Hello everyone, Welcome to Computer Programming course. Our topic for today is Structures. Before we delve into our topic for the day, let us look at Arrays. Shall we? Because having an idea about Arrays is essential for us to understand Structures.

What are arrays?

We know that an Array is a way of organising data into consecutive memory locations, provided that all the data are of the same data type. From the definition of the array we can also understand that an Array is not suitable for storing data belonging to different data types.

Now, would such a situation arise where we require multiple data types which have to be grouped together?

Let us look at the following two scenarios:

- Keep a record of employees in an organization
- Keep a record of students in a school

An organisation will have hundreds of employees and each employee will have:

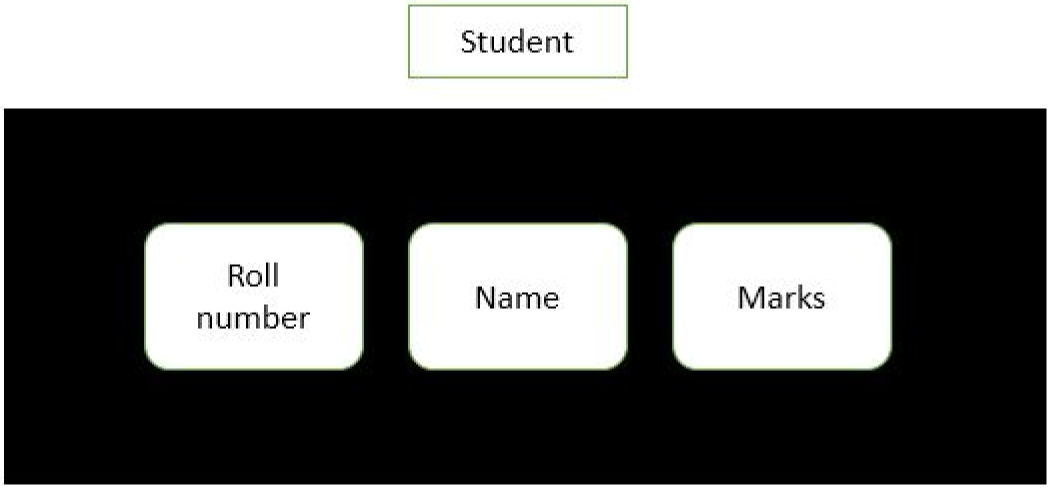
- Name
- Employee ID
- Salary etc

So how will we use array to store all such details? It is possible, but rather a cumbersome process.

Hence C provides us with Structures.

A Structure in C, is a method to organize related data under one name. Unlike arrays, in Structure we can have multiple data types.

You can visualize a structure like the picture given below:



As you can see, a structure under the name 'Student' having a roll number belonging to the integer data type, Name belonging to character array and Marks belonging to the float data type.

How can we write the code to declare a Structure ?

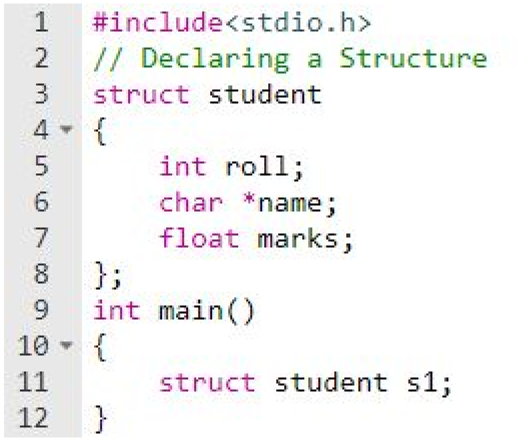
As you know with anything in programming, there is a syntax which needs to be followed.

```
struct student
{
    int roll;
    char *name;
    float marks;
};
```

In the code given above, the name of the structure is 'student'.

1. We need to remember that the name of the structure has to be preceded by the 'struct' keyword.
2. Also please note the opening braces and closing braces, enclosed within are the variables of different data types. These variables which are declared within the structure are called as <u>structure members.</u>
3. Quite often people tend to forget the semicolon after the closing brace. Please make sure that you type in the semicolon at the end

Let us look at a sample program involving structure:

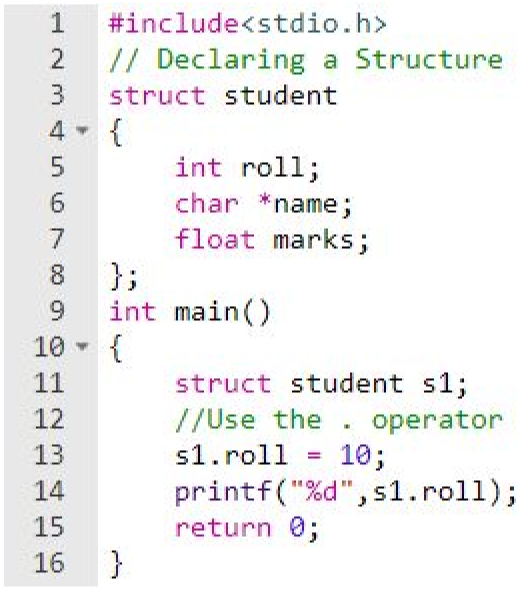


```
1   #include<stdio.h>
2   // Declaring a Structure
3   struct student
4   {
5       int roll;
6       char *name;
7       float marks;
8   };
9   int main()
10  {
11      struct student s1;
12  }
```

What do you observe?

- Our student structure is declared outside the main function. It is not necessary that you have to write a structure outside the main function. You can also write it within the main function. It has to do with scope, which i am sure you will have studied in your earlier classes.
- Now have a look within the main function, in line number 11 - Here <u>s1</u> is called as a <u>structure variable</u>. Study the syntax for declaring a structure variable.
  - [struct keyword] [name of the structure] [name of the structure variable]
- We can think about the structure variable as a representation of the structure ‘student’. It is using the structure variable that we are going to access members of the structure as well as assign values to them. Thus we can use s1, to access roll, name and marks of the structure ‘student’.

How do we initialize/ give values to the structure members (roll, marks etc)



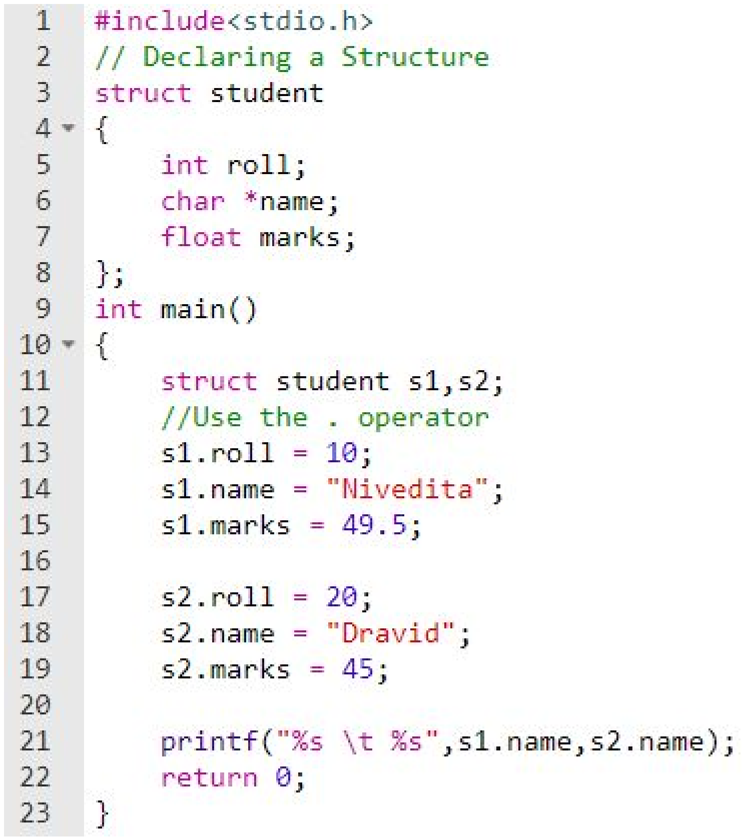
```c
1  #include<stdio.h>
2  // Declaring a Structure
3  struct student
4  {
5      int roll;
6      char *name;
7      float marks;
8  };
9  int main()
10 {
11     struct student s1;
12     //Use the . operator
13     s1.roll = 10;
14     printf("%d",s1.roll);
15     return 0;
16 }
```

To initialize the structure members, C provides us with a <u>structure variable</u> and the <u>DOT (.) operator</u>.

Study line number 13 and 14.

We can use the structure variable alongside the DOT operator to access the members of the structure and give values to them.

Assume a scenario where you have to enter the details of two students. How will the program given above be edited for the purpose?

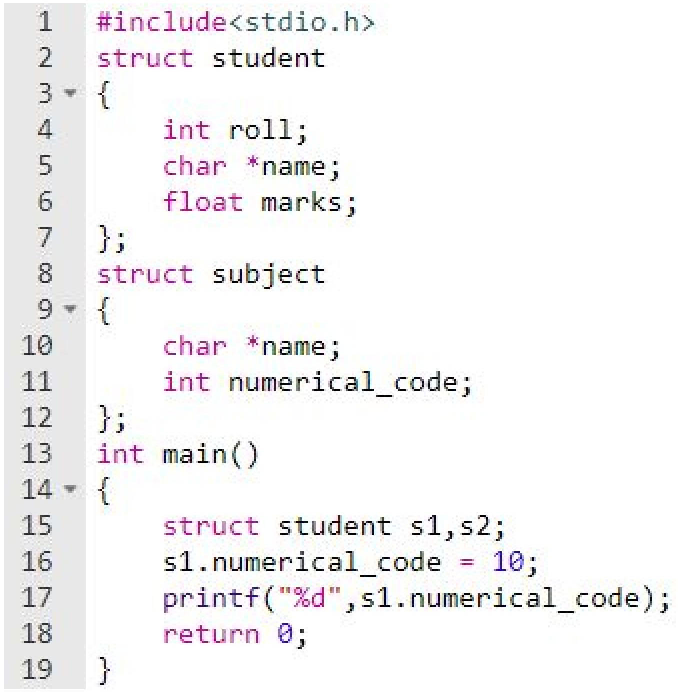


```c
1  #include<stdio.h>
2  // Declaring a Structure
3  struct student
4  {
5      int roll;
6      char *name;
7      float marks;
8  };
9  int main()
10 {
11     struct student s1,s2;
12     //Use the . operator
13     s1.roll = 10;
14     s1.name = "Nivedita";
15     s1.marks = 49.5;
16 
17     s2.roll = 20;
18     s2.name = "Dravid";
19     s2.marks = 45;
20 
21     printf("%s \t %s",s1.name,s2.name);
22     return 0;
23 }
```

We know that s1 represents the structure student. But s1 can only be used to access the records of one student. If there is another student whose records are required, then we have to use another structure variable.

Please look at line number 11. s1 and s2 are two structure which can be used to access the structure members.

Now let us go for a Find the Output type of program



```c
#include<stdio.h>
struct student
{
    int roll;
    char *name;
    float marks;
};
struct subject
{
    char *name;
    int numerical_code;
};
int main()
{
    struct student s1,s2;
    s1.numerical_code = 10;
    printf("%d",s1.numerical_code);
    return 0;
}
```

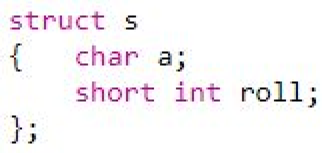
In a real life scenario, is it possible to include all the information in one structure. Think about a school and if you are to make a C program that had to read and retrieve the information for any school related purpose, would you have only one structure? You will need to have multiple structures right? Study the program on the left ? Can you find any incorrect line of code before you run the program?

<u>Write down the reasons (if any line of code is incorrect)</u>

<u>How will you modify the code to access the 'numerical_code' structure member?</u>

How much memory does a structure occupy ?

It is very much possible that some amongst you will be surprised that the memory of a structure does not always take up space that we had calculated it would. Sometimes, depending upon the compiler, a structure takes up more memory space than what we had thought it would consume. This is because of Data Alignment and Padding. Normally memory is stored in the computer in such a way that the data can be retrieved using a minimum of data fetch cyles as possible. To reduce the data fetch cycles, the data stored needs to be aligned. For example: a char data type typically occupies 1 byte of memory. Let us assume that the memory for the structure begins with the address 0x2000. Let the structure be defined as follows:



```c
struct s
{   char a;
    short int roll;
};
```

If char occupied 1 byte at 0x2000, then it would be mean that short int which typically requires 2 bytes of memory would have to be stored at an odd numbered memory location 0x2001, to avoid this, the compiler adds a padding byte after 0x2000 so that short int is stored in a memory location beginning with a multiple of 2.

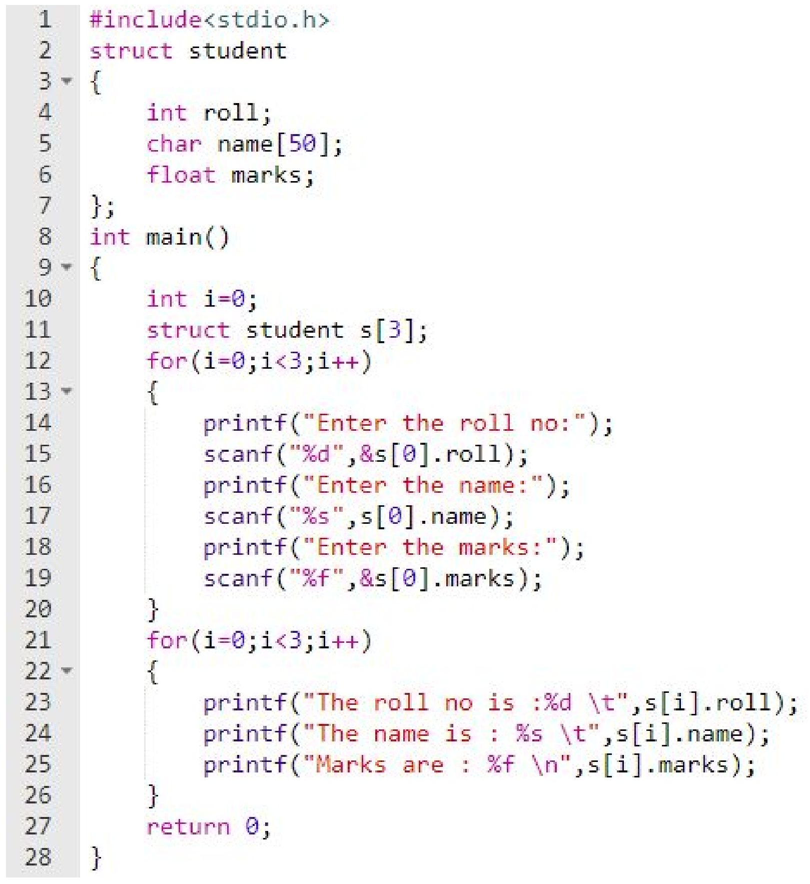
Hence, while we cannot exactly know the memory occupied by a structure, we can be sure that the minimum memory occupied by the structure is equal to the sum of memory occupied by each data type in the structure.

Array of structure variables

If we are to enter the details of two or three students, then what are our options?

We can declare three structure variables and access the members of the structure.

But, what if we are asked to enter the details of 100 students. Would it be wise to declare 100 structure variables ? Perhaps not. Hence, C provides with an option of using Array of Structure (Array of structure variables). We will learn more once we write a sample code of the same.



```
 1  #include<stdio.h>
 2  struct student
 3  {
 4      int roll;
 5      char name[50];
 6      float marks;
 7  };
 8  int main()
 9  {
10      int i=0;
11      struct student s[3];
12      for(i=0;i<3;i++)
13      {
14          printf("Enter the roll no:");
15          scanf("%d",&s[0].roll);
16          printf("Enter the name:");
17          scanf("%s",s[0].name);
18          printf("Enter the marks:");
19          scanf("%f",&s[0].marks);
20      }
21      for(i=0;i<3;i++)
22      {
23          printf("The roll no is :%d \t",s[i].roll);
24          printf("The name is : %s \t",s[i].name);
25          printf("Marks are : %f \n",s[i].marks);
26      }
27      return 0;
28  }
```

In line number 11, we can see the declaration of an array of structures (structure variables).

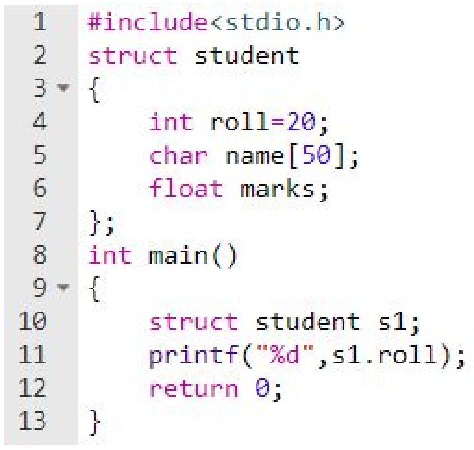
The syntax is more or less the same as that for declaring other structure variables.

Use the for loop to iterate through each structure variable and read the input (line numbers 12-20)

Use the for loop to iterate through each structure variable to display the value of each member of each structure variable (line numbers 21 - 26)

Pointer to a structure

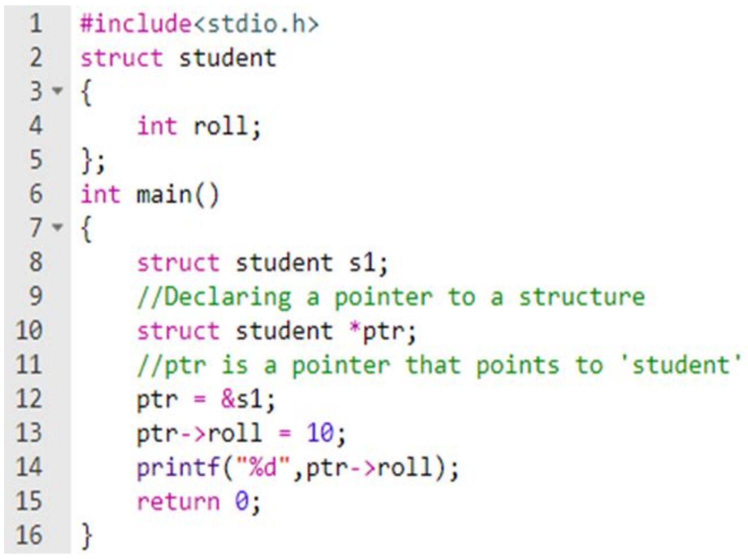
We have already looked into the concept of pointers. A pointer holds the memory location of another variable, isn't it so? We know that a pointer can point to (so to speak) to an integer variable, it can point to a floating point variable and so on. Can it point to a structure? To answer this question, we need to go back into the definition of a structure. Ask yourselves - what is a structure ? In the early slides i had mentioned that a structure is a user defined data type, right?



```c
1   #include<stdio.h>
2   struct student
3   {
4       int roll=20;
5       char name[50];
6       float marks;
7   };
8   int main()
9   {
10      struct student s1;
11      printf("%d",s1.roll);
12      return 0;
13  }
```

Run this code and study the reasons for the output. We will be surprised that this code gives an error. The error might be that 'roll' does not exist in the structure 'student' or something similar to that. <u>We are initializing the structure member 'roll' with the value 2 within the structure. But when we compile the code we are getting an error. What might be the reason for this?</u>

It is because of the fact that we tried to initialize a structure member assuming it would behave like a variable. But remember our definition in our early slide - Structure student as a whole is a data type and we must not assume that it is a variable. Thus we cannot write like we did in line number 4. <u>The fact that structure is a data type enables us to use a pointer to point to the address</u>. A little bit of code might bring in more clarity.



```c
1   #include<stdio.h>
2   struct student
3   {
4       int roll;
5   };
6   int main()
7   {
8       struct student s1;
9       //Declaring a pointer to a structure
10      struct student *ptr;
11      //ptr is a pointer that points to 'student'
12      ptr = &s1;
13      ptr->roll = 10;
14      printf("%d",ptr->roll);
15      return 0;
16  }
```

Line no: 8 - a structure variable by the name s1 (s1 is of the structure 'student') has been declared.

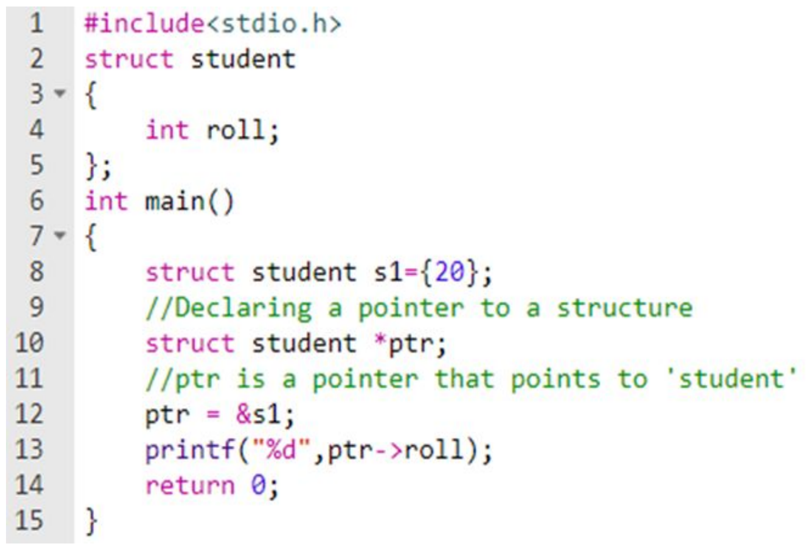
Line no: 10 - a pointer by the name ptr to the structure 'student' has been declared.

Line no:12 - the address of s1 is assigned to ptr

Line no:13 - accessing structure members

Other than pointer, the '->' operator also has to be emphasised in the program given above. It is used to access the structure data members using a pointer. A question that is often asked by many is what is the necessity of pointers? Are we not better off without pointers? We shall come back to this question after a while.

Before moving on further, please note one more method of initializing structure members.

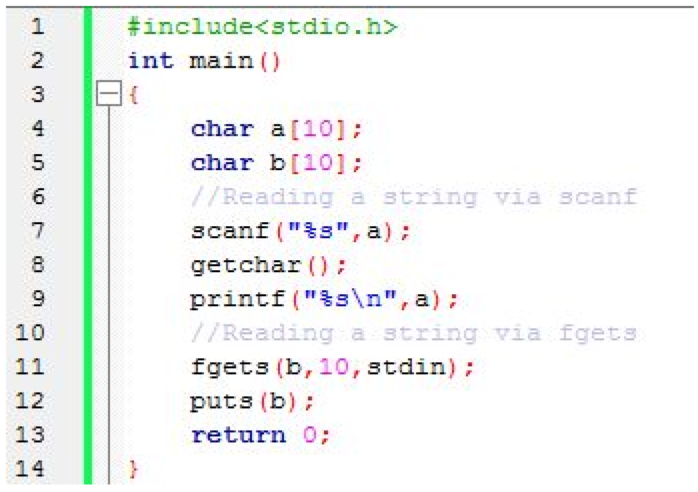


```
1  #include<stdio.h>
2  struct student
3  {
4      int roll;
5  };
6  int main()
7  {
8      struct student s1={20};
9      //Declaring a pointer to a structure
10     struct student *ptr;
11     //ptr is a pointer that points to 'student'
12     ptr = &s1;
13     printf("%d",ptr->roll);
14     return 0;
15 }
```

Line number 8 - another method to initialize structure members

Dynamic memory allocation

Let us look at the code given below:



```
1  #include<stdio.h>
2  int main()
3  {
4      char a[10];
5      char b[10];
6      //Reading a string via scanf
7      scanf("%s",a);
8      getchar();
9      printf("%s\n",a);
10     //Reading a string via fgets
11     fgets(b,10,stdin);
12     puts(b);
13     return 0;
14 }
```

As is plainly understood by looking at the code, it seeks to read user input, a string and display it back, using scanf and using fgets.

Line numbers 4 and 5 inform the compiler that memory for 10 characters have to be set aside. Since it is an array, it has to be 10 consecutive bytes. Such type of memory allocation is called as 'static' memory allocation as the memory is allocated at compile time and not at run time.

Static memory allocation:

You should have heard about global scope and local scope.
Variables that can be accessed from anywhere in the program are called as global variables. When it comes to global variables, memory is allocated to them at the start of the program and the memory remain allotted to them throughout the program's existence.

What happens to the allocated memory of the global variables?

It is freed. Meaning, that some other program can use this memory after the program terminates. There is no need an explicit code to free the memory allocated for global variables.

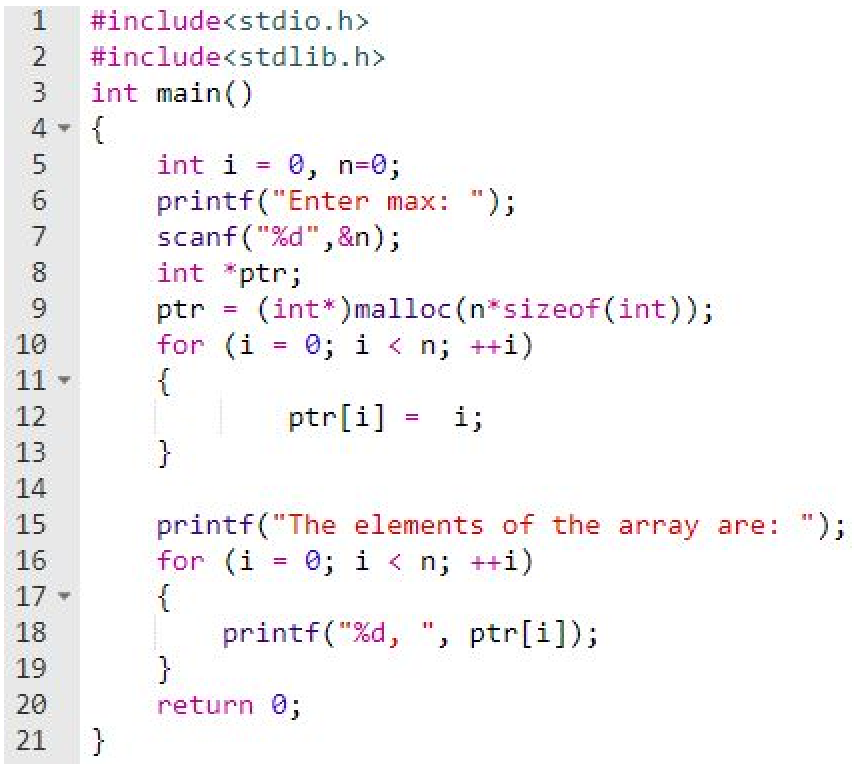
Now consider local variables. We know that local variables are limited in their scope within the function or loop within which they are declared and when the function terminates the scope of that particular local variable also terminates. The memory allocated to it is freed just like a global variable.

Although the memory for global and local variables is allocated and freed implicitly, there is one disadvantage, the array size is fixed at the time of compilation. In the scenario where the user needs more memory at run time, the memory cannot be extended or reduced (if the memory requirements are lesser)

Dynamic memory allocation provides us with options to manage the memory which is available. To allocate memory at run time, C provides us with <stdlib.h> header file which has four functions namely,

- malloc - (cast-type*)malloc(n* sizeof(type))
- calloc - (cast-type*)calloc(n,sizeof(type))
- realloc- (cast-type*)realloc(ptr, n*sizeof(type))
- free

malloc - It allocates a memory block of size n bytes. These n bytes are allocated in consecutive memory locations. This function returns a pointer to the beginning of the memory block or NULL if there is no space available. The pointer it returns is of type void which can be cast to any type. To get further insight into malloc, let us take a look at the code given below:

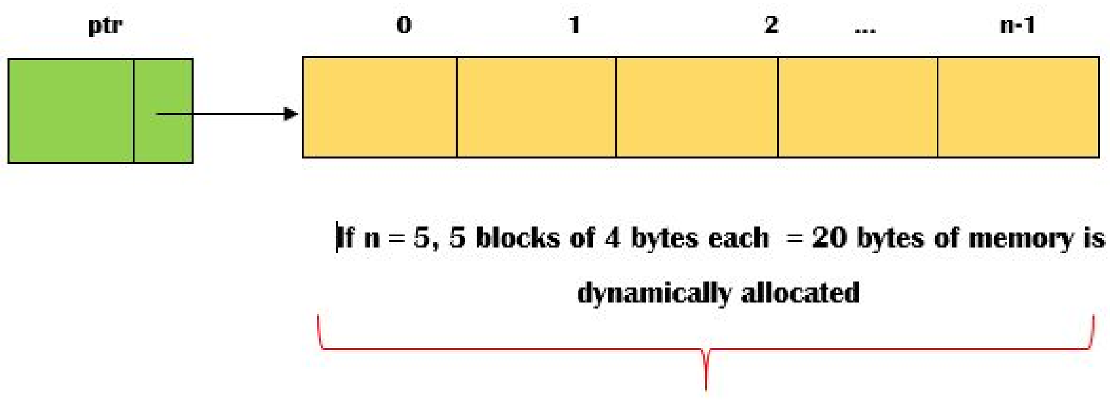


```c
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main()
4  {
5      int i = 0, n=0;
6      printf("Enter max: ");
7      scanf("%d",&n);
8      int *ptr;
9      ptr = (int*)malloc(n*sizeof(int));
10     for (i = 0; i < n; ++i)
11     {
12             ptr[i] =  i;
13     }
14
15     printf("The elements of the array are: ");
16     for (i = 0; i < n; ++i)
17     {
18         printf("%d, ", ptr[i]);
19     }
20     return 0;
21 }
```

Here we are declaring a pointer of integer data type.

Line number 9 - the value of 'n' entered by the user decides the memory to be allocated.

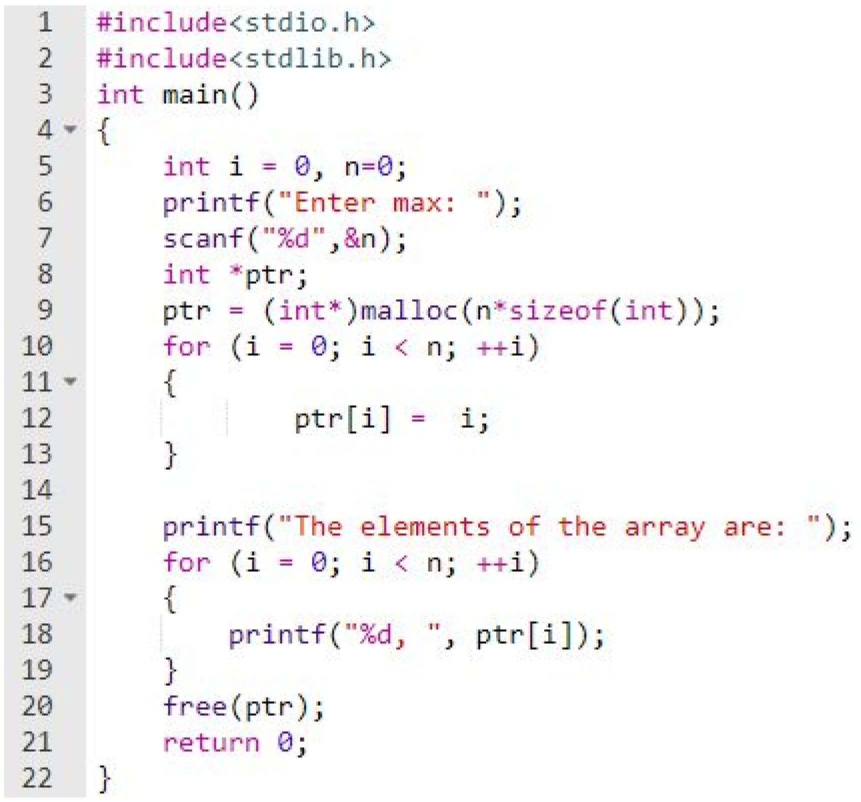
Line number 9 - the pointer in this instance is cast to type 'int' . The pointer now points to the base address (first address) of the n consecutive memory blocks allocated for the program.



If n = 5, 5 blocks of 4 bytes each = 20 bytes of memory is dynamically allocated

Consecutive memory locations (a memory block)

ptr points to the first byte, whereas ++ptr points to the next byte and so on



```
1   #include<stdio.h>
2   #include<stdlib.h>
3   int main()
4   {
5       int i = 0, n=0;
6       printf("Enter max: ");
7       scanf("%d",&n);
8       int *ptr;
9       ptr = (int*)malloc(n*sizeof(int));
10      for (i = 0; i < n; ++i)
11      {
12              ptr[i] =  i;
13      }
14
15      printf("The elements of the array are: ");
16      for (i = 0; i < n; ++i)
17      {
18          printf("%d, ", ptr[i]);
19      }
20      free(ptr);
21      return 0;
22  }
```

This program adds one more line to the earlier one.

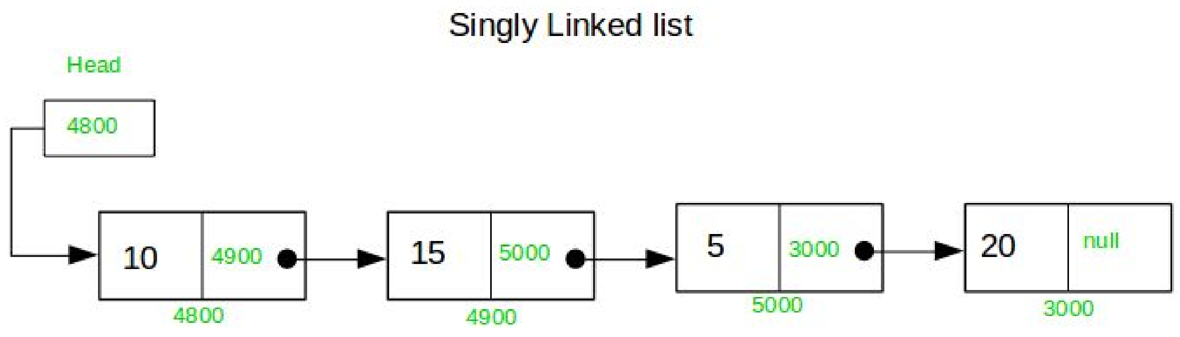
Line number 20, frees the memory allocated earlier in line number 9

C does not have a garbage collection like Java, so memory which is in no more use has to be explicitly deleted.

The difference between calloc and malloc is that in calloc the allocated blocks of consecutive memory locations are set to zero, whereas in malloc, we are not sure if the blocks of memory have some left over value from an earlier program that can probably cause problems going further. realloc() is used to reorganize the memory already allocated using malloc or realloc, if that memory is proven to insufficient.

DMA in Linked Lists

Sometimes we do not know how many numbers or characters does the user want to enter. Under such circumstances we cannot use static memory allocation. Say for example, we want to read real time data from a source, how can we allocate memory statically? A possible solution for such circumstances is use linked lists to dynamically grow the memory requirement. A linked list can be visualized as given in the picture below:



(Pic courtesy: geekforgeeks.org)