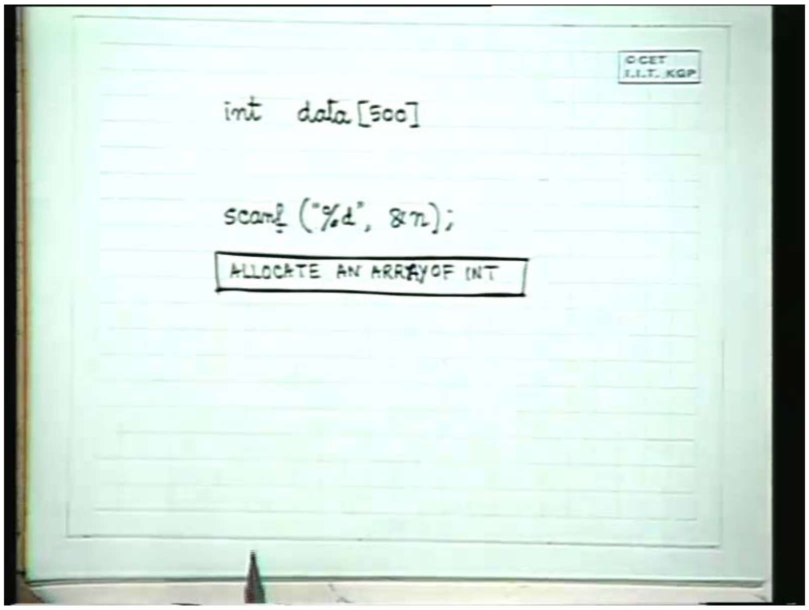
Programming & Data Structures

Dr. P.P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 16
Dynamic Allocation
Part– I

We shall today study dynamic allocation of data and as we discussed in the previous class that suppose we have to write a program which sorts n numbers where the value of n is to be read and the maximum value of n is unknown then we cannot declare an array of size of some size say 5000, 500 because we do not know what is the maximum size.

(Refer Slide Time: 02:33)

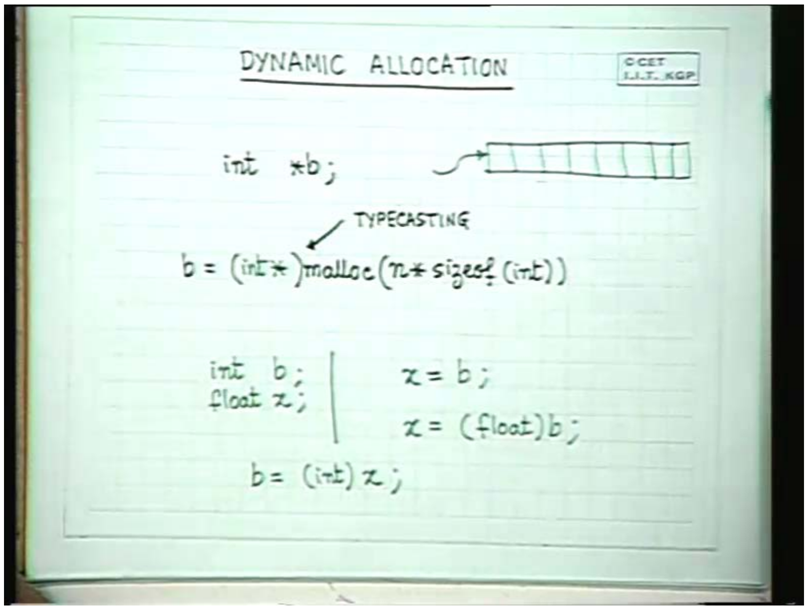


```
int  data[500]

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

ALLOCATE AN ARRAY OF INT

So we have to dynamically allocate within the program some array after having read the value of n. So in the program we will have scanf and then we will have to allocate an array of integers of size n. So we do this by a sequence of definitions and by some standard functions provided in C. Now before we do this, we will see how, what we have to declare to define an array of such size.

(Refer Slide Time: 07:01)



DYNAMIC ALLOCATION

```
int *b;

b = (int*)malloc(n* sizeof (int))

int b;      |   x = b;
float x;    |
            |   x = (float)b;
b= (int) x;
```

TYPECASTING
© CET I.I.T. KGP

So in order to declare an array of size n, after having read n we will call the statement and we will have to define an array name for it. Now the name of the array is not known to us. So we will define a variable which will, whose content will be the starting address of that array. So we will define b to be a pointer to an integer, alright. We will define b to be a pointer to an integer and then we will allocate n elements of that size. And for that we use malloc and we will have to allocate data of size n elements where each element is a size of an integer.

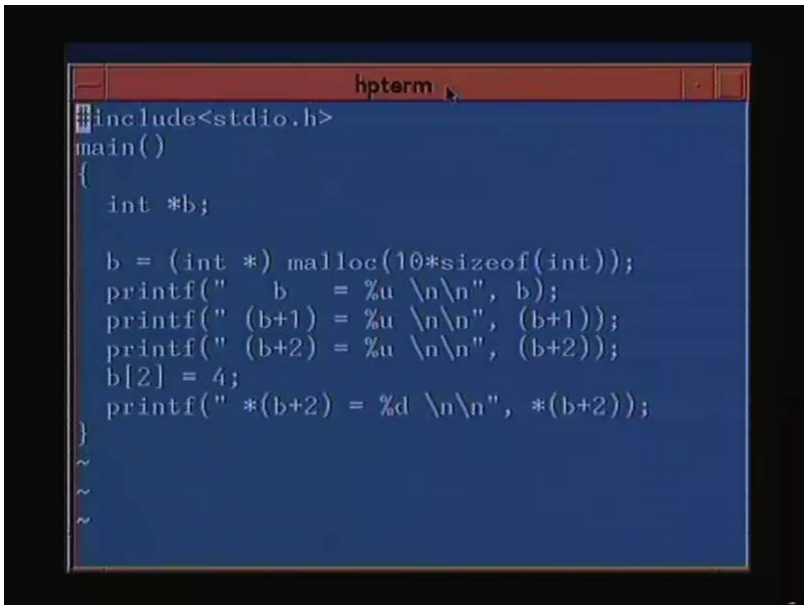
So, n multiplied by size of an integer. Size of is a predefined function, this can be any type, it can even be a structure type where we have defined it to be integer. So for any type automatically size of will give me the size of that structure, complex or student record whatever you want you can put it here and n is the number that we wanted after having read n. And we will assign it to b but before assigning it to b, we have to do what is called type casting.

Now what is type casting? In C language suppose I have defined an integer b and I have defined a floating point number x. Now suppose I want to do x is equal to b and normally when x is made assigned to b, system doesn't automatic type change but if you want to explicitly do this type change you have to write x is equal to… this converts the expression here to this type and then this is assigned.

On the other hand if I want to put b is equal to x, I will have to convert it to an integer and do this conversion of type. Now here I have just got a block of n elements which I have allocated n elements of type integer. So this will just, suppose this will be computed to be say 4 and n into 4 suppose n was 10 and n into 4, 40 will be this expression will evaluate to 40 and a block of 40 will be allocated and address of that location will come.

But we want this to be converted to the same type as this. So the type of this is what? Just remove this variable, you will get its type. a very easy way to find out what the type is, just remove the variable name you will get its type, int star. Now after having done this now b, the content of b has got the address. Now, this malloc will return the address int star this will return the address. So b is now pointing to an array of n elements. You have physically allocated this array of n elements. Now you can read in the n elements for sorting. So we will see three examples of dynamic allocation in this class and see how we can allocate and depending on what we want to allocate and depending on what we write, what is being allocated. So we will see them on computer and move back on to the terminal as and when required.

(Refer Slide Time: 07:32)

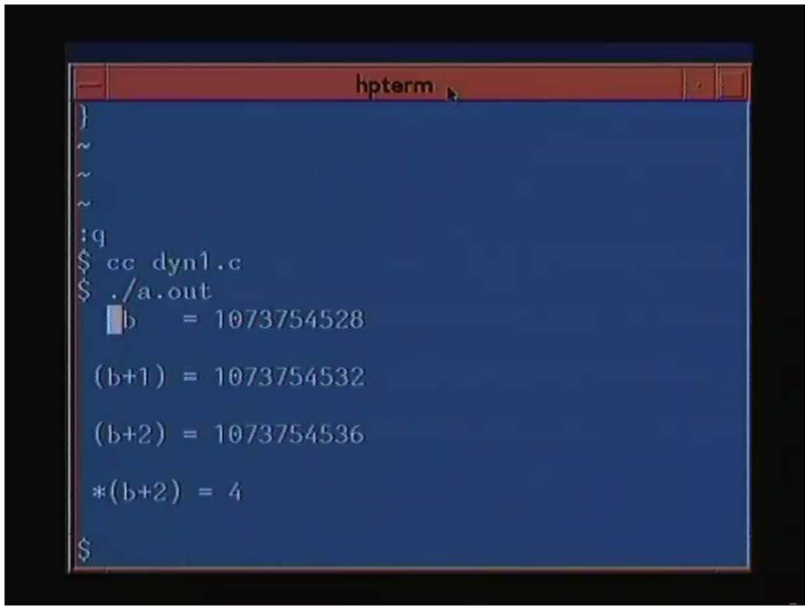


```
#include<stdio.h>
main()
{
  int *b;

  b = (int *) malloc(10*sizeof(int));
  printf("   b    = %u \n\n", b);
  printf(" (b+1) = %u \n\n", (b+1));
  printf(" (b+2) = %u \n\n", (b+2));
  b[2] = 4;
  printf(" *(b+2) = %d \n\n", *(b+2));
}
~
~
~
```

The first one is the simple one. It is just as we discussed. We defined a pointer variable int star b and here we have done the allocation, b is equal to int star typecasting malloc ten into size of integer. Now let us see what happens after malloc. B, what is b? The b will give you the content of b. Mind you when I write b it means the content of b because it’s a variable will give you the address of this, b plus 1 since it’s a pointer to an integer, it will increment it by the size of an integer, b plus 2 will increment it by size of integer one more. And you can now use it just like you use an array, b [2] which is equivalent to what? We wrote down in the previous day, when in any array when I write b [2] this is converted to this expression, star of b plus 2 which is the same as here.

So you will see when I do b [2] equal to 4, I will get star of b plus 2 is equal to 4. So let us see what gets printed at the b, b plus 1 b plus 2 and star of b plus 1. This b is 1073, this is the starting location, alright. This is this and star of b plus 2 is equal to 4.

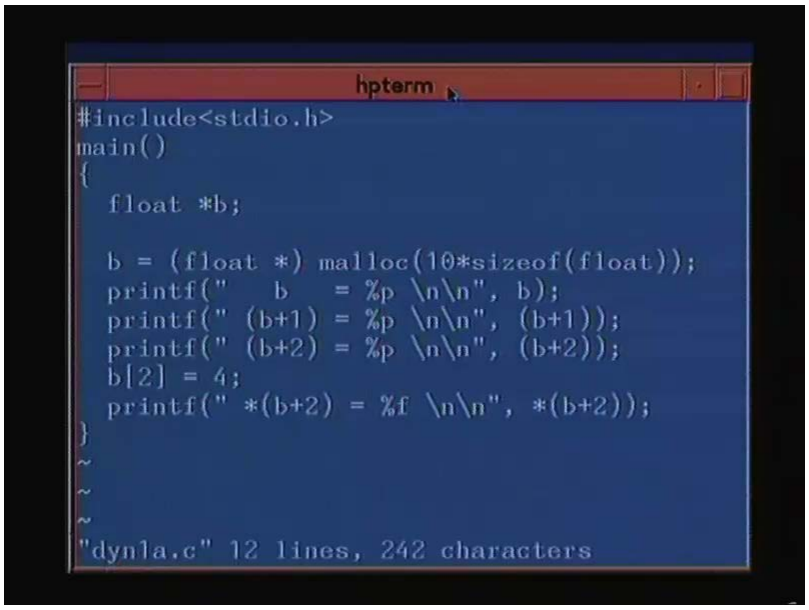
(Refer Slide Time: 09:15)



```
}
~
~
~
:q
$ cc dyn1.c
$ ./a.out
   b    = 1073754528

 (b+1) = 1073754532

 (b+2) = 1073754536

 *(b+2) = 4

$
```

Suppose you made into a floating point number then this casting we will have to change, size of float the rest will remain identical, there is no change in the rest of them but this b plus one will now jump by a different amount and which we will see now.

(Refer Slide Time: 11:25)

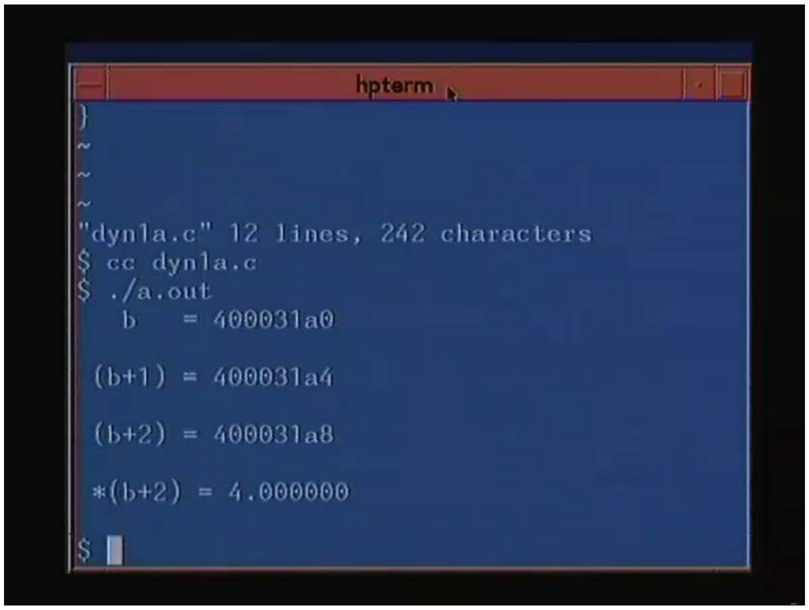


```
#include<stdio.h>
main()
{
  float *b;

  b = (float *) malloc(10*sizeof(float));
  printf("   b   = %p \n\n", b);
  printf(" (b+1) = %p \n\n", (b+1));
  printf(" (b+2) = %p \n\n", (b+2));
  b[2] = 4;
  printf(" *(b+2) = %f \n\n", *(b+2));
}
~
~
~
"dyn1a.c" 12 lines, 242 characters
```

Now this is the program. Did you compile it here? % f we did not give, that’s right. No, but here we have given % u that’s alright but let us see. Here we have given b plus 1, this is a pointer to a floating point number.

a 0 to a 4, so here again 4 are allocated, a 4 to a 8 and b is four point but only 4 places were allocated b to b plus 1 and b plus 1 to b plus 2.

(Refer Slide Time: 11:35)

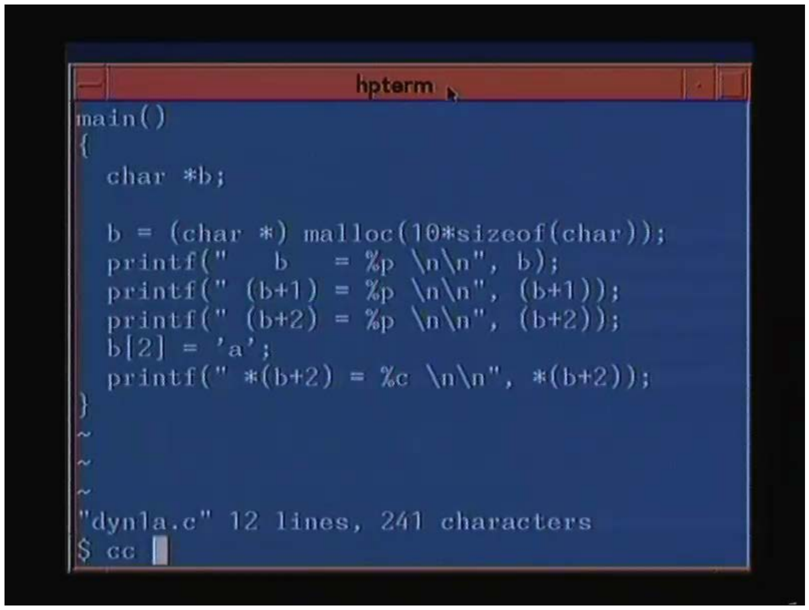


hpterm

```
}
~
~
~
"dyn1a.c" 12 lines, 242 characters
$ cc dyn1a.c
$ ./a.out
   b    = 400031a0

 (b+1) = 400031a4

 (b+2) = 400031a8

 *(b+2) = 4.000000

$ 
```

When you define it to be a character, on the other hand the sizes were only 1, just 1 1 and 1.

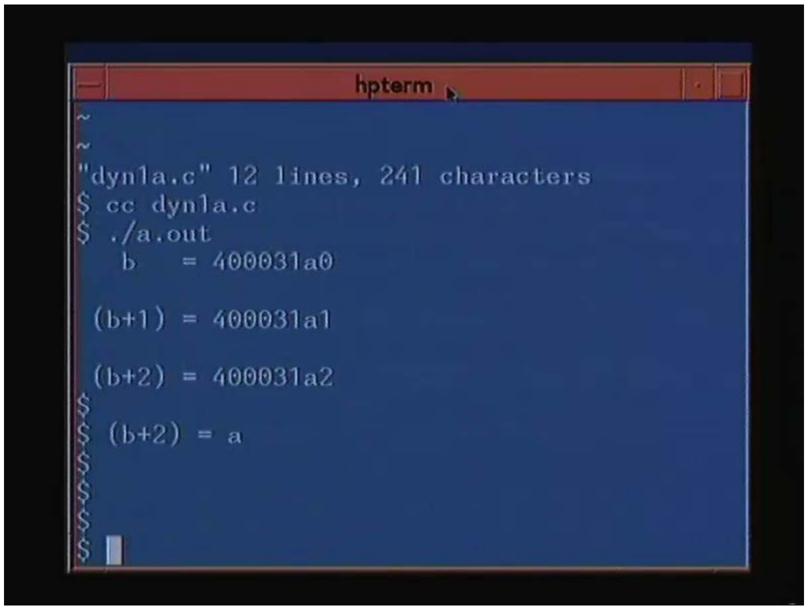
(Refer Slide Time: 12:55)



hpterm

```
main()
{
  char *b;

  b = (char *) malloc(10*sizeof(char));
  printf("   b    = %p \n\n", b);
  printf(" (b+1) = %p \n\n", (b+1));
  printf(" (b+2) = %p \n\n", (b+2));
  b[2] = 'a';
  printf(" *(b+2) = %c \n\n", *(b+2));
}
~
~
~
"dyn1a.c" 12 lines, 241 characters
$ cc 
```

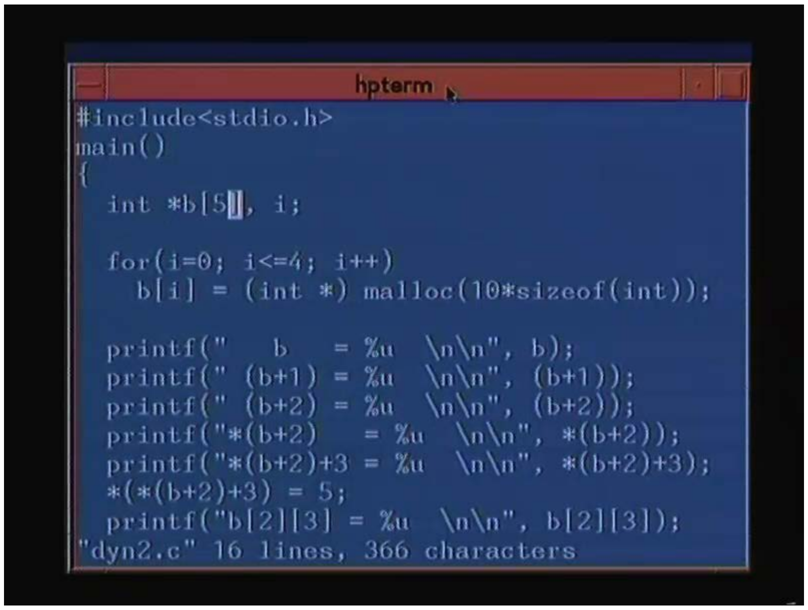
(Refer Slide Time: 13:25)



```
~
~
"dyn1a.c" 12 lines, 241 characters
$ cc dyn1a.c
$ ./a.out
   b    = 400031a0

 (b+1) = 400031a1

 (b+2) = 400031a2
$
$ (b+2) = a
$
$
$
$
```

Now let us see another program and let us see this declaration int star b [5]. So lets analyze what's this means.

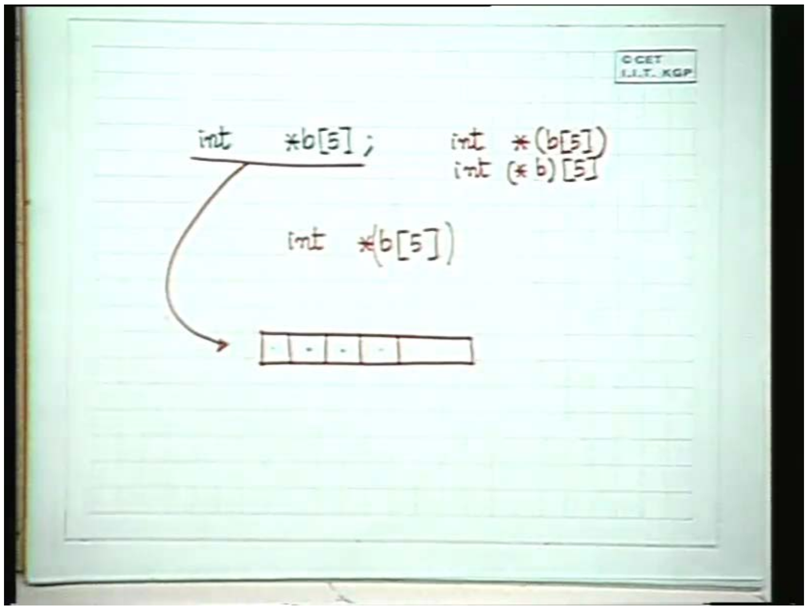
(Refer Slide Time: 13:40)



```
#include<stdio.h>
main()
{
  int *b[5], i;

  for(i=0; i<=4; i++)
    b[i] = (int *) malloc(10*sizeof(int));

  printf("   b    = %u  \n\n", b);
  printf(" (b+1) = %u  \n\n", (b+1));
  printf(" (b+2) = %u  \n\n", (b+2));
  printf("*(b+2)   = %u  \n\n", *(b+2));
  printf("*(b+2)+3 = %u  \n\n", *(b+2)+3);
  *(*(b+2)+3) = 5;
  printf("b[2][3] = %u  \n\n", b[2][3]);
"dyn2.c" 16 lines, 366 characters
```

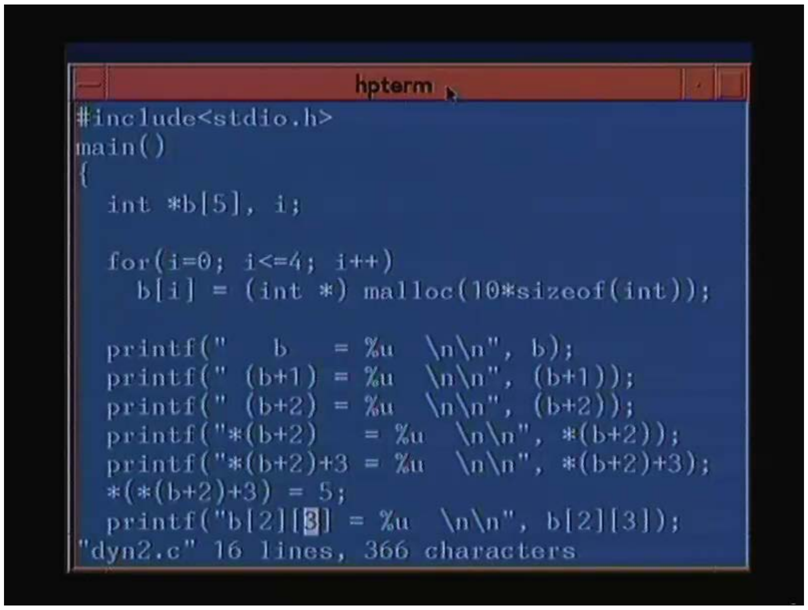
Suppose I define int star b [5], what does this mean? This is equivalent to int or is it equivalent to or are they same. Can you see these two things?

(Refer Slide Time: 16:01)



What does this mean and what does this mean? This means that an array of 5 elements, this one means that there is an array of 5 elements and each of them is a pointer to an integer, int star b [5], an array of 5 elements and each of them is a pointer to an integer. So this one, this definition leads to an array of 5 elements, so 5 elements are allocated of pointer type and each of them is a pointer to an integer. So you can now, if you malloc on b [0] with 10 elements, you will get 10 elements here. If you malloc on b [1] you will get another 10 elements here, if you malloc on b [2], malloc on b [3], malloc on b [4], alright.

(Refer Slide Time: 17:30)



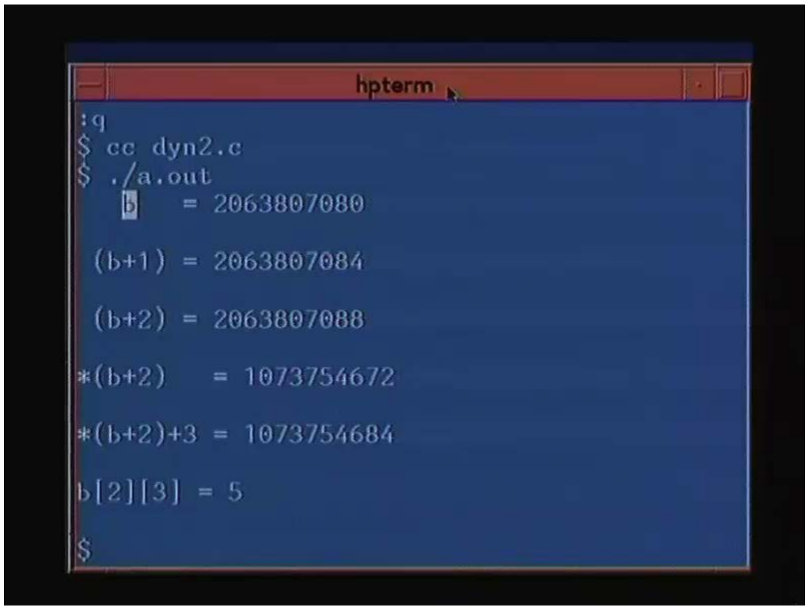
```
#include<stdio.h>
main()
{
  int *b[5], i;

  for(i=0; i<=4; i++)
    b[i] = (int *) malloc(10*sizeof(int));

  printf("   b    = %u  \n\n", b);
  printf(" (b+1) = %u  \n\n", (b+1));
  printf(" (b+2) = %u  \n\n", (b+2));
  printf("*(b+2)    = %u  \n\n", *(b+2));
  printf("*(b+2)+3 = %u  \n\n", *(b+2)+3);
  *(*(b+2)+3) = 5;
  printf("b[2][3] = %u  \n\n", b[2][3]);
"dyn2.c" 16 lines, 366 characters
```

So let us see how we will get this program. Here we have done int star b [5], so we have defined an array of pointers to integers. And in a loop from 0 to 5, we have allocated each b[i] to 10 integers, so that way we have defined an array of, a two dimensional array has been defined. A two dimensional array has been defined, 5 by 10 alright and you can access that two dimensional array for example b plus 2, what will b plus 2 give you? This is an array of pointer, so it will give you the element which is at the second. It will give you the value of the second pointer, third pointer b [2]. And the content of that will give you the start address of that array plus 3 will give you the third element address in that array and content of that will give you b [2][3], alright.

So we will see what is b, b plus 1, b plus 2, star of b plus 2, star of b plus 2 plus 3 and this is the star of the whole of that which is the same as b [2][3]. So this is the another way, this is one way in which you can define a two dimensional array dynamically. Though we have kept this size to be constant for all of them, this size could be varied for each of the i's. For each of these i's this size, the size here this 10 could have been different. So you could have got an array of variable dimensions on each of these columns.

If these are the rows, if there are 5 rows then on each of the columns you would have got a different dimension. b is the start address, b plus 1, b plus 2 this is the content of b plus 2 which is the malloc address, this is the malloc address which you have malloc. These were all initially allocated in that array.

(Refer Slide Time: 18:35)

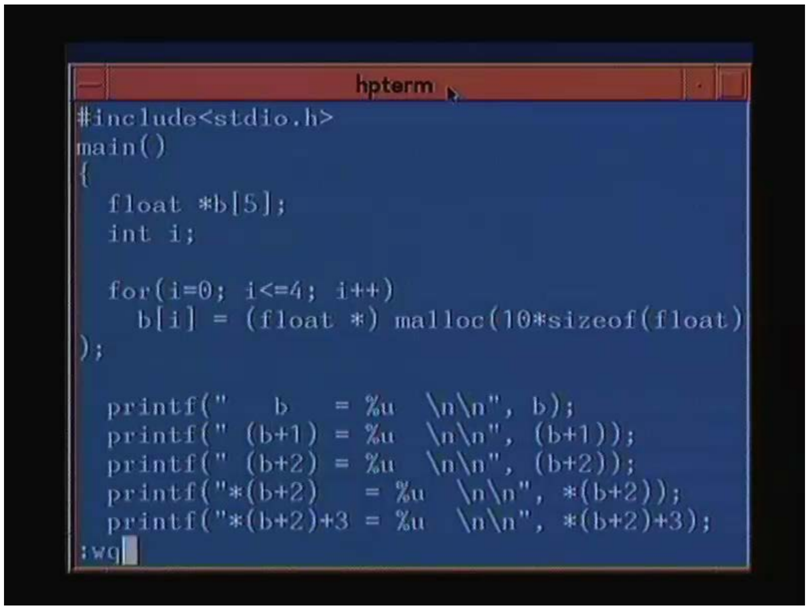


```
:q
$ cc dyn2.c
$ ./a.out
   b    = 2063807080

 (b+1) = 2063807084

 (b+2) = 2063807088

*(b+2)    = 1073754672

*(b+2)+3 = 1073754684

b[2][3] = 5

$
```

This is the malloc address, content of that b plus 2 plus 3, plus 3 will move by 12 and then what did we do. We did the star of this, we assigned to 5, this is the same as b [2][3] is equal to 5.

So we have defined a floating point array (Refer Slide Time: 20:08)). line 7, it says an incompatible type definition, b[i] which will define real numbers, each of them will be a real number.

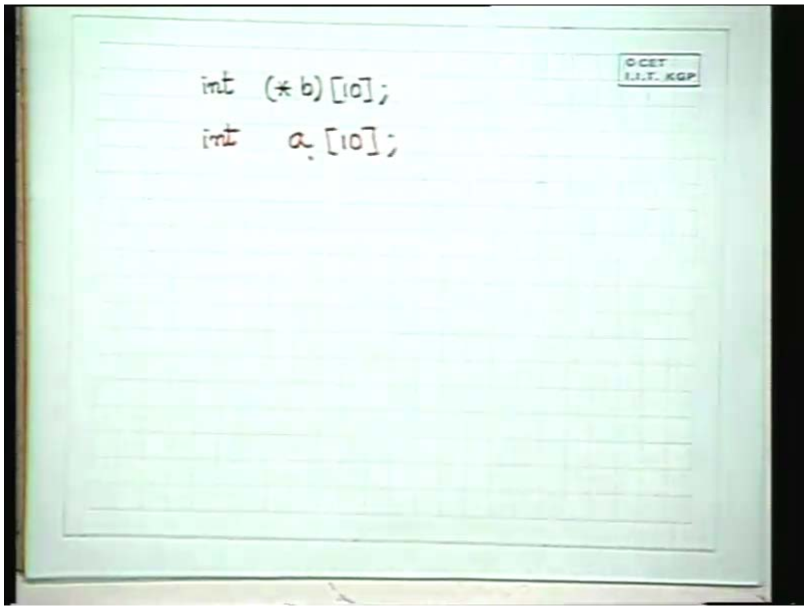
(Refer Slide Time: 20:42)



```
#include<stdio.h>
main()
{
  float *b[5];
  int i;

  for(i=0; i<=4; i++)
    b[i] = (float *) malloc(10*sizeof(float)
);

  printf("   b    = %u  \n\n", b);
  printf(" (b+1) = %u  \n\n", (b+1));
  printf(" (b+2) = %u  \n\n", (b+2));
  printf("*(b+2)    = %u  \n\n", *(b+2));
  printf("*(b+2)+3 = %u  \n\n", *(b+2)+3);
:wq
```

Similarly if you want to define an array of pointers to characters, you could have defined it. An array of pointers to complex numbers, this should have been just complex provided you have defined the type definition of complex and you would have been able to do that.

(Refer Slide Time: 24:16)



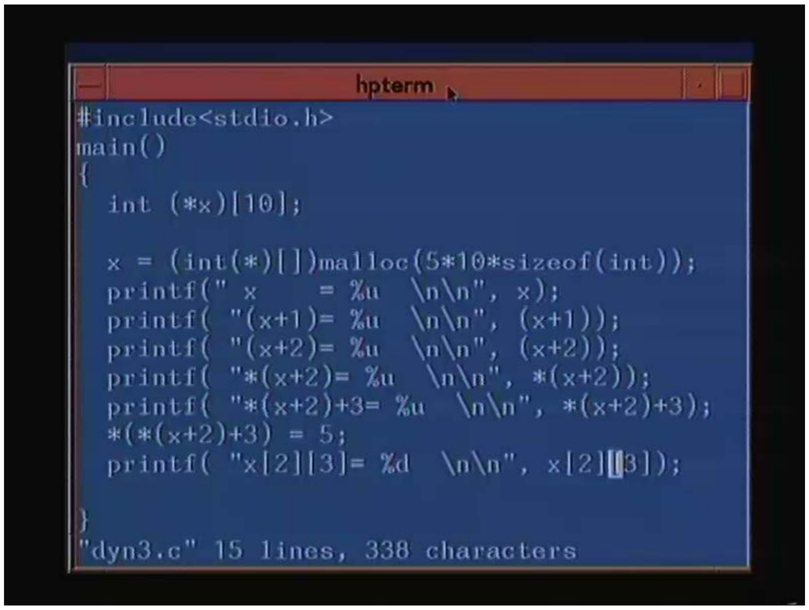
```
int  (*b)[10];
int   a[10];
```

Now coming back to the other one, this one what does this mean? To understand what this means let us see what this means. To understand what this means let us see what this means. This means a [10], so there is an array a of size 10 integers. So now look at these two, a is the address of the array. So this has the content of b is the address of an array of ten integers. So what is b? b is a pointer to the first such content and in order to define a two dimensional array, if you define 5 such blocks of b you will get a 5 by 10 array.

If you define 20 such blocks of b because each block is of size 10 elements. The other one was a, previous one was a pointer to arrays. The advantage with the previous one, see look at the previous one, look at this one. Suppose you wanted to define a, you are writing an editor and you want to define lines, alright. Now each line will have variable size, whoever is writing an editor you will define an character, so you would like to define pointer to character and then whenever if somebody has written one line, you will malloc only that size. In the next line you will malloc only that size whatever is in so many characters, the third line, the fourth line so on. This way you can malloc variable sizes, if it is a pointer array, this is a pointer array.

On the other hand this, each of them is a block of 5 characters, 5 integers, 5 arrays of 10 elements. So if you do b plus 1, you will jump 10 integers. If you do b plus 2, you will jump 10 integers. So using this also you can define a two dimensional array as we shall see now, int star x 10. Again malloc 5 into 10 into size of int that is fine but the type casting has to be given carefully. One easy way of giving typecasting as I told you just remove variable name you will get the typecasting type.

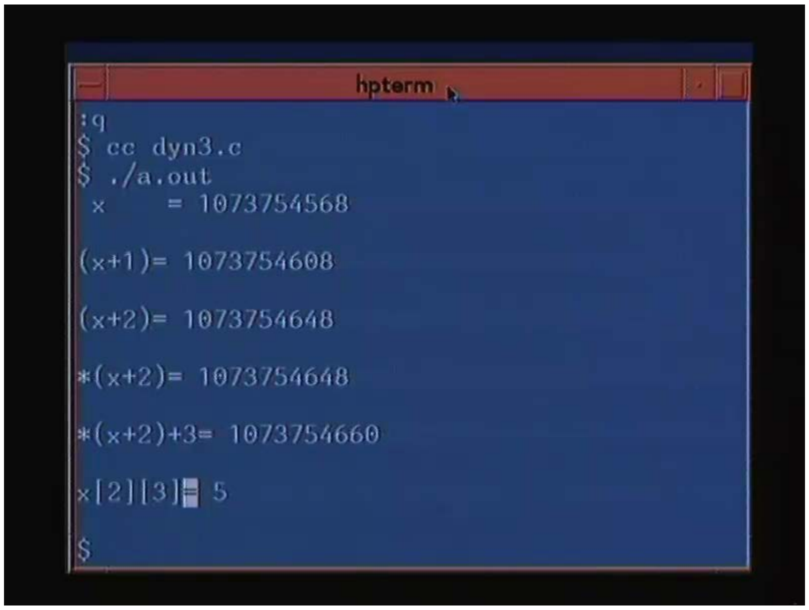
(Refer Slide Time: 25:15)



```
#include<stdio.h>
main()
{
  int (*x)[10];

  x = (int(*)[])malloc(5*10*sizeof(int));
  printf(" x     = %u  \n\n", x);
  printf( "(x+1)= %u  \n\n", (x+1));
  printf( "(x+2)= %u  \n\n", (x+2));
  printf( "*(x+2)= %u  \n\n", *(x+2));
  printf( "*(x+2)+3= %u  \n\n", *(x+2)+3);
  *(*(x+2)+3) = 5;
  printf( "x[2][3]= %d  \n\n", x[2][3]);

}
"dyn3.c" 15 lines, 338 characters
```

Remove all the variables names and the values of the constants, you will get the type. Here just remove this x and this 10, this is one thumb rule which is often usefully applied. Malloc 5 into 10 into size of int, total 50 blocks are being allocated. Then you select what is x, what is x plus 1, what is x plus 2, what is the content of x plus 1, x plus 2, what is the content of x plus 2, plus 3 and so on.

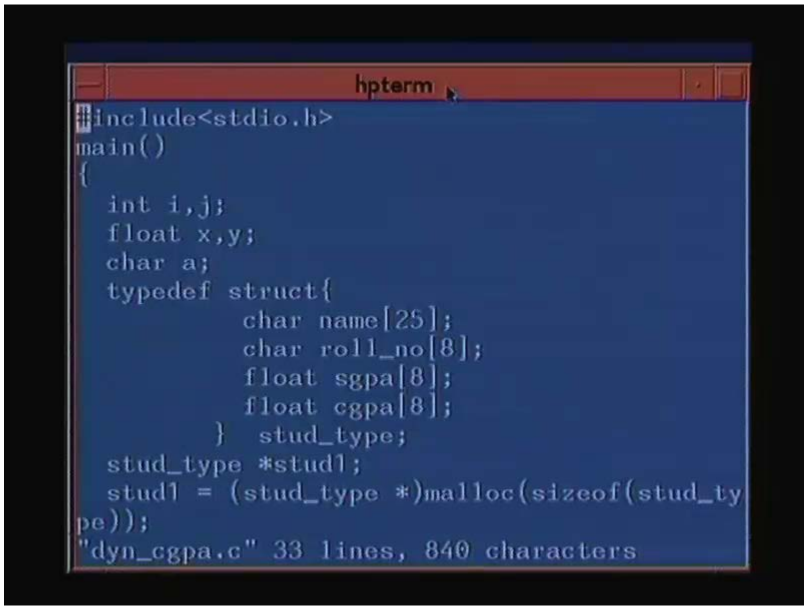
(Refer Slide Time: 26:00)



```
:q
$ cc dyn3.c
$ ./a.out
 x     = 1073754568

(x+1)= 1073754608

(x+2)= 1073754648

*(x+2)= 1073754648

*(x+2)+3= 1073754660

x[2][3]= 5

$
```

So let us see what this gives. x is this, x plus 1 have you noted that it has jumped by 40 blocks, x plus 2 has jumped by 40 blocks, content of x plus 2 is the same as this because it points to that two dimensional array and plus 3, now it will jump by 12 blocks and then x [2] [3] is equal to 5. So this is another way in which you can define two dimensional array. But these arrays are, each of them are of size fixed 10 10 10 10 10, the way they will be stored in the memory will be slightly different depending on, you can see from the way the address are stored. We will see how they are stored in the memory.

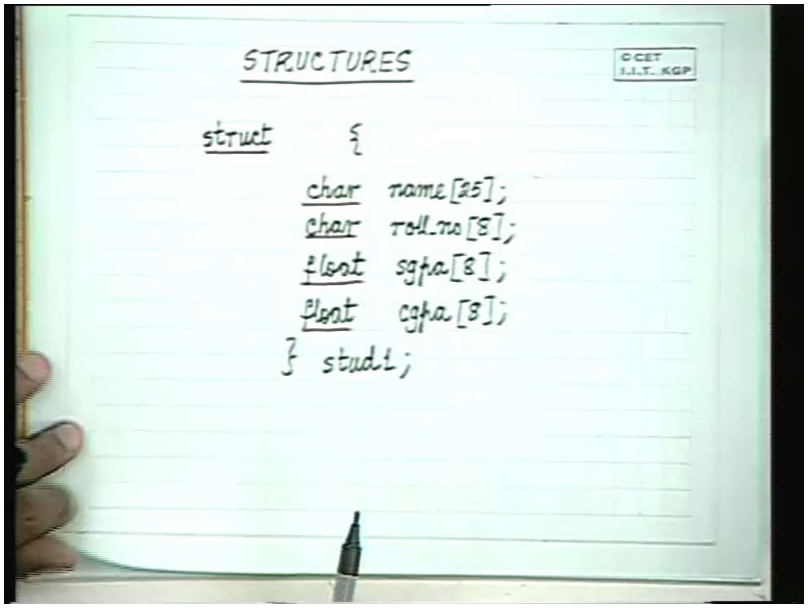
(Refer Slide Time: 26:50)



```
#include<stdio.h>
main()
{
  int i,j;
  float x,y;
  char a;
  typedef struct{
           char name[25];
           char roll_no[8];
           float sgpa[8];
           float cgpa[8];
         }  stud_type;
  stud_type *stud1;
  stud1 = (stud_type *)malloc(sizeof(stud_ty
pe));
"dyn_cgpa.c" 33 lines, 840 characters
```

So now what we want is, we will now define, for one student we will allocate. So we have given a type def structure for the student. Let's work it out. Let's work it out. For the student record, if you recall the structure of the student record was something like this struct char, char, float, float, stud one, alright.

(Refer Slide Time: 27:19)



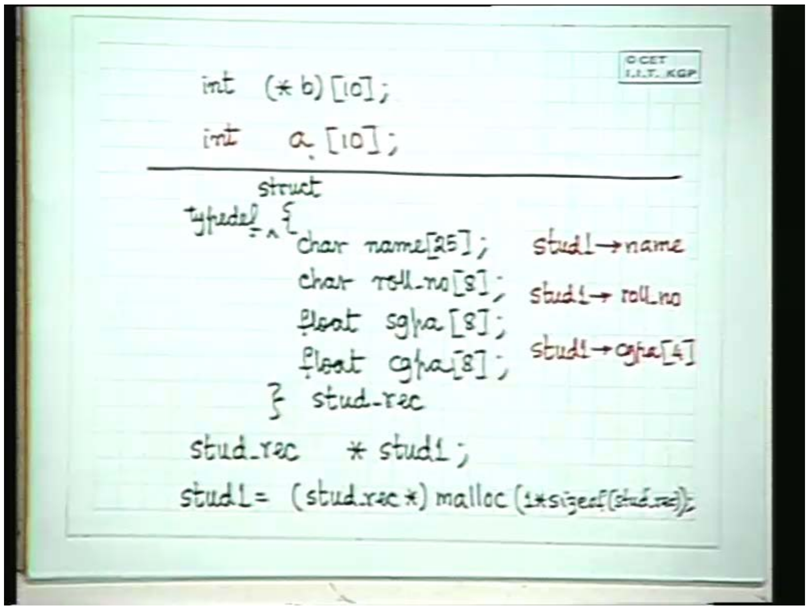
STRUCTURES

```
struct     {
        char   name[25];
        char   roll_no[8];
        float  sgpa[8];
        float  cgpa[8];
       } stud1;
```

So what we will now do is we will now define something like this. We will define only a pointer and whenever required we will allocate one student record.

So we will do a type def char name [25], type def struct has to be written, char roll number [8], float sgpa [8], float cgpa [8]. This is what we have defined.

(Refer Slide Time: 30:35)

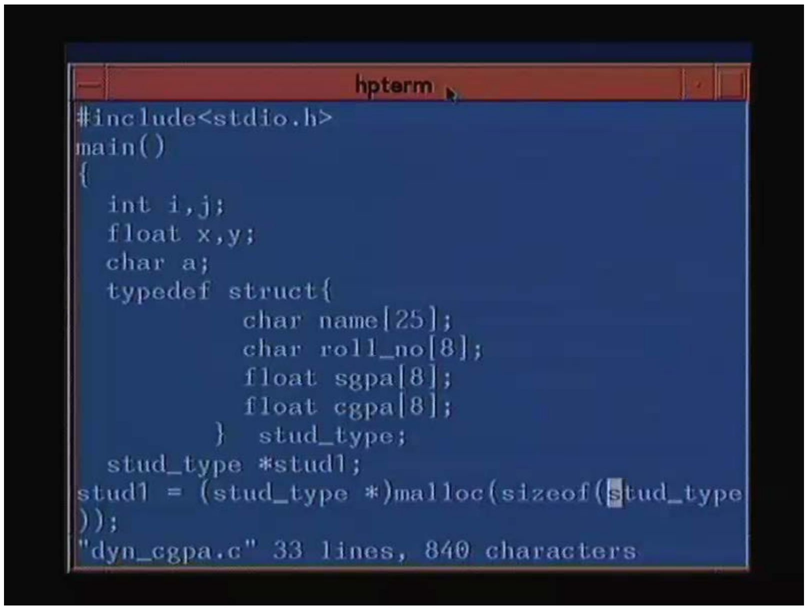


```
int  (*b)[10];
int   a[10];
________________________________________
        struct
typedef  {
          char name[25];      stud1→name
          char roll_no[8];    stud1→roll_no
          float sgpa[8];
          float cgpa[8];      stud1→cgpa[4]
        } stud_rec
stud_rec   *stud1;
stud1= (stud_rec *) malloc (1*sizeof(stud_rec));
```

Anyway I think you also got this definition before and if you want to have, you don't want, this is just type declaration and you can define stud_rec. Now you have defined a pointer to that and you can dynamically allocate it as stud 1 is equal to stud rec star malloc 1 into, let us say we want only one, if you want 20, 30, whatever you can do. Suppose you want just one then 1 into size of stud rec. If you want 20, you can allocate 20 an array. If you want 30 you can allocate 30 elements, whatever you want but if you want one, you can allocate only one. So this is how we allocate it and but this is now a pointer.

So if you want to access the name of this student then we will have to write down stud 1 pointer, roll number will be stud 1 pointer, the 8th semester 5th semester cgpa of this student will be because its 0 2. So this is just the modification in the student record program that we had written which will lead us to this structure. So we can have a quick look at that program now.

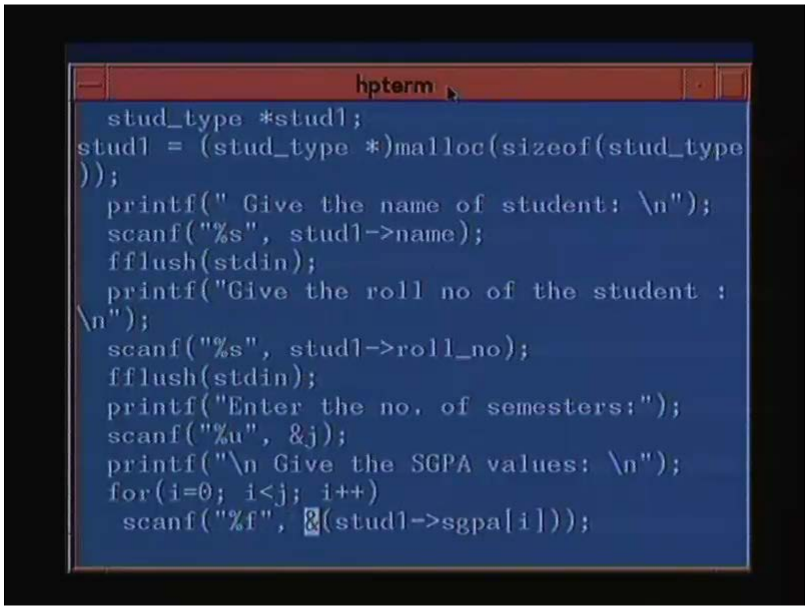
(Refer Slide Time: 31:03)



```
#include<stdio.h>
main()
{
  int i,j;
  float x,y;
  char a;
  typedef struct{
           char name[25];
           char roll_no[8];
           float sgpa[8];
           float cgpa[8];
         }  stud_type;
  stud_type *stud1;
stud1 = (stud_type *)malloc(sizeof(stud_type
));
"dyn_cgpa.c" 33 lines, 840 characters
```

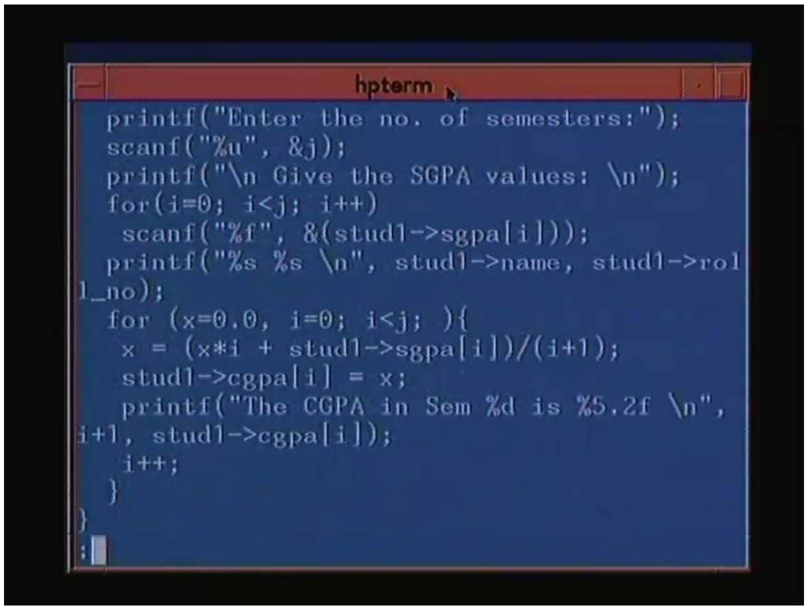
So this is where we have defined. This stud 1 is a student type, we malloc this. Stud 1 is equal to stud type star malloc size of stud type size. If nothing is written, it is 1 into then give the name of the student.

(Refer Slide Time: 31:31)



```
  stud_type *stud1;
stud1 = (stud_type *)malloc(sizeof(stud_type
));
  printf(" Give the name of student: \n");
  scanf("%s", stud1->name);
  fflush(stdin);
  printf("Give the roll no of the student :
\n");
  scanf("%s", stud1->roll_no);
  fflush(stdin);
  printf("Enter the no. of semesters:");
  scanf("%u", &j);
  printf("\n Give the SGPA values: \n");
  for(i=0; i<j; i++)
   scanf("%f", &(stud1->sgpa[i]));
```

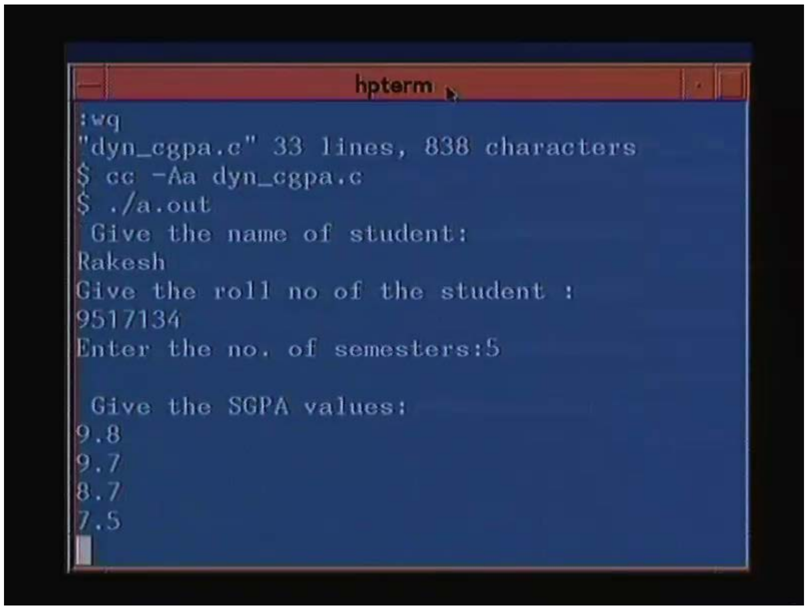
(Refer Slide Time: 31:46)



```
hpterm
  printf("Enter the no. of semesters:");
  scanf("%u", &j);
  printf("\n Give the SGPA values: \n");
  for(i=0; i<j; i++)
   scanf("%f", &(stud1->sgpa[i]));
  printf("%s %s \n", stud1->name, stud1->rol
l_no);
  for (x=0.0, i=0; i<j; ){
   x = (x*i + stud1->sgpa[i])/(i+1);
   stud1->cgpa[i] = x;
   printf("The CGPA in Sem %d is %5.2f \n",
i+1, stud1->cgpa[i]);
   i++;
  }
}
:
```

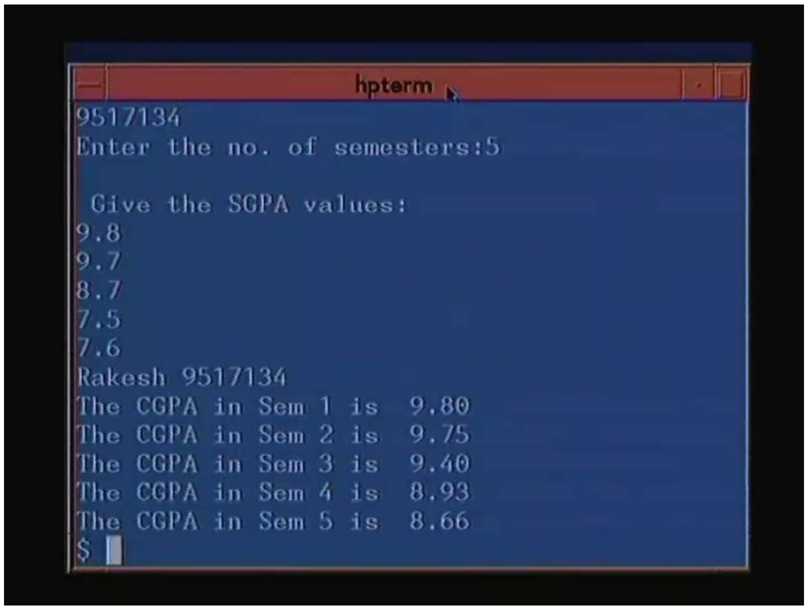
Now in scanf, since it’s a string stud one dot name will give the name of the string, you don’t have to give here, then roll number pointer then the number of semesters. Here you will have to make it and because this is the floating point number stud 1 pointer cgpa sgpa [i], just the dots are changed to pointer notations and the whole thing will run. So if you just, so this is how we can allocate and deallocate even structures. And as I said before here you can declare even an array of such structures by defining the number of students 10 into 20 into whatever.

(Refer Slide Time: 32:24)



```
hpterm
:wq
"dyn_cgpa.c" 33 lines, 838 characters
$ cc -Aa dyn_cgpa.c
$ ./a.out
 Give the name of student:
Rakesh
Give the roll no of the student :
9517134
Enter the no. of semesters:5

 Give the SGPA values:
9.8
9.7
8.7
7.5
```

(Refer Slide Time: 32:30)


hpterm

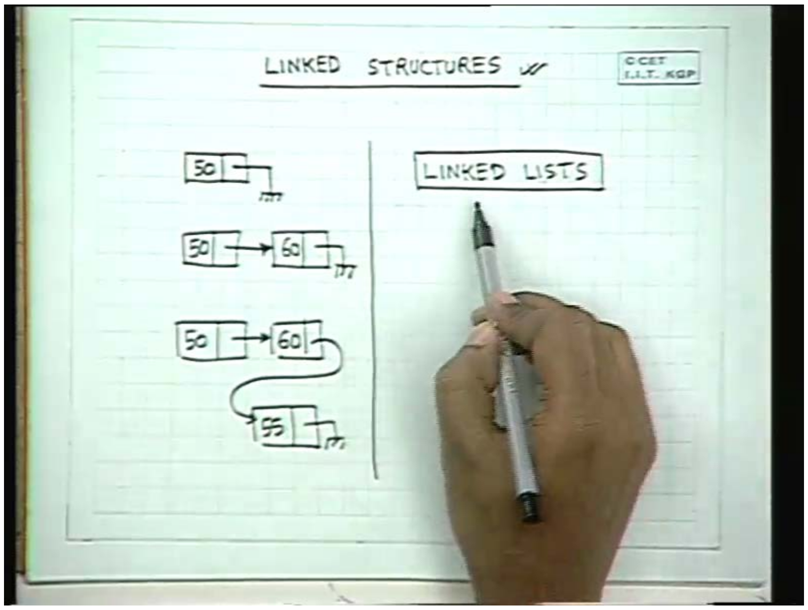
```
9517134
Enter the no. of semesters:5

 Give the SGPA values:
9.8
9.7
8.7
7.5
7.6
Rakesh 9517134
The CGPA in Sem 1 is  9.80
The CGPA in Sem 2 is  9.75
The CGPA in Sem 3 is  9.40
The CGPA in Sem 4 is  8.93
The CGPA in Sem 5 is  8.66
$
```

And then you can access it by stud 1[5] pointer roll number. Suppose you allocate 20 here and you want to access the 5th element then it will be something like this. Stud 1[5] that is the 5th element, so we have seen how dynamically we can allocate structures and we can allocate data. But if I give you this problem that I will tell you to sort n numbers alright or I will tell you to even print n numbers but I will not tell you how many numbers are there, neither will I tell you the value of n in the beginning. You will read n numbers and then I will ask you, do you want to read another number yes or no. And you will continue like this till I say no.

So now given n, n is the constant. There is no constant known for n neither do we have a value which is read saying n is equal to, you read n and then you allocate n elements. I will tell you to read elements and I will tell you to stop when the user wants you to stop. So you cannot really now allocate so many elements together and you cannot allocate one element at a time because you need so many pointer variables to allocate and you won’t know even the size of the pointer array to declare, if you want to declare it as a pointer set of pointers.

So how do we define those structures where the inputs come dynamically and you have to stop only when you will have to stop at a particular point of time. For that we will have to define what are called linked structures. That is I will read in one element and in this element I will store two things. One thing I will store is the value of this element and the next thing I will store is a pointer to the next element if it is read.

(Refer Slide Time: 36:58)



So first I read in 50 and this is what we call nullify, grounded that is no, we say there is nothing like this. And when you read in another element, you malloc a new element and make this pointer point to that element. So you point, you have 50, the next element. You read in another element it may say 60 then you read in another element, you allocate another element say 55 and you make this pointer point to this one. This way we can define link structures and you can just read in as many elements as and when required.

So next day we will study how to define, use and manipulate such link structures which are in the simplest variations are called linked lists. But if you think of matrices, you can also have a two dimensional matrix stored as a linked list in chains in both directions, the rows and the columns. So linked lists in general will be the topic of our future discussion.