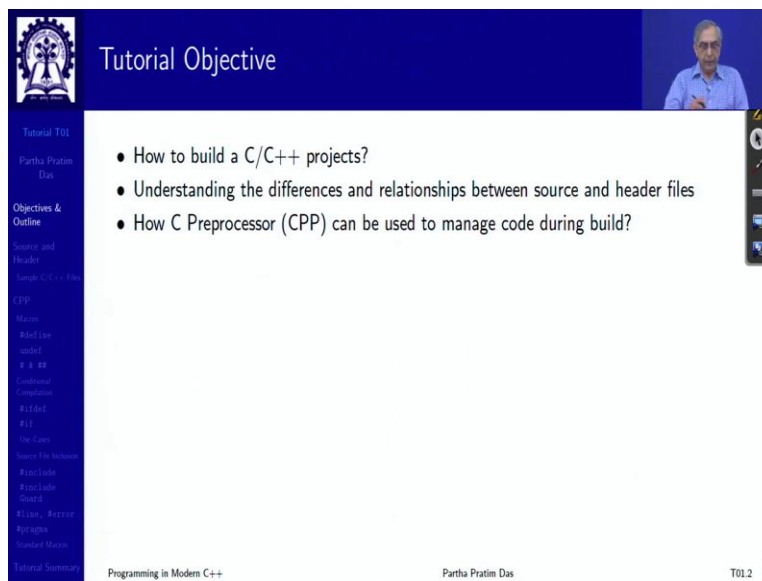


**Programming in Modern C++**  
**Professor. Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture No. 09**  
**How to build a C/C++ program?**  
**Part 1: C Preprocessor (CPP)**

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

(Refer Slide Time: 00:54)



The screenshot shows a presentation slide with a dark blue header and a white main area. The header contains the IIT Kharagpur logo on the left and the text 'Tutorial Objective' in the center. A small video inset of the professor is in the top right corner. The main area contains a bulleted list of three objectives. A vertical navigation menu is on the left side of the slide, and a control bar is on the right. The footer contains 'Programming in Modern C++', 'Partha Pratim Das', and 'T01.2'.

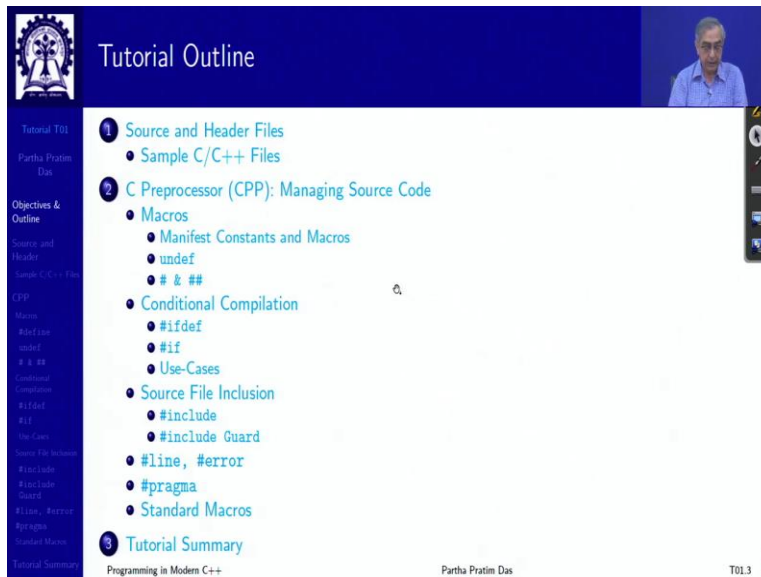
**Tutorial Objective**

- How to build a C/C++ projects?
- Understanding the differences and relationships between source and header files
- How C Preprocessor (CPP) can be used to manage code during build?

Programming in Modern C++ Partha Pratim Das T01.2

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

(Refer Slide Time: 01:43)



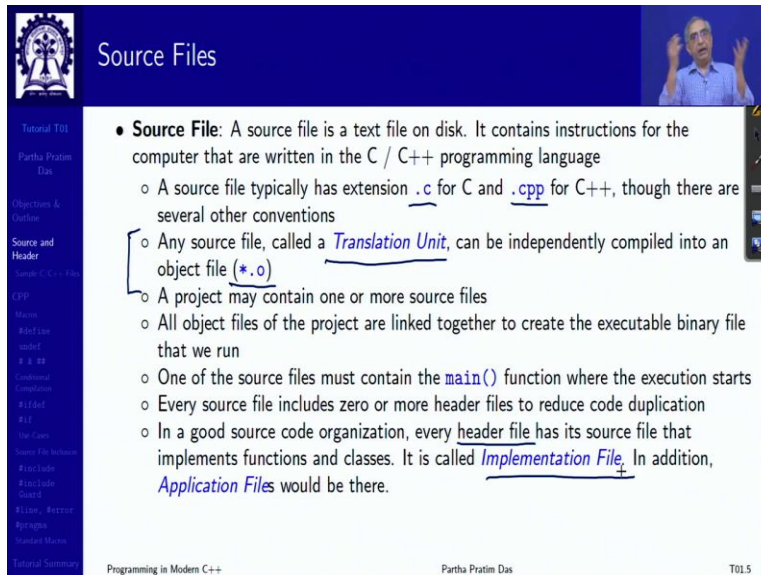
The screenshot shows a video lecture slide titled "Tutorial Outline". On the left is a navigation menu with items like "Tutorial T01", "Objectives & Outline", "Source and Header", "CPP", "Main", "#ifdef", "#if", "#include", "#pragma", and "Tutorial Summary". The main content area lists the tutorial's structure:

- 1 Source and Header Files
  - Sample C/C++ Files
- 2 C Preprocessor (CPP): Managing Source Code
  - Macros
    - Manifest Constants and Macros
    - undef
    - # & ##
  - Conditional Compilation
    - #ifndef
    - #if
    - Use-Cases
  - Source File Inclusion
    - #include
    - #include Guard
    - #line, #error
    - #pragma
    - Standard Macros
- 3 Tutorial Summary

At the bottom, it says "Programming in Modern C++" and "Partha Pratim Das". The slide number "T01.3" is in the bottom right corner.

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

(Refer Slide Time: 01:47)



The screenshot shows a video lecture slide titled "Source Files". The navigation menu on the left is similar to the previous slide, but the "Source and Header" item is highlighted. The main content area defines a source file:

- **Source File:** A source file is a text file on disk. It contains instructions for the computer that are written in the C / C++ programming language
  - A source file typically has extension `.c` for C and `.cpp` for C++, though there are several other conventions
  - Any source file, called a Translation Unit, can be independently compiled into an object file (`*.o`)
  - A project may contain one or more source files
  - All object files of the project are linked together to create the executable binary file that we run
  - One of the source files must contain the `main()` function where the execution starts
  - Every source file includes zero or more header files to reduce code duplication
  - In a good source code organization, every header file has its source file that implements functions and classes. It is called Implementation File. In addition, Application Files would be there.

At the bottom, it says "Programming in Modern C++" and "Partha Pratim Das". The slide number "T01.5" is in the bottom right corner.

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

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

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

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

(Refer Slide Time: 03:24)

The screenshot shows a presentation slide titled "Header Files" with a blue header and a small video inset of a speaker in the top right. The slide content includes:

- **Header File:** A header file is a text file on disk. It contains function declarations & macro definitions (C/C++) and class & template definitions (C++) to be shared between several source files
  - A header file typically has extension `.h` for C and `.h` or `.hpp` for C++, though there are several other conventions (or no extension for C++ Standard Library)
  - A header file is included in one or more source or header files
  - A header file is compiled as a part of the source file/s it is included in
    - ▷ **Precompiled header (PCH):** A header file may be compiled into an intermediate form that is faster to process for the compiler. Usage of PCH may significantly reduce compilation time, especially when applied to large header files, header files that include many other header files, or header files that are included in many translation units.
  - There are two types of header files. (More information in 10)
    - ▷ Files that the programmer writes are included as `#include "file"`
    - ▷ Files that comes with the compiler (*Standard Library*) are included as `#include <file>`. For C++
      - These have no extension and are specified within `std` namespace
      - The standard library files of C are prefixed with "c" with no extension in C++

Handwritten annotations on the slide include a blue circle around the word "file" in the first C++ include example, and a blue bracket around the angle brackets in the second C++ include example.

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

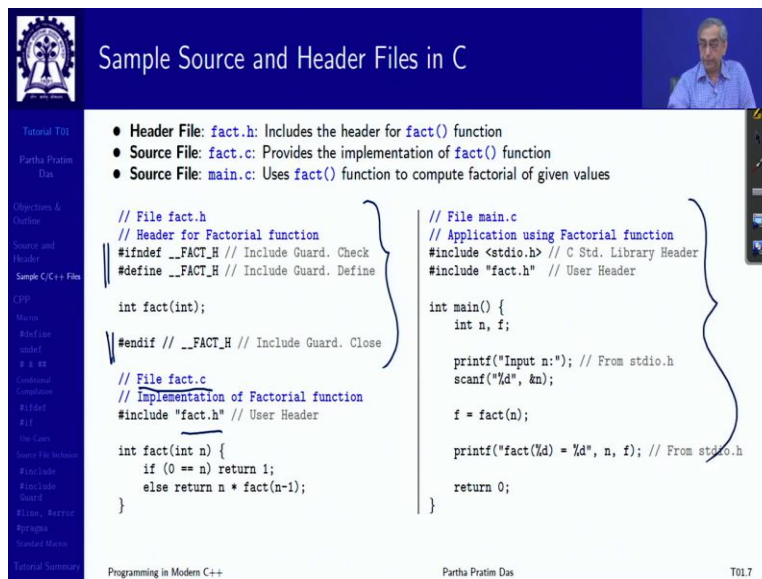
The important thing to note is there are two conventions for including the header file double quote, to be used when the header is written by you the application programmer or the, the programmer person. And this corner bracket should be used for the library inclusion you will wonder as to why these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

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

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

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

(Refer Slide Time: 05:26)



The slide, titled "Sample Source and Header Files in C", features a blue header with a logo on the left and a small video inset of the presenter on the right. The main content area is white with blue text for code and black text for bullet points. A vertical navigation menu is on the left side of the slide. The code is organized into three sections, each enclosed in a large curly brace on the right side. The first section, "Header File: fact.h", contains include guard macros and a function declaration. The second section, "Source File: fact.c", shows the implementation of the factorial function. The third section, "Source File: main.c", shows the main function that uses the factorial function. The footer of the slide includes the text "Programming in Modern C++", the name "Partha Pratim Das", and the number "T01.7".

- Header File: `fact.h`: Includes the header for `fact()` function
- Source File: `fact.c`: Provides the implementation of `fact()` function
- Source File: `main.c`: Uses `fact()` function to compute factorial of given values

```
// File fact.h
// Header for Factorial function
#ifndef _FACT_H // Include Guard. Check
#define _FACT_H // Include Guard. Define

int fact(int);

#endif // _FACT_H // Include Guard. Close

// File fact.c
// Implementation of Factorial function
#include "fact.h" // User Header

int fact(int n) {
    if (0 == n) return 1;
    else return n * fact(n-1);
}

// File main.c
// Application using Factorial function
#include <stdio.h> // C Std. Library Header
#include "fact.h" // User Header

int main() {
    int n, f;

    printf("Input n:"); // From stdio.h
    scanf("%d", &n);

    f = fact(n);

    printf("fact(%d) = %d", n, f); // From stdio.h

    return 0;
}
```

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

(Refer Slide Time: 05:56)

**Sample Source and Header Files in C**

- **Header File: Solver.h:** Includes the header for quadraticEquationSolver() function
- **Source File: Solver.c:** Provides the implementation of quadraticEquationSolver() function
- **Source File: main.c:** Uses quadraticEquationSolver() to solve a quadratic equation

```
// File Solver.h
// User Header files
#ifndef _SOLVER_H // Include Guard. Check
#define _SOLVER_H // Include Guard. Define
int quadraticEquationSolver(
    double, double, double, double*, double*);
#endif // _SOLVER_H // Include Guard. Close

// File Solver.c
// User Implementation files
#include <math.h> // C Std. Library Header
#include "Solver.h" // User Header

int quadraticEquationSolver(
    double a, double b, double c, // I/P Coeff
    double* r1, double* r2) { // O/P Roots
    // Uses double sqrt(double) from math.h
    // ...
    return 0;
}

// File main.c
// Application files
#include <stdio.h> // C Std. Library Header
#include "Solver.h" // User Header

int main() {
    double a, b, c, r1, r2;
    // ...
    // Invoke the solver function from Solver.h
    int status = quadraticEquationSolver(
        a, b, c, &r1, &r2);

    // int printf(char *format, ...) from stdio.h
    printf("Soln. for %dx^2+%dx+%d is %d %d",
        a, b, c, r1, r2);
    // ...
    return 0;
}
```

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

(Refer Slide Time: 06:17)

**Sample Source and Header Files in C++**

- **Header File: Solver.h:** Includes the header for quadraticEquationSolver() function
- **Source File: Solver.cpp:** Provides the implementation of quadraticEquationSolver() function
- **Source File: main.cpp:** Uses quadraticEquationSolver() to solve a quadratic equation

```
// File Solver.h: User Header files
#ifndef _SOLVER_H // Include Guard. Check
#define _SOLVER_H // Include Guard. Define
int quadraticEquationSolver(
    double, double, double, double*, double*);
#endif // _SOLVER_H // Include Guard. Close

// File Solver.cpp: User Implementation files
#include <cmath> // C Std. Lib. Header in C++
using namespace std; // C++ Std. Lib. in std
#include "Solver.h" // User Header

int quadraticEquationSolver(
    double a, double b, double c, // I/P Coeff.
    double* r1, double* r2) { // O/P Roots
    // Uses double sqrt(double) from cmath
    // ...
    return 0;
}

// File main.cpp: Application file
#include <iostream> // C++ Std. Library Header
using namespace std; // C++ Std. Lib. in std
#include "Solver.h" // User Header

int main() {
    double a, b, c, r1, r2;
    // ...
    // Invoke the solver function from Solver.h
    int status = quadraticEquationSolver(
        a, b, c, &r1, &r2);

    // From iostream
    cout<<"Soln. for "<<a<<"x^2+"<<b<<"x+"<<c<<"=0 is ";
    cout<< r1 << r2 << endl;
    // ...
    return 0;
}
```

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

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

(Refer Slide Time: 07:05)

**C Preprocessor (CPP): Managing Source Code**

- The CPP is the macro preprocessor for the C and C++. CPP provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.
- The CPP is driven by a set of directives
  - Preprocessor directives are lines included in the code of programs preceded by a #
  - These lines are not program statements but directives for the preprocessor
  - The CPP examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements
  - The CPP directives have the following characteristics:
    - ▷ CPP directives extend only across a single line of code
    - ▷ As soon as a newline character is found, the preprocessor directive is ends
    - ▷ No semicolon (;) is expected at the end of a preprocessor directive
    - ▷ The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\)

Programming in Modern C++ | Partha Pratim Das | T01.11

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

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

(Refer Slide Time: 07:50)

**C Preprocessor (CPP):**  
Macro definitions: `#define`, `#undef`

- To define preprocessor macros we can use `#define`. Its syntax is:  
`#define identifier replacement` ✓
- This replaces any occurrence of identifier in the rest of the code by replacement. CPP does not understand C/C++, it simply textually replaces  
`#define TABLE_SIZE 100` ✓  
`int table[TABLE_SIZE];`  
`int table2[TABLE_SIZE];`
- After CPP has replaced `TABLE_SIZE`, the code becomes equivalent to:  
`int table[100];`  
`int table2[100];`
- We can define a symbol by `-D name` option from the command line. This predefines `name` as a macro, with definition 1. The following code compiles and outputs 1 when compiled with  
`$ g++ Macros.cpp -D FLAG` ✓  
`#include <iostream> // File Macros.cpp`  
`int main() { std::cout << (FLAG=1) << std::endl; return 0; }`
- **Note that `#define` is important to define constants (like size, pi, etc.), usually in a header (or beginning of a source) and use everywhere. `const` in a variable declaration is a better solution in C++ and C11 onward**

Programming in Modern C++ Partha Pratim Das T01.12

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

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

(Refer Slide Time: 08:47)

**C Preprocessor (CPP):**  
Macro definitions: `#define`, `#undef`

- `#define` can work also with parameters to define function macros:  
`#define getmax(a,b) a>b?a:b`
- This replaces a occurrence of `getmax` followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as a function:  
`// function macro`  
`#include <iostream>`  
`using namespace std;`  
  
`#define getmax(a,b) ((a)>(b)?(a):(b))`  
  
`int main() {`  
`int x = 5, y;`  
`y = getmax(x,2);`  
`cout << y << endl << getmax(7,x) << endl;`  
`return 0;`  
`}`
- **Note that a `#define` function macro can make a small function efficient and usable with different types of parameters. In C++, inline functions & templates achieve this functionality in a better way**

Programming in Modern C++ Partha Pratim Das T01.13

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

(Refer Slide Time: 09:01)

**C Preprocessor (CPP):**  
Macro definitions: `#define`, `#undef`

- Defined macros are not affected by block structure. A macro lasts until it is undefined with the `#undef` preprocessor directive:  

```
#define TABLE_SIZE 100 ✓  
int table1[TABLE_SIZE];  
#undef TABLE_SIZE ✓  
#define TABLE_SIZE 200 ✓  
int table2[TABLE_SIZE];
```
- This would generate the same code as:  

```
int table1[100];  
int table2[200];
```
- We can un-define a symbol by `-U name` option from the command line. This cancels any previous definition of `name`, either built in or provided with a `-D` option  

```
$ g++ file.cpp -U FLAG
```
- **Note that `#undef` is primarily used to ensure that a symbol is not unknowingly being defined and used through some include path**

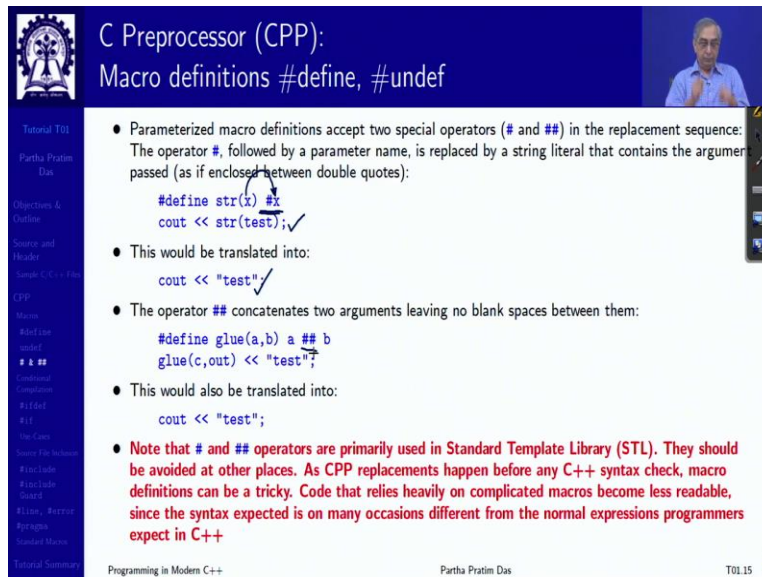
Programming in Modern C++ Partha Pratim Das T01.14

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

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



(Refer Slide Time: 09:35)



**C Preprocessor (CPP):**  
Macro definitions `#define`, `#undef`

- Parameterized macro definitions accept two special operators (`#` and `##`) in the replacement sequence: The operator `#`, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):  

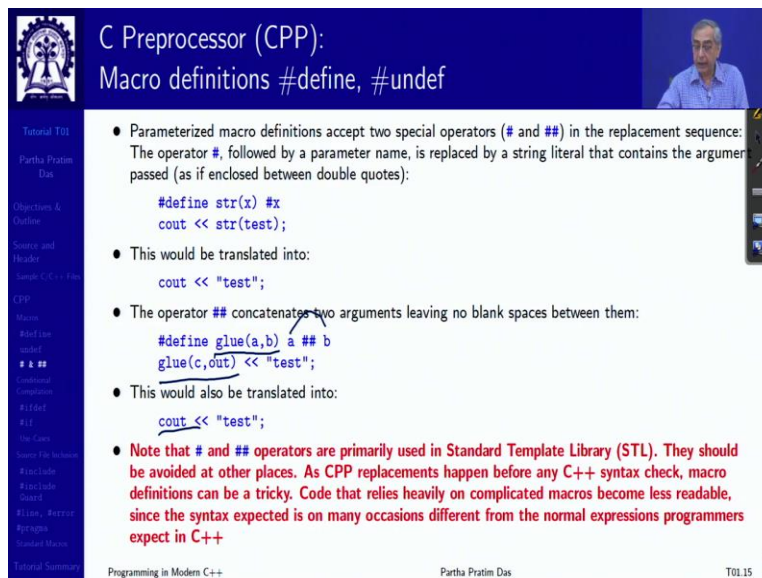
```
#define str(x) #x  
cout << str(test);
```
- This would be translated into:  

```
cout << "test";
```
- The operator `##` concatenates two arguments leaving no blank spaces between them:  

```
#define glue(a,b) a ## b  
glue(c,out) << "test";
```
- This would also be translated into:  

```
cout << "test";
```
- Note that `#` and `##` operators are primarily used in Standard Template Library (STL). They should be avoided at other places. As CPP replacements happen before any C++ syntax check, macro definitions can be a tricky. Code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++**

Programming in Modern C++ Partha Pratim Das T01.15



**C Preprocessor (CPP):**  
Macro definitions `#define`, `#undef`

- Parameterized macro definitions accept two special operators (`#` and `##`) in the replacement sequence: The operator `#`, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):  

```
#define str(x) #x  
cout << str(test);
```
- This would be translated into:  

```
cout << "test";
```
- The operator `##` concatenates two arguments leaving no blank spaces between them:  

```
#define glue(a,b) a ## b  
glue(c,out) << "test";
```
- This would also be translated into:  

```
cout << "test";
```
- Note that `#` and `##` operators are primarily used in Standard Template Library (STL). They should be avoided at other places. As CPP replacements happen before any C++ syntax check, macro definitions can be a tricky. Code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++**

Programming in Modern C++ Partha Pratim Das T01.15

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

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

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

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

(Refer Slide Time: 11:17)

**C Preprocessor (CPP):**  
**Conditional Inclusions: #ifndef, #ifdef, #if, #endif, #**

- These directives allow to include or discard part of the code of a program if a certain condition is met. This is known as **Conditional Inclusion** or **Conditional Compilation**
- **#ifdef** (*if defined*) allows a section of a program to be compiled only if the macro that is specified as the parameter has been **#define**, no matter which its value is. For example:  

```
#ifdef TABLE_SIZE ✓  
int table[TABLE_SIZE]; ✓  
#endif
```

In this case, the line of code `int table[TABLE_SIZE];` is only compiled if `TABLE_SIZE` was previously defined with `#define`, independently of its value. If it was not defined, that line will not be included in the program compilation
- **#ifndef** (*if not defined*) serves for the exact opposite: the code between **#ifndef** and **#endif** directives is only compiled if the specified identifier has not been previously defined. For example:  

```
#ifndef TABLE_SIZE ✓  
#define TABLE_SIZE 100 ✓  
#endif ✓  
int table[TABLE_SIZE]; ←
```

In this case, if when arriving at this piece of code, the `TABLE_SIZE` macro has not been defined yet, it would be defined to a value of `100`. If it already existed it would keep its previous value since the `#define` directive would not be executed.

Programming in Modern C++ Partha Pratim Das T01.16

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

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

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

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

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

(Refer Slide Time: 13:20)

**C Preprocessor (CPP):**  
Conditional Inclusions: `#ifdef`, `#ifndef`, `#if`, `#endif`, `#`

- The `#if`, `#else` and `#elif` (else if) directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` or `#elif` can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#define TABLE_SIZE 50

#else
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```
- Notice how the entire structure of `#if`, `#elif` and `#else` chained directives ends with `#endif`
- The behavior of `#ifdef` and `#ifndef` can also be achieved by using the special operators `defined` and `!defined` (not defined) respectively in any `#if` or `#elif` directive:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

Programming in Modern C++ | Partha Pratim Das | T01.17

**C Preprocessor (CPP):**  
Conditional Inclusions: `#ifdef`, `#ifndef`, `#if`, `#endif`, `#else` & `#e`

- The `#if`, `#else` and `#elif` (else if) directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` or `#elif` can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#define TABLE_SIZE 50

#else
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```
- Notice how the entire structure of `#if`, `#elif` and `#else` chained directives ends with `#endif`
- The behavior of `#ifdef` and `#ifndef` can also be achieved by using the special operators `defined` and `!defined` (not defined) respectively in any `#if` or `#elif` directive:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

Programming in Modern C++ | Partha Pratim Das | T01.17

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

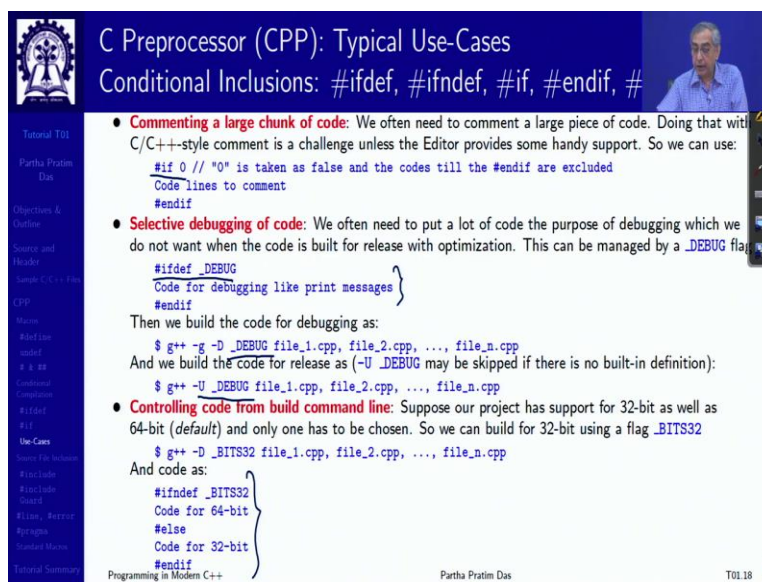
than 200, then this will succeed, then what it will do it will undef. So, you are what you are doing, you are reducing the TABLE\_SIZE.

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

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

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

(Refer Slide Time: 16:09)



The image shows a presentation slide titled "C Preprocessor (CPP): Typical Use-Cases" with a sidebar on the left and a video inset on the right. The slide content is as follows:

### C Preprocessor (CPP): Typical Use-Cases

#### Conditional Inclusions: #ifdef, #ifndef, #if, #endif, #

- **Commenting a large chunk of code:** We often need to comment a large piece of code. Doing that with C/C++-style comment is a challenge unless the Editor provides some handy support. So we can use:  

```
#if 0 // "0" is taken as false and the codes till the #endif are excluded  
Code lines to comment  
#endif
```
- **Selective debugging of code:** We often need to put a lot of code the purpose of debugging which we do not want when the code is built for release with optimization. This can be managed by a `_DEBUG` flag.  

```
#ifdef _DEBUG  
Code for debugging like print messages  
#endif
```

Then we build the code for debugging as:

```
$ g++ -g -D _DEBUG file_1.cpp, file_2.cpp, ..., file_n.cpp
```

And we build the code for release as (`-U _DEBUG` may be skipped if there is no built-in definition):

```
$ g++ -U _DEBUG file_1.cpp, file_2.cpp, ..., file_n.cpp
```
- **Controlling code from build command line:** Suppose our project has support for 32-bit as well as 64-bit (default) and only one has to be chosen. So we can build for 32-bit using a flag `_BITS32`  

```
$ g++ -D _BITS32 file_1.cpp, file_2.cpp, ..., file_n.cpp
```

And code as:

```
#ifndef _BITS32  
Code for 64-bit  
#else  
Code for 32-bit  
#endif
```

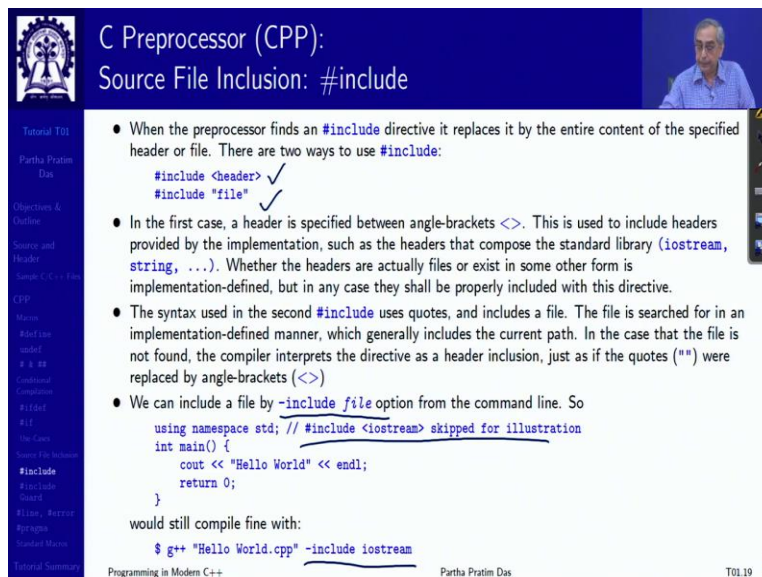
Programming in Modern C++ Partha Pratim Das T01.18

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

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

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

(Refer Slide Time: 17:25)



**C Preprocessor (CPP):**  
**Source File Inclusion: #include**

- When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified header or file. There are two ways to use `#include`:  
`#include <header>` ✓  
`#include "file"` ✓
- In the first case, a header is specified between angle-brackets `<>`. This is used to include headers provided by the implementation, such as the headers that compose the standard library (`iostream`, `string`, ...). Whether the headers are actually files or exist in some other form is implementation-defined, but in any case they shall be properly included with this directive.
- The syntax used in the second `#include` uses quotes, and includes a file. The file is searched for in an implementation-defined manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes ("`\"`") were replaced by angle-brackets (`<>`)
- We can include a file by `-include file` option from the command line. So  

```
using namespace std; // #include <iostream> skipped for illustration
int main() {
    cout << "Hello World" << endl;
    return 0;
}
```

would still compile fine with:  

```
$ g++ "Hello World.cpp" -include iostream
```

Programming in Modern C++ Partha Pratim Das T01.19

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

(Refer Slide Time: 18:11)

**C Preprocessor (CPP):**  
**Source File Inclusion: #include Guard**

- Inclusions of header files may lead to the problems of Multiple Inclusion and / or Circular Inclusion
- An #include guard, sometimes called a macro guard, header guard or file guard, is a particular construct used to avoid the problem of double inclusion when dealing with the include directive
- **Multiple Inclusion:** Consider the following files:

Without Guard	With Guard
<pre>// File "grandparent.h" struct foo { int member; };  // File "parent.h" #include "grandparent.h"  // File "child.c" #include "grandparent.h" #include "parent.h"  // Expanded "child.c": WRONG // Duplicate definition struct foo { int member; }; struct foo { int member; };</pre>	<pre>// File "grandparent.h" #ifndef GRANDPARENT_H // undefined first time #define GRANDPARENT_H // Defined for the first time struct foo { int member; }; #endif /* GRANDPARENT_H */  // File "parent.h" #ifndef PARENT_H // undefined first time #define PARENT_H // Defined for the first time #include "grandparent.h" #endif /* PARENT_H */  // File "child.c" #include "grandparent.h" #include "parent.h"  // Expanded "child.c": RIGHT: Only one definition struct foo { int member; };</pre>

Programming in Modern C++  
Partha Pratim Das  
T01.20

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

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

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


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

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


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

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

(Refer Slide Time: 21:56)



## C Preprocessor (CPP): Source File Inclusion: #include Guard



• **Circular Inclusion:** Consider the following files:

Without Guard	With Guard
<pre> o Class Flight: Needs the info of service provider o Class Service: Needs the info of flights it offers  #include&lt;iostream&gt; // File main.h #include&lt;vector&gt; using namespace std; #include "main.h" // File Service.h #include "Flight.h" class Flight; class Service { vector&lt;Flight*&gt; m_Flt; /* ... */ }; #include "main.h" // File Flight.h #include "Service.h" class Service; class Flight { Service* m_pServ; /* ... */ }; #include "main.h" // File main.cpp #include "Service.h" #include "Flight.h" int main() { /* ... */ return 0; };                     </pre>	<pre> #include&lt;iostream&gt; // File main.h #include&lt;vector&gt; using namespace std; #ifndef __SERVICE_H #define __SERVICE_H #include "main.h" // File Service.h #include "Flight.h" class Flight; class Service { vector&lt;Flight*&gt; m_Flt; /* ... */ }; #endif // __SERVICE_H #ifndef __FLIGHT_H #define __FLIGHT_H #include "main.h" // File Flight.h #include "Service.h" class Service; class Flight { Service* m_pServ; /* ... */ }; #endif // __FLIGHT_H #include "main.h" // File main.cpp #include "Service.h" #include "Flight.h" int main() { /* ... */ return 0; };                     </pre>
<p>o <b>Class Flight and Class Service has cross-references</b>  o <b>Hence, circular inclusion of header files lead to infinite loop during compilation</b></p>	

Programming in Modern C++
Partha Pratim Das
T01.21

## C Preprocessor (CPP): Source File Inclusion: #include Guard

• Circular Inclusion: Consider the following files:

Without Guard	With Guard
<pre> o Class Flight: Needs the info of service provider o Class Service: Needs the info of flights it offers  #include&lt;iostream&gt; // File main.h #include&lt;vector&gt; using namespace std; #include "main.h" // File Service.h #include "Flight.h" // File Flight.h class Flight; class Service { vector&lt;Flight*&gt; m_Flt; /* ... */ }; #include "main.h" // File Flight.h #include "Service.h" // File Service.h class Service; class Flight { Service* m_pServ; /* ... */ }; #include "main.h" // File main.cpp #include "Service.h" #include "Flight.h" int main() { /* ... */ return 0; }; </pre>	<pre> #include&lt;iostream&gt; // File main.h #include&lt;vector&gt; using namespace std; #ifndef __SERVICE_H #define __SERVICE_H #include "main.h" // File Service.h #include "Flight.h" class Flight; class Service { vector&lt;Flight*&gt; m_Flt; /* ... */ }; #endif // __SERVICE_H #ifndef __FLIGHT_H #define __FLIGHT_H #include "main.h" // File Flight.h #include "Service.h" class Service; class Flight { Service* m_pServ; /* ... */ }; #endif // __FLIGHT_H #include "main.h" // File main.cpp #include "Service.h" #include "Flight.h" int main() { /* ... */ return 0; }; </pre>

o Class Flight and Class Service has cross-references  
o Hence, circular inclusion of header files lead to infinite loop during compilation

Programming in Modern C++ Partha Pratim Das T01.21

## C Preprocessor (CPP): Source File Inclusion: #include Guard

• Circular Inclusion: Consider the following files:

Without Guard	With Guard
<pre> o Class Flight: Needs the info of service provider o Class Service: Needs the info of flights it offers  #include&lt;iostream&gt; // File main.h #include&lt;vector&gt; using namespace std; #include "main.h" // File Service.h #include "Flight.h" // File Flight.h class Flight; class Service { vector&lt;Flight*&gt; m_Flt; /* ... */ }; #include "main.h" // File Flight.h #include "Service.h" // File Service.h class Service; class Flight { Service* m_pServ; /* ... */ }; #include "main.h" // File main.cpp #include "Service.h" #include "Flight.h" int main() { /* ... */ return 0; }; </pre>	<pre> #include&lt;iostream&gt; // File main.h #include&lt;vector&gt; using namespace std; #ifndef __SERVICE_H #define __SERVICE_H #include "main.h" // File Service.h #include "Flight.h" class Flight; class Service { vector&lt;Flight*&gt; m_Flt; /* ... */ }; #endif // __SERVICE_H #ifndef __FLIGHT_H #define __FLIGHT_H #include "main.h" // File Flight.h #include "Service.h" class Service; class Flight { Service* m_pServ; /* ... */ }; #endif // __FLIGHT_H #include "main.h" // File main.cpp #include "Service.h" #include "Flight.h" int main() { /* ... */ return 0; }; </pre>

o Class Flight and Class Service has cross-references  
o Hence, circular inclusion of header files lead to infinite loop during compilation

Programming in Modern C++ Partha Pratim Das T01.21

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

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

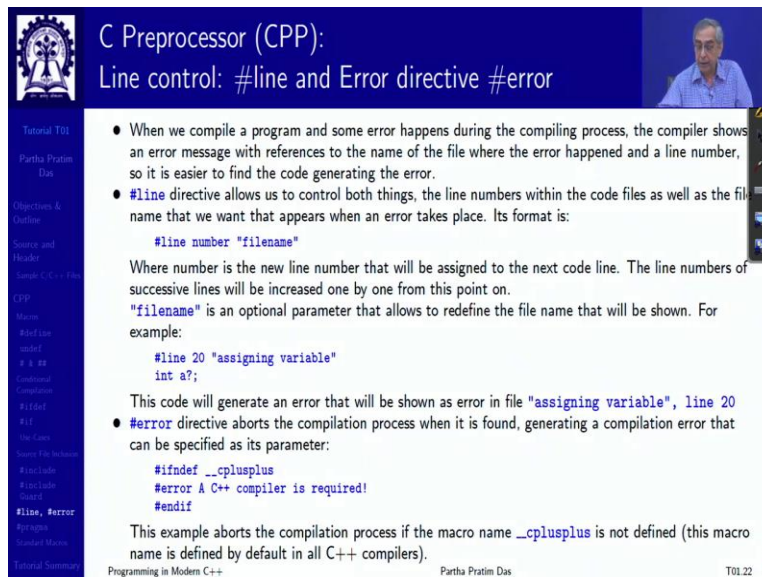


to include Flight.h. So, you come include this code, you try to include Service.h, you go back try to include Flight.h come back try to include Service.h indefinite circular inclusions.

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

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

(Refer Slide Time: 24:50)



**C Preprocessor (CPP):**  
**Line control: #line and Error directive #error**

- When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.
- **#line** directive allows us to control both things, the line numbers within the code files as well as the filename that we want that appears when an error takes place. Its format is:  

```
#line number "filename"
```

Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.  
"filename" is an optional parameter that allows to redefine the file name that will be shown. For example:

```
#line 20 "assigning variable"  
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 20
- **#error** directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter:  

```
#ifndef __cplusplus  
#error A C++ compiler is required!  
#endif
```

This example aborts the compilation process if the macro name `__cplusplus` is not defined (this macro name is defined by default in all C++ compilers).

Programming in Modern C++ Partha Pratim Das T01.22

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

(Refer Slide Time: 25:24)

**C Preprocessor (CPP):**  
**Pragma directive: #pragma**

- This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with **#pragma**.
- If the compiler does not support a specific argument for **#pragma**, it is ignored - no syntax error is generated.
- Many compilers, including GCC, supports **#pragma once** which can be used as **#include guard**. So

```
#ifndef __FLIGHT_H
#define __FLIGHT_H
#include "main.h" // File Flight.h
#include "Service.h"
class Service;
class Flight { Service* m_pServ; /* ... */ };
#endif // __FLIGHT_H
```

can also be written as:

```
#pragma once
#include "main.h" // File Flight.h
#include "Service.h"
class Service;
class Flight { Service* m_pServ; /* ... */ };
```

**This is cleaner, but may have portability issue across machines and compilers**

Programming in Modern C++ Partha Pratim Das T01.23

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

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

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

(Refer Slide Time: 27:13)

**C Preprocessor (CPP):  
Predefined Macro Names**

- The following macro names are always defined (they begin and end with two underscore characters, -)

Macro	Value
<code>__LINE__</code>	Integer value representing the current line in the source code file being compiled
<code>__FILE__</code>	A string literal containing the presumed name of the source file being compiled
<code>__DATE__</code>	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began
<code>__TIME__</code>	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began
<code>__cplusplus</code>	An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler: <ul style="list-style-type: none"><li>199711L: ISO C++ 1998/2003</li><li>201103L: ISO C++ 2011</li></ul> Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above
<code>__STDC_HOSTED__</code>	1 if the implementation is a hosted implementation (with all standard headers available) 0 otherwise

Programming in Modern C++ Partha Pratim Das T01.24

**C Preprocessor (CPP):  
Predefined Macro Names**

- The following macros are optionally defined, generally depending on whether a feature is available:

Macro	Value
<code>__STDC__</code>	In C: if defined to 1, the implementation conforms to the C standard. In C++: Implementation defined
<code>__STDC_VERSION__</code>	In C: <ul style="list-style-type: none"><li>199401L: ISO C 1990, Amendment 1</li><li>199901L: ISO C 1999</li><li>201112L: ISO C 2011</li></ul> In C++: Implementation defined
<code>__STDC_MB_MIGHT_NEQ_WC__</code>	1 if multibyte encoding might give a character a different value in character literals
<code>__STDC_ISO_10646__</code>	A value in the form yyyyMM, specifying the date of the Unicode standard followed by the encoding of wchar_t characters
<code>__STDCPP_STRICT_POINTER_SAFETY__</code>	1 if the implementation has strict pointer safety (see <code>get_pointer_safety</code> )
<code>__STDCPP_THREADS__</code>	1 if the program can have more than one thread

- Macros marked in blue are frequently used

Programming in Modern C++ Partha Pratim Das T01.25

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

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

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

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

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

(Refer Slide Time: 29:44)

**C Preprocessor (CPP): Standard Macro Examples**

- Consider:

```
// standard macro names
#include <iostream>
using namespace std;

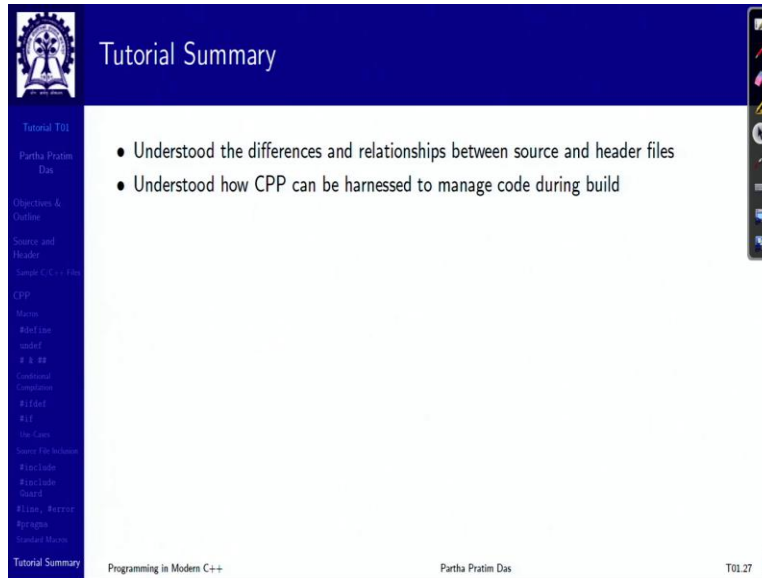
int main()
{
    cout << "This is the line number " << __LINE__;
    cout << " of file " << __FILE__ << ".\n";
    cout << "Its compilation began " << __DATE__;
    cout << " at " << __TIME__ << ".\n";
    cout << "The compiler gives a __cplusplus value of " << __cplusplus;
    return 0;
}
```
- The output is:

```
This is the line number 7 of file Macros.c
Its compilation began Sep 13 2021 at 11:30:07
The compiler gives a __cplusplus value of 201402
```
- Note that `__LINE__`, `__FILE__`, `__DATE__`, and `__TIME__` important for details in error reporting

Programming in Modern C++ | Partha Pratim Das | T01.26

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

(Refer Slide Time: 30:31)



The screenshot shows a presentation slide with a dark blue header and footer. The header contains the text 'Tutorial Summary' in white. The main content area is white and contains two bullet points: '• Understood the differences and relationships between source and header files' and '• Understood how CPP can be harnesses to manage code during build'. On the left side, there is a vertical navigation menu with various items like 'Tutorial T01', 'Partha Pratim Das', 'Objectives & Outline', 'Source and Header', 'CPP', 'Makefile', 'CMake', 'Project', 'Build', 'The Code', 'System File Structure', 'Makefile', 'CMake', 'Project', 'Build', 'Project', 'Build', 'Project', 'Build'. At the bottom, the footer contains 'Programming in Modern C++', 'Partha Pratim Das', and 'T01.27'. There is also a small logo in the top left corner and a toolbar in the top right corner.

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

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

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