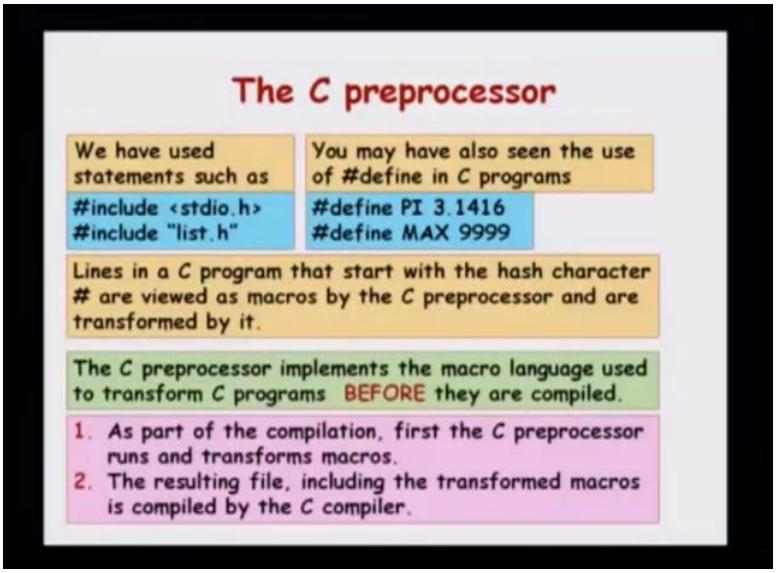
Introduction to Programming in C
Department of Computer Science and Engineering

(Refer Slide Time: 00:09)

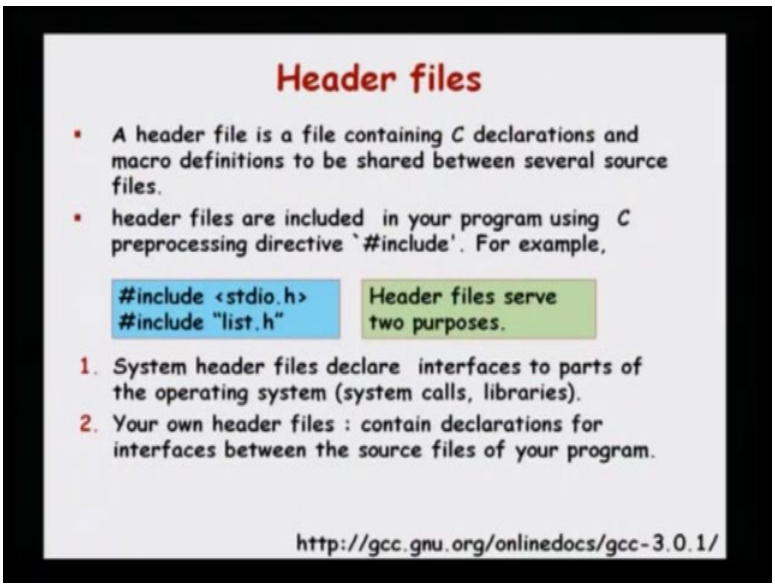
In this lecture, let us look at one particular part of the C compiler which is very important, namely the preprocessor. Technically speaking, the preprocessor is the step before compilation. So, let us understand this in detail

.(Refer Slide Time: 00:18)



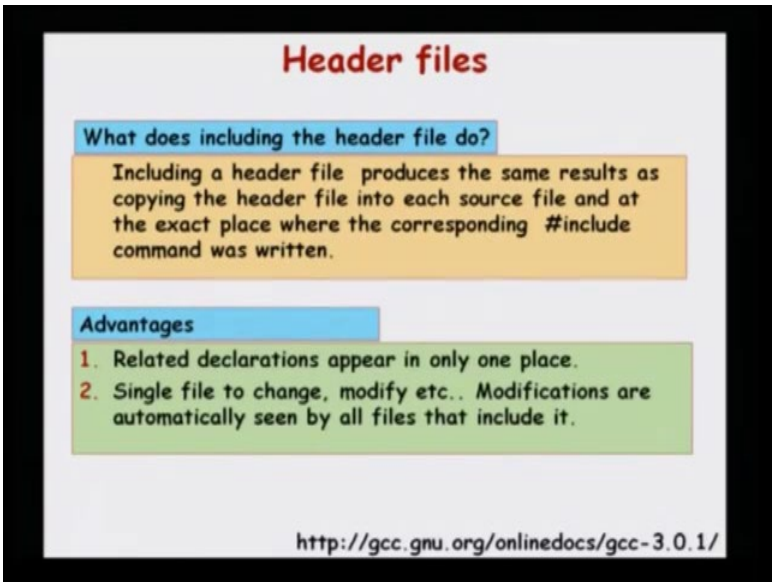
We use statement such as #include <stdio.h>; also #include “list.h”. And you may have seen C code which looks like this. You say #define PI to be 3.1416; #define MAX to be 9999, something like this. So lines in a C program, that is start with a hash symbol are called macros. And they are viewed and they are processed by the C preprocessor. Now, the C preprocessor implements what is known as a macro language; part of C. And it is used to transform C programs before they are compiled. So, C preprocessor is the step just before compilation. We do not explicitly call the C preprocessor. But when you write gcc, some file name, gcc the first step is the preprocessor step. So, as part of the compilation, first the C preprocessor runs and then transforms the macros. The resulting file, including the transformed macros is compiled by the C compiler.

(Refer Slide Time: 01:33)



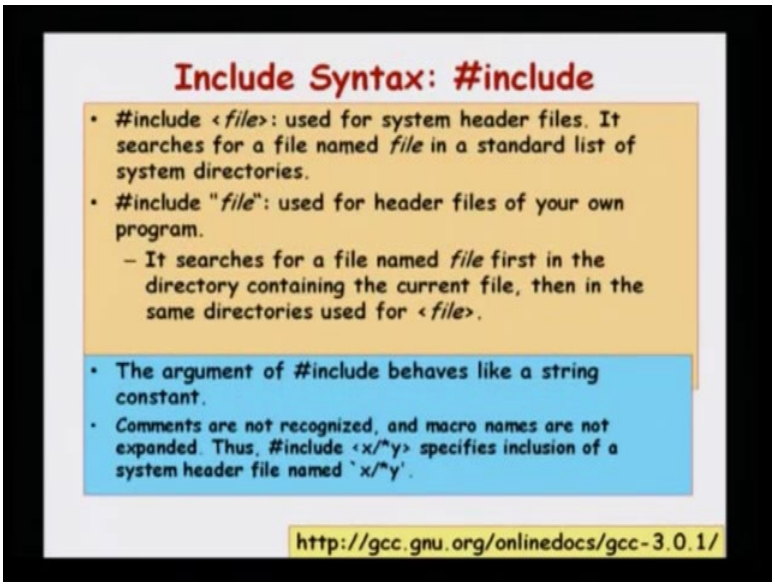
So, let us look at header files. A header file is a file containing C declarations, macro definitions etcetera to be shared between several source files. Header files are included in your program using C preprocessor directives “hash include”. For example, we have seen <stdio.h> and within quotes list.h. So, header files serves two purposes that we have seen. First is that it could be a system header files. This declares interfaces to part of the operating system including system calls, C libraries and so on. Or you could have your own header files, which you have written to contain declarations of your code of the functions in your code.

(Refer Slide Time: 02:31)



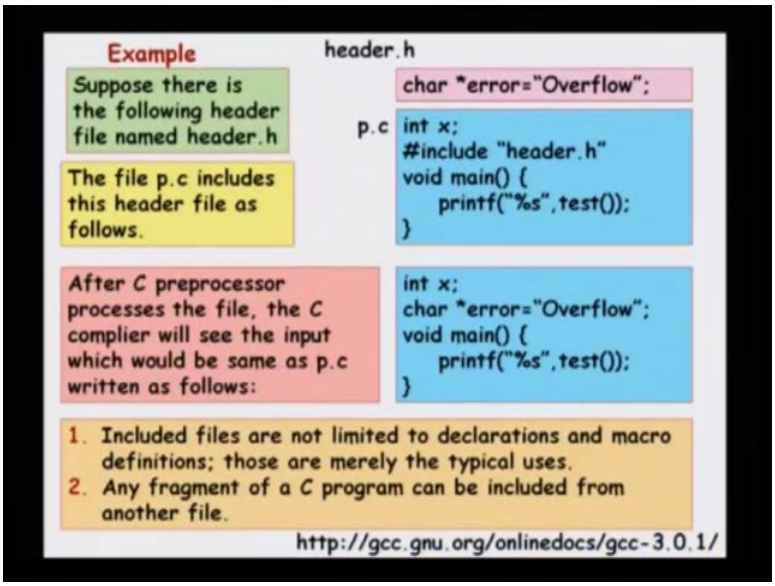
Now, what happens exactly when you include a header file in a C file? Including a header file produces the same results as copying the header file into each source files. So, when you say include<stdio.h>, it is essentially taking the contents of the <stdio.h> file and copy, pasting in to your source code. So, this happens at exactly the place where the corresponding hash include command was written. The advantages of having these header file and hash include is that related declarations appear only in one place. And if you want to change a particular function or declaration of a function, you just have to change it in a single file. And all files which include that header file will automatically see the change.

(Refer Slide Time: 03:24)



So, here is something that we have mentioned in the previous video. If the difference between angular bracket in the double quotes; so the angular bracket is usually used in system header files and it searches for the file named let say <stdio.h> in a standard list of system directories. If you say within double quotes, on the other hand like list.h, it searches for this list.h first in the current directory. If it is not found in the current directory, then it goes again in to the standard list of that. Now, the argument for hash include; whether you include it in a angular bracket or in a double quotes, it behaves like a string constant and it is literally put there. So, if you have like comments, the comments are not recognized as just comments. If you have a * symbol, for example, it will be just put exactly like a * symbol. So, it is just treated as a string constant and no interpretation is done.

(Refer Slide Time: 04:30)

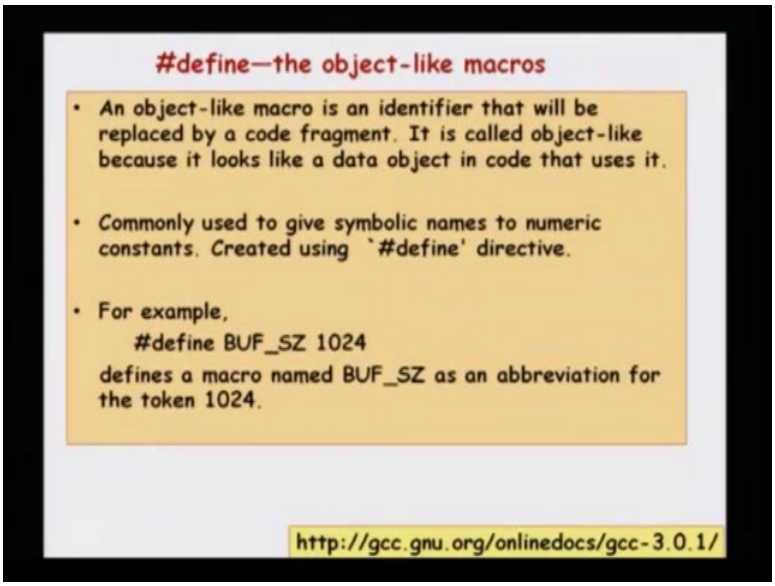


Now, let us look at a very special case that could happen in the header file. Typically, you would not do this. So, suppose you have; within the “header.h” you have a text char *error=“Overflow”. Typically, you do not initialize variables in a header file. But, let us say that in a particular “header.h”, we have this char *error=“Overflow”.

Now, in p.c I write this very peculiar thing. I write int x and then in the middle of the code I say #include “header.h”. Till now we have always used hash include headers in the beginning of the file. But, suppose what happens if I do it in the middle? Now, after the C preprocessor processes the file, what happens is that whatever text is there in “header.h” is copy, pasted at that position. So, for example, this code will be transformed by the C preprocessor to look like this. It will say int x and the “header.h” contain the single line; char *error=“Overflow”. So, that text will come here. Now, this transformed text is what the C compiler sees and it will compiled in and produce the object code or the executive.

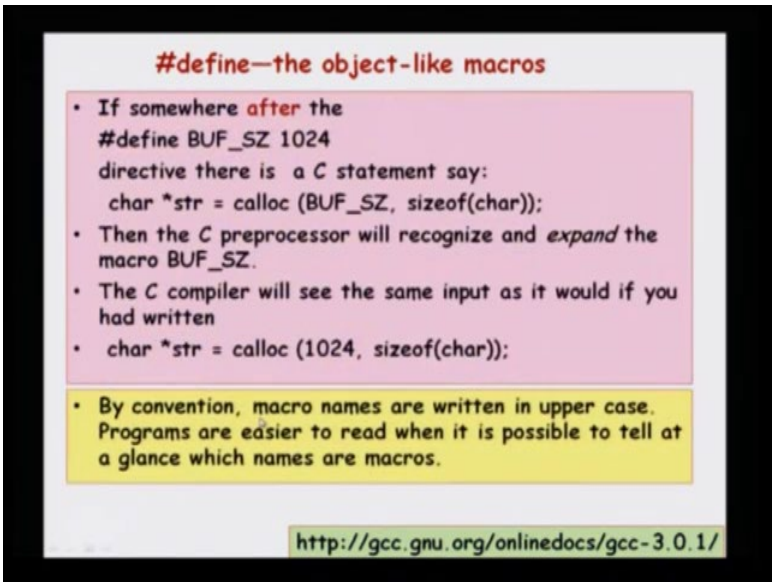
So, included files are not limited to declarations and macro definitions, these are merely the typical uses. You can put any text there. And when you include that header file, the text will be copy, pasted into the position. Typically though you would want to avoid this, you would want only declarations in a header file.

(Refer Slide Time: 06:13)



Now, let as look at some other features that the C preprocessor gives. We have seen in some code, this use of #define. So, #define is used for what are known as object-like macros. An object-like macro is basically an identifier and it will be a replaced by some code text. It is called object-like because it looks like an object. So, its most common use is to give symbolic names to numeric constants. Suppose, you want to write a program in which the maximum array size is let us say 1024, instead of putting 1024 in several places, the typical usage in a program would be to say #define BUF_SZ; so buffer size to be 1024. So, you have used #define to define this identifier BUF_SZ and BUF_SZ will be assigned the text 1024. So, this says that I am defining a macro named BUF_SZ. And it is an abbreviation, the short form for the token 1024.

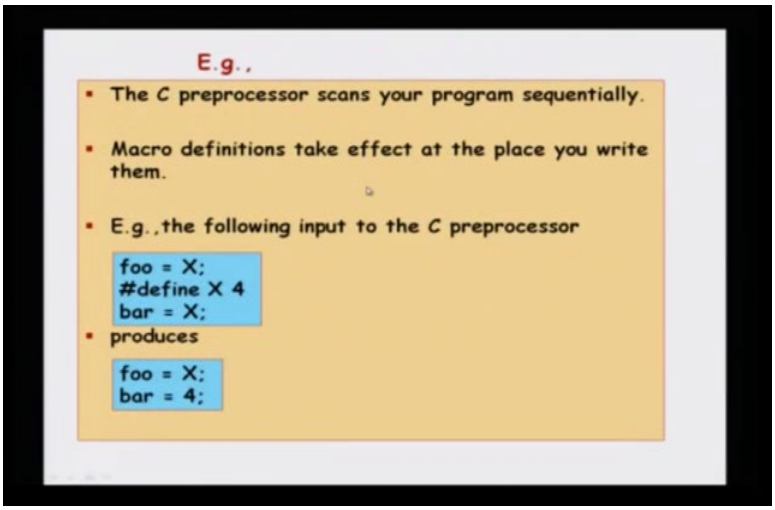
(Refer Slide Time: 07:34)



Now, if somewhere in your code if you say #define BUF_SZ 1024, in all places after that, suppose you say like char *str= malloc or calloc(BUF_SZ, sizeof(char)), what the preprocessor will do is that it will look at this string constant. It is the name of a macro. And it will replace it with 1024 which is value of the macro. So, the transformed text will look like this; char *str= malloc or calloc (1024, sizeof(char)).

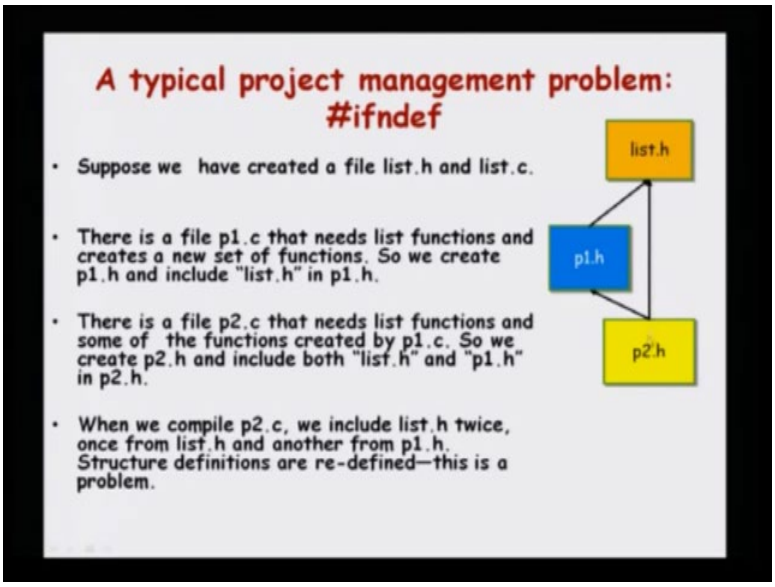
Now, by convention macro names are return in upper case, so that somebody who reach the code will be aware that code this could be a macro; because if I write it in lower case, there are chances that somebody would think, that it is a variable name and look for the variable. So, writing it in capital letters is a way of indicating to the user that this is actually a macro. So, please look at in a header file for example.

(Refer Slide Time: 08:44)



So, the C preprocessor scans through a program sequentially. This is an important thing to understand. And macro definitions take effect at the place you write them. So, let us look at a very specific example to illustrate this point. So, suppose you write foo = X, after that you have a line say #define X 4 and then bar = X. What will the C preprocessor do? It will look through the file and say foo = X. fine. It does not know what X is. It will not transform that line. Then it sees the #define X 4. Now, it knows that X is a macro and it has the value four. And then it sees bar = X, but now X is a macro. The preprocessor knows about this. So, it will replace X with four. So, that transformed text will be foo = X; bar = 4. It is natural to imagine that I would have four = four. But, that is not what happens; because the way the source code was written, the #define happened after four = X. So, anything that happens before the macros was defined is not changed.

(Refer Slide Time: 10:13)



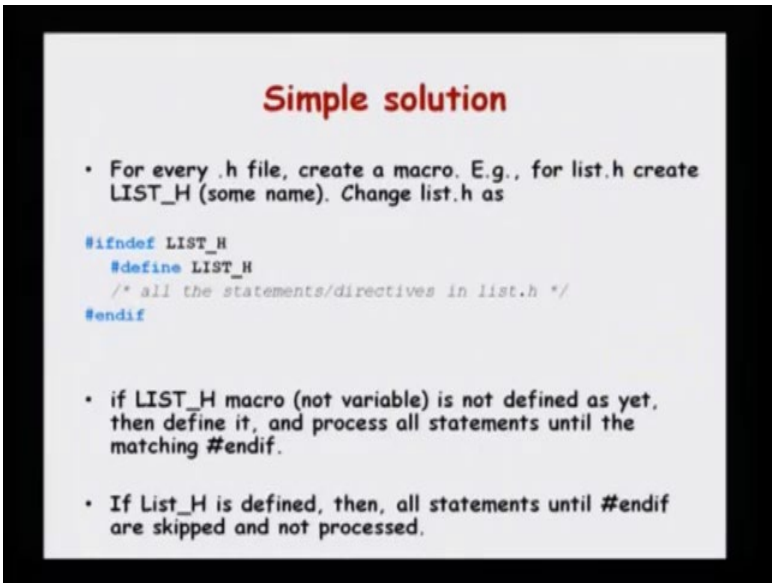
Now, let us conclude this discussion of preprocessor with a very typical project management problem. And we will see a third macro, that is, third operation that is done by the C preprocessor. This is something called # ifndef. This is used typically when you have multiple files in your project and you need to include multiple header files into a single a source file.

So, let us discuss what is the problem with the particular example? Suppose, we have a list.h and the list.c So, I have this header file list.h. Now, I have a program p1.c; that needs the list functions and also creates a bunch of new functions. So, its declarations will be included in p1.h. Now in p1.h, I would say include list.h. So, this is okay. I will have a corresponding p1.c, which will we just say include p1.h. Now, suppose that I have another file p2.c; the p2.c needs some functions in list.h and some functions n should p1.h. So now, there, when I write p2.h, I will say include p1.h and include list.h. Now, what happens is that when we compile p2.c, list.h gets included twice. First, because it indirectly includes list.h, and second because it includes p1.h which itself includes list.h. So, list.h code will be a copy, pasted twice in p2.h.

So, for example, this is the problem, because if list.h contains a structure definition, it will be included twice and the compiler will produce an error. So, this is the standard problem in large projects; where you want to include a file, but you do not want to

include it multiple times. So, in this particular example I want to improve list.h, but I want to avoid the possibility that list.h is included multiple times leading to compiler errors.

(Refer Slide Time: 12:33)

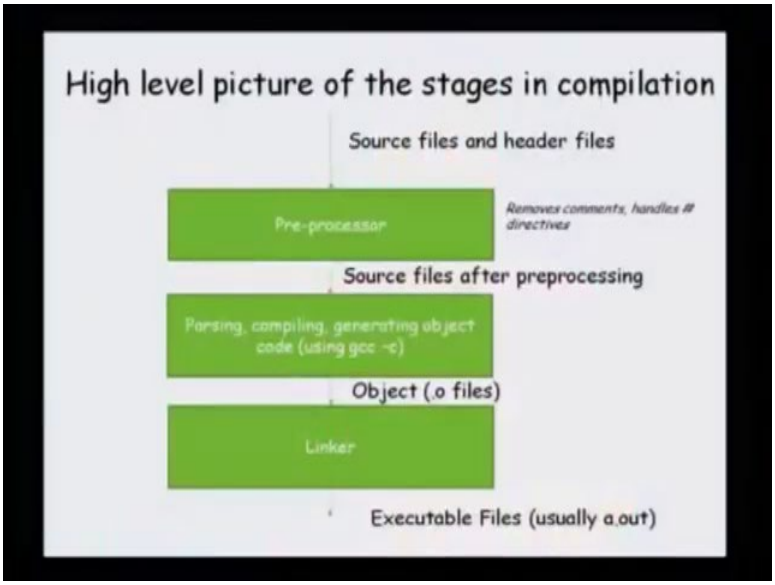


So, how do we solve this? So, this is a facility provided by the C preprocessor. You can say you can use this #ifndef. So, it is saying that if this macro is not defined, then do something. So in list.h, I will write the following: #ifndef. This is a macro that I will define. So, usually if a file is list.h the macro will be named in capital letters; LIST_H. If this macro is not defined, then #define list.h. So, this says define list.h from me and then all the remaining statements in list.h. And then it will be enclosed in an end if.

So, now, what happens is that suppose list.h is included for the first time in p1.h, then list.h is not defined. So, it will define list.h and then include list.h in p1.h. Now, p2.h includes p1.h and list.h. So, now when list.h is included for the second time, the C preprocessor will look at this statement; ifndef LIST_H. That has been defined because p 2 p1.h has already defined. It calls it to be defined. So, it says that LIST_H is defined, so I will skip the entire thing until ifndef. So, this is one way to say that. So, if LIST_H macro is not defined as yet, then define it and process all statements until the matching ifndef. If it is already defined, this happens when you are trying to include it for the second time, then all statements until the ifndef are skipped. So, you do not copy, paste it

for the second time. So, this is the standard way to avoid including one file multiple types.

(Refer Slide Time: 14:40)



So, the high level picture of the stages in compilation. You have; we take up. So, the high level picture of the stages in compilation. You have source files and then it run through this preprocessor, it produces the transformed files. And then after compilation using gcc -c, it produces object files. And after the object files are done, they are linked to produce the executable files. So when you press gcc, some source file, internally it first runs the C preprocessor, then it runs the compiler and then it is runs the linker. And gcc provides facilities to stop the compilation at any stage. And so for example, we have seen in the previous video that you can stop the compilation just after the compilation itself by using gcc -c. So, it will produce .o files. And several .o files can be later linked to produce the a.out file.