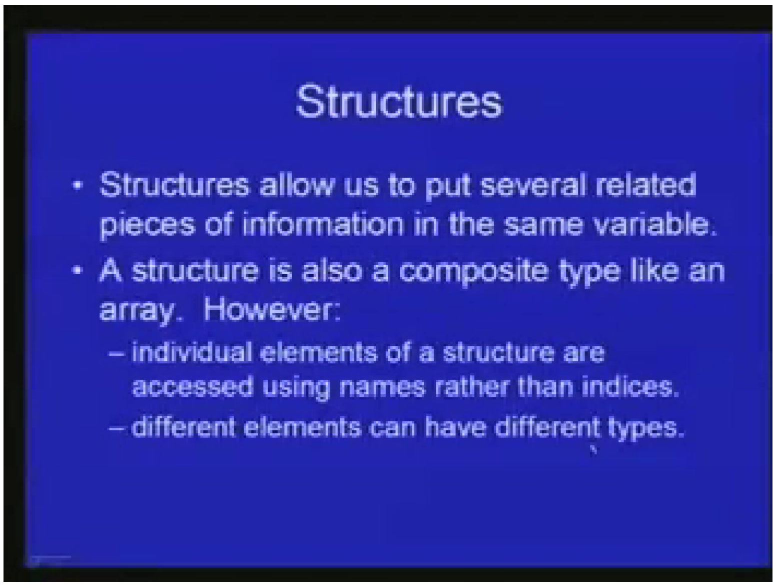
Introduction to Problem Solving and Programming

Prof. Deepak Gupta
Department of Computer Science Engineering
Indian Institute of Technology, Kanpur

Lecture No. # 23

In today's lecture, we will talk about very important and useful featured provide by the c programming language and that is notion of structure.

(Refer Slide Time: 00:23)



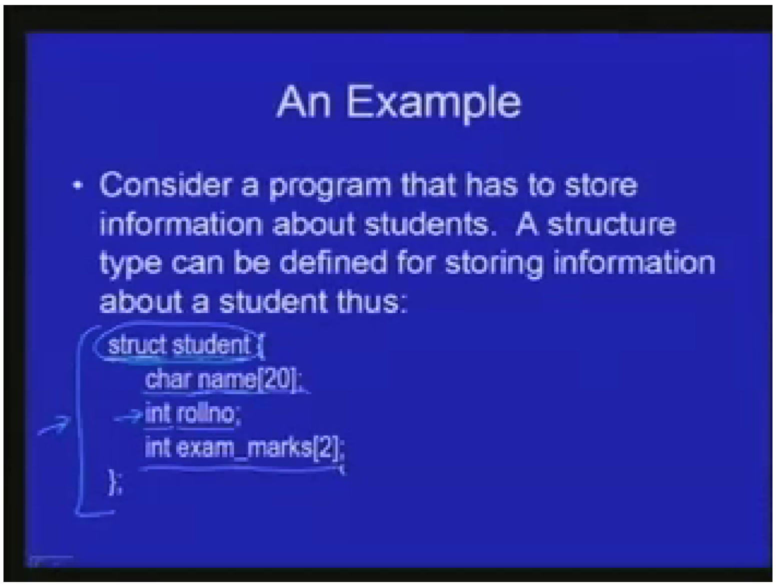
Structures

- Structures allow us to put several related pieces of information in the same variable.
- A structure is also a composite type like an array. However:
  - individual elements of a structure are accessed using names rather than indices.
  - different elements can have different types.

So, essentially the use of structures says the using structure, they can put several pieces of related information at the same place in a single variable instead of spreading number on program in different variable. And so as you can see the structure is also a composite type because it is a structure is comprised of smaller units of data; however, the difference between the array and structure is the following an array is also a composite type like structures except that all element of an array are of the same type whereas, with structures we can have different element of different types.

The other difference between arrays and structures is that in arrays the individual element of the array are accessed using numerical index which is an integer, whereas the different element in a structured accessed by different name you can give different names to different elements and that makes it very useful. And (( )) other difference is that the different (audio not clear) element structured in different type where as the different element of the array all have to have the same type.

(Refer Slide Time: 01:32)



An Example

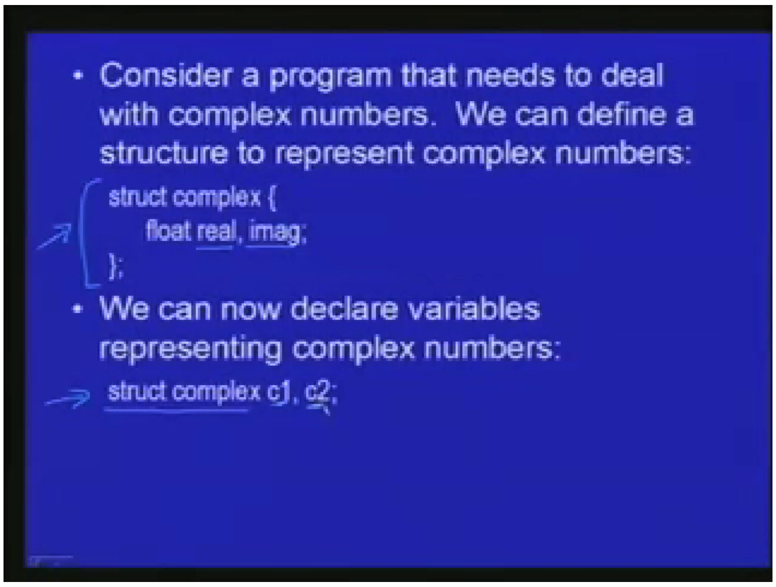
- Consider a program that has to store information about students. A structure type can be defined for storing information about a student thus:

```
struct student {
    char name[20];
    int rollno;
    int exam_marks[2];
};
```

So, let us consider a simple example where we might found to use structures. So, suppose we are writing a program there is maintaining student record and so on. And so we need to maintain various pieces of information about every student. So, we could define a structure for storing information about the student such a declaration might look like this. So, this entire declaration defines a new type whose name is struct student. The keyword struct indicates that this is the structure that we are defining and the word student is a name for the structured type and then we will (( )) follow the declaration of the various components of the structure and this will have arbitrary type.

So, the fourth element in this case is name - that is the name of the element which is store the name of the student and the type of this element is an array of 20 characters that is why that we can have arbitrary type inside the structure so the element can really have any other type. So, next element of the structure is equal to roll no which is equal to type n. So, this will probably be used store the roll number of the student, and finally we have an array of two integers for storing may be the exam marks of the particular student.

(Refer Slide Time: 03:01)



- Consider a program that needs to deal with complex numbers. We can define a structure to represent complex numbers:

```
struct complex {
    float real, imag;
};
```

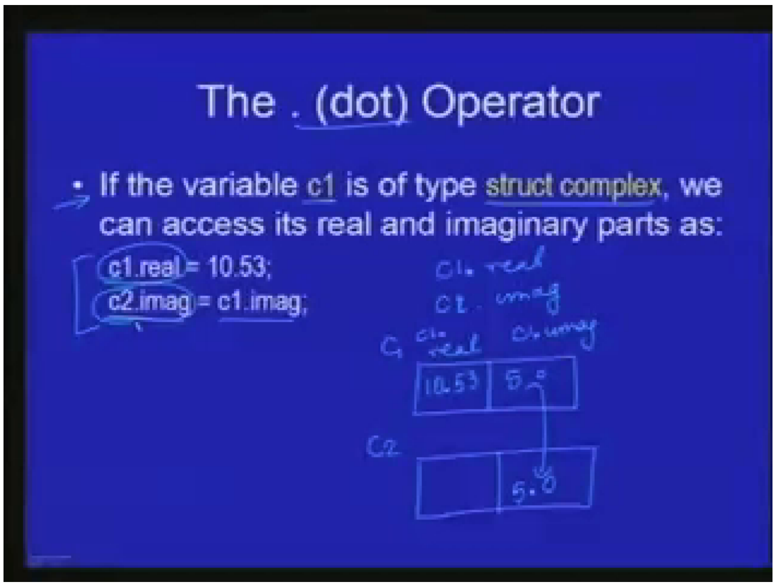
- We can now declare variables representing complex numbers:

```
struct complex c1, c2;
```

Before we look at such complex structure, let us first look at some examples involving simplex structure. So, suppose we need to write a program that deals with complex numbers; that you already know the complex number has two parts a real part and imaginary part, and to present the complex number essentially we need to store both these parts. So, the ideal representation of a complex number would be a single variable stores both these parts. And so therefore, we declare a struct complex, which define a new type for complex number and you can see that there are two elements or two fields of the structure with names real and imag and both are of type float. So, this this complex number will have two parts - a real part and imaginary part which are both floating point number.

So, once you have declaration like this, we can now declare as many variables of this type as we want. So, this thing, this declaration does not declare any new variable which has declared a new type. This is the first time that we are seeing such a such a notion, but once we have declare such a type where types that complex now available to us and we can declare any number of variables of this type. So, for example, this declaration declares two variables c 1 and c 2 which are both of type struct complex which means that c 1 as well as c 2 will both have their own real parts and their own imaginary parts.

(Refer Slide Time: 04:42)



The . (dot) Operator

- If the variable c1 is of type struct complex, we can access its real and imaginary parts as:

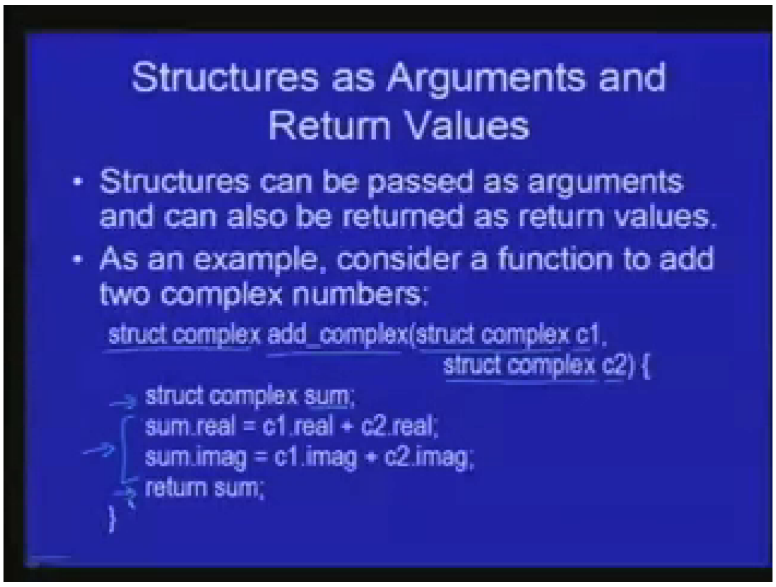
```
c1.real = 10.53;
c2.imag = c1.imag;
```

c1. real
c2. imag
c1
c1. real
c1. imag
10.53
5.0
c2
5.0

Now, we use the dot operators to accept individual fields of a structure, so obviously, where given structure we need to be able to manipulates the different fields of that structure. And the dot operator is used to obtain an element of a structure given the structure itself. So, as an example let us assume the same struct complex declaration. So, let us say c 1 is a variable which we have declared to be of type struct complex and now we can access its real and imaginary parts as shown. So, essentially C 1 dot real when denote the real part of the structure C 1 and C 2, C 1 dot imag will denote the imaginary part of the structure. So, essentially C 1 is a variable which compares of two boxes both large enough to whole integer and the names of these boxes you could think of has been C 1 dot real and C 1 dot imag and these assignment statement show are these expression can be used on either side of an assignment.

So, C 1 dot real is equal to 10.53, both essentially assigned value of the real part of the structure C 1 to 10.53 and C 2 dot imag is assigned C 1 dot imag will copy the value of C 1 dot imag - that is the imaginary part of the variable C 1 to the imaginary part of the variable C 2. So, C 2 is also lets assume is of a similar type, so this assignment statement will copy this value to this value. So, whatever which happen to be let say 5.0 and this also becomes 5.0. And as you can see the types of these expressions would be the types declared for these digits. So, we have declared real to be of type float and imag also to be of type float, and therefore, C 1 dot real will have the type float and two dot imag will also have the type float.

(Refer Slide Time: 06:59)



Structures as Arguments and Return Values

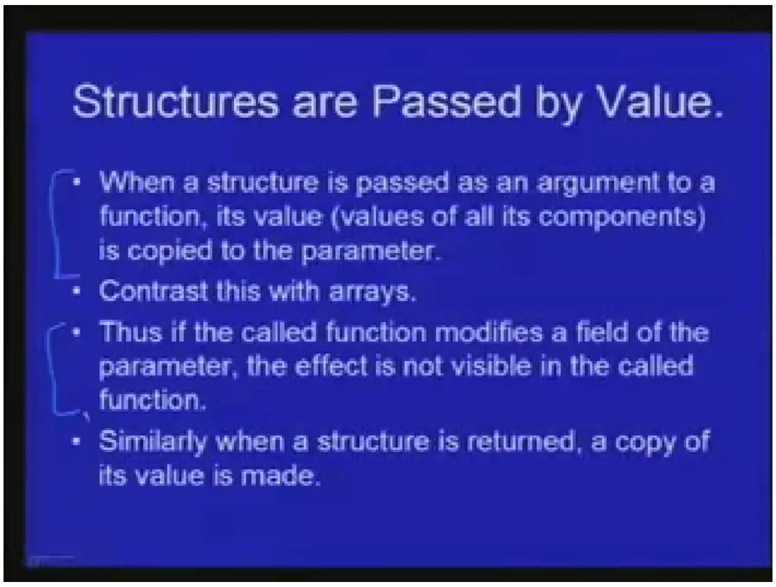
- Structures can be passed as arguments and can also be returned as return values.
- As an example, consider a function to add two complex numbers:

```
struct complex add_complex(struct complex c1,
                           struct complex c2) {
    struct complex sum;
    sum.real = c1.real + c2.real;
    sum.imag = c1.imag + c2.imag;
    return sum;
}
```

Now structures are full fledged types, and we can do all the useful kinds of operations that we can do with other types. In particular, we can past structures arguments and we can also write functions which return structures as return value. So, as an example for passing structure arguments and returning structure return value. Let us consider a function to add two complex numbers. So, we have a function add complex which supposed to add two complex numbers there are two parameters c 1 and c 2; those of type struct complex and the return type is also struct complex. Now how do you add two complex numbers - that is very easy you just add up serial parts and the imaginary part separately. So, that is what this function does it declares a local variable called sum of type struct complex and sum dot real is assigned values c 1 dot real plus c 2 dot real, and sum dot imag is assigned value c 1 dot imag plus c 2 dot imag, and finally the value of sum is returned.

So, here is the function add complex to add the complex numbers which has two parameters c 1 and c 2, both are of type struct complex that is c 1 and c 2 are both complex numbers. And the return type is also struct complex - that is we are going to return another complex from this function. So it declares a local variable called sum which is also type struct complex, and now to add c 1 and c 2 all we need to do is to separately add the real and imaginary parts and that is what this two (( )) the variable from after this two assignments, we will have the we will represents the complex numbers c 1 plus c 2. So, sum dot real is c 1 dot real plus c 2 dot real sum dot imag is c 1 dot imag plus c 2 dot imag, and finally function returns the value of sum which is c 1 plus c 2.

(Refer Slide Time: 09:05)



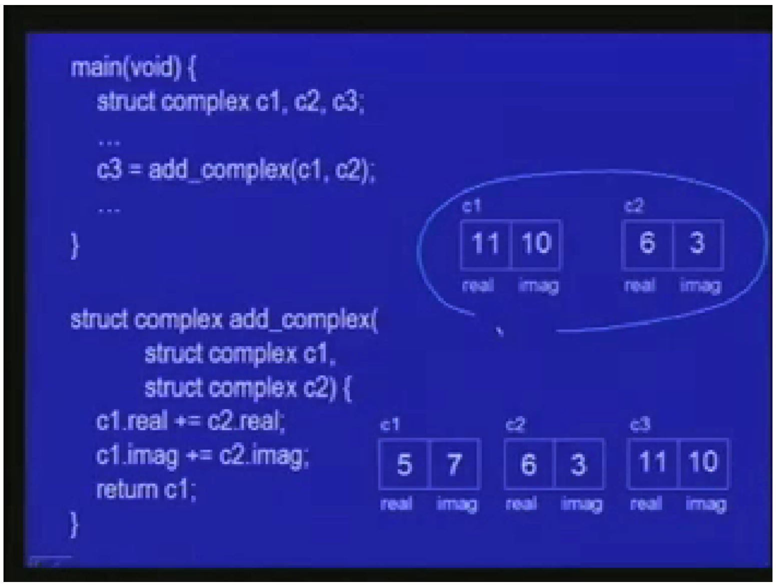
Structures are Passed by Value.

- When a structure is passed as an argument to a function, its value (values of all its components) is copied to the parameter.
- Contrast this with arrays.
- Thus if the called function modifies a field of the parameter, the effect is not visible in the called function.
- Similarly when a structure is returned, a copy of its value is made.

Now one thing that we need to understand about structures suppose to arise or the structures are passed by values that is when I declared a structure variable that variable represent the structure itself and not the address of that structure. And therefore, when a structure is passed to a argument to a function its value is copied to the parameter and its value is nothing but the value of all of its component are all of its yield. So, the values of all its yield are copied to the corresponding spaces in the parameter. And this is different than for arrays, for arrays, as for arrays as you know when you pass an array what is actually getting passed is just starting letters of the array and not these values of the elements of the array. And therefore the implication of the fact is that is call function modifies the yield of the parameter which happens to be structured, and the effect is not visible in the called function.

We will see an example using a variant of the add complex function that we just saw and similarly when a structured is returned from a function a copy of its value is made and that is that copy is what is returned to the calling function.

(Refer Slide Time: 10:24)



```
main(void) {
   struct complex c1, c2, c3;
   ...
   c3 = add_complex(c1, c2);
   ...
}

struct complex add_complex(
         struct complex c1,
         struct complex c2) {
   c1.real += c2.real;
   c1.imag += c2.imag;
   return c1;
}
```

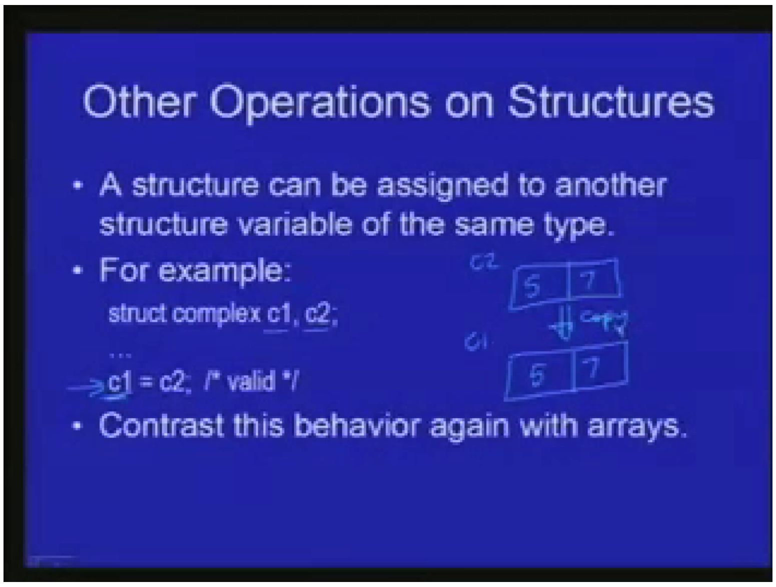
So, let us see an example of these struct using a slightly different version of the add complex function that you just put, here you do not have a local variable come and what is being done is the sum is being computed in the parameter c 1 itself. So, c 1 dot real plus equal to c 2 dot real; that means, the value of c 2 dot real is added to the existing value of c 1 dot real, and similarly c 2 dot imag is added to c 1 dot imag, and finally the value of c 1 is returned. And let us say this is how this function is getting called from the main program. We have three local variable c 1, c 2, c 3 of main, and we call add complex c 1, c 2, and the return value is told in the variable c 3.

So, let us assume that this plan this function as add complex is called this is the (( )) three variable c 1, c 2, c 3 of the main function looks like. c 1 has the real part and the imaginary part and similarly c 2, c 3. And let us say c 1 is 5 plus 7 i and c 2 is 6 plus 3 i and c 3 is uninitialized. So, when the function is called as you know what happened first is that space is created for the parameters of the called function which in this case happened to be c 1 and c 2. So, this space for parameter c 1 and parameter c 2 of the function add complex has been created. Next what is going to happen is that the values of the arguments are copied into the corresponding parameters, so the arguments are expression c 1 and c 2 when c 1 is the value of it; it evaluate the entire structured 5 and 7, and similarly when c 2 is evaluated it evaluate structure containing 6 and 3.

So, when this copy happens this structured value is copied to c 1 of add complex and this value of c 2 is copied to the parameter of c 2 of add complex. So, c 1 gets value 5 plus 7 i and c 2 also c 2 gets the value 6 plus 3 i. Now one this statement is executed c 1 dot real is plus equal to c 2 dot real we are executing within the function add complex, and therefore, c 1 refers to this c 1 and not to this c 1 and similarly c 2 refers to this c 2 and not to this c 2. So, this statement will modify the real part of this c 1 which is this value. So, this value will become 5 plus 6 that is eleven and the imaginary part becomes 7 plus 3 which is 10.

So, in the next step c 1 dot real becomes eleven and in the next step c 1 dot imaginary becomes 10, and finally the value of c 1 is returned. So, when this value of c 1 is returned this value will be copied in to the variable which is going to whole the return value in this case c 3, and so this copy is made and now the function add complex has returned, so as you know the space for its local variable and parameters will be removed or reallocated. So, these variable will now c 2 exists and c 3 has become the sum of c 1 and c 2, but the important point is that these changes to c 1 did not affect the value of the variable c 1 in the calling function.

(Refer Slide Time: 14:04)



Other Operations on Structures

- A structure can be assigned to another structure variable of the same type.
- For example:

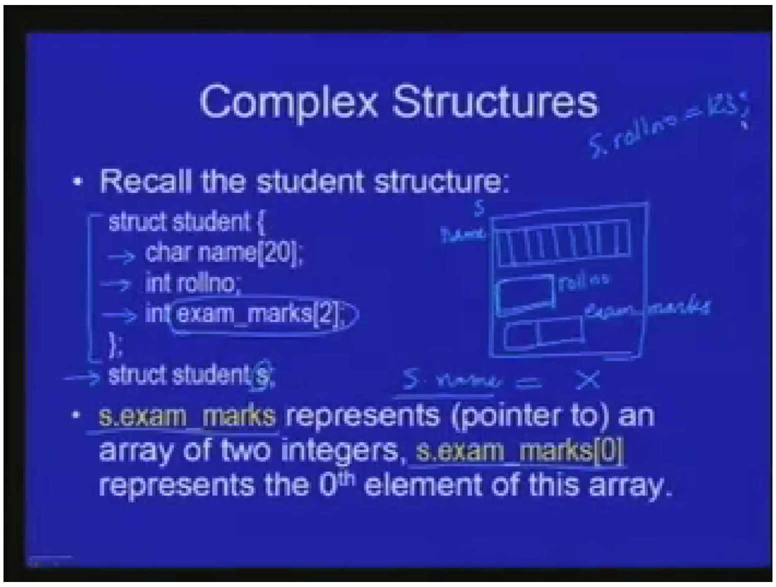
```
struct complex c1, c2;
...
c1 = c2; /* valid */
```

- Contrast this behavior again with arrays.

Now let us look at some other operations on structures, and the basic operation that we can do is that structure can be assigned to another structured variable of these same type. So, let us say again c 1 and c 2 are variables of type struct complex that is they are complex numbers. So, this assignment is valid note that the left hand side is the variable which is of structured type and the right hand side is is also an expression which has the same structured type. So, c 1 assigned c 2 is valid and whatever is the value of the complex number c 2 that gets copied into the complex number c 1, again contrast the arrays with the arrays n array variable cannot appear on the left hand side of an of an assignment expression because an array variable has an expression denote the starting address of the array and not the contents of the array itself.

And the starting address of an array is 6, it is assigned by the complier that cannot be changed, and therefore an array variable cannot appear on the left hand side of an assignment. Whereas for in the case of structures a structure variable represent the contents of the structure and not its starting address and therefore, it can appear on the left side of an assignment like this. So, if c 2 happen to have the value 5 plus 7 i then after this assignment statement, c 1 will also have the value 5 plus 7 i and this value that is copied here.

(Refer Slide Time: 15:36)



Complex Structures

s.rollno = 123;

- Recall the student structure:

```
struct student {
  char name[20];
  int rollno;
  int exam_marks[2];
};
struct student s;
```

s.name = X

- s.exam_marks represents (pointer to) an array of two integers, s.exam_marks[0] represents the 0th element of this array.

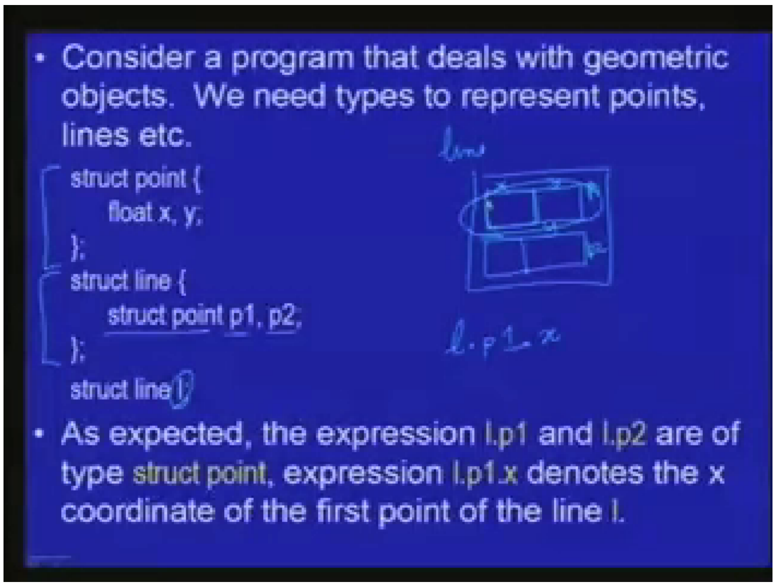
s
name
rollno
exam_marks

Let us now look at some more complex structures and the simple structure that we have seen. So, let us recall the student structure that we already saw the struct student contains three fields - the name of the student, the roll number, and the two exam marks.

And let us assume that we declared variable s of type struct student. Now what is s dot exam marks represents? s dot exam mark represent this component of the structure s which is an array. So, therefore, s dot exam mark represent really a pointer to an array of two integer, and so s dot exam mark 0, and so s dot exam mark 1 will represent the two element of this array. Note that s dot exam mark itself is an array, and therefore as for array it represents the starting address of the array rather than the contents of the array. So, if you now happened to pass s dot exam mark is an argument to a function then it is the starting address of this array which is being passed. So, we have to look at the structure pictorially this is what it looks like, it has an array of 20 character which is filed called name and then it has a single integer called roll number and then it has a array of two integers called exam marks.

And f is the name of the entire structure that the variable, and therefore s dot name is an array type it it is an array of twenty character and therefore, this variable itself represents the starting address of this array. This expression s dot name represent starting address of this array and again it cannot appear on the left side of an assignment because the starting address of this array is fixed, but s dot roll number can assign on the left side of an assignment because this represent just an integer variable this particular variable. And so this could be assigned to the value 123 or whatever as it happen to be and… Similarly s dot exam marks represents an array and therefore, the starting address of that array, and so the user rules for array apply, whereas for the s dot roll number the rules for integer (( )) apply.

(Refer Slide Time: 18:00)



- Consider a program that deals with geometric objects. We need types to represent points, lines etc.

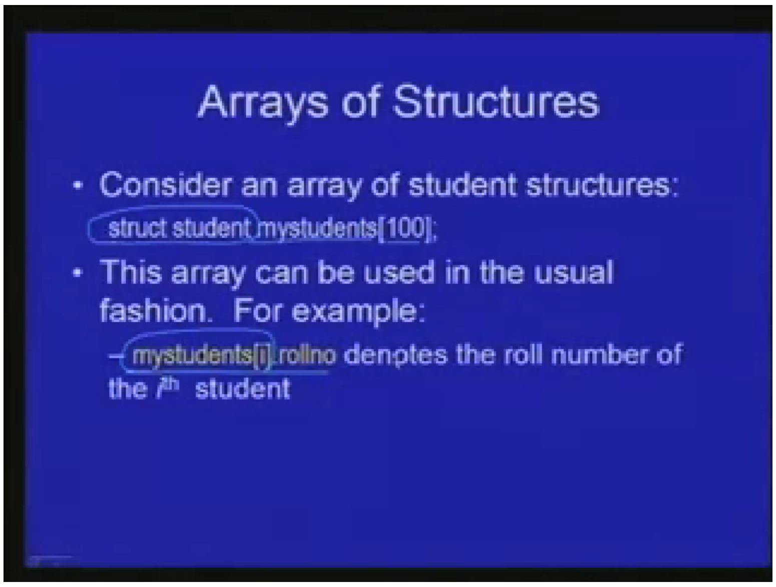
```
struct point {
    float x, y;
};
struct line {
    struct point p1, p2;
};
struct line l;
```

- As expected, the expression l.p1 and l.p2 are of type struct point, expression l.p1.x denotes the x coordinate of the first point of the line l.

Now we can also have structure in size structure. So, we can have one structure type as a field or a component of another a more complex structured type. As an example consider a program that deals with geometric objects, so for this program we need to represent we need types to represent points, lines and so on and so forth. So, we quote represent a point by just its x y coordinates is declaration for a structure which represents a point.

Now a line perhaps represented by its two end points or line segment are represented by its two end points which are both points. So, this structure contains two fields, p 1 and p 2 which are of type struct point. So, therefore, if you look at a pictorial representation of a line, it will have inside it two structures called p 1 and p 2. And these in turn contain two fields each which are called x and y, x and y. So, now, if we have to cleared n as a variable struct line then l dot p 1 represent this entire structure, and therefore l dot p 1 dot x represent the x part or the x coordinator of this particular point p 1 of the line l, and similarly l dot p 2 dot y will represent the y coordinate of the second n point of the line and so on and so forth.

(Refer Slide Time: 19:34)



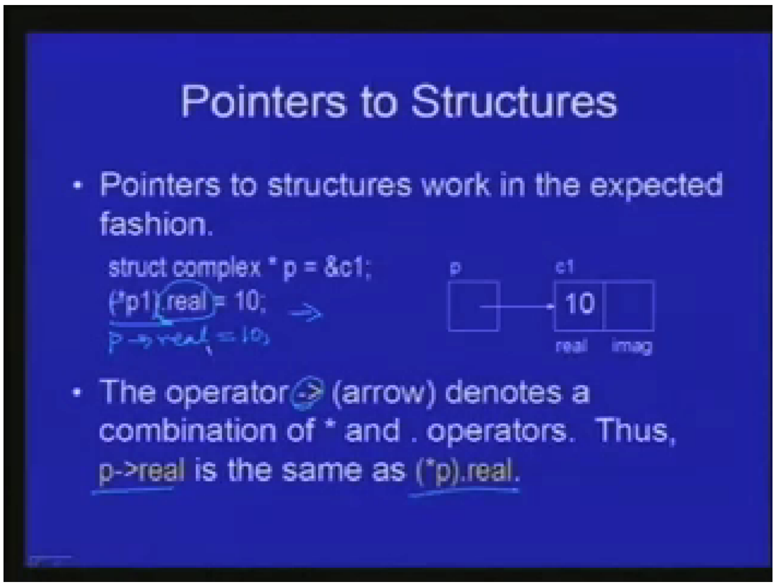
Arrays of Structures

- Consider an array of student structures:
  struct student mystudents[100];
- This array can be used in the usual fashion. For example:
  - mystudents[i].rollno denotes the roll number of the i^{th} student

They could also have arrays of the structure not only could we have structure containing arrays, but we could also have arrays of structures, and those structures themselves in turn could actually have other arrays within them. So, you you can see that know using these kinds of building blocks we can build some very complicated data structures. So, let us come back to the student structure that we were talking about earlier.

Now suppose a program needs to maintain a record of all the students in a particular class they probably need to maintain the student's records of the various students in an array. So, here we have declared an array called my student which is an size 100 and the type of each element of the array my student s struct student. So, therefore, my student is an array each of whose elements is a student structure, and so for example, if I have this question my student i dot roll number, so my student i the i eth element of the array my student which happens to be student structure and therefore, my student i dot roll number denotes the roll number of the i eth student in this array. And similarly my student i dot name would represent the name of the add student and so on and so forth.

(Refer Slide Time: 20:53)



Pointers to Structures

- Pointers to structures work in the expected fashion.

```
struct complex * p = &c1;
(*p1).real = 10;
p->real = 10;
```

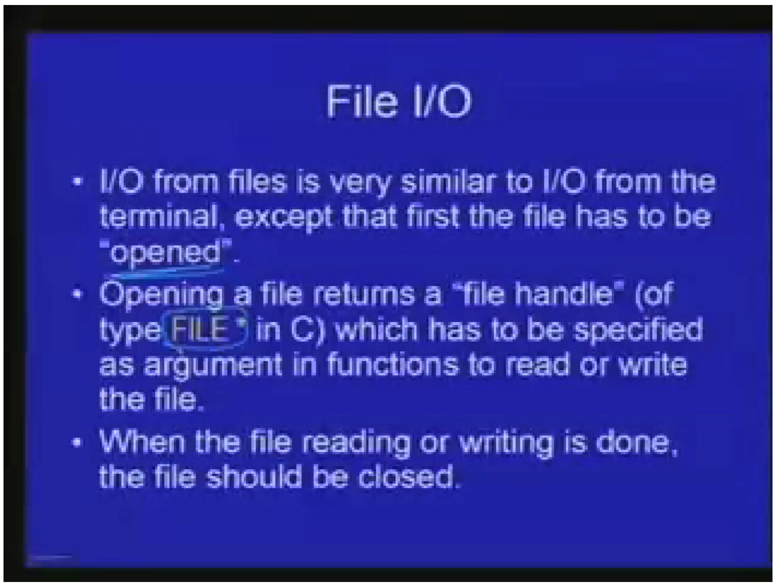
- The operator -> (arrow) denotes a combination of * and . operators. Thus, p->real is the same as (*p).real.

They can also off course have pointers to structures, just like we can have pointers to other kinds of variable, and these pointers to structures that work in the same way that pointers to other kinds of variable works. So, for example, suppose we have a variable c 1 of type struct complex and what we declaring here is a variable p, which is of type struct complex star and what that means is that p as a type which is a pointer to a struct complex, and now what we are doing is start p 1 dot real assigned 10. So, what does this mean which should be star p, not star p 1. So star, so p is a pointer to a structure, and so therefore, star p represents that structure, and therefore start p dot real represent the real the field called real of that structure.

So, in terms of a picture this what a really happening here the c 1 the variable c 1 is the structure where the real part and imaginary part and variable p is a pointer which points to a structure p 1 and so star p, so star p will represent this structure. And therefore, star p dot real will represent this field of this structure and that is getting assigned the value of 10. So, this assignment statement causes c 1 dot real to become 10 because p points to the variable c 1 and therefore, star p dot real represent the same thing as c 1 dot real. And that is very common operation that p represent a pointer to obtain the structure to which the pointer the point and then referred to a particular field of the structure.

So, these two operators apply together c provides a shortcut, so the operator arrow which consists of minus mark followed by greater than sign denotes the combination of the star and dot operator. And this is how we use it p arrow real is exactly the same as the star p dot real we could have written the same statement as p arrow real assigned 10 and off course we could have also used it on the left hand side on the right hand side of the assignment and so on and so forth.

(Refer Slide Time: 23:27)



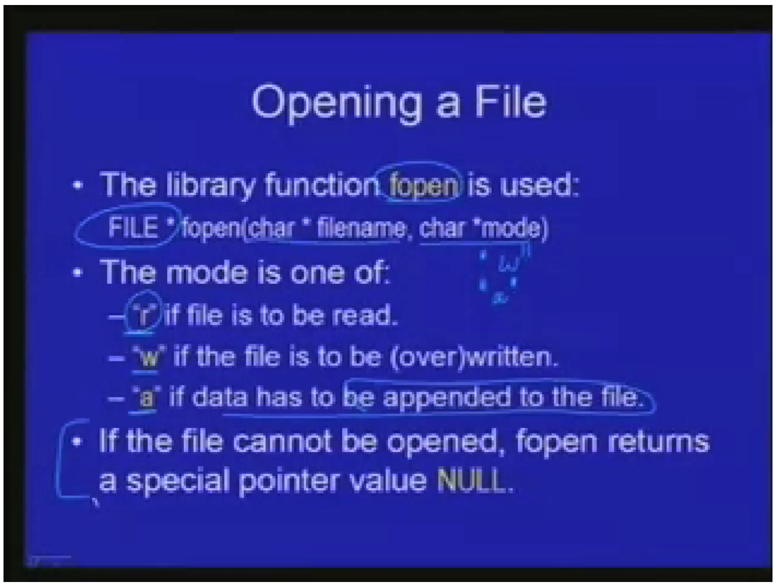
File I/O

- I/O from files is very similar to I/O from the terminal, except that first the file has to be "opened".
- Opening a file returns a "file handle" (of type FILE * in C) which has to be specified as argument in functions to read or write the file.
- When the file reading or writing is done, the file should be closed.

Having looked at structures we will rapid this lecture by briefly talking about input and output from file. So far we have been doing all are for input output from the terminal that is all the inputs that the program excepted the user have typed at the terminal and all the output again went back to a terminal. Now general in when we are writing programs that require large number of inputs and the output is also luminous we want the program to be able to directly read and write files instead of excepting all the input from the terminals, and that off course can be done and that is actually very similar to to the way input and output is done from the terminal will shortly see the (( )) function for doing that, but the basic difference between IO from a terminal and IO from a file is that before we can read or write a file from our C program.

We need to open it first and when we open a file, what we get is what is known as file handle. A handle for a file or in C it is also called a file pointer. The type of the file pointer, file handle in C is file star, file is a type which is declare, which is defined by the standard IO library of C and file star is the type for the file handle which is returned to us when we open a file. And now once we have once we are open the file and obtain a file handle for the file then we can read or write to the file depending on how the file was opened, and whenever we want to read or write the file we need to specify the file handle for the file instead of the file name for the file in every call. When we open the file that is only time that we really specify the file handle for the file name for the file, and finally when the program is finished reading or writing the file it should close the file again by specifying the file handle. We will see the function to do all this in the next slide.

(Refer Slide Time: 25:32)



Opening a File

- The library function fopen is used:
  FILE *fopen(char * filename, char *mode)
- The mode is one of:
  - 'r' if file is to be read.
  - 'w' if the file is to be (over)written.
  - 'a' if data has to be appended to the file.
- If the file cannot be opened, fopen returns a special pointer value NULL.

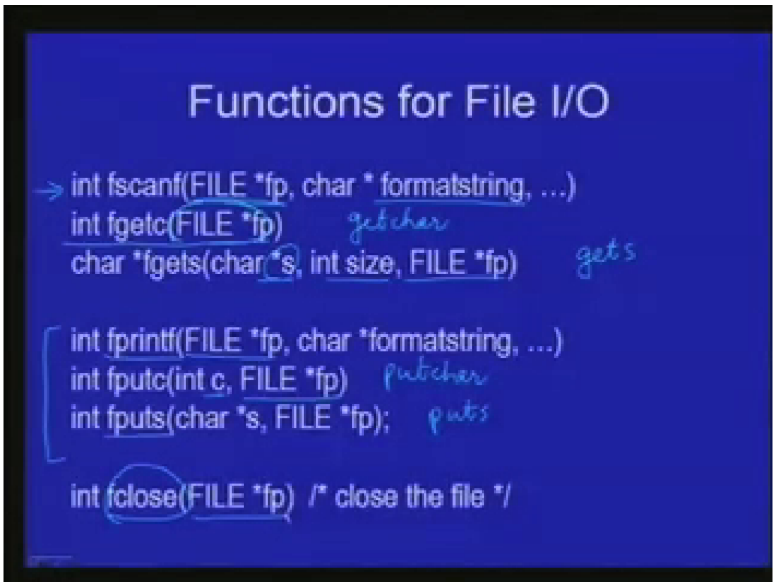
So, first opening a file is easy. You need to use the fopen function of the standard library. There are two arguments - the first is the string which is the name of the file that you wish to open, and second is a mode against the string which represent the kind of operation that we are going to read (( )), for example, or we going to read the file or write the file and so on and so forth, and the return type is file star which has been is the name as the type for the file handle.

For this mode should be a string containing (( )) character r if you want to read the file. It should be w, if the file is to be written. So, we specify w as the mode for the file; that means, that the file will be created automatically if the if it does not already exist and if it does already exist then whatever data it already contain will be deleted. And whatever now the program writes will be the only surviving contents of the file. So, that means, that existing contents of the file will be removed and whatever the data or program write to its file that will overwrite the existing content. And if we do not want to do that if we do not want to delete the existing content of the file, we can open the file for appending by specifying the string containing a as the mode and in that case whatever data the program write will be appended to the file.

And off course again if the file does not already exist and the file will be created, but when the mode is r, when we want to read the file then the file must already exists otherwise it will be an error and will not get a proper file handle. So, if whatever reason the file cannot be opened, for example, we are trying to open it reading the file does not exist or if we are trying to we are trying to open the file for writing and we do not have permissions to write to the file or may be the file does not exist and we do not have permission to create the file and so on or maybe we are trying to read the file we do not have permission to read the file. If anything goes wrong and the file cannot be opened then as opened returned special pointer value null which will also have notation to see in the next lecture, this is very useful pointer value, but essentially this this is a special pointer value it does not point to any legal memory location.

So, the pointers that whenever we open a file, we must check the file handle that has been obtained and if that is equal to null; that means, we were not able to successfully open the file, and so we should print an error message in the program should exist (( )) what is the appropriate.

(Refer Slide Time: 28:21)



Functions for File I/O

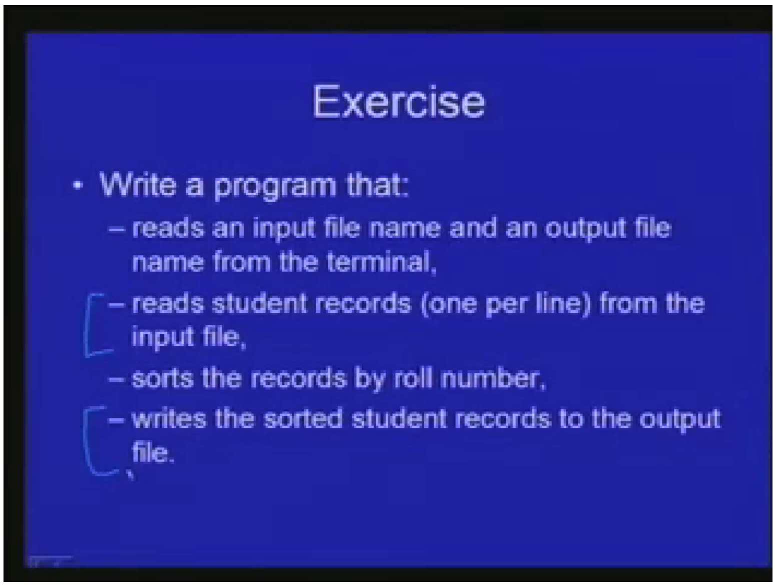
```
int fscanf(FILE *fp, char * formatstring, ...)
int fgetc(FILE *fp)                       getchar
char *fgets(char *s, int size, FILE *fp)  gets

int fprintf(FILE *fp, char *formatstring, ...)
int fputc(int c, FILE *fp)     putchar
int fputs(char *s, FILE *fp);  puts

int fclose(FILE *fp) /* close the file */
```

So, once we have obtain the file handle to a file, we can now use the file handle for doing actual I O on the file and the function for reading and writing a file as they are very similar to the one for reading and writing for the terminal, for example, we have the f scanner function which is same as the scanner function that we already seen. So, the scanner function leads from the terminal whereas s scanner function reads from a file the only difference in the argument is a first argument has to be file pointer, and then the format string as usual the format string exactly has the same meaning as in scanner then the function fgetc is again the same function get char for reading one character from the terminal.

This also returns (( )) from one character from the file and the argument is the file pointer. As fgets is the equivalent of the get s function, you specify the pointer to n character array where the data has to be added. Now there are two additional argument in this case, the internal argument is the file pointer and the second argument is the size of the array which has been specified s, and get s function ensured that no more this many bytes are actually returned of this array. Otherwise off course as you know there is a array bound bounces violation get s function does not actually check that array bounds violation. So, for writing to a file, the functions are again similar to the function for writing to the terminal. The printer function is exactly the same as function the printer function except that the there is an f star p argument which is the file pointer of the file are to which we want to write.

And this file pointer must be opened for writing or for a binding, and similar to put char function we have the f put c function which takes a character, and additionally a file pointer of the file to which this character has to been written and f put s function is equivalent of the put s function which size string to a particular file, and finally, this f close function is used to close a file as I said once program finish reading or writing the file it should proper the close the file and we use that f close function for doing that all we need to specify is the file handle or the file pointer as the argument.

(Refer Slide Time: 30:46)



Exercise

- Write a program that:
  - reads an input file name and an output file name from the terminal,
  - reads student records (one per line) from the input file,
  - sorts the records by roll number,
  - writes the sorted student records to the output file.

So, this is the end of this lecture, but before we stop here is the simple exercise for you to try using the concepts that we have learned in the last couple of lectures. So, the exercise to write a program that will read an input file name and an output file name from the terminal that is the user will type the name of input file and the output file. The input file is supposed to contain student record of the type just saw sort one per line. So, the program will read the student records from the file or the records by roll number using sorting algorithm that we already seen, and finally, write the sorted student record in the order of the roll number to the output file. In the next lecture, we will talk about another very important technique in programming and that is the technique of dynamic memory allocation which allows us to overcome the limitations of arrays.