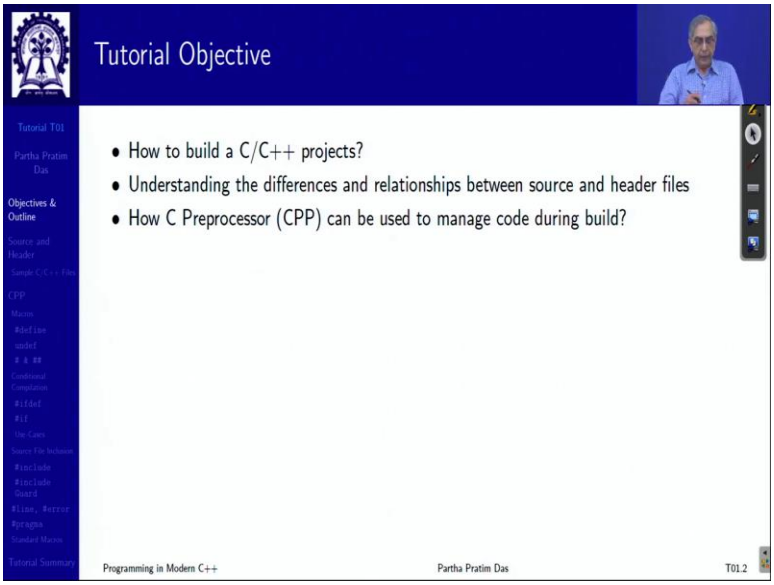
Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture No. 09
How to build a C/C++ program?
Part 1: C Preprocessor (CPP)

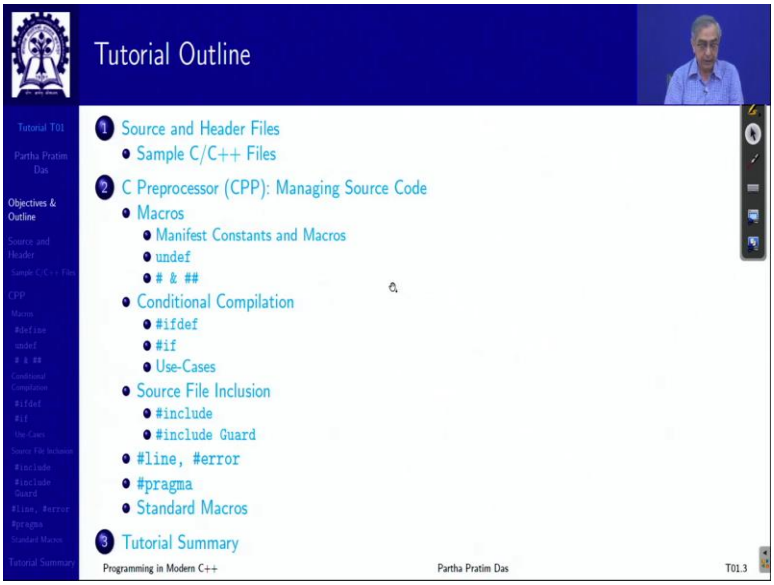
Welcome to programming in modern C++. This is a tutorial and I should rather say that this is a tutorial to complement your course, in modern C++, we will have several of them.

(Refer Slide Time: 00:54)



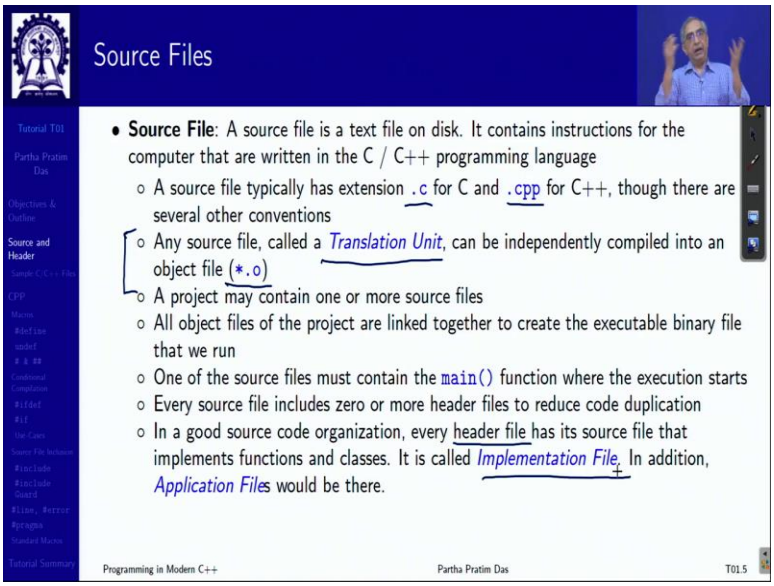
So, this is the first one. The objective of this tutorial is to discuss how to build C/C++ projects. This is not a primary discussion on the language or its features. But you will not be good with the language or programming unless you know how to build big C/C++ projects programs. So, we will initially have a series of 4 tutorials over which we will discuss different ways to build and manage C/C++ projects.

(Refer Slide Time: 01:43)



In the first one, we will talk about source and header files and the C preprocessor that is basic.

(Refer Slide Time: 01:47)



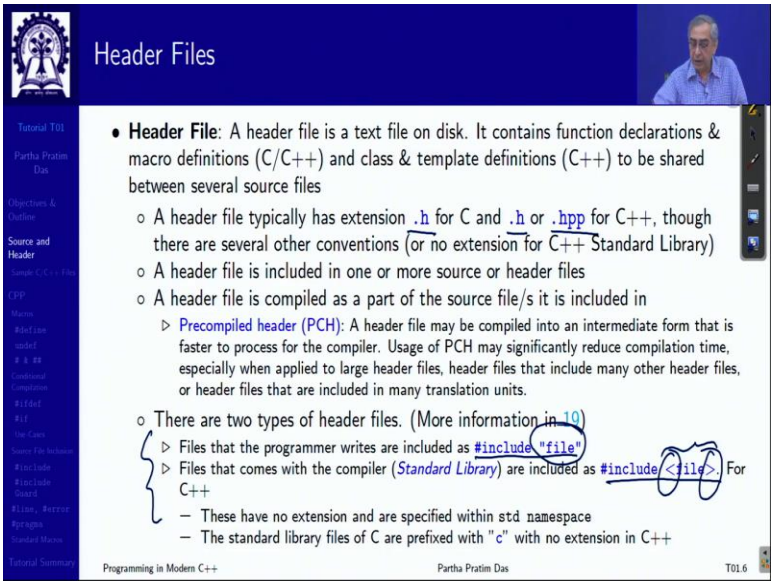
So, loosely you all understand what is a source file, it is a text file on disk. And they have typical extensions, .c for C files and C++ .cpp for C++ files. But there are several conventions if your company is using a certain convention just to follow that. Conceptually, what is the source file?

A source file is one which is a translation unit. That is every individual source file can be compiled independently by the compiler. That is the actual technical definition of a source file. So, it will compile and create a binary object file. Now one of the source files certainly has to

include the main function. And when all source files in the projects have been compiled, their .o's will be linked for the final executable, we will come to more of that.

In a in a typical style, we expect that the header files, I will also define formally what header files is, but you grossly understand, every header file that you write should have a corresponding source file. That is a cleanest design. If I have complex.h, there has to be a complex.cpp. If I have point.h, there must be a point.cpp like that, so that they can be independently compiled. These are called implementation files. And finally, we will have application files to do whatever application you are into.

(Refer Slide Time: 03:24)



A header file is also a text file typically, extension .h for C and .h or .hpp for C++ though several other extensions have been used or as you have already come to see that in C++ standard library headers do not have an extension. Now, the difference is headers must be included in one or more source files, because they are not independently compliable. So, they are not translation units. There could be precompiled headers and all that.

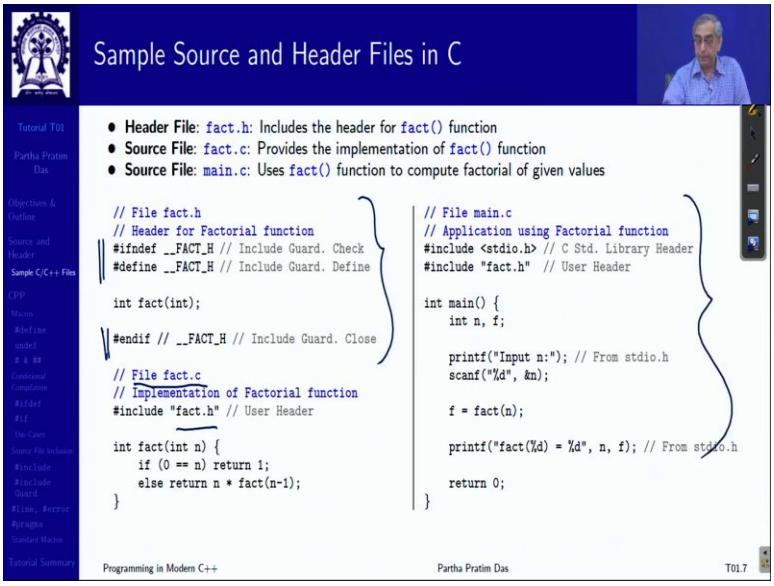
The important thing to note is there are two conventions for including the header file double quote, to be used when the header is written by you the application programmer or the, the programmer person. And this corner bracket should be used for the library inclusion you will wonder as to why these two differences? very simple. What does the preprocessor do is based on

the type of quoting of whether it is double quote or corner quote, it knows what is the directory, what is the location in your system, where it will look for that header file.

If you have not given the entire directory path, you are just saying, complex.h or you are just saying iostream, how does the system know where does that header file exist? So, for standard library the system has to know and system looks there based on this notation for user defined header files, the system has a protocol to follow like, it will look at the current folder or it will look at if there are paths defined into that and so on so forth.

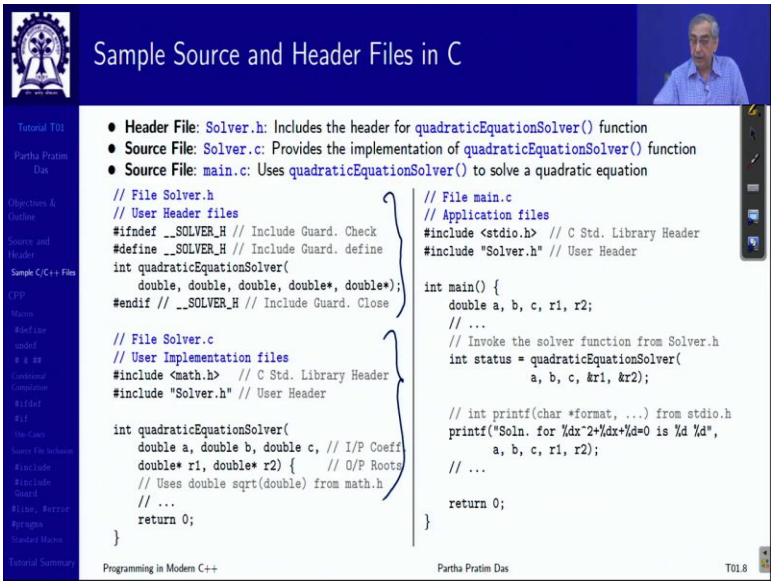
Things vary between Linux and Windows and Mac and so on. But that is the reason you need to distinguish the header inclusion syntax.

(Refer Slide Time: 05:26)



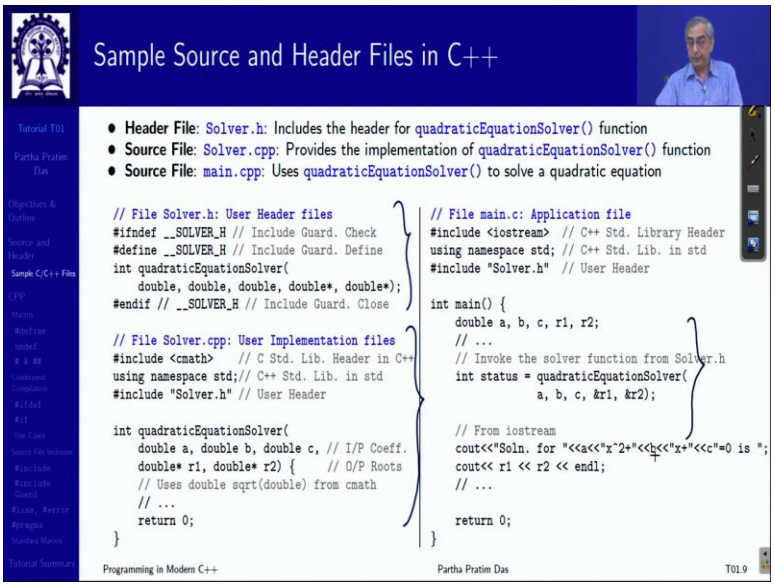
So, quickly this is all you know, this is a header file for factorial, you can see that I have done something here I will explain that this is called include guard, I will talk about that later. This is the implementation file fact.c, including fact dot h and this is the application source file, main.c. So, you should always you should not put all of these into one file and start doing things you should always organize in this manner.

(Refer Slide Time: 05:56)



Similarly, there is a, this is another example in C with a quadratic equation solver. So, the solver.h is here, which tells you the solver function, prototype and so on. Then the actual function implementation in solver.c, the main to use it.

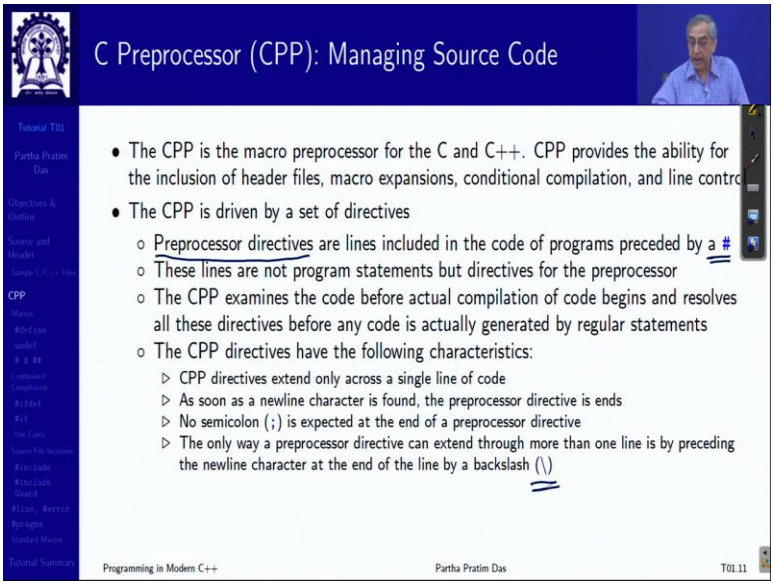
(Refer Slide Time: 06:17)



This is the same thing you can do in C++ and you should be doing that. So, this is a solver.h this is solver.cpp including solver.h and the main to make use of it. The same thing will apply whether you are just having, functions or you are having functions and classes both. Now, the first thing your code hits. So, we said we will we have written the program will compile and run.

But as we go to compile what does your code encounter first, what it first encounters is a C preprocessor, which is same for C as well as C++ it is called CPP.

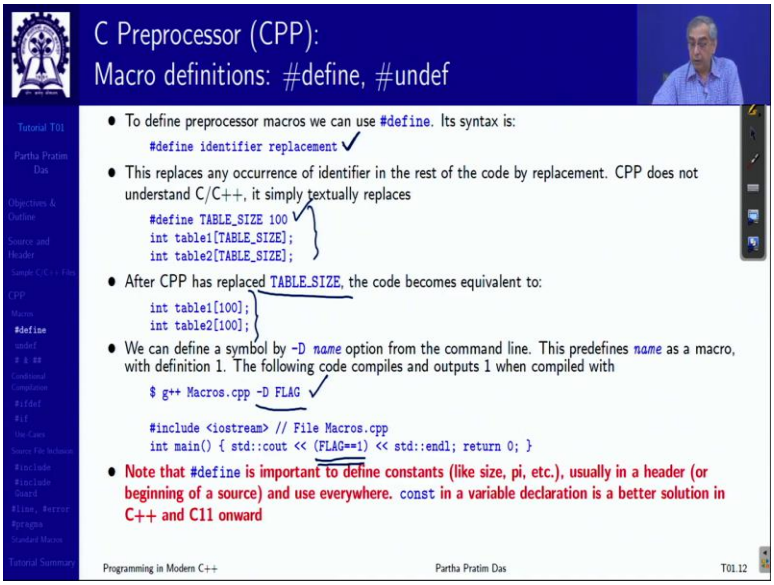
(Refer Slide Time: 07:05)



Because it manages your code. It is a macro processor; it does not directly look into your language. Rather, it uses the preprocessor directives which are preceded by #; #include, #define these you are already familiar with. So, these are preprocessor directives, every line of your any preprocessor directive has to be on a separate line, starting with the #, and ends with the newline character.

If you want to extend into multiple lines, you can put a backslash at the end, there is no need to give semicolon or anything because it is not a part of the language. He is just trying to manage your code and some of the hyper symbols.

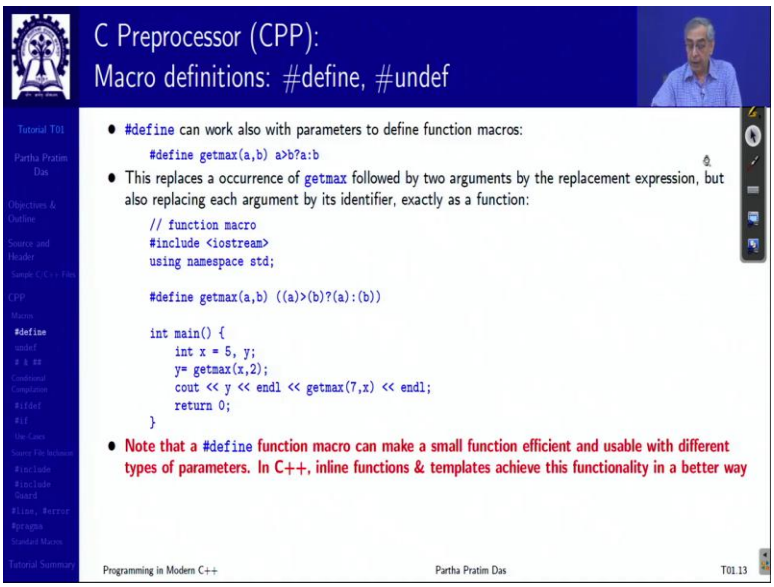
(Refer Slide Time: 07:50)



So, quickly this is this is you already know you can this is the #define is an identifier and then replacement whether there is a form with the parameters form without parameters, you know all of that. So, when I write this, CPP will replace this and generate this and that is what will be going to the actual compiler. So, you can define a symbol not only within the source, but you can define it also at the compilation line.

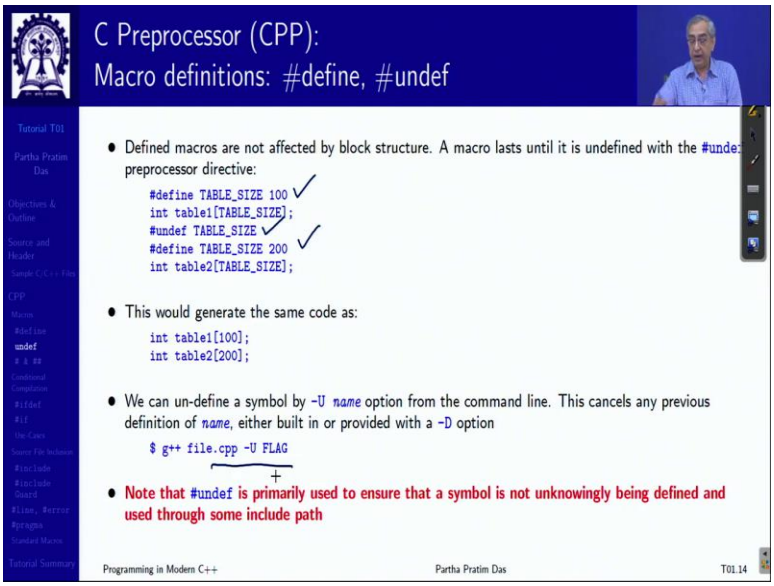
So, if you say that macro.cpp -D FLAG, then flag will be taken by cpp to have been defined. So, if you have a check here for flag is 1 once it is defined, it will be taken as 1.

(Refer Slide Time: 08:47)



Now, macro with parameters there, we have discussed in depth.

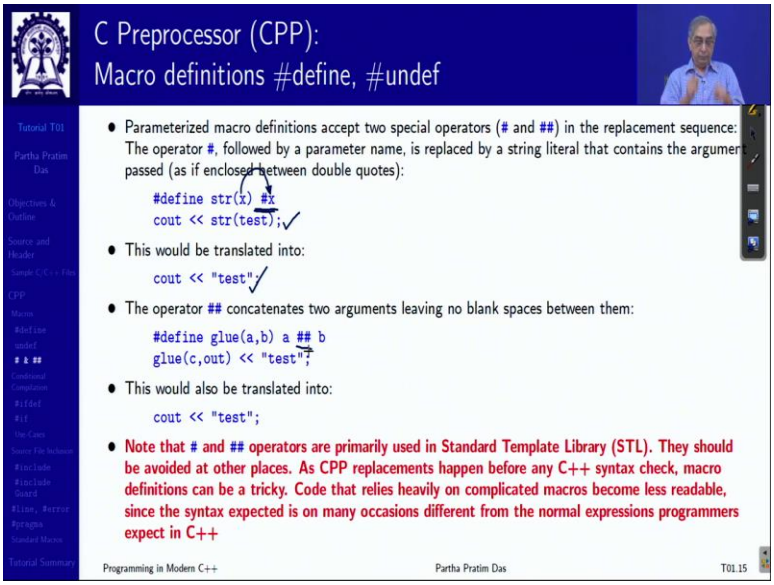
(Refer Slide Time: 09:01)

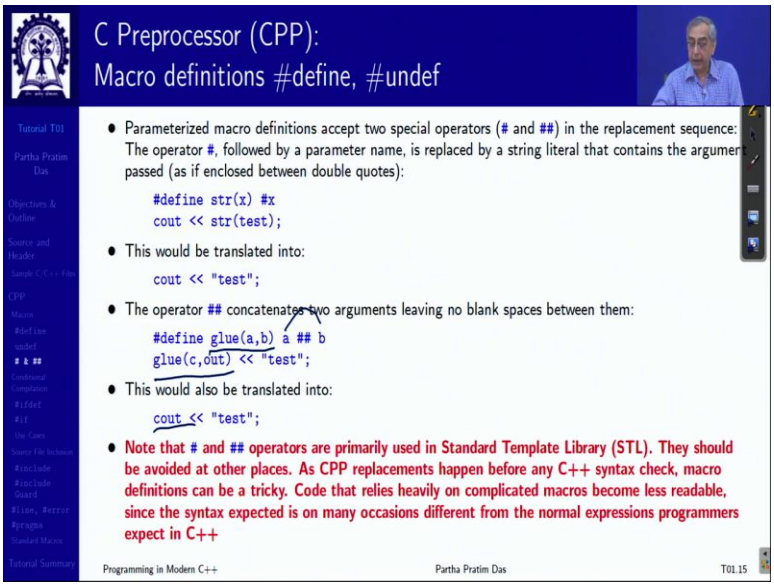


Now, what is important for hash define also is the fact that you can undefine a symbol. I am not sure whether you have seen this before. So, you have defined this and you can undefine it. Because if once you have defined it, it will be always replaced by that value. If you want to change that into a different value or a different expression, you can do undefine and define it again.

So, this is a typical example, like -D defines a symbol -U will undefine a symbol if you do it from the compiler command line.

(Refer Slide Time: 09:35)





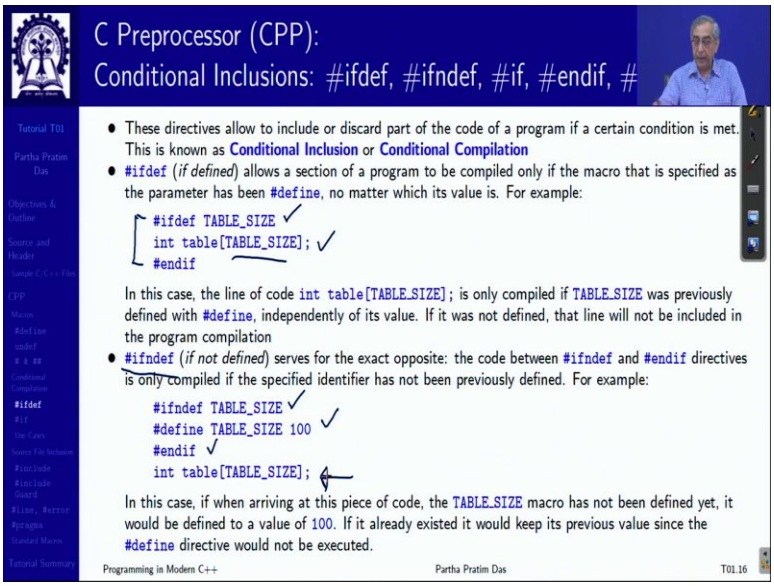
Now, there is an interesting feature, it is the # operator. What it does is it replaces the like it is like a macro parameter but what it does is it is if I pass x and write #x, then it will actually put the string that have passed. So, as I do str(test), it will actually generate a coded string like this, which is very useful in making certain different types of, output strings and format strings and so on.

Another very nice thing is you can concatenate two parameter strings into one, put them together. So, let us say if I put if I said there is a macro called glue, which takes a and b and I said that this is to be concatenated. So, I can write cout what you will, but just to show you what

can be done, I can write cout as glue (cout). So, this will take C as a string out as a string put them together cout and this is what it is.

So, these are these are primarily used in Standard Template Library, because a lot of template manipulations are required. So, I would not say that, I encourage you a lot to use them, but know that these kinds of things exist.

(Refer Slide Time: 11:17)



Now, the second thing that the CPP does is conditional compilation or conditional inclusion. That is, you may want that depending on what has been defined certain part of the code to be included in the compilation certain part not to be included in the compilation. This is not an if statement of your execution.

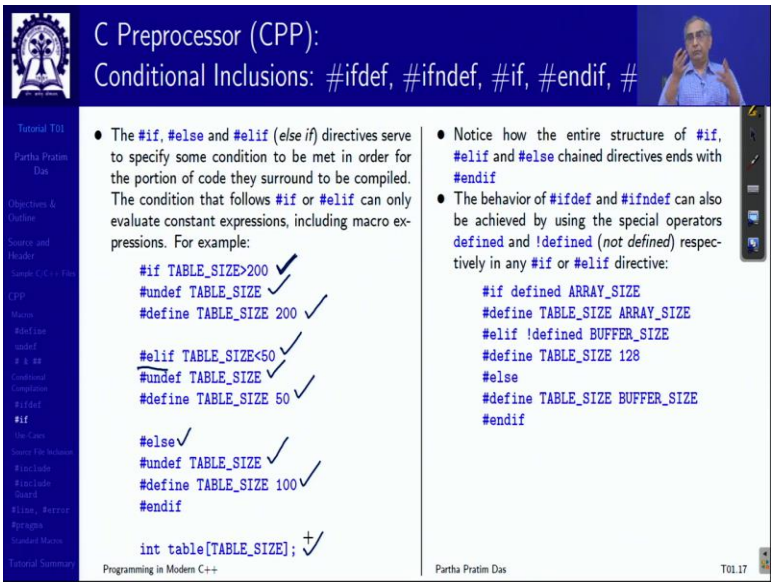
Where a code is the runtime is checking and doing it here you are saying that, do I include this code in my compilation or I do not. Something that I include will be retained by the CPP something that I do not include will be removed by the CPP before the code goes to the compiler. So, for example, I say ifdef TABLE_SIZE, I do not know if the table size is been given say it is expected to be given from the compiler command line.

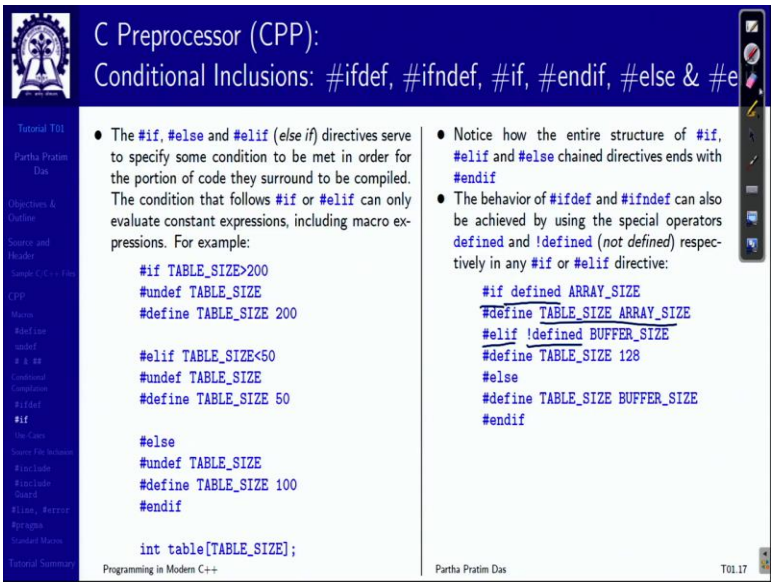
But I do not know whether that has been defined, if it has been defined, then I will use it may have been defined anywhere earlier. So, ifdef will check whether it there exists a definition for it, if it does, then I can use it. And any if ifdef this kind of directives will always end with a #endif.

So, if it is previously defined, this will be done I can also do if endif, if not defined. For example, I do it here I said if not defined, not already defined, then define it done with it.

So, when I come to this point, it was either earlier defined or it has been defined now. So, this is this is the tiny different controls you can easily do in your code.

(Refer Slide Time: 13:20)





You can also use else and elseif. So, let us say you said table sizes if table size is greater than 200. Here it is not ifdef it is if. So, when I say if it is more like a normal if, that is if TABLE_SIZE is not defined, this will fail. If TABLE_SIZE is defined, and its value is greater

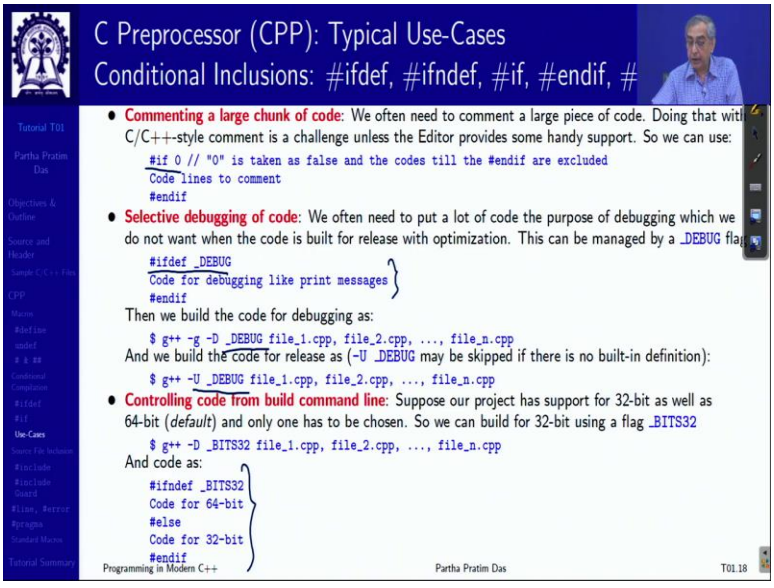
than 200, then this will succeed, then what it will do it will undef. So, you are what you are doing, you are reducing the TABLE_SIZE.

Else, it is called elseif as in else, and again, you want to do if combined together into a macro directive elseif you check a check for greater than 200 now you are checking for less than 50. If it is then you undefine and make it 50. So, greater than if it is greater than 200 you are making it 200 if it is less than 50 you are making it 50 else. So, in a chain of if elseif elseif else if the last one has to be else, else you just undefine and define a fresh value 100.

So, if TABLE_SIZE is not defined at all it will fall to the else. If it is defined between anything between 50 and 200 it will fall into 100 and then you use the TABLE_SIZE. So, you can make any this kind of logic then and make use that there is another form of ifdef also which is used by the defined macro. So, I say #if defined ARRAY_SIZE which is which will become truly if array size defined. And then you are saying that table sizes array size similar elseif not defined BUFFER_SIZE.

So, actually more than when you want to do this if elseif chain this defined and not define are useful otherwise if you are just doing one check then you can always use if def or if endif. These are very, very important directives to make your code build as you want it to work.

(Refer Slide Time: 16:09)

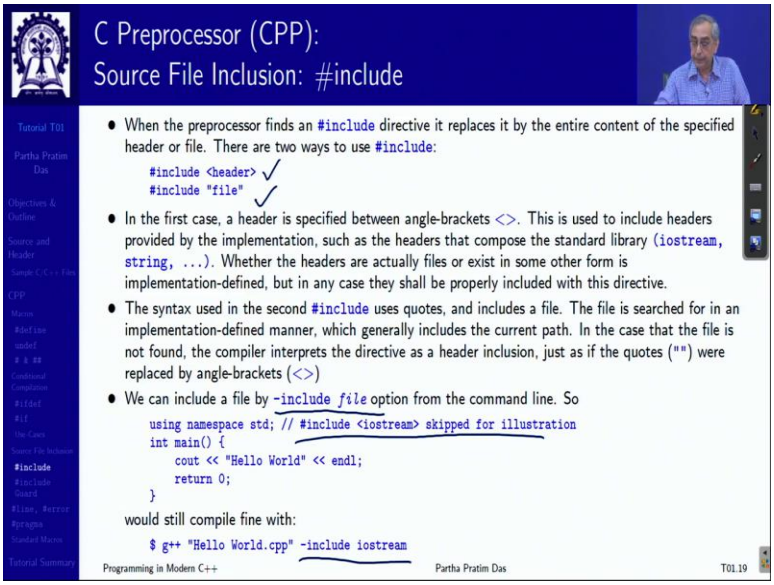


So, what are some of the typical use ifdef or if is a good way to comment out big chunks of code instead of doing // // like that, you can just say if 0, 0 is false if 0 and then endif that whole code

is got commented. You may want to do a selective inclusion you say #ifdef underscore debug. Now, during compilation from the command line, if you define debug, then it will include the debugging code in the build if you do not include the debug, it will not include that it will be really spilled.

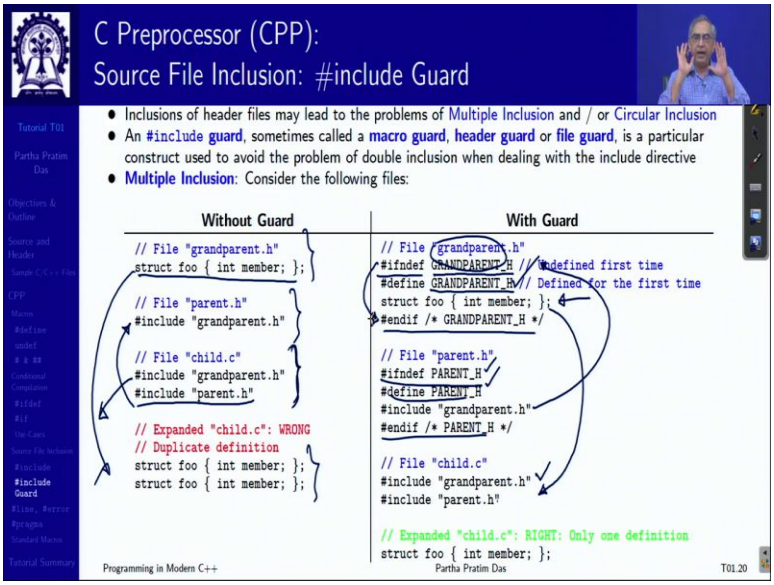
So, that is what we are showing here. And if by default debug is defined, then you can also undefined debug you can control the build command line in this way by defining the code with ifdef or with if endif. Just try this out so that along with the programming, you also become comfortable in terms of handling bigger and bigger code.

(Refer Slide Time: 17:25)



Source file inclusion you already know and I have already explained why there are two types of headers. So, that explanation is here. So, what does the CPP do is in place of the #include it will replace that entire file and #include you will not be able to see you can also actually include a header from the command line by using - include and file name. So, you have not given iostream here, but from the build you can do this try this out good fun.

(Refer Slide Time: 18:11)



This is an interesting hack that you must know which is called macro guard or header guard or file guard include guard like that. The issue is supposed you have multiple header files like one is grandparent.h which has defined a structure you have another parent.h which included grandparent.h. Now, then you are writing so, there are two headers, then you are writing a source file and in the source file, you have included you do not know whether you have to go to parent or grandparent you have included both of them included both of them.

Now, what will happen this grandparent will include the structure then you come to include parent, include parent will again include grandparent. So, it will again include the structure. So the structure will come one after the other twice. And to C or C++ this is an error, this redefinition of a structure by the same name. So, your code will not compile. There is nothing wrong in the code.

But all that you need is if since you are including several header files and header files mutually include each other all that you need to ensure is if a header file is already included in your source, then it should not be included again. That is include only once and you can do that very easily by this macros that we have seen, what you do is say the grandparent.h, you define a symbol GRANDPARENT_H.

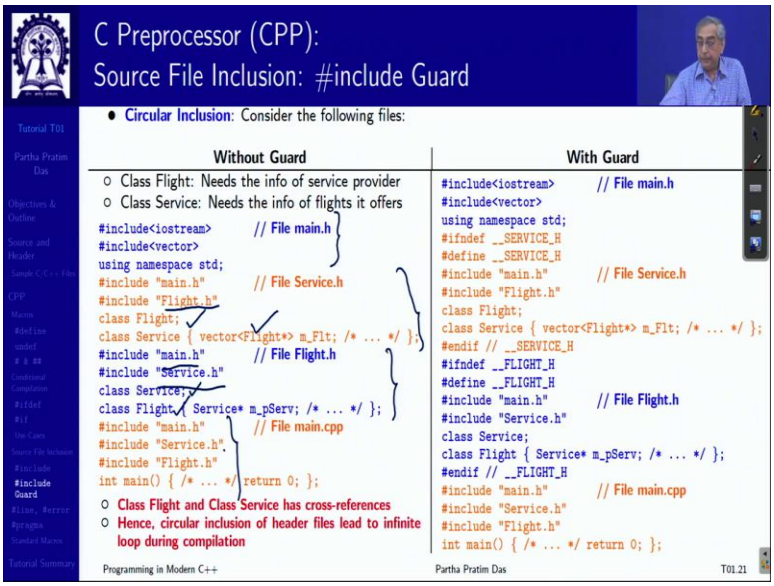
So, you say ifndef GRANDPARENT_H define and if and within that you put the code this is your actual code. Similarly, you have done that in the PARENT.H. Now, what will happen when

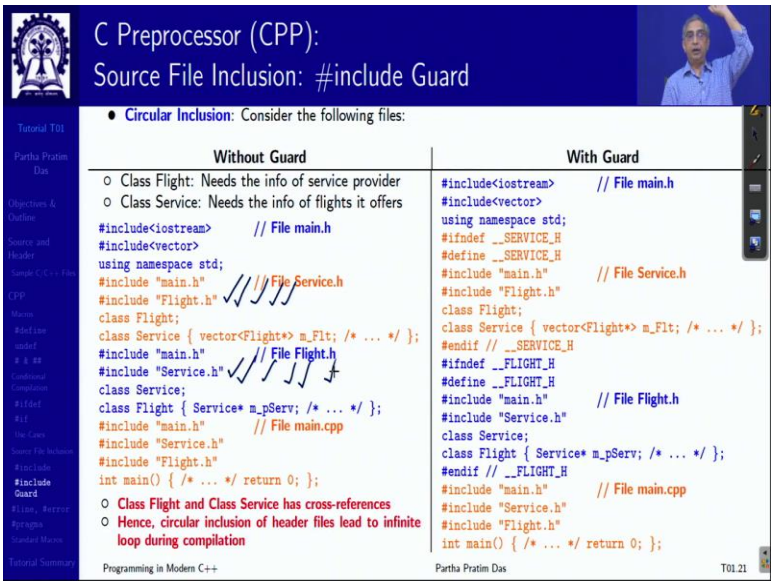
you include this, this is not defined, this symbol is still not defined. So, this will get defined as it get defined, this gets included at this point the surrogates included. And now you go to parent.h try to include parent.h this is not included this is not defined it is not happened yet.

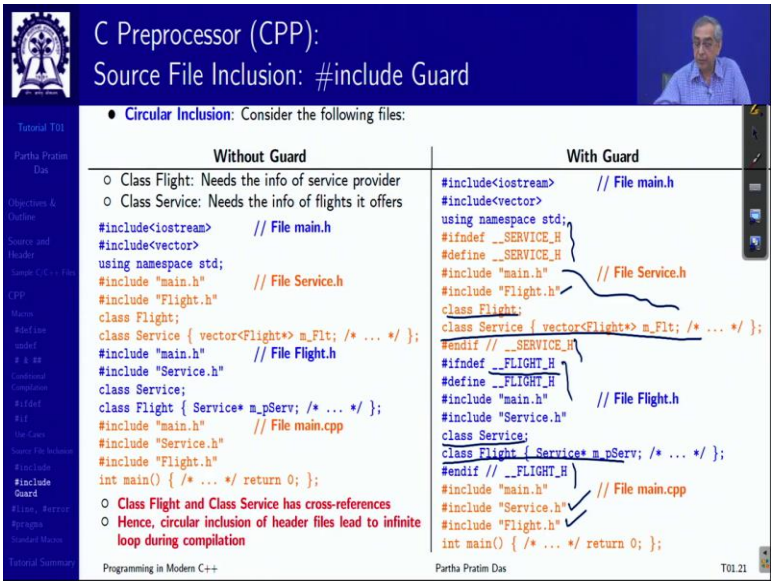
So, this gets defined. So, this goes to include grandparent.h. Now, grandparent.h has already set GRANDPARENT_H symbol as defined. So, ifndef will fail now, because it is defined. So, it is ifndef is not going to succeed, which means, though yet including it here, the inclusion control will jump to the end of this file and nothing will get included that is all the guard works.

So, you check if it is not defined, define it, put the code and so that next time, if it gets included through some other header files or even by the programmer in the source file a second time by mistake, the actual inclusion will not happen. So, this is a very very useful so, every header you write, you must have your include guard put in that header to make sure that everything is included only once.

(Refer Slide Time: 21:56)







So, this is just a just a little different example, this is called circular inclusion. There is a main.h in blue, which has this to be included in main. There is a file service.h which includes main, Flight.h, class flight, service. There is a Flight.h, which includes main Service.h for our declaration of class service class flight, so, what is that is a flight they are service. So, a service needs to know the flight, flight needs to know the service.

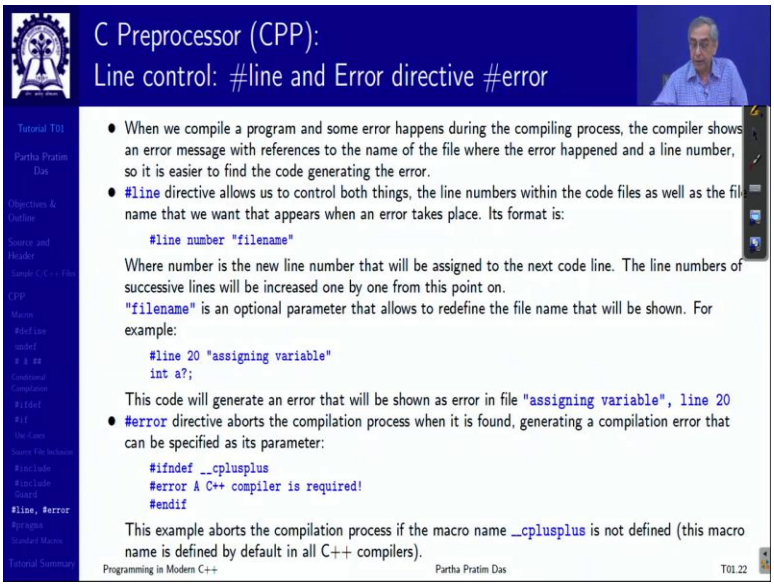
So, the header of flight includes the service header of service includes the fight. Now, naturally, when you write the main, you will include both of them because they are across references. So, what will happen when you write this the service.h, service.h is here. So, you get in here you try

to include Flight.h. So, you come include this code, you try to include Service.h, you go back try to include Flight.h come back try to include Service.h indefinite circular inclusions.

So, this is these are common pitfalls to have that is what is the that is what is experiencing. So, if you put the guard, then what will happen you have Service.h so, you will include service.h that is you are including this so, it will include Flight.h. So, we will have class Flight class Service class Service forward declaration which is incomplete type and then the complete declaration of class Flight this is done.

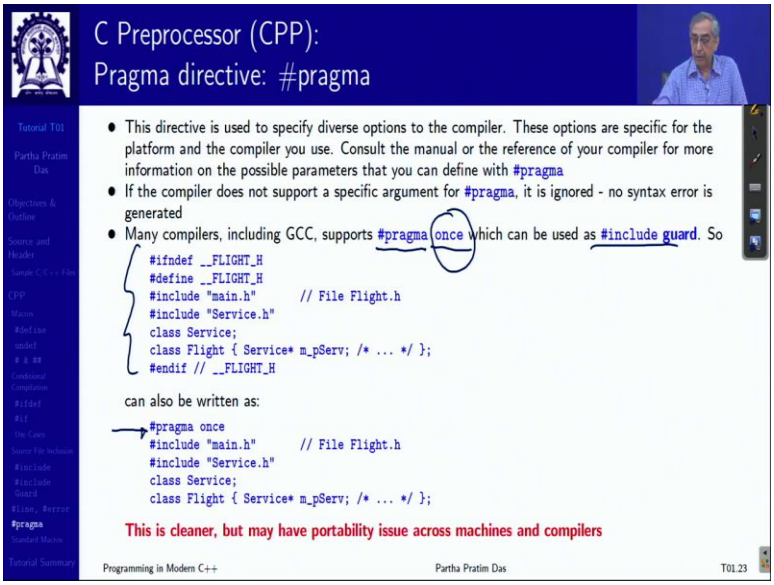
Now, when you try to include Flight.h you already will get this symbol as defined and you will do nothing. So, very simply by one inclusion principle you have been able to break this circular chain and the possibilities of such circularity at all happening in your code. So, this include guard is strong.

(Refer Slide Time: 24:50)



Now, there are several others these are less frequently used by programmers one is at any point you can print the line number in your original file because by putting the #line directive, you can also print an error message during this by putting the #error directive and so on. So, you can just try them out I am not.

(Refer Slide Time: 25:24)

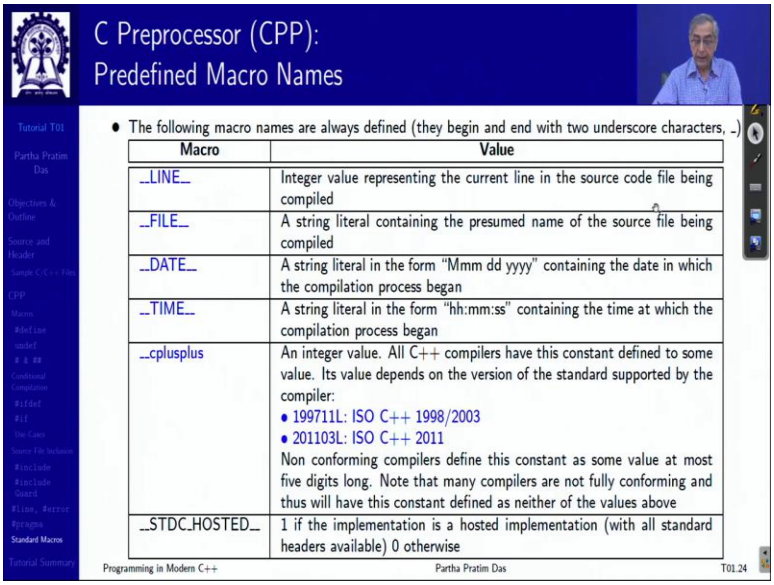


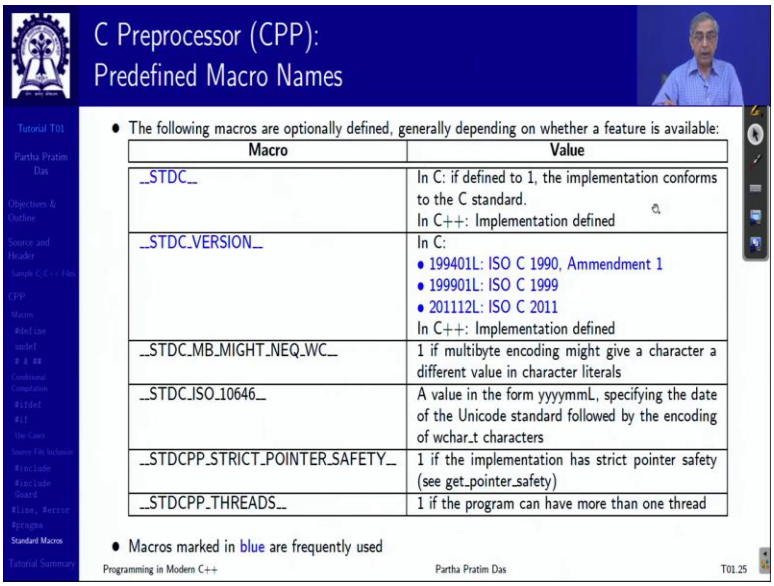
There is another directive which you will find in the code if you it is called #pragma. #pragma is actually not does not have any specific meaning it has been given so that any compiler vendor who is writing the compiler can define #pragma with some subsequent parameter or name whatever you say. Now, those are implementation defined that is this will not in general hold for all CPP processes, but for that particular.

And several times compilers want to do that, because they may want to start directing their code inclusion for example, they may want to control they may have multiple libraries compiled with multiple optimizations and they may want to control which library will be actually included and so on so forth. So, pragma once is, is one which is commonly used by many compiler vendors in place of include guard, that is whatever we achieve by doing this include guard if you just say pragma once it does that, it will internally make sure that you cannot include it once more.

I will still not advise you to use this because it is it looks cleaner, it is easier to do, I am sure, but the portability is not guaranteed. So, your code might work on your compiler on a different one, it will it may not work it may fail. So, but know that this is available, when you are in a structured company you will possibly.

(Refer Slide Time: 27:13)





Then there are some names which are predefined in the macros like the start in this these are the names that you have one is a line which tells you the particular line where you are in. One is the file which tells you what is the name of the translation unit that is being compiled, one is the current date and the time and in C++, you also have an __, C++ macro define which tells you the version of the C++ language.

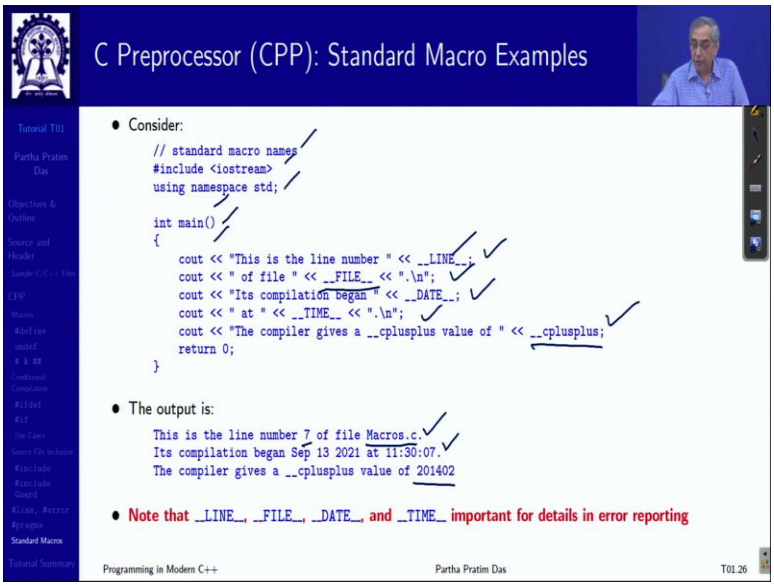
Now, this is these four-line file date time are very important to give different kinds of messages for example, if you are trying to write some error message in your code, and you may have a very similar type of message object construction failed something like that at multiple places in

the code. So, when you get the error how do you know where did it come from? Like very simply, when you are we are using the compiler we when you have an error or a warning we get in this file on this line this is there.

Now how do you generate that in this file and on this line, so we can do that by using this macro names. So, treat it as a string. And if you put that in your code, it will simply expand while it is compiling, it will simply expand to the current translation unit name so that you can give very source contextual messages using the file and line macro names, date and time or for if you want, when it happened and so on compiler gives that as well.

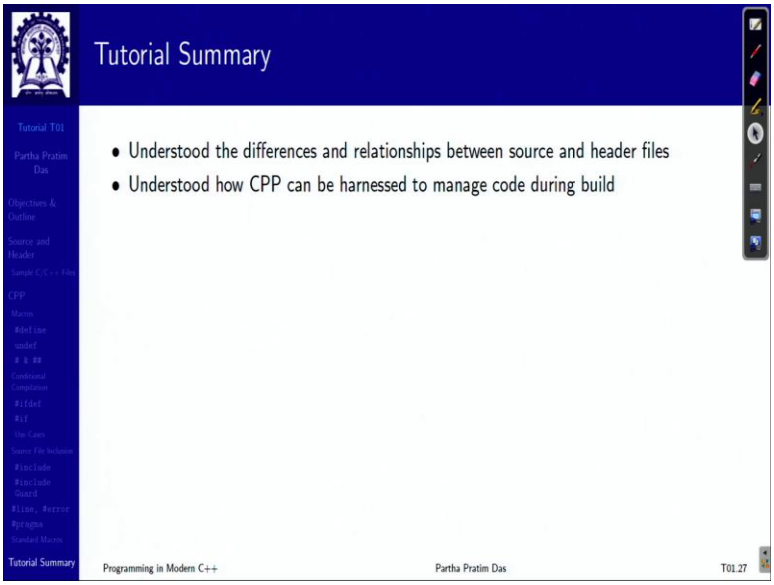
Some of the other macro names involve versioning of C versioning different version numbers, how they are available and so on so forth. Not frequently used, but versioning of C and C++ is important because you will need to know what particular version you are using.

(Refer Slide Time: 29:44)



So, this is an example where I am showing that how does these things work? So, this is the line number 7 of macro.c. 1, 2, 3, 4, 5, 6 this is 7. So, this is this 7, it was macro.c is the name of the file that I used is the line number you got. And this is a date time I had done this, and it tells us C++ value is this, which tells me what is the version of C++ compiler that has been used to build this.

(Refer Slide Time: 30:31)



C preprocessor gives you lot of handle and lot of information about how you manage your code and how you make your projects really flexible. Mind you it does not have anything to do with your actual language codes or that it is just organizing it, it is just doing inclusions it is giving you flexible ways to comment out code, it is giving you efficient ways to correctly include headers it is giving you ways to manage the whole code scenario and make a build.

So, thank you very much for your attention do raise questions on this particularly during the interactive sessions that will follow. I would love that if you ask questions also from this tutorial because the tutorial is important the reason, I am doing this is just knowing the language will not get you anywhere.

You finally need to be highly employable. And employability depends on practicing. So, tutorials we are building to those aspects of supporting the hands on. So, do practice that I will come back in the next tutorial discussing about that after the CPP what happens, how to build the what is the total build pipeline that you do. Thank you very much for your attention, see you.