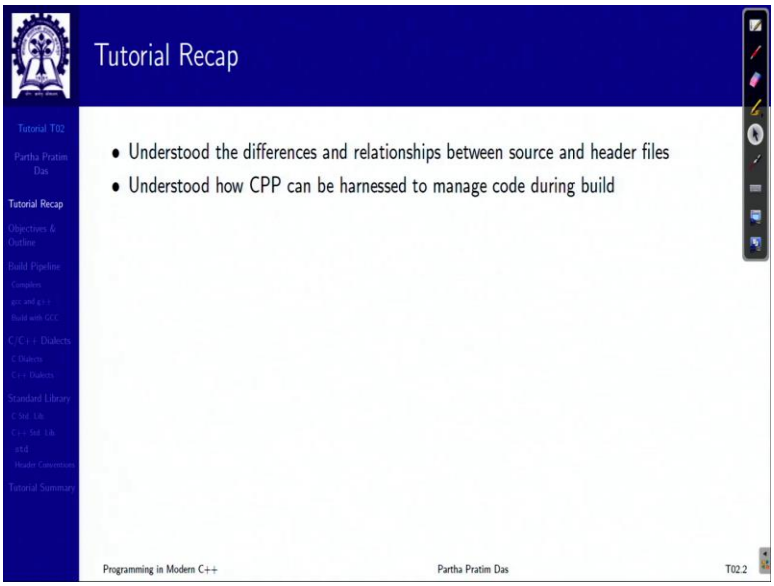
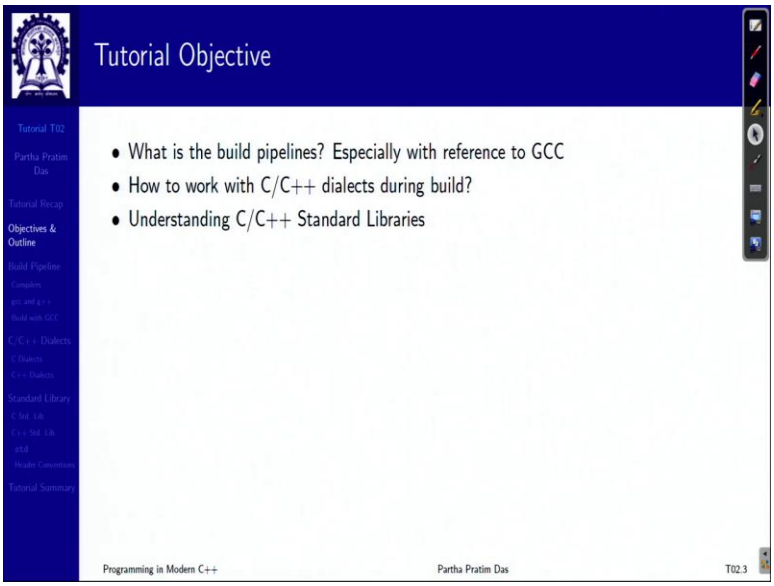
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Tutorial 02
How to build C/C++ Program?
Part 2: Build Pipeline

Welcome to programming in modern C++, we are going to discuss the second tutorial.

(Refer Slide Time: 00:34)

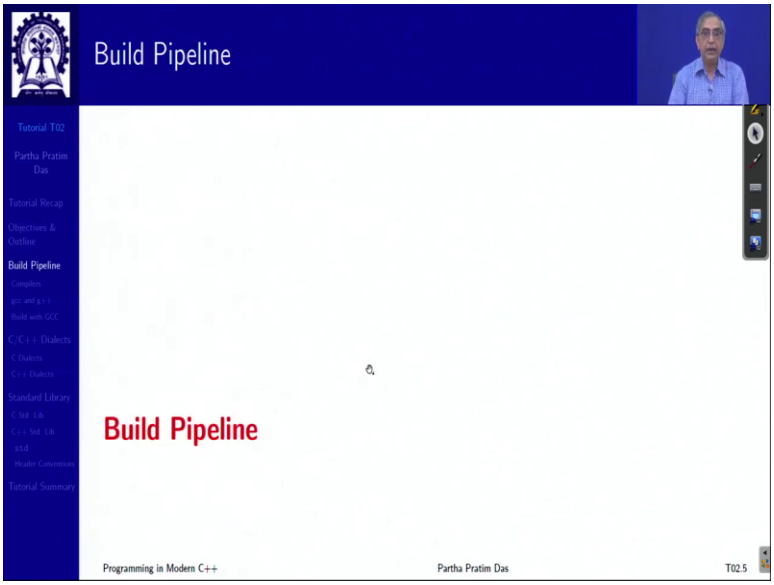


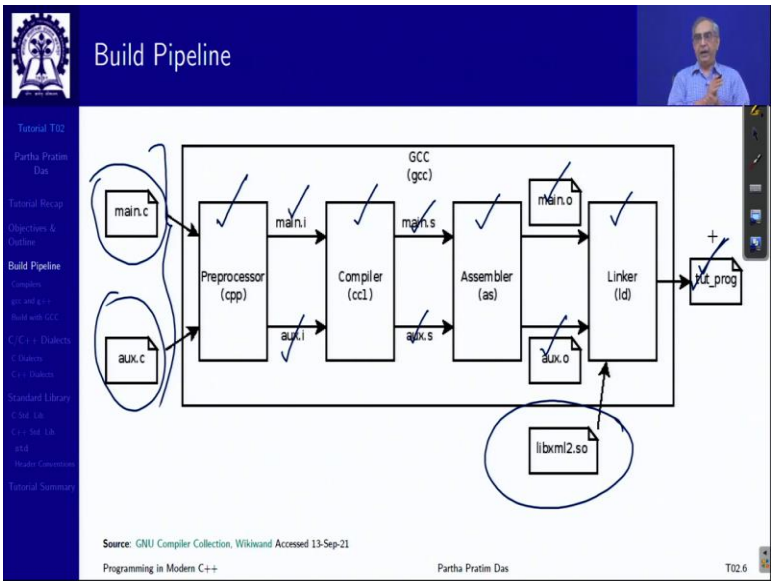


In the first one we have talked about what is the relationship between source and header files and what are the differences and particularly we took a step by step look into how does C preprocessor, CPP can be harnessed to manage code during build. So this tutorial is also continuing on the project building process.

We are going to discuss about the build pipeline, specifically with reference to GCC which is the compiler we have been using, how to work with C or C++ dialects during the build and understanding little bit more on the C++ standard libraries. So this is the outline.

(Refer Slide Time: 01:37)





So let us start with the build pipeline. Let us take a look this is where I have my source files, the translation units we have talked about. In tutorial 1 we have discussed how CPP changes that file, replaces the include, the #define, does conditional compilation and so on so forth. After that it generates the corresponding C or C++ source files which does not include any preprocessor directives.

It is a pure C or C++ program, there is all library, headers, everything is included in that one single file that is typically has an extension .i. Then the compiler will actually do the compilation process; will generate the assembly corresponding to this that is a huge translation process. We do not need to understand how it is done but it is translated into the assembly of the processor on which you are targeting.

And those files generated corresponding to every translation unit is typically designated by .s. Then the assembler ticks in, automatically we do not have to do anything for that and that will convert the assembly language program into a binary code, which is known as the object file.

It is a binary code, it is no more assembly is still has a textual representation, you can read the assembly, you can write assembly but this binary code is just bit patterns, so it is not human readable. These are object files, so these are shown by the extension .o. Then the linker kicks in. What does the linker do is you have seen that we have say in a stdio.h we have printf function.

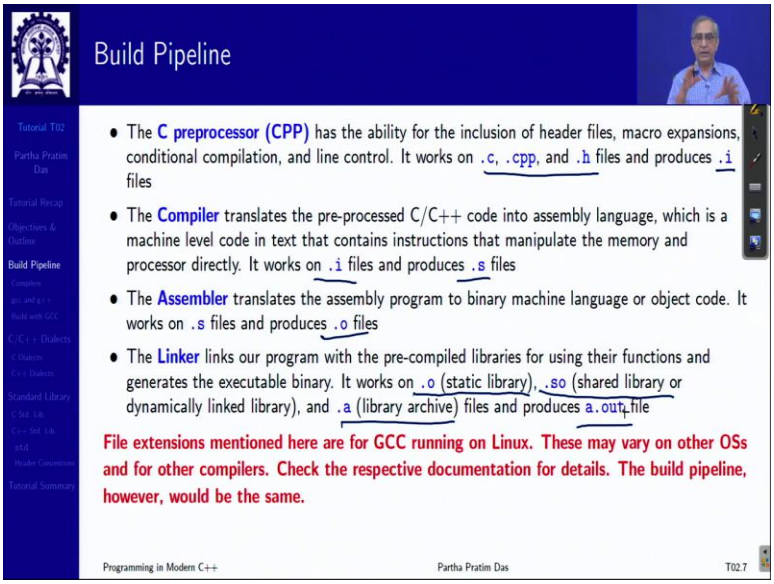
So stdio.h has just the header of that function, but the actual body of that function is in some other translation unit, which is already compiled and available as .o. Now from the main if I am giving a call to printf then naturally I need the body of that function in terms of the object file, the printf function to be included in my executable.

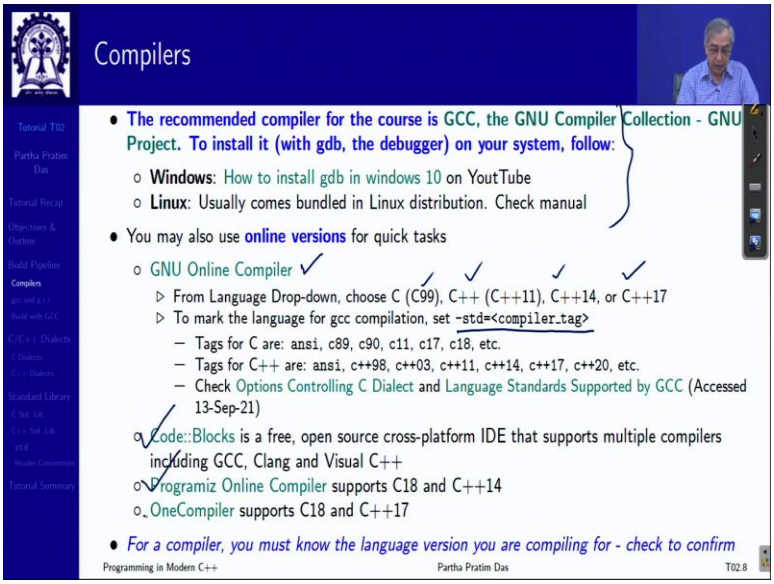
So linker is that part of the compiler which actually finds out for every translation unit what are the symbols, say functions and global variables, which it uses but does not define within it and those which it defines and makes available for others. So it relates them across the different translation units, so that finally I can put all the object codes of all translation units into one single binary file which is the executable file.

While doing this the library codes that are already available in the system, they are already compiled in the object file are also looked for, they are also linked, there are a variety of libraries we will see, here is the particular one being shown is a .so library called the shared object library, there are other kinds as well.

So when we just do GCC or g++ this whole thing happens taking this collection of all translation units that are involved in the project and finally generating an executable. So this goes by stages, so what we will do is we will see what happens in every stage and we will see options as to if I want to look at the output of any stage in the build pipeline to understand better then what are the options to do that.

(Refer Slide Time: 06:06)





So having said that let me now move on to the actual pipeline, so you have a C preprocessor which takes source and header files, then the compiler and generates a .i file. Compiler comes in compiles and generates the assembly language .s file, assembler translates the .s into .o object file, linker links all .o files with the existing libraries, these are called static libraries .o or shared library or dynamically linked library.

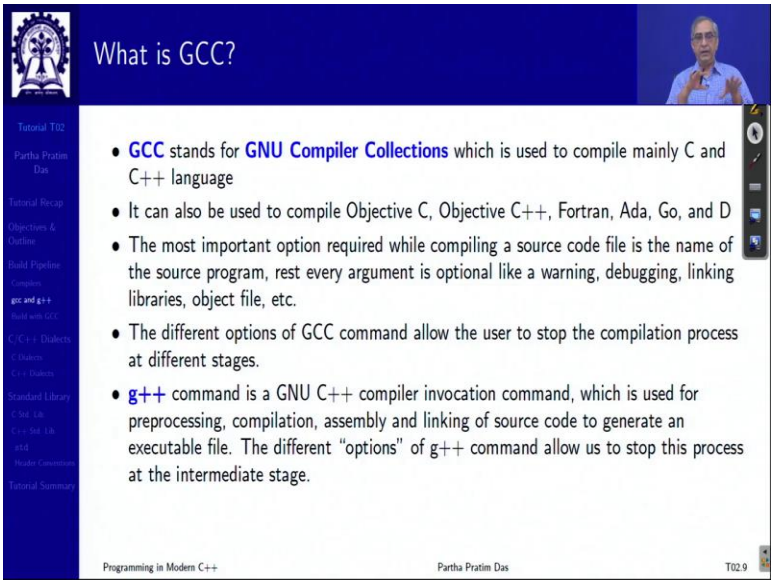
We will have a separate tutorial on this to discuss what does that mean, or library archive, we will also talk about that and finally produces a single executable file, a.out. Now these extensions that I have mentioned is for a GCC running on linux on a different OS, on a different system, these extensions could be different.

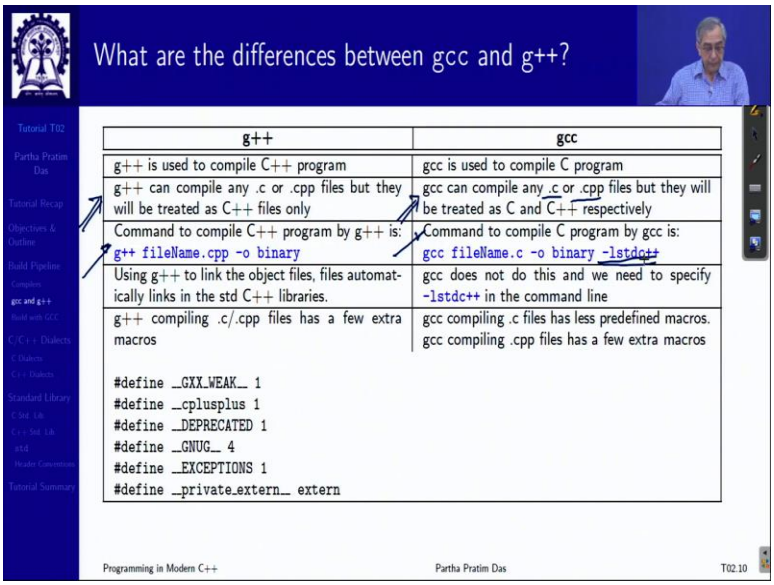
So you will have to check the documentation of the compiler to see exactly what are there. I mentioned this earlier also but here I would reiterate that you must have the GCC installed in your local system. If you are using windows, then use MinGW, I have given the links and all that earlier, if you are using linux, usually it is included in the linux distribution, check that manual.

So with this you will be able to every example that we are discussing in terms of the modules you should be able to just make them into source and header files and compile them using the GCC g++ to get going. Now if you want to do a quick check of a small code and so on, it is not good for big projects but if you want to do a quick check then you can use online versions of various compiler.

Online available compiler like GCC GNU compiler is what I will again highly recommend which has a variety of different language dialects to choose from and you can use tags like in here, we will talk more to select which particular version or which particular dialect you want to compile with. Then you have from similar online compilers from CodeBlock, Programiz is one compiler and so on, and while you are using the compiler you must know the version or the dialect of the language you are using.

(Refer Slide Time: 08:50)





What is GCC; if you have a question then GCC is a GNU Compiler Collection. It is mainly referred to for C and C++ plus but there is a huge range of languages for which the GCC compilers exist, starting from Objective C, C++, Fortran, Ada, Go, new D language and so on so forth. Now there are different options which will allow the GCC compiler to be controlled at different stages and that is what we will now take a look at.

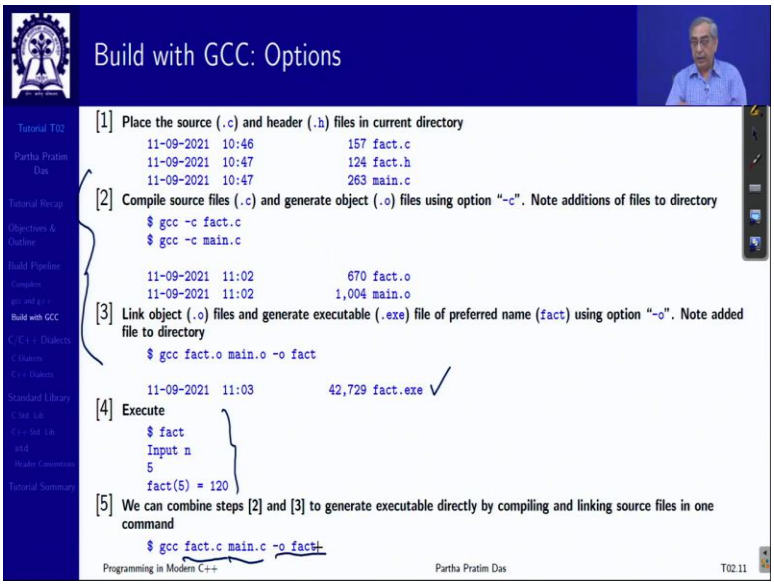
So you would have noticed that I am talking about compiling with GCC or compiling with g++, so there are two compilation commands in the GNU compiler collection with reference to C/C++ bunch. So you use GCC to compile typically compile C program and use g++ to compile C++ program and the difference is what actually they are the same compiler but it is a different set of rules that you are applying.

So this second point is most important that is g++ can compile any .c that is C source or .cpp that is C++ source but whether it is a C source or a C++ source if you use g++ it will always treat it as by the C++ rules only. So even the C program will be compiled with the C++ compiler, so if there are compatibility issues you will have problems.

Whereas GCC makes a selective choice, GCC decides which rules to apply based on the extension of the program. If there is it is a .c extension then it will be treated by C rules it will be treated and compiled as a C program, if it has .cpp extension that will be compiled as C++ program respectively, so you can make your choice.

Now when you compile with g++ you are doing a C++ compilation, so it does automatically link all your necessary standard libraries I should say, whereas when you do for GCC and you are specifically want to build a C program then as a final executable then you will have to link the standard library of C++ if you need to include that. So these are some of the things, let us go to the command chain and you will see all of that happening.

(Refer Slide Time: 12:00)



So this is just by stages of the pipeline, so initially what you do to keep things simple just put all your source and header files into the correct directory, you can distribute, have organization will talk about all those later but just to see the compilation process put all your files in the current directory and if you check the directory with ls or dir depending on the system you are on then you will see something like this the output that I am showing here are from my windows 10 machine using the command prompt.

So here I have three files one is a factorial header fact.h, one is a factorial source which has a function implementation fact.c, and then there is a main program which is using the factorial function including fact.h. So you can see that there are three files there in, now we can compile using, so you have to compile each and every translation unit each and every source files you will have to compile, this you have to compile this.

So when I compile I am using an option -c, typically when you say minus on the command line of the GCC or g++ it means an option which does behave in a certain way. So if you put -c what it does it will generate the object file in this manner, so after you have compiled both with -c you can again list the directory you will have the source and headers also I am just for brevity I have not shown them but these two files get added that the object files have been created.

Now you can use those object files fact.o and main.o so you have come up to the object file directly so all that you need to do is to take all translation units and try to link them together. The reason you need this -c here is in this case none of these programs or none of this source files are independently complete to be converted into the final binary executable.

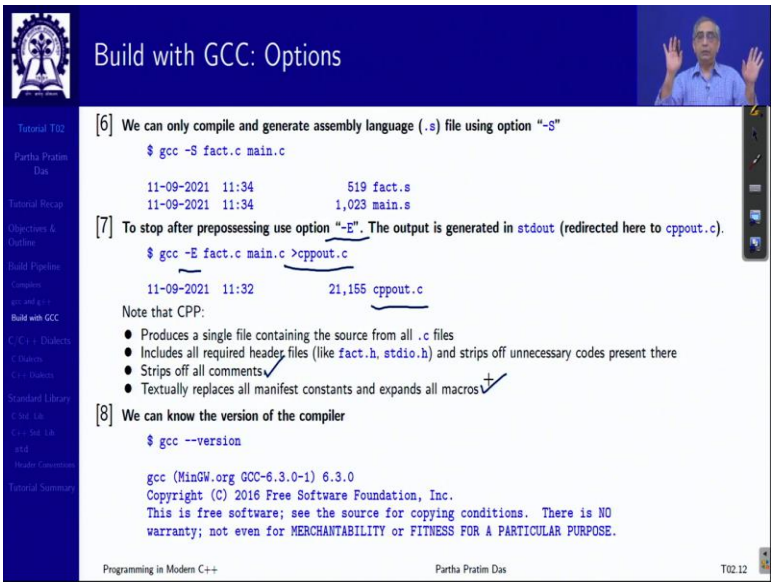
So you want to tell the compiler that you compile up to the object level but do not try to link, if you just do GCC main.c, the compiler will generate main.o and then start shouting that I cannot find where the fact is. Similarly if you just compile fact.c it will complain that there is no main function in this source ok so you need to stop it at the object level and then take all of these objects to link together.

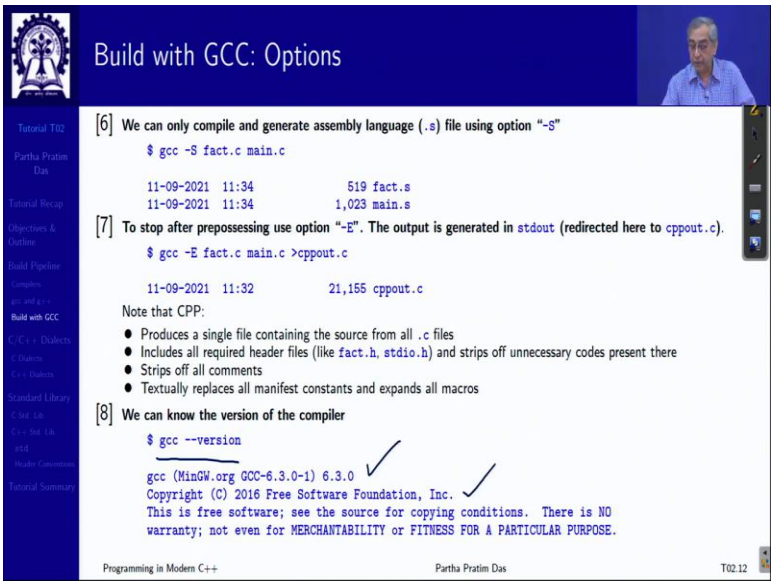
And you do not have to particularly call that linker like ld and so on you do gcc again and the fact that you have given .o file that gcc knows that this is the object file which needs to be linked, and if you do not give this remaining part then it will generate in a.out or a.exe depending on the system you are on but if you want to give a specific name to the binary executable that you are creating you can do that by using the -o option.

So I am calling it fact right so once I have done that my executable is ready and I can see that added to my directory folder fact.exe because I am on windows, and then I execute fact and this is, so this is a basic, you know step to build. Now I can also combine these two and three together by giving multiple source files translated in units to the command line at one go.

So gcc, fact.c, main.c will tell gcc to compile and generate object file and then try to link them all together generate a binary so this can be done in a simpler way like this in one step also.

(Refer Slide Time: 16:26)





Now if you want to you know see what is happening at the intermediate stages then you can use different options. For example, you can tell gcc that just do the compilation and stop at the assembly do not kick in the assembler, use a -s option. If you use -s option then the compilation does not go up to the object file, it stops after the translation only at the assembly level.

So with this different translation units that you have given you will see that you have two .s files, you can use a text editor to open it, and you will see that the assembly language program corresponding to your C program has been created in that. Of course, to understand that you will have to know the assembly language for your system, for your processor.

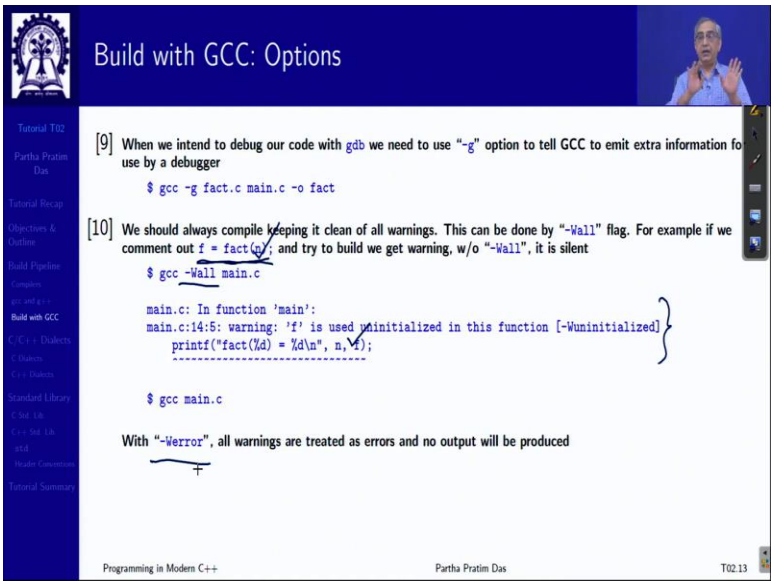
If you want to stop even earlier that is after preprocessing, we just want to see that what does CPP has done, it is a great learning that after replacing #include, #ifdef, #define and all that what is that you get then you can use the -E option now this will generate the output into your stdout that is your standard out on the on the console itself.

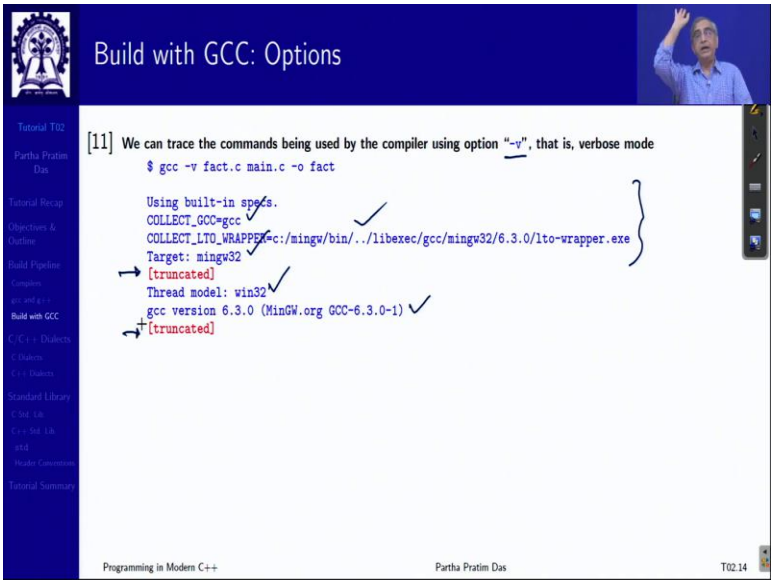
So if you want to, and it usually is huge, so if you want to keep it and see separately later then you can redirect, this is the redirection greater than, redirect to a different file and I have specifically called it by an extension .c, because after the replacement the it is a purely C program. So it will also strip all comments.

It will textually replace macros, and all unnecessary codes which you had done #if zero or something are all removed, it is a fun to see that please try it out I cannot show you on slides because it is just too huge to show. Now, finally if you want to know what is the version of your compiler not the language, language dialects will come separately but what is the version of the compiler, compiler is regularly getting released.

What is the version that you are using then you can always do --, there are two minus required --version and it will tell you the version where is it coming from and so on so forth.

(Refer Slide Time: 19:23)





Now often we would like to do a debugging of the code that is not just run it but start running it, stop in the middle and of execution, check what is the value of the variable, may be set a specific value to a variable give a break point and so on I will talk separately on debugging in one of the tutorials because that is a very important process and gcc has a corresponding debugger called gdb which is gcc debugger.

Now if you want your code to be debugged then your normal compilation will not do so you will need to give it -g option. So everything else remains same you give a -g option by that the final binary that gets generated is empowered with annotations for the debugging and also it does not do any optimization of your code, so that you can really go and keep on during the execution you can keep on checking.

So, this is something which you will very frequently need. I have a recommendation for you that besides the errors if the compiler has an error naturally after that stage you cannot proceed you

have to fix that error but compiler also gives a number of say warnings. So he says this is the warning and so on.

For example if we comment out you will have to refer to the previous tutorial to see the code for example if we comment out this and try to build then we will get “f is used uninitialized in this function” that is I am trying to print f and I have not computed f. So f has not got any value yet, so language wise it is not an error but programming wise certainly the compiler is wondering why do you want to print a value that you have not even initialized.

So the compiler does not say it is an error because it is a correct program otherwise program logic is programmers prerogative but the compiler wants to warn. It is good to build with -W all. It looks like wall it is actually not that -W is to say that what kind of warning you want the compiler to give or what kind of warning you do not want the compiler to give, all says that you give me all kinds of compile warnings.

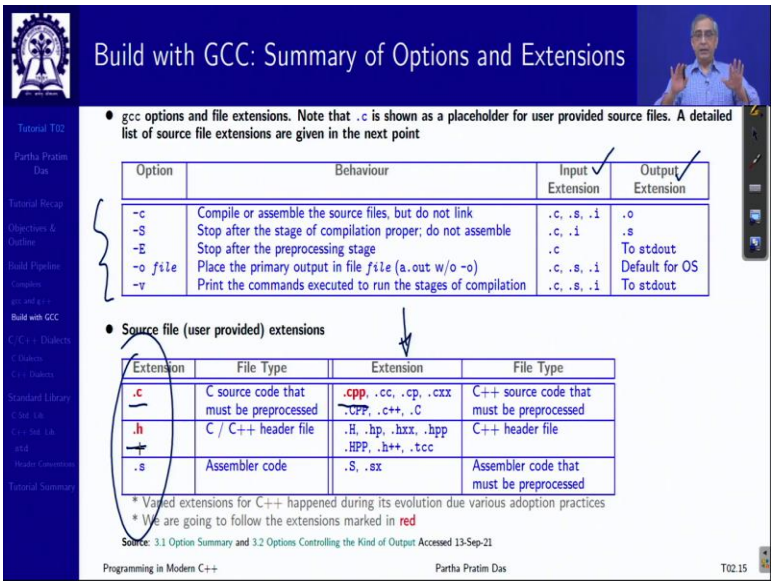
So -W all will give you all kinds of warnings and it is good to remove all warning with the warning also you can proceed but it is best to remove all warnings because with the warnings, if the warning is given there is something that is probably not correct.

So for example here you could proceed but you will certainly have a garbage value and you have to come back and check and debug but if you take care of this warning at the compilation time itself your logical error of having omitted the right value to f will get detected and fixed. You can also tell the compiler that I do not ever want to work with warning by telling that giving an option that -W error.

So then the compiler will do it will treat every warning as an error and it will not allow you to proceed unless you fix that, so these are these are the additional things you can do. Finally if you are really motivated to trace what the compiler is doing along this build pipeline from one to the other then you can tell the compiler with an option -v, v here stands for verbose that the compiler becomes talkative though usually compilers only give errors or warning messages, now it will start giving all kinds of what it is doing.

So it is hundreds and hundreds of lines so I have just shown you few initial ones it says what is that, what is the folder it is taking from, what is the target that it will generate, what is the kind of thread it will use, what is the version of the compiler, but the very right truncated there are hundreds and hundreds of lines of dump that comes in you can put -v and see for yourself. It is good fun but on a regular basis you will not be doing this of course.

(Refer Slide Time: 24:36)

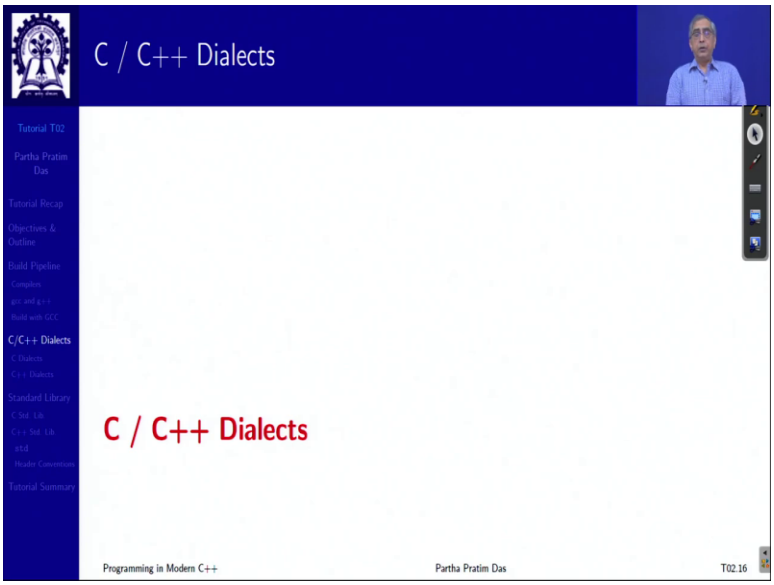


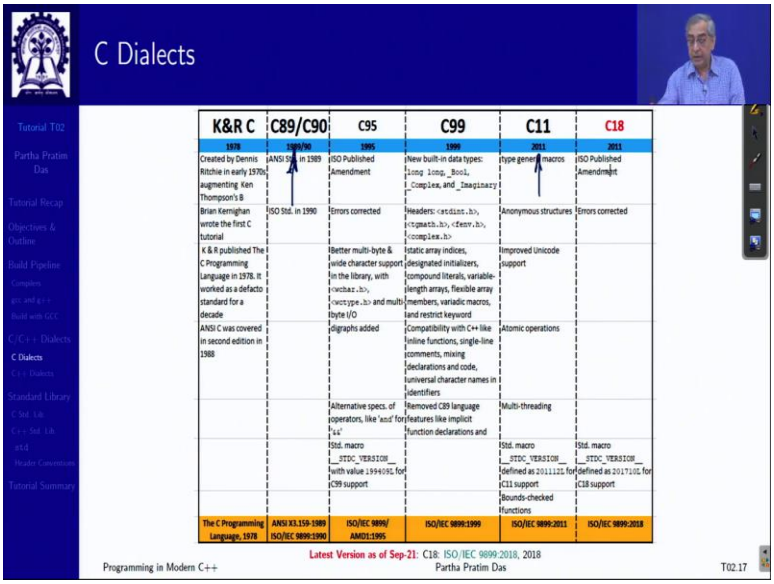
So just to summarize these are the different you can just keep this chart handy, these are you will get them in gcc manual also but I have accepted only the common part which you in in gcc manual this runs into maybe 10 pages. I just accepted that part which you will need 95 or maybe 99 percent of the time. So these are the options the behavior under that option, what is the input extension and what is the output extension.

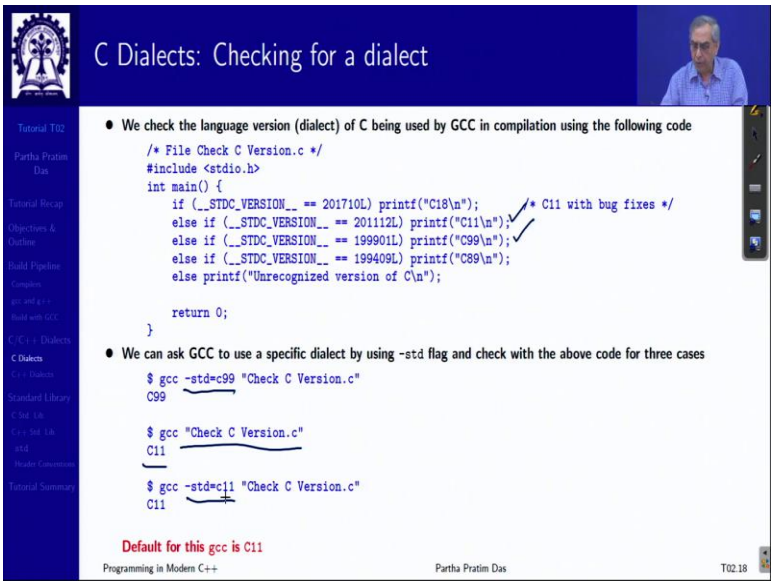
So if you have this maybe you can paste it in front of on the wall where you work so that you can at any point of time just look up and do take that appropriate action. In terms of the source file I have already talked about these are the three possible extension .c for source, .h for header and .s for if you have generated assembly.

Now the corresponding file type as you will see that I had mentioned this in tutorial 1.2, that there are variety of standards people have worked with variety of extensions, so in an existing system when you see in a say in Github or somewhere or maybe in your company you will see a whole lot of different extensions being used. So learn what extensions mean what but the most common and standard extensions are which are highlighted in red and when you are writing it try to always follow this convention and that is what we are going to do.

(Refer Slide Time: 26:19)


C / C++ Dialects
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib
C++ Std. Lib
std
Header Conventions
Tutorial Summary
C / C++ Dialects
Programming in Modern C++
Partha Pratim Das
T02.16


C Dialects
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib
C++ Std. Lib
std
Header Conventions
Tutorial Summary
K&R C
1978
Created by Dennis Ritchie in early 1970s augmenting Ken Thompson's B
Brian Kernighan wrote the first C tutorial
K & R published The C Programming Language in 1978. It worked as a defacto standard for a decade
ANSI C was covered in second edition in 1988
The C Programming Language, 1978
C89/C90
1989/90
ANSI Std. in 1989
ISO Std. in 1990
ANSI X3.159-1989 ISO/IEC 9899:1990
C95
1995
ISO Published Amendment
Errors corrected
Better multi-byte & wide character support in the library, with <wchar.h>, <wctype.h> and multi-byte I/O
digraphs added
Alternative specs. of operators, like 'and' for '&&'
Std. macro __STDC_VERSION__ with value 199409L for C99 support
ISO/IEC 9899/ AMD1:1995
C99
1999
New built-in data types: long long, _Bool, _Complex, and _Imaginary
Headers: <stdint.h>, <tgmath.h>, <fenv.h>, <complex.h>
static array indices, designated initializers, compound literals, variable-length arrays, flexible array members, variadic macros, and restrict keyword
Compatibility with C++ like inline functions, single-line comments, mixing declarations and code, universal character names in identifiers
Removed C89 language features like implicit function declarations and
ISO/IEC 9899:1999
C11
2011
type generic macros
Anonymous structures
Improved Unicode support
Atomic operations
Multi-threading
Std. macro __STDC_VERSION__ defined as 201112L for C11 support
Bounds-checked functions
ISO/IEC 9899:2011
C18
2011
ISO Published Amendment
Errors corrected
Std. macro __STDC_VERSION__ defined as 201710L for C18 support
ISO/IEC 9899:2018
Latest Version as of Sep-21: C18: ISO/IEC 9899:2018, 2018
Programming in Modern C++
Partha Pratim Das
T02.17



This was the basic stuff that I wanted to discuss with you in this tutorial; I will just quickly run through few related issues here. One is I said in module 1 discussion that there are different dialects as standards have happened, so these are the different dialects and we will keep on switching between typically between C90 and C11 this is not C++ this is C11.

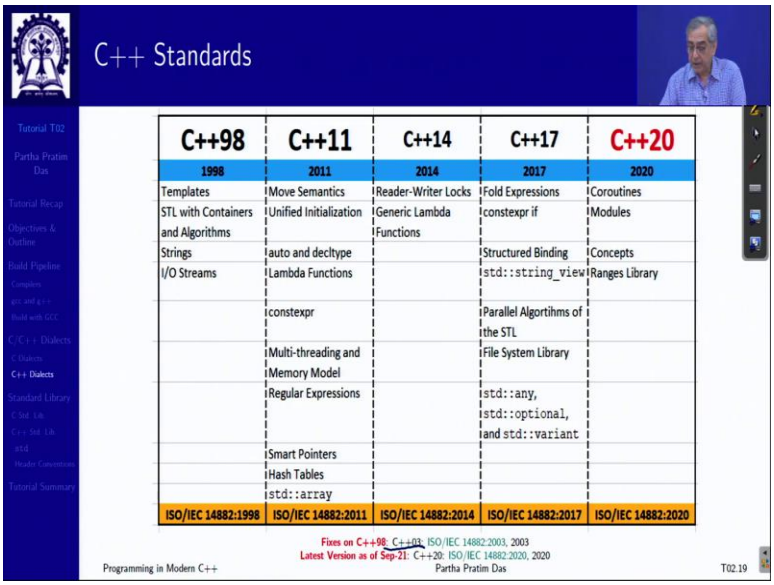
So the standard that you have this is, so when you use your C compiler say you are using gcc with .c file extension you might want to know or you might want to decide even dictate that what kind of which version should you use, because there are different support that changing from one version to the other.

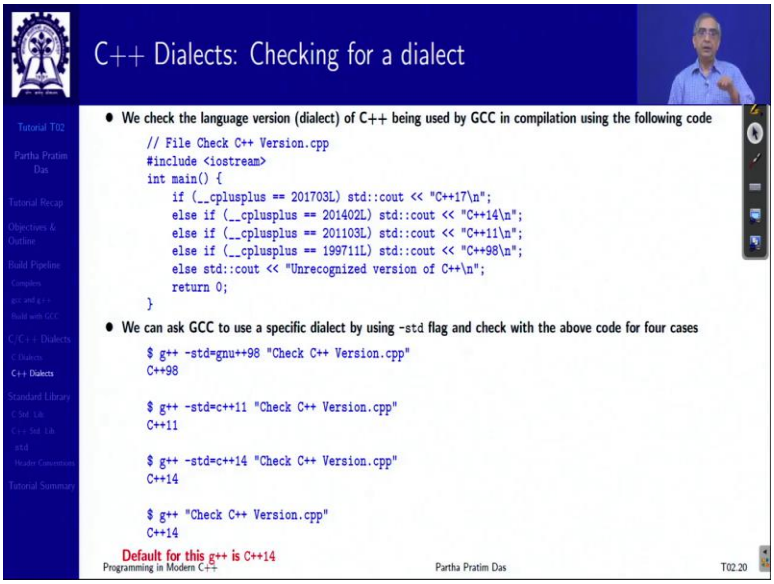
This is a small piece of code with some magic numbers which are defined in the CPPs macro for every standard, which you can use to detect which particular version you are using. So if you want to use say C99 version then you can write it as -std, -std is an compiler option which say what standard and C99 is a code for the C99 standard.

So if you run, if you build the above code with -std = c99 and run you will get this output C99 output. If you just build this of course it may be different in your system because it depends on the compiler version you have actually installed but if you do MinGW now six version plus then the default is C11, so that is the reason I said that C11 is one.

So the default is C11, so you will get this. Also you can set that I want to build with C11 so say to ensure that well I do not care about the default so -std = C11 will build the code in C11, so gcc has whole lot of different options for standards and so on I have just taken the few which are important.

(Refer Slide Time: 29:04)

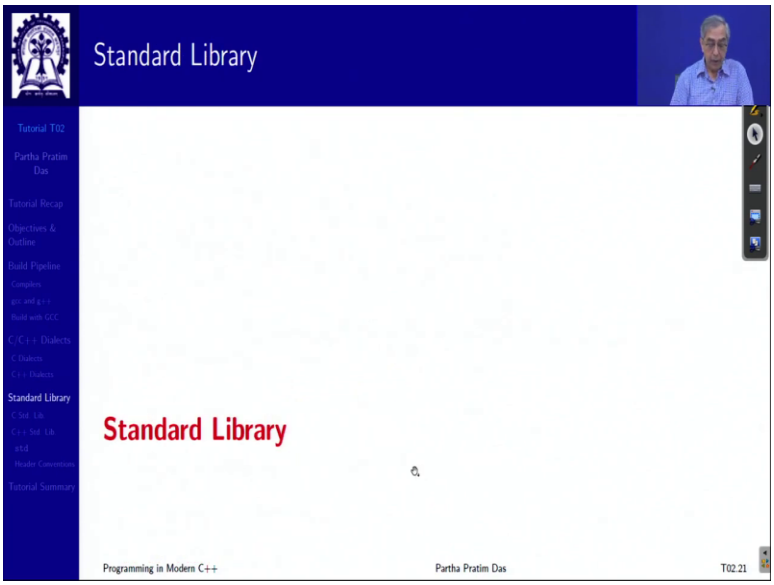


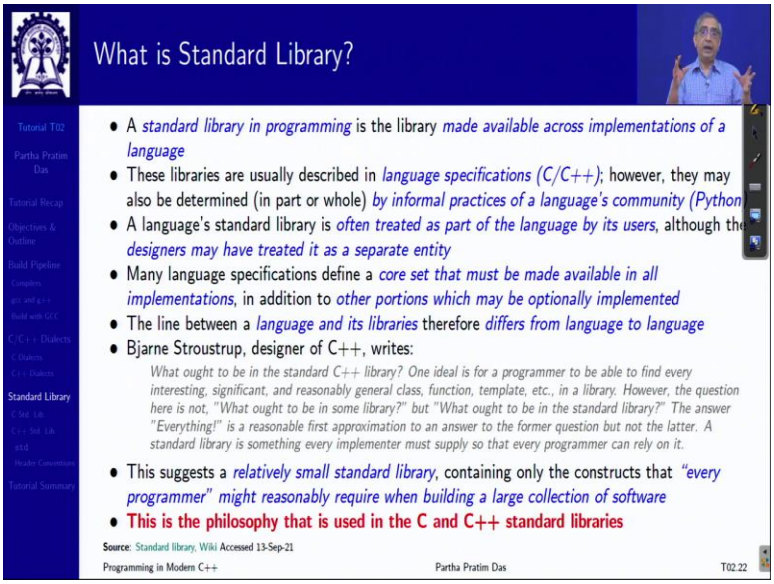


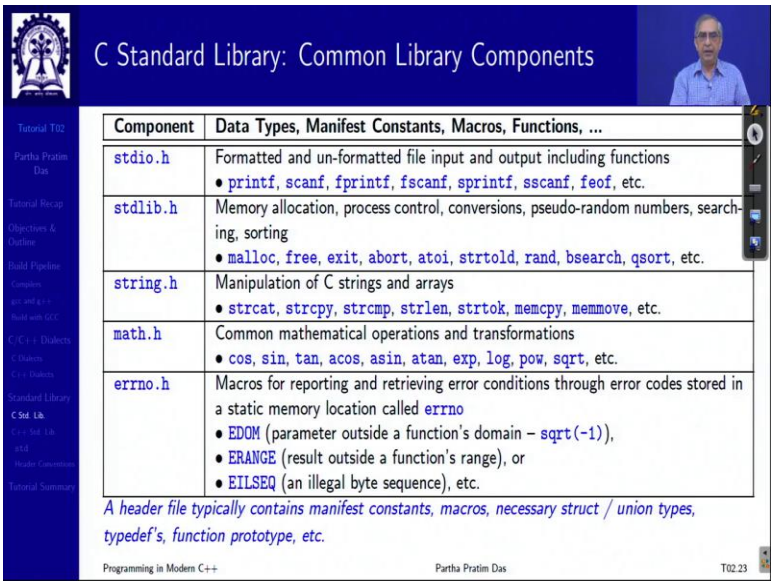
Similarly for C++ you have different dialects as from the first standard at C++98 which is what we are doing now and I mean over weeks one to about eight or nine we will talk about this standard only before we jump into the C++11. Now when we talk about C++11 we will actually talk also about some 14, 17 features may be some 20, C++20 features also but our primary choice will be C++98 and C++11.

C++98 for the first nine weeks, now also note that there is a C++03 version and you may have noticed that I keep on using these terms interchangeably I say C++03, I say C++98. Actually C++03 is the same standard as C++98 except that C++98 had some bugs which have been fixed in the C++03 dialects. So like in C here is an equivalent for ++, you can check the version you can set the version and the details you can really work out.

(Refer Slide Time: 30:30)


Standard Library
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions
Tutorial Summary
Standard Library
Programming in Modern C++
Partha Pratim Das
T02.21


What is Standard Library?
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions
Tutorial Summary
• A standard library in programming is the library made available across implementations of a language
• These libraries are usually described in language specifications (C/C++); however, they may also be determined (in part or whole) by informal practices of a language's community (Python
• A language's standard library is often treated as part of the language by its users, although the designers may have treated it as a separate entity
• Many language specifications define a core set that must be made available in all implementations, in addition to other portions which may be optionally implemented
• The line between a language and its libraries therefore differs from language to language
• Bjarne Stroustrup, designer of C++, writes:
What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not, "What ought to be in some library?" but "What ought to be in the standard library?" The answer "Everything!" is a reasonable first approximation to an answer to the former question but not the latter. A standard library is something every implementer must supply so that every programmer can rely on it.
• This suggests a relatively small standard library, containing only the constructs that "every programmer" might reasonably require when building a large collection of software
• This is the philosophy that is used in the C and C++ standard libraries
Source: Standard library, Wiki Accessed 13-Sep-21
Programming in Modern C++
Partha Pratim Das
T02.22



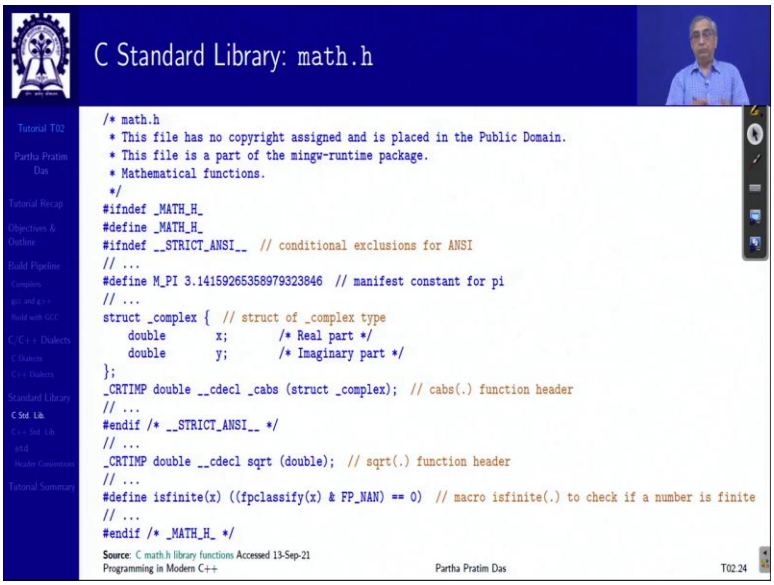
Obviously there are standard libraries as you all know and which is along with the language the standard library extends the overall capability of programming, it makes it easier. So as you learn the language it is very very important to learn the standard library, because that will make your programming really easier because lot of things like a lot of C programmers are writing a short routine without probably knowing that Qsort is available in the library, so that kind of I discuss that.

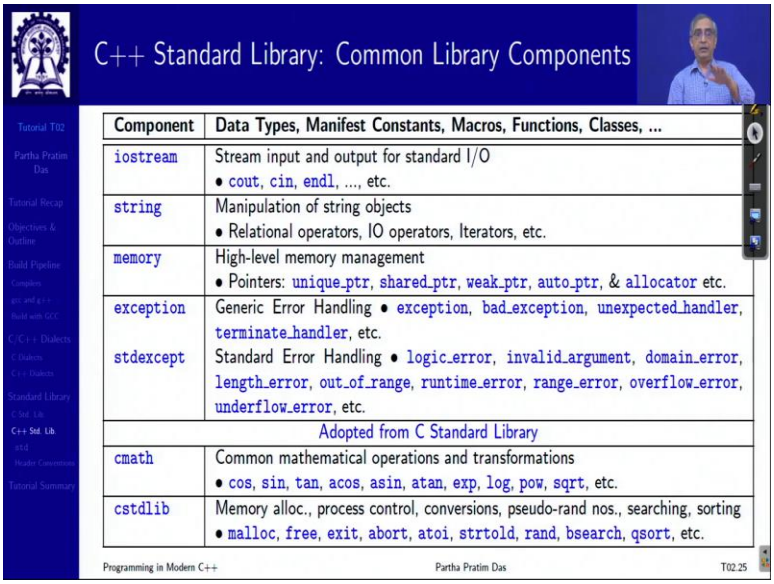
Now the question is how much is in the language and how much is in the library it depends on the languages choice. For example in C there is no string type it is a string library which provides a C++ as made string also in the library but as a more complete type. Even stronger example is dynamic memory management, in C it is a part of standard library.

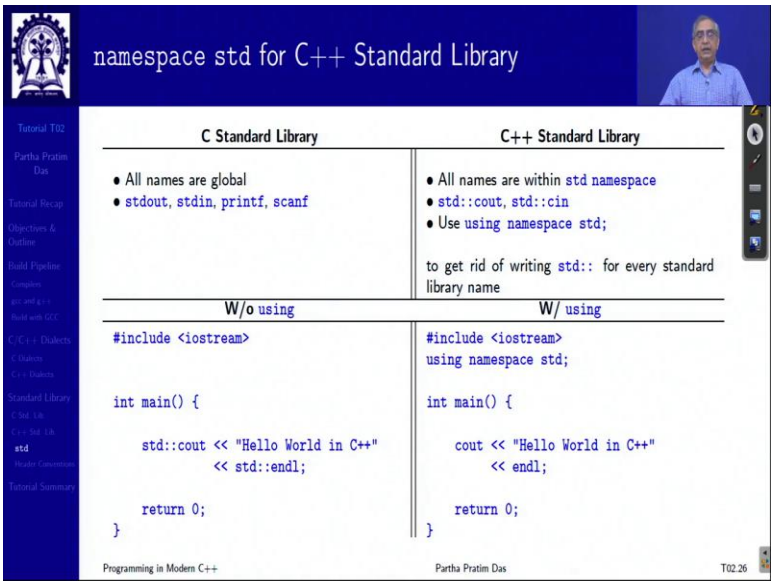
You have all these malloc free and so on; in C++ you have language operators, new delete you must have heard me on the specific modules on that. So it is a language and standard library design is a very careful thing which has to be small enough, so that people can learn and remember. At the same time it must be reasonably powerful so that a large collection of software can benefit from that.

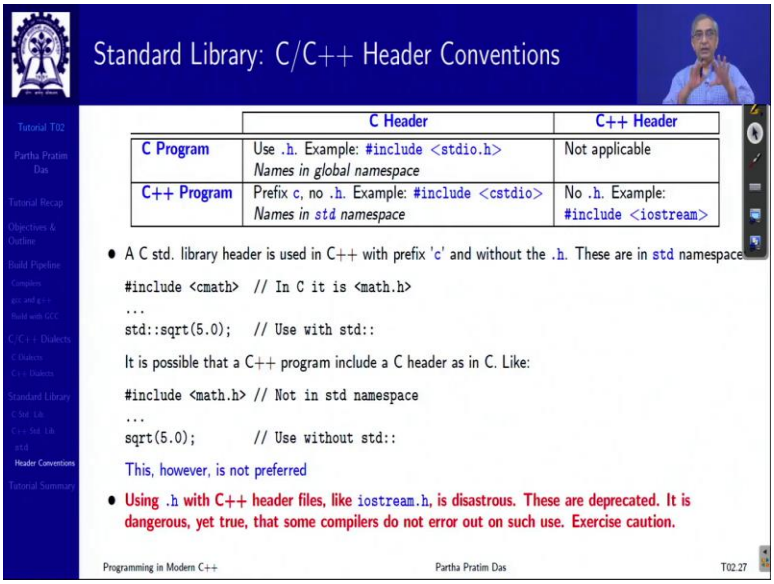
These are some of the standard I will not go through these you can just refer to the slide. These are some of the very common and useful in you must have seen that most of the examples are using this stdio, stdlib, string, math these are the four that you more often use when you handle error you will also use

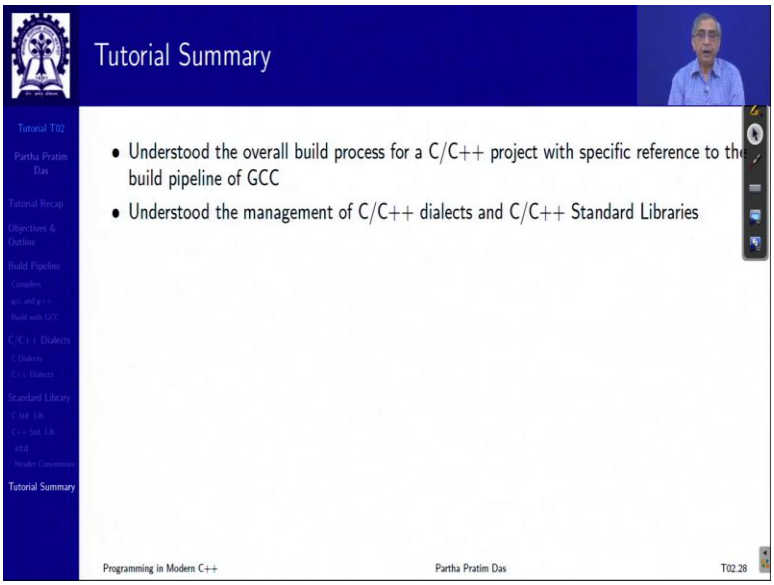
(Refer Slide Time: 32:30)


C Standard Library: math.h
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions
Tutorial Summary
/* math.h
* This file has no copyright assigned and is placed in the Public Domain.
* This file is a part of the mingw-runtime package.
* Mathematical functions.
*/
#ifndef _MATH_H_
#define _MATH_H_
#ifndef __STRICT_ANSI__ // conditional exclusions for ANSI
// ...
#define M_PI 3.14159265358979323846 // manifest constant for pi
// ...
struct _complex { // struct of _complex type
double x; /* Real part */
double y; /* Imaginary part */
};
_CRTIMP double __cdecl _cabs (struct _complex); // cabs(.) function header
// ...
#endif /* __STRICT_ANSI__ */
// ...
_CRTIMP double __cdecl sqrt (double); // sqrt(.) function header
// ...
#define isfinite(x) ((fpclassify(x) & FP_NAN) == 0) // macro isfinite(.) to check if a number is finite
// ...
#endif /* _MATH_H_ */
Source: C math.h library functions Accessed 13-Sep-21
Programming in Modern C++
Partha Pratim Das
T02.24


C++ Standard Library: Common Library Components
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions
Tutorial Summary
Component
Data Types, Manifest Constants, Macros, Functions, Classes, ...
iostream
Stream input and output for standard I/O
• cout, cin, endl, ..., etc.
string
Manipulation of string objects
• Relational operators, IO operators, Iterators, etc.
memory
High-level memory management
• Pointers: unique_ptr, shared_ptr, weak_ptr, auto_ptr, & allocator etc.
exception
Generic Error Handling • exception, bad_exception, unexpected_handler, terminate_handler, etc.
stdexcept
Standard Error Handling • logic_error, invalid_argument, domain_error, length_error, out_of_range, runtime_error, range_error, overflow_error, underflow_error, etc.
Adopted from C Standard Library
cmath
Common mathematical operations and transformations
• cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt, etc.
cstdlib
Memory alloc., process control, conversions, pseudo-rand nos., searching, sorting
• malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.
Programming in Modern C++
Partha Pratim Das
T02.25


namespace std for C++ Standard Library
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions
Tutorial Summary
C Standard Library
All names are global
stdout, stdin, printf, scanf
C++ Standard Library
All names are within std namespace
std::cout, std::cin
Use using namespace std;
to get rid of writing std:: for every standard library name
W/o using
#include <iostream>
int main() {
std::cout << "Hello World in C++"
<< std::endl;
return 0;
}
W/ using
#include <iostream>
using namespace std;
int main() {
cout << "Hello World in C++"
<< endl;
return 0;
}
Programming in Modern C++
Partha Pratim Das
T02:26


Standard Library: C/C++ Header Conventions
Tutorial T02
Partha Pratim Das
Tutorial Recap
Objectives & Outline
Build Pipeline
Compilers
gcc and g++
Build with GCC
C/C++ Dialects
C Dialects
C++ Dialects
Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions
Tutorial Summary
C Header
C++ Header
C Program
Use .h. Example: #include <stdio.h>
Names in global namespace
Not applicable
C++ Program
Prefix c, no .h. Example: #include <cstdio>
Names in std namespace
No .h. Example: #include <iostream>
A C std. library header is used in C++ with prefix 'c' and without the .h. These are in std namespace
#include <cmath> // In C it is <math.h>
...
std::sqrt(5.0); // Use with std::
It is possible that a C++ program include a C header as in C. Like:
#include <math.h> // Not in std namespace
...
sqrt(5.0); // Use without std::
This, however, is not preferred
Using .h with C++ header files, like iostream.h, is disastrous. These are deprecated. It is dangerous, yet true, that some compilers do not error out on such use. Exercise caution.
Programming in Modern C++
Partha Pratim Das
T02:27



So some snippets of what is inside the header, similarly C++ libraries there are many again and I will slowly introduce them as we go through the modules but iostream, string, memory, cmath, from C, cstdlib or cstring vector these are some of the important ones. I will not discuss this in the tutorial because we have already discussed this in the module in terms of what is the difference between the std namespace and other and what are the header conventions these we have already discussed in the module.

So just remember to follow them. So I hope you have got a good sense of how to do this build process I will also make a small video later on to actually do this steps while I use my gcc so that even after this discussion if you have difficulties you can just follow those steps in the video. I will also provide that at a later point of time. Thank you very much for your attention and see you in the next tutorial on the same project building process. Thank you.