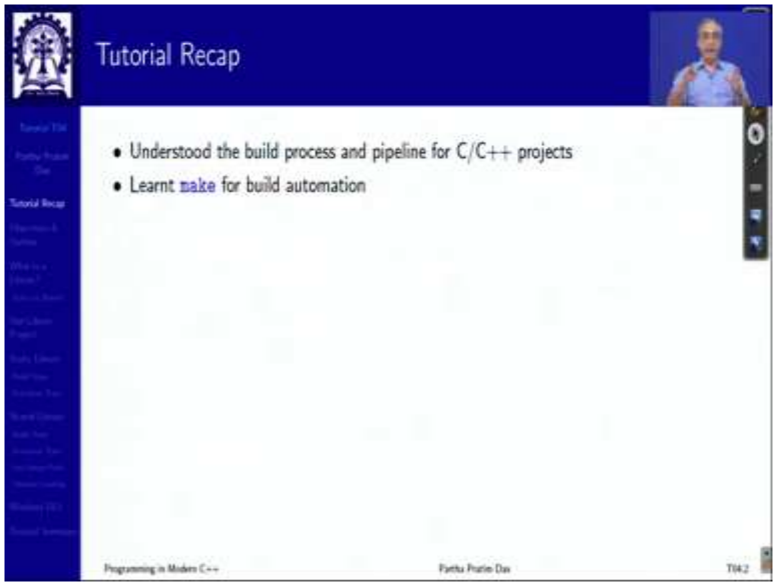
Programming in Modern C++

Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technologies, Kharagpur
Tutorial – 04
How to build a C/C++ Program?
Part 04: Static and Dynamic Library

Welcome to programming in modern C++. We are talking about a tutorial series we are on the fourth of that series, where we are primarily trying to learn about how to build C, C++ programs.

(Refer Slide Time: 00:45)



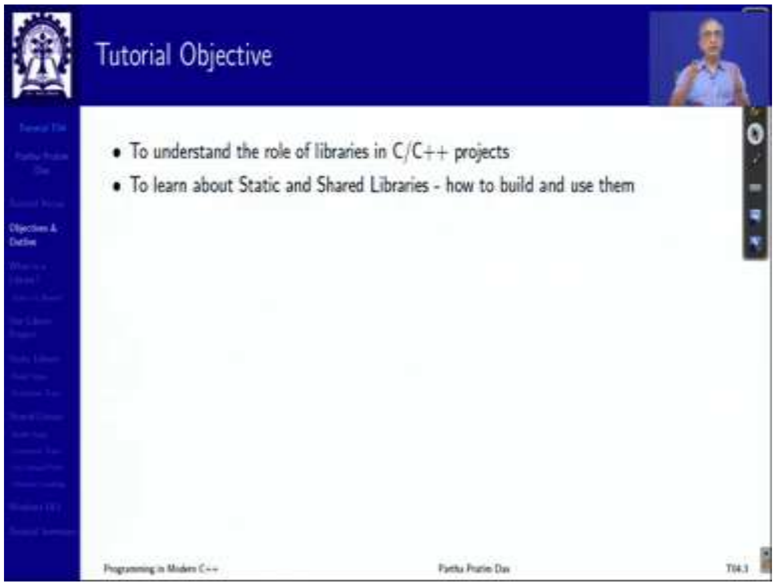
Tutorial Recap

- Understood the build process and pipeline for C/C++ projects
- Learnt make for build automation

Programming in Modern C++ Partha Pratim Das T04.2

Earlier, we have understood the overall build process, particularly the GCC pipeline of build for C, C++ project. And in the last tutorial, we have learnt about the make utility which can really make it efficient.

(Refer Slide Time: 01:00)



Tutorial Objective

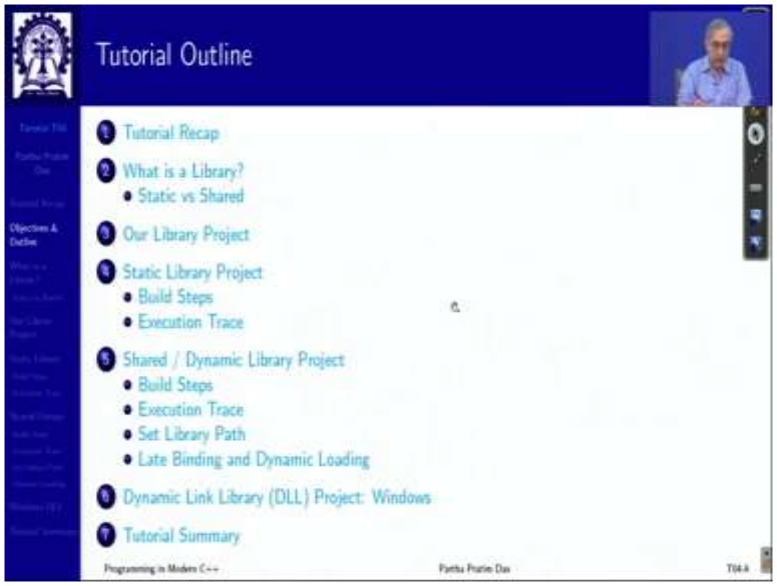
- To understand the role of libraries in C/C++ projects
- To learn about Static and Shared Libraries - how to build and use them

Programming in Modern C++ Partha Pratim Das T04.3

Now, in this tutorial, we will understand the role of libraries in C, C++. While talking about both of these languages, we are repeatedly saying we have standard library we can use printf from stdio.h or we can use vector from the vector component in C++ standard library. So, libraries are mechanisms for code reuse and to have very reliable, well tested code.

So, when they are provided with the language by the compiler vendor, those are specified in the standard library. But, each one of us while building big projects, would like to build our own libraries in addition to the standard library and use them. So, in this tutorial, we intend to understand what libraries are, what does it mean, when we say we have a static library or we have a shared library and so on, how to build these libraries and how to use this libraries.

(Refer Slide Time: 02:14)



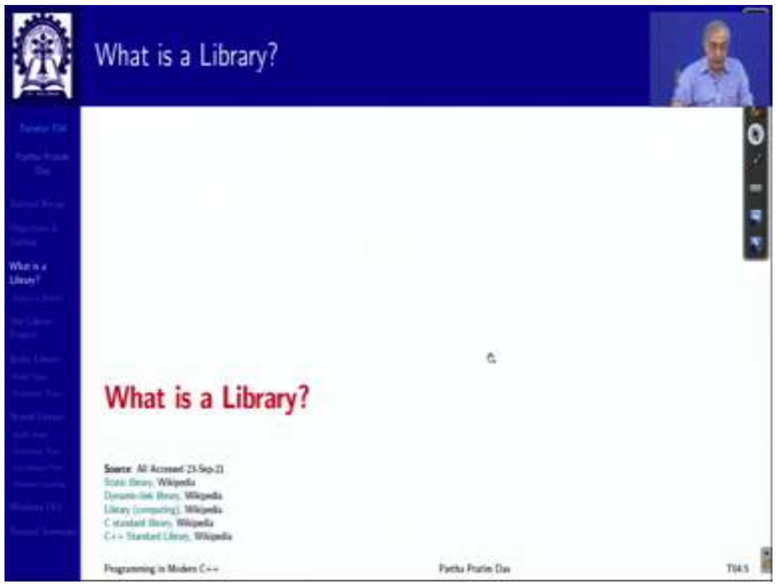
Tutorial Outline

1. Tutorial Recap
2. What is a Library?
   - Static vs Shared
3. Our Library Project
4. Static Library Project
   - Build Steps
   - Execution Trace
5. Shared / Dynamic Library Project
   - Build Steps
   - Execution Trace
   - Set Library Path
   - Late Binding and Dynamic Loading
6. Dynamic Link Library (DLL) Project: Windows
7. Tutorial Summary

Programming in Modern C++ Partha Pratim Das T04.4

So, this is the last of the build sequence of tutorial series we are going to discuss on.

(Refer Slide Time: 02:20)

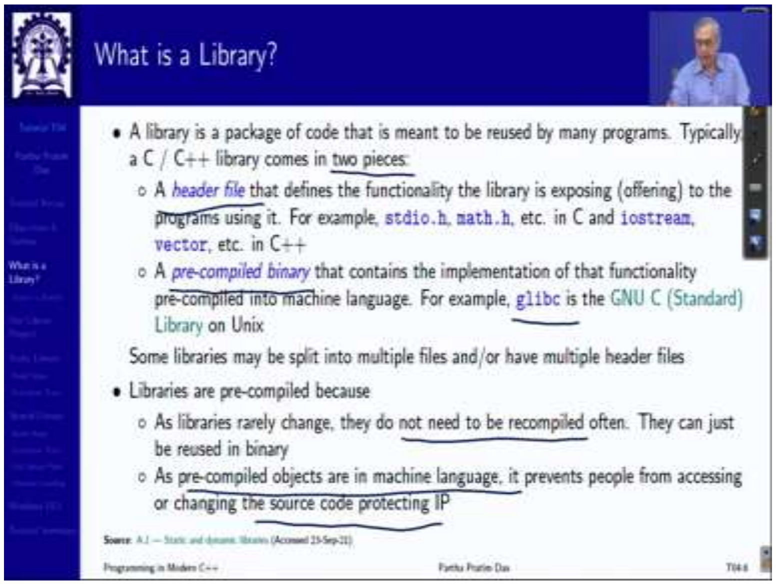


What is a Library?

What is a Library?

Source: All Accessed 23-Sep-21
Static library, Wikipedia
Dynamic-link library, Wikipedia
Library (computing), Wikipedia
C standard library, Wikipedia
C++ Standard Library, Wikipedia

Programming in Modern C++ Partha Pratim Das T04.5



What is a Library?

- A library is a package of code that is meant to be reused by many programs. Typically, a C / C++ library comes in two pieces:
  - A *header file* that defines the functionality the library is exposing (offering) to the programs using it. For example, `stdio.h`, `math.h`, etc. in C and `iostream`, `vector`, etc. in C++
  - A *pre-compiled binary* that contains the implementation of that functionality pre-compiled into machine language. For example, `glibc` is the GNU C (Standard) Library on Unix

  Some libraries may be split into multiple files and/or have multiple header files
- Libraries are pre-compiled because
  - As libraries rarely change, they do not need to be recompiled often. They can just be reused in binary
  - As pre-compiled objects are in machine language, it prevents people from accessing or changing the source code protecting IP

Source: A.1 — Static and dynamic libraries (Accessed 23-Sep-21)

Programming in Modern C++ Partha Pratim Das T04.6

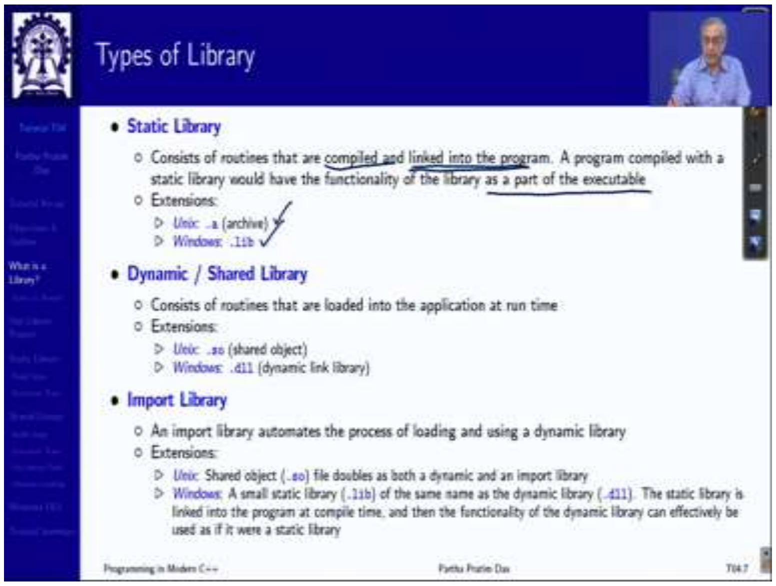
So, what is a library? A library is a package of code that is meant for use by many programs. Naturally, a header is meant for reused by multiple translation units within the same projects within the same project of the same program, but libraries are a package of code which I want to use across multiple projects. And typically a C, C++ library will come in two pieces. One is the text part of the library, which is the header, which gives you the prototypes of functions for C or, classes and operator overloads for C++ and the other component is a precompiled binary.

Now a library is something which has been written with a lot of care well tested for a given functionality. So, you do not expect that it will change frequently. So, you will not want for every project that the library be built again. So, you pre compile it and keep it in the binary form, say something equivalent of .o or some equivalent of that, for example, if I worked with GNU C, I have a library called glibc which is a binary precompiled binary of library of C, g stands for GNU, lib stands for library, C stands for the C language, it is available on Unix.

Now some libraries may have multiple source files, multiple header files may have multiple binaries also so we will have all of them. So, they do not change so I, we do not need to recompile and since they are already compiled objects in the machine language, they cannot be read by human beings or even if you use a program, you will not be able to figure much out of that library.

So, unless the source changes, you do not need to recreate that and the source being opaque, it gives a good protection of your IP if your library has an algorithm which you do not want to tell everyone, then it is all hidden in the binary code. Of course, so we will see subsequently that there has been certain philosophical drifts in this open era and besides these binaries, people are also talking about having libraries, which are open source that is source file is available.

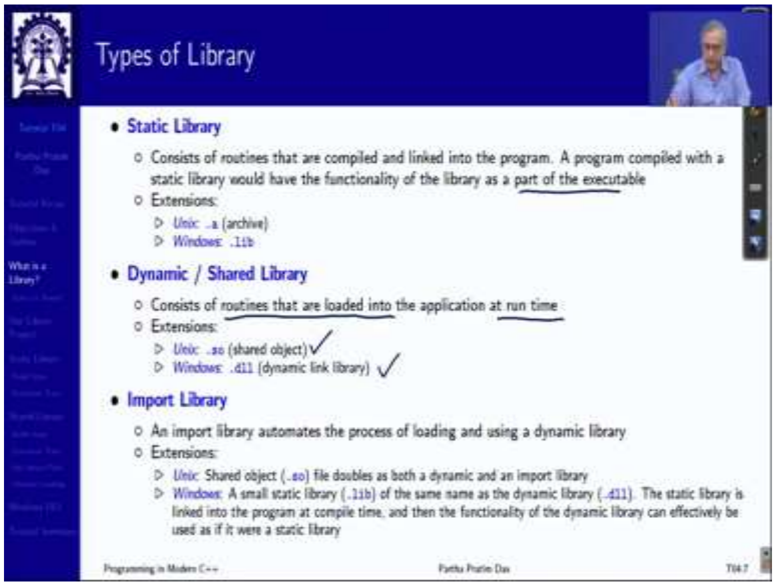
(Refer Slide Time: 05:07)



Types of Library

- **Static Library**
  - Consists of routines that are compiled and linked into the program. A program compiled with a static library would have the functionality of the library as a part of the executable
  - Extensions:
    - Unix: `.a` (archive)
    - Windows: `.lib`
- **Dynamic / Shared Library**
  - Consists of routines that are loaded into the application at run time
  - Extensions:
    - Unix: `.so` (shared object)
    - Windows: `.dll` (dynamic link library)
- **Import Library**
  - An import library automates the process of loading and using a dynamic library
  - Extensions:
    - Unix: Shared object (`.so`) file doubles as both a dynamic and an import library
    - Windows: A small static library (`.lib`) of the same name as the dynamic library (`.dll`). The static library is linked into the program at compile time, and then the functionality of the dynamic library can effectively be used as if it were a static library

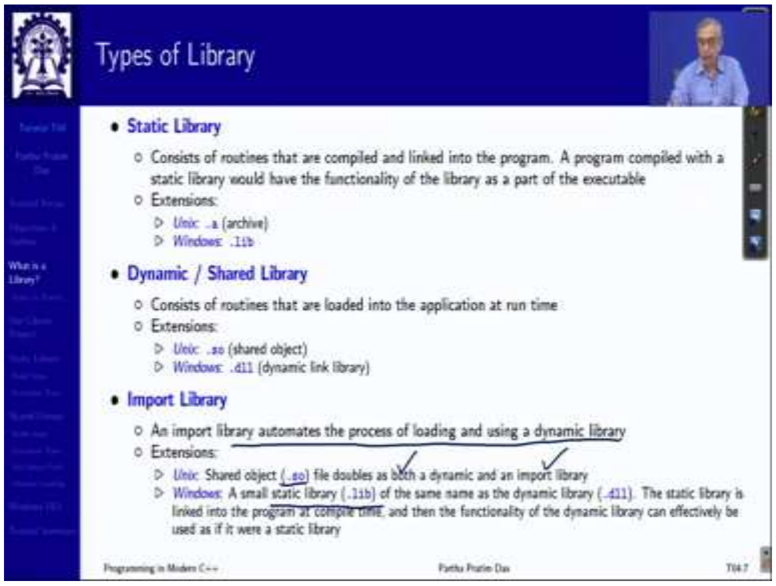
Programming in Modern C++ Partha Pratim Das T04.7



Types of Library

- **Static Library**
  - Consists of routines that are compiled and linked into the program. A program compiled with a static library would have the functionality of the library as a part of the executable
  - Extensions:
    - Unix: `.a` (archive)
    - Windows: `.lib`
- **Dynamic / Shared Library**
  - Consists of routines that are loaded into the application at run time
  - Extensions:
    - Unix: `.so` (shared object)
    - Windows: `.dll` (dynamic link library)
- **Import Library**
  - An import library automates the process of loading and using a dynamic library
  - Extensions:
    - Unix: Shared object (`.so`) file doubles as both a dynamic and an import library
    - Windows: A small static library (`.lib`) of the same name as the dynamic library (`.dll`). The static library is linked into the program at compile time, and then the functionality of the dynamic library can effectively be used as if it were a static library

Programming in Modern C++ Partha Pratim Das T04.7



Types of Library

- **Static Library**
  - Consists of routines that are compiled and linked into the program. A program compiled with a static library would have the functionality of the library as a part of the executable
  - Extensions:
    - Unix: .a (archive)
    - Windows: .lib
- **Dynamic / Shared Library**
  - Consists of routines that are loaded into the application at run time
  - Extensions:
    - Unix: .so (shared object)
    - Windows: .dll (dynamic link library)
- **Import Library**
  - An import library automates the process of loading and using a dynamic library
  - Extensions:
    - Unix: Shared object (.so) file doubles as both a dynamic and an import library
    - Windows: A small static library (.lib) of the same name as the dynamic library (.dll). The static library is linked into the program at compile time, and then the functionality of the dynamic library can effectively be used as if it were a static library

Programming in Modern C++ Partha Pratim Das T04.7

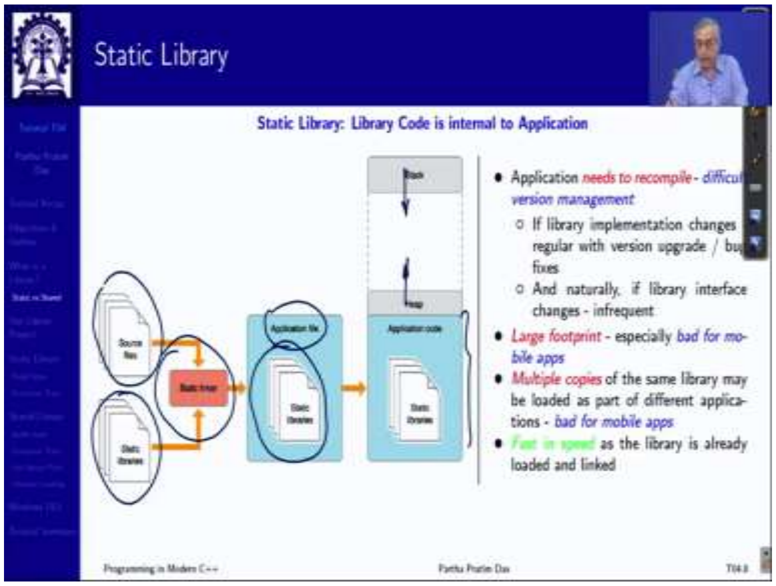
So, their libraries could be of multiple types, one is the static library, which is what most of the time you have been seeing. So, they are compiled and linked into the program, this is a, so a static library becomes a part of your binary code. So, your functionality of the library becomes a part of the executable that you create. Such libraries in Unix are typically has file extension .a, in Windows, they have .lib that says these are static libraries.

The other is known as dynamic library or shared library, these two words have two different meanings, but they describe the two aspects of the same type of library. Here, what you have is these are routines, they are not part of your executable, they are not inside your executable, but they are outside when you need, they will get loaded in the memory, we will show you how that structure works.

So, they are they get available to the application at the runtime. Whereas here, they are available to the application in the compile time, the static I mean rather the linking time. So, it always is there. A shared object is shared library in Unix is called shared object .so. And in Windows, it is called dll, .dll or dynamic link library. There are certain variants of this which are known as import library, where which can automate the process of loading and using a dynamic library.

So, it is kind of a little variation of the import library. So, .so in Unix doubles as both dynamic as well as import library, whereas in Windows, a small static library will be created, we will explain why we need this by the same name as the dynamic library. So, dll needs to be there but there may be additional small static library, which tells you how to what all you need to find in that library.

(Refer Slide Time: 07:38)

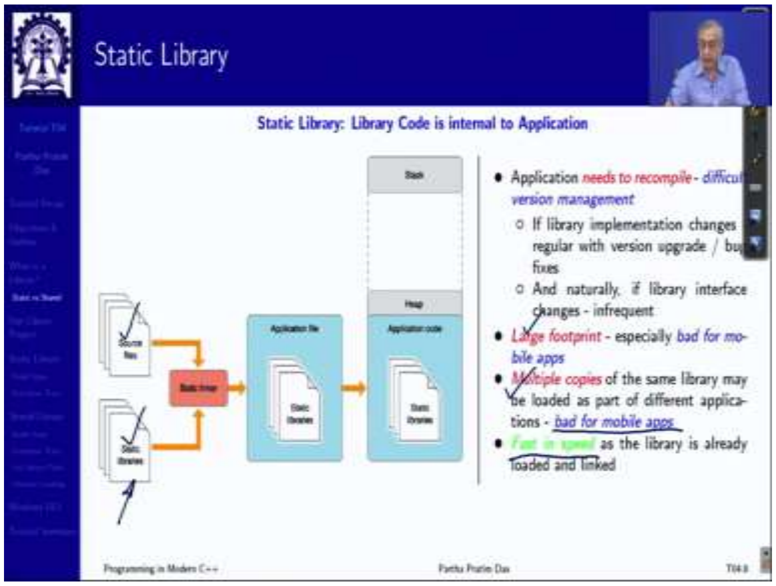


Static Library

Static Library: Library Code is internal to Application

- Application needs to recompile - difficul version management
  - If library implementation changes regular with version upgrade / bu fixes
  - And naturally, if library interface changes - infrequent
- Large footprint - especially bad for mobile apps
- Multiple copies of the same library may be loaded as part of different applications - bad for mobile apps
- Fast in speed as the library is already loaded and linked

Programming in Modern C++ Partha Pratim Das T04.8



Static Library

Static Library: Library Code is internal to Application

- Application needs to recompile - difficul version management
  - If library implementation changes regular with version upgrade / bu fixes
  - And naturally, if library interface changes - infrequent
- Large footprint - especially bad for mobile apps
- Multiple copies of the same library may be loaded as part of different applications - bad for mobile apps
- Fast in speed as the library is already loaded and linked

Programming in Modern C++ Partha Pratim Das T04.8

So, this is a schematic diagram. So, I have the source files, and I have the static libraries. So, what happened when I run the linker, I run the static linker or I links statically. Let us say what it does, then it takes the application file and puts the static library as a part of it. It is inside that. So, when it executes, I have the stack for my function calls the heap for my dynamic allocation and in the text segment of the memory, I have the entire application code loaded with the static library. They are all together.

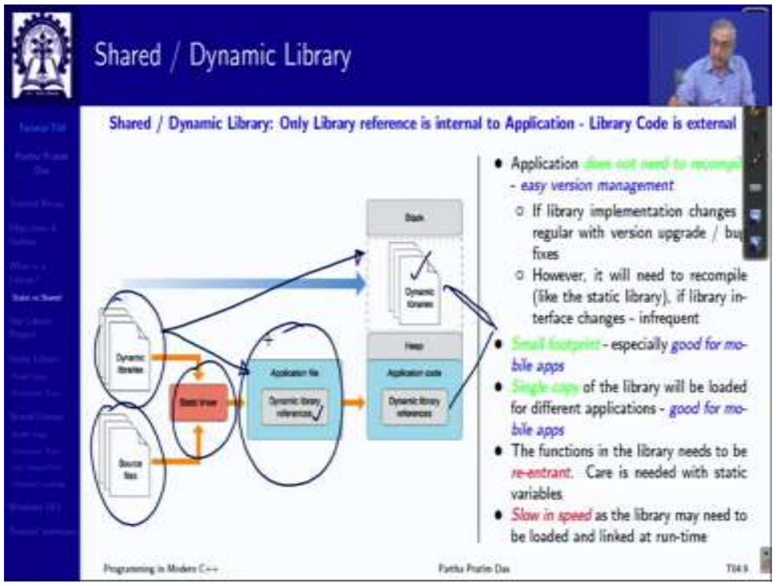
Now this has a lot of advantages and disadvantages. So, it applications will need a recompile this is my application source and this is my static library. Obviously, if I change my application source, I will need a recompile, but here I will need a recompilation, at least a linking, even if just my static library changes, I have not done that anything in my source code.

It has a large footprint, because you can think of stdio.h is a huge collection of functions. So even if the linker is smart, that it only picks up that part of those functions, which I have actually used my code it is a huge code and with every application, that code will get added. So, at a time in the memory, then maybe several processes running several programs. Understandably, each one of them will have input output. If we assume that all our C programs or C++ programs, they will have input output, let us say they are C programs.

So, you can think each one of that has a copy of the printf binary body. And all of them are so in the memory at the runtime there are multiple copies of these static library loaded. So, it is a waste of space, the footprint is large. So, particularly in small memory devices like mobile phones and so on this become difficult. But many times they are fast in speed at the execution, because they might take more time to load because they are bigger.

But once loaded, then there they may be fast because they are already in memory, which the dynamic or shared object library may not be.

(Refer Slide Time: 10:38)

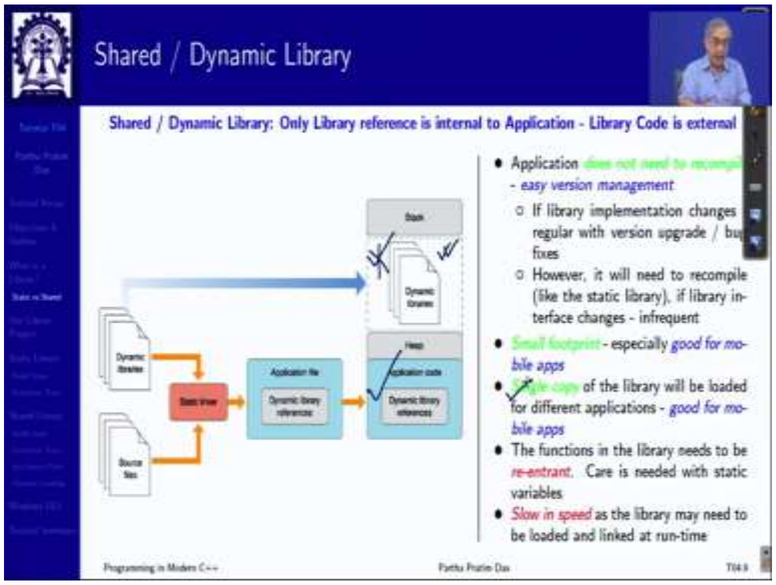


Shared / Dynamic Library

Shared / Dynamic Library: Only Library reference is internal to Application - Library Code is external

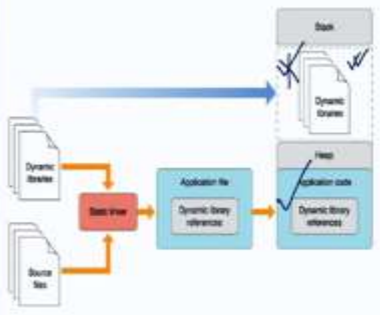
- Application does not need to recompi
  - *easy version management*
  - If library implementation changes regular with version upgrade / bu fixes
  - However, it will need to recompile (like the static library), if library interface changes - infrequent
- Small footprint - especially *good for mobile apps*
- Single copy of the library will be loaded for different applications - *good for mobile apps*
- The functions in the library needs to be *re-entrant*. Care is needed with static variables
- *Slow in speed* as the library may need to be loaded and linked at run-time

Programming in Modern C++ Partha Pratim Das T04.8

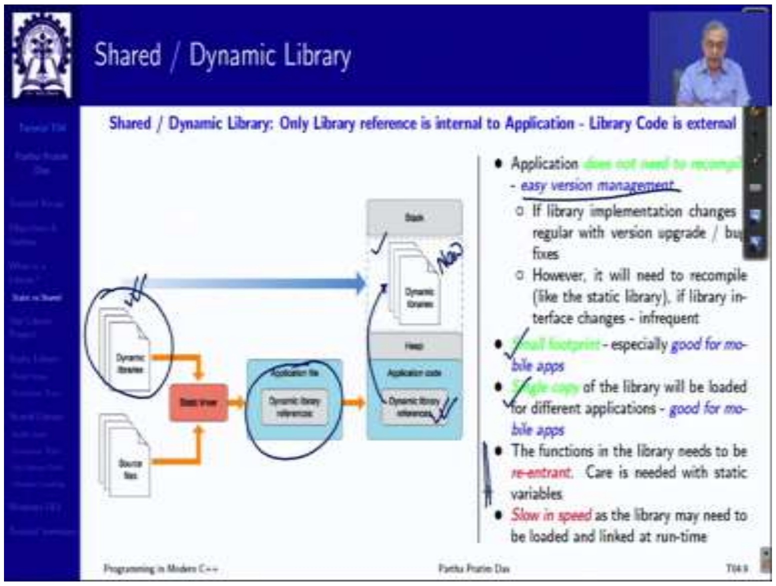


Shared / Dynamic Library

Shared / Dynamic Library: Only Library reference is internal to Application - Library Code is external



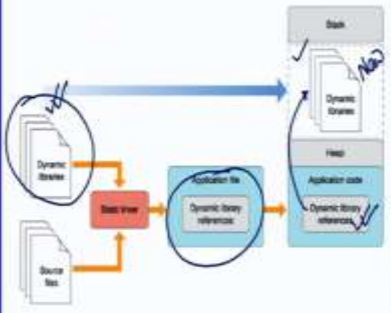
- Application does not need to recompil
  - *easy version management*
    - If library implementation changes regular with version upgrade / bu fixes
    - However, it will need to recompile (like the static library), if library interface changes - infrequent
- Small footprint - especially *good for mobile apps*
- Single copy of the library will be loaded for different applications - *good for mobile apps*
- The functions in the library needs to be *re-entrant*. Care is needed with static variables
- *Slow in speed* as the library may need to be loaded and linked at run-time

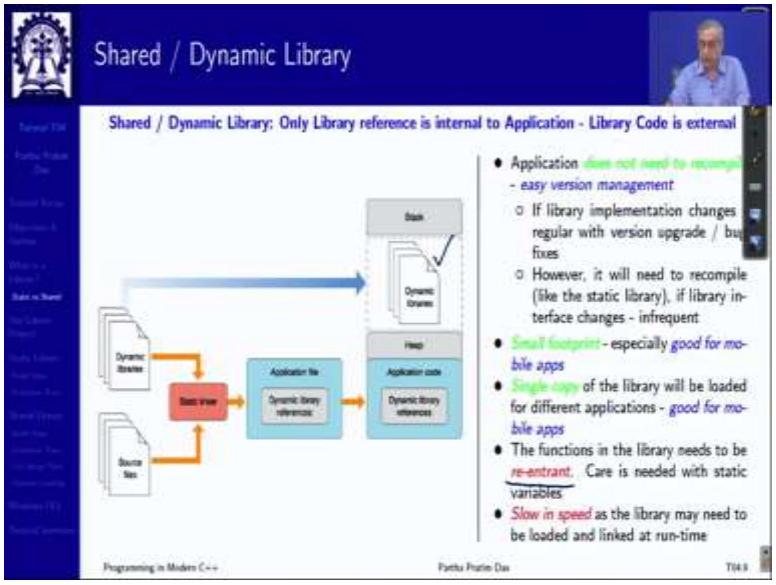


Shared / Dynamic Library

Shared / Dynamic Library: Only Library reference is internal to Application - Library Code is external

- Application does not need to recompil
  - *easy version management*
    - If library implementation changes regular with version upgrade / bu fixes
    - However, it will need to recompile (like the static library), if library interface changes - infrequent
- Small footprint - especially *good for mobile apps*
- Single copy of the library will be loaded for different applications - *good for mobile apps*
- The functions in the library needs to be *re-entrant*. Care is needed with static variables
- *Slow in speed* as the library may need to be loaded and linked at run-time





Shared / Dynamic Library

Shared / Dynamic Library: Only Library reference is internal to Application - Library Code is external

- Application does not need to recompi
  - *easy version management*
  - If library implementation changes regular with version upgrade / bu fixes
  - However, it will need to recompile (like the static library), if library interface changes - infrequent
- Small footprint - especially *good for mobile apps*
- Single copy of the library will be loaded for different applications - *good for mobile apps*
- The functions in the library needs to be re-entrant. Care is needed with static variables
- *Slow in speed* as the library may need to be loaded and linked at run-time

Programming in Modern C++ Partha Pratim Das T04.9

So, this is the view of a shared or dynamic library, where I have the source files as before of the dynamic libraries, I have my linker which is happening at the static time. But it is doing kind of a dynamic linking what it says that, from the dynamic library, it does not include everything here is this includes enough information so that the application program when it needs to find a particular function, we will get that information here.

But the function actually is not here it is in the dynamic library body. So, dynamic library is in terms of two components; one which has the references of what to find the function and the actual function. So, what becomes a part of the application file is not the entire code of printf. But some, abstracted information which says that, this is where printf can be found, when I actually want to run it. So, the you can see that these are independent binary components.

So, when the application is loaded, this may not have been loaded, this can be loaded only when it is required. Now, if it is loaded and some other function, some other program not function, some other program in a different process wants to also do printf it will not need to load it again. Because the printf is the same that does not change between projects. So, it can actually load the actual binary of the dynamic library in only one copy.

And use it from multiple projects, which is the static cannot do in the static library case, by every executable has a copy of the total binary code of the library here, there is only one copy. So, the single copy is a big big advantage. Naturally, the added advantages are that if something has changed in the dynamic library, over a period of time something has changed here. So, you get a different code.

Unlike in static library, you do not need to rebuild the system unless of course you make so fundamental changes that references itself change like whatever was printf you decide to from tomorrow, you decide to call it PPD printf then obviously, we need to change. But if you do not do that, then you are finding out the actual binary code at runtime. So, all that you need to do is if you change the dynamic library, shared library, all that you need to do you replace the original in the present one.

In fact, you can have multiple versions also. So, this becomes this has become very, very important. This has become very, very useful in terms of version management of applications. I mean, we will often see that in our mobile phone or our laptop say that well I need to download this fixes, is it downloading everything, reinstalling everything? No, what it does it is bringing in few components where changes have been made.

Now how does it work with the rest of the system? The rest of the system already knows the references. If it was static, then you would have required to change all of your operating system code if the start, it is just different pieces of dynamic libraries. As long as the reference to the dynamic library does not change, all that you need to do is just replace the available implementation with the new implementation.

What is important is you could keep multiple copies you say okay, if it is before, say seventeenth October, then you can use this stand copy of the dynamic library because you have an older version and if it is after then you use this. So, he can automatically manage all that. So, all those putting patches to your code, regular updates and all that would have been horrendously difficult if you were working only with static libraries.

So, this is very, very important for easy version management of course, it has a smaller footprint, because most of the binary code is outside it has a single copy which is of advantage in the overall memory management particularly good for small memory devices like mobile, but it may be, it depends it may be a little slow, when you want to load it for the first time, you happen to load it for the first time the load time will come in, which is not there for the static library, because it is pre-loaded.

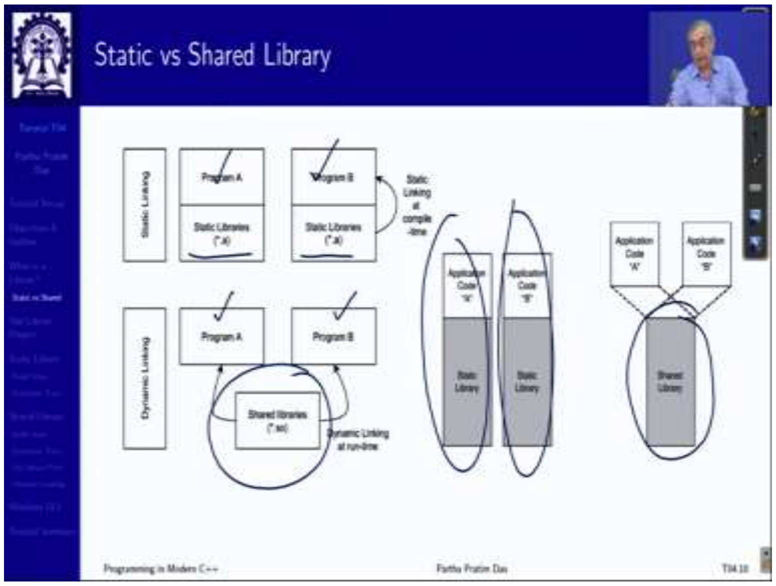
It may be a little slow, because here you are not going in directly to, invoke the library function, but you have to go through the this reference, you will first go to the reference find where that actual binary code is there. So, one step so, it may be a little slower, but in many cases, overall, it becomes much faster, because much less amount of code loading in the memory is at all needed. There is a little caveat in all this is in terms of dynamic library, your function codes, classes have to be reentrant.

Reentrant means, because, you can see that, if you had two projects, both having the same static library, they will have two sets of variable, allocations independent. So, anything which is static in the static library, any variable that is static will be having separate copies in terms of the two projects, but here you have only one copy of the library. So, if something is static, then there is only one copy.

So, if two projects are using it, one project changes and the other project will get affected. So, what you have to make sure is that you do not use that kind of static variables and that is put in a little more a little different way with a little bit of twist, when you say that the code that you write is reentrant which means that a function when it is already executing in one process can start executing in a different process, there is no there be no variable clash, you will have to ensure that then you have it reentrancy of the function.

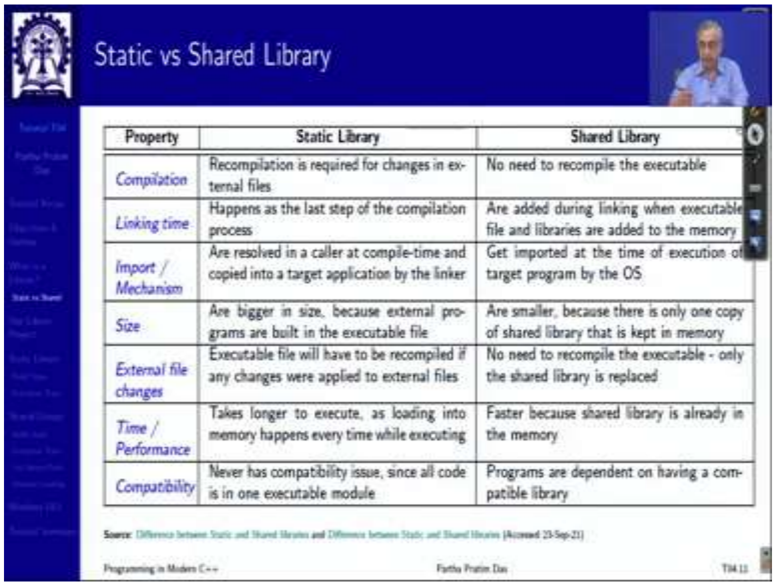
Not not very important for your overall library understanding but just wanted to mention this on the passing because that is a critical point.

(Refer Slide Time: 18:04)


Static vs Shared Library

So, that is schematically what finally so you have a program A program B you have static libraries, and each one of them will be statically linked into these kinds of codes. If you have dynamic library your program A program B we have a one common shared library, so which will be dynamically linked at the runtime like in here. So, different schematic way of looking at the whole thing.

(Refer Slide Time: 18:35)


Static vs Shared Library

Property	Static Library	Shared Library
Compilation	Recompilation is required for changes in external files	No need to recompile the executable
Linking time	Happens as the last step of the compilation process	Are added during linking when executable file and libraries are added to the memory
Import / Mechanism	Are resolved in a caller at compile-time and copied into a target application by the linker	Get imported at the time of execution of target program by the OS
Size	Are bigger in size, because external programs are built in the executable file	Are smaller, because there is only one copy of shared library that is kept in memory
External file changes	Executable file will have to be recompiled if any changes were applied to external files	No need to recompile the executable - only the shared library is replaced
Time / Performance	Takes longer to execute, as loading into memory happens every time while executing	Faster because shared library is already in the memory
Compatibility	Never has compatibility issue, since all code is in one executable module	Programs are dependent on having a compatible library

Source: Difference between Static and Shared libraries and Difference between Static and Shared libraries (Accessed 23-Sep-21)

So, all that we have discussed are summarized as a comparison between the static and shared library in terms of the compilation linking time import mechanism size, external file changes, time performance compatibility and so on. I do not want to run through each point again we have already we have already discussed this, it is this for your summarized reference.

(Refer Slide Time: 19:00)

Our Library Project

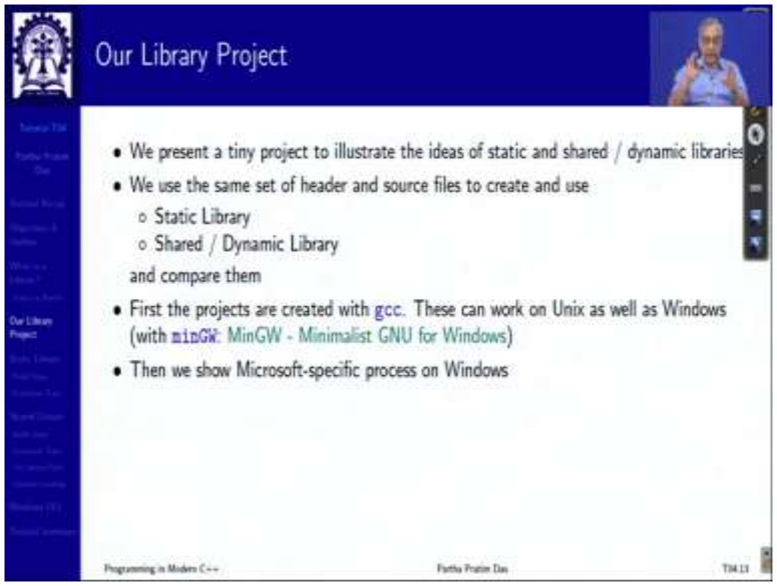
Our Library Project

Sources: All Accessed 26-Sep-21
Building And Using Static And Shared "C" Libraries
Shared libraries with GCC on Linux
MinGW Static and Dynamic Libraries
Static and Dynamic Libraries in C Language

Programming in Modern C++ Partha Pratim Das T04.12

Now, we come to the main so this is was I mean though we brought in as a part of the tutorial, but this was not that much of hands on this was to understand the concepts of libraries particularly the difference between static and dynamic library. So, now to really get the hold we will execute a library project.

(Refer Slide Time: 19:26)



Our Library Project

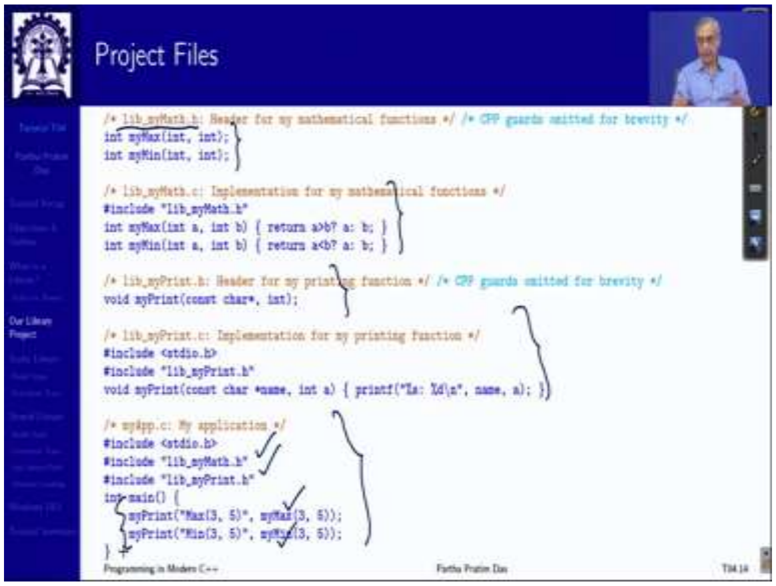
- We present a tiny project to illustrate the ideas of static and shared / dynamic libraries
- We use the same set of header and source files to create and use
  - Static Library
  - Shared / Dynamic Library

  and compare them
- First the projects are created with gcc. These can work on Unix as well as Windows (with minGW: MinGW - Minimalist GNU for Windows)
- Then we show Microsoft-specific process on Windows

Programming in Modern C++ Partha Pratim Das T04.13

So, we present a tiny project and we will build static as well as dynamic libraries for that project. With the difference that our with the condition that it will use the same set of header and source files. And we will see how things work we will later on show some Microsoft specific stuff as well.

(Refer Slide Time: 19:48)



Project Files

```
/* lib_myMath.h: Header for my mathematical functions */ /* CPP guards omitted for brevity */
int myMax(int, int);
int myMin(int, int);

/* lib_myMath.c: Implementation for my mathematical functions */
#include "lib_myMath.h"
int myMax(int a, int b) { return a>b? a: b; }
int myMin(int a, int b) { return a<b? a: b; }

/* lib_myPrint.h: Header for my printing function */ /* CPP guards omitted for brevity */
void myPrint(const char*, int);

/* lib_myPrint.c: Implementation for my printing function */
#include <stdio.h>
#include "lib_myPrint.h"
void myPrint(const char *name, int a) { printf("%s: %d\n", name, a); }

/* myApp.c: My application */
#include <stdio.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```

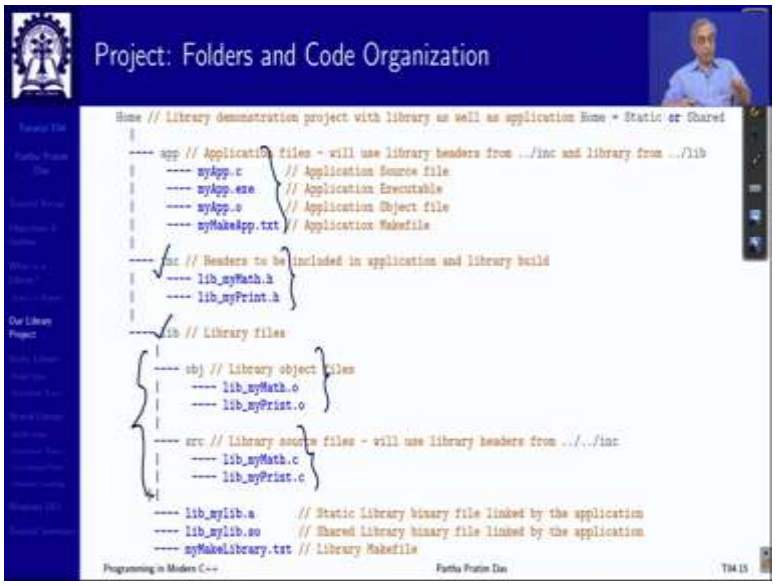
Programming in Modern C++ Partha Pratim Das T04.14

So, let us understand the project once and for all. We will not come back to this every type so it is a very pathological project. It has a library math .h header, which defines header of two math functions finding max finding min defined by me. This is a header file so it will have the CPP guards and all that just for compactness, I am not showing all that. Naturally, you have an implementation of this in terms of the library math .c source file.

I have a print header containing a printing function, prototype of a myPrint function. And I have implementation of that in the corresponding lib myPrint.c. So, two math functions one printing function, two headers corresponding .c files. And using that I write my application where I include both the headers, I use the myMax function from library math .c as implemented to compute the maximum, this is to compute the minimum and I use the myPrint function in the myPrint.c, lib_myPrint.c file to print the balance.

So, I have tried to keep the actual algorithmic functionality requirements to the simplest, so that we can focus on the real issues. I mean you can think about any big project but this is the basic, skeletal abstraction I mean skeletal architecture that we will always have.

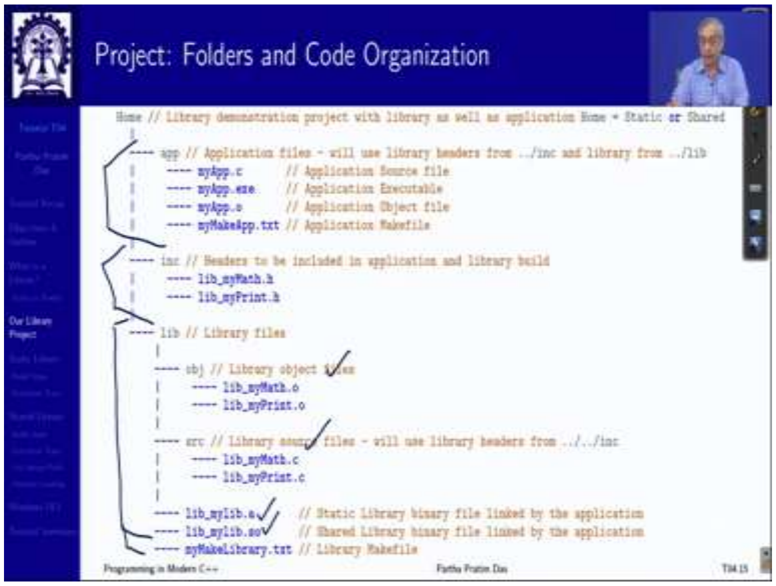
(Refer Slide Time: 21:49)



Project: Folders and Code Organization

```
Home // Library demonstration project with library as well as application Home = Static or Shared
  |
  ---- app // Application files - will use library headers from ../inc and library from ../lib
  |     ---- myApp.c        // Application Source file
  |     ---- myApp.exe      // Application Executable
  |     ---- myApp.o        // Application Object file
  |     ---- myMakeApp.txt  // Application Makefile
  |
  ---- inc // Headers to be included in application and library build
  |     ---- lib_myMath.h
  |     ---- lib_myPrint.h
  |
  ---- lib // Library files
        |
        ---- obj // Library object files
        |     ---- lib_myMath.o
        |     ---- lib_myPrint.o
        |
        ---- src // Library source files - will use library headers from ../../inc
        |     ---- lib_myMath.c
        |     ---- lib_myPrint.c
        |
        ---- lib_mylib.a       // Static Library binary file linked by the application
        ---- lib_mylib.so      // Shared Library binary file linked by the application
        ---- myMakeLibrary.txt // Library Makefile
```

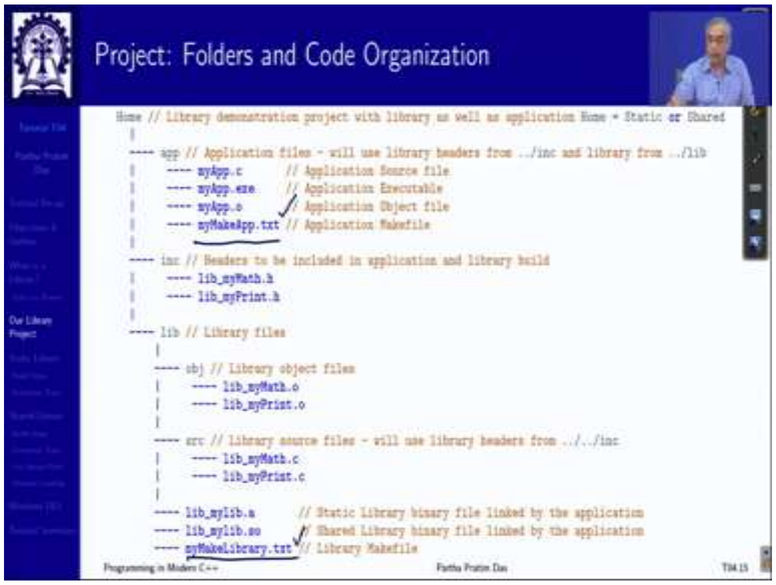
Programming in Modern C++ | Partha Pratim Das | T04.15



Project: Folders and Code Organization

```
Home // Library demonstration project with library as well as application Home = Static or Shared
  |
  ---- app // Application files - will use library headers from ../inc and library from ../lib
  |     ---- myApp.c        // Application Source file
  |     ---- myApp.exe      // Application Executable
  |     ---- myApp.o        // Application Object file
  |     ---- myMakeApp.txt  // Application Makefile
  |
  ---- inc // Headers to be included in application and library build
  |     ---- lib_myMath.h
  |     ---- lib_myPrint.h
  |
  ---- lib // Library files
        |
        ---- obj // Library object files
        |     ---- lib_myMath.o
        |     ---- lib_myPrint.o
        |
        ---- src // Library source files - will use library headers from ../../inc
        |     ---- lib_myMath.c
        |     ---- lib_myPrint.c
        |
        ---- lib_mylib.a       // Static Library binary file linked by the application
        ---- lib_mylib.so      // Shared Library binary file linked by the application
        ---- myMakeLibrary.txt // Library Makefile
```

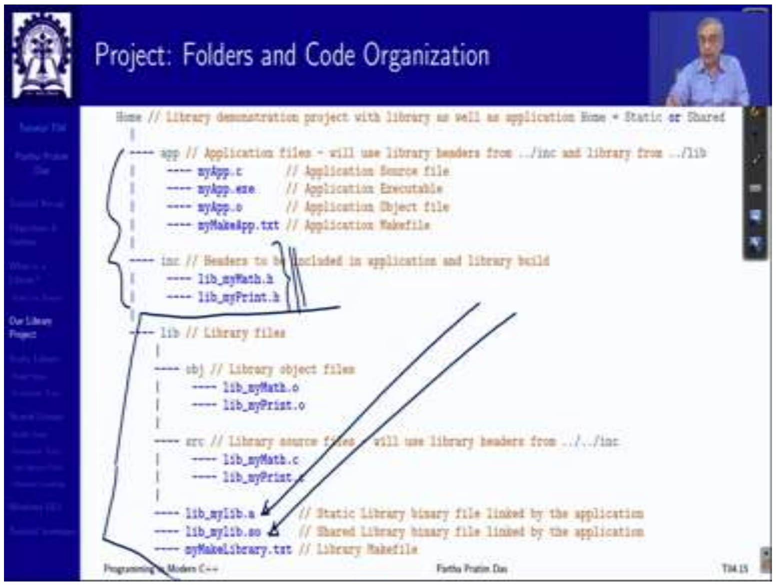
Programming in Modern C++ | Partha Pratim Das | T04.15



Project: Folders and Code Organization

```
Home // Library demonstration project with library as well as application Home = Static or Shared
  |
 ---- app // Application files - will use library headers from ../inc and library from ../lib
  |     ---- myApp.c       // Application Source file
  |     ---- myApp.exe     // Application Executable
  |     ---- myApp.o       // Application Object file
  |     ---- myMakeApp.txt // Application Makefile
  |
 ---- inc // Headers to be included in application and library build
  |     ---- lib_myMath.h
  |     ---- lib_myPrint.h
  |
 ---- lib // Library files
       |
      ---- obj // Library object files
       |     ---- lib_myMath.o
       |     ---- lib_myPrint.o
       |
      ---- src // Library source files - will use library headers from ../../inc
       |     ---- lib_myMath.c
       |     ---- lib_myPrint.c
       |
      ---- lib_mylib.a       // Static Library binary file linked by the application
      ---- lib_mylib.so      // Shared Library binary file linked by the application
      ---- myMakeLibrary.txt // Library Makefile
```

Programming in Modern C++ Partha Pratim Das T04.15



Project: Folders and Code Organization

```
Home // Library demonstration project with library as well as application Home = Static or Shared
  |
 ---- app // Application files - will use library headers from ../inc and library from ../lib
  |     ---- myApp.c       // Application Source file
  |     ---- myApp.exe     // Application Executable
  |     ---- myApp.o       // Application Object file
  |     ---- myMakeApp.txt // Application Makefile
  |
 ---- inc // Headers to be included in application and library build
  |     ---- lib_myMath.h
  |     ---- lib_myPrint.h
  |
 ---- lib // Library files
       |
      ---- obj // Library object files
       |     ---- lib_myMath.o
       |     ---- lib_myPrint.o
       |
      ---- src // Library source files - will use library headers from ../../inc
       |     ---- lib_myMath.c
       |     ---- lib_myPrint.c
       |
      ---- lib_mylib.a       // Static Library binary file linked by the application
      ---- lib_mylib.so      // Shared Library binary file linked by the application
      ---- myMakeLibrary.txt // Library Makefile
```

Programming in Modern C++ Partha Pratim Das T04.15

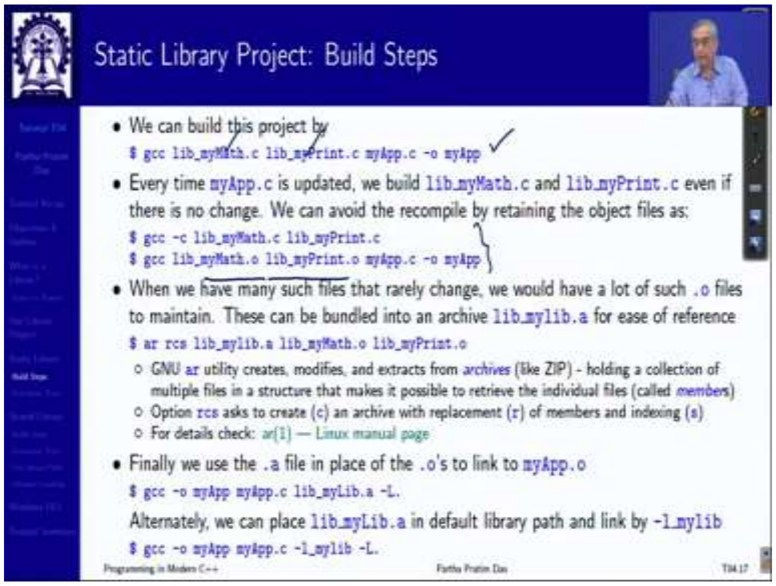
So, based on this, let us organize it, we have learned from the last tutorial, that code should better be organized well, so I have my include directory having my both header files, I have my object directory of both math.o and print.o object files I have the source directory for the header implementations. And based on that, I have the application directory which has my source my object file my your final executable and so on.

At the in the then I have the library directory which includes the object and the source. So, this is the library part. So, this is the library part of the whole story, this is the application part and this is a bridge the header which allows the information to be linked appropriately. So, in the library other than the object and the source files, I have a .a file, which I would create show how to create for the static library alternately, I could create a .so for my for making a dynamic library.

I have two make files one for building my library, which is here and one for building my application which you see here. So, this is there is nothing sacrosanct about this, it is not that it will have to be like this. But this is just one way of doing it. Normally when we deal with normally when we deal with this is the part that we do not get to see. Like this part of stdio.h or math.h or iostream or vector, you do not get to see it is all done somewhere somehow and all that I get to see is either this or this and we play around only in this part.

So, this is where equivalent of your io dot, I mean and stdio.h or iostream or vector. So, this is but now we are trying to not only see the use, but we want to see how to build this library. So, I have gotten into the inner details of the library structure and possibilities.

(Refer Slide Time: 24:27)



Static Library Project: Build Steps

- We can build this project by

```
$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp
```

✓

- Every time `myApp.c` is updated, we build `lib_myMath.c` and `lib_myPrint.c` even if there is no change. We can avoid the recompile by retaining the object files as:

```
$ gcc -c lib_myMath.c lib_myPrint.c
$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp
```

- When we have many such files that rarely change, we would have a lot of such `.o` files to maintain. These can be bundled into an archive `lib_mylib.a` for ease of reference

```
$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o
```

  - GNU `ar` utility creates, modifies, and extracts from *archives* (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called *members*)
  - Option `rcs` asks to create (`c`) an archive with replacement (`r`) of members and indexing (`s`)
  - For details check: ar(1) — Linux manual page

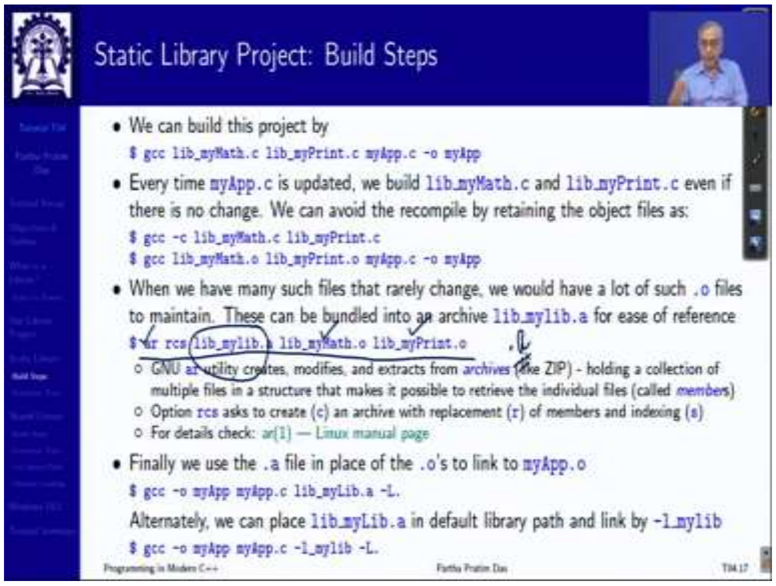
- Finally we use the `.a` file in place of the `.o`'s to link to `myApp.o`

```
$ gcc -o myApp myApp.c lib_myLib.a -L.
```

Alternately, we can place `lib_myLib.a` in default library path and link by `-l_mylib`

```
$ gcc -o myApp myApp.c -l_mylib -L.
```

Programming in Modern C++ Partha Pratim Das T4.17



Static Library Project: Build Steps

- We can build this project by

```
$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp
```

- Every time `myApp.c` is updated, we build `lib_myMath.c` and `lib_myPrint.c` even if there is no change. We can avoid the recompile by retaining the object files as:

```
$ gcc -c lib_myMath.c lib_myPrint.c
$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp
```

- When we have many such files that rarely change, we would have a lot of such `.o` files to maintain. These can be bundled into an archive `lib_mylib.a` for ease of reference

```
$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o
```

  - GNU `ar` utility creates, modifies, and extracts from archives (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called members)
  - Option `rcs` asks to create (`c`) an archive with replacement (`r`) of members and indexing (`s`)
  - For details check: ar(1) — Linux manual page

- Finally we use the `.a` file in place of the `.o`'s to link to `myApp.o`

```
$ gcc -o myApp myApp.c lib_myLib.a -L.
```

Alternately, we can place `lib_myLib.a` in default library path and link by `-l_mylib`

```
$ gcc -o myApp myApp.c -l_mylib -L.
```

Static Library Project: Build Steps

- We can build this project by

```
$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp
```

- Every time `myApp.c` is updated, we build `lib_myMath.c` and `lib_myPrint.c` even if there is no change. We can avoid the recompile by retaining the object files as:

```
$ gcc -c lib_myMath.c lib_myPrint.c
$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp
```

- When we have many such files that rarely change, we would have a lot of such `.o` files to maintain. These can be bundled into an archive `lib_mylib.a` for ease of reference

```
$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o
```

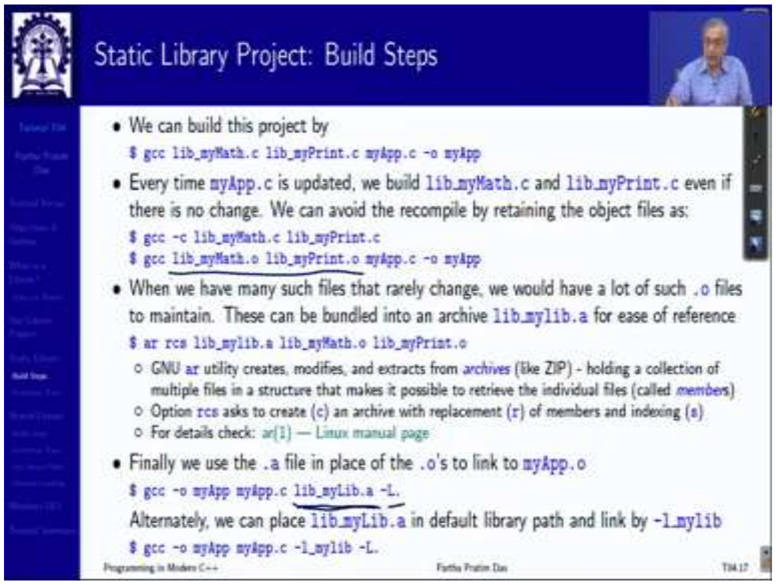
  - GNU `ar` utility creates, modifies, and extracts from archives (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called members)
  - Option `rcs` asks to create (`c`) an archive with replacement (`r`) of members and indexing (`s`)
  - For details check: ar(1) — Linux manual page

- Finally we use the `.a` file in place of the `.o`'s to link to `myApp.o`

```
$ gcc -o myApp myApp.c lib_myLib.a -L.
```

Alternately, we can place `lib_myLib.a` in default library path and link by `-l_mylib`

```
$ gcc -o myApp myApp.c -l_mylib -L.
```



Static Library Project: Build Steps

- We can build this project by

  `$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp`
- Every time `myApp.c` is updated, we build `lib_myMath.c` and `lib_myPrint.c` even if there is no change. We can avoid the recompile by retaining the object files as:

  `$ gcc -c lib_myMath.c lib_myPrint.c`

  `$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp`
- When we have many such files that rarely change, we would have a lot of such `.o` files to maintain. These can be bundled into an archive `lib_mylib.a` for ease of reference

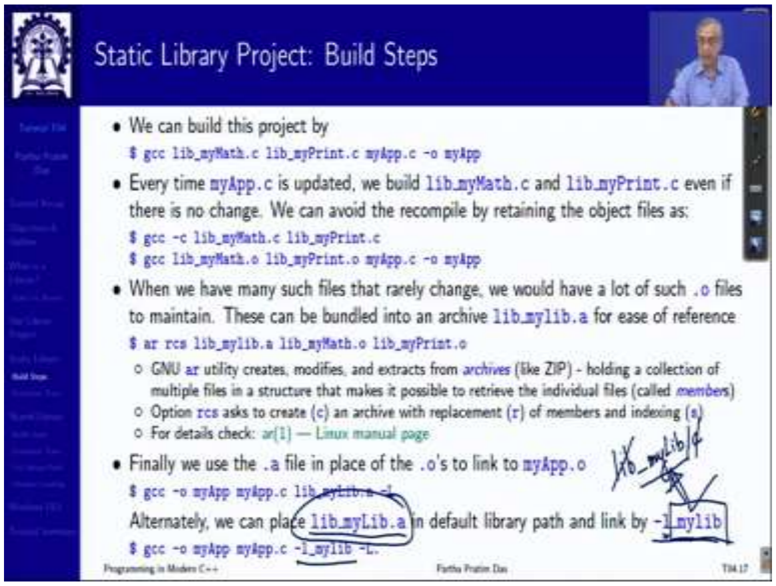
  `$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o`
  - GNU `ar` utility creates, modifies, and extracts from archives (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called members)
  - Option `rcs` asks to create (`c`) an archive with replacement (`r`) of members and indexing (`s`)
  - For details check: ar(1) — Linux manual page
- Finally we use the `.a` file in place of the `.o`'s to link to `myApp.o`

  `$ gcc -o myApp myApp.c lib_myLib.a -L.`

  Alternately, we can place `lib_myLib.a` in default library path and link by `-l_mylib`

  `$ gcc -o myApp myApp.c -l_mylib -L.`

Programming in Modern C++ Partha Pratim Das T04.17



Static Library Project: Build Steps

- We can build this project by

  `$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp`
- Every time `myApp.c` is updated, we build `lib_myMath.c` and `lib_myPrint.c` even if there is no change. We can avoid the recompile by retaining the object files as:

  `$ gcc -c lib_myMath.c lib_myPrint.c`

  `$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp`
- When we have many such files that rarely change, we would have a lot of such `.o` files to maintain. These can be bundled into an archive `lib_mylib.a` for ease of reference

  `$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o`
  - GNU `ar` utility creates, modifies, and extracts from archives (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called members)
  - Option `rcs` asks to create (`c`) an archive with replacement (`r`) of members and indexing (`s`)
  - For details check: ar(1) — Linux manual page
- Finally we use the `.a` file in place of the `.o`'s to link to `myApp.o`

  `$ gcc -o myApp myApp.c lib~~_myLib.a -L.~~`

  Alternately, we can place `lib_myLib.a` in default library path and link by `-l_mylib`

  `$ gcc -o myApp myApp.c -l_mylib -L.`

~~lib~~_mylib.a

Programming in Modern C++ Partha Pratim Das T04.17

So, let me first build the static library now static library project. So, what I do is I can build this project by putting everything together, if I did not do a library, my map.c print.c, app.c and into myApp. Obviously, I do not want to do that because neither of this should be changing usually because these are core provided well tested. So, every time I change my code in my application, I will still need to do a rebuild of these math.c and myPrint.c.

So, I would not like to do that, because they rarely change. So, what I will do is I will keep their .o already built. But then, the problem is there are multiple headers, there will be a bunch of .os coming in, how to manage all of that. So, how do manage when you have too many files, and you want to possibly attach it to a mail or, put it to a folder, what do you do? You bundle them together, you say we zip them together, so we exactly do the same thing.

So, what we do is we build the .o files, as we do, as we have seen earlier. And then we do something like this, taking this .os. The command to do this is ar stands for archive, which is generic class of what we say as zip files. And we give the target name as something here I have given lib_mylib.a, .a is means that it is an archive file. Now in the process of doing that, since I do not, I am not going to usually expand this, extract the individual components of .o, I am going to do things inside that finding out symbols and all that.

So, this ar is something a little different from your zip or seven z or WinRAR, or stuff like that. This is a archiving utility, which takes say, multiple options rcs is a very common, you can find out other options also, c tells that you create, r tells that you replace that if it is something is already there, we replace that and s tells that, not only you zip, but you have an index on the top so that I do not need to if I want an information, I would not need to unzip the whole thing.

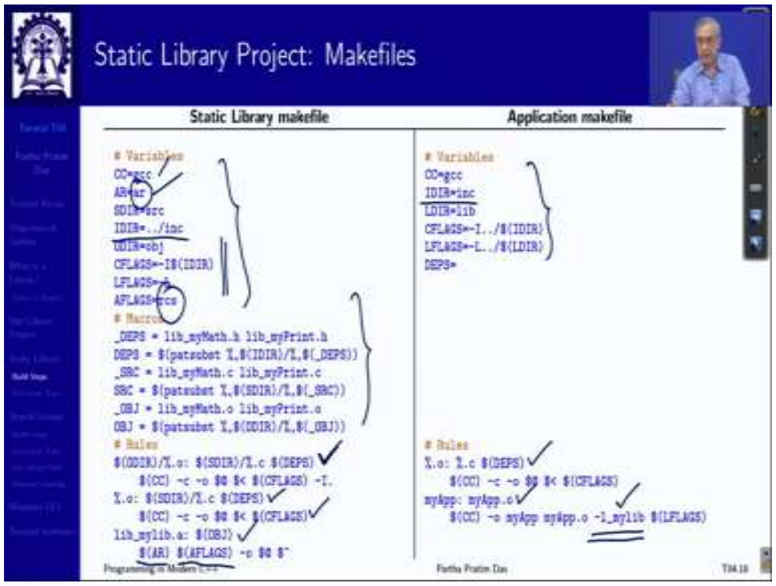
And use it, I can use the index and get to know where to find it inside the archive file. So, this is this is kind of is a typical process of doing ar rcs the archive name, followed by all the .o that you want to put. So, once we have bundled them together, then the way we can use it is simply in the place of where we were writing the .os here, here we were writing this .o. In place of that you put the archive because it already has both.

And then rest of the GCC command remain same, we say -L dot means the library to be found in the current folder. We can also replace, we can instead of keeping it in my current, I can put it at a standard library path also we will talk about that we can put in a default library path. And if we do that, then I can simply do -L. So, just remember what the .a assumes lib is a fixed starting word, followed by that is the actual name of the library.

So, the actual name of the library in this case is underscore mylib. And then you have .a. So, if you want to refer to it from not by the file name, this is the file name. But by the concept that it is mylib library, then you strip the lib part, you strip the dot a part, the name of that is L and name of that is minus, underscore my lib. And then you put L in front of that to mean that it is a library.

And so what you are is saying that -L that is what library I am going to use -L says that I am going to use a library what library am I going to use this. And when I say that what it will look for, it will look for a actual file which is lib_mylib.a. So, this is the, basic process that you will you will have.

(Refer Slide Time: 30:10)



Static Library Project: Makefiles

Static Library makefile

```
# Variables
CC=gcc
AR=ar
SDIR=src
IDIR=../inc
ODIR=obj
CFLAGS=-I$(IDIR)
LFLAGS=-L
AFLAGS=rcs
# Macros
_DEPS = lib_myMath.h lib_myPrint.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = lib_myMath.c lib_myPrint.c
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = lib_myMath.o lib_myPrint.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))
# Rules
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS) -I.
%.o: $(SDIR)/%.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)
lib_mylib.a: $(OBJ)
	$(AR) $(AFLAGS) -o $@ $^
```

Application makefile

```
# Variables
CC=gcc
IDIR=inc
LDIR=lib
CFLAGS=-I../$(IDIR)
LFLAGS=-L../$(LDIR)
DEPS=
# Rules
%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)
myApp: myApp.o
	$(CC) -o myApp myApp.o -l_mylib $(LFLAGS)
```

Programming in Modern C++ Partha Pratim Das T04.18

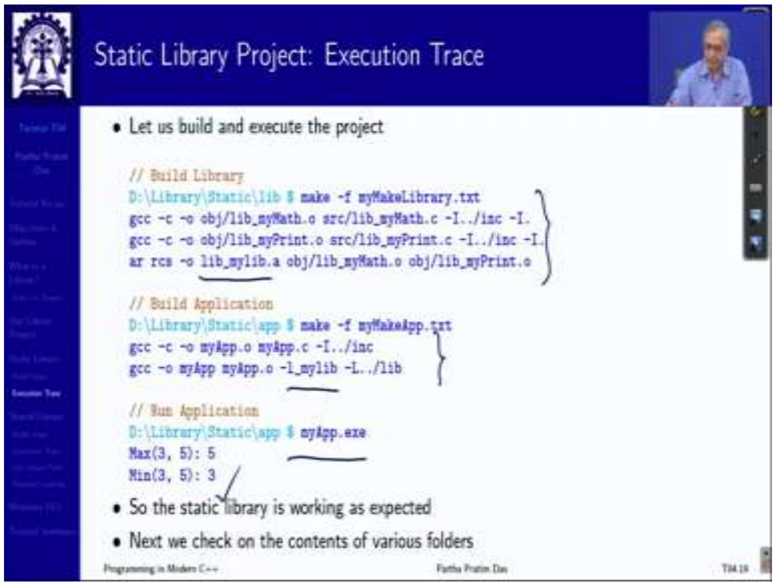
So, now the two make files, one for the library, which we normally do not see and now we are trying to learn and one is the application make file, application make file is very easy to see that typical variables as we have seen the rule for .o the rule for the final application building. And this is the only difference. I am saying that I will use this library from the default location. Now to make this to build this I have the static library make file.

This is there ar needs to be defined because now I am going to create an archive, then I have to make sure according to my directory structure, and the application position as to where this should actually be because I and c must be available to both. We can make of course we can make copies. But normally, at least I am initially suggesting that we do not make copies because if you make copies and happen to change one include file it will not be reflected in the other and you will spend your entire night looking for what is the bug.

So, other flags and folders these are dependency macros, which expand the things which we have seen already. This is the rule to build .o files you have seen that and this is the way to build your final output. This is the .o and here you have the archive command. So, you have $AR, which is ar you have dollar a flags, which is rcs, so it in that process it builds. So, once you have run this make file your static library will be built.

And then you can use that in your application make file to build the final application, the static library becomes a part of your application. Try this out with this example and then start building up further on the example.

(Refer Slide Time: 32:27)



Static Library Project: Execution Trace

- Let us build and execute the project

```
// Build Library
D:\Library\Static\lib $ make -f myMakeLibrary.txt
gcc -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
ar rcs -o lib_mylib.a obj/lib_myMath.o obj/lib_myPrint.o

// Build Application
D:\Library\Static\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o -l_mylib -L../lib

// Run Application
D:\Library\Static\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

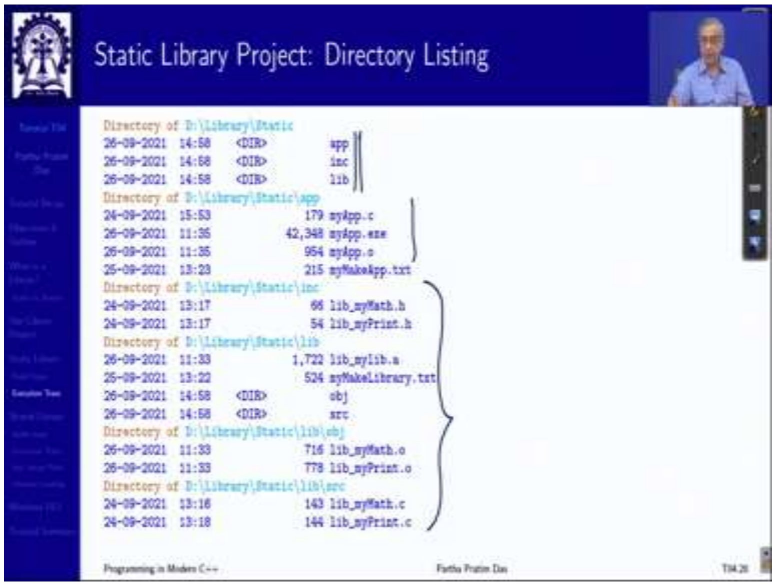
- So the static library is working as expected
- Next we check on the contents of various folders

Programming in Modern C++ Partha Pratim Das T04.19

And if you just run that and see the execution trace, because make will keep on writing what it is doing on your console, it will be able to see that for the static library it does this it is builds the two .o and then finally archives them into the dot archives, all of that into the .a and the application will build the .o and then link with the .a. And then finally you execute, it will be able to see the expected result.

The only thing is now the actual function implementations have all been separated into a static library with an archive. Big learning what you thought is only compiler writers can do is no you can do it so easily yourself is just using make in a little different way.

(Refer Slide Time: 33:25)



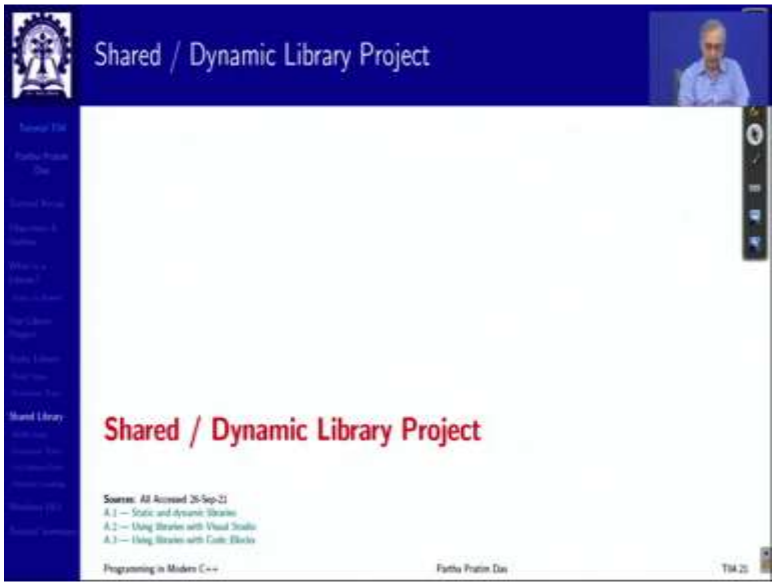
Static Library Project: Directory Listing

```
Directory of D:\Library\Static
26-09-2021  14:58    <DIR>          app
26-09-2021  14:58    <DIR>          inc
26-09-2021  14:58    <DIR>          lib
Directory of D:\Library\Static\app
24-09-2021  15:53               179 myApp.c
26-09-2021  11:35            42,348 myApp.exe
26-09-2021  11:35               954 myApp.o
25-09-2021  13:23               215 myMakeApp.txt
Directory of D:\Library\Static\inc
24-09-2021  13:17                66 lib_myMath.h
24-09-2021  13:17                54 lib_myPrint.h
Directory of D:\Library\Static\lib
26-09-2021  11:33             1,722 lib_mylib.a
25-09-2021  13:22               524 myMakeLibrary.txt
26-09-2021  14:58    <DIR>          obj
26-09-2021  14:58    <DIR>          src
Directory of D:\Library\Static\lib\obj
26-09-2021  11:33               716 lib_myMath.o
26-09-2021  11:33               778 lib_myPrint.o
Directory of D:\Library\Static\lib\src
24-09-2021  13:16               143 lib_myMath.c
24-09-2021  13:18               144 lib_myPrint.c
```

Programming in Modern C++ Partha Pratim Das T04.20

If you go back and check on your directory folder, this is actually the dump my of my directory on that date I did it over 24 to 26, September 2021, then you will find that these are my different app, int and lib directories and these are my application ones this is my whole library stuff and so on. I would just insist just reproduce this first in your system so that you get confidence and then you start building with different variants of this.

(Refer Slide Time: 34:01)



Shared / Dynamic Library Project

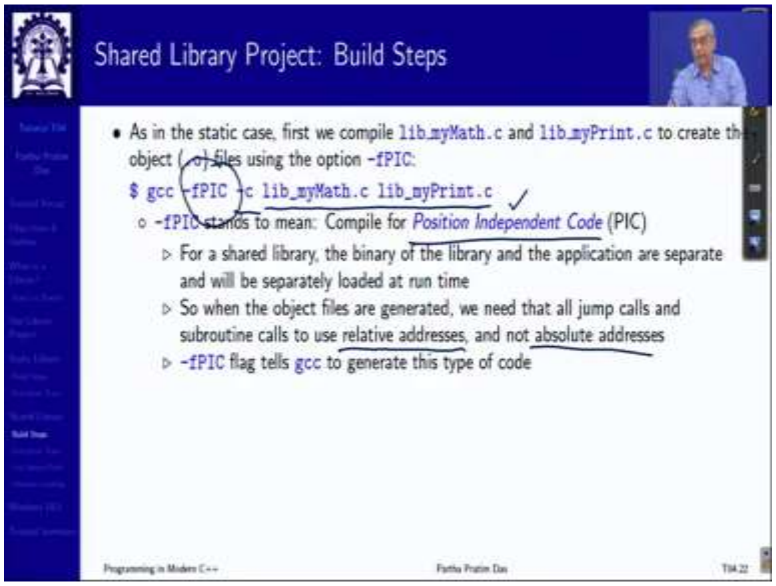
Shared / Dynamic Library Project

Sources: All Accessed 26-Sep-21
A.1 — Static and dynamic libraries
A.2 — Using libraries with Visual Studio
A.3 — Using libraries with Code::Blocks

Programming in Modern C++ Partha Pratim Das T04.21

What happens if I have a shared if I want to do a shared or dynamic library? So, this was all about static libraries. So, the code has archive ar code to my _mylib is become a part of my final application.

(Refer Slide Time: 34:18)



Shared Library Project: Build Steps

- As in the static case, first we compile `lib_myMath.c` and `lib_myPrint.c` to create th object (.o) files using the option `-fPIC`:

```
$ gcc -fPIC -c lib_myMath.c lib_myPrint.c
```

  - `-fPIC` stands to mean: Compile for *Position Independent Code* (PIC)
    - For a shared library, the binary of the library and the application are separate and will be separately loaded at run time
    - So when the object files are generated, we need that all jump calls and subroutine calls to use relative addresses, and not absolute addresses
    - `-fPIC` flag tells `gcc` to generate this type of code

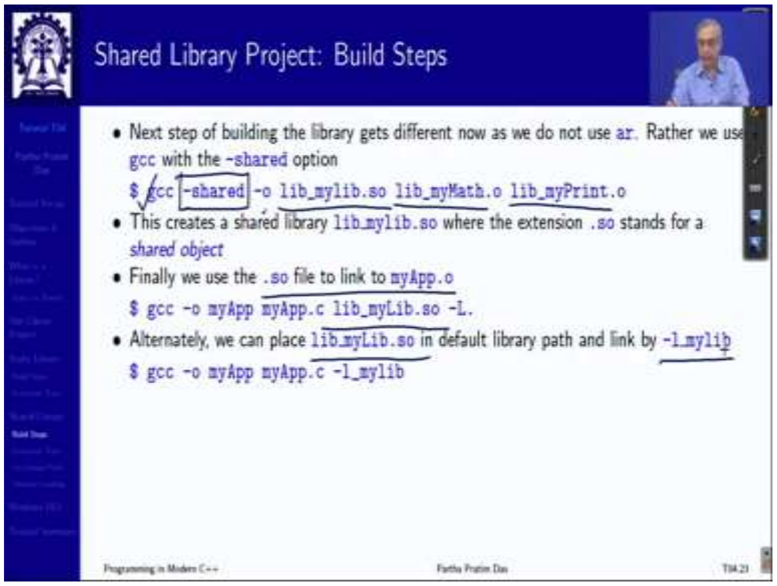
Programming in Modern C++ Partha Pratim Das T04.22

Now to build a to create a shared library or to create a .o for the shared library object file for the shared library. This part is the same. What is specifically put as an option to GCC is -fPIC. Very commonly the community refers to it as fPIC. What it means position independent code. So, what you are trying to say just try to understand when you are making a static library, you are going to put it attach it to your application function main is calling myMax or myMin.

So, the codes are put together. So, the linker knows in the main, if it is calling myMax, where it is to be found with respect to that point, absolute address when you make dynamic, you do not have that, because that part, the actual function body of myMax or my min will not be a part of the application, it is somewhere else it will come in at the runtime you do not know where it will come in. So, wherever it comes in, it has to work.

So, all that you need to do is to generate a code which does not depend on the address or rather, it will depend on a relative address that is you are not saying that with respect to myMin or call of myMax in the min, this is where you can find myMax, you are saying this call says that I will find it somewhere. Wherever I load in that, I have the offset to find where is myMax. So, that is the basic position independent code, again, a little bit of part of theory, but helps to understand.

(Refer Slide Time: 36:25)



Shared Library Project: Build Steps

- Next step of building the library gets different now as we do not use ar. Rather we use gcc with the -shared option
  ```
  $ gcc -shared -o lib_mylib.so lib_myMath.o lib_myPrint.o
  ```
- This creates a shared library lib_mylib.so where the extension .so stands for a *shared object*
- Finally we use the .so file to link to myApp.o
  ```
  $ gcc -o myApp myApp.c lib_myLib.so -L.
  ```
- Alternately, we can place lib_myLib.so in default library path and link by -l_mylib
  ```
  $ gcc -o myApp myApp.c -l_mylib
  ```

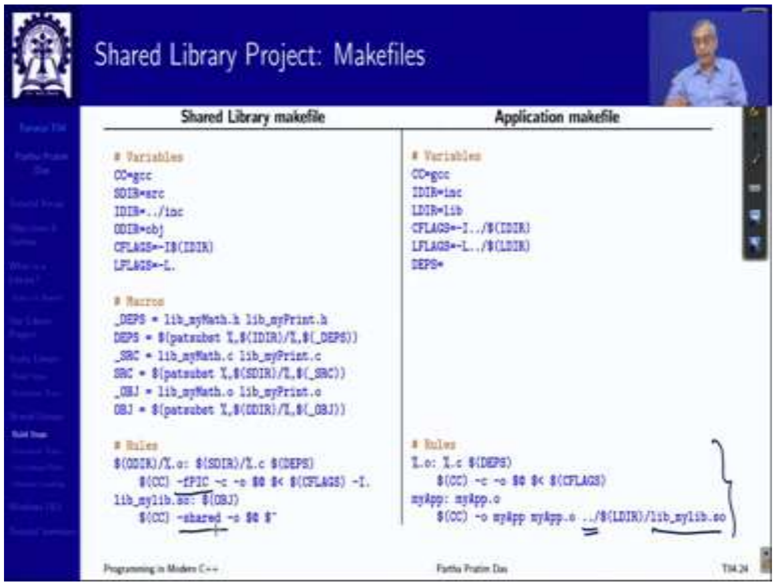
Programming in Modern C++ Partha Pratim Das T04.21

So, then you have multiple of them. So, you have .o, which are now, fPIC .o that is their position independent .o. So, you want to kind of bundle them together or tie them together. So, you do not use ar, for shared liability, GCC has its own option, it is called -shared. So, what you do is you take the .o files, run GCC on it, put your target as lib underscore mylib.so not .a shared object, that is just a different, protocol and you use this option -shared.

So, it does the same thing, it is kind of puts them archives puts them all together, but also creates enough linking information, dynamic referencing information for you. So, to use this, we will use the same so in the final executable built. And we can similarly put .so in the default library path and use it by the same syntax of -L _mylib, these are all same and similar.

The only two differences are when you first create the .o for all those translation units which will go into the shared library you need to do -fPIC. And the other is when you bundle them together you do not use ar command use GCC with -shared option and you should be you would be ready to go.

(Refer Slide Time: 38:25)



Shared Library Project: Makefiles

Shared Library makefile

```
# Variables
CC=gcc
SDIR=src
IDIR=../inc
ODIR=obj
CFLAGS=-I$(IDIR)
LFLAGS=-L.

# Macros
_DEPS = lib_myMath.h lib_myPrint.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = lib_myMath.c lib_myPrint.c
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = lib_myMath.o lib_myPrint.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

# Rules
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -fPIC -c -o $@ $< $(CFLAGS) -I.
lib_mylib.so: $(OBJ)
    $(CC) -shared -o $@ $^
```

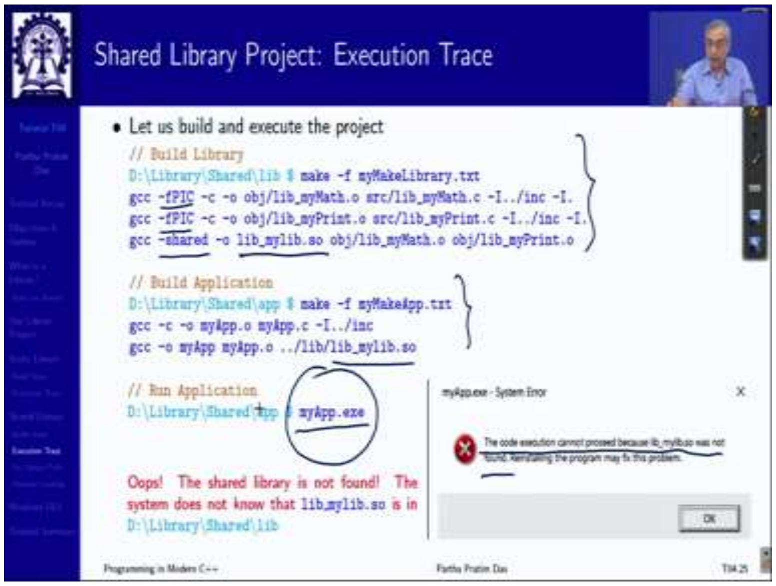
Application makefile

```
# Variables
CC=gcc
IDIR=inc
LDIR=lib
CFLAGS=-I../$(IDIR)
LFLAGS=-L../$(LDIR)
DEPS=

# Rules
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
myApp: myApp.o
    $(CC) -o myApp myApp.o ../$(LDIR)/lib_mylib.so
```

Programming in Modern C++ Partha Pratim Das T04.24

So, in terms of the make file, they are very easy to see here there is hardly any difference except this .so this dot dot are for my directory structure here you can see the differences in terms of this, good enough.

(Refer Slide Time: 38:46)



Shared Library Project: Execution Trace

- Let us build and execute the project

```
// Build Library
D:\Library\Shared\lib $ make -f myMakeLibrary.txt
gcc -fPIC -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -fPIC -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
gcc -shared -o lib_mylib.so obj/lib_myMath.o obj/lib_myPrint.o

// Build Application
D:\Library\Shared\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o ../lib/lib_mylib.so

// Run Application
D:\Library\Shared\app $ myApp.exe
```

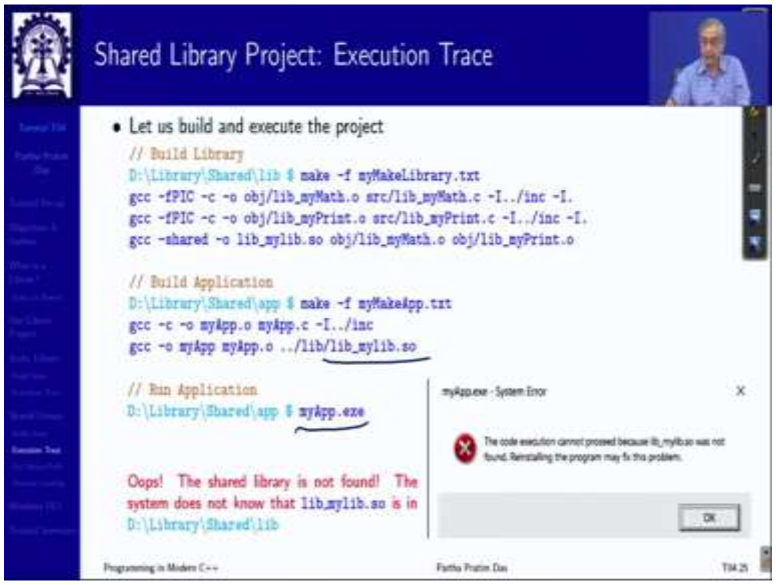
myApp.exe - System Error

The code execution cannot proceed because lib_mylib.so was not found. Reinstalling the program may fix this problem.

OK

Oops! The shared library is not found! The system does not know that lib_mylib.so is in D:\Library\Shared\lib

Programming in Modern C++ Partha Pratim Das T04.25



Shared Library Project: Execution Trace

- Let us build and execute the project

```
// Build Library
D:\Library\Shared\lib $ make -f myMakeLibrary.txt
gcc -fPIC -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -fPIC -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
gcc -shared -o lib_mylib.so obj/lib_myMath.o obj/lib_myPrint.o

// Build Application
D:\Library\Shared\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o ../lib/lib_mylib.so

// Run Application
D:\Library\Shared\app $ myApp.exe
```

myApp.exe - System Error

The code execution cannot proceed because lib_mylib.so was not found. Reinstalling the program may fix this problem.

OK

Oops! The shared library is not found! The system does not know that lib_mylib.so is in D:\Library\Shared\lib

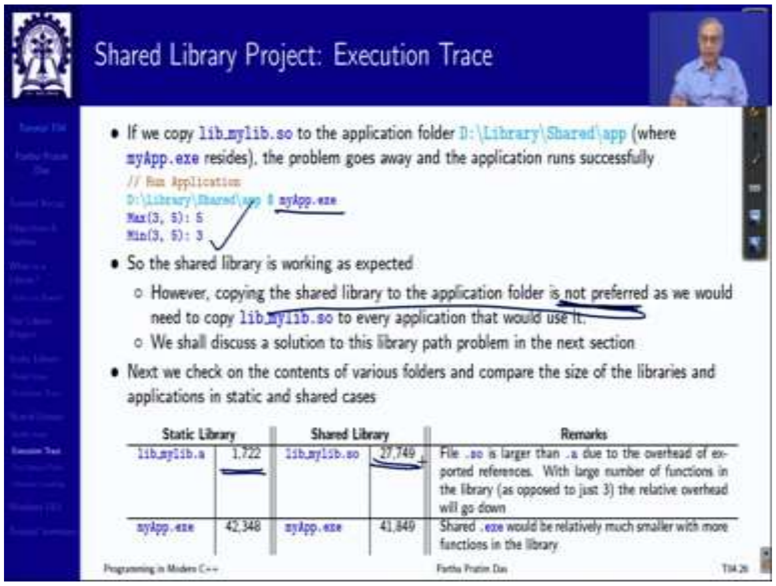
Programming in Modern C++ Partha Pratim Das T04.25

So, now let us go ahead and complete the task. So, I will do I will see the traces of execution. This is my build process for the library. It does the same thing with -fPIC for the .o -shared a my this object will get created shared object. I use the shared object do the same thing again, to build my application. I am very excited I want to try running it. It does not it throws an exception it crashes it gives a system error.

It says I cannot find the code execution cannot proceed because lib_mylib.so was not found. Why does it say so? It says so because unlike the static library, in the static library, these quotes were part of myapp.exe lib_mylib .a became a part of mylib.a myapp.exe. Now it is not so now I to be able to run to be able to execute this, I also need this file at the runtime because that is what the actual function quotes are.

So, I need to tell the system I need to somehow communicate as to where this .so is otherwise it will not be able to say that it go there and start executing the function. So, that is the basic gap that happens.

(Refer Slide Time: 40:36)



Shared Library Project: Execution Trace

- If we copy lib_mylib.so to the application folder D:\Library\Shared\app (where myApp.exe resides), the problem goes away and the application runs successfully

```
// Run Application
D:\Library\Shared\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

- So the shared library is working as expected
  - However, copying the shared library to the application folder is not preferred as we would need to copy lib_mylib.so to every application that would use it.
  - We shall discuss a solution to this library path problem in the next section
- Next we check on the contents of various folders and compare the size of the libraries and applications in static and shared cases

Static Library		Shared Library		Remarks
lib_mylib.a	1,722	lib_mylib.so	27,749	File .so is larger than .a due to the overhead of exported references. With large number of functions in the library (as opposed to just 3) the relative overhead will go down
myApp.exe	42,348	myApp.exe	41,849	Shared .exe would be relatively much smaller with more functions in the library

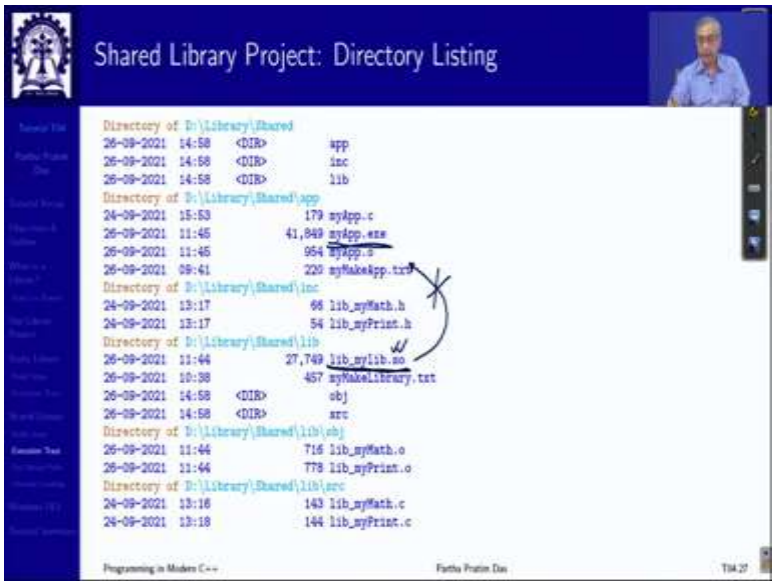
Programming in Modern C++ Partha Pratim Das T04.26

So, there could be some ways of handling that one could be a very simple way could be that I copy this my lib_mylib.so in the application folder. The current folder, it will always look for current folder. If I copy that simply if I copy that file, and again try this, you will find that it works fine. But do not get overly excited. Because certainly, there will be several such .so files and there are several projects which are using them what would you do you will keep on copying all .so into all projects that would be a mess.

And again, you did shared object or dynamic library, one of the objective was to manage version but now you are making multiple copies of that dynamic library code. Therefore, you are bringing back the version management nightmare. So, that is, it is certainly not preferred at all. So, we have to find a solution to that you can also see what the different size and all that are there in in contrast to what I had said theoretically that, static library will be bigger and dynamic library will be smaller here you will see that it is exactly the opposite.

That is because we have very small libraries. So, naturally dynamic library or shared library has a lot of index and overhead code that need to be put into that so that it can be found at the runtime. So, that code is actually dominating, but if you see bigger libraries than what I said earlier is actually will turn out to be correct.

(Refer Slide Time: 42:34)



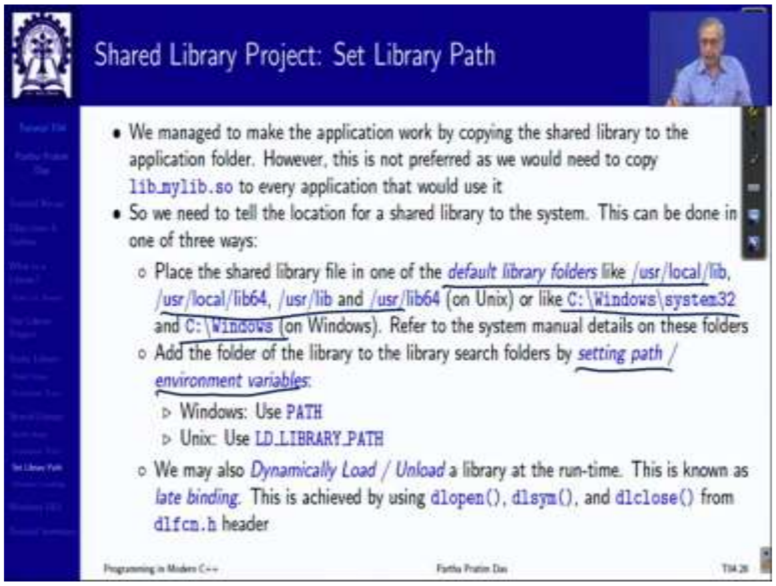
Shared Library Project: Directory Listing

```
Directory of D:\Library\Shared
26-09-2021  14:58    <DIR>          app
26-09-2021  14:58    <DIR>          inc
26-09-2021  14:58    <DIR>          lib
Directory of D:\Library\Shared\app
24-09-2021  15:53               179 myApp.c
26-09-2021  11:45            41,849 myApp.exe
26-09-2021  11:45               964 myApp.o
26-09-2021  09:41               220 myMakeApp.txt
Directory of D:\Library\Shared\inc
24-09-2021  13:17                66 lib_myMath.h
24-09-2021  13:17                54 lib_myPrint.h
Directory of D:\Library\Shared\lib
26-09-2021  11:44            27,749 lib_mylib.so
26-09-2021  10:38               457 myMakeLibrary.txt
26-09-2021  14:58    <DIR>          obj
26-09-2021  14:58    <DIR>          src
Directory of D:\Library\Shared\lib\obj
26-09-2021  11:44               716 lib_myMath.o
26-09-2021  11:44               778 lib_myPrint.o
Directory of D:\Library\Shared\lib\src
24-09-2021  13:16               143 lib_myMath.c
24-09-2021  13:18               144 lib_myPrint.c
```

Programming in Modern C++ Partha Pratim Das T04.27

This is my directory listing and what is important is this is where my .so is and this is where my exe is and therefore, it does not is not able to find it out when I tried to execute, I will have to copy it here, which is not a preferred solution. So, somehow I have to be able to tell my application that go there and you will find the .so that is certainly the preferred option.

(Refer Slide Time: 43:11)



Shared Library Project: Set Library Path

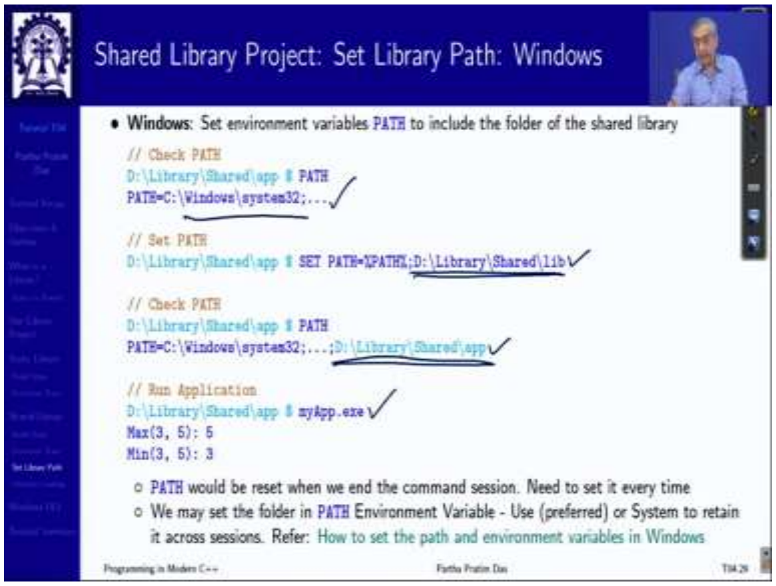
- We managed to make the application work by copying the shared library to the application folder. However, this is not preferred as we would need to copy `lib_mylib.so` to every application that would use it
- So we need to tell the location for a shared library to the system. This can be done in one of three ways:
  - Place the shared library file in one of the *default library folders* like /usr/local/lib, /usr/local/lib64, /usr/lib and /usr/lib64 (on Unix) or like `C:\Windows\system32` and `C:\Windows` (on Windows). Refer to the system manual details on these folders
  - Add the folder of the library to the library search folders by *setting path / environment variables*:
    - Windows: Use `PATH`
    - Unix: Use `LD_LIBRARY_PATH`
  - We may also *Dynamically Load / Unload* a library at the run-time. This is known as *late binding*. This is achieved by using `dlopen()`, `dlsym()`, and `dlclose()` from `dlfcn.h` header

Programming in Modern C++ Partha Pratim Das T04.28

So, that process is known as Set Library path tell the system that where the you expect that library such libraries or library binary is to be found. So, that it can be done in multiple ways when one is both in Linux, these are certain folders in directories in Linux, which are conventionally taken to be locations have shared libraries. So, if you can keep it in any one of these and the system already knows that shared libraries will have to be looked for there. On Windows, these are typically Windows system 32 or Windows folders.

So, one is to place them in the defaults other would be you can just set a path or an environmental variable. There is a in environmental path variable in Windows, there is LD library path in Unix you can just once in your system set that to a folder where you expect your .so or .dll programs, binary is to be put and they will be found in those. Alternately, there is a third mechanism that you can dynamically find and also use that at the runtime through the program.

(Refer Slide Time: 44:40)



Shared Library Project: Set Library Path: Windows

- **Windows**: Set environment variables PATH to include the folder of the shared library

```
// Check PATH
D:\Library\Shared\app $ PATH
PATH=C:\Windows\system32;...

// Set PATH
D:\Library\Shared\app $ SET PATH=%PATH%;D:\Library\Shared\lib

// Check PATH
D:\Library\Shared\app $ PATH
PATH=C:\Windows\system32;...;D:\Library\Shared\app

// Run Application
D:\Library\Shared\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

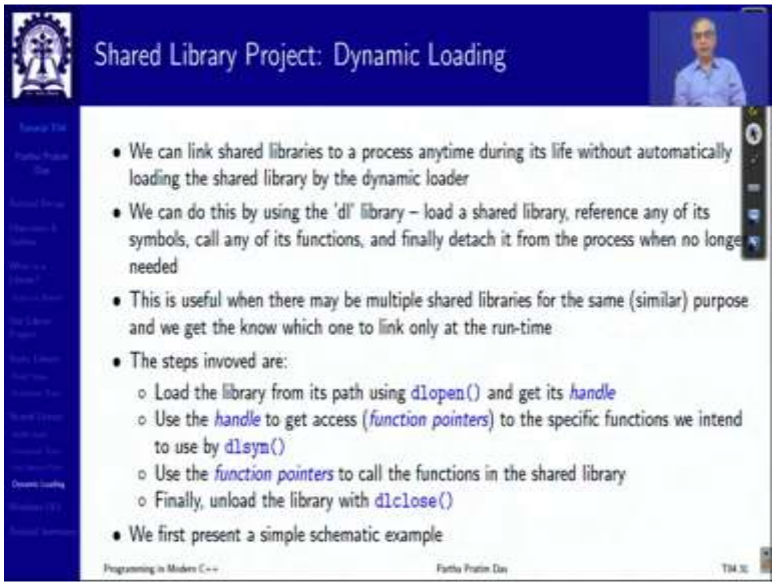
- PATH would be reset when we end the command session. Need to set it every time
- We may set the folder in PATH Environment Variable - Use (preferred) or System to retain it across sessions. Refer: How to set the path and environment variables in Windows

Programming in Modern C++ Partha Pratim Das T04.29

So, if you do the set path like you can do something like this and then you check the path is correct. So, you have added this, along with this party have added your own path where your library so is there. So, if you check you will find it there and then your application runs fine. This will work fine, but the only catch is that this is for the particular session of your command prompt. So, you come to the next session, these are this disappeared.

So, if you want them to be persistent, you will have to go and edit the path variables. And there are different ways you can edit that in, in Unix, as well as in Windows.

(Refer Slide Time: 45:28)



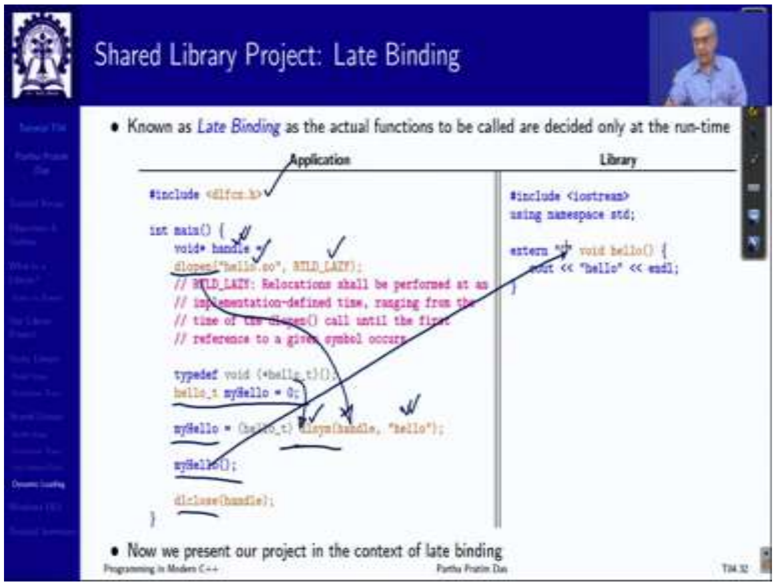
Shared Library Project: Dynamic Loading

- We can link shared libraries to a process anytime during its life without automatically loading the shared library by the dynamic loader
- We can do this by using the 'dl' library – load a shared library, reference any of its symbols, call any of its functions, and finally detach it from the process when no longer needed
- This is useful when there may be multiple shared libraries for the same (similar) purpose and we get the know which one to link only at the run-time
- The steps invoved are:
  - Load the library from its path using `dlopen()` and get its *handle*
  - Use the *handle* to get access (*function pointers*) to the specific functions we intend to use by `dlsym()`
  - Use the *function pointers* to call the functions in the shared library
  - Finally, unload the library with `dlclose()`
- We first present a simple schematic example

Programming in Modern C++ Partha Pratim Das T04.31

Now, the third mechanism, so one was to copy in the one was to put it in the default folders, one was to tell the system that this is the folder where you find it. The third is to find it dynamically. So, that is a really interesting, what you say is, I will in my program, I will tell that this is the name of the of my dynamic library and ask the system to load that, then I will tell that this is the name of my function, so that it can go into that dynamic library and find that function and returns my function pointer, values like that. But it is simple, two levels of indirection.

(Refer Slide Time: 46:15)



Shared Library Project: Late Binding

- Known as *Late Binding* as the actual functions to be called are decided only at the run-time

Application

```
#include <dlfcn.h>

int main() {
    void* handle =
    dlopen("hello.so", RTLD_LAZY);
    // RTLD_LAZY: Relocations shall be performed at an
    // implementation-defined time, ranging from the
    // time of the dlopen() call until the first
    // reference to a given symbol occurs

    typedef void (*hello_t)();
    hello_t myHello = 0;

    myHello = (hello_t) dlsym(handle, "hello");

    myHello();

    dlclose(handle);
}
```

Library

```
#include <iostream>
using namespace std;

extern "C" void hello() {
    cout << "hello" << endl;
}
```

- Now we present our project in the context of late binding

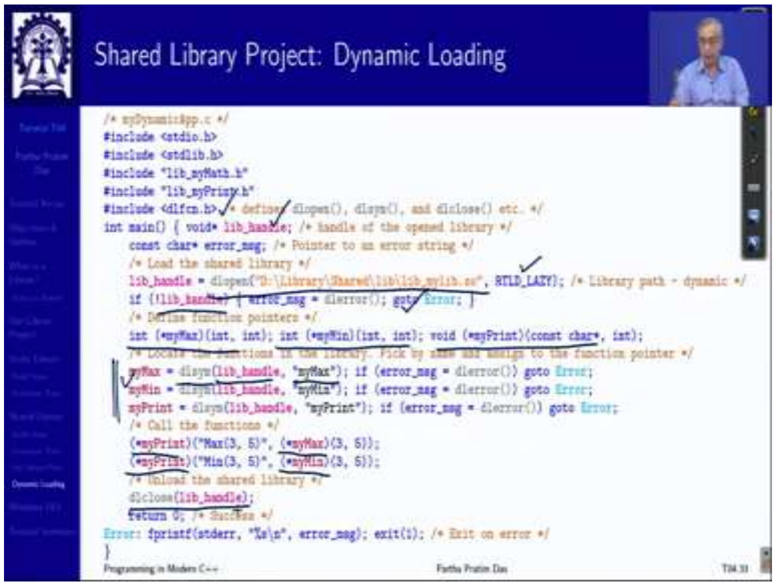
Programming in Modern C++ Partha Pratim Das T04.32

So, you use a header dynamic library function dlfcn, which has some, library functions like dlopen, dlsym, dlclose like that. So, you pass the name of the standard library, name of the shared library to dl open this is a parameter to say, how to load it, do you load it immediately or do you load it when you need so RT LAZY. So, it will be done at the implementation defined time, then you what you say is, we have not done this, you define a handle.

Handle is a function pointer, which takes nothing gives you nothing takes a void returns a void. So, because you do not know what kind of function is there, so you need to assume that it is a function without any parameter and returning nothing. So, then in dlsym, you pass the handle in which you have opened your .so. This comes here and specify the particular function you are looking for in your shared object that returns you a pointer of this type.

Cast it to myHello, put it to myHello, call myHello. myHello will now call this function because it has got that handle. So, that is that is the is in abstraction, the simple mechanism.

(Refer Slide Time: 48:08)



Shared Library Project: Dynamic Loading

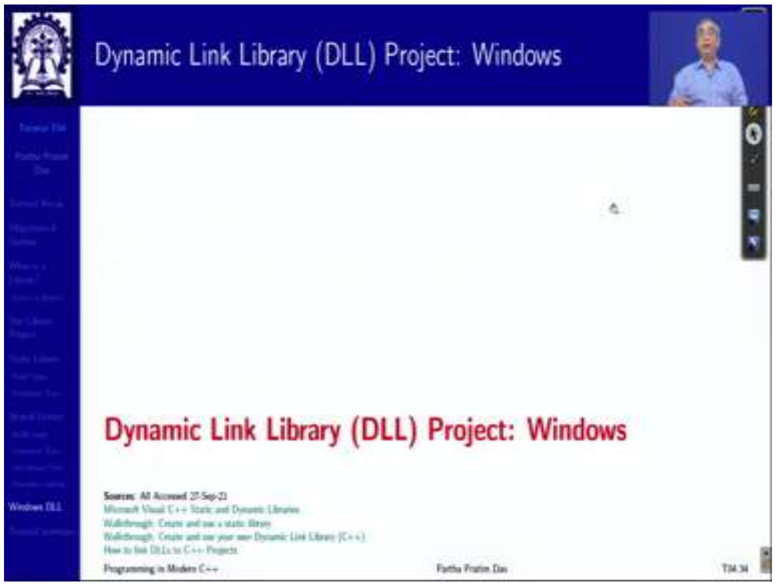
```
/* myDynamicApp.c */
#include <stdio.h>
#include <stdlib.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
#include <dlfcn.h> /* defines dlopen(), dlsym(), and dlclose() etc. */
int main() { void* lib_handle; /* handle of the opened library */
    const char* error_msg; /* Pointer to an error string */
    /* Load the shared library */
    lib_handle = dlopen("D:\Library\Shared\lib\lib_mylib.so", RTLD_LAZY); /* Library path - dynamic */
    if (!lib_handle) { error_msg = dlerror(); goto Error; }
    /* Define function pointers */
    int (*myMax)(int, int); int (*myMin)(int, int); void (*myPrint)(const char*, int);
    /* Locate the functions in the library. Pick by name and assign to the function pointer */
    myMax = dlsym(lib_handle, "myMax"); if (error_msg = dlerror()) goto Error;
    myMin = dlsym(lib_handle, "myMin"); if (error_msg = dlerror()) goto Error;
    myPrint = dlsym(lib_handle, "myPrint"); if (error_msg = dlerror()) goto Error;
    /* Call the functions */
    (*myPrint)("Max(3, 5)", (*myMax)(3, 5));
    (*myPrint)("Min(3, 5)", (*myMin)(3, 5));
    /* Unload the shared library */
    dlclose(lib_handle);
    return 0; /* Success */
Error: fprintf(stderr, "%s\n", error_msg); exit(1); /* Exit on error */
}
```

Programming in Modern C++ Partha Pratim Das T34.33

And here I have given the more detailed code here is the inclusion here is a library handle. So, I do a dlopen of my code with RT LAZY, if it is not found at that point or there is some error, then the handle will be null, I go out and give an error. Otherwise, I will use that handle to find the function by name. So, myMax specify, it returns me dlsym returns a pointer to that which I put to myMax.

So, for that, these function pointers, myMax, myMin, myPrint has been defined. And I get these three function pointers, according to the relative address within my library, so and then I dereference these functions and use them. When I am done, I close the handle. So, this is a third mechanism where I can control specifically when I want to load it. And when I close, and I am done with it, so I am not going to have it anymore.

(Refer Slide Time: 49:26)



Dynamic Link Library (DLL) Project: Windows

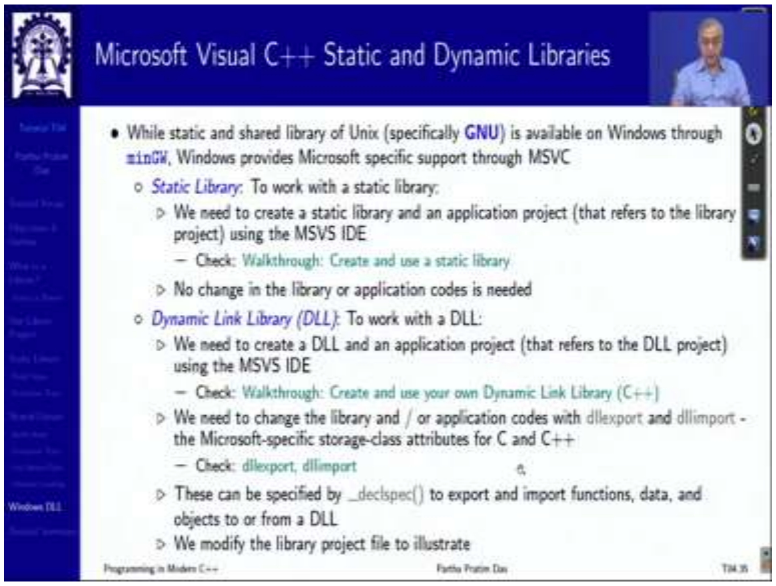
Dynamic Link Library (DLL) Project: Windows

Sources: All Accessed 27-Sep-21
Microsoft Visual C++ Static and Dynamic Libraries
Walkthrough: Create and use a static library
Walkthrough: Create and use your own Dynamic Link Library (C++)
How to link DLLs to C++ Projects

Programming in Modern C++ Partha Pratim Das T04.34

So, if you have more, personalized, dynamic libraries, we will be using that. So, that is another that is an alternate mechanism. And both of all of these mechanisms are available both on Unix as well as on Windows.

(Refer Slide Time: 49:43)



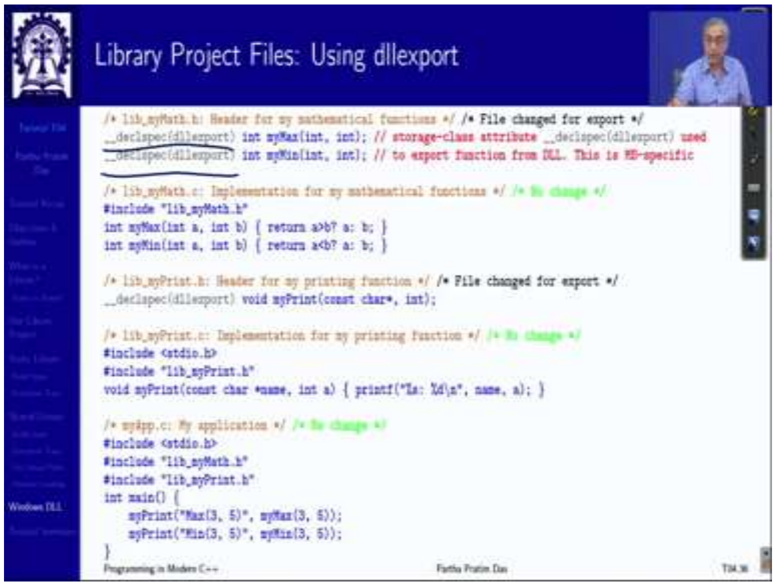
Microsoft Visual C++ Static and Dynamic Libraries

- While static and shared library of Unix (specifically **GNU**) is available on Windows through `minGW`, Windows provides Microsoft specific support through MSVC
  - *Static Library*: To work with a static library:
    - We need to create a static library and an application project (that refers to the library project) using the MSVS IDE
      - Check: Walkthrough: Create and use a static library
    - No change in the library or application codes is needed
  - *Dynamic Link Library (DLL)*: To work with a DLL:
    - We need to create a DLL and an application project (that refers to the DLL project) using the MSVS IDE
      - Check: Walkthrough: Create and use your own Dynamic Link Library (C++)
    - We need to change the library and / or application codes with dllexport and dllimport - the Microsoft-specific storage-class attributes for C and C++
      - Check: dllexport, dllimport
    - These can be specified by __declspec() to export and import functions, data, and objects to or from a DLL
    - We modify the library project file to illustrate

Programming in Modern C++ Partha Pratim Das T04.35

In addition, windows have specific dynamic library dll projects, and the static library mechanisms. I will not I am not going through the details of that because these are to be done through the Visual Studio IDE. It is a commercial one, though, some academic versions are available, but in general, it is a commercial one. So, we do not intend to go deep into that.

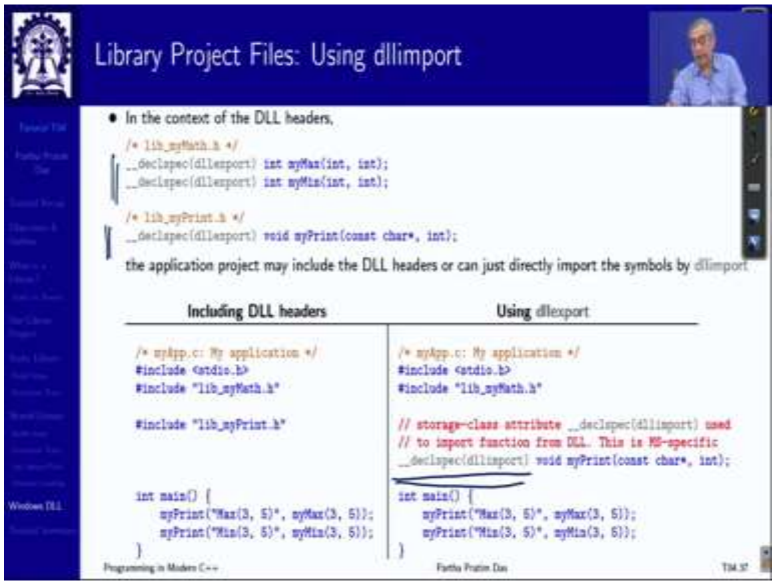
(Refer Slide Time: 50:09)



Library Project Files: Using dllexport

```
/* lib_myMath.h: Header for my mathematical functions */ /* File changed for export */
__declspec(dllexport) int myMax(int, int); // storage-class attribute __declspec(dllexport) used
__declspec(dllexport) int myMin(int, int); // to export function from DLL. This is MS-specific

/* lib_myMath.c: Implementation for my mathematical functions */ /* No change */
#include "lib_myMath.h"
int myMax(int a, int b) { return a>b? a: b; }
int myMin(int a, int b) { return a<b? a: b; }

/* lib_myPrint.h: Header for my printing function */ /* File changed for export */
__declspec(dllexport) void myPrint(const char*, int);

/* lib_myPrint.c: Implementation for my printing function */ /* No change */
#include <stdio.h>
#include "lib_myPrint.h"
void myPrint(const char *name, int a) { printf("%s: %d\n", name, a); }

/* myApp.c: My application */ /* No change */
#include <stdio.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```

Programming in Modern C++ Partha Pratim Das T04.36



Library Project Files: Using dllimport

- In the context of the DLL headers,

```
/* lib_myMath.h */
__declspec(dllexport) int myMax(int, int);
__declspec(dllexport) int myMin(int, int);

/* lib_myPrint.h */
__declspec(dllexport) void myPrint(const char*, int);
```

the application project may include the DLL headers or can just directly import the symbols by dllimport

Including DLL headers	Using dllexport

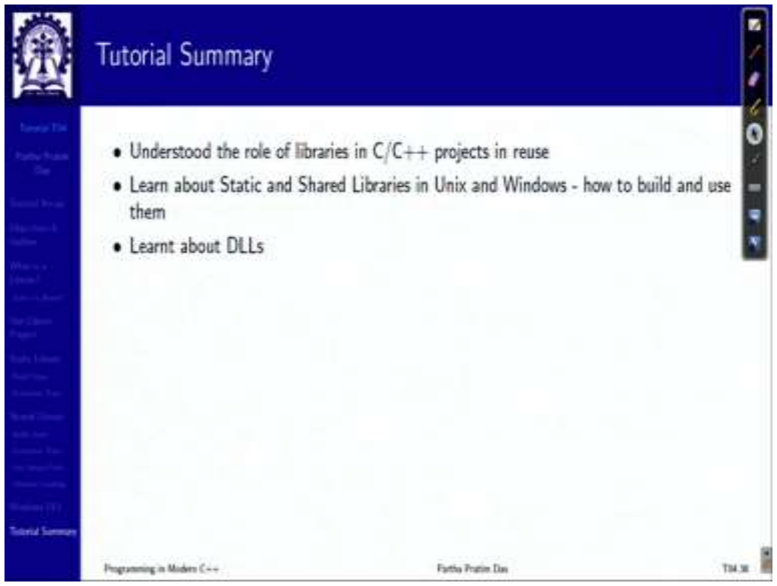
`/* myApp.c: My application */`
`#include <stdio.h>`
`#include "lib_myMath.h"`
 	
`#include "lib_myPrint.h"`
 	
`int main() {`
`    myPrint("Max(3, 5)", myMax(3, 5));`
`    myPrint("Min(3, 5)", myMin(3, 5));`
`}`
	`/* myApp.c: My application */`
	`#include <stdio.h>`
	`#include "lib_myMath.h"`
	
	`// storage-class attribute __declspec(dllimport) used`
	`// to import function from DLL. This is MS-specific`
	`__declspec(dllimport) void myPrint(const char*, int);`
	
	`int main() {`
	`    myPrint("Max(3, 5)", myMax(3, 5));`
	`    myPrint("Min(3, 5)", myMin(3, 5));`
	`}`

Programming in Modern C++ Partha Pratim Das T04.37

But if you if you happen to use that the only, difference that you will see that there are certain storage class attributes called dll specs, dllexport. So, from your doc header, you are saying that I am exporting this symbol, so I am making it available on the dll. Similarly, when you use it, you include and use it. And you can have a dllimport on this. So, you have dllexports and you have dllimport here to use in your application.

So, this is basically what the difference in as it visualizes in the windows, the rest of the build process can be executed through Windows, Mac, or through the Visual Studio IDE. So, all that I wanted to mention is there is a window specific process, particularly dealing with similar shared objects or dynamic libraries, which are called dynamically linked library or dll in Windows. But certainly, if you really need to do that, you will have to look up the manual and do this.

(Refer Slide Time: 51:29)



Tutorial Summary

- Understood the role of libraries in C/C++ projects in reuse
- Learn about Static and Shared Libraries in Unix and Windows - how to build and use them
- Learnt about DLLs

Programming in Modern C++ Partha Pratim Das T04.38

So, this brings us to the end of the tutorial series of 1 to 4 where we have taken a look at the overall build process of C, C++ project with, almost all major, objectives that it wants to serve, whether what is build pipeline, how does C preprocessor can be used to manage code, how to make build process efficient by using the make utility, how should you organize your code? And at the end, how should you build your static and dynamic libraries.

So, if you practice this, just listening to this is not enough. But if you practice this, if you execute those examples that have given and also have further examples, then you will really have no difficulty. With your C, C++ knowledge you will be able to walk into your company into your interview and actually show them that, how to do programming and how to manage projects.

So, all the best for that we conclude on this tutorial series. We will have some more tutorials on other aspects of programming project and language but that is later. Thank you very much for your attention.