structure. We will write functions to read and display the details of a player.

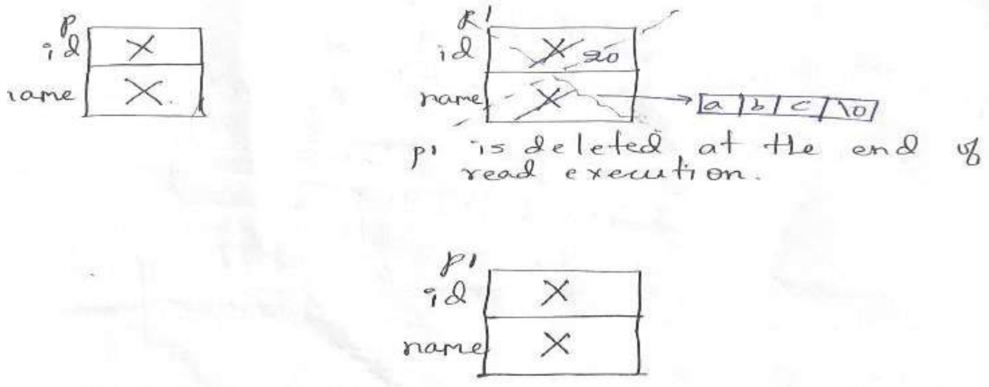
```
struct player
{       int id;
        char name[20];
};
```

Version 1:

```
int main()
{
        struct player p;
        printf("Enter id and name\n");
        read(p);        disp(p);
        return 0;
}
```

Parameter passing is always by value in C. So, copy of the argument is passed to the parameter p1. Whatever is modified, it is made to parameter p1.

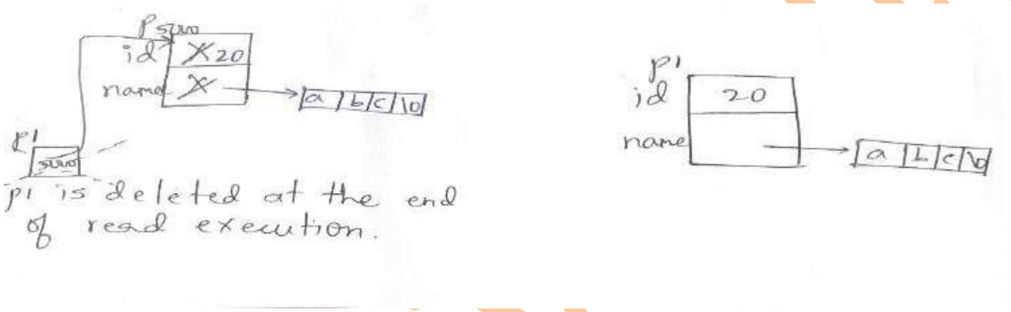
```
void read(struct player p1)
{
        scanf("%d ", &p1.id);           // user enters 20
        scanf("%[^\n]s", p1.name);
}
void disp(struct player p1)
{
        printf("%d\n",p1.id);                   // undefined values get printed
        printf("%s",p1.name);
}
```



Version 2: If you want to change the structure, pass a pointer to a structure(l-value) as an argument.

```
int main()
{
        struct player p;
        printf("Enter id and name\n");
        read(&p);      disp(p);                 // passing &p to read()
        / / Think, can we also pass &p to disp(). Is there any harm?
        return 0;
}
```

```
void read(struct player *p1)
{
        scanf("%d ", &(p1->id));              // or &((*p1).id)       // user enters 20
        scanf("%[^\n]s", p1->name);                              // abc entered
}
void disp(struct player p1)
{
        printf("%d ",p1.id);                  // actual values will be printed.
        printf("%s",p1.name);
}
```

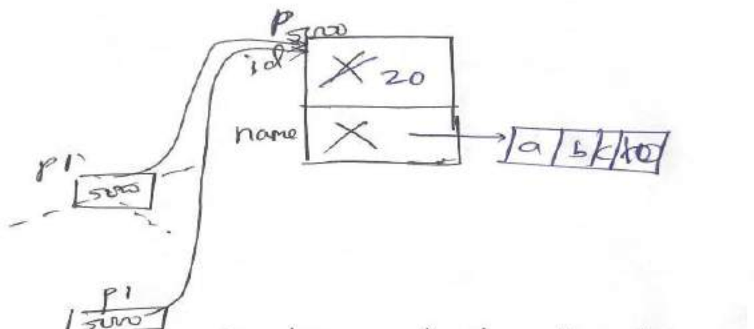


Version 3:

Assume the player structure contains many data members. In this case, can we just use structure variable to disp function? If we do so, this is considered as a bad practice. As every member of argument is copied to every member of parameter, it requires more space and more time to copy.

Also, when we are very sure that **we do not want to make any changes to the argument, parameter can be a pointer to constant structure**

```
void disp(const struct player* p1)
{       // p1.id = 34;          // throws error as p1 is a pointer to constant structure
        printf("%d ",p1->id);                 // actual values will be printed.
        printf("%s",p1->name);
}
```



p1 is a pointer to constant structure.

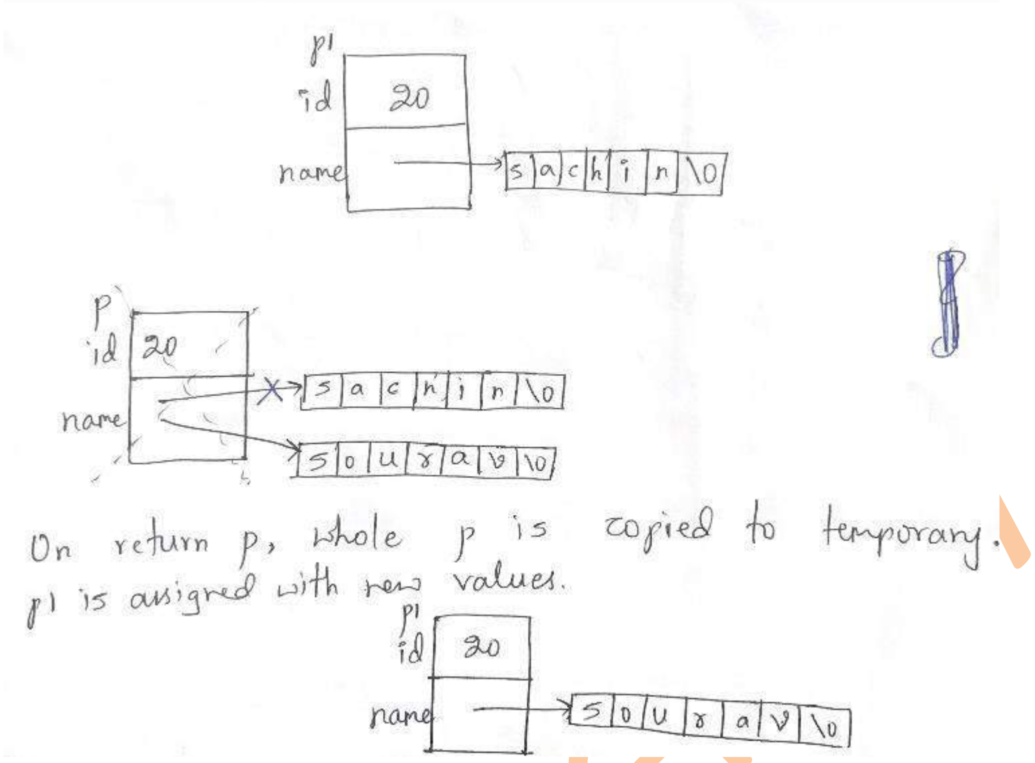
Can we return a structure variable? Yes.

Consider the below example code.

```
        struct player
        {       int id; char name[20]; };
        struct player modify(struct player);

        int main()
        {
                struct player p1={20,"sachin"};
                printf("before change %s",p1.name);
                p1=modify(p1);
                printf("after change %s",p1.name);
                return 0;
        }
```

When the function modify() returns a changed parameter by value, it is copied to a temporary. Then in the client code, the temporary is assigned to p1.

```
struct player modify(struct player p)
{
        strcpy(p.name,"Sourav");
        return p;
}
```



On return p, whole p is copied to temporary.
p1 is assigned with new values.

Think about these points:

- Can we make const struct player p as the parameter for read?
- Can we say struct player *p as the parameter without changing the client code?
- If we change the client code, can we use struct player *p as the parameter?
- What changes must be done in modify() to copy Sourav to name
- Can we return pointer to structure?

Usage of typedef

typedef is a keyword which is used to give a type, a new name. Used to assign alternative names to existing data types. It is used to provide an interface for the client. Once a new name is created using typedef, the client may use this without knowing the underlying type.

I want to store the age of a person. Best way to define a variable is int age = 80; Alternatively, you can also say, typedef int age; //another name for int is age.
Then age a = 80;
Can we say age age = 80?   // Think about this.

If we know how to declare a variable, then prefixing the declaration with typedef makes that a new name.

```
int b[10];                    // b is an array variable to store 10 integers
typedef int c[10];            // c is a name for an array of 10 integers
```

typedef is mostly used with user defined data types when names of the data types become slightly longer and complicated to use in programs.

Version 1:

```
struct player
{       int id;
        char name[20];
};
typedef struct player player_t;         // player_t is a new name to struct player.
int main()
{  player_t  p1;                        // p1 is a variable
}
```

Version 2: You can include typedef when defining a new type itself.

```
typedef struct player
{       int id;
```

```
        char name[20];

}player_t;              // player_t is a new name to struct player.
int main()
{  player_t  p1;                                  // p1 is a variable

}
```

Nested Structures

Structure written inside another structure is called as nesting of two structures. It can bed done in two ways.

1. **Separate structures:** Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct dob
{
      int date;
      int month;
      int year;
};

struct student            // template declaration which describes how student entity looks like
{
      int roll_no;         // these three are members of the structure
      char name[20];
      int marks;
      struct dob d1;                  //using structure variable of dob inside student
 };
```

It can be **understood from example that dob (date of birth) is the structure with members as date, month and year. Dob is then** used as a member in Student structure.

2. **Embedded structure:** It enables us to declare the structure inside the structure.

```
struct student            // template declaration which describes how student entity looks like
{
    int roll_no;          // these three are members of the structure
    char name[20];
    int marks;
    struct dob            //declaring structure dob inside student
    {
        int date;
        int month;
        int year;
    }d1;
};
```

Accessing Nested Structure Members: We can access the member of the nested structure by using **Outer_Structure.Nested_Structure.member** as given below:

```
int main()
{
    struct student s1 = {24, "John", 78, {20, 03, 2000}};   //Declaration and Initialization
    printf("\n Roll no. = %d, Name= %s, Marks = %d, dob = %d.%d.%d ", s1.roll_no,
    s1.name,s1.marks, s1.d1.date,s1.d1.month, s1.d1.year);
    return 0;
}
```

Structures

Structure is a user-defined data type in C language which allows us to combine data of different types together. It helps to construct a complex data type which is more meaningful. It is often convenient to have a single name with which to refer to a **collection of related items of different types**. Structures provide a way of storing many different values in variables of potentially different types under the same name. Structures are generally useful whenever a lot of data needs to be grouped together. For instance, they can be used to hold records from a database or to store information about contacts in an address book.

Why Structures?

Need of Structure is discussed through below mentioned programs.

Let us try to interactively store the Roll_number, Total_marks and Name of 20 students.

```
char names[20][15];
int marks[20];
int roll_num[20];
for(int i=0;i<20;i++)
{       printf("Enter Roll_number, marks and name\n");
        scanf("%d",&roll_num[i]);
        scanf("%d",&marks[i]);
        scanf("%[^\n]s",names[i]);
}
```

Display the entries and check whether this code works.

By using separate arrays for each of the details of students, nowhere we saying all these three are related. So, we need Structures.

Introduction to Structures

A structure creates a data type that can be used to group items of possibly different types into a single type. Grouping of attributes into a single entity is known as Structure. Creating a new type decides the binary layout of the type. Order of fields and the total size of a variable of that type is decided when the new type is created.

Characteristics/properties of structures:

- A user defined data type/non-primary/secondary
- Contains one or more components – Generally known as data members. These are named ones.
- Collection of homogeneous or heterogeneous data members
- Size of a structure depends on implementation. Memory allocation would be at least equal to the sum of the sizes of all the data members in a structure. Offset is decided at compile time.
- Compatible structures may be assigned to each other.

To define a new type/entity, use the **keyword - struct**

The format for declaring a structure is as below:

```
struct Tag
{
        data_type member1;
        data_type member2;
        …..
        data_typemember;
};                         // Don't forget semicolon here
```

where Tag is the name of the entire structure and Members are the elements within the struct.

Example: User defined type Student entity is created.

```
struct Student
{       int roll_no;
        char name[20];
        int marks;        };    //No memory allocation for declaration/description of the structure.
```

Accessing members of a structure

We can access the members of a structure only when we create instance variables of that type. If struct Student is the type, we can create the instance variable as:

struct student s1;   // s1 is the instance variable of type struct Student. Now memory gets allocated.

struct student* s2;   // s2 is the instance variable of type struct student*. Now memory gets allocated. s2 is pointer to structure

Operators used for accessing the members of a structure.

1. Dot operator (.)
2. Arrow operator (->)

Any member of a structure can be accessed using the structure variable as:

structure_variable_name.member_name

Suppose, s1 is the structure variable name and we want to access roll_no member of s1. Then, it can be accessed as:

s1.roll_no

Any member of a structure can be accessed using the pointer to a structure as:
pointer_variable->member_name

Suppose, s2 is the pointer to structure variable and we want to access roll_no member of s2. Then, it can be accessed as:

s2->roll_no

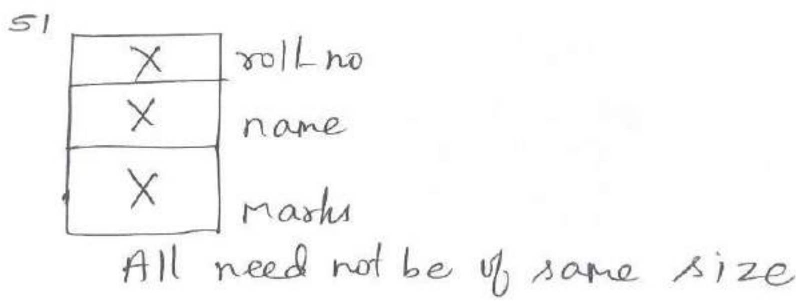
Declaration and Initialization

```
struct student                    // template declaration which describes how student entity looks like
{
        int roll_no;                        // these three are members of the structure
        char name[20];
        int marks;
};
```

Given the above user defined type, look at below client codes

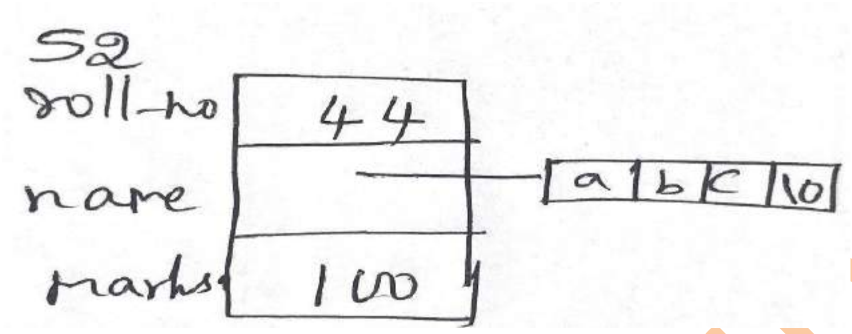
Version 1:

```
        struct student s1;                        // Declaration

        printf("%s\n",s1.name);
        printf("%d\n",s1.roll_no);
        printf("%d\n",s1.marks);

        // some undefined value gets printed.
```



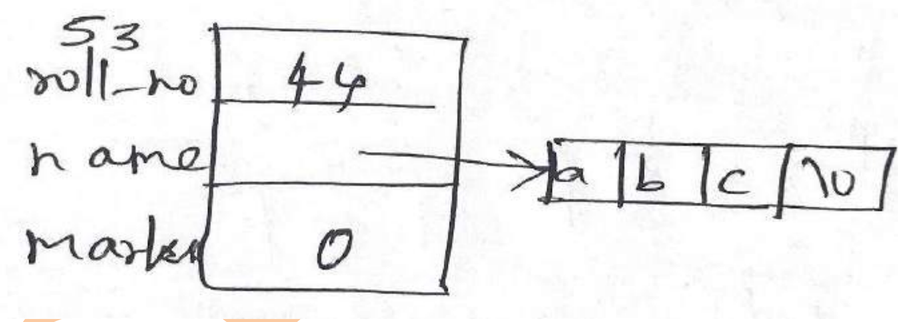
Version 2:

```
struct student s2 = {44, "abc", 100};                    // Initialization
printf("%d %d %s", s2.marks, s2.roll_no, s2.name);          // 100 44 abc
```



Version 3:

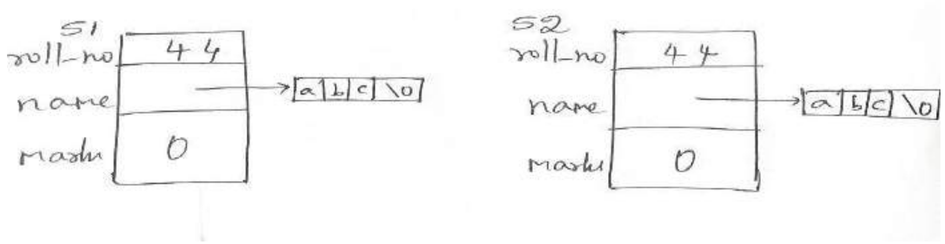
```
struct student s3 = {44, "abc"};          // Partial Initialization
// Explicitly few members are initialized. others are initialized to default value
printf("%d %d %s", s3.marks, s3.roll_no, s3.name);          // 0 44  abc
```



Version 4: Designated Initializers in C

Specify the name of a field to initialize with '.member_name =' or 'member_name:' before the element value. others are initialized to default value.

```
struct student s1 = {.name = "abc", .roll_no = 44};
struct student s2 = {name : "abc", roll_no : 44};
printf("%d %d %s\n", s1.roll_no, s1.marks, s1.name);        // 44 0 abc
printf("%d %d %s", s2.roll_no, s2.marks, s2.name);          // 44 0 abc
```



Memory Allocation for Structure Variables

Every data type in C has alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

More details, Refer to below links.

https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/

https://www.geeksforgeeks.org/data-structure-alignment/

What is the minimum number of bytes allocated for the structure variable in each of these cases? – Sum of the sizes of all the data members. Size of data members is implementation specific.

What is the maximum number of bytes allocated for the structure variable in each of these cases? **Implementation Specific.**

Below programs are run on Windows7 machine and 32 bit GCC compiler

```
struct test
{       int i;
        char j;
};
struct test1
{       char j;
        int i;
};
struct test2
{      char   k;
        char


    j; int i;
};
struct test3
{       int i;
        char k;
        int j;
};
int main()
{
        printf("size of the structure is %lu\n",sizeof(struct test));    //8bytes        4+4
        struct test t;
        printf("size of the structure is %lu\n",sizeof(t));             //     8bytes     4+4
        printf("size of the structure is %lu\n",sizeof(struct test1));  //8bytes         4+4
        printf("size of the structure is %lu\n",sizeof(struct test2));  //8bytes         4+4
        printf("size of the structure is %lu\n",sizeof(struct test3));  //12bytes        4+4+4
        return 0;
}
```

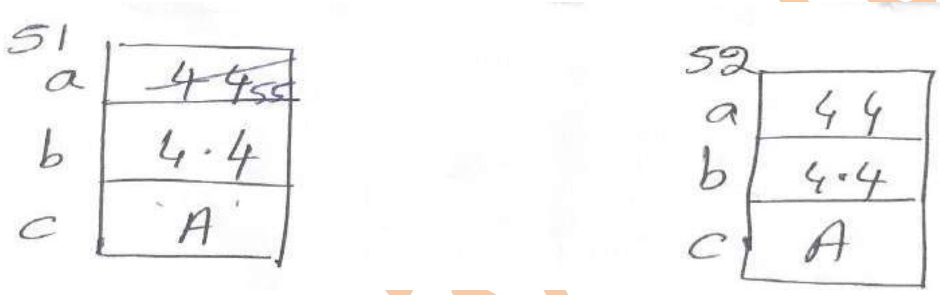
Comparison of Structures

```
         struct testing
        {       int a;   float b;         char c;
        };
        int main()
        {       struct testing s1 = {44, 4.4, 'A'};
                struct testing s2 = {44, 4.4, ‘A’};
                //printf(“%d”, s1== s2); // Compiletime Error.
                // == operator cannot be applied on structures for comparing
                // Think about s1- s2
                // Write a function to check for equality of every member of the structure
                printf(“%d”,is_equal(s1,s2));          // 1      // all fields are equal
                struct testing s3 = {33, 3.3, ‘A’};
                printf(“%d”,is_equal(s1,s3));          // 0      //first condition itself fails
                return 0;
        }
        int is_equal(struct testing s1, struct testing s2)
        {           return (s1.a == s2.a && s1.b == s2.b && s1.c == s2.c);   }
```

Member - wise copy

Structures of same type are assignment compatible. When you assign one structure variable to another structure variable of same type, member- wise copy happens. Both of them do not point to the same memory location. The compiler generates the code for member – wise copy.

```
struct testing
{       int a;
        float b;
        char c;
};
int main()
{       struct testing s1 = {44,4.4, 'A'};
        struct testing s2 = s1;
        s1.a = 55;
        printf("%d\n",s1.a);
        printf("%d\n",s2.a);
        return 0;
}
```



Few points to think:

- Structures can be assigned even if the structure contains an array. Is it True?
- If the structure contains pointer, how member-wise copy happens?

Parameter passing and return

Consider the Player structure. We will write functions to read and display the details of a player.

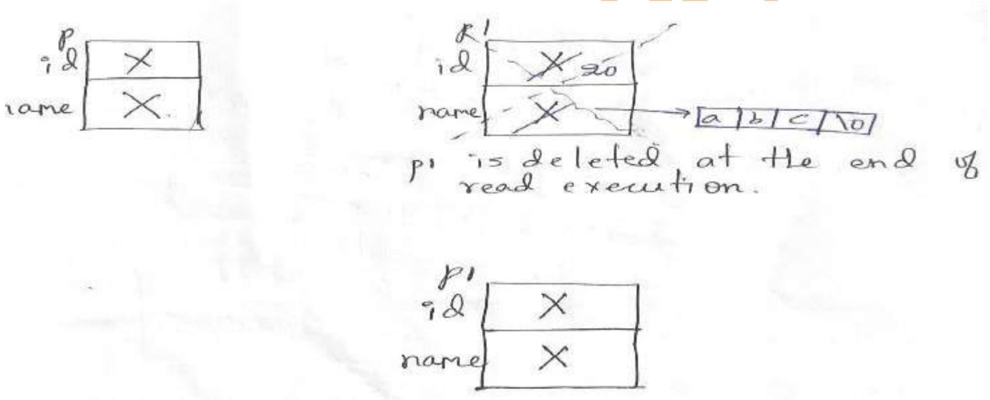
```
struct player
{       int id;
        char name[20];
};
```

Version 1:

```
int main()
{
        struct player p;
        printf("Enter id and name\n");
        read(p);        disp(p);
        return 0;
}
```

Parameter passing is always by value in C. So, copy of the argument is passed to the parameter p1. Whatever is modified, it is made to parameter p1.

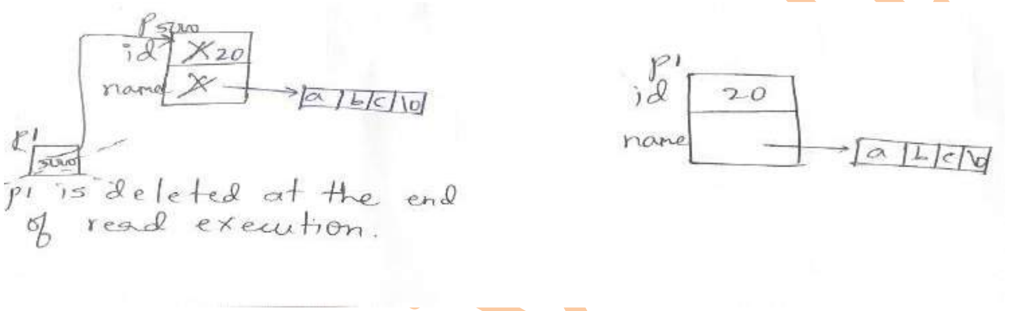
```
void read(struct player p1)
{
        scanf("%d ", &p1.id);          // user enters 20
        scanf("%[^\n]s", p1.name);
}
void disp(struct player p1)
{
        printf("%d\n",p1.id);                  // undefined values get printed
        printf("%s",p1.name);
}
```



Version 2: If you want to change the structure, pass a pointer to a structure(l-value) as an argument.

```
int main()
{
        struct player p;
        printf("Enter id and name\n");
        read(&p);      disp(p);                // passing &p to read()
        / / Think, can we also pass &p to disp(). Is there any harm?
        return 0;
}
```

```
void read(struct player *p1)
{
        scanf("%d ", &(p1->id));            // or &((*p1).id)       // user enters 20
        scanf("%[^\n]s", p1->name);                        // abc entered
}
void disp(struct player p1)
{
        printf("%d ",p1.id);                // actual values will be printed.
        printf("%s",p1.name);
}
```

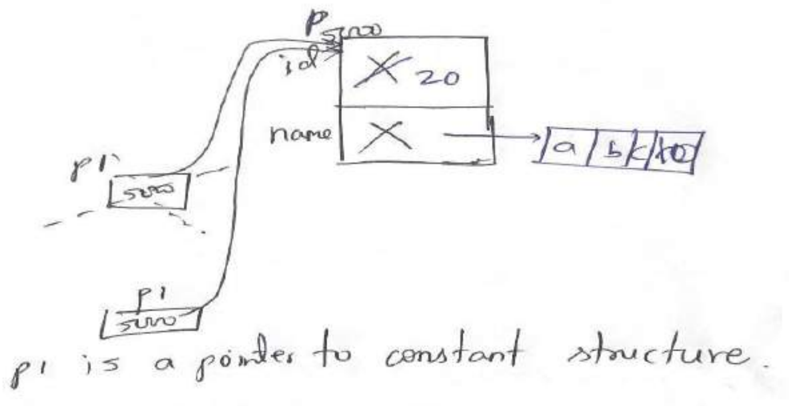


Version 3:

Assume the player structure contains many data members. In this case, can we just use structure variable to disp function? If we do so, this is considered as a bad practice. As every member of argument is copied to every member of parameter, it requires more space and more time to copy.

Also, when we are very sure that **we do not want to make any changes to the argument, parameter can be a pointer to constant structure**

```
void disp(const struct player* p1)
{       // p1.id = 34;          // throws error as p1 is a pointer to constant structure
        printf("%d ",p1->id);               // actual values will be printed.
        printf("%s",p1->name);
}
```



p1 is a pointer to constant structure.

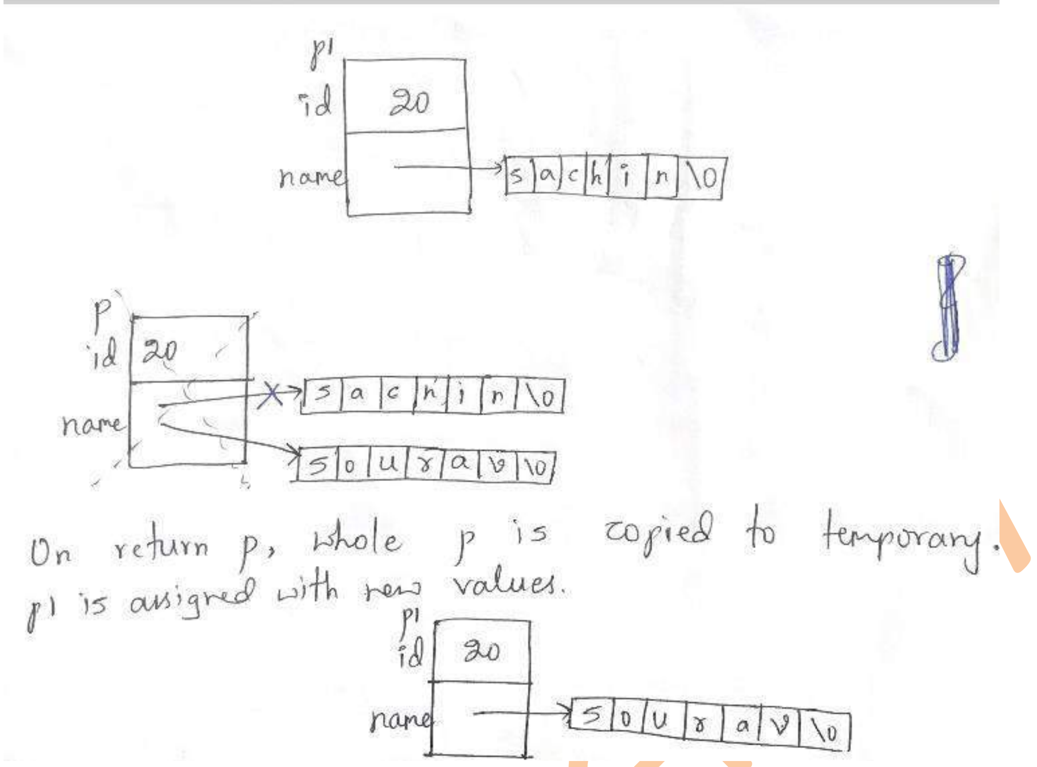
Can we return a structure variable? Yes.

Consider the below example code.

```
        struct player
        {       int id; char name[20]; };
        struct player modify(struct player);

        int main()
        {
                struct player p1={20,"sachin"};
                printf("before change %s",p1.name);
                p1=modify(p1);
                printf("after change %s",p1.name);
                return 0;
        }
```

When the function modify() returns a changed parameter by value, it is copied to a temporary. Then in the client code, the temporary is assigned to p1.

```
struct player modify(struct player p)
{
        strcpy(p.name,"Sourav");
        return p;
}
```



On return p, whole p is copied to temporary.
p1 is assigned with new values.

Think about these points:

- Can we make const struct player p as the parameter for read?
- Can we say struct player *p as the parameter without changing the client code?
- If we change the client code, can we use struct player *p as the parameter?
- What changes must be done in modify() to copy Sourav to name
- Can we return pointer to structure?

Usage of typedef

typedef is a keyword which is used to give a type, a new name. Used to assign alternative names to existing data types. It is used to provide an interface for the client. Once a new name is created using typedef, the client may use this without knowing the underlying type.

I want to store the age of a person. Best way to define a variable is int age = 80; Alternatively, you can also say, typedef int age; //another name for int is age.
Then age a = 80;
Can we say age age = 80?   // Think about this.

If we know how to declare a variable, then prefixing the declaration with typedef makes that a new name.

```
int b[10];                // b is an array variable to store 10 integers
typedef int c[10];        // c is a name for an array of 10 integers
```

typedef is mostly used with user defined data types when names of the data types become slightly longer and complicated to use in programs.

Version 1:

```
struct player
{       int id;
        char name[20];
};
typedef struct player player_t;          // player_t is a new name to struct player.
int main()
{  player_t  p1;                         // p1 is a variable
}
```

Version 2: You can include typedef when defining a new type itself.

```
typedef struct player
{       int id;
```

```
        char name[20];
}player_t;              // player_t is a new name to struct player.
int main()
{  player_t  p1;                                    // p1 is a variable
}
```

Nested Structures

Structure written inside another structure is called as nesting of two structures. It can bed done in two ways.

1. **Separate structures:** Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct dob
{
     int date;
     int month;
     int year;
};

struct student          // template declaration which describes how student entity looks like
{
     int roll_no;       // these three are members of the structure
     char name[20];
     int marks;
     struct dob d1;               //using structure variable of dob inside student
 };
```

It can be **understood from example that dob (date of birth) is the structure with members as date, month and year. Dob is then** used as a member in Student structure.

2. **Embedded structure:** It enables us to declare the structure inside the structure.

```
struct student              // template declaration which describes how student entity looks like
{
    int roll_no;             // these three are members of the structure
    char name[20];
    int marks;
    struct dob              //declaring structure dob inside student
    {
        int date;
        int month;
        int year;
    }d1;
};
```

Accessing Nested Structure Members: We can access the member of the nested structure by using **Outer_Structure.Nested_Structure.member** as given below:

```
int main()
{
    struct student s1 = {24, "John", 78, {20, 03, 2000}};   //Declaration and Initialization
    printf("\n Roll no. = %d, Name= %s, Marks = %d, dob = %d.%d.%d ", s1.roll_no,
    s1.name,s1.marks, s1.d1.date,s1.d1.month, s1.d1.year);
    return 0;
}
```

Array of Structures

Consider,

```
struct student
{       int roll_no;
        char name[20];
        int marks;
};
```

Given the above user defined type for student entity, to store the details of one student, create one variable of this structure type.

```
struct student s1;
```

To store the details of 2 students, create two variables of this structure type.

```
struct student s1,s2;
```

To store the details of 100 students, there is a need to create 100 **different structure variables**. As all the variables are of same type, we can use array here. So this can be done easily by making use of **Array of structures.**

```
struct student s[100];
```

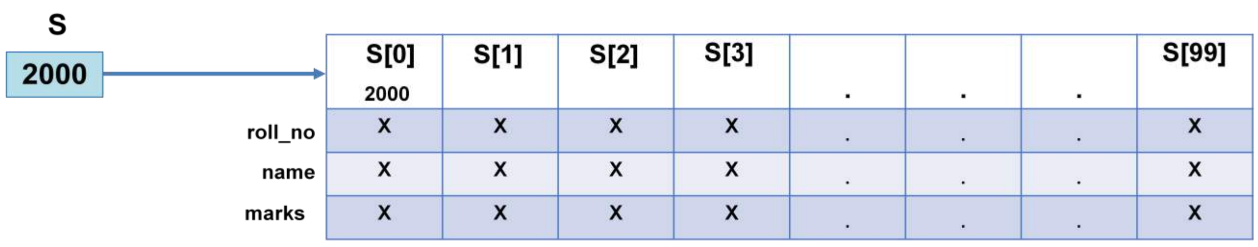
Declaration: Can be done in two ways

1. Along with structure declaration after closing } before semicolon (;) of structure body.

```
struct student
{
        int roll_no;
        char name[100];
        int marks;
}s[100];
```

2. After structure declaration (either inside main or globally using struct keyword)

```
struct student s[100];
```



Initialization:

Involves **Compile-time Initialization** and **Runtime Initialization.**

Compile-time Initialization:

struct student S1[] = { {1, "John", 60}, {2,"Jack", 40}, {3, "Jill", 77} };

// **size is decided by the compiler** based on how many students details are stored

struct student S2[3] = { {1, "John", 60}, {2,"Jack", 40}, {3, "Jill", 77} };

// size is specified and initialized values are exactly matching with the size specified.

struct student S3[3] = {1, "John", 60, 2,"Jack", 40, 3, "Jill", 77};

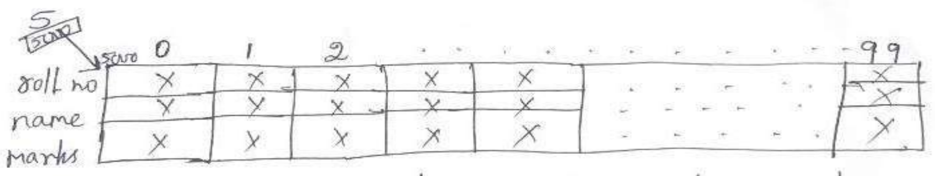
//initialization can also be done this way

struct student S4[5] = { {1, "John", 60} };

//**partial initialization**. Default values are stored in remaining memory locations

Runtime Initialization: Example code is as below.

```
int main()
{         struct student s[100];             // 100 student variables created with the same name s.
          // all members are initialized to undefined values in the beginning
          //printf("%d---------\n",s[3].marks); // prints undefined value
```



Then from s[0] to s[2], values entered by the user are stored and displayed.

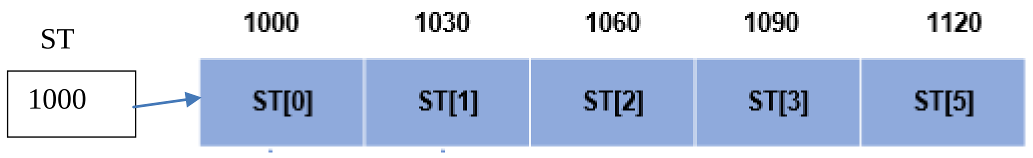
```
int i;
for( i=0;i<3;i++)
{
        printf("enter roll_num, name and marks\n");
        scanf("%d",&s[i].roll_no);                    // observe . operator
        scanf("%s",s[i].name);
        fflush(stdin);
        scanf("%d",&s[i].marks);
}
printf("details entered are --------- \n");
for(i=0;i<3;i++)
{       printf("roll_num, name and marks\n");
        printf("%d\t",s[i].roll_no);
        printf("“%s\t”, s[i].name);
        printf("%d\n",s[i].marks);
}
return 0;
}
```

Pointer to an Array of Structures

Pointer is used to access the array of structure variables efficiently. Suppose we have to store record of 5 students, then using array of structure it can be easily done as:

```
struct student
{       int roll_no;

        char name[22];

        int marks;

}ST[5];
```

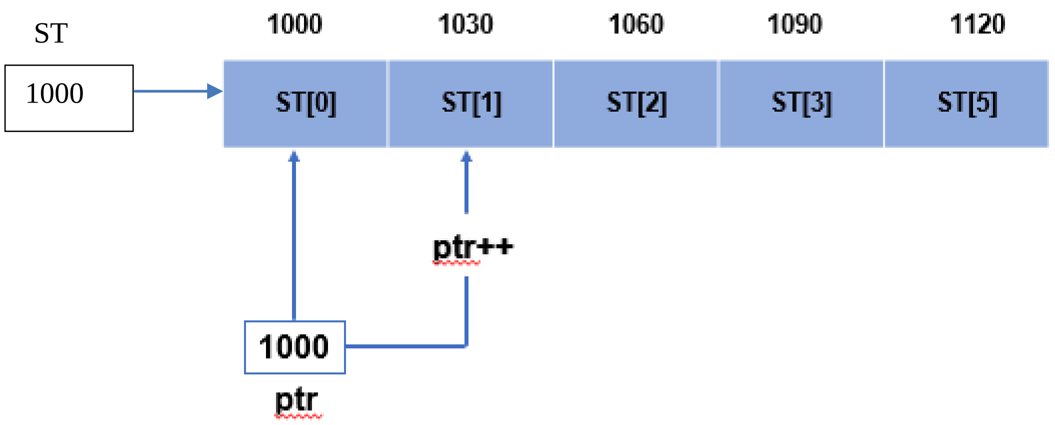
The sizeof operator when applied on structure student will take at least 30 bytes in memory. (Please note this is implementation specific). Suppose, base address of the array ST is 1000, then pictorial representation of the same in memory will be:



Now, if use the pointer ptr initialized to ST as:

struct student *ptr = ST;

It means that pointer ptr is of type student structure and is holding the address pointed by the array of structure ST (which is 1000).



Now, suppose if we perform ptr++, the ptr gets updated and now it will start pointing to next array record starting from 1030 (ptr++ → ptr+1 → 1000 + 30 = 1030), which is the record of next student.

Example:

```
struct student ST[] = {{1, "John", 60}, {2, "Jack", 40}, {3, "Jill", 77}, {4, "Sam", 78 }, {5, "Dean", 80}};

struct student *ptr = ST;        //&ST is illogical

printf("\n%d\n", *ptr);          //prints 1

printf("%d \t %s \t %d\n", ptr->Roll_no, (ptr+1)->Name, (ptr+2)->Marks);

        //prints 1 Jack 77

ptr++;          // ptr now points to ST[1]

printf("\n%d\n", *ptr);          //prints 2
```

How we can use this ptr to print the details of all students??

```
int i;
for(i = 0; i < (sizeof(ST)/sizeof(ST[0])) ; i++)
{    printf("roll_num, name and marks\n");
     printf("%d\t",(ptr+i)->roll_no);   // ptr[i].roll_no
     printf("“%s\t”, (ptr+i)->name);
     printf("%d\n",(ptr+i)->marks);
}
```

Let us write the C code by creating a type called Book. The Book entity contains these data members: id, title, author, price, year of publication. Write separate functions to read details of n books and display it. Also include functions to do the following.

Fetch all the details of books published in the year entered by the user.

Fetch all the details of books whose author name is entered by the user. Display appropriate message if no data found.

Separate the interface and implementation.

Let us begin with the header file. **Header file contains the below code: book.h**

```
typedef struct Book {
    int id;
    char title[100];
    char author[100];
    int price;
    int year;
}book_t;
void read_details(book_t *b,int n);
void display_details(book_t *b, int n);
int fetch_books_year(book_t *b,int n, int year, book_t*);
int fetch_books_author(book_t *b,int n, char *author, book_t*);
```

Client code is as below: book_client.c

```
//Add appropriate header files and int main() and return 0 with { and }

        book_t b[100];

        book_t b_year[100];
        book_t b_author[100];
        printf("How many books details you want to enter?");
        int n;
        scanf("%d",&n);
        printf("enter the details of %d books\n",n);
        read_details(b,n);
        int count;
        printf("enter the year of publication to find list of books in that year");
        int year;
        scanf("%d",&year);
        count = fetch_books_year(b,n, year,b_year);
        if(count)
        {
                printf("List of books with %d as year of publication\n",year);
                display_details(b_year, count);

        }
        else

                printf("books published in %d is not available in the dataset\n",year);

        printf("\n");
        printf("enter the author name to find the list of books by that author\n");
        char author[100];
        scanf("%s",author);
        count = fetch_books_author(b,n, author,b_author); if(count)
        if (count)
        {   printf("List of books by %s\n",author);
            display_details(b_author, count);
        }

        else
            printf("books by %s is not available in the dataset\n",author);
```

Server code is as below: book.c

```
#include "book.h"
#include<stdio.h>
#include "string.h"
void read_details(book_t *b,int n)
{       int i;
        for(i = 0; i< n;i++)
        {
                printf("enter id, title, author, price and year of publication for book %d\n",i+1);
                scanf("%d%s%s%d%d",&b[i].id, b[i].title, b[i].author, &b[i].price, &b[i].year);
                // &((b+i)->id) is also valid in scanf. Using the pointer notation
        }
}
void display_details(book_t *b, int n)
{       int i;
        for(i = 0; i< n;i++)
        {
            printf("\n-->%d\t%s\t%s\t%d\t%d\n",b[i].id, b[i].title, b[i].author, b[i].price,
        b[i].year);
            // (b+i)->id is a valid pointer notation to print id
        }
}

int fetch_books_year(book_t *b,int n, int year,book_t *b_year)
{       int i;int count = 0;
        for(i = 0; i< n;i++)
        {       if (b[i].year == year)
                {
                        count++;        b_year[count-1] = b[i];
                }
        }
        return count;
}
```

```
int fetch_books_author(book_t *b,int n, char *author, book_t *b_author)
{
        int i; int count = 0;
        for(i = 0; i< n;i++)
        {       if (!(strcmp(b[i].author, author)))
                {
                        count++;        b_author[count-1] = b[i];
                }
        }
        return count;
}
```

Corresponding make file is as below and executable name is book

```
book : book.o book_client.o
        gcc book.o book_client.o -o book
book.o: book.c book.h
        gcc -c book.c
book_client.o: book_client.c book.h
        gcc -c book_client.cs
```

Let us consider solving the below problem.

Create a structure which holds various attributes (e.g. name, id, basic_salary, DA%, HRA%, total_salary etc.) of an employee. Write a program which allows you to scan these (except total_salary) attributes for n employees. The program should support following operations:

i. calculate_total_salary (total salary of the selected employee)

ii. Max (find and display name of the employee with maximum basic salary)

Solution:

Client code provided here and I do not want to change the client at any cost. Adding the the source files and header files as per this client only.

```
// employee_client.c
#include<stdio.h>
#include "employee.h"
int main()
{
        employee_t e[1000];
        printf("Enter the number of employees for which the details has to be entered\n");
        int n, id;
        scanf("%d", &n);
        printf("enter name, id and basic salary of %d employees\n",n);
        scan_details(e,n);
        printf("enter the id of the employee whose total salary you want to know\n");
        scanf("%d",&id);
        int tot_salary = calculate_total_salary(e,n,id);
        if (!tot_salary)
          printf("employee with entered id doesnt exist\n");
        else
          printf("total salary of employee with id %d is %d\n",id,tot_salary);
        printf("Employee details entered is as below\n");
        display_details(e,n);
        employee_t e_max = max(e,n);
        printf("employee with maximum basic salary is %s",e_max.name);
        return 0;
        }
```

Header file: employee.h

```
typedef struct employee
{
        char name[20];
        int id;
        int basic_salary;
        int total_salary;
}employee_t;
// You might want to have DA and HRA as data members. Try that. But make sure client is not
        touched
void scan_details(employee_t*, int);
void display_details(const employee_t*, int);
int calculate_total_salary(employee_t*, int,int);
employee_t max(employee_t*, int);
```

Implementations of these functions are available in **source file employee.c** as below.

```
#include"employee.h"
#include<stdio.h>
void scan_details(employee_t* e, int n)
{
        int i;
        for(i = 0;i<n;i++)
          scanf("%s %d %d",(e+i)->name,&(e+i)->id, &(e+i)->basic_salary);
}
void display_details(const employee_t* e, int n)
{
        int i;
        for(i = 0;i<n;i++)
          printf("%s\t%d\t%d\n",(e+i)->name,(e+i)->id, (e+i)->basic_salary);
}

int calculate_total_salary(employee_t* e, int n,int id)
{
        int i;
        int da;
        int hra;
        int t_salary = 0;
        for(i = 0;i<n;i++)
        {
          if((e+i)->id == id)
          {
                da = 0.8*(e+i)->basic_salary;
                hra = 0.2*(e+i)->basic_salary;
                (e+i)->total_salary = da+hra+(e+i)->basic_salary;
                t_salary = (e+i)->total_salary;
          }
        }
        return t_salary;
}
employee_t max(employee_t* e, int n)
{       int i;
        int max = 0;
        employee_t e_max;
        for(i = 0;i<n;i++)
        {
          if((e+i)->basic_salary > max)
          {
                max = (e+i)->basic_salary;
                e_max = *(e+i);
          }
        }
        return e_max;
}
```

Few more programs for you to solve are listed here.

Happy Coding..!!

- Program to read and display the details of n books using functions. Clearly separate interface and implementation.
- Program to display all information about a student who has scored highest marks in a class. Consider inputs: 5 students (info.- name, roll_no, dob, marks scored in 5 different subjects)
- Program to store the details of 5 flights. Also display all flights information sorted in order of their arrival times.

Dynamic Memory Management

Points to think:

- In case of an array, memory is allocated before the execution time. Is there over utilization or under utilization of memory?
- Can we take the size of the array at runtime and request to allocate memory for the array at runtime? – This is Variable length Array(VLA).

  It is not a good idea to use this as there is no functionality in VLA to check the non availability of the space. If the size of large, results in code crash without any intimation.
- Can we avoid this by allocating memory whenever and how much ever we require using some of the functions?

In C, Memory can be allocated for variables using different techniques.

1. **Static allocation**

   decided by the compiler

   allocation at load time [before the execution or run time]
2. **Automatic allocation**

   decided by the compiler

   allocation at run time

   allocation on entry to the block and deallocation on exit
3. **Dynamic allocation**

   code generated by the compiler

   allocation and deallocation on call to memory allocation functions: malloc, calloc, realloc and deallocation function: free

In this chapter, we are dealing with Dynamic Memory Management functions

Dynamic Allocation of Memory

The process **of allocating memory at runtime/execution time** is known as dynamic memory allocation. This **happens in the heap region of Memory segment**.

More details on Memory layout of C can be studied in the below link.

https://www.geeksforgeeks.org/memory-layout-of-c-program/

There is **no operator in C to support dynamic memory management**. Library routines known as "Memory management functions" are used for allocating and freeing/releasing memory during execution of a program. These functions are defined in stdlib.h

malloc():

Allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space. Prototype**: void *malloc(size_t size);**

The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. This returns a void pointer to the allocated memory that is suitably aligned for any kind of variable. On error, these functions return NULL. NULL may also be returned by a successful call to malloc() with a size of zero.

calloc():

Allocates space for elements, initialize them to zero and then return a void pointer to the memory. prototype: **void *calloc(size_t nmemb, size_t size);**

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The calloc() function return a pointer to the allocated memory that is suitably aligned for any kind of variable. On error, this function return NULL. NULL may also be returned by a successful call to calloc() with nmemb or size equal to zero

realloc():

Modifies the size of previously allocated space using above functions. copies the old values into the new memory locations. **Prototype: void *realloc(void *ptr, size_t size);**

This function changes the size of the memory block pointed to by ptr to size bytes. The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new size is larger than the old size, the added memory will not be initialized.

Note:

If ptr is NULL, then the call is equivalent to malloc(size), for all values of size.

If size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).

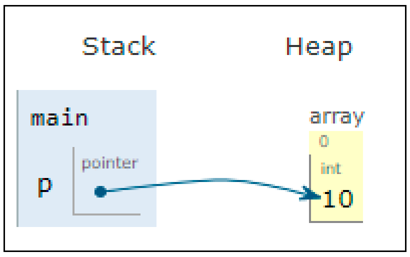
Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done. The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails the original block is left untouched, and it is not freed or moved

free(): Releases the allocated memory and returns it back to heap. Must be applied with malloc(), calloc() or realloc()

Let us consider the Memory Allocation using malloc

Code 1:

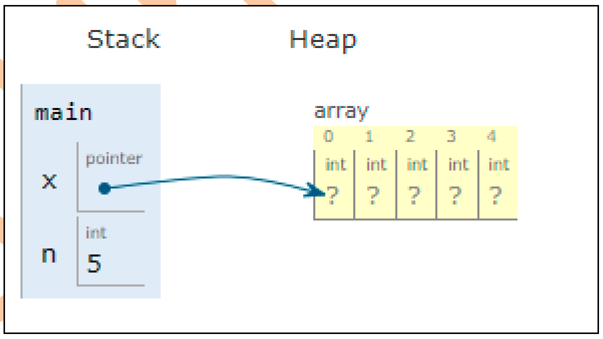
```
 int* p = (int*)malloc(sizeof(int)); //dynamic allocation for one integer
                                        //address of that location is returned in p
*p = 10;
printf("p = %d", *p);
```

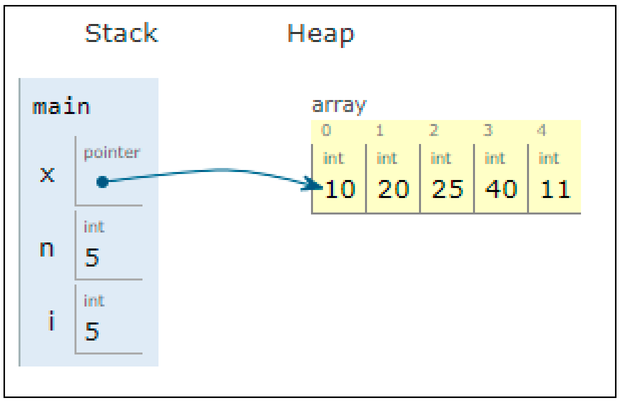


Code 2:

```
int* x;
int n, i;
printf("\n\nEnter number of elements:\n");
scanf("%d", &n);
```

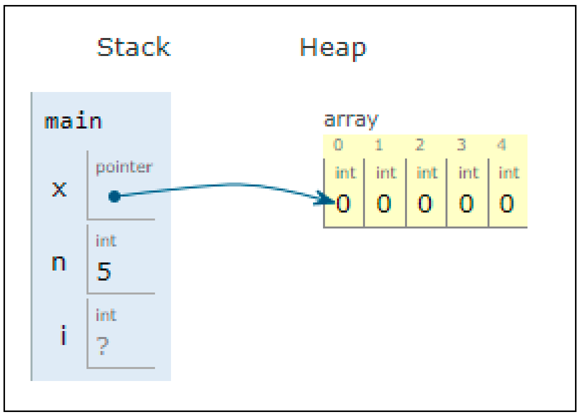
```
x = (int*)malloc(n * sizeof(int));  //memory will be allocated for 5 integers
// It is important to check if the memory has been successfully allocated by malloc or
not
if (x == NULL)
        printf("Memory not allocated.\n");

else
{
        printf("Memory successfully allocated using malloc.\n");
        printf("Enter the integer values:\n ");
        for (i = 0; i < n; ++i) // Scan the values
                scanf("%d", x+i);

        printf("Output: ");// Printing
        for (i = 0; i < n; ++i)
                printf("%d\t", *(x+i));
}
```

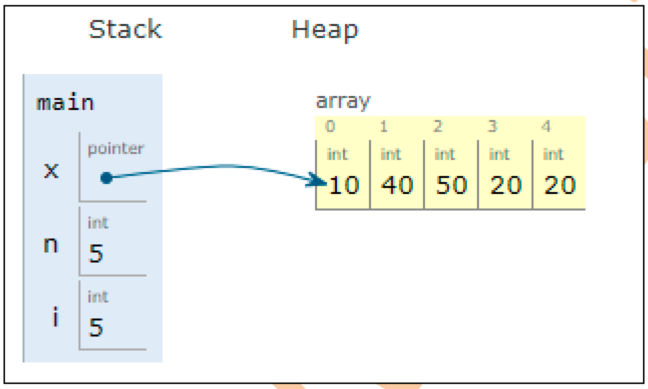




Let us consider the Memory Allocation using calloc

```
int* x;
int n, i;
printf("Enter number of elements: \n");
scanf("%d", &n);
x = (int*)calloc(n, sizeof(int));          // a block of n integer(array) is created dynamically

// To check if the memory has been successfully allocated by calloc

if (x == NULL)
            printf("Memory not allocated.\n");

else
{
            printf("Memory successfully allocated\n");
            printf("The elements of the array before reading from the user\n");
            for (i = 0; i < n; ++i)
            {
                        printf("%d, ", *(x+i));
            }
            printf("\n Enter the elements:\n");
            for (i = 0; i < n; ++i)   //Scan elements of array
             {
                        //scanf("%d", &x[i]);
                        scanf("%d", (x+i));
            }

            printf("The elements of the array are: ");// Printing the elements
            for (i = 0; i < n; ++i)
            {
                        printf("%d, ", x[i]);
            }
}
```



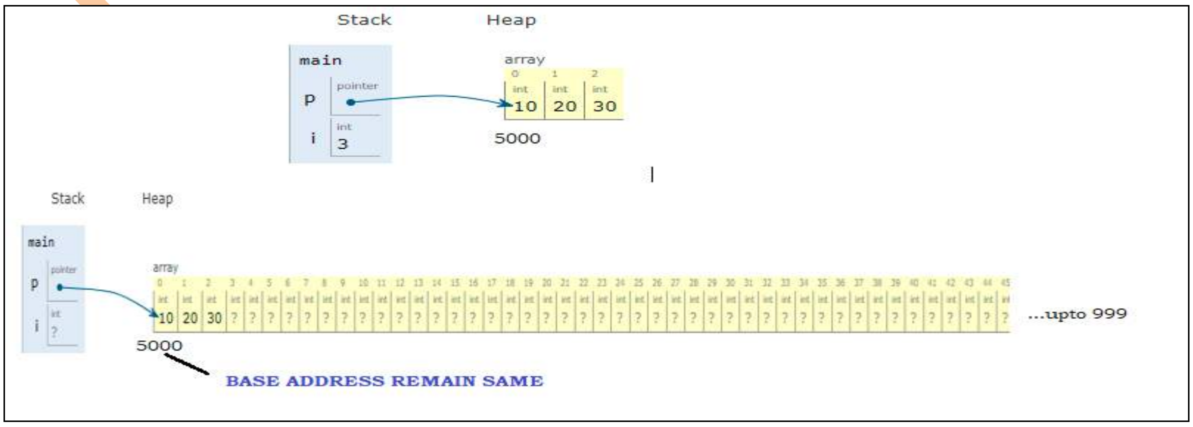


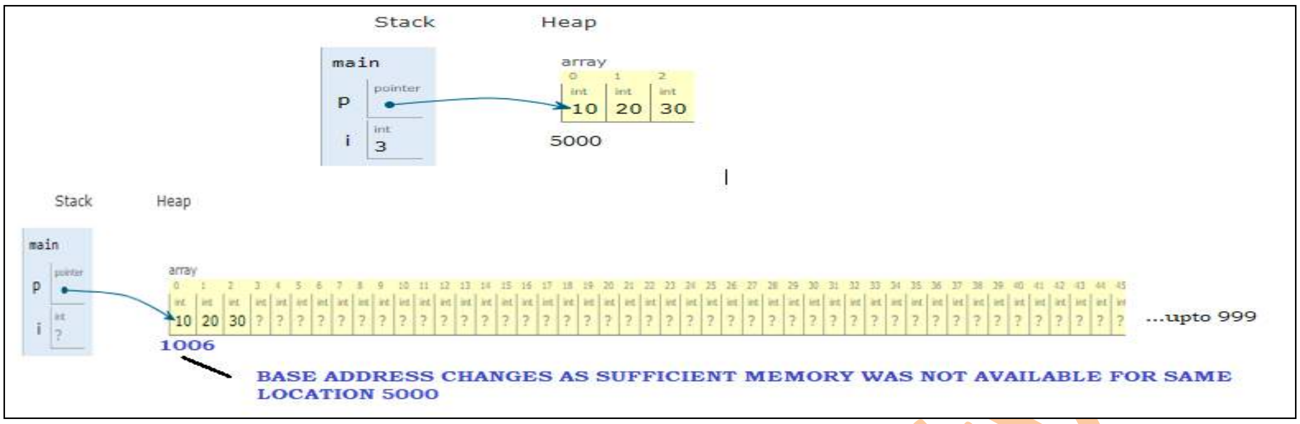
Let us consider the use of realloc function.

```
int main()
{
    int* p = (int*)malloc(3*sizeof(int));
    if(p == NULL)
            printf("memory not allocated\n");
    else
    {
            int i;
            printf("initial address is %p\n",p);
            printf("enter three elements\n");
            for(i = 0;i<3;i++)
            {
                    scanf("%d",&p[i]);
            }
            printf("entered elements are\n");
```

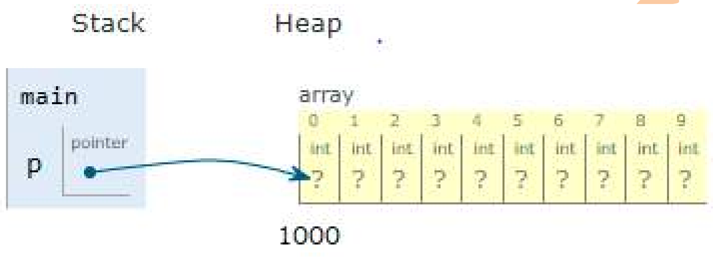
```
                for(i = 0;i<3;i++)
                {
                        printf("%d\t",p[i]);
                }
        }
        printf("enter the new size\n")
        int new_size;
        scanf("%d",&new_size);
        p = (int*)realloc(p,new_size*sizeof(int));
        if(p==NULL)
                printf("not allocated\n");
        else
        {
                int i;
                printf("\nnew address is %p\n",p);  // this address might not be the same as the initial
address when new size if more than 3(which is hardcoded initially). If new size is lesser, this
address and initial address must be same.
                for(i = 0;i<new_size;i++)
                {
                        printf("%d\t",p[i]);  // all the values are copied to new location if extending
the memory space was not possible. So prints first three elements properly. Then it is undefined
                }
        }
        return 0;
}
```

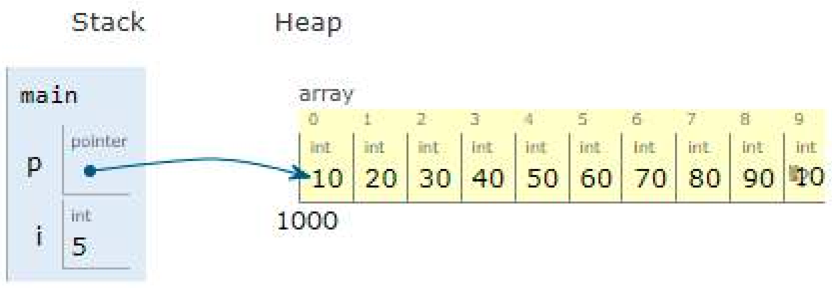
When new size is greater than 3, here are the pictorial representations

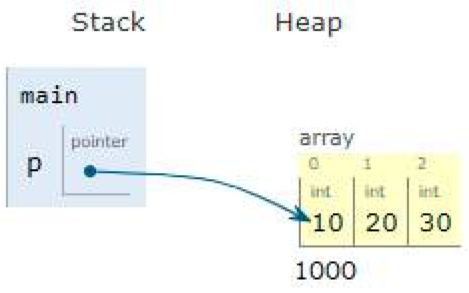



Stack
Heap
main
p
pointer
i
int
3
array
10 20 30
5000
...upto 999
1006
BASE ADDRESS CHANGES AS SUFFICIENT MEMORY WAS NOT AVAILABLE FOR SAME LOCATION 5000

When new size is less than 3, here is the pictorial representation

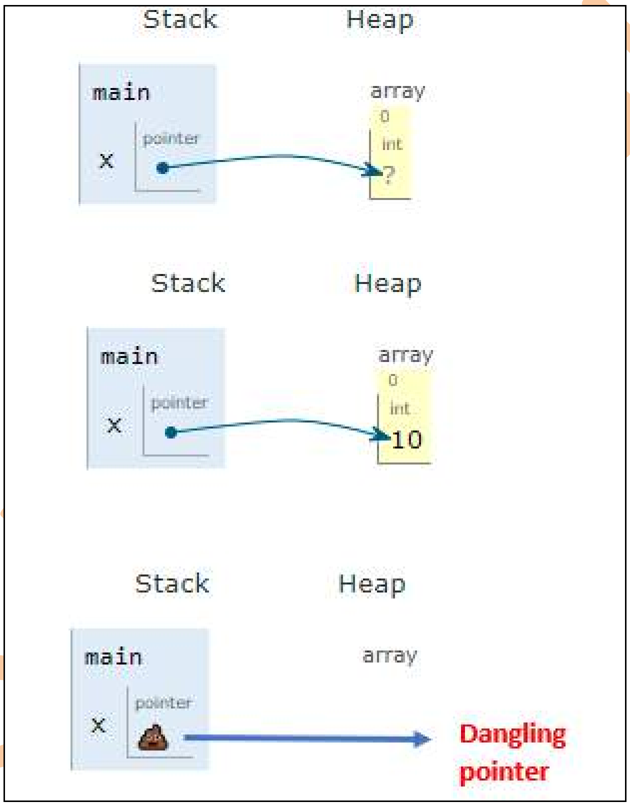






Let us consider the usage of free

```
int *x;
x = (int*)malloc(sizeof(int));
*x = 10;
printf ("\nx = %p", x);
printf("\n x is pointing to = %d", *x);
// Free the memory
  free(x);
  printf("\nMalloc Memory successfully freed.\n");
```



How does free function knows how much memory must be returned/released?

On allocation of Memory using memory management functions, it stores somewhere in memory the # of bytes allocated. This is known as **book keeping information**. We cannot access this info. But implementation has access to it. The function free finds this size given the pointer returned by allocation functions and de-allocates the required amount of memory.

By observing both the diagrams above, we can get to know that block allocated using functions will be deallocated and hence the pointer becomes dangling.

Points to think:

- Can free function take one more argument along with the pointer? -- No.
- Is it possible to call free on p1 in version 1 ?
- Can you use free on the same pointer twice?
- What happens when free(p2+1) in version 2?

Common Programming Errors during Dynamic Memory Management

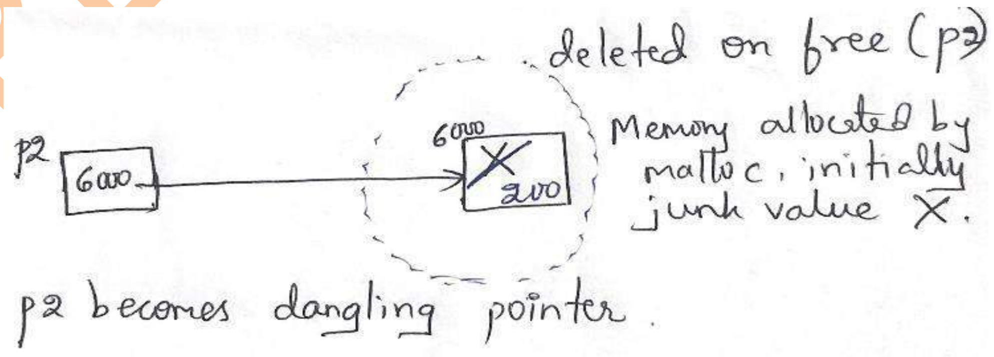
Dangling Pointer

- Points to a location which doesn't exist
- Freeing the memory results in dangling pointer
- Using new pointer variable to store return address in realloc results in dangling pointer
- Dereferencing the dangling pointer results in undefined behavior
- Can happen anywhere in the memory segment
- Solution is assigning the pointer to NULL

Case 1: Freeing the memory results in dangling pointer

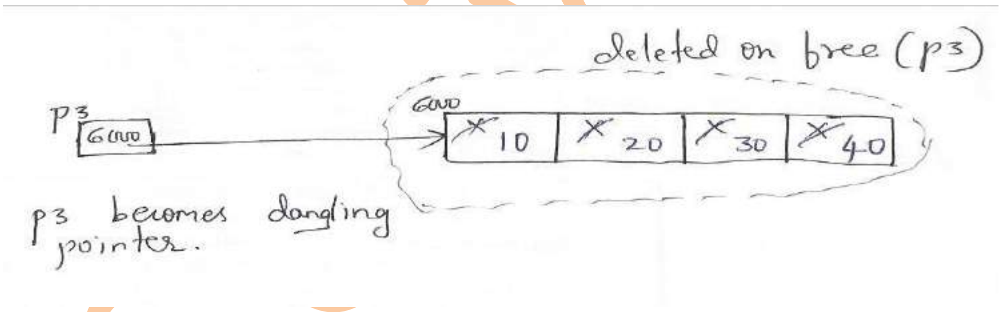
Version 1:

```
int *p2 = (int*)malloc(sizeof(int));    // conversion from void* to int*
// allocate memory at runtime, malloc returns void*
printf("%p\n",p2);
//p2 = &200;  //Error
*p2 = 200;
printf("%p\n", p2);
printf(" %d\n", *p2);              // 200
free(p2);
// returns the memory allocated using malloc
```


deleted on free(p2)
p2
6000
6000
200
Memory allocated by malloc, initially junk value X.
p2 becomes dangling pointer.

Version 2:

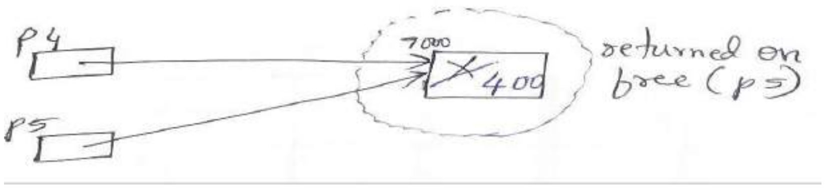
```
printf("Enter the number of integers you want to store\n");
int n; int i;
scanf("%d", &n);                        // user entered 4
int *p3 = (int*)malloc(n*sizeof(int));  // initially all values undefined values
printf("Enter %d elements\n", n);
for(i = 0;i<n; i++)
        scanf("%d",&p3[i]);        // (p3+i)      // user entered 10 20 30 40
printf("Entered elements are\n");
for(i = 0;i<n; i++)
        printf("%d\n",p3[i]);      // *(p3+i)
free(p3);                          // returns the memory allocated using malloc
```



Version3:

```
int *p4 = (int*)malloc(sizeof(int));
*p4 = 400;
printf(" %d\n", *p4);          // 400
int *p5 = p4;
free(p5);        // value of the pointer has the book keeping info available
```

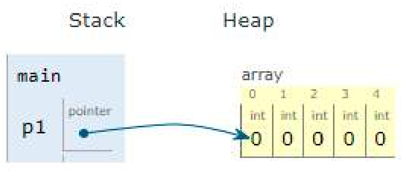
//Both p4 and p5 becomes dangling pointer

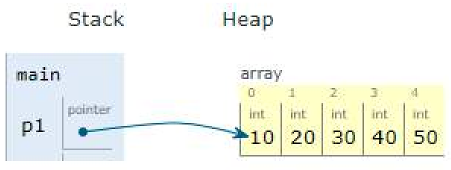


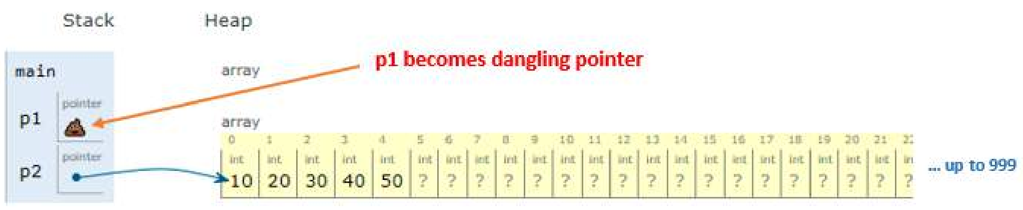
Case 2: Using new pointer variable to store return address in realloc results in dangling pointer

Consider the below code.

```
int *p1 = (int *) calloc(5, sizeof(int));
printf("before %p\n", p1);
printf("enter the elements\n");
for(int i = 0; i < 5; i++)
        scanf("%d", &p1[i]);//given input 1,2,3,4,5
for(int i = 0; i < 5; i++)
        printf("%d ", p1[i]);
int *p2 = (int*) realloc(p1, 1000*sizeof(int)); // p1 becomes dangling if new
locations allotted
        printf("after increasing size %p\n", p2); // same address as p1 if same location can
be extended else Different address
printf("content at address pointed by p2 = %d\n", *p2);
printf("after realloc %p\n", p1);
printf("contents at address pointed by p1 = %d\n", *p1);
//undefined behaviour- p1 becomes dangling pointer
```





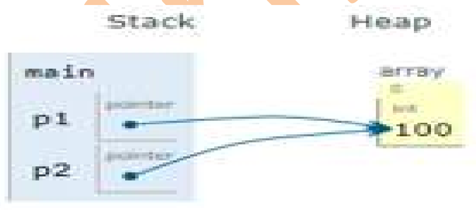


NULL Pointer

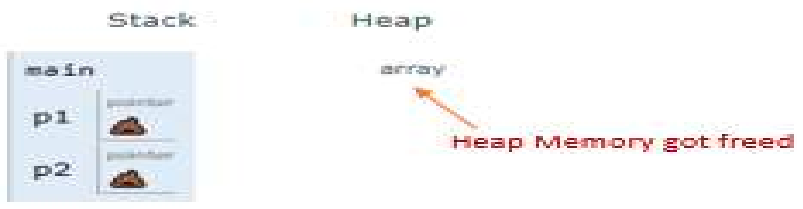
It is always a good practice to assign the pointer to NULL once after the usage of free on the pointer to avoid dangling pointers. This results in NULL Pointer. **Dereferencing the NULL pointer results in Guaranteed crash.**

Consider the below code.

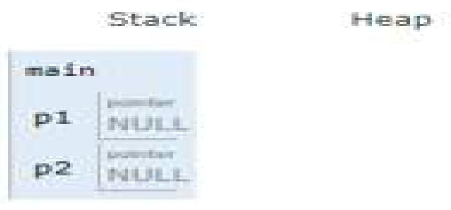
```
int *p1 = (int*)malloc(sizeof(int));
*p1 = 100;
printf(" %d\n", *p1); // 100
int *p2 = p1;
printf(" %d\n", *p2); // 100
free(p1); // p1 and p2 both becomes dangling pointer.
p1 = NULL; // safe programming
p2 = NULL;
printf(" %d %d\n", *p1, *p2);// Guaranteed crash
```



Both p1 and p2 are pointing to same heap location



As free(p1) happens, p1 and p2 becomes dangling pointer.



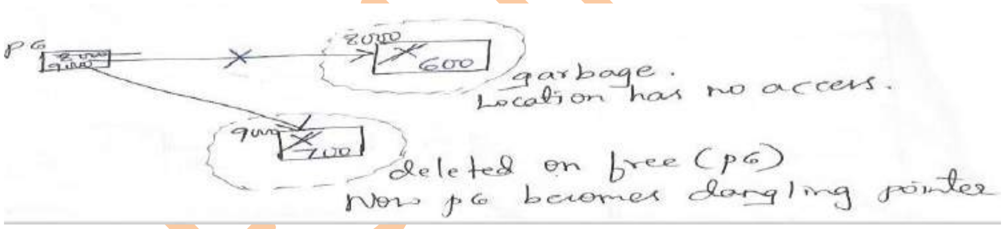
Assigning p1 and p2 as NULL pointer makes the code safe

Garbage and Memory Leak

- Garbage is a location which has no name and hence no access
- If the same pointer is allocated memory more than once using the DMA functions, initially allocated spaces becomes a garbage.
- Garbage in turn results in memory leak.
- Memory leak can happen only in Heap region

Consider the below code.

```
        int *p6 = (int*)malloc(sizeof(int));
        *p6 = 600;
        printf(" %d\n", *p6); // 600
        p6  =  (int*)malloc(sizeof(int));       // changing p6 loses the pointer to the location
allocated by the previous malloc
        // So memory allocated by previous malloc has no access. Hence becomes garbage
        *p6 = 700;
        printf(" %d\n", *p6);            // 700
        free(p6);
```



Double free error: DO NOT TRY THIS

- If free() is used on a memory that is already freed before
- Leads to undefined behavior.
- Might corrupt the state of the memory manager that can cause existing blocks of memory to get corrupted or future allocations to fail.
- Can cause the program to crash or alter the execution flow

Consider the below code.

```
int* p = (int*)malloc(sizeof(int));
*p = 10;
printf("p = %d", *p);
free(p);
free(p);
```

The above might lead to undefined behavior. So, it is a good practice to make the pointer NULL after the usage of free. By chance, if the pointer is freed again, it doesn't do anything with NULL.

```
free(p);
p = NULL; //Function free does nothing with a NULL pointer.
free(p);
```

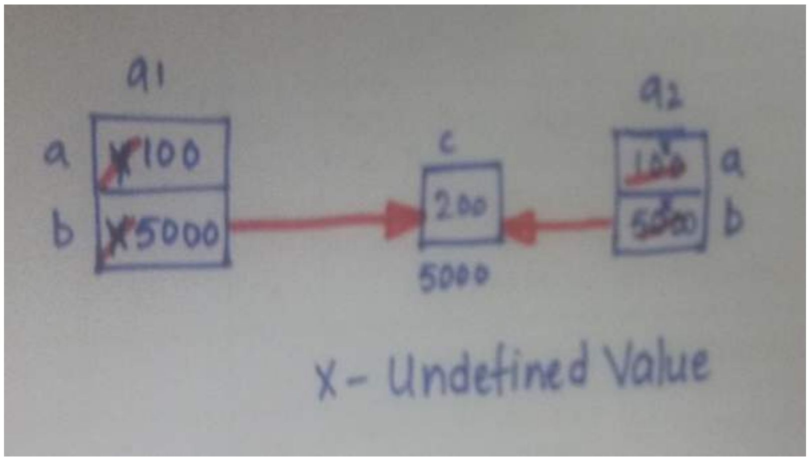
INTRODUCTION

A linked list is a collection of nodes that are connected via links. The node and link has a special meaning which we will be discussing in this chapter. Before we deal with this in detail, we need to answer few questions.

Can we have pointer data member inside a structure? Yes. Refer to below codes.

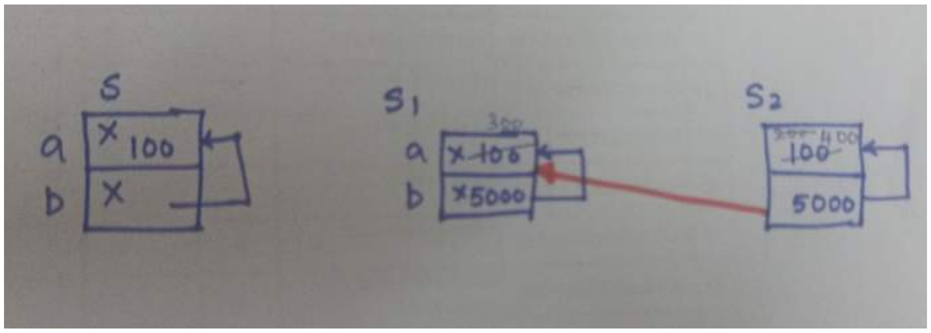
Version 1:

```
struct Sample A
{
        int a;
        int *b;
};
int main()
{
struct A a1;
struct A a2;
a1.a = 100;
int c = 200;
a1.b = &c;
printf("a1 values:%d and %d\n", a1.a, *(a1.b));         // 100 200
a2 = a1;
printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 200 a1.a = 300;
*(a1.b) = 400;
printf("a2 values:%d and %d\n", a2.a, *(a2.b));         // 100 400
}
```



Version 2:

```
struct Sample
{
        int a;
        int *b;
};

#include<stdio.h>
int main()
{
       struct Sample s;
        s.a = 100;
        s.b = &(s.a);
        printf("%d %d",s.a,*(s.b));
        struct Sample s1;
        s1.a = 100;
        s1.b = &(s1.a);
        printf("%d %d\n",s1.a,*(s1.b));
        struct Sample s2 = s1;
        printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s2.a,*(s2.b));
        s2.a = 200;
        printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s1.a,*(s1.b));
        printf("-----%d %d\n",s2.a,*(s2.b)); // very imp
        *(s2.b) = 300;
        printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s1.a,*(s1.b));
        printf("%d %d\n",s2.a,*(s2.b));
        s2.b = &(s2.a);
        *(s2.b) = 400;
        printf("%p %p\n",s1.b,s2.b);
        printf("%d %d\n",s1.a,*(s1.b));
        printf("%d %d\n",s2.a,*(s2.b));
        */
}
```



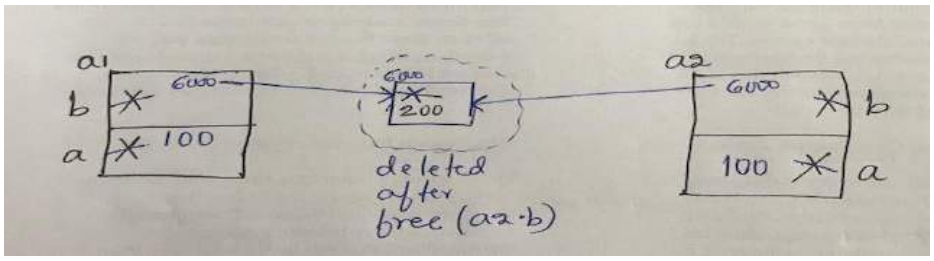
Version 3:

```
int main()
{
 struct A a1;
struct A a2;
a1.a = 100;
a1.b = (int*) malloc(sizeof(int));
*(a1.b) = 200;
printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
a2 = a1;
```

printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 200

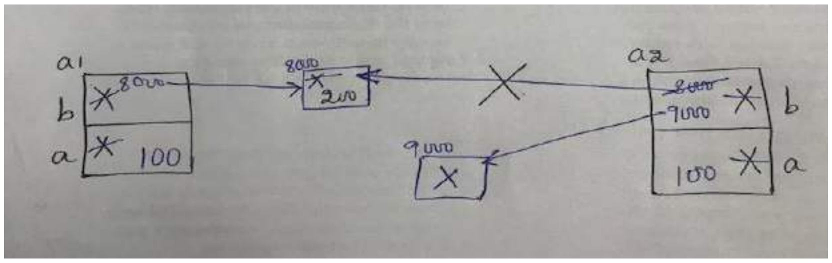
free(a2.b); // a1.b too becomes dangling pointer

printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 undefined behaviour



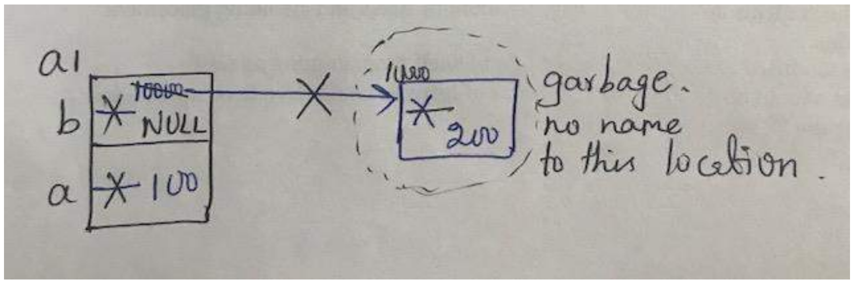
Version 4:

```
int main()
{
struct A a1;
struct A a2;
a1.a = 100;
};

a1.b = (int*) malloc(sizeof(int));
*(a1.b) = 200;
printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
a2 = a1;
a2.b = (int*) malloc(sizeof(int));
// changing this to a1.b creates garbage and output differs
printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 undefined value
printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
```



Version 5 :

```
int main()
{
    struct A a1;
    a1.a = 100;
   a1.b = (int*) malloc(sizeof(int));
   *(a1.b) = 200; a1.p = NULL; // This creates garbage in heap
```



Can we have structure variable inside another structure?

```
struct A
 {
 int a;
 };

struct B
 {
int a;
struct A a1;
}; // structure variable a1 in B
 int main()
{
printf("sizeof A %d\n",sizeof(structA);
printf("sizeof B %d\n",sizeof(struct B));
}
```

Can we have a structure variable inside the same structure? – No.

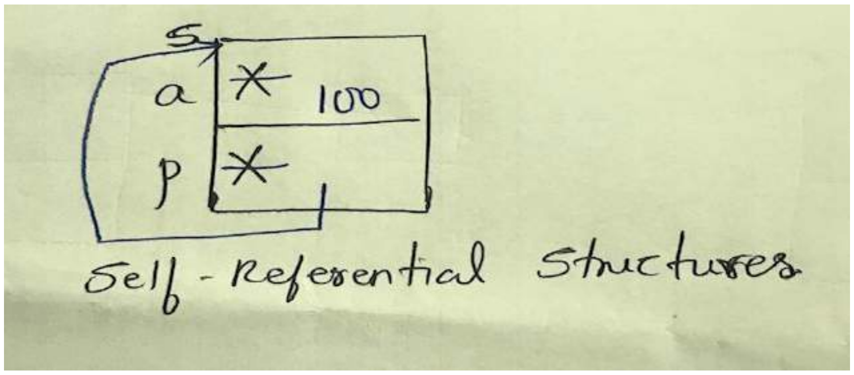
```
struct A {
int a;
struct A a1;
 };
//Compile time Error: field a1 has incomplete type
// size of struct A we cannot find as the field a1 which itself is a structure of the same kind.
```

The solution to the previous version is to have a pointer of the same type inside the structure. The size of a pointer to any type is fixed. Hence, the size of the structure can be computed by the compiler at compile time.

SELF REFERENTIAL STRUCTURE

A structure which has a pointer to itself as a data member is called a self - referential structure.

```
struct Sample
 {
     int a;
     struct Sample *p;
};

int main()
 {
     printf("%d\n",sizeof(struct Sample) ); // implementation specific struct Sample s;
     s.a = 100; s.p = &s;
     printf("%d %d %d\n", s.a, s.p->a, s.p->p->a); // all 100
     return 0;
}
```


s
a
p
*
100
*
Self-Referential Structures

LINKED LIST CHARACTERSTICS:

- A data structure that consists of zero or more nodes. Every node is composed of two fields: a data/component field and a pointer field. The pointer field of every node points to the next node in the sequence.
- We can access the nodes one after the other. There is no way to access the node directly as random access is not possible in a linked list. Lists have sequential access.
- Insertion and deletion in a list at a given position requires no shifting of elements.

Difference Between Arrays and Linked List

1. Array - a collection of similar type data elements

Linked list -a collection of unordered linked elements known as nodes.

2. Array- traversal through indexes.

linked list - traversal through the head until we reach the node.

3. Array - Elements are stored in contiguous address space

Linked List - Elements are at random address spaces.

4. Array - Access is faster.

Linked List - Access is slower.

5. Array- Insertion, and Deletion of an element is not that efficient.

Linked List - Insertion and Deletion of an element is efficient.

6. Array - Fixed Size.

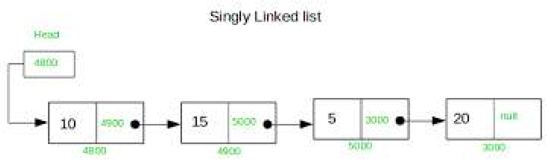
Linked List - Dynamic Size.

7. Array - memory assigned during compile time/ static allocation.

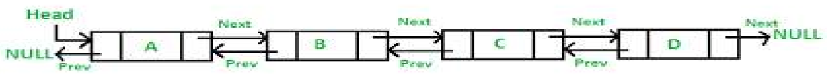
Linked List - Memory assigned during runtime / dynamic allocation.

Types of Linked List

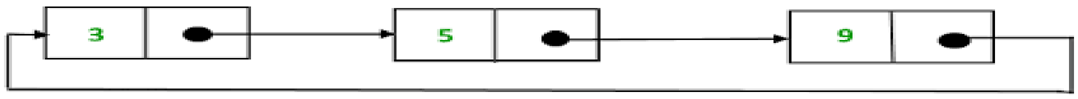
i. Singly Linked List



ii. Doubly Linked List



Iii Circular Linked List



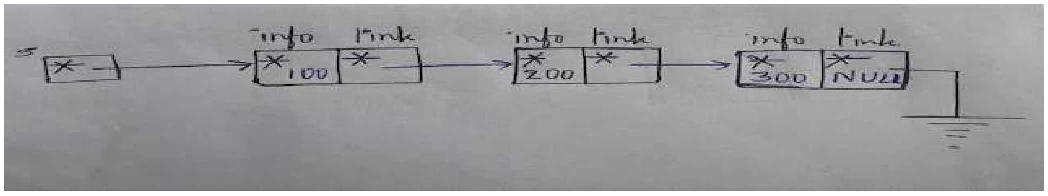
Implementation of Linked List

Structure definition of a node is as

```
struct node
{
int info; // component field
struct node *link; // pointer field
};
typedef struct node NODE_T;
```

Now,

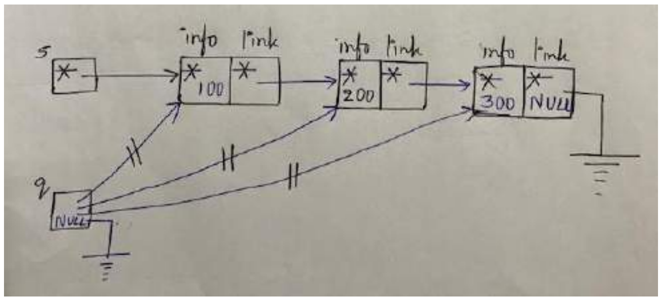
```
int main()
{
NODE_T *s;
s = (NODE_T*) malloc(sizeof(NODE_T));
s->info = 100;
s->link = (NODE_T*) malloc(sizeof(NODE_T));
s->link->info = 200;
s->link->link = (NODE_T*) malloc(sizeof(NODE_T));
s->link->link-> info = 300;
```



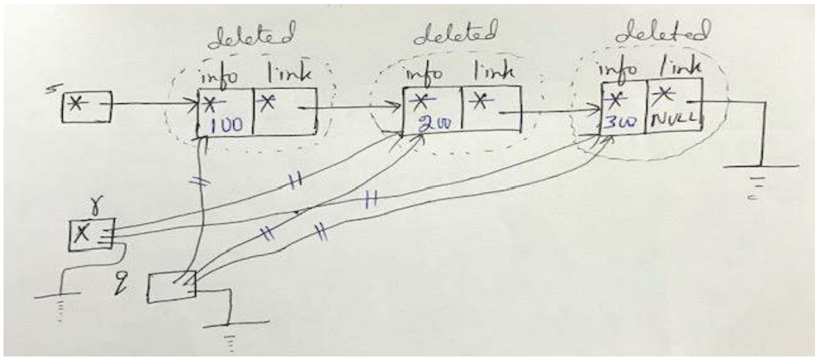
```
s->link->link -> link = NULL;
display(s);
freelist(s); //Why can't we say just free(s); free(s) would release the
//first node and creates garbage for the rest of the list
}
```

Let us write functions to display this list and free every node in the list.

```
void display(NODE_T* q)
{
while(q != NULL)
{
printf("%d\t", q->info); q = q->link;
}
}
```



```
void freelist(NODE_T* q)
{
NODE_T* r; while(q != NULL){
printf("\n%d deleted",q->info);
 r = q->link;
free(q);
q = r;
}
}
```



Ordered List

An ordered list has nodes arranged in the order of info field. Let us create a list of nodes in the increasing order of info in every node.

First we will create empty list and initialize the list to NULL. As and when user enters elements, make a node and insert this node to the list based on the order of the elements of the node. Some of the conditions must be handled while inserting new node to the list. We will discuss the implementations in detail.

Header file is as below.

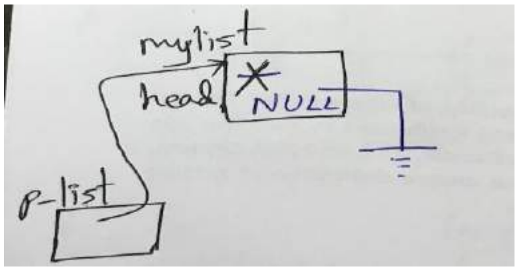
```
struct node
{
int info;
struct node *link;
};
typedef struct node NODE_T;
struct mylist // ordered list contains node.
{
NODE_T* head;
};
typedef struct mylist MYLIST_T;
void initialize_list(MYLIST_T*);
void insert_list(MYLIST_T*,int);
void display_list(const MYLIST_T*);
void free_list(MYLIST_T*);
```

Client Code is as below:

```
 #include"ordered_list.h"
```

```
#include<stdio.h>
int main()
{
MYLIST_T mylist;
//MYLIST_T *mylist;
 initialize_list(&mylist);
printf("enter the number of elements u want to insert\n");
int n,ele;
scanf("%d",&n);
for(int i = 0;i<n; i++)
{
printf("enter the element-->"); scanf("%d",&ele);
insert_list(&mylist,ele);
}
printf("Elements of the list are\n"); display_list(&mylist);
free_list(&mylist);
}
```

Server Side :



```
void initialize_list(MYLIST_T* p_list)
{
p_list->head = NULL;
 }
```

Inserting a node to the list:

There are four cases to be considered while inserting node to the ordered list:

1. empty list
2. insert in the middle
3. insert at the beginning
4. insert at the end

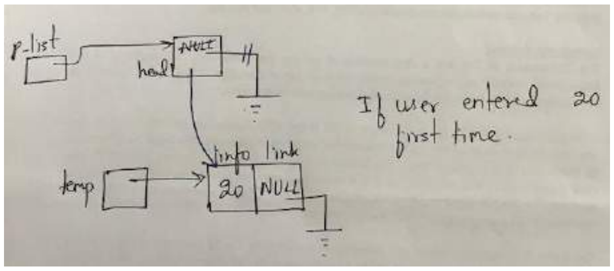
Insert_list: The insert_list function creates a node from the element entered by the user and makes the link field of the node to NULL.

```
NODE_T* temp = (NODE_T*)malloc(sizeof(NODE_T));
temp -> info = ele;
temp -> link = NULL;
```

Then inserts this node in the right position in the list based on the order of the elements and whether the list is empty or not.

Case 1: Inserting to Empty List

```
if(p_list -> head == NULL)
p_list->head = temp;
```



Case 2: Inserting to a Non-empty List

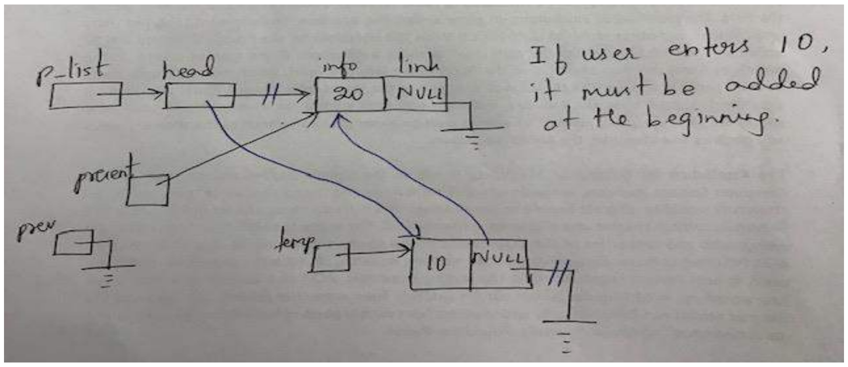
We have to find the position to insert. Make a pointer present point to the first node of the list. Compare the info in that node with the new node pointed to by temp. If the later is greater than the former, we move to the next node by changing present. present = present→link. But once we pass that node, we get to know that we should have inserted the node previous position itself. As we cannot move back, have a prev pointer which always points to a node previous to present. Initially, prev points to

```
NULL.NODE_T* present = p_list->head;
NODE_T* prev = NULL;
while(present != NULL && present->info < ele) // traversal
{
prev = present;
present = present->link;
}
```

This loop exits due to either of these two conditions: found a position to insert the node or we have reached the end of the list.

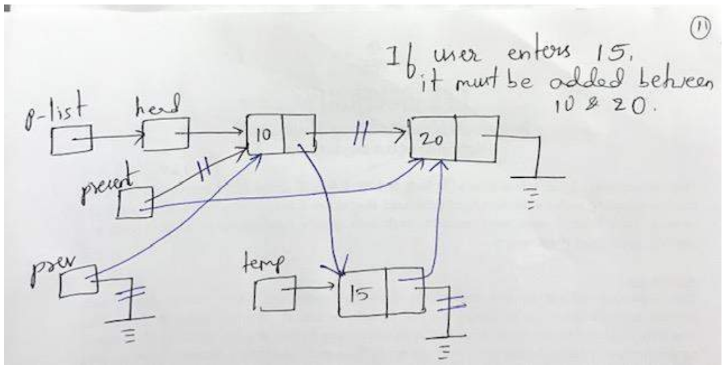
When prev is not changed, it means that node has to be inserted in the beginning only

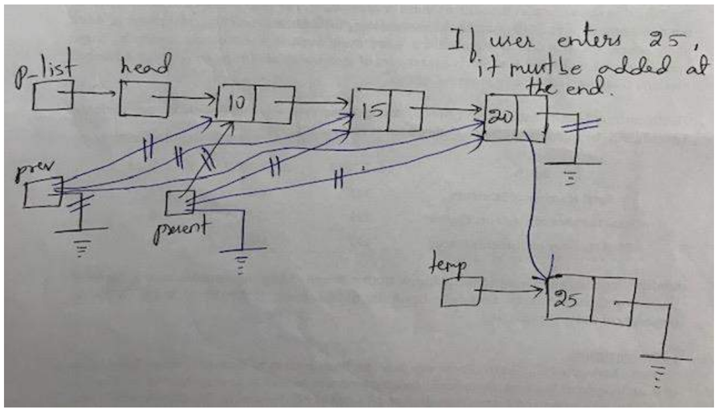
```
if(prev == NULL)
{
temp->link = present;
 p_list->head = temp;
}
```



Code for inserting a node at the end or in the middle is same.

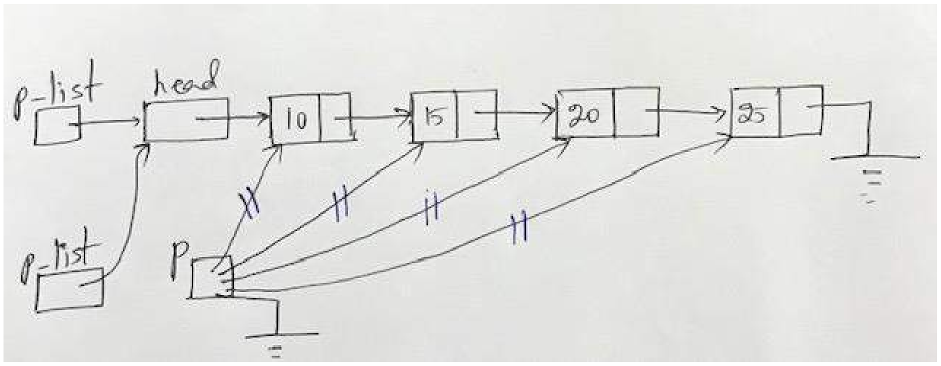
```
else
{
temp->link = present;
prev -> link =temp;
}
```





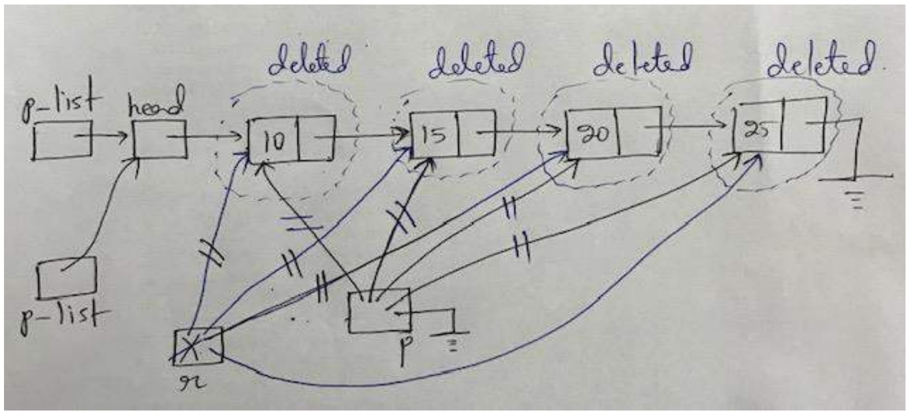
Displaying the list:

```
void display_list(const MYLIST_T* p_list)
// not making any changes to the list. so const
{
NODE_T* p = p_list->head;
while(p != NULL)
{
printf("%d->",p->info);
p = p->link;
}
printf("\n");
}
```



Freeing the list:

```
void free_list(MYLIST_T* p_list)
{
NODE_T* p = p_list->head;
NODE_T* r;
 while(p!=NULL)
{
r = p;
p = p->link;
free(r);
printf("\nnode is freed");
}
}
```



Problem Solving: Scheduling Events

Given a set of events and dates of those events, process the event or schedule the events based on the year/month/day - choose any one for coding.

Problem statement in detail:

Create a structure date with required data members. Create an event with date and details. Create an array of events. Read and display array of events. Problems to be solved based on above structures:

1) Count the number of events in any month specified by the user.

2) Check whether the given two dates of events are in sequence.

Optional: Invalid dates are not taken care. Try handling invalid dates

Requirements:

1. Creation of **two new user defined types**: **date and event**

   typedef struct date {int dd; int mm; int yy; } date_t;

   typedef struct event { char event_name[10]; date_t date; } event_t;

2. Add functions to read and display the date. Use these functions to read and display the event details.

3. Add **is_month_matching** and **is_date_in_order** functions in date. Use these functions to check whether the **event is in the particular month** and **whether the event is in order or not**(consider either date or month).

4. Add **functions to read and display an array of events**. Also add a function to **find the count of events in that month.**

Client code requirement:

Step1: Create an array of events.

Step2: Read the number of events from the user. Say n.

Step3: Call read and display functions to read and display the event details.

Step4: Read the month number from the user and check whether it is valid month. If yes, print the number of events in that month. Else, display appropriate message.

Step5: Read two event numbers to check whether they are in order. Display appropriate message if the entered number is not valid

Let us start the solution with the creation of header files

// date.h is as below

```
struct date
{
    int dd;int mm;int yy;

};
typedef struct date date_t;
void read(date_t*);
void disp(const date_t*);
int is_month_matching(const date_t *,int);
int is_date_in_order(const date_t *,const date_t *);
```

// event.h as below

```
#include"date.h"
struct event
{
    char event_name[10];
    date_t event_date;
};
typedef struct event event_t;
void read_event(event_t*);
void display_event(const event_t*);
int is_event_in_month(const event_t*,int);
int is_event_in_order(const event_t* lhs,const event_t* rhs);
```

// event_array.h as below

```
#include"event.h"
void read_event_array(event_t*,int);
void display_event_array(const event_t*,int);
int count_of_events_in_month(const event_t[],int,int);
```

Consider the client code as below:

// event_client.h

```
#include<stdio.h>
#include"event_array.h"
int main()
{
    event_t e[1000];
    printf("how many events you want to create\n");
    int n;
    scanf("%d",&n);
    printf("enter event details\n");
    read_event_array(e,n);
    printf("Details of events entered by the user are here\n");
    display_event_array(e,n);
    int mon;

    printf("If you want to know the count of events in paarticular month,enter the month
number between 1 and 12\n");
    scanf("%d",&mon);
    if (mon<1 || mon>12)
            printf("this month number is invalid\n");
    else
            printf("number of events in month number %d are
%d\n",mon,count_of_events_in_month(e,n,mon));

    int i,j;
    printf("enter two event numbers to check whether they are in order\n");
    scanf("%d %d",&i,&j);
    if(i>=0 && i<n && j>=0 && j<n)
    {
            if(is_event_in_order(&e[i],&e[j]))
                    printf("in order\n");
            else
```

```
                        printf("Not in order\n");
        }
        else
        {
                printf("entered event number is not valid");
        }
        printf("THANK YOU\n");
        return 0;
}
```

Note: Compile the client code and keep the object file ready. Do not touch this again and again.

Let us start with adding the implementations for all the functions available in header files.

//date.c

```
      #include<stdio.h>
      #include"date.h"   // observe why it is added here?
      void read(date_t* d)
      {
        scanf("%d %d %d",&(d->dd),&(d->mm),&(d->yy));
      }
      void disp(const date_t* d)
      {
        printf("%d/%d/%d\n",d->dd,d->mm,d->yy);
      }
      int is_month_matching(const date_t *d,int month)
      {
        return (d->mm == month);
      }
      int is_date_in_order(const date_t *lhs,const date_t *rhs)
      {
        return lhs->dd < rhs->dd;
      }
```

// event.c

```
#include<stdio.h>
#include"event.h"            // observe this
void read_event(event_t* e)
{
    printf("enter the name of the event\n");
    scanf("%s",e->event_name);
    printf("enter the date\n");
    read(&(e->event_date));
}
void display_event(const event_t* e)
{
    printf("%s\t",e->event_name);
    disp(&(e->event_date));
}
int is_event_in_month(const event_t* e,int month)
{
    return is_month_matching(&(e->event_date),month);
}
int is_event_in_order(const event_t* e1,const event_t* e2)
{
    return is_date_in_order(&(e1->event_date),&(e2->event_date));
}
```

//event_array.c

```
#include"event_array.h"
void read_event_array(event_t *e,int n)
{   int i;
    for(i = 0; i< n; i++)
    {
            read_event(&(e[i]));  // e+i is pointer notation
    }
}
```

```
void display_event_array(const event_t *e,int n )
{
    int i;
    for(i = 0; i< n; i++)
    {
            display_event(&(e[i]));    // e+i
    }
}
int count_of_events_in_month(const event_t *e,int n, int month)
{
    int i;
    int count = 0;
    for(i = 0; i< n; i++)
    {
            if(is_event_in_month(&(e[i]),month)) count++;
    }
    return count;
}
```

Note: Compile each of the above source files separately. And then link 4 object files to get the executable.

```
gcc -c date.c
gcc -c event.c
gcc -c event_array.c
gcc event_client.o event.o event_array.o date.o -o event.exe  // renaming the executable. Not a requirement though
```

Then run event.exe to get the solution

As there are multiple files involved in the solution for the problem, it is good to write make file and use make command so that the developer need not remember every now and then, to which file changes are made.

//**event.mk** is as below

```
event.exe : event_client.o event.o event_array.o date.o
    gcc event_client.o event.o event_array.o date.o -o event.exe
event_client.o: event_client.c event_array.h
    gcc -c event_client.c
event.o:event.c event.h
    gcc -c event.c
event_array.o : event_array.c event_array.h
    gcc -c event_array.c
date.o: date.c date.h
    gcc -c date.c
```

Execution:

- **In windows OS**, **mingw32-make –f event.mk** and press enter. Then run the executable **event.exe.**
- **In ubuntu OS**, small changes must be done in the make file. First two lines in the make file are as below. Others remain the same

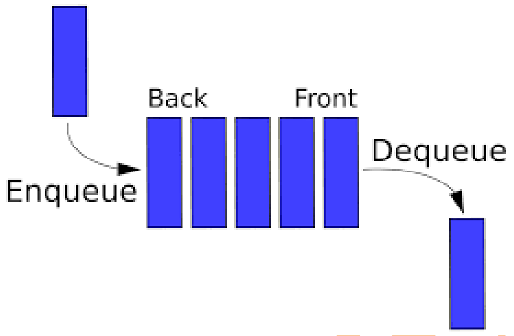
```
event.out : event_client.o event.o event_array.o date.o
        gcc event_client.o event.o event_array.o date.o -o event.out
```

Command used to run the make file: **make –f event.mk** and press enter. Then run the event.out using **./event.out**

Priority Queue Implementation

Queue:

A line or a sequence of people or vehicles waiting for their turn to be attended or to proceed. In computer Science, **a list of data items, commands, etc., stored so as to be retrievable in a definite order. A Non- Linear data structure which has 2 ends - Rear end and a Front end**. Data elements are inserted into the queue from the rear(back) end and deleted from the front end. Hence a queue is also known as **First In First Out (FIFO)** data structure.



Two major Operations on a queue:

- enqueue() − add (store) an item to the queue from rear end.
- dequeue() − remove (access) an item from the queue from front end.

Others operations may include,

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

Types of queue:

- **Ordinary queue** - insertion takes place at rear and deletion takes place at front.
- **Priority queue** - special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue
- **Circular queue** - Last element points to first element of queue making circular link.
- **Double ended queue** - insertion and removal of elements can be performed from either from the front or rear.

Priority Queue

A data structure which is like a regular **queue** but where additionally each element has a "**priority**" associated with it. This priority decides about the deque operation. The enqueue operation stores the item and the priority information.

Types of Priority Queue:

- **Ascending priority queue:**
  - The smallest number, the highest priority.
- **Descending priority queue:**
  - The highest number, the highest priority.

Applications of Priority Queue:

1. Dijkstra's Algorithm
2. Prims Algorithm
3. Data Compression
4. Artificial Intelligence
5. Heap Sort

Priority Queue can be implemented using different ways.

- Using an Unordered Array
- Using an Ordered Array
- Using an Unordered Linked list
- Using an Ordered Linked list
- Using Heap

Let us consider the implementation of Descending Priority Queue using an Unordered Linked list.

We will clearly separate the interface and implementation.

The component is a type that contains details and priority. The node contains the component.

Priority Queue contains a head which is a pointer to a node.

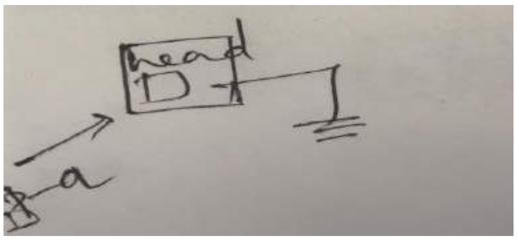
priority.h contains the below code.

```
struct Component
{
    char details[20];
    int priority;
};
typedef struct Component compo_t;
struct node
{
        compo_t c;
        struct node *link;
};
typedef struct node node_t;
struct prio_queue
{
        node_t *head;
};
typedef  struct  prio_queue  prio_t;
void init_queue(prio_t *p_queue);
void  enque(prio_t*  p_q,compo_t  c);
void deque(prio_t* p_q);
void disp(const prio_t* p_q);
```

priority_client.c contains the below code.

```
#include<stdio.h>
#include “priority.h”
int main()
{
    prio_t  queue;
    init_queue(&queue);
    compo_t c;
    printf("enter ua choice\n1.insert\n2.delete\n3.disp\n4.exit\n");
    int choice;
    scanf("%d",&choice);
        do
        {
            switch(choice)
                {
                    case 1: printf("enter detail and priority");
                            scanf("%s%d",c.details,&c.priority);
                            enque(&queue,&c);
                            break;
                    case 2: deque(&queue); break;
                    case 3: disp(&queue); break;
                    default: exit(0);  // to terminate the program
                }
            printf("choice\n1.insert\n2.delete\n3.disp\n4.exit\n)
        scanf("%d",&choice);
}while(choice<4);
return 0;
}
```

The client creates a priority queue and initializes it.



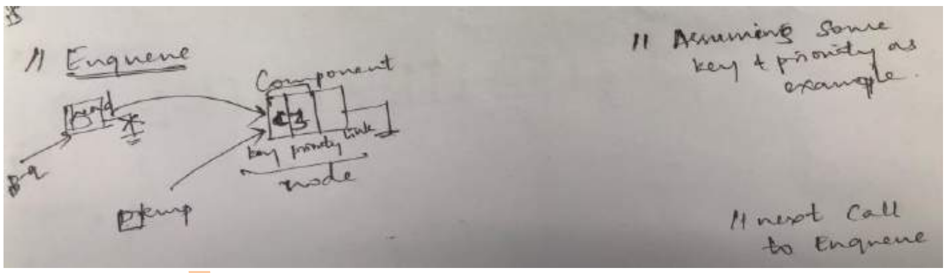
Then based on the choice entered by the user, enqueue, dequeue and display happens. When the choice is 1, the user is allowed to give defined values for component structure: details and priority. The en queue takes priority queue and component as arguments. When the choice is 2, de queue takes priority queue and stores the component to be deleted based on priority in a local variable and frees the node to be deleted. When choice is 3, display is called which is self-explanatory. These are repeated in a looping structure.

Now the source file **priority.c** is as below.

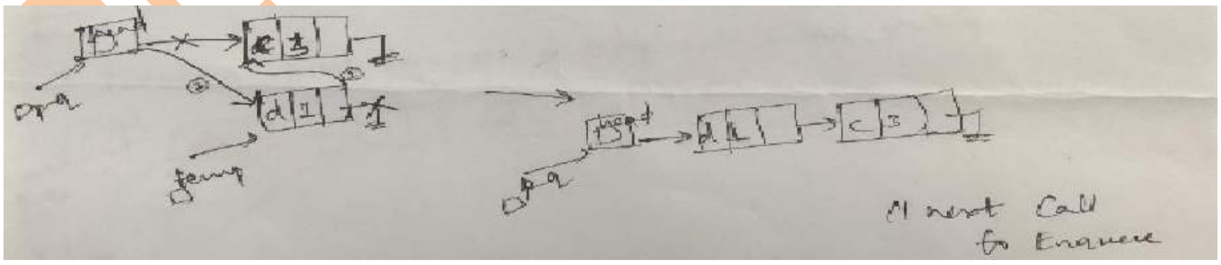
```
void init_queue(prio_t *p_queue) // Used to initialize the priority queue
{
        p_queue -> head = NULL;
}
```

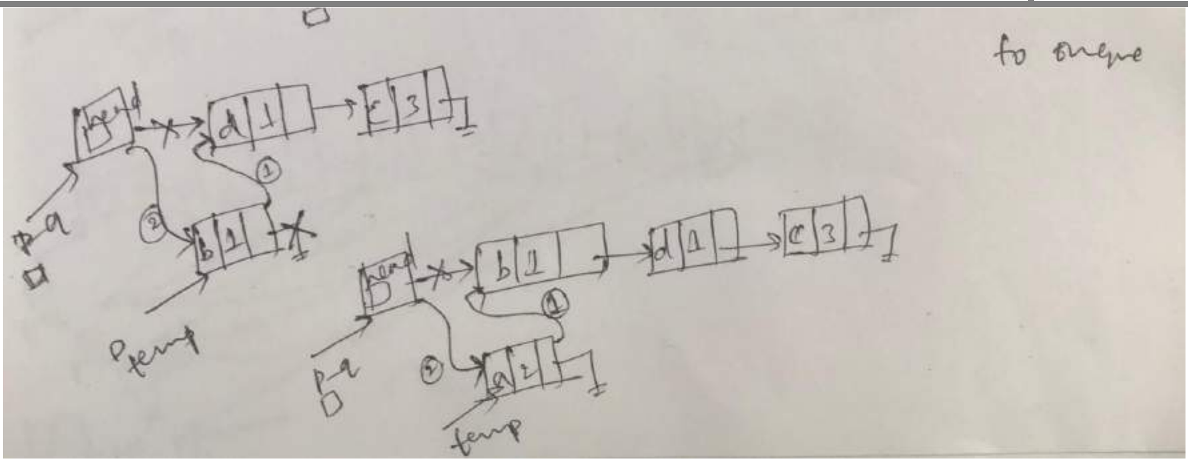
void enque(prio_t* p_q,compo_t* com) // This function creates a node using the component from the client and adds this node in the beginning of the queue.

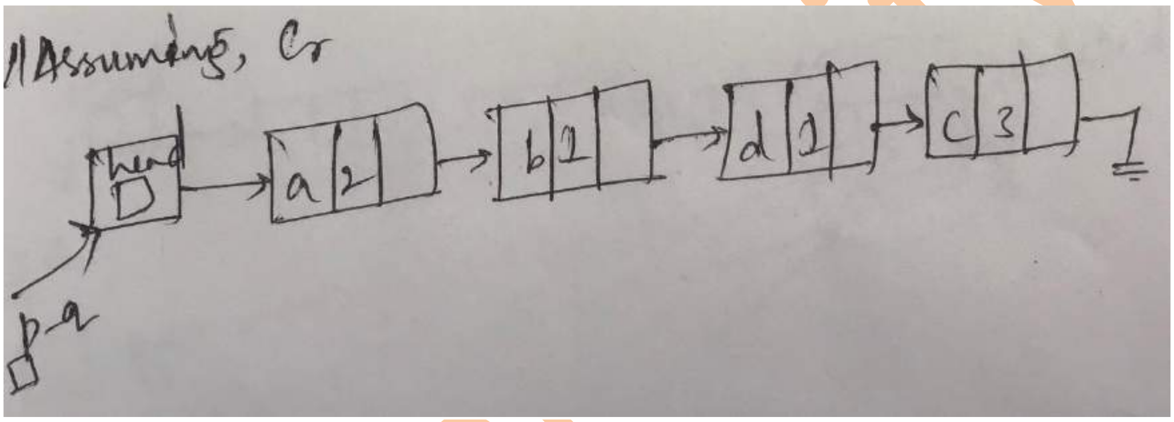
```
{
    node_t   *temp   =   (node_t*)malloc(sizeof(node_t));
    strcpy(temp ->c.details,com->details);

    temp->c.priority = com->priority;

    //temp->link = NULL;

    temp->link = p_q ->head;

    p_q ->head = temp;

}
```



// key in diagram is same as details

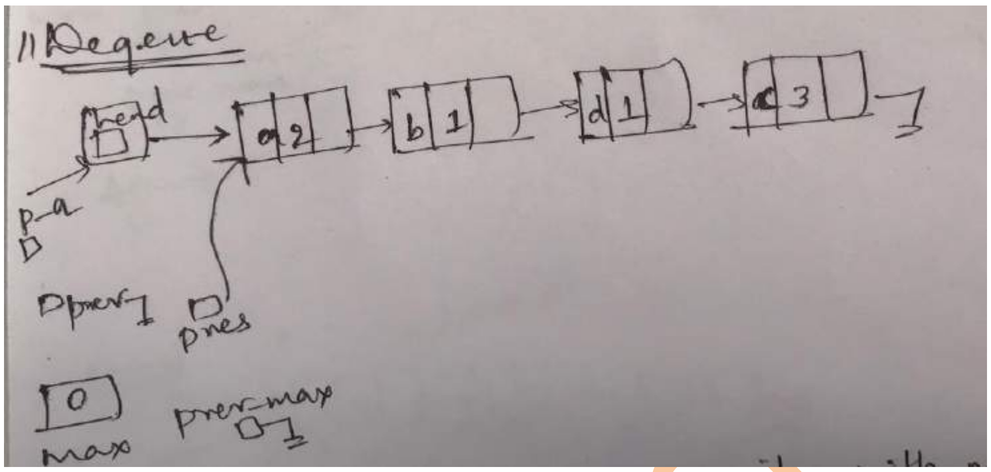


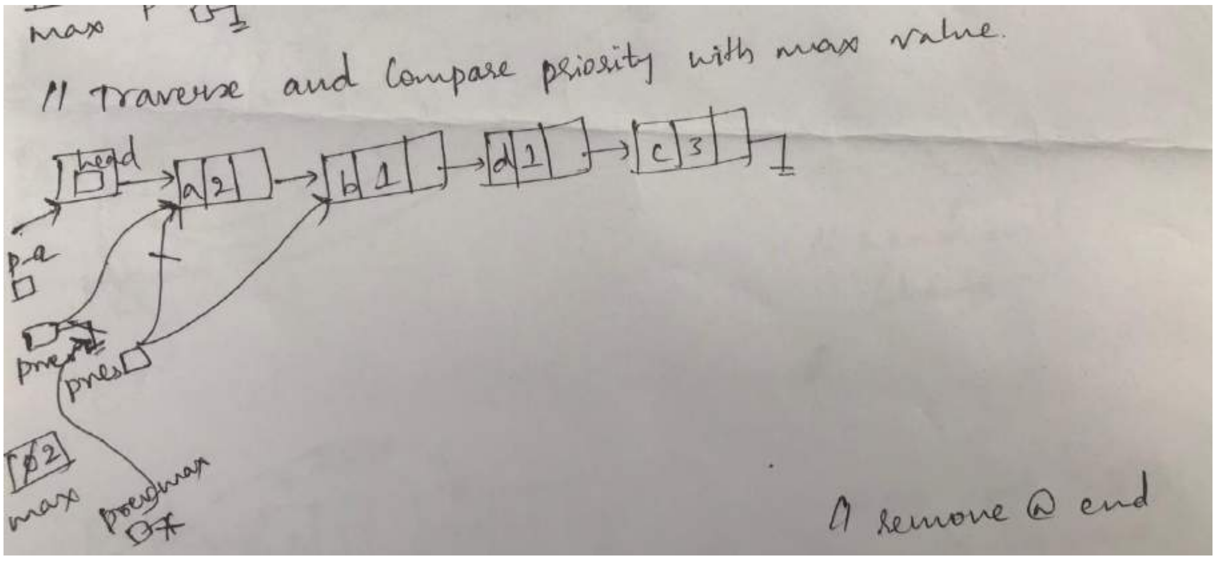


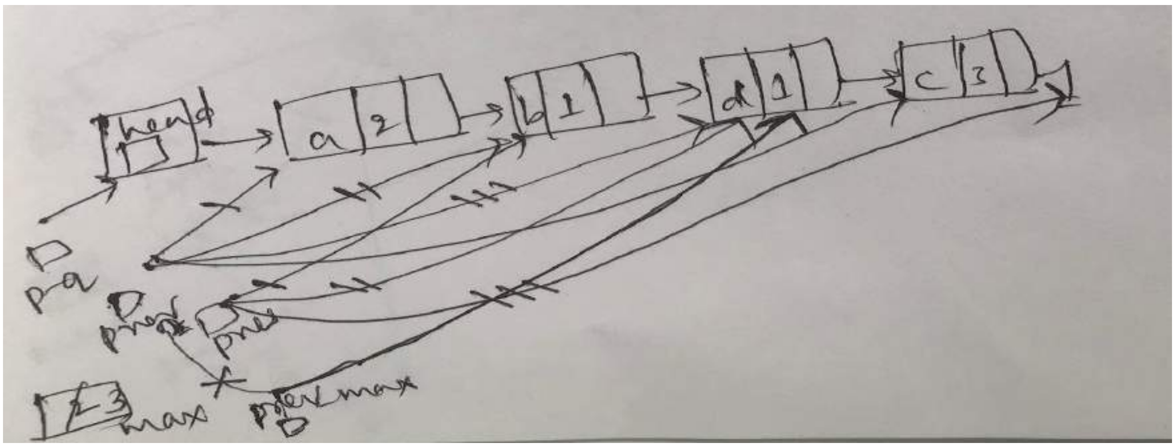


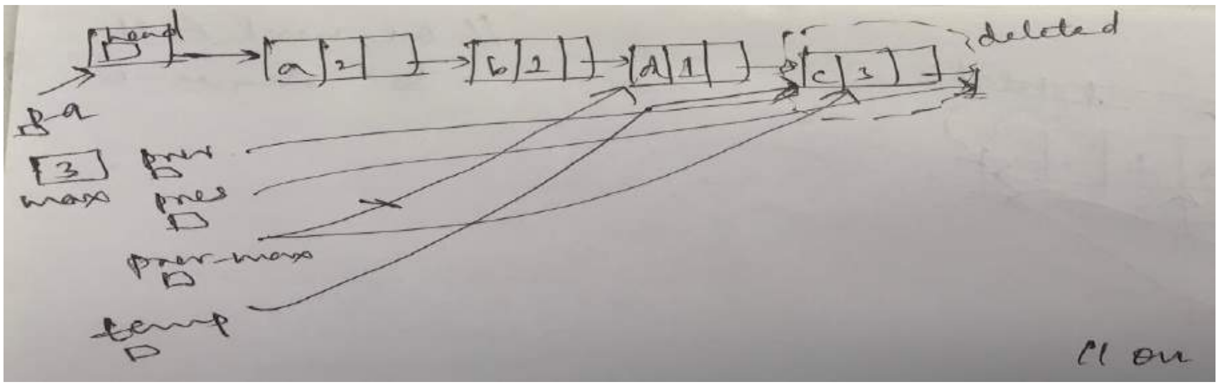
// After 4 times enqueue, Final list looks like above.

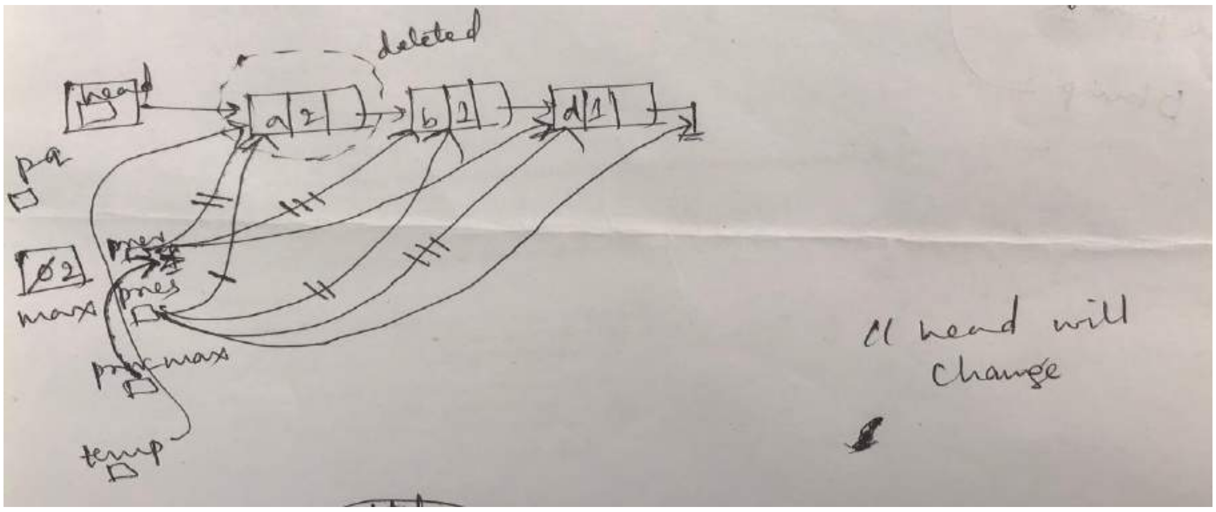
```
void deque(prio_t* p_q) // used to delete the node from the queue based on the priority
    {
        if(p_q->head == NULL)
            printf("No elements in the Q to delete\n");
        else
        {
            node_t *present = p_q ->head; // present points to wherever head is pointing
            node_t *prev = NULL;
            // prev points to NULL initially. But later one before the present node
            int max = 0;
            node_t *prev_max = NULL; // link field of this points to the node with highest
            priority in the component.
            while(present != NULL)
            {
                    if(present->c.priority >= max)
                    {
                        max = present->c.priority;
                         prev_max = prev;
                    }
                    prev = present;
                    present = present->link;
                }
                compo_t compo;// to store the details of removed node. Just for printing purpose
                if(prev_max != NULL)
                {
```
//Remove in the middle or the end of the list. The head will not change.
```
                  node_t *temp = prev_max->link;
                  prev_max->link = temp->link;
```
//copy the node information to component before deleting the node
```
                  strcpy(compo.details,temp->c.details);
                  compo.priority = temp->c.priority;
                  free(temp);
                }
                else
                {
```
// if the first node has the highest priority, head must change
```
                    node_t  *temp  =  p_q  ->head;
                    p_q->head = p_q ->head>link;
```
// copy the node information to component before deleting the node
```
                    strcpy(compo.details,temp->c.details);
                    compo.priority  =  temp->c.priority;
                    free(temp);
                }
            printf("Deleted component is %s with %d priority\n", compo.details, compo.priority);
            }
    }
```

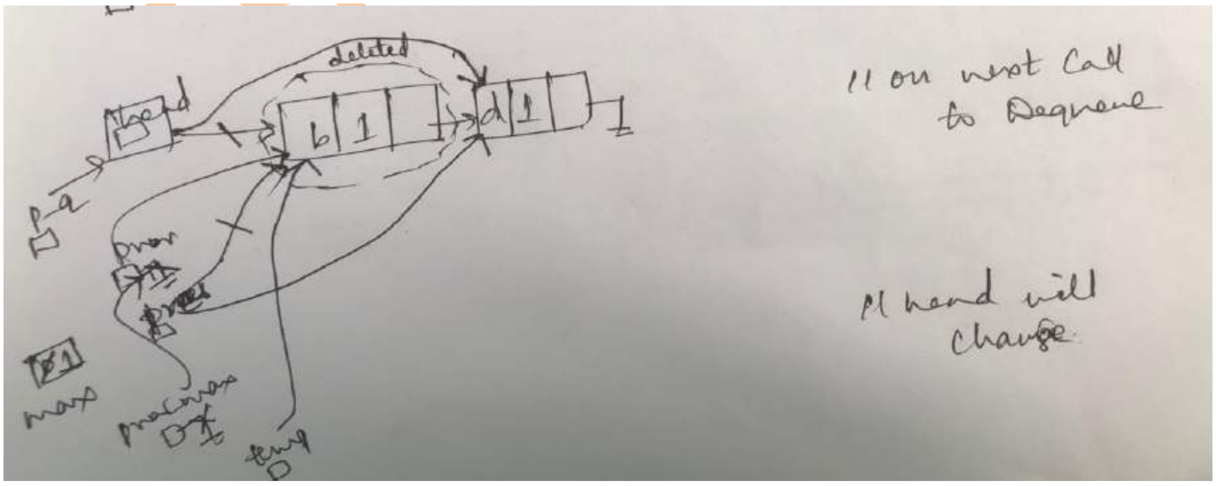

// Dequeue
head
a 2
b 1
d 1
c 3
p_a
prev
pres
0
max
prev_max

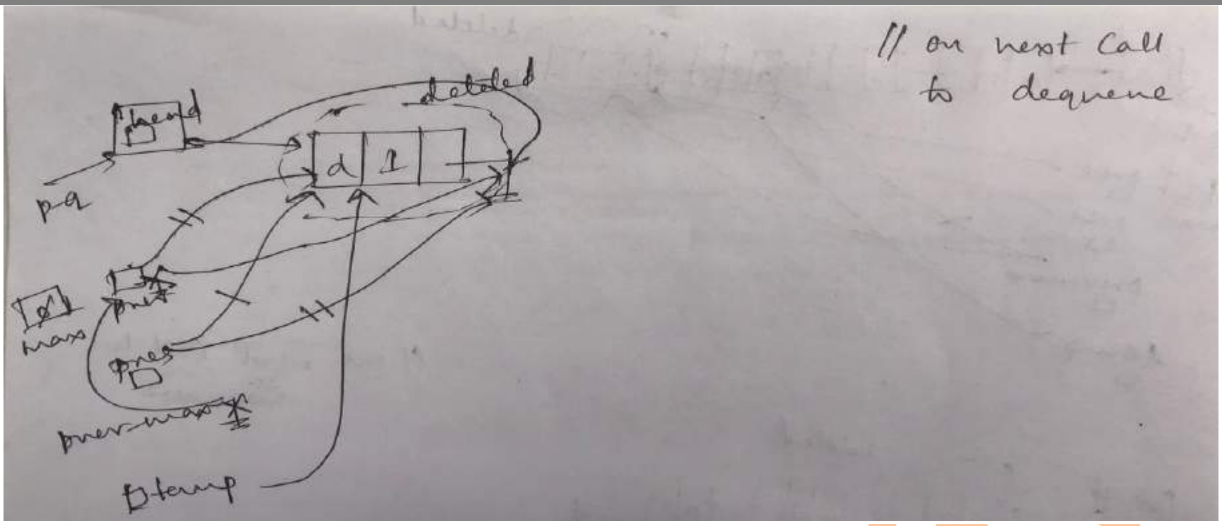

max
// Traverse and compare priority with max value.
head
a 2
b 1
d 1
c 3
p_a
prev
pres
2
max
prev_max
// remove @ end


head
a 2
b 1
d 1
c 3
p_a
prev
pres
3
max
prev_max


head
p-a
3
max
prev
pres
prev-max
temp
a 2
b 1
d 1
c 3
deleted
// on


deleted
head
p-a
a 2
b 1
d 1
prev
pres
max
prev-max
temp
// head will change


deleted
head
b 1
d 1
p-a
prev
pres
max
prevmax
temp
// on next Call to Dequeue
// head will change



void disp(const prio_t* p_q) // **display the nodes, in turn components of the priority queue**

```
{
        node_t *p = p_q ->head;
        if(p == NULL)
        {
                printf("No elements in queue");
        }
        else
        {
                while(p != NULL)
                {
                        printf("%s %d\n",p->c.details,p->c.priority);
                        p = p->link;
                }
        }
}
```

Usage of GNU Debugger

We will be discussing two important errors and how to use GDB to debug these kind of errors in our C Code.

Segmentation Fault

When a program runs, it has access to certain portions of memory. It may have some local variables in every function. These are stored in the Stack. May have some memory allocated during runtime. These are allocated on the Heap. Program can only touch the memory that belongs to it - the memory previously mentioned. Any access outside this area will cause a segmentation fault. Segmentation faults are commonly referred to as segfaults.

- They are caused by a program trying to read or write an illegal memory location. A common condition that causes programs to crash.
- An error returned by hardware with memory protection that tells the Operating System(OS) that memory violation has occurred
- The OS usually reacts by telling the offending process about the error through a signal and then it performs some sort of corrective action
- Example scenarios

    When scanf unable to perform write operation

    Performing write operation on a read only location

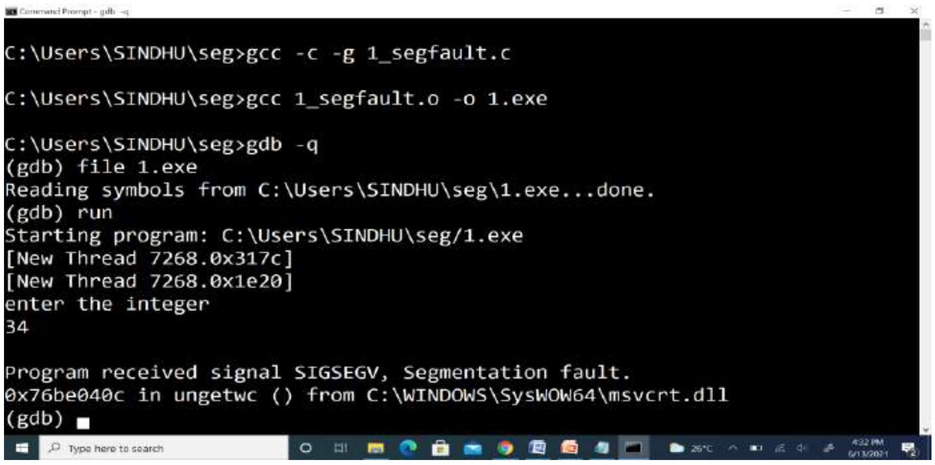
    Performing read or write operation on freed block of memory once it is NULL

Note: compile all the codes with –g option.

Example code 1:

```
#include<stdio.h>
int main()
{
    int x;
    printf("enter the integer\n");
    scanf("%d",x);
    printf("user entered %d",x);
    return 0;
}
```

When you pass the executable of above code to gdb, here is the snapshot. Observe the signal, SIGSEGV

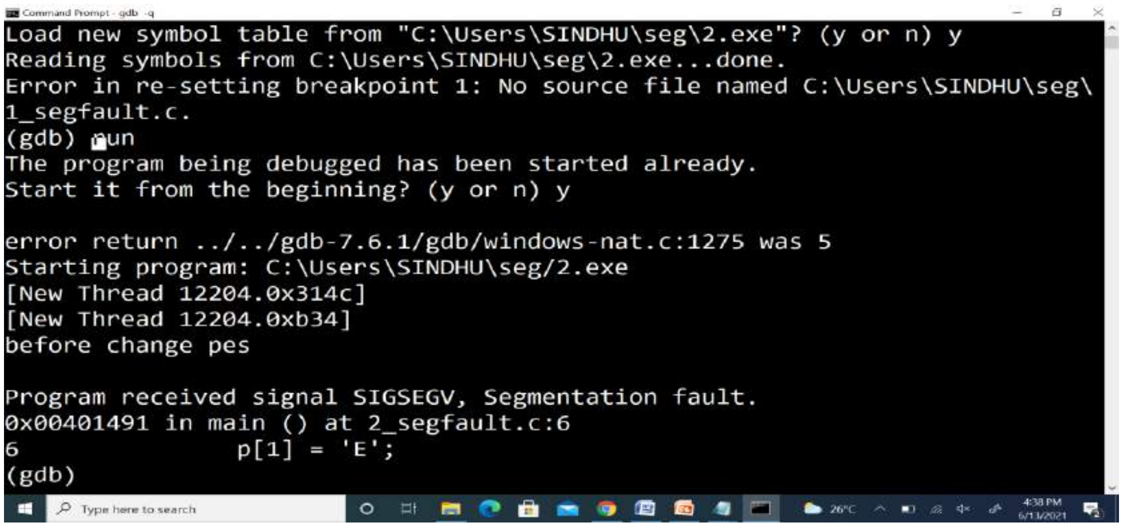


```
C:\Users\SINDHU\seg>gcc -c -g 1_segfault.c

C:\Users\SINDHU\seg>gcc 1_segfault.o -o 1.exe

C:\Users\SINDHU\seg>gdb -q
(gdb) file 1.exe
Reading symbols from C:\Users\SINDHU\seg\1.exe...done.
(gdb) run
Starting program: C:\Users\SINDHU\seg/1.exe
[New Thread 7268.0x317c]
[New Thread 7268.0x1e20]
enter the integer
34

Program received signal SIGSEGV, Segmentation fault.
0x76be040c in ungetwc () from C:\WINDOWS\SysWOW64\msvcrt.dll
(gdb)
```

Example code 2:

```
#include<stdio.h>
int main()
{
    char* p = "pes";
    printf("before change %s\n",p);
    p[1] = 'E';
    printf("after change %s\n",p);
    return 0;
}
```

When you pass the executable of above code to gdb, here is the snapshot. Observe the signal, SIGSEGV.

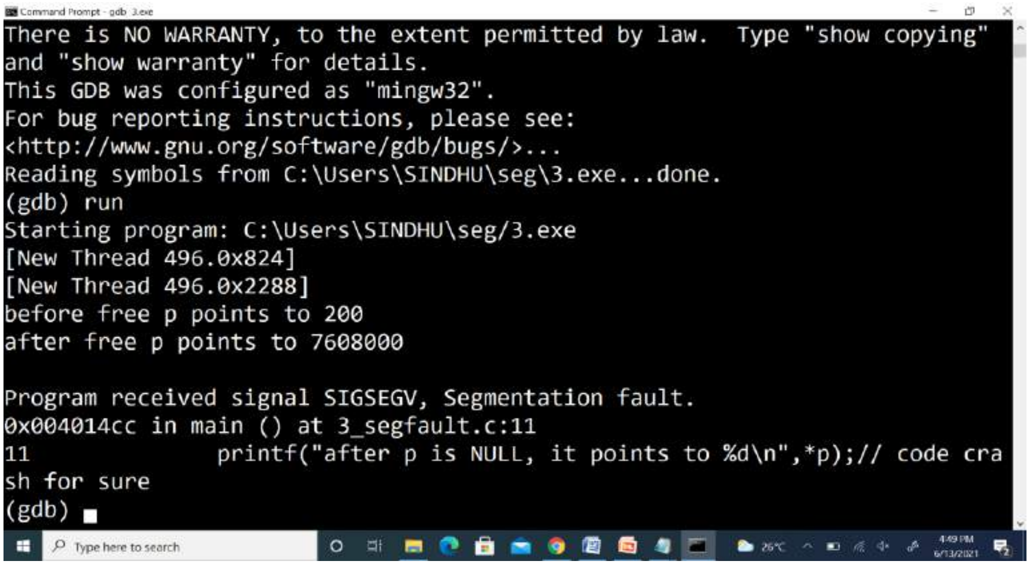


```
Load new symbol table from "C:\Users\SINDHU\seg\2.exe"? (y or n) y
Reading symbols from C:\Users\SINDHU\seg\2.exe...done.
Error in re-setting breakpoint 1: No source file named C:\Users\SINDHU\seg\
1_segfault.c.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

error return ../../gdb-7.6.1/gdb/windows-nat.c:1275 was 5
Starting program: C:\Users\SINDHU\seg/2.exe
[New Thread 12204.0x314c]
[New Thread 12204.0xb34]
before change pes

Program received signal SIGSEGV, Segmentation fault.
0x00401491 in main () at 2_segfault.c:6
6               p[1] = 'E';
(gdb)
```

Example code 3:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int* p = (int*)malloc(sizeof(int));
    *p = 200;
    printf("before free p points to %d\n",*p) ;
    free(p);
    printf("after free p points to %d\n",*p); //undefined behavior. So might crash here too
    p = NULL;
    printf("after p is NULL, it points to %d\n",*p);// code crash for sure
    return 0;
}
```

When you pass the executable of above code to gdb, here is the snapshot. Observe the signal, SIGSEGV.



```
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from C:\Users\SINDHU\seg\3.exe...done.
(gdb) run
Starting program: C:\Users\SINDHU\seg/3.exe
[New Thread 496.0x824]
[New Thread 496.0x2288]
before free p points to 200
after free p points to 7608000

Program received signal SIGSEGV, Segmentation fault.
0x004014cc in main () at 3_segfault.c:11
11              printf("after p is NULL, it points to %d\n",*p);// code cra
sh for sure
(gdb)
```

Memory Leak

- Location with no name and hence no access
- If free is not used on the pointer after allocating memory on the heap, it becomes a garbage if pointer is pointing to a new memory location resulting in memory leak.

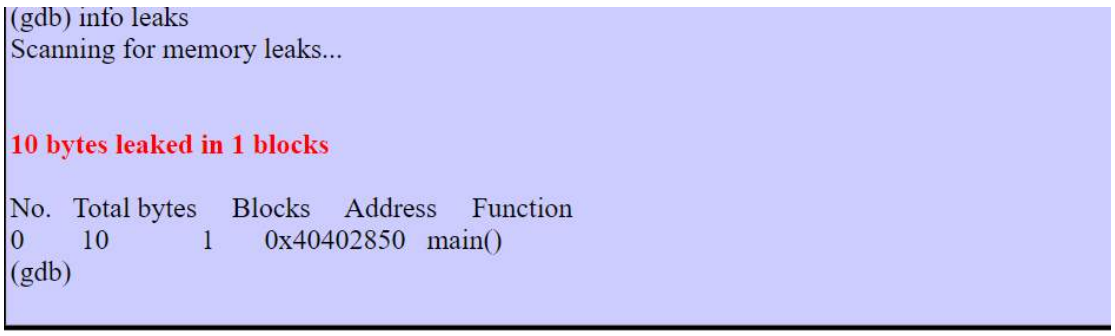
Consider the below code snippet.

```
int* p = (int*)malloc(sizeof(int));
*p = 200;
printf("Initially p is pointing to %d\n",*p);

p = (int*)malloc(sizeof(int));
*p = 300;
printf("Then p is pointing to %d\n",*p);

p = (int*)malloc(sizeof(int));
*p = 400;
printf("Then p is pointing to %d\n",*p);
```

The memory which is allocated on the heap is not freed and the same pointer the made to point to different locations on the heap. This code creates memory leak.

Let us see how GDB finds memory leak in an executable.

- This is not supported by all versions of GDB
- **Valgrind** is used in Linux to locate memory leaks
- **info leaks and info heap** are commands that are available in gdb only on **HP-UX**



```
(gdb) info leaks
Scanning for memory leaks...


10 bytes leaked in 1 blocks

No.  Total bytes    Blocks    Address     Function
0      10            1      0x40402850   main()
(gdb)
```

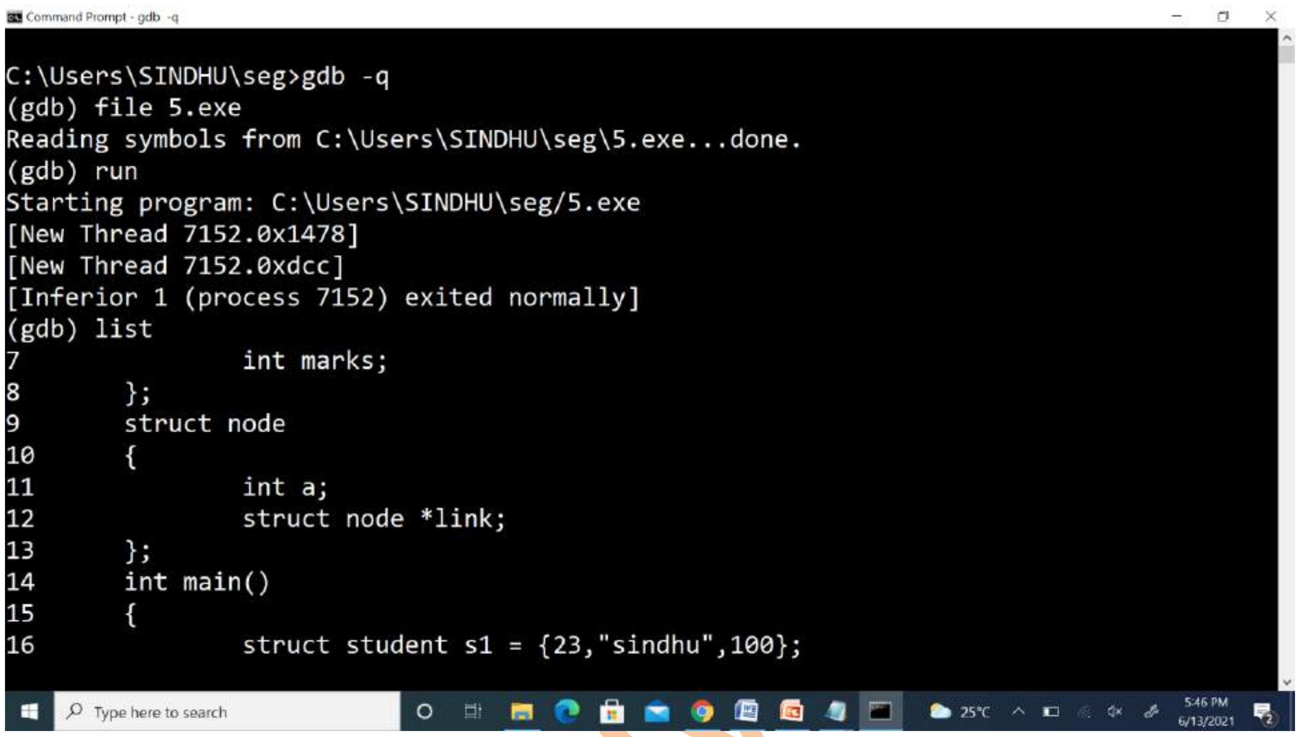
Picture credits: http://www.geocities.ws/kmuthu_gct/memoryleak_gdb.html#2._Memory_leak_in_a_library_shared_

ptype: Prints the description of data type for a given variable. Differs from whatis by printing a detailed description, instead of just the name of the type.

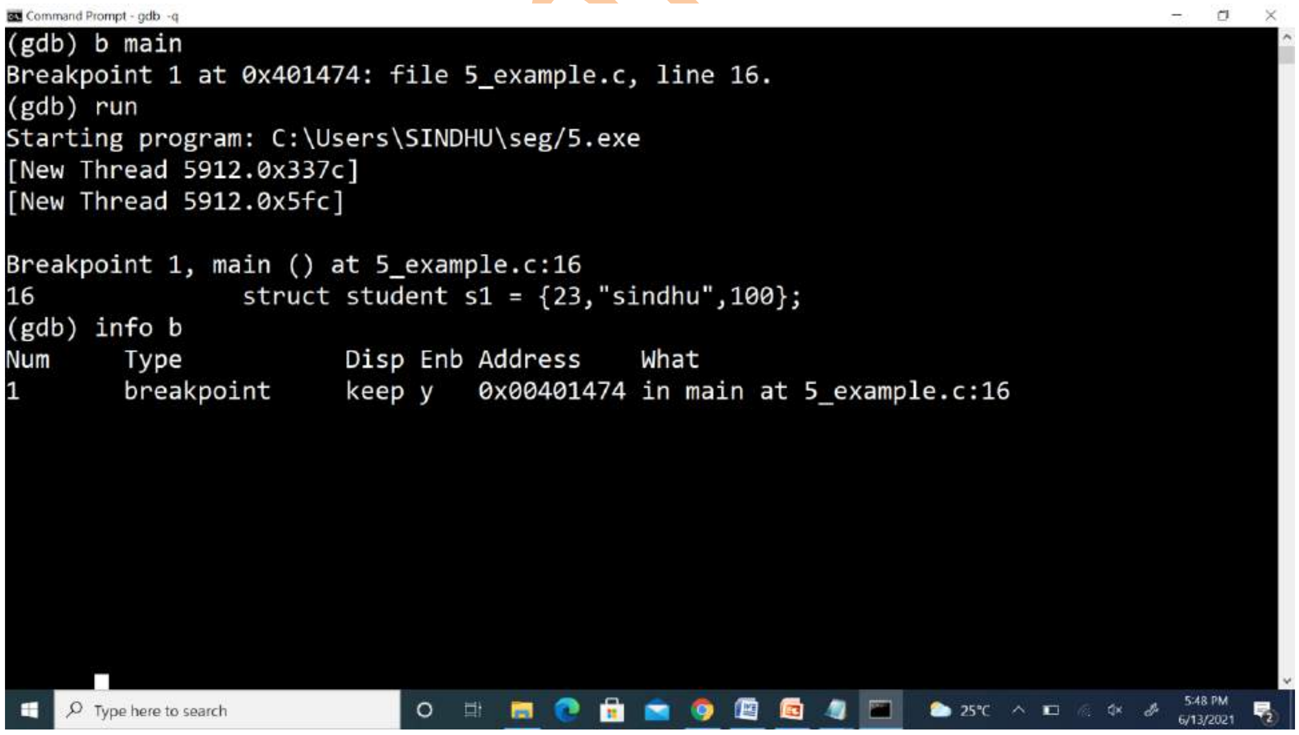
Consider the below code:

```
#include<stdio.h>
#include<stdlib.h>
struct student
{
        int roll_no;
        char name[100];
        int marks;
};
struct node
{
        int a;
        struct node *link;
};
int main()
{
        struct student s1 = {23,"sindhu",100};
        struct student *s2 = &s1;
        struct node* n1 = (struct node*)malloc(sizeof(struct node)) ;
        n1->a = 25;
        n1->link = (struct node*)malloc(sizeof(struct node)) ;
        n1->link->a = 26;
        n1->link->link = NULL;
        return 0;
}
```

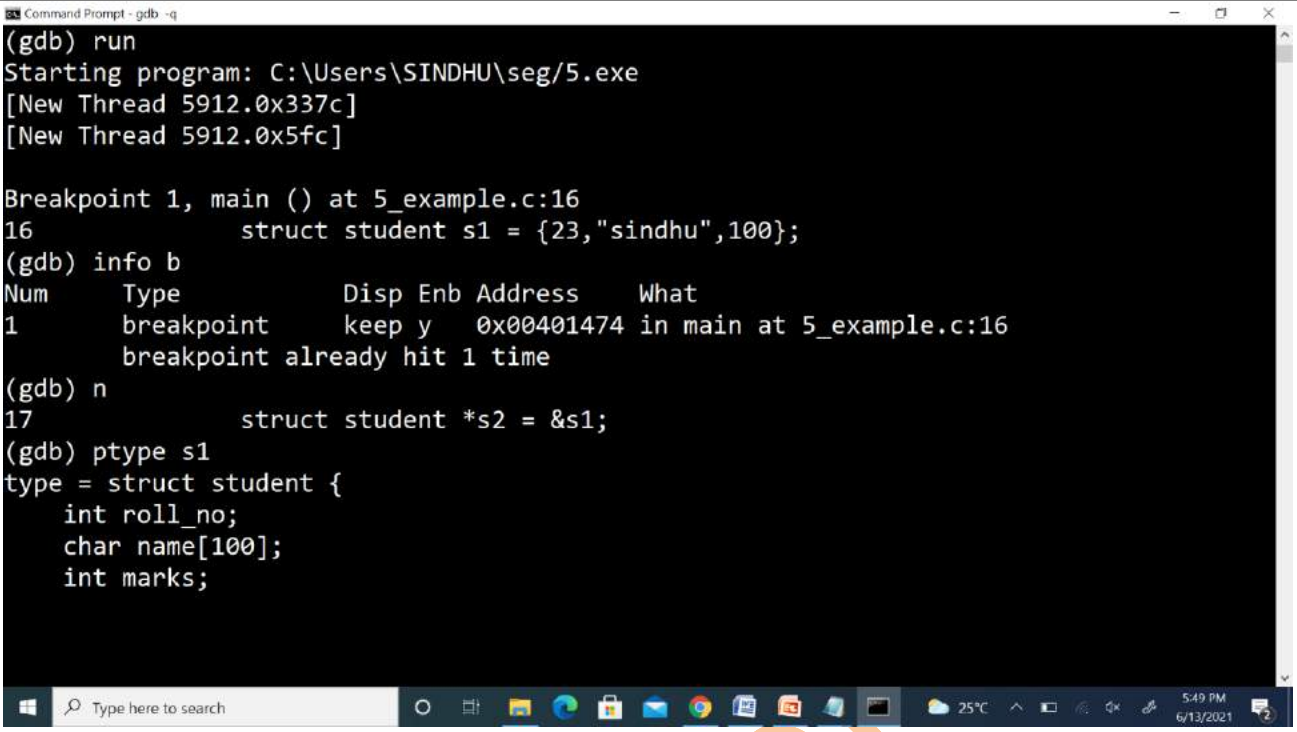
Usage of gdb on the executable of above code is described in below screenshots.



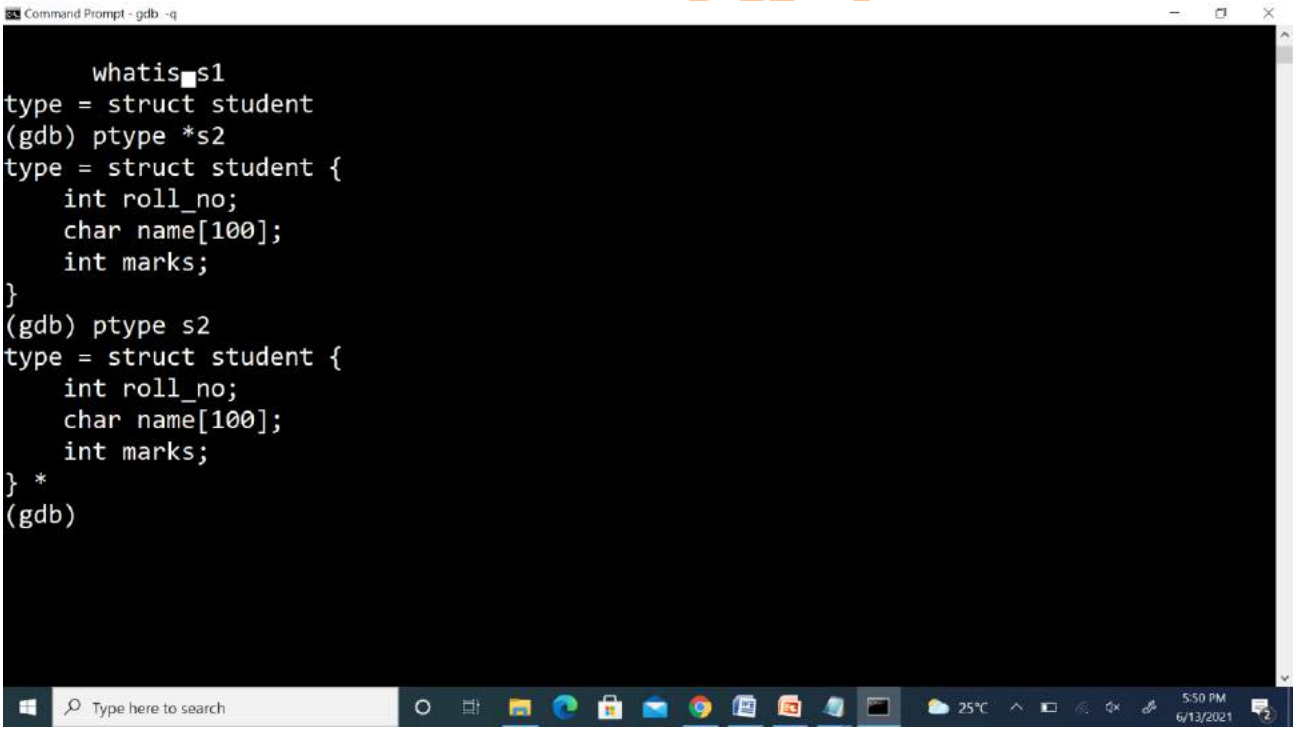
```
C:\Users\SINDHU\seg>gdb -q
(gdb) file 5.exe
Reading symbols from C:\Users\SINDHU\seg\5.exe...done.
(gdb) run
Starting program: C:\Users\SINDHU\seg/5.exe
[New Thread 7152.0x1478]
[New Thread 7152.0xdcc]
[Inferior 1 (process 7152) exited normally]
(gdb) list
7               int marks;
8       };
9       struct node
10      {
11              int a;
12              struct node *link;
13      };
14      int main()
15      {
16              struct student s1 = {23,"sindhu",100};
```



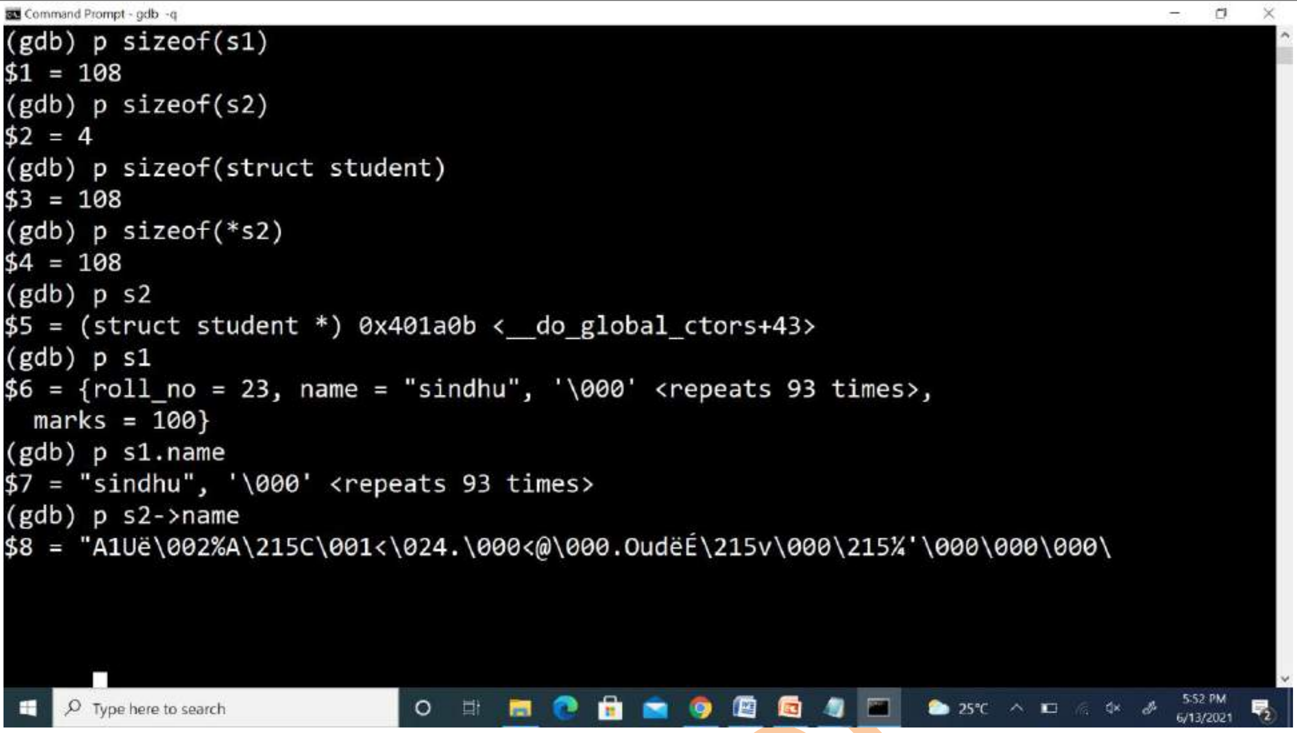
```
(gdb) b main
Breakpoint 1 at 0x401474: file 5_example.c, line 16.
(gdb) run
Starting program: C:\Users\SINDHU\seg/5.exe
[New Thread 5912.0x337c]
[New Thread 5912.0x5fc]

Breakpoint 1, main () at 5_example.c:16
16              struct student s1 = {23,"sindhu",100};
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00401474 in main at 5_example.c:16
```



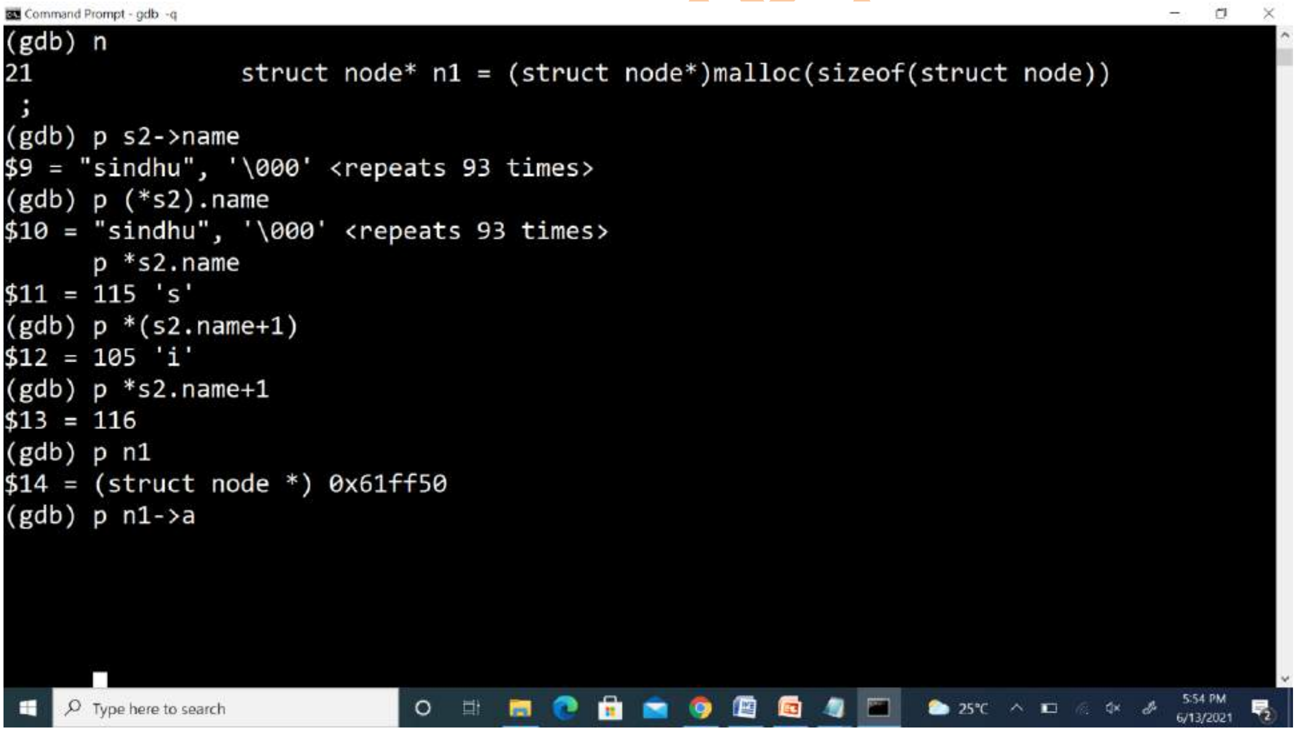
```
(gdb) run
Starting program: C:\Users\SINDHU\seg/5.exe
[New Thread 5912.0x337c]
[New Thread 5912.0x5fc]

Breakpoint 1, main () at 5_example.c:16
16              struct student s1 = {23,"sindhu",100};
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00401474 in main at 5_example.c:16
        breakpoint already hit 1 time
(gdb) n
17              struct student *s2 = &s1;
(gdb) ptype s1
type = struct student {
    int roll_no;
    char name[100];
    int marks;
```



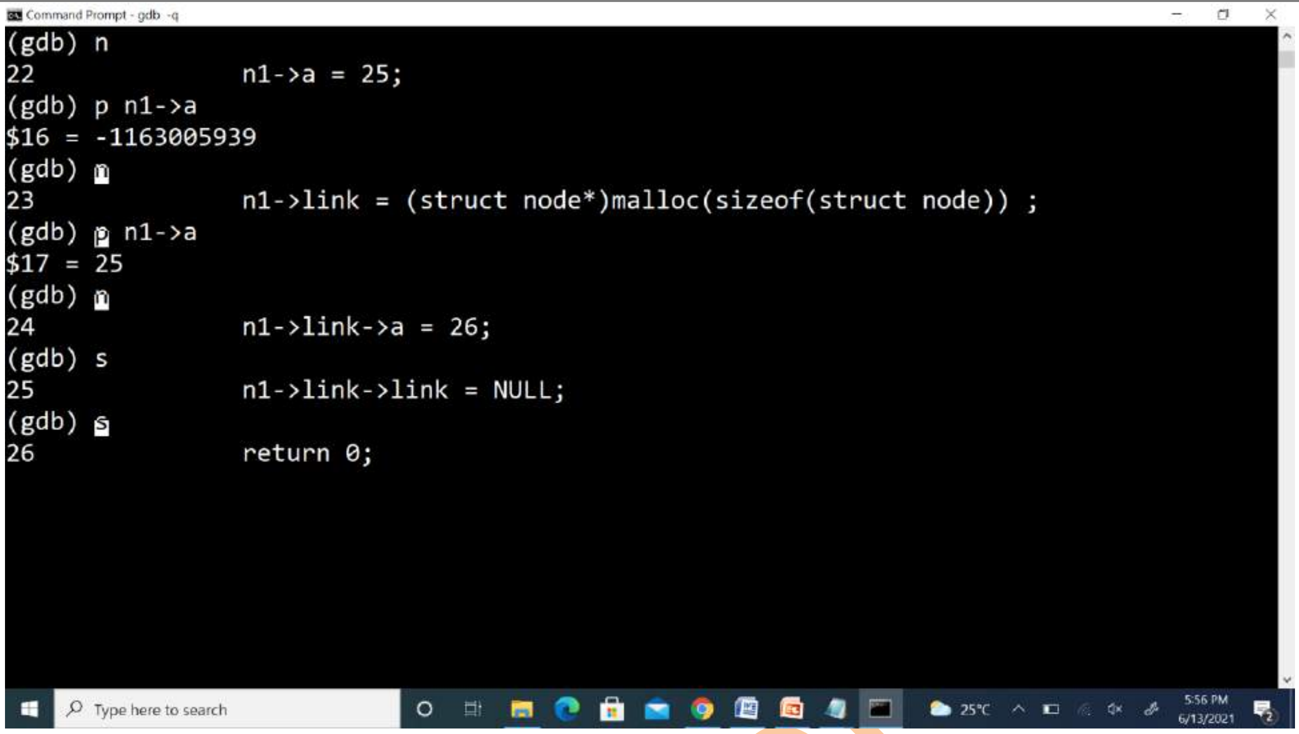
```
      whatis s1
type = struct student
(gdb) ptype *s2
type = struct student {
    int roll_no;
    char name[100];
    int marks;
}
(gdb) ptype s2
type = struct student {
    int roll_no;
    char name[100];
    int marks;
} *
(gdb)
```



```
(gdb) p sizeof(s1)
$1 = 108
(gdb) p sizeof(s2)
$2 = 4
(gdb) p sizeof(struct student)
$3 = 108
(gdb) p sizeof(*s2)
$4 = 108
(gdb) p s2
$5 = (struct student *) 0x401a0b <__do_global_ctors+43>
(gdb) p s1
$6 = {roll_no = 23, name = "sindhu", '\000' <repeats 93 times>,
  marks = 100}
(gdb) p s1.name
$7 = "sindhu", '\000' <repeats 93 times>
(gdb) p s2->name
$8 = "A1Uë\002%A\215C\001<\024.\000<@\000.OudëÉ\215v\000\215¼'\000\000\000\
```



```
(gdb) n
21              struct node* n1 = (struct node*)malloc(sizeof(struct node))
 ;
(gdb) p s2->name
$9 = "sindhu", '\000' <repeats 93 times>
(gdb) p (*s2).name
$10 = "sindhu", '\000' <repeats 93 times>
      p *s2.name
$11 = 115 's'
(gdb) p *(s2.name+1)
$12 = 105 'i'
(gdb) p *s2.name+1
$13 = 116
(gdb) p n1
$14 = (struct node *) 0x61ff50
(gdb) p n1->a
```



```
(gdb) n
22              n1->a = 25;
(gdb) p n1->a
$16 = -1163005939
(gdb) n
23              n1->link = (struct node*)malloc(sizeof(struct node)) ;
(gdb) p n1->a
$17 = 25
(gdb) n
24              n1->link->a = 26;
(gdb) s
25              n1->link->link = NULL;
(gdb) s
26              return 0;
```