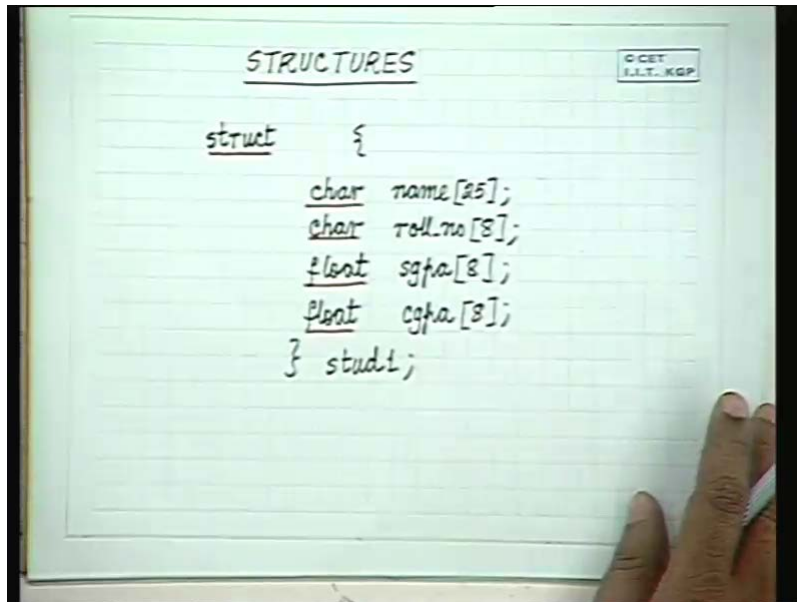


Programming & Data Structure
Dr. P.P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 15
Structures – II

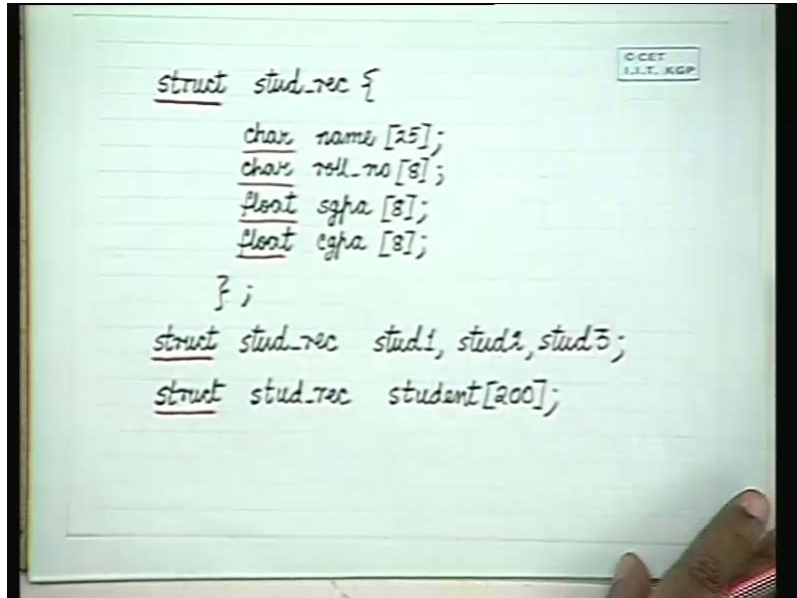
We shall continue our study of structures and we have seen in the previous day that structures are used for encapsulation of data, so that they are represented in a more comprehensive format and the examples which we saw for a student record. One we are declaring a structure variable, this is a structure variable was struct and then within brackets you define the structure of that variable.

(Refer Slide Time: 01:50)



And this is the name of the variable and as soon as you declare this name, this area will be allocated to this name. The area corresponding to 25 characters plus 8 characters plus 8 plus 8, 16 floating point numbers will be allocated to this student record. Other than pure variable structure declarations, we can also define structural types and define structures based on these types which we also saw. That is first you declare a structural type, here we declare a structural type stud rec which was a student record.

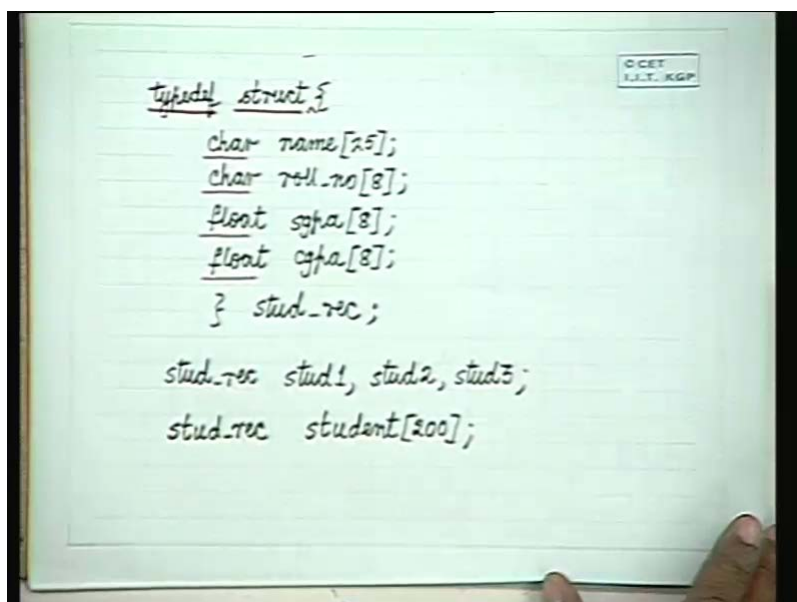
(Refer Slide Time: 02:50)



```
struct stud_rec {  
    char name [25];  
    char roll_no [8];  
    float sgpa [8];  
    float cgpa [8];  
};  
struct stud_rec stud1, stud2, stud3;  
struct stud_rec student[200];
```

And here we defined all these 25 characters a roll number of size 8, float sgpa 8 and float cgpa 8 and the name of this structure was called stud rec and this is a type declaration and not a variable declaration like integers, characters, etcetera this is a declaration of a particular type. And if you want to declare variables of this type, you have to write struct stud rec, so 3 variables of these types were declared. Here again a variable array student e of size 200 where each element is of this type struct stud rec was defined. And alternative definition of types is by the type def mechanism. Here the other way of writing it is type def struct and then the name of the type comes here.

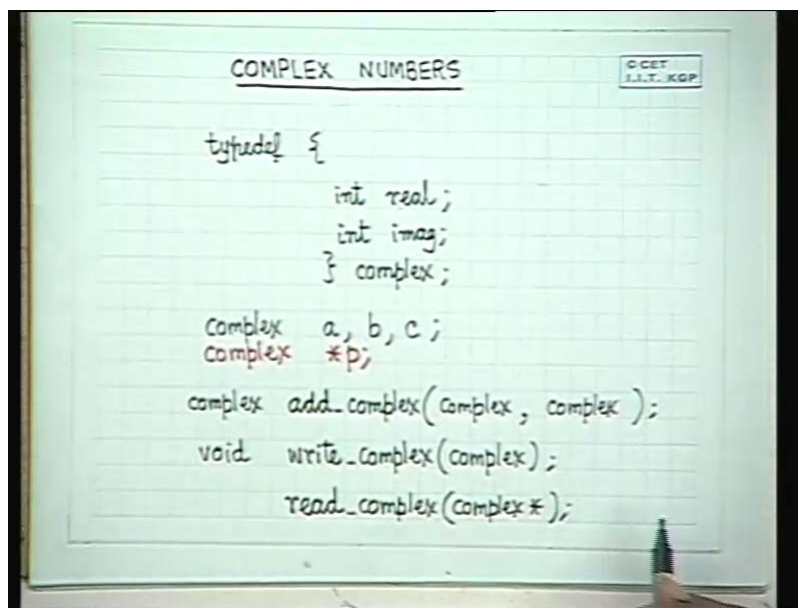
(Refer Slide Time: 03:35)



```
typedef struct {  
    char name [25];  
    char roll_no [8];  
    float sgpa [8];  
    float cgpa [8];  
} stud_rec;  
stud_rec stud1, stud2, stud3;  
stud_rec student[200];
```

This is the name of the type. Now this is slightly more versatile than the other type because here another name can be put in and the utility of putting another name here we will see when we come to self-referential structures. And in order to use this type definition, we saw that you have to just write `stud_rec stud 1 stud 2 stud 3` or `stud_rec student 200`. So this is how you can declare structures and they are used for data encapsulation. Now we will see simple program in which we will use structural definitions to perform what is called encapsulated programming with such structures. And this program is just to highlight how structural types that is you can define new data types using structures and you can write functions to operate on these data types and we will work on complex numbers.

(Refer Slide Time: 09:02)



So our intention is to write a C program which will be capable of defining a type for complex like integer, real, character we will define a complex type and once we define a complex type, we will define functions to read a complex number, write a complex number and also add two complex numbers. And this will show you how we can write it out in an elegant way and how we can define new types in our programming language.

So our structural definition of a complex number which contains real and integer, we will see, we will put in a type def, int real and int imaginary and call this type complex. So this is the type declaration, we have defined a complex type. In order to declare some variables of type complex, all that we have to do is write `complex a b c`. Now each of them will be a variable whose allocation will be one real, one integer for, two integers will be allocated to each of them. So now we have been able to define a new type called complex. Now we have to also additionally define our functions.

How will we define the structure of a function which adds complex numbers? The structure of a function which adds complex numbers say the function is called add

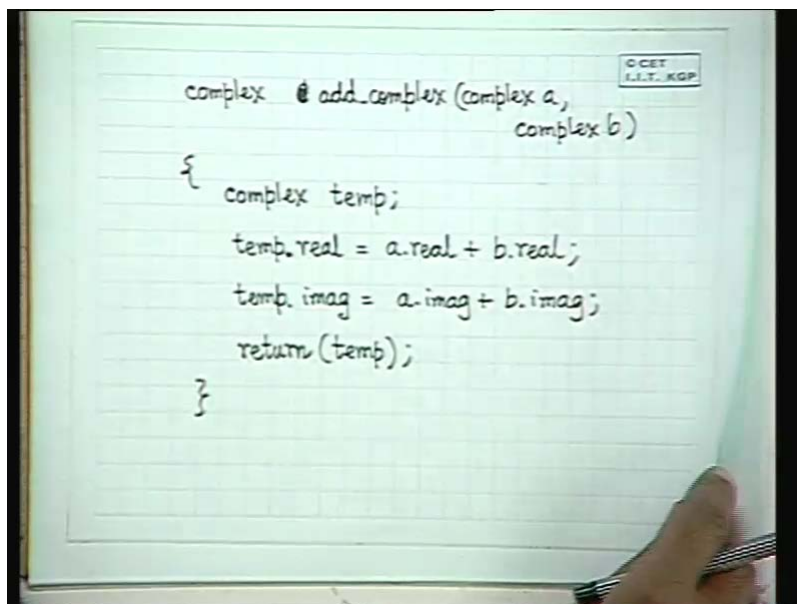
complex. What will be the two arguments? One will be, this will be of complex type and this will be of complex type. Two elements of type complex it will take and what will it return, what type? It will return another complex type. So now using such type definitions, we can return from functions, structures as well.

Similarly if you want to put in write complex then it will just take in a complex type and print it inside the function. Finally read complex, so it will return nothing, you can write void or integer or anything or you need not write anything. In general, if you want to make it complete and that to indicate very clearly that it does not write anything, return anything you write void.

Now coming to read complex. Can you tell me what read complex will take? Will it take a complex as argument? It will take the address of a complex number as argument. So an address of a complex number can be written as pointer to a complex. If you remove the variable name, all you left in the structural definition is like this. Suppose you want to write down a complex declaration, you write this.

Suppose you want to write down p is a pointer to complex number, then how do you write? This means this is, p is an address of a complex number. This is the address, this is a variable alright but it will take the address of a complex number. So its definition is complex star, the address pointer to a complex number. So these are the three definitions that we will have and now we are left to write down these routines. Complex, sorry add complex, it will take in two arguments complex a and complex b. You could have also written a b and written here complex a comma b in that same way, we just wrote it out this way.

(Refer Slide Time: 11:42)

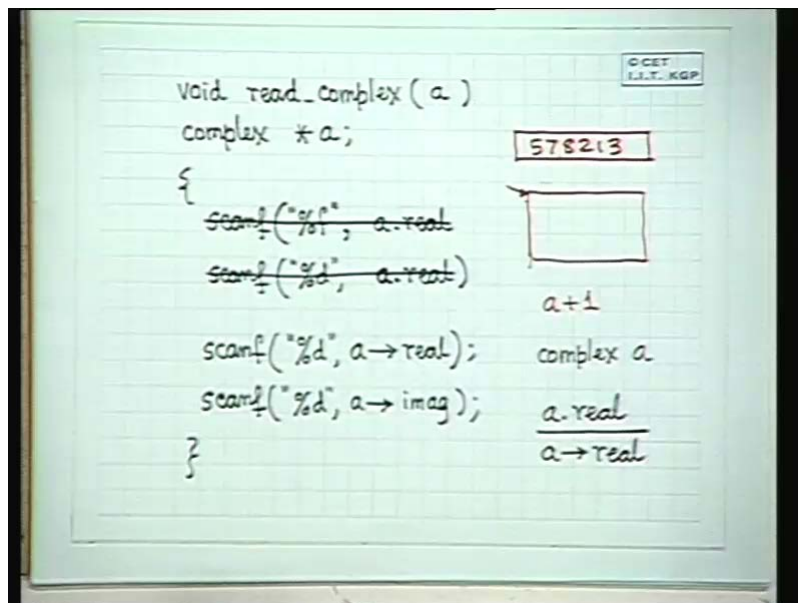


```
complex *add_complex (complex a,  
                      complex b)  
{  
    complex temp;  
    temp.real = a.real + b.real;  
    temp.imag = a.imag + b.imag;  
    return (temp);  
}
```

The image shows a hand-drawn code snippet on a piece of paper. The code defines a function named 'add_complex' that takes two 'complex' type arguments, 'a' and 'b', and returns a 'complex' type pointer. Inside the function, a local variable 'temp' of type 'complex' is declared. The real and imaginary parts of 'temp' are calculated as the sum of the real and imaginary parts of 'a' and 'b' respectively. Finally, the address of 'temp' is returned. A small logo in the top right corner of the paper reads 'CET I.I.T. KGP'.

And what would you do? You would have to declare some place where the real and imaginary parts would get separately added, so we require a temporary variable var to store this. Either you could do it this way, so you could declare a complex temp and write temp dot real dot, for a structure this is the structure name. Isn't it? For a structure name we use this is equal to a dot real plus b dot real temp dot imaginary is equal to a dot imaginary plus b dot imaginary and return temp. So, the intension of showing this is that how we have used a complex type is just as a pure type definition and inside the function, we have computed the value because now inside the function you need to know the structure of the variable. And using that structure we have computed the function. Anybody else who uses the complex type and wants to add two complex numbers will only call this routine, will need not access the internal portions of this structure. Write complex is obvious. Now let us come to read complex.

(Refer Slide Time: 17:43)



Sorry, this will be a and a will be a pointer to a complex number or you could have written here `complex star a` also. Here it is, a is a pointer to a complex number just like you did in `int star a`, when you did swapping of two numbers. Now let us note, a is a variable, alright. This is a variable whose, now in order to understand this let us be very clear. This is a variable. So a variable will get a storage location and the content of this variable is what? Suppose the content is 5 7 8 2 1 3.

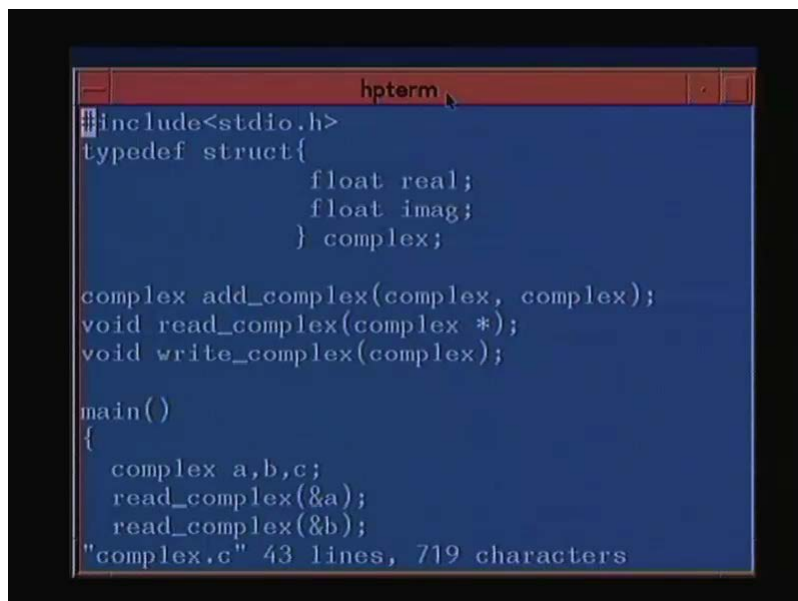
This content is the address of a complex number. So this content is the pointer to a complex number. So if there is a complex number which starts from 5 7 8 2 1 3 then this variable a will contain this location. And if you write a plus 1 this will jump by the size of this complex number. The increment will be made according to the size of this complex number, alright.

So if you write, if the complex number is of size 2 integers which is say 8 bytes then a plus 1 will increment it by 8 bytes. So if you have a variable which points to an integer, incrementing that variable will increment it by that many bytes. If it points to a character it will increment it by that many bytes. If it points to a structure, it will increment it by the size of that structure. So we must first remember that when you declare this complex star a, this is the variable but data according, for this is only here this will come as an argument.

Now we have to read in two numbers. Scanf, let us read in the first one. We would normally, in our normal mode we would read like this a dot real, sorry this is % d, if we define it to be integer. But this will not work that is this a dot real works when a is of type complex, if a is of type complex like this then a dot real is to access this number. But since a is a pointer to a complex number, there is a difference between being a complex number. If it was a complex number then its address would be the value of this. The address of a is different from the value of a here.

Since a is a pointer to a complex number, a will also have its address and value, its value is a pointer to a complex number. Whereas when you write complex a, the address of a is a pointer to, is the address of the complex number. So when you write this a dot real, this we have already got the address. in order to do it for such declarations, we have to write... this will take the content and from that content whatever value it gets, now it has got the address. So there is a difference between a dot real and this is used when you have already got the address and this is used when you have got a pointer to the address, alright. So you will do a scanf of this, a scanf and return and automatically inside a this values will be modified. So is that clear?

(Refer Slide Time: 18:02)



```
hpterm
#include<stdio.h>
typedef struct{
    float real;
    float imag;
} complex;

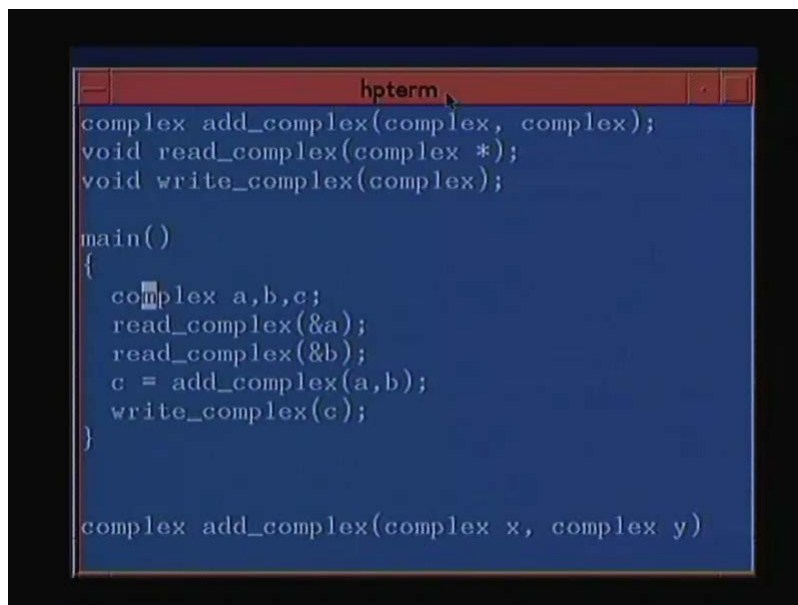
complex add_complex(complex, complex);
void read_complex(complex *);
void write_complex(complex);

main()
{
    complex a,b,c;
    read_complex(&a);
    read_complex(&b);
}
"complex.c" 43 lines, 719 characters
```

So let us now see the final version of the program on the machine. So here we have got the include definition and look at it. Outside the main, we have defined this because this declaration is a global declaration. This declaration is a global declaration outside the main and before the main we have used these function and we have also defined the structures of the functions outside the main. This is because the structures, these functions can be used in the main, they can be used at other places and each can use the other people also.

So in order to define the structures of the functions ahead, you can define them by just defining their type declaration. We have made no variable declarations, please note. Here type def I have taken float real and float imaginary and called it complex and I have defined complex add, add returns complex, add complex. First argument is of complex type, second argument is of complex type. Void read complex, it has got only one argument which is a pointer to a complex type and void write complex, it has also got one argument which is a complex type. So there is a difference between giving an argument which is a pointer to complex type and a complex type.

(Refer Slide Time: 19:40)



```
hpterm
complex add_complex(complex, complex);
void read_complex(complex *);
void write_complex(complex);

main()
{
    complex a,b,c;
    read_complex(&a);
    read_complex(&b);
    c = add_complex(a,b);
    write_complex(c);
}

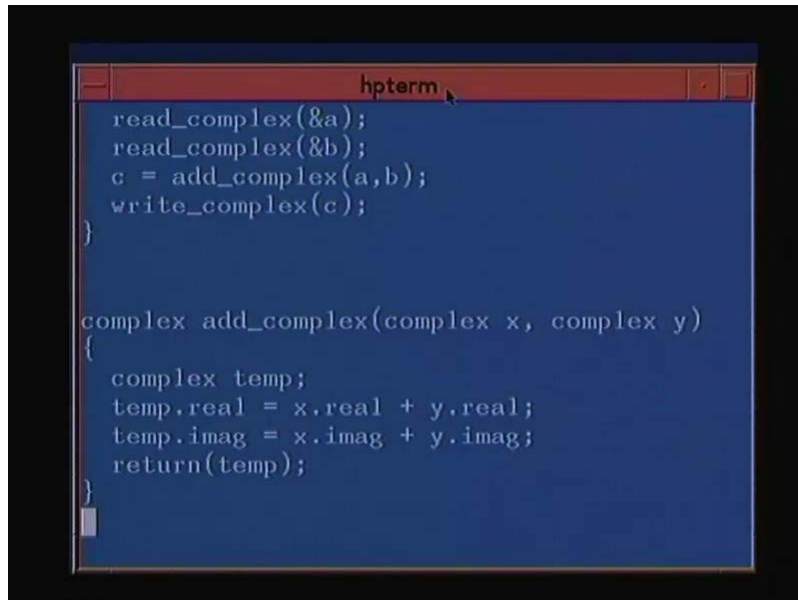
complex add_complex(complex x, complex y)
```

And in the simple main program we have declared three complex numbers a b and c because this type definition is global it can be used anywhere in any function. On the other hand if you have defined this inside this main only, you would have to redefine it in the functions. And then read complex and a, just like if it were an int. What I wish to stress here is that we have defined a new type and you can, you will use it just as you use the other types and a.

Again another one read complex and b, c is equal to... Now this returns a complex type and it's assigned to a complex number a b and then write complex c. So we have defined a new type called complex and we have used this type and assignments can be made

between the elements of the same type structure directly. So add complex is just as we discussed, complex temp, temp dot real equal to x dot real plus y dot real temp equal to imaginary, temp dot imaginary equal to x dot imaginary plus y dot imaginary, return temp.

(Refer Slide Time: 20:42)

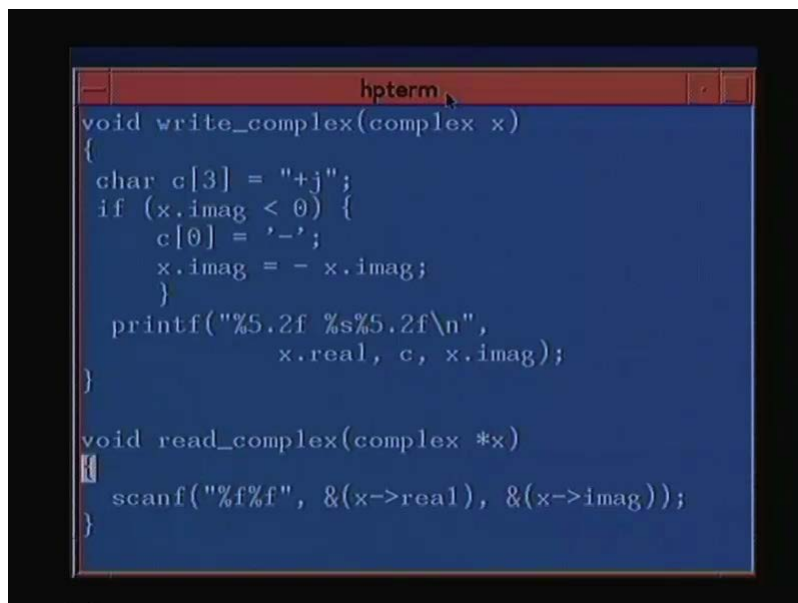


```
hpterm
read_complex(&a);
read_complex(&b);
c = add_complex(a,b);
write_complex(c);
}

complex add_complex(complex x, complex y)
{
    complex temp;
    temp.real = x.real + y.real;
    temp.imag = x.imag + y.imag;
    return(temp);
}
```

Write complex has been made a little colorful in order to define that plus j. So write complex x is of complex type, we define a character array of three characters and we assign that is a string of three element and c is assigned, initialized to plus j.

(Refer Slide Time: 22:04)



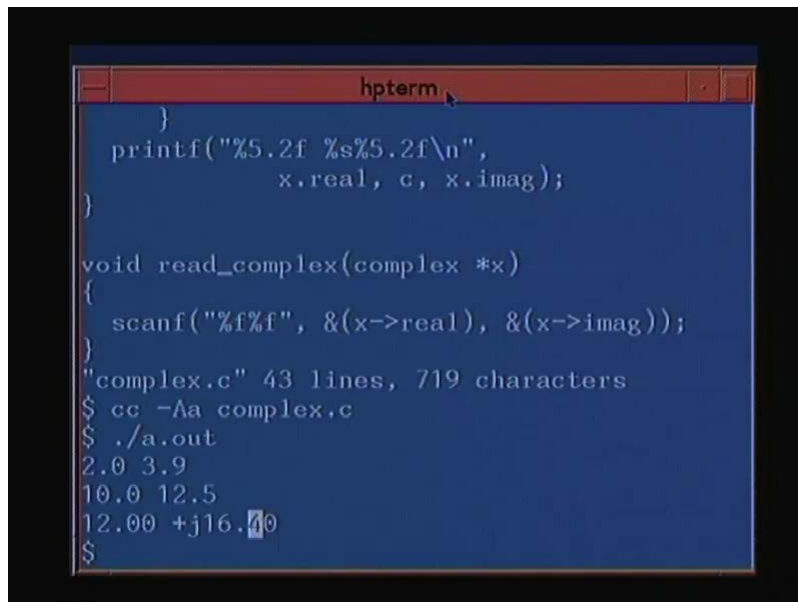
```
hpterm
void write_complex(complex x)
{
    char c[3] = "+j";
    if (x.imag < 0) {
        c[0] = '-';
        x.imag = - x.imag;
    }
    printf("%5.2f %s%5.2f\n",
           x.real, c, x.imag);
}

void read_complex(complex *x)
{
    scanf("%f%f", &(x->real), &(x->imag));
}
```


And we say if x dot imaginary is less than 0, $c[0]$ is made minus, alright and x dot imaginary is equal to minus x dot imaginary, just do and we printed it. To print it, we print out x dot real, this c which prints that plus j and x dot imaginary. And complex, read complex as we mentioned is defined as a pointer to a complex number and scanf **am sorry I made a mistake here** this x real, this part this will give the value of the real number. Isn't it? x pointer dot real will give you the value of the real number. If you want the address of the real part only and you have to use scanf. In scanf again we have to pass the address. In scanf we have to pass the address, if you want to scanf of an integer x , you have to pass the address of that integer.

If you want to scanf of a floating point number, you have to pass the address of the floating point number. On the other hand x pointer dot real will give you the content of that. If you want to get the address of that location, again you have to give the AND. here x pointer dot imaginary will give you the content and AND of that will give you the address because scanf requires giving the address. I made a mistake when I wrote down the program on the piece of paper it should be AND out there. So is that okay? So we have given two numbers two point, the real part is 2.0, imaginary part is 3.9 of one number because in the program, we first read in two numbers.

(Refer Slide Time: 24:08)



```
hpterm
}
printf("%5.2f %s%5.2f\n",
       x.real, c, x.imag);
}

void read_complex(complex *x)
{
    scanf("%f%f", &(x->real), &(x->imag));
}

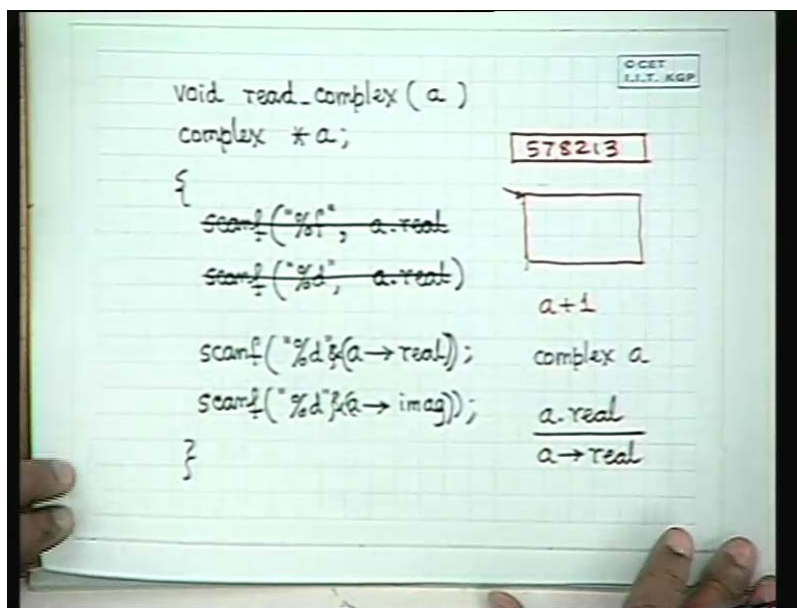
"complex.c" 43 lines, 719 characters
$ cc -la complex.c
$ ./a.out
2.0 3.9
10.0 12.5
12.00 +j16.40
$
```

So we are just reading in the two numbers, 10.0 and 12.5 and we get 12.0 plus j 16.40, whatever because the format is 5.2, therefore it will give it in this format. So just to have a look again at the program, you first define the type and define your new type called complex and use that type throughout in complex declaration the etc, etc. This will lead towards what is called encapsulated. Here we have encapsulated the data and now we are trying to put in functions which uses this data. And once we are able to encapsulate both the data and the function, we will get what is called a data structure.

Suppose you could define a function where the main was manipulate complex, there was no main. This was the type definition and these were function that you could use. And you would give it to somebody to say that here I am giving you a declaration for complex numbers and I am giving you the functions for complex numbers. So this is called a data structure for complex numbers, a data type and the functions to manipulate on that data structure on these data types.

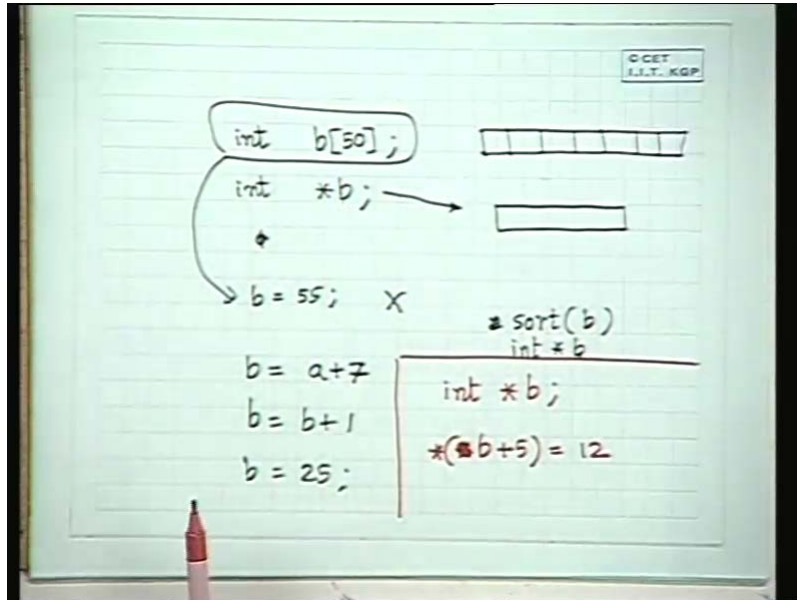
So coming back to, just first the correction. this should be AND because a dot, like a dot real gives you the value of the real number, a pointer dot real also gives you the value of that number, to get the address since scanf requires address, you have to give AND.

(Refer Slide Time: 25:49)



Now we need to understand something else. If I define int b [50], I define an array b of 50 elements and b is the address of the array, the start location of the array and we have seen the values of addresses in programs. If I define this then I define b to be a pointer to an integer.

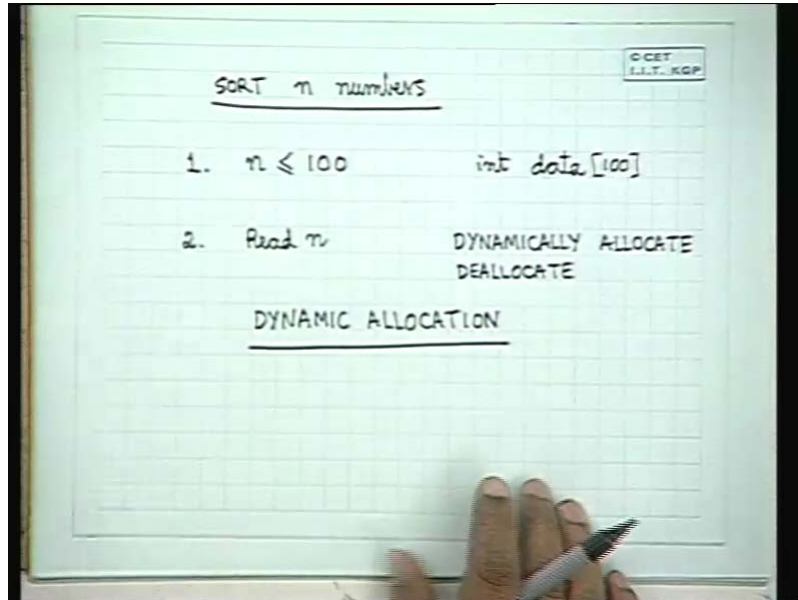
(Refer Slide Time: 29:30)



Now this `b` in the array declaration cannot be modified, you cannot write `b` is equal to 55 inside the program, alright. This `b` this is not allowed but this one is a variable. This `b` is a variable and this `b` can be changed, this `b` can be assigned, `b` is equal to `a` plus 7, `b` is equal to `b` plus 1, `b` is equal to 25, all these things can be done because this is a variable which is of type pointer to an integer. So, when this is declared, 50 integers are allocated, 50 elements each of the size of an integer are allocated and `b` is the address of this and it's a constant. When this is declared, one element of size of a pointer that is of size whose content can be an address. So its size will be a size of a pointer to an integer.

What is a pointer to an integer? It is the content of the memory location, so its size will be the content of the memory location and only one such is allocated. When we used it inside a function say `sort`, we allocate it only `b` but we used the content, we got the value from here automatically passed on here. And this was used, we did not allocate more than that data but we knew that from here the array starts and that array is stored in the memory, so we were able to use it. But here if you start writing down for this declaration, suppose I declare `int *b` and I write down content of `b` plus 5, okay content of `b` plus 5. `b` plus 5 is the fifth element after this is equal to 12, don't think that the array, there is an array declaration here automatically of 5 elements. You have not declared any array here. You will be modifying the content of the location `b` plus 5, whatever it stores and wherever it stores. So you must be very careful about what is allocated in what declaration and what is a constant and what is a variable.

(Refer Slide Time: 32:30)



Now we will come to another question. Suppose I give you a problem in which I tell you, you have to sort n numbers. And the first thing I tell you is that n is less than equal to 100. Then obviously you will declare `int data[100]` but if I tell you read n and I don't know what it is. Then the problem is what to declare, what size will I declare 200, 300, 400, 900. Whatever you declare, there requirement may be more than that. So for that you need to dynamically allocate. Within the program I need to allocate data. So we need to allocate data within the program and this is called dynamic allocation, alright.

Now unless you have dynamic allocation in programming languages, it will become difficult for you to define many such data structures. For example if I define student records and you do not know and we have got insertion, deletion, etc then even if you know that it will be less than 5000 but on an average only 20 or 30 are used, then it is inefficient to declare a block of 5000 elements of data. We are uselessly keeping that data, that area of memory which is of no use to you.

So you can even efficiently use data, when you know that your requirement is always much much less than your maximum. And you can use and allocate only whatever you require. So you can dynamically allocate or de-allocate data. So in a programming language, in order to have more efficiency of implementation and use of space, you need to dynamically allocate and de-allocate data. And may be if required even for problems, where you will have to read in the size when required. So the next topics that we will declare, talk on is on dynamic allocation.