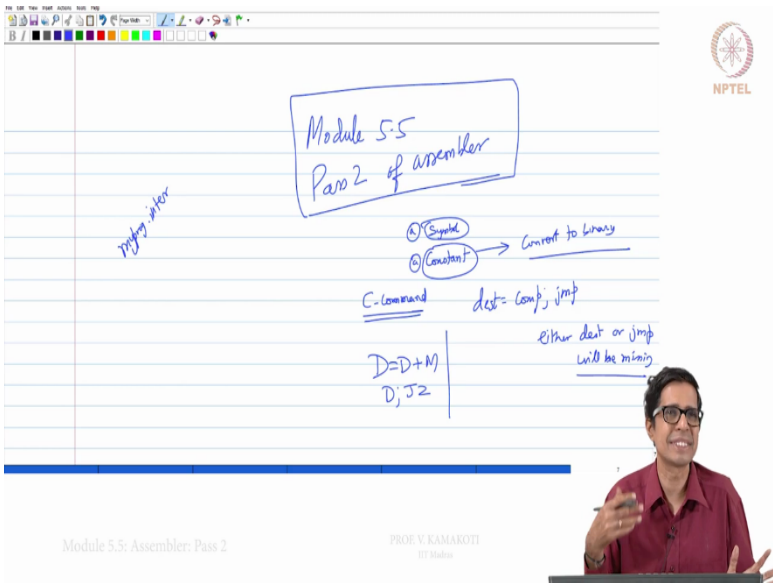
**Foundations to Computer Systems Design**
**Professor V. Kamakoti**
**Department of Computer Science Engineering**
**Indian Institute of Technology Madras**
**Module 5.5**
**Assembler: Pass 2**

(Refer Slide Time: 0:39)

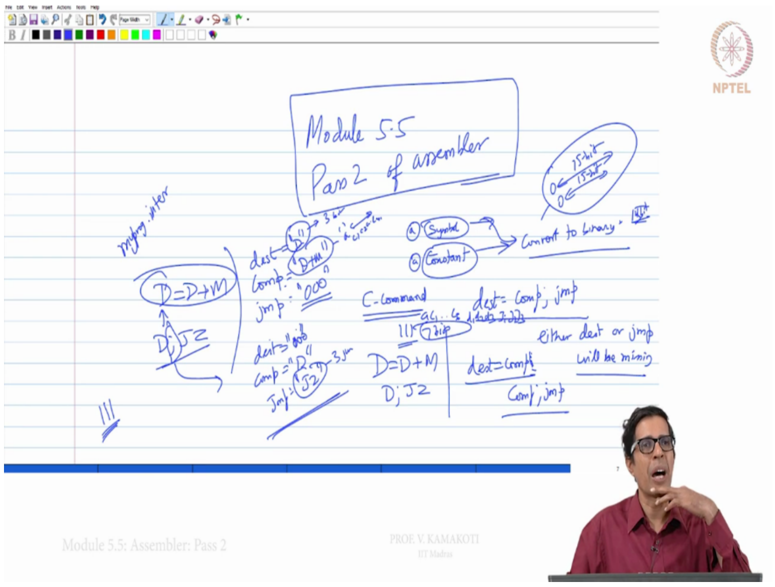


So welcome to module 5.5 and in this module we will see the Pass 2 of the assembler. So what is the input to this pass 2 is that dot inter file so myprog.inter file and as you have seen in the previous module, it is completely no comments are there and no white spaces are there. So what we will see in this file is 2 types of commands, the CA instruction or the C instructions, right and those are the things that are put into this file. Symbol all the symbol table is available to us and so we will see at symbol or at constant, the moment I see at symbol, now we know how to extract the symbol we have already done, take this symbol to the symbol table, look up the symbol table and get back the address.

And that address you actually, that address will be in decimal so you convert it to binary, so there is a routine that will convert decimal address to binary so after that it will give you a 14 bit binary or whatever so because these things would be sorry 15 bit binary because this will be and these are all both the symbols and constants are nonnegative decimal in the range of 0 to 32767 so in the range of 0 to 2 power 15 – 1 right. So when the moment so the first character will be at because we have decommented we have divide space so the moment I see an at, then I see whether it is a symbol or a constant, you check whether the 1st character is between 0 to 9 and then if it is a symbol, go to the symbol table, get the address and convert it to binary and then you put an instruction which is 0 followed by 15 bit binary.

If it is a constant you may convert it to binary and put it as 0 to 15 bit, so this will be the assembled instruction for that. Now the next thing that you may see is a C command, C command will be of the form destination is equal to computing semicolon jump, and either destination or jump will be missing right. If it is a jump instruction, I will have comp semicolon jump and if it is you know the compute instruction then it will be destination equal to comp okay destination equal to comp. So what we do is 1st we start scanning, we do not know whether when we start scanning the past comment like I have D = D + N, first I am looking at D, at this point I will never know that this is a compute instructions or it is going to be jump instruction because I could also have D colon.

(Refer Slide Time: 4:08)



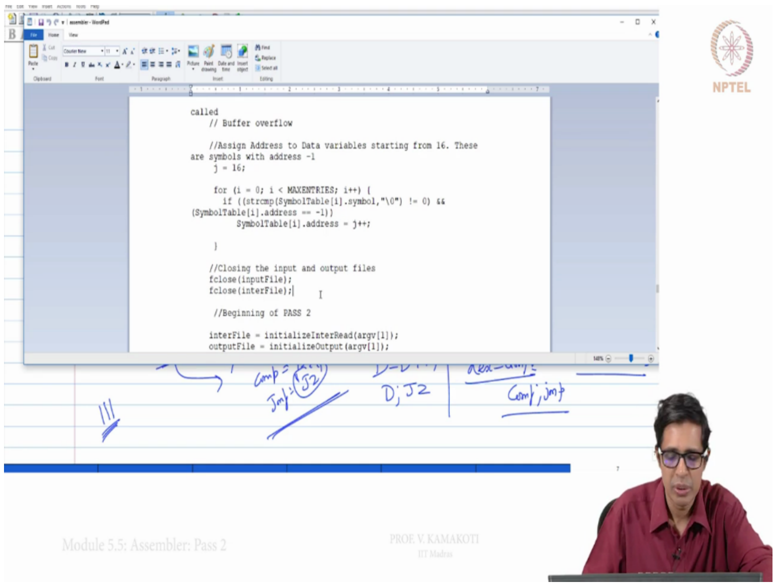
I have only seen D so I will start filling up 2 components, one is called the compute component, another is called the destination component, and another is called the jump component. So now I will make compute component, this is also a string I will just put the on the destination component and I have also put D. The moment I see =, then I know that this is a computation instruction so I will remove of this so I know that destination is equal to D and I will now start populating compere and the compere now become D + N whatever is that till the end of this thing. But on the other hand this is what will happen here so I start with destination is D and paralleli comp also is D and the moment I see = then I know that this is destination = comp so I will retain D here and comp I will now populate it with D + N.

In this case again I will start with destination as D and .comp ad D in the previous case comp will become D + m and of course the jump the 3 bits there are 3 bits which basically talks about jump and those will become 00 right, right this is done. Now in the case of this D colon, I start with dest = D, I will also start with comp = D, the moment I see this semicolon here immediately I will make dest destination also 3 bit, I will make destination as 000 and the jump as J z, so this is how we basically to the C command. So at the end of this level of passing I will have something for some string for dest, comp and jump, right. And if it is a computation structure, jump will be 000, if it is a jump instruction then the destination will be 000.

Now this sprint I take it to a routine which will translate which is basically implementing the table right, the table that we have seen in the past right. So D + m means what? D + m means there will be some big pattern for this, this will give you those A bit + those 6 bits c1 c2 to c5 c6, this will determine the 3 destination bits and this will determine the 3 jump bits, so this will become 3 + 7, 10 + 3, 13 bits and for the C instructions the first 3 bits are 1 1 1. So the moment I see a C command, now I see the 1st 3 bits as 1 1 1 from the dest string from the comp I will populate the 7 bits including a c1 to c 6 and then the dest will populate d 1, d 2, d 3 and the jump will populate j1, j2, j3 so we have to just do a string matching to do this.

And there are some tricks to make it, there are some care that we can take to make it more robust now this I so I implement pass 2. Now let us see the code for past 2.

(Refer Slide Time: 6:33)



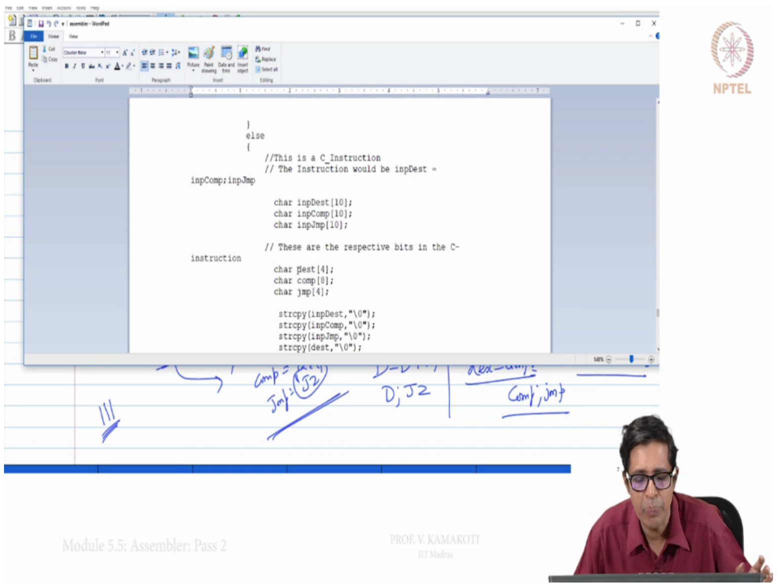
```
called
    // Buffer overflow

    //Assign Address to Data variables starting from 16. These
are symbols with address -1
     j = 16;

     for (i = 0; i < MAXENTRIES; i++) {
       if ((strcmp(SymbolTable[i].symbol,"\0") != 0) &&
(SymbolTable[i].address == -1))
            SymbolTable[i].address = j++;

     }

    //Closing the input and output files
    fclose(inputFile);
    fclose(interFile);

     //Beginning of PASS 2

    interFile = initializeInterRead(argv[1]);
    outputFile = initializeOutput(argv[1]);
```

This is the code for pass 2 just to give to I am now opening up the intermediate file for reading and of course the output file will be dot hac and I am reading again f getters I am using f getters I am reading it one after another one by one and then I go and do a command type. This is command type, the first character is at if it is A command, if the first character is not at it is a C command, all commands are now removed and all, right. Now if it is an A command immediately I go and check, if it is at of a constant or at of a symbol and if the if so if it is easier 1st character is less than 0 or greater than 9 character then it means it is a symbol so we just go and extract that symbol.

Extract that symbol means just copy all the things starting from that point till slash end and then you go and do this content get address of the symbol. This content get address will find out the address, 1st it will go and check whether that symbol is contained in the symbol table, if it is then it will get you the address for that, right. And if it is not contained in the symbol then you print an error so this is very-very important right so you print an error saying this is a symbol which is not bad. Otherwise if it is and after you get that address you convert to binary, convert to binary is nothing but just keep dividing by 2 and writing the remainder and that is. So that will fill up this 15 bits of this instruction; the first bit is 0 A instruction then remaining things.

If the command is constant then please note that this is a string right, so you have to convert that constant string into actually now decimal constant that is what we are doing here right and then that decimal constant you just convert to binary. And this assembly instruction that you say is that 16 bit you know string where we populate each bit accordingly as we have seen. In the case of E instruction it is 15, 0 to 15, 15th bit will be 0 and the remaining 15 bits will be filled by this convert to binary, convert to binary is a very simple routinely can write it.

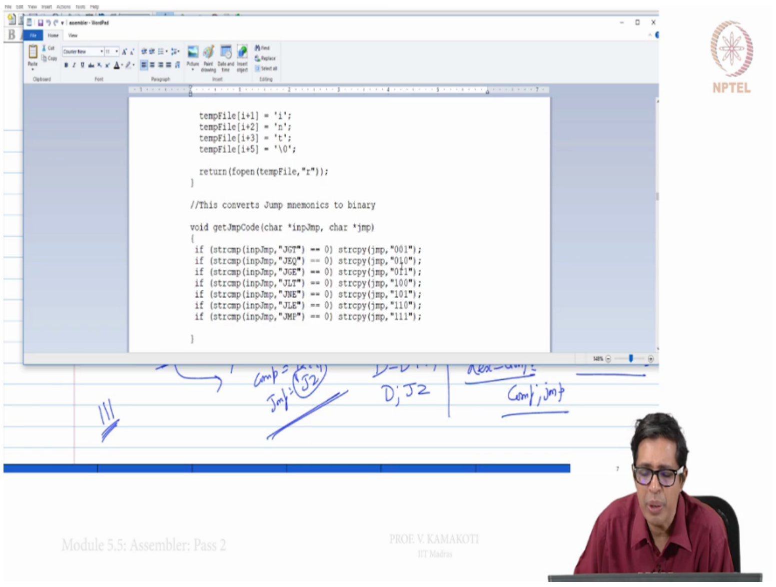
(Refer Slide Time: 9:06)



```
            }
            else
            {
                //This is a C_Instruction
                // The Instruction would be inpDest =
inpComp;inpJmp

                  char inpDest[10];
                  char inpComp[10];
                  char inpJmp[10];

                // These are the respective bits in the C-
instruction
                  char dest[4];
                  char comp[8];
                  char jmp[4];

                   strcpy(inpDest,"\0");
                   strcpy(inpComp,"\0");
                   strcpy(inpJmp,"\0");
                   strcpy(dest,"\0");
```

Now if it is a C instruction that is what we are seeing here then we have these 3 components dest, comp and jump, we have input dest, input comp and input jump so we basically extract so we start here right, we start as I told you we start populating both in dest an in comp and at some point of time when we find it is =, we make you know if it is = then we make jump as 0 right, if its is = I make you know we again start doing for in comp, I retain the in dest but again I make k = 0 as you see here I basically redo the comp right as we have mentioned.

(Refer Slide Time: 10:56)

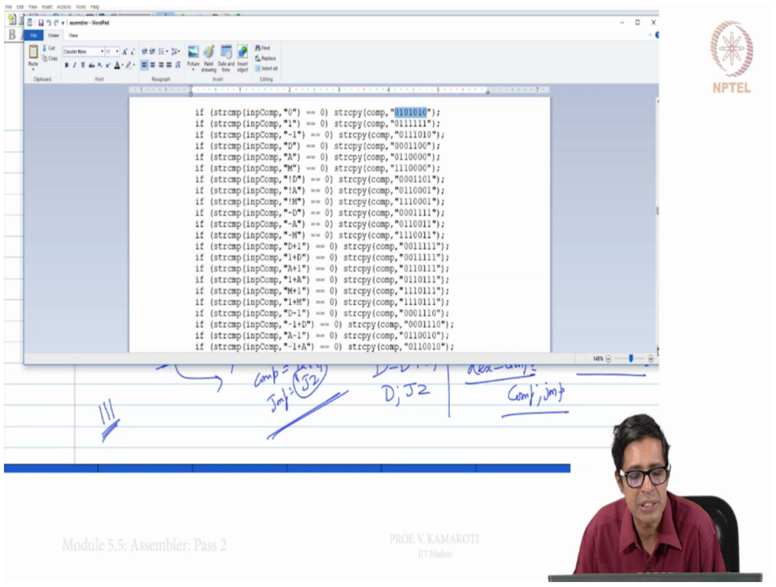


```
  tempFile[i+1] = 'i';
  tempFile[i+2] = 'n';
  tempFile[i+3] = 't';
  tempFile[i+5] = '\0';

  return(fopen(tempFile,"r"));
}

//This converts Jump mnemonics to binary

void getJmpCode(char *inpJmp, char *jmp)
{
 if (strcmp(inpJmp,"JGT") == 0) strcpy(jmp,"001");
 if (strcmp(inpJmp,"JEQ") == 0) strcpy(jmp,"010");
 if (strcmp(inpJmp,"JGE") == 0) strcpy(jmp,"011");
 if (strcmp(inpJmp,"JLT") == 0) strcpy(jmp,"100");
 if (strcmp(inpJmp,"JNE") == 0) strcpy(jmp,"101");
 if (strcmp(inpJmp,"JLE") == 0) strcpy(jmp,"110");
 if (strcmp(inpJmp,"JMP") == 0) strcpy(jmp,"111");

}
```

So we start by populating both dest and comp as I scan through this string but this = I I redo the comp while retaining the dest and your jump becomes 000 okay. And but if I encounter a semicolon that is this is a semicolon then I I retain the comp, I make the dest as 000 and the remaining things become jump. So at the end of this passing I know what is dest, what is jump and what is comp, the string. Now that will go and do here then we have this very interesting, so this is in if I say jjt then I know it is 001, I get the corresponding 3 bits equivalent for every jump and this is there in the table, I encode this table.

Similarly I also get for the destination code, if the destination is same I know the MD, but MD some can write it as DM, similarly AM somebody can write it as MA also, AD can be written as DA, AMD can be written as ADM, MAD, MDA DAM, DMA, all possible for all the things the same bit pattern can come. So this will make you know otherwise assembler fellow has to remember which alphabet. If I just put MAD alone then and all these combinations are missing then this assembler will not assemble that particular thing so we have to make it robust all combinations so this gives more flexibility for person who is actually rating the assembly code to put the destination like this. But this becomes slightly more complex when we go for the compute part right, this is the other table.

(Refer Slide Time: 12:14)

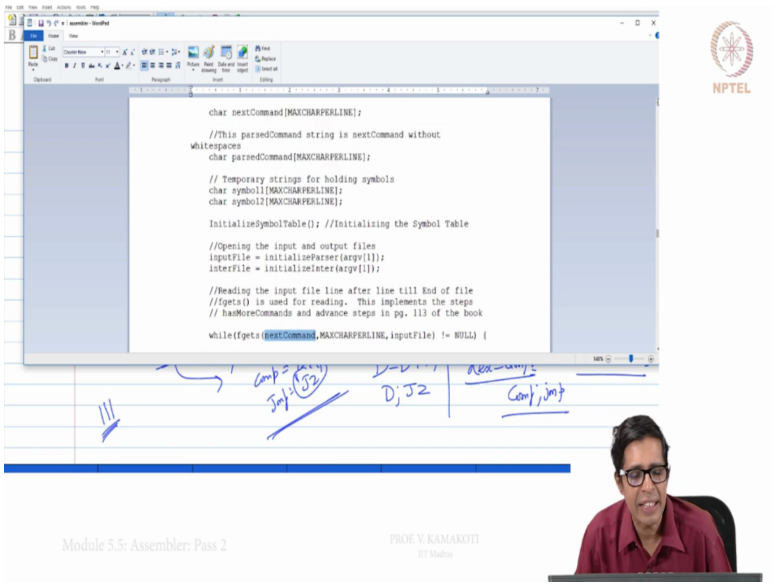


```
if (strcmp(inpComp,"0") == 0) strcpy(comp,"0101010");
if (strcmp(inpComp,"1") == 0) strcpy(comp,"0111111");
if (strcmp(inpComp,"-1") == 0) strcpy(comp,"0111010");
if (strcmp(inpComp,"D") == 0) strcpy(comp,"0001100");
if (strcmp(inpComp,"A") == 0) strcpy(comp,"0110000");
if (strcmp(inpComp,"M") == 0) strcpy(comp,"1110000");
if (strcmp(inpComp,"!D") == 0) strcpy(comp,"0001101");
if (strcmp(inpComp,"!A") == 0) strcpy(comp,"0110001");
if (strcmp(inpComp,"!M") == 0) strcpy(comp,"1110001");
if (strcmp(inpComp,"-D") == 0) strcpy(comp,"0001111");
if (strcmp(inpComp,"-A") == 0) strcpy(comp,"0110011");
if (strcmp(inpComp,"-M") == 0) strcpy(comp,"1110011");
if (strcmp(inpComp,"D+1") == 0) strcpy(comp,"0011111");
if (strcmp(inpComp,"1+D") == 0) strcpy(comp,"0011111");
if (strcmp(inpComp,"A+1") == 0) strcpy(comp,"0110111");
if (strcmp(inpComp,"1+A") == 0) strcpy(comp,"0110111");
if (strcmp(inpComp,"M+1") == 0) strcpy(comp,"1110111");
if (strcmp(inpComp,"1+M") == 0) strcpy(comp,"1110111");
if (strcmp(inpComp,"D-1") == 0) strcpy(comp,"0001110");
if (strcmp(inpComp,"-1+D") == 0) strcpy(comp,"0001110");
if (strcmp(inpComp,"A-1") == 0) strcpy(comp,"0110010");
if (strcmp(inpComp,"-1+A") == 0) strcpy(comp,"0110010");
```

Now if I want to 0 as a compute then this is the 7 bits 0 1 0 1 0 1 0 like that 1 – 1 DA not DA – 1, this is the table that we had for calculating those 6 bits. But then I could have D + 1, I also have 1 + D, A + 1, 1 + A, M + 1, 1 + M, D – 1, – 1 + D, A – 1, – 1 + A so all these combinations I have put, we are writing 28 combinations that we saw, now we have more than that so all these combinations, not much but this is what. So now we get assembled instructions and then we output that instruction so this is how we do the pass 2 so this is very-very simple course, you can spend just one or two hours you can finish it but you will also revise lot of things about C right, C as a programming language or use other languages like C++, Java, lots of things that we can learn here.

Lots of points you can also think that if there is an error then we need to we need to highlight that error right, so we need to write an assembler which is very friendly for development and one of the most thing is that we need to recover from errors. Recover from error means this is the error so this where there is a problem, you stop there, right you cannot say that there is an error and get away, you should pinpoint to that error and that is something which is the basic necessity of writing such type of system software right. And the last thing that I want to say before we wind up this is that we have been using something called f getters right. Now what will f getters do? It will read from your file whatever the full line it will read one line and it will put it into this buffer called next command.

(Refer Slide Time: 14:00)



```
    char nextCommand[MAXCHARPERLINE];

    //This parsedCommand string is nextCommand without
whitespaces
    char parsedCommand[MAXCHARPERLINE];

    // Temporary strings for holding symbols
    char symbol1[MAXCHARPERLINE];
    char symbol2[MAXCHARPERLINE];

    InitializeSymbolTable(); //Initializing the Symbol Table

    //Opening the input and output files
    inputFile = initializeParser(argv[1]);
    interFile = initializeInter(argv[1]);

    //Reading the input file line after line till End of file
    //fgets() is used for reading.  This implements the steps
    // hasMoreCommands and advance steps in pg. 113 of the book

    while(fgets(nextCommand,MAXCHARPERLINE,inputFile) != NULL) {
```

This next command is someone character array as you see here, which can hold up to say some 80 characters or 90 characters okay. So this will just basically your f getters will basically full this right. But one of the things is, suppose my input file has more than max char per line, this f getters will not check for it, it will just go on override things right, and this is what we call as buffer overflow, right. This is basically buffer overflow and what is the consequence of buffer overflow? Just because we are not checking this, down the line this buffer overflow has been the major source of hacking into system.

Today as you see one of the major issues that we are talking is about cyber security, information security and how does this cyber-attack happen? They happen because somebody can get access into your system, and how do people actually get access into your system? They do it by hacking into your system. And one of the biggest of the vulnerabilities that allows somebody to hack into your system is what we call as buffer overflow, and what is buffer overflow? One of the examples is this f getters, though I say that next command is restricted to some 80, max char per line is 80 here 80 char, if the input file has 85 this will still copy from next command from next command 0 to next command 79 but then some locations will be there after that it will go and override those things, f getters does not check this.

So in an uncontrolled environment like you write a software which is going to be used by somebody from the Internet, you cannot use this type of function which does not do a proper check, right but this is a very control environment. There is nobody is going to direct the assembler but if you see the software stack there is somebody who is going to use the assembler and that there is only one that the interpreter which is going to use the assembler, the output of the interpreter is going to be fed into the assembler.

An interpreter can take care that such type of overflows does not happen, so this is a very controlled environment so in a controlled environment we can use but in an uncontrolled environment this can potentially cause a cyber-attack right, it can be vulnerability which can be exploited to create a cyber-attack, so this is something very-very interesting that we need to talk off as a part of this question. So I hope you have a now good understanding of how to go about doing pass 1 and pass 2 and as a teacher I really want to practice and then preach, I have written the code right, I am showing you the code but this is a C code, you can write much better code in C++ or Java is more you know object-oriented you see, and if you do not know this do not worry you know C.

If you do not know C that this is the time where you learn C, and the way you learn C is there is a book called C Kernighan & Ritchie, I have learned it 35 years before. That a beautiful book it has seen generations, at least 3 generations of students I say every decade it is a generation, so I will say 3 generations of students. Please take that book, but how do you read that book? Now you have to read the entire… There is nothing as reading a programming language, we just start coding. So there will be lots of examples and we start doing the example problems, but most importantly do the exercise problems.

If you do the exercise problem, you do that Kernighan & Ritchie, it is basically 200-250 pages if I remember correct, and if you do all the exercise problems then take a month's time, you can learn any other programming language, it is my opinion. If you know C, you can quickly grasp the other languages very quickly right, C is a very fundamental language and the way to learn C is to go through Kernighan & Ritchie's book which can give you very deep insight into how do you go and do C. And knowing this programming language is very important from your professional career point of view, placement, whatever right.

And even if you want to teach the clarity of how a software has to be designed, that clarity comes when you do lot of programming, if you do not do programming you are out right, you can never claim to be an engineer. So with this I finish off this module 5.5 and we will meet in the next module 5.6, thank you.