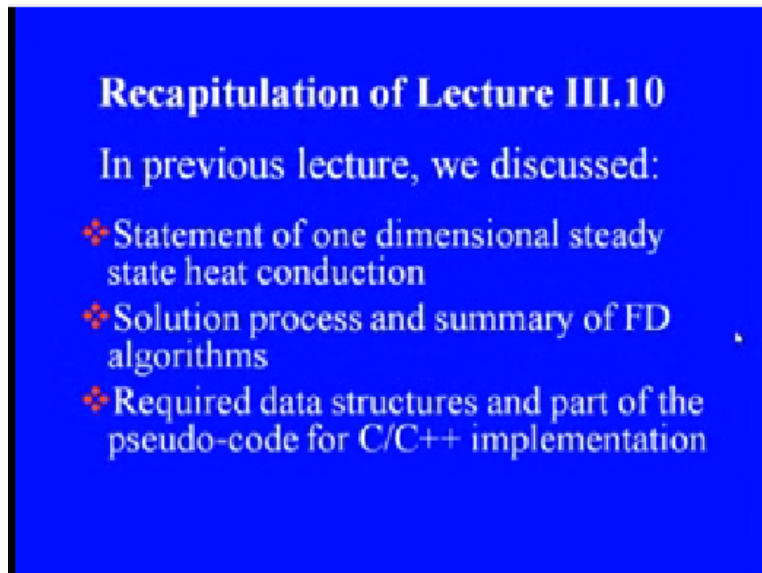


**Computational Fluid Dynamics**  
**Dr. Krishna M. Singh**  
**Department of Mechanical and Industrial Engineering**  
**Indian Institute of Technology - Roorkee**

**Lecture – 20**  
**Computer Implementation of FDM for Steady State Heat Diffusion Problems - 2**

Welcome back to our last lecture in module 3 on finite difference method. Today's lecture is in continuation of what we discussed yesterday regarding computer implementation for 1-dimensional heat conduction. So we will proceed further and have a look at the actual implementation in a standard development environment of the pseudo-code part of it which I wrote yesterday.

**(Refer Slide Time: 00:52)**

A blue rectangular slide with white text. The title is 'Recapitulation of Lecture III.10'. Below it, the text says 'In previous lecture, we discussed:'. This is followed by a bulleted list with three items, each preceded by a red diamond symbol. The items are: 'Statement of one dimensional steady state heat conduction', 'Solution process and summary of FD algorithms', and 'Required data structures and part of the pseudo-code for C/C++ implementation'.

**Recapitulation of Lecture III.10**

In previous lecture, we discussed:

- ❖ Statement of one dimensional steady state heat conduction
- ❖ Solution process and summary of FD algorithms
- ❖ Required data structures and part of the pseudo-code for C/C++ implementation

So let us recap of what we did yesterday. We restated our heat conduction equation. We had a brief review of the solution process and summary of finite difference algorithms we are going to use for solving this particular problem and then we discussed what are required data structures and a part of the pseudo-code for C and C++ implementation. So today's lecture is continuation of yesterday's last lecture on computer implementation on finite difference method for steady state heat conduction problems.

**(Refer Slide Time: 01:29)**

## LECTURE OUTLINE

- ❖ Statement of one dimensional steady state heat conduction
- ❖ Required data structures and pseudo-code for C/C++ implementation
- ❖ C++ Implementation (using DevC++ IDE)

So we will have to in a brief look at what we did yesterday before we can continue further. So the statement of the problem required data structures and then we will have a detailed look at C++ implementation in a free development environment called Developer C++ IDE. So this was our problem statement.

(Refer Slide Time: 01:52)

## One Dimensional Heat Conduction

Steady state heat conduction in a slab of width  $L$  with thermal conductivity  $k$  and heat generation is governed by

$$k \frac{\partial^2 T}{\partial x^2} + q_g = 0$$

Boundary conditions at either end could be

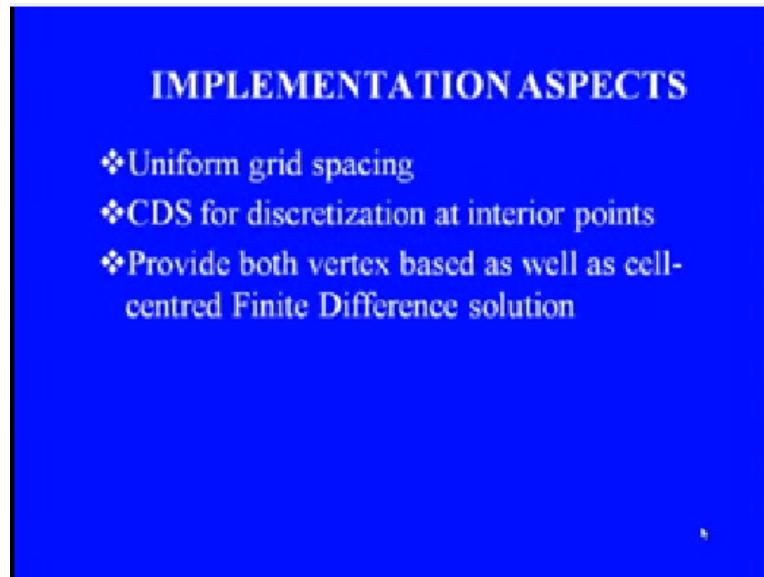
Dirichlet (specified temperature):  $T = \bar{T}$

Neumann (specified flux):  $q = \bar{q}$

Convection:  $q| = h(T - T_a)$

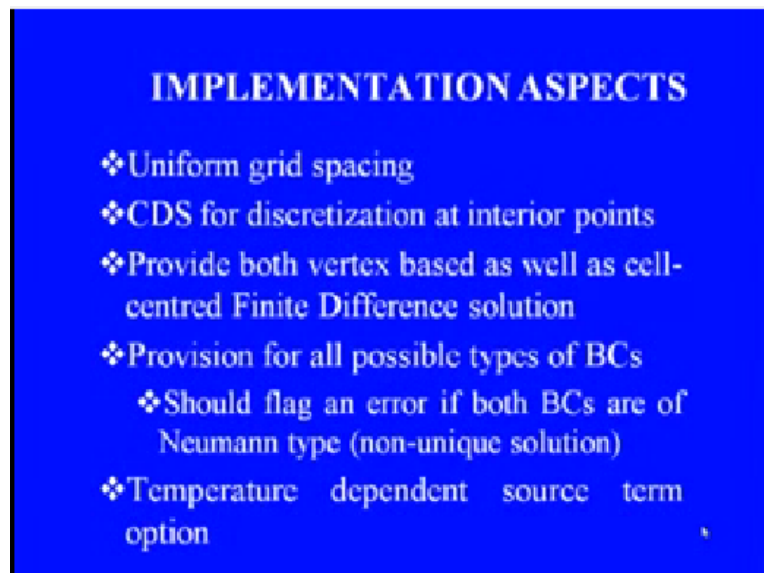
We are dealing with a 1-D heat conduction problem, with heat generation and constant thermal conductivity and we said our codes would provide for any combination of value condition that is we can have specified temperature boundary conditions, Neumann boundary condition, or convective boundary condition at either end with some restriction or specification of both end Neumann boundary condition which we should be aware of.

(Refer Slide Time: 02:22)



Once again our implementation what we assume uniform grid spacing and CDS for discretization and we should ideally provide for both vertex based as well as cell-centred Finite Difference solution algorithm in our code.

(Refer Slide Time: 02:34)



And provision for all types of boundary conditions, our source term would be temperature dependent, that is what we had decided.

(Refer Slide Time: 02:42)

## CODE DESIGN

Design separate modules (functions) for

- ❖ Input of geometry and boundary conditions (input)
- ❖ Generation of system matrix and implementation of BCs (systemMatrix)
- ❖ Solution of algebraic system (solve)
- ❖ Output of solution (output)

And we said we are going to break our code into different modules for simple implementation. It is not just for the simplicity of implementations. It should also help us in testing each model separately before we can combine or synthesise in our final code. So we said we are going to develop a set of modules, one model for geometry and boundary conditions and then module for generation of the system matrix and we will have one module for solution of algebraic systems.

And then one separate module model for output of boundary conditions and then we said okay, let us come up with required data structures, okay.

**(Refer Slide Time: 03:28)**

## REQUIRED DATA STRUCTURES

- ❖ Decision based on input and output parameters, and
- ❖ Simple interface for different modules called by main().
  - ❖ Grid
  - ❖ Matrix
  - ❖ BCType
  - ❖ Source

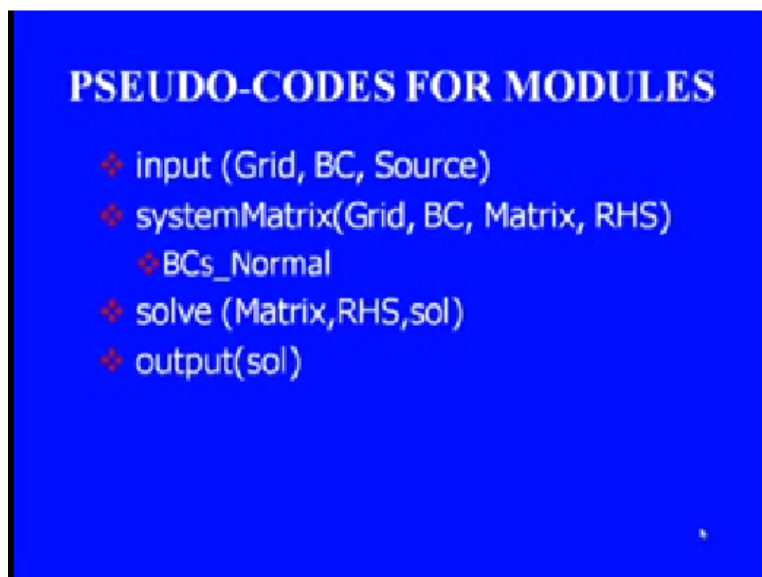
And our decision was based on for input-output parameters we require and the simplicity of

interface between different modules, okay. So we decided upon few data restructures which had look grid, this is the one which is going to contain all the geometric information of the grid, the endpoints, the grid spacing, the number of divisions, the number of computational nodes and material properties that some other conductivity included there.

Then we will introduce one more data structure which we called matrix which should have the details about the discrete algebraic system which we get from finite difference discretization and for boundary conditions, we will have different types of boundary conditions we might have at 2 ends.

So let us have a new datatype to hold this boundary condition information, the type of boundary condition and the pertinent data and similarly we achieved our source term to be temperature dependent and primarily let us take a case of linear dependence and we should support it by a separate data structure.

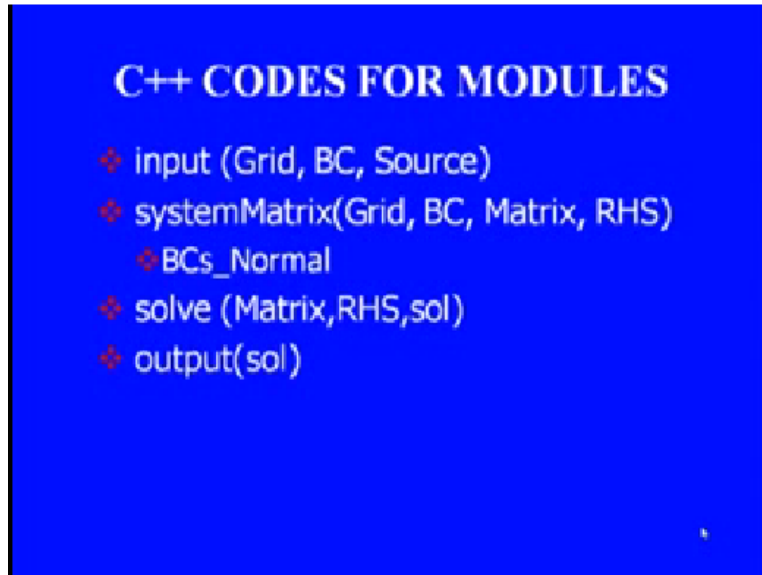
**(Refer Slide Time: 04:35)**



And then we discussed our pseudo-codes for some of the modules of how what will be the look, what should our input module do for so the systemMatrix module do and in systemMatrix module which would provide 2 submodules, one which takes care of the implementation of boundary conditions in case we choose our normal finite difference approximation that is our vertex based approximations.

And we should provide for a separate module or submodule for systemMatrix which can implement boundary conditions for the cell-centred approach. And then one simple solution module, an output module which should output the data in the modular format.

**(Refer Slide Time: 05:24)**



So now today, we are going to have a look at the code implementation of these modules starting from the pseudo-code which we discussed in the previous lecture for the input systemMatrix and we will first take up the case of our normal implementation of normal finite difference approximation and how do we implement the boundary conditions. Cell-centred approximations, we are going to take up as a refinement to the code which we develop later on.

Then we will assume for the time being that we have got to solve the routine available to us based on tridiagonal matrix algorithm. We will not discuss this in detail and then we will discuss our output routine. Okay, now let us have a look at the implementation aspects.

**(Refer Slide Time: 06:06)**

### Code Implementation

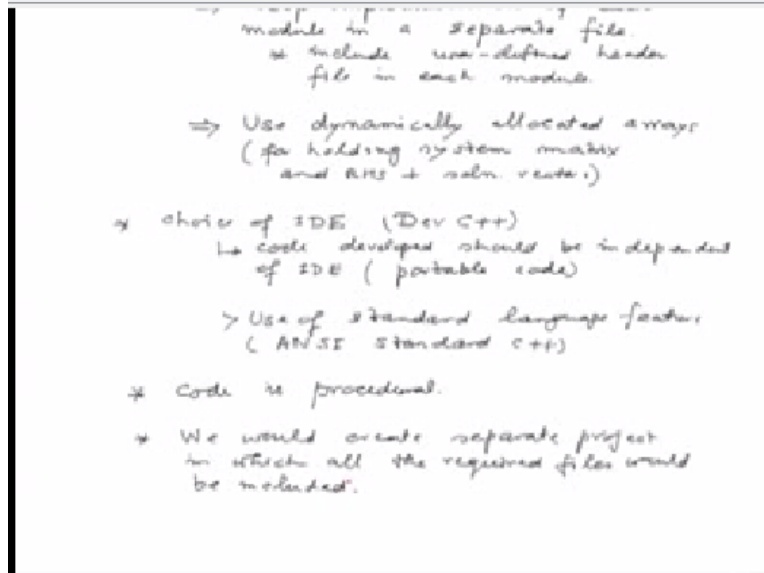
- \* Separate the interface from implementation
  - ⇒ Collect all data structure definitions and function prototypes in a user-defined header file
  - ⇒ Keep implementation of each module in a separate file.
    - \* Include user-defined header file in each module.
  - ⇒ Use dynamically allocated arrays (for holding system matrix and RHS + soln vector)

So our code implementation, what we will do is we are going to separate what is free float in programming parlances, separate the interface from implementation. So for this purpose, what we will do is, we are going to collect all data structure definitions and function prototypes in a user-defined header file.

So that is one thing which we will do and the second decision we should make is, let us keep the implementation of each module in a separate file, okay and in each of these separate files, we need to incorporate or we need to include user-defined header file in each module. The next programming decision which we will take is, we are going to use variable size arrays or dynamically allocated arrays for holding of systemMatrix and RHS plus solution vectors.

So we are not going to just limit the size of array statically, it will not be known as compiled time, it would be allocated based on the number of grid nodes we would actually specify it. So we are dealing with the multiple files on Windows operating system. We will choose what we call a project mode, so what choice of IDE you can work with any integrated development environment available to you. I have chosen here developer C++ which is freely available but the code which we develop, code developed, should be independent of your IDE.

**(Refer Slide Time: 10:34)**



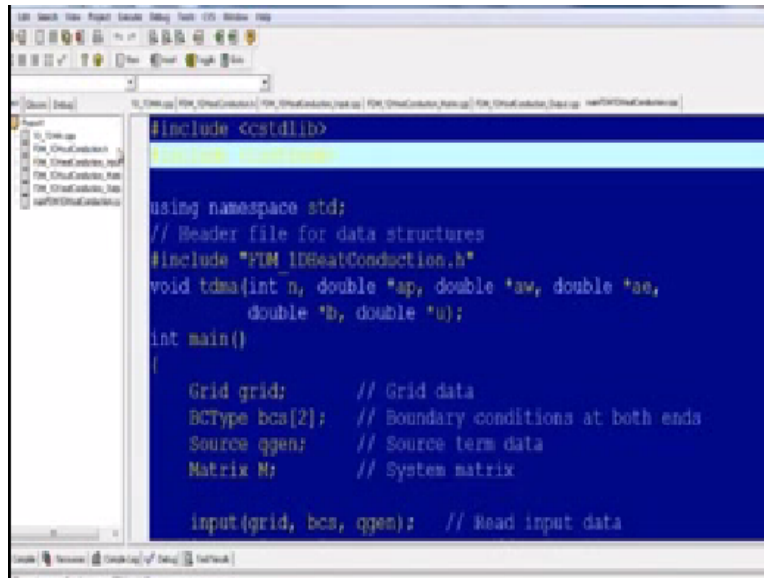
This is what we refer to as portable code. So the set of files which have developed here, you can use it in any development environment, we choose Borland C++ or Visual Studio Developer C++ code block or anything. You can take it on even Linux machine and you can use GCC compiler there by making use of a make file, or just compiling all the files together and should work equally well. So that is why we are not going to use any IDE specific features.

We will restrict ourselves to use of standard language features, in particular we will restrict ourselves to ANSI standard C++. For the sake of simplicity, I have chosen C++ but this code could be slightly modified to work entirely with a C compiler. We are not using any object-oriented feature or code is entirely procedural.

Okay, since we have broken our code into different files, so on Windows operating system and in our development environment, we would create a separate project in which all the required files would be included and this would facilitate our job of compilation of the code and is debugging and running of the code. So now, let us move on, let us move to Developer C++ environment.

**(Refer Slide Time: 13:06)**



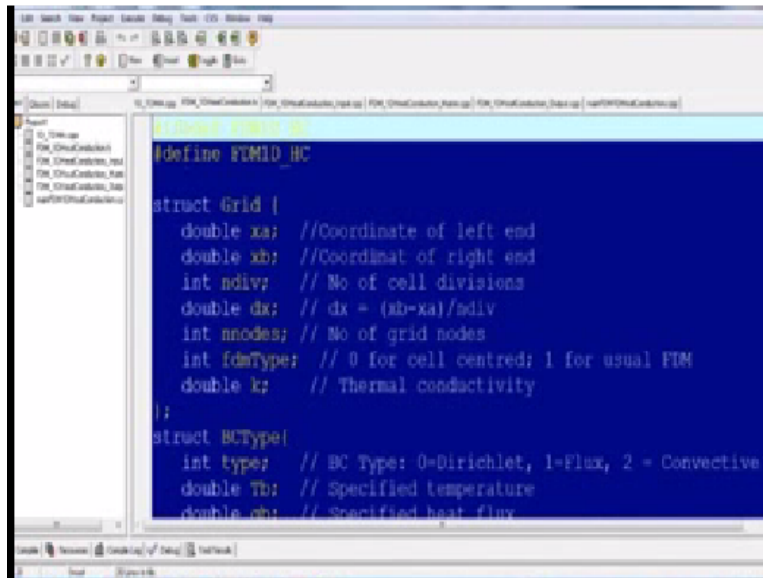


Let us look at some simple features, if you want to start of a fresh, open your develop environment, go to your file menu and chose a project and normally you will be given plenty of options, for instance in Developer C++ environment, yes, you would prefer Windows application, console application, static library, DLL and so on and so forth.

On Microsoft visual studio, it might be slightly different setup options but user choose, this console application option would be there, so chose this console application option, chose your project to be C++ project, chose the name, whatever name you want to. For instance, for our application, we have chosen a project with its name, this is called Project 1 and there are various files which are included as a part of this project.

Okay, now first let us have a look at our data structure implementations.

**(Refer Slide Time: 14:13)**



```
#ifndef FDM1D_HC
#define FDM1D_HC

struct Grid {
    double xa; //Coordinate of left end
    double xb; //Coordinate of right end
    int ndiv; // No of cell divisions
    double dx; // dx = (xb-xa)/ndiv
    int nnodes; // No of grid nodes
    int fdmType; // 0 for cell centred; 1 for usual FDM
    double k; // Thermal conductivity
};

struct BCType {
    int type; // BC Type: 0=Dirichlet, 1=Flux, 2 = Convective
    double Tb; // Specified temperature
    double qh; // Specified heat flux
};

#endif
```

So we have chosen a header file which we called FDM1D, FDM1D heatconduction.H, this is our user-defined header file and this is how the setup file looks like. Whenever we define the header file, it is always advisable to include all the set of definitions within this if and if block. So if and if detail is constant FDM1D\_HC.

You could choose whatever name you want to choose for this constant, okay. Normally we will choose the constant depending on our nature of the project. For instance, we are dealing with finite difference method solution for 1-dimensional heat conduction, so I have chosen the acronym very similar to that, FDM1D\_HC, HC stands for heat conduction. So if and if, if not defined then define this constant FDM1D\_HC.

Now this if and if end has defined these 2 statements will help us avoiding multiple inclusion of the contents of this file whenever we include the setup file. So that is why we require that. Now let us have a look at each structure or rather each data structure which we said we are going to define. The first one is grid.

We have used the keyword struct to make sure that all the elements or all the members are public members, we could have actually used class if we wanted in place of struct in C++ language, but these set of definitions would work equally well in C language as well. So that is the reason I have used this keyword struct to define new datatype.

So struct Grid following the convention that the first letter of a data type would be capitalized. Now as we discussed yesterday what data which we need which would be part of this struct Grid datatype. So grid should have the 2 endpoints of a domain. So therefore in double xa, double xb and we will provide comments here, coordinate xa refers to coordinate of left end. Double xb, this refers to coordinate of the right hand. There are number of subdivisions or number of cell divisions in our finite difference discretization. So we used an integer parameter ndiv.

We have assumed uniform grid spacing, so we can have only a single double value. Let us call this as dx, double dx, so  $dx = (xb - xa) / ndiv$ , that is how we are going to set it in our input routine. The number of computational nodes that will depend on our formulation, it would be one more the number of divisions in the case of usual finite difference formulation and would be equal to number of divisions in the case of cell-centred formulation.

And that FDM formulation, we will distinguish or we will flag off using our integer flag fdmType. Now this fdmType, could be 0 for cell-centred finite difference scheme and one for our usual finite difference approximation. We have also included in this grid data structure our thermal conductivities. so the only material property which was involved in our steady-state heat conduction problem. So for the sake of simplicity, let us include this also in our grid data structure.

Next, we have our BCType to hold the boundary condition information. First member would be an integer type, so BC Type 0 would stand for Dirichlet boundary condition, that is temperature is specified; if user enters 1, that represents a flux and 2 stands for convective boundary condition. So we have put all the required comments or documentation right here that what this particular variable stands for, what would be specific use.

**(Refer Slide Time: 18:40)**

```

int ndiv; // No of cell divisions
double dx; // dx = (xb-xa)/ndiv
int nnodes; // No of grid nodes
int fctType; // 0 for cell centred; 1 for usual FDM
double k; // Thermal conductivity
};

struct BCType{
    int type; // BC Type: 0=Dirichlet, 1=Flux, 2 = Convective
    double Tb; // Specified temperature
    double qb; // Specified heat flux
    double h; // Convective heat transfer coeff.
    double Ta; // Ambient temperature
};

struct Matrix {

```

Next, we would provide for the required data for boundary conditions. For instance, Tb, Tb would stand for the specified temperature; qb for specified heat flux; h for convective heat transfer coefficient and Ta for ambient temperature in case of a convective boundary condition and we will set these things appropriately in our input module.

**(Refer Slide Time: 19:09)**

```

double Tb; // Specified temperature
double qb; // Specified heat flux
double h; // Convective heat transfer coeff.
double Ta; // Ambient temperature
};

struct Matrix {
    double *AE; // pointers for FDM array AE[]
    double *AP; // pointers for FDM array AP[]
    double *AW; // pointers for FDM array AW[]
    double *B; // RHS vector in [A][T] = [B]
    double *T; // Solution vector
};

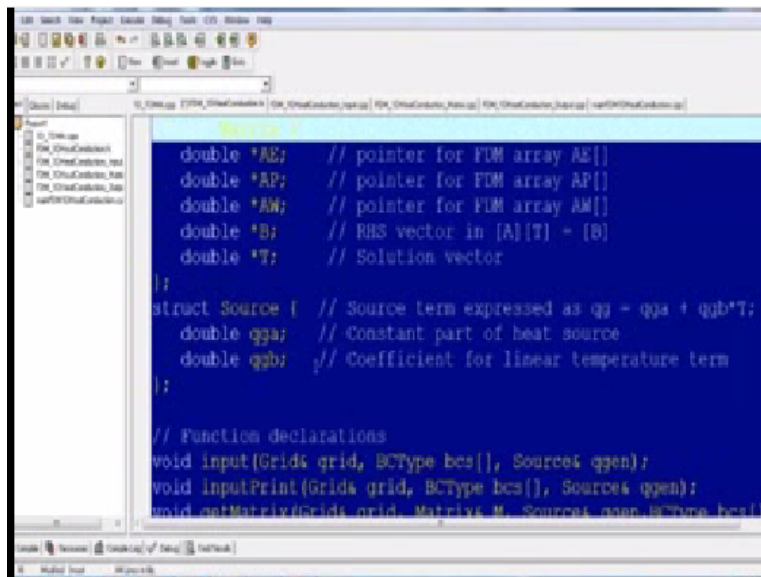
struct Source { // Source term expressed as qg = qga + qgb*T;
    double qga; // Constant part of heat source
    double qgb; // Coefficient for linear temperature term
};

```

Next is our discrete algebraic system. We need a data structure for that and remember our matrix was a tridiagonal matrix. So we need just to free diagonal elements plus our unknown vector and lower vector, these were the arrays which were involved in the definition of a matrix. So struct matrix, the first element at double \*AE which is pointer for FDM array AE. Double \*AP pointer for that FDM array.

AP that is to say our AP stands for the main diagonal and AQ stands for our lower diagonal. Next, double star B, this stands for, this provides, this star is location for right-hand side vector in the system algebraic system  $AT=B$  and T is our solution vector for temperature. Now all these arrays that we have pointers which would be linked to dynamically created arrays of appropriate size.

**(Refer Slide Time: 20:30)**



```
double *AE; // pointer for FDM array AE[]
double *AP; // pointer for FDM array AP[]
double *AQ; // pointer for FDM array AQ[]
double *B; // RHS vector in [A][T] = [B]
double *T; // Solution vector

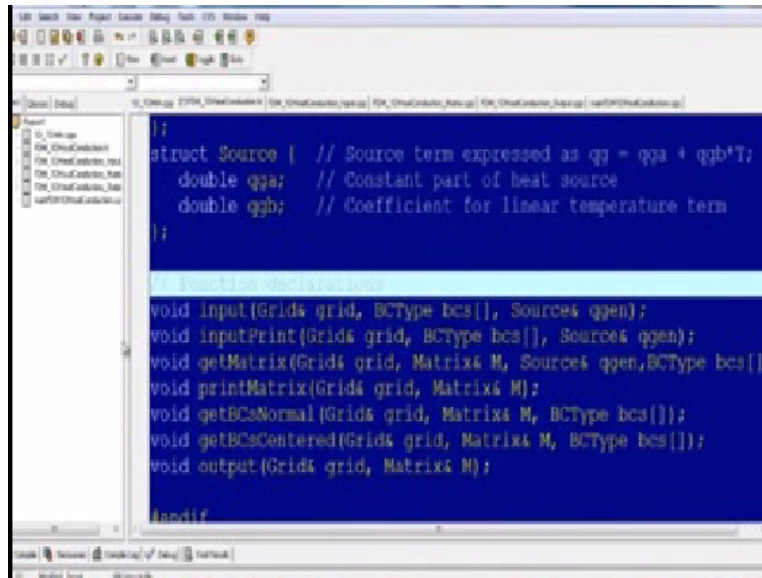
};

struct Source { // Source term expressed as qg = qga + qgb*T;
    double qga; // Constant part of heat source
    double qgb; // Coefficient for linear temperature term
};

// Function declarations
void input(Grids grid, BCType bcs[], Source* qgen);
void inputPrint(Grids grid, BCType bcs[], Source* qgen);
void getMatrix(Grids grid, Matrix* M, Source* qgen, BCType bcs[]);
```

Similarly, we need a data structure for source too. So source term usually a linear variation or a source which depends linearly on temperature, we can have linear variation,  $qg=qga+qgb$ . So this particular data structure would store these 2 coefficients qga and qgb. So struct source double qga, double qgb.

**(Refer Slide Time: 21:00)**



```
//  
struct Source { // Source term expressed as  $q_g = q_{ga} + q_{gb}T$ ;  
    double qga; // Constant part of heat source  
    double qgb; // Coefficient for linear temperature term  
};  
  
// Function declarations  
void input(Grid& grid, BCType bcs[], Source& qgen);  
void inputPrint(Grid& grid, BCType bcs[], Source& qgen);  
void getMatrix(Grid& grid, Matrix& M, Source& qgen, BCType bcs[]);  
void printMatrix(Grid& grid, Matrix& M);  
void getBCsNormal(Grid& grid, Matrix& M, BCType bcs[]);  
void getBCsCentered(Grid& grid, Matrix& M, BCType bcs[]);  
void output(Grid& grid, Matrix& M);  
  
#endif
```

We will collect in this header file, all the function prototypes which we would generate as a part of our implementation, so that these functions can be referred to in any module or in any file. So first function is our void input which corresponds to input module. It has got the parameters of valid grid boundary conditions and source too. Now this Grid&, this & stands for what we call in C++ as a reference type and so Grid& grid, BCType will have 2 boundary conditions as far as we need an array, this is an area dimension 2 and similarly a reference for the data qgen.

Similarly, we have also provided for a routine which will just print in the user specified information for crosschecking purposes. So again the parameters would be Grid & grid, BCType bcs and source& qgen. The next function corresponds to our matrix module that is to generate our computer matrix coefficient and we need to provide its input, the things related to grid, grid spacing, number of nodes, material property and so on.

We also need to provide matrix, in fact this Matrix& M, we would get all over competed matrix coefficients, so matrix entries. As input, we also need to provide the source term and the boundary condition types. For checking our matrix which has been generated or matrix in place which have been generated by grid matrix, we can have a print matrix function which will simply print out our entries matrix as well as right-hand side arrays.

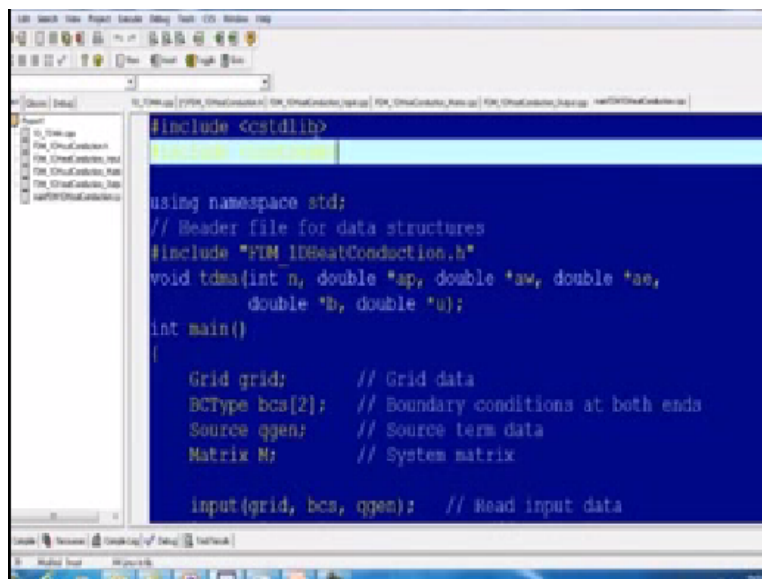
Then we have decided to incorporate or to introduce 2 submodules which will be called by our

get matrix function. So these 2 functions stand for 2 different finite difference formulations which we said we would use. So getBCSnormal, this particular function stands for the case where we would choose our vertex-based finite difference formulation and it would modify the matrix entries corresponding to the boundary conditions at (()) (23:48).

Similarly, for a cell-centred formulation, we can have a separate function which will do the same job and in the end, our last module was our output module. So we will have 1 function, void output provide Grid& grid, the grid information and the matrix information which contains our solution vector. So this is our header file which wherein we have collected all the data structure definitions and the prototypes of all the functions.

I would just like to point out the names which we have used here, they are just dummy names. If you want, you can get rid of this, the name here grid, we can just simply have capital G or ID&, that should be good enough. So all the names here are simply dummy names. In prototype, we primarily need the type information, okay.

**(Refer Slide Time: 24:51)**



```
#include <cstdlib>
#include <iostream>

using namespace std;
// Header file for data structures
#include "1DHeatConduction.h"
void tdata(int n, double *ap, double *aw, double *ae,
          double *b, double *u);
int main()
{
    Grid grid;           // Grid data
    BCType bcs[2];       // Boundary conditions at both ends
    Source qgen;         // Source term data
    Matrix M;            // System matrix

    input(grid, bcs, qgen); // Read input data
```

Next let us have our main or driver function, how will that look like, so what are included statements, #include cstdlib in case we want to use some functions from stand C library and usual C header, iostream which provides for input output as objects of ciostream library using namespace std so that all the standard library names are available to us and we have to include

our user defined header file which contains the definitions of all the data structures as well as function prototypes.

So `#include "FDM_1DHeatConduction.h"` this was the header file which we have generated. Since it is user-defined header file, we must put it double cotes. Next for solution purposes, we would use an external function, okay. Let us call it `tdma` which incorporates over tridiagonal matrix algorithm. We will discuss this in detail in our next module on solution of algebraic systems.

So why `tdma`, we need to provide the size in `n` and provide the size of the system and this `ap`, `aw`, and `ae`, they refer to 3 diagonals, `ap` is the main diagonal, `aw` stands for the lower diagonal and `ae` stands for the upper diagonal, then `b` stands for array which will store our right-hand side vector and we will get solution in vector `u` when this routine or this function returns.

Now let us have a look at our main function of the interface which we will have here. So in `main`, first thing which we have to do is, we have to define the appropriate datatypes. So `Grid` and let us chose the same name `grid` starting with small `g`, please remember that C++ is a case sensitive language, so capital `Grid` and small `grid`, they are 2 different identifiers. So that is why I have used this small `grid` as the name for our grid data.

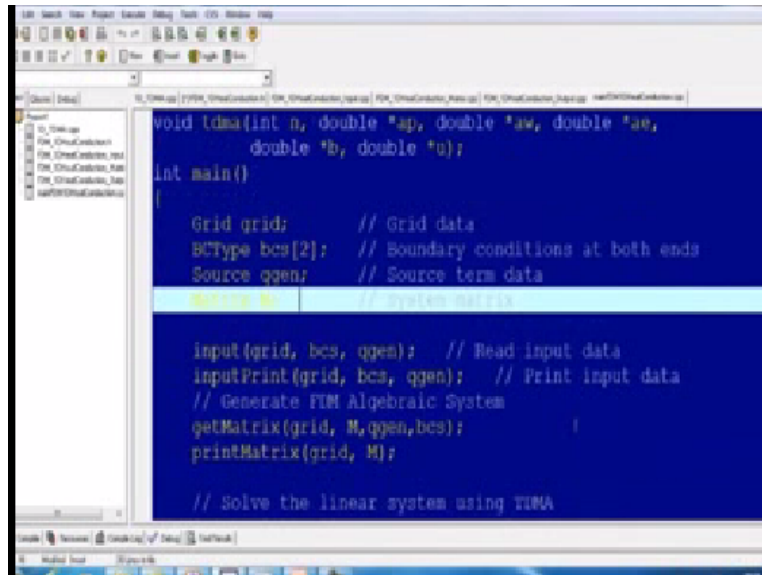
Next is `BCType` `bcs`, let us call this short form for boundary condition `bcs` within brackets 2, we have got 2 ends where boundary condition are there. So that is why we need to provide or we need to make it in array of size 2. Source `qgen`, this would hold the information about our heat generation term and matrix `M`, matrix for the data structure, we now `(( ))` (27:35) object of type or variable called `M` which will hold our complete finite difference algebraic system.

Remainder of the `main`, we will just going to call all the functions which correspond to different modules like for instance let us call the input module. So input name of this module is `input` and we will provide this 3 objects as the parameters. So grid related information would be returned by the input function in the object `grid`. Similarly, boundary condition information would be returned in this array `bcs` and source template related information would be returned in this



object qgen.

(Refer Slide Time: 28:28)



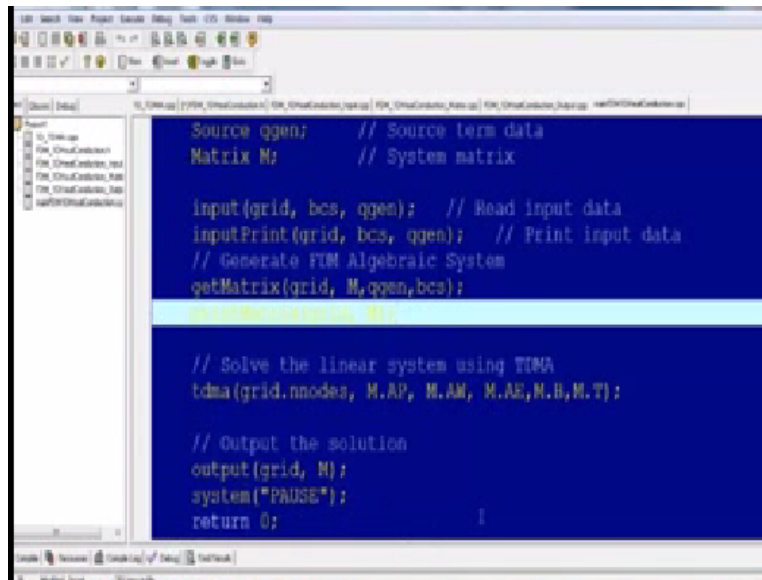
```
void tdm(int n, double *ap, double *aw, double *ae,  
double *b, double *u);  
  
int main()  
{  
    Grid grid;      // Grid data  
    BCType bcs[2];  // Boundary conditions at both ends  
    Source qgen;    // Source term data  
    double M;       // System matrix  
  
    input(grid, bcs, qgen); // Read input data  
    inputPrint(grid, bcs, qgen); // Print input data  
    // Generate FIM Algebraic System  
    getMatrix(grid, M, qgen, bcs);  
    printMatrix(grid, M);  
  
    // Solve the linear system using TMA
```

If you want to crosscheck, so we can call input print that is given input as grid bcs qgen, so it simply prints out all the information which have been input. In addition to that, it also prints out the number of computational nodes and the grid spacing. After input, we need to generate finite differences algebraic system. So call our matrix module which is represented by a function getMatrix. So getMatrix grid goes as input.

So grid, qgen and bcs, these 3 objects are provided as input to this getMatrix module and then it returns the output that is the assembled matrix entries ap, ae, aw and b, they are the arrays which are contained in this object M. So all those arrays taking care of the appropriate boundary conditions and the source term, they are all returned by our function getMatrix.

We can also print before it proceeds for solution, we can say that the entries which have been generated, the matrix entries whether they are correct or not, so we can call simple function print matrix which will print the matrix entries corresponding to each node. Then solve the linear system using tridiagonal matrix algorithm.

(Refer Slide Time: 29:52)



```
Source qgen; // Source term data
Matrix M; // System matrix

input(grid, bcs, qgen); // Read input data
inputPrint(grid, bcs, qgen); // Print input data
// Generate FIM Algebraic System
getMatrix(grid, M, qgen, bcs);

// Solve the linear system using TDMA
tdma(grid.nnodes, M.AP, M.AM, M.AE, M.B, M.T);

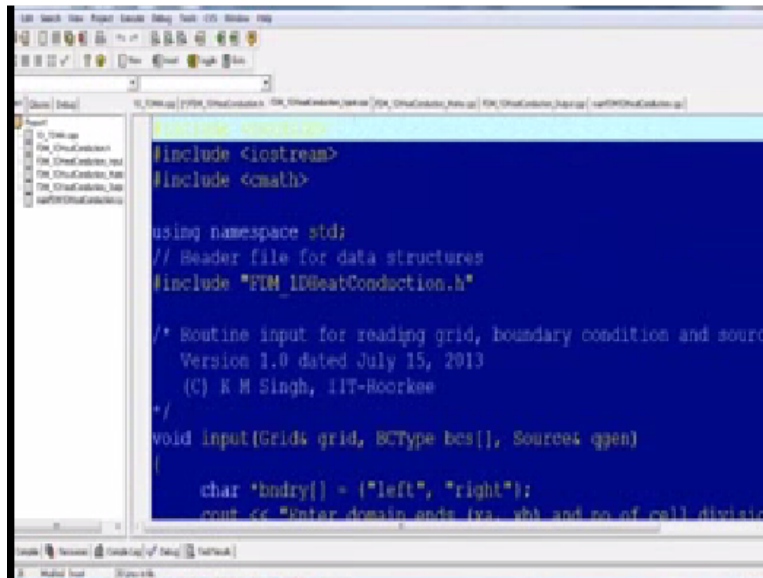
// Output the solution
output(grid, M);
system("PAUSE");
return 0;
```

So you will use this external function `tdma` `grid.nnodes`, `nnodes` is the number of grid nodes and then `M.AP`, this gives us our main diagonal, `M.AM` lower diagonal, `M.AE` upper diagonal, `M.B` is our RHS vector and the solution would be returned in vector `M.T` and once we have got the solution, output it, print in tabular form.

So output past these 2 parameters or objects as input, `grid` and/or matrix `M`, `grid` will contain the information about the number of grid nodes and x coordinates we want to print that out of each grid point and `M` would contain our solution vector service vector as `M.T` once it has return 0, this will be our main function aims.

So if you look at the implementation main function, it looks very simple, very clean. So that is how we normally proceed in writing any code, any useful or what a practical code, we will not have the actual implementation of different functions in our driver program or what we call main function. Driver program will only act as a controller and it will call different functions to help perform different set of actions. So input generate matrix call the solver and call the routine to provide the output, that would be a typical structure for any CFD code.

**(Refer Slide Time: 31:39)**



```
#include <iostream>
#include <cmath>

using namespace std;
// Header file for data structures
#include "FDM1DHeatConduction.h"

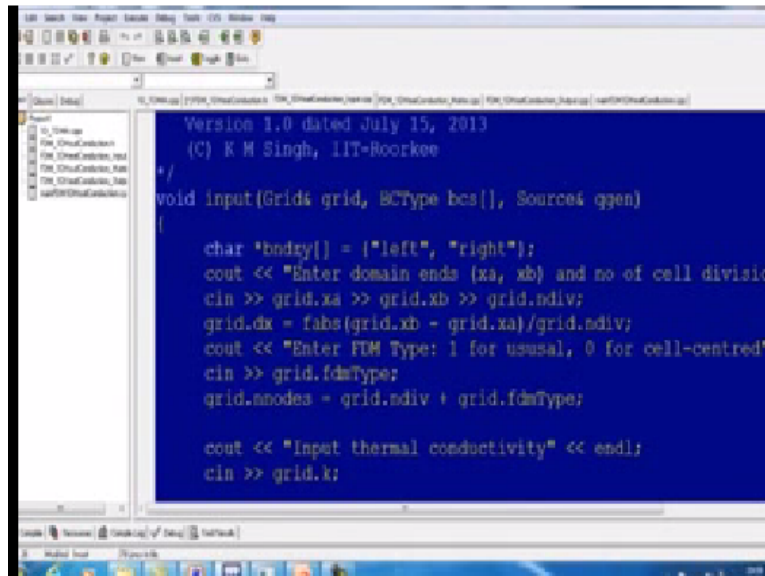
/* Routine input for reading grid, boundary condition and source
Version 1.0 dated July 15, 2013
(C) K M Singh, IIT-Bharat
*/
void input(Grid& grid, BCType bcs[], Source& qgen)
{
    char *bdry[] = {"left", "right"};
    cout << "Enter domain ends (x1, x2) and no. of cell divisions\n";
```

Now let us have a look at implementation of each module separately. So first let us go to our input module, okay #include iostream and I have just included a cmath in case we might need sometimes some mathematical functions from C library using namespace std for standard input and output objects and we must include the user-defined data structure file in each of our modules.

So #include "FDM1DHeatConduction.h" okay and this module we have got 2 functions, the main module which corresponds to function input and one function input print, okay. Let us look at the documentation, that is the usual coding approach that we should provide few lines of comment as to what that particular function is going to do. Although this is not complete comment, maybe should have also written what are these input things grid bcs and qgen.

So this would be 1 line comment each in this here that what that grid stands for, what is bcs stands for, what qgen stands for and then we should provide a version that on which date the code was written and if you want, you can also write the name of the developer. So this will help keep track of the changes which have been made to each module of function so that we know the updated features of each module.

**(Refer Slide Time: 33:14)**

A screenshot of a C++ IDE, likely Visual Studio, showing a code editor with a function definition for 'input'. The code is written in C++ and includes comments and standard input/output operations. The function signature is 'void input(Grid& grid, BCType bcs[], Source& qgen)'. The code prompts the user for domain ends (xa, xb) and the number of cell divisions (ndiv), calculates the grid spacing (dx), prompts for the FDM type (1 for usual, 0 for cell-centred), calculates the number of nodes, and prompts for thermal conductivity (k).

```
Version 1.0 dated July 15, 2013
(C) K M Singh, IIT-Bharatpur
*/
void input(Grid& grid, BCType bcs[], Source& qgen)
{
    char *bndry[] = {"left", "right"};
    cout << "Enter domain ends (xa, xb) and no. of cell divisions\n";
    cin >> grid.xa >> grid.xb >> grid.ndiv;
    grid.dx = fabs(grid.xb - grid.xa)/grid.ndiv;
    cout << "Enter FDM Type: 1 for usual, 0 for cell-centred\n";
    cin >> grid.fdmType;
    grid.nnodes = grid.ndiv + grid.fdmType;

    cout << "Input thermal conductivity\n";
    cin >> grid.k;
}
```

Now let us come to our actual function module. So void input, this particular module did not return anything, why its return typed. So that is why you have put return type as void. Everything returned through these parameters. So grid is a reference type parameter, grid&, grid. Please remember if you revoke this &, that becomes what we call in C++ language as pass value. So nothing will be returned back to the main program. We will not get anything out. So that is why this reference type is required.

If you want to instead of C++, you want to change to C language, user does not state change in reference to your pointer and create that pointer in the main function instead of an object call grid, okay, the same thing, the same comment holds good for this source object, source type object, qgen, but if you do not want to use references which are available only in C++ language, just change it to the pointer, so that will do a job.

We have defined a local variable called bndry, this is an array, character array just for the sake of prompting the user whether we are a left boundary or right boundary for inputting the boundary conditions. Now the way I have written this module is for an interactive way, okay. This can also be changed in a non-interactive fashion where all the input can be read from a file. So those modifications I think I would leave as a simple exercise, all that we need to do is just replace the cin by an appropriate file link object, rest other things would work in identical fashion.

So we would prompt the user to input each type data. So cout is that is what it does, it prints out a set of lines or set of characters on our console at inter-domain end xa, xb and number of cell divisions. So the first input which we are going to prompt the user and then user will input the data and that data is stored in grid.xa, that is x left in; grid.xb, the right end and grid.ndiv, this is the number of divisions.

Next we can compute the cell spacing, grid.dx which is equal to  $\text{grid.xb} - \text{grid.xa} / \text{grid.ndiv}$ . I have got this fabs, just in case if the user enters the things in a different order, it enters the right coordinate first and left coordinate later. So that can be taken care of because grid spacing has always what will be a positive number. Then we prompt the user to enter what we call finite difference or FDM Type whether we are going to use usual formulation.

So if input 1 for the usual finite difference and 0 for cell-centred ones. So cin grid.fdmType and this choice of 1 and 0 was made with a specific purpose and our usual finite difference formulations vertex based ones, our number of grid nodes is 1 more than the grid divisions. So if you put here number 1, the number of grid node is simply  $\text{grid.ndiv} + 1$  and for cell-centred, the number of grid nodes is equal to number of divisions. So this 0 works pretty well here.

So we need a single statement here. So  $\text{grid.nnodes} = \text{grid.ndiv} + \text{grid.fdmType}$ . Then prompt the user to input the thermal conductivity.

**(Refer Slide Time: 36:59)**

```

cout << "Enter domain ends (xa, xb) and no of cell division\n";
cin >> grid.xa >> grid.xb >> grid.ndiv;
grid.dx = fabs(grid.xb - grid.xa)/grid.ndiv;
cout << "Enter FDM Type: 1 for usual, 0 for cell-centred\n";
cin >> grid.fdmType;
grid.nnodes = grid.ndiv + grid.fdmType;

cout << "Input thermal conductivity" << endl;
cin >> grid.k;

// Input boundary conditions for the problem
for(int i = 0; i < 2; i++){
    cout << "Input boundary condition type at "
         << bcs[i] << " boundary" << endl;
    cin >> bcs[i].type;
    bcs[i].Tb = 0.0; bcs[i].qb = 0.0; // Default values
    bcs[i].h = 0.0; bcs[i].Ta = 0.0;
}

```

So cin grid.k and then we have to prompt the user for boundary conditions at both the ends. So let us use simple fall loop for int I=0, I< 2, i++ because we would store a boundary conditions in an array of length 2. The first bcs0 would hold the boundary condition in the left end and bcs1 that object would hold the boundary conditions at our right end. So let us prompt the users cout input boundary condition type at the given boundary node.

**(Refer Slide Time: 37:44)**

```

cout << "Input thermal conductivity" << endl;
cin >> grid.k;

// Input boundary conditions for the problem
for(int i = 0; i < 2; i++){
    cout << "Input boundary condition type at "
         << bcs[i] << " boundary" << endl;
    cin >> bcs[i].type;
    bcs[i].Tb = 0.0; bcs[i].qb = 0.0; // Default values
    bcs[i].h = 0.0; bcs[i].Ta = 0.0;
    if(bcs[i].type == 0){
        cout << "Enter specified boundary temperature" << endl;
        cin >> bcs[i].Tb;
    } else if(bcs[i].type == 1){
        cout << "Enter specified boundary flux" << endl;
    }
}

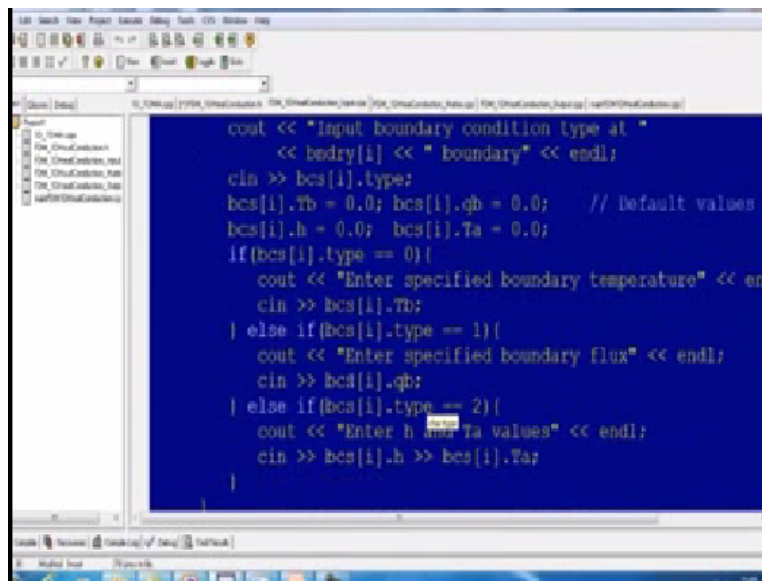
```

First data which would be input as bcs.type and once this bcs.type has been entered, we provide some default values. Bcs1.type could be anything, it could be 0, 1 or 2 that is Dirichlet boundary condition, Neumann boundary condition, or convective boundary condition, so let us just initialize these values because this Tb, qb, h and Ta, would not be specified by the user for each

time. User would specify only the required information.

So now let us provides some default values to all the parameters, default value of 0. So  $bcs[i].Tb=0$ ,  $qb=0$ ,  $h=0$  and  $Ta=0$  and then depending on the type, if type is equal to 0 that is Dirichlet boundary condition we have prompt the user to input specified boundary temperature and read it in  $bcs[i].Tb$ .

**(Refer Slide Time: 38:38)**

A screenshot of a C++ code editor window. The code defines a loop for boundary conditions. It prompts the user for a boundary type. If the type is 0, it asks for a temperature (Tb). If the type is 1, it asks for a flux (qb). If the type is 2, it asks for convective parameters (h and Ta). Default values of 0 are assigned to Tb, qb, h, and Ta.

```
cout << "Input boundary condition type at "
    << bndry[i] << " boundary" << endl;
cin >> bcs[i].type;
bcs[i].Tb = 0.0; bcs[i].qb = 0.0; // Default values
bcs[i].h = 0.0; bcs[i].Ta = 0.0;
if(bcs[i].type == 0){
    cout << "Enter specified boundary temperature" << endl;
    cin >> bcs[i].Tb;
} else if(bcs[i].type == 1){
    cout << "Enter specified boundary flux" << endl;
    cin >> bcs[i].qb;
} else if(bcs[i].type == 2){
    cout << "Enter h and Ta values" << endl;
    cin >> bcs[i].h >> bcs[i].Ta;
}
```

Else if if the boundary condition type is 1 that is the flux is specified, so ask the user to enter the specified boundary flux. So  $cin$   $bcs[i].qb$ , else if  $bcs$  type is equal to 2, that is to say, that it is convective boundary condition. So prompt the user to enter  $h$  and  $Ta$  values and read them in  $bcs[i].h$  and  $bcs[i].Ta$ . Now please be careful while writing a program.

So  $bcs[i].type$  for testing equality in C++ or C language, we have to provide 2 equal to signs. Do not just put a single equal to sign, single equal to sign stands for assignment, okay. So if you put a single equal to sign, that will lead to an error, a logical error, there will not be any compilation error prompt by the compiler but your program will not work correctly. So now we have input, the boundary condition data.

**(Refer Slide Time: 39:34)**

```

    cout << "Enter specified boundary flux" << endl;
    cin >> bcs[i].qb;
    | else if(bcs[i].type == 2){
    cout << "Enter h and Ta values" << endl;
    cin >> bcs[i].h >> bcs[i].Ta;
    |
    }
    // Input source term
    cout << "Assume heat generation rate expressed as: " << endl;
    << "qg = qga + qgb*T. Enter qga and qgb" << endl;

    cin >> qgen.qga >> qgen.qgb;

    // Print the input data
    void inputPrint(Grid& grid, BCType bcs[], Source& qgen)

```

Next we need to input the source terms. So prompt the user, prompt the message that in what form we expect the input to be, so assume heat generation rate is expressed as  $qg = qga + qg \cdot T$  that is a linear variation of temperature and then ask the user to enter qga and qgb in that order and read it, cin qgen.qga which will contain the constant part of the source term and qgen.qgb which will contain the coefficient of the temperature and thereby that finishes our input module.

**(Refer Slide Time: 40:22)**

```

    // Input source term
    cout << "Assume heat generation rate expressed as: " << endl;
    << "qg = qga + qgb*T. Enter qga and qgb" << endl;

    cin >> qgen.qga >> qgen.qgb;

    // Print the input data
    void inputPrint(Grid& grid, BCType bcs[], Source& qgen)
    {
        char *bdry[] = {"left", "right"};
        cout << "\n INPUT data entered" << endl;
        cout << "xa = " << grid.xa << " xb = " << grid.xb
        << " ndiv = " << grid.ndiv << endl;
        cout << "fdmType = " << grid.fdmType << " nNodes = "

```

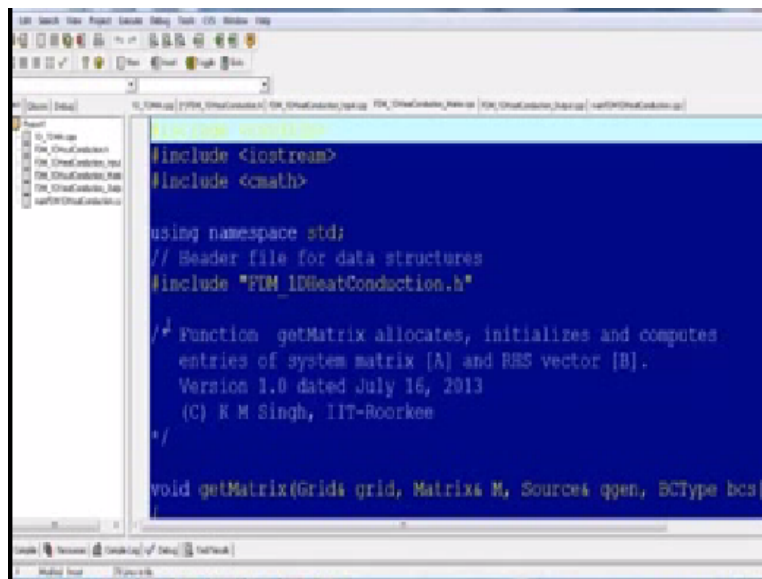
For the testing purposes, we can just print our input in a tabular format. So we can just have this testing routine which will print the input data void input print, ask the grid object BCType and source gen and let it print it in nice tabular format for us, print the grid ends, number of divisions, FDM formulation type, number of grid nodes, grid spacing and so on, print the thermal



conductivity, print boundary conditions, print the source term.

So now the collection of these 2 routines that completes our input module which will read the input data for us and it will also provide the printout of the input data which has been entered by the user for a crosscheck. So we can make sure that we have entered all the data correctly. Next let us proceed to our matrix module.

**(Refer Slide Time: 41:24)**



```
#include <iostream>
#include <cmath>

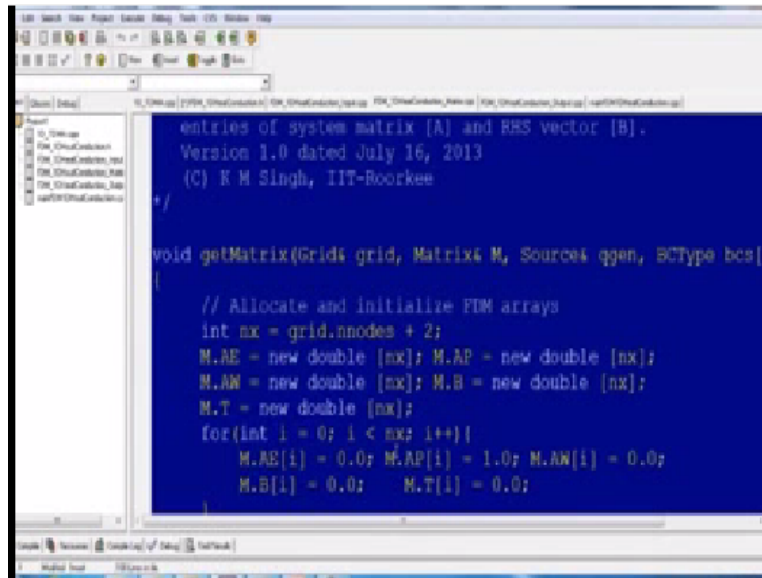
using namespace std;
// Header file for data structures
#include "FDM_1DHeatConduction.h"

/* Function getMatrix allocates, initializes and computes
entries of system matrix [A] and RHS vector [B].
Version 1.0 dated July 16, 2013
(C) K M Singh, IIT-Roorkee
*/

void getMatrix(Grids grid, Matrix& M, Sources& qgen, BCType bcs)
```

So once again we have included this header file FDM1DHeatConduction.h and let us put a comment here saying its function. So this function getMatrix, in this module, we put this function getMatrix which will allocate all the arrays which are part of our data structure M. So it allocates, initialises and computes the entries of system matrix A and right-hand side vector B. It also allocates memory and initialises our solution vector T. So let us put the version and date.

**(Refer Slide Time: 42:12)**



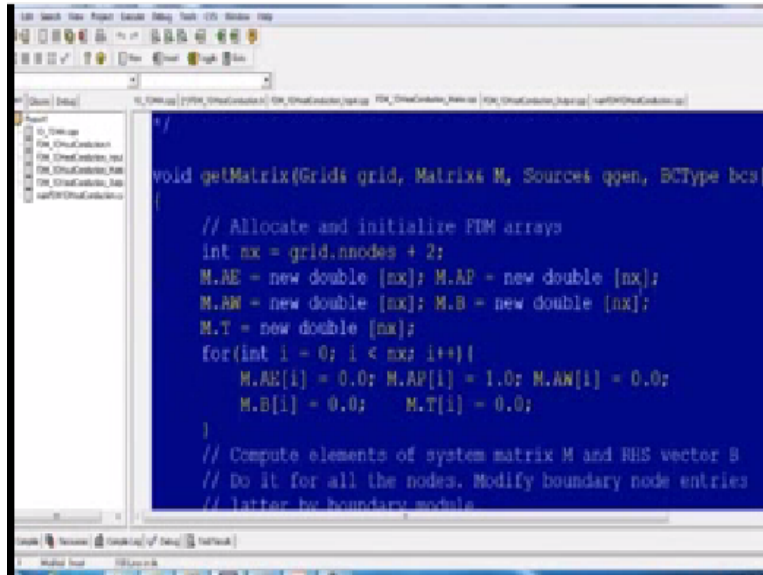
```
entries of system matrix [A] and RHS vector [B].
Version 1.0 dated July 16, 2013
(C) R M Singh, IIT-Borkee
+/-

void getMatrix(Grid& grid, Matrix& M, Source& qgen, BCType bcs)
{
    // Allocate and initialize FDM arrays
    int nx = grid.nnodes + 2;
    M.AE = new double [nx]; M.AP = new double [nx];
    M.AW = new double [nx]; M.B = new double [nx];
    M.T = new double [nx];
    for(int i = 0; i < nx; i++)
        M.AE[i] = 0.0; M.AP[i] = 1.0; M.AW[i] = 0.0;
        M.B[i] = 0.0; M.T[i] = 0.0;
}
```

Now let us have a look at the implementation. Once again return type is void because we do not need a single data to be returned here. So let it be void, getMatrix our mode function name. The first parameter is a reference type parameter, Grid& grid which is an input to this module. Matrix& M, this is where the output from this mode would be returned, Source& qgen and BCType bcs will again our input to this module.

In our standard commercial codes, we would put keyword const before these input only parameters grid qgen and similarly this array BCS because this particular function is not supposed to change these. So if you want you can put this keyword const here, so that will ensure that no modifications were made to grid data or source data or boundary condition data inadvertently by this module.

**(Refer Slide Time: 43:20)**

A screenshot of a C++ IDE, likely Visual Studio, showing a code editor with a blue background. The code is written in C++ and defines a function named `getMatrix`. The function takes four parameters: `Grid& grid`, `Matrix& M`, `Source& qgen`, and `BCType bcs`. Inside the function, there is a comment `// Allocate and initialize FEM arrays` followed by the declaration of a local integer variable `nx = grid.nnodes + 2;`. Then, four dynamic arrays are allocated using the `new` operator: `M.AE = new double [nx];`, `M.AP = new double [nx];`, `M.AW = new double [nx];`, and `M.B = new double [nx];`. A fifth array, `M.T`, is also declared as `M.T = new double [nx];`. A `for` loop is used to initialize the arrays from index 1 to `nx`. Inside the loop, the following assignments are made: `M.AE[i] = 0.0;`, `M.AP[i] = 1.0;`, `M.AW[i] = 0.0;`, and `M.B[i] = 0.0;`. The `M.T[i]` entry is also set to 0.0. After the loop, there are three more comments: `// Compute elements of system matrix M and RHS vector B`, `// Do it for all the nodes. Modify boundary node entries`, and `// latter by boundary module`. The IDE interface includes a menu bar at the top, a toolbar, and a file explorer on the left side.

Okay, so the first task which this module is suppose to do is allocate the space because all the arrays which are part of M data structural, they are all dynamic arrays. So we need to allocate space for them and initialise them. Now let us introduce local variable `int nx=grid.nnodes+2`. We have added this +2 for a specific reason. Index 0, all the matrix and matrices in C or arrays in C or C++, their indices start from 0 but that 0 index we are not using here. Our grids, we have followed a standard format 1, 2, 3 and so on and in the case, we will go up to `n+1`.

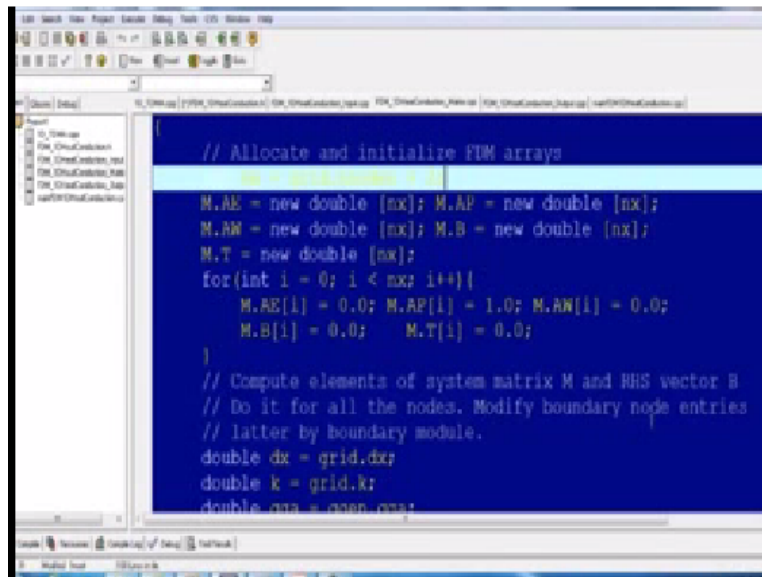
So what we should do is, we should provide the size of our allocated array to be slightly larger to take care of the first 0th entries. Let us provided for an additional host note to the right. So that is why this additional size +2 has been provided here and we would use simple new operator available in C++. So `M.A=new double [nx]`.

What does it do, it creates a space, memory space which can hold `nx` double entries and address of that memory space which will represent an array is returned and stored in this pointer `M.AE`, okay. So `M.AE` now becomes a dynamically allocated array of size `nx`. Similarly, `M.AP=new double nx`, `M.AW`, `M.B` and `M.T`, they have all been dynamically allocated.

Next is, let us initialises this. See initializing is very important. We should provide initial values to all the variables which we use in a code. Now let us initialize them with some appropriate values. So `AE`, `AW`, `B` and `T`, these have been initialised to 0.0. `AP`, I have purposely initialised as

1 because of the algorithm, use AP in the denominator while solving the system equations that if at 0, it might create some problems for us. So let us by default set all the values of the main diagonal at 1.

(Refer Slide Time: 46:01)



```

// Allocate and initialize FEM arrays
nx = grid.nnodes + 1;

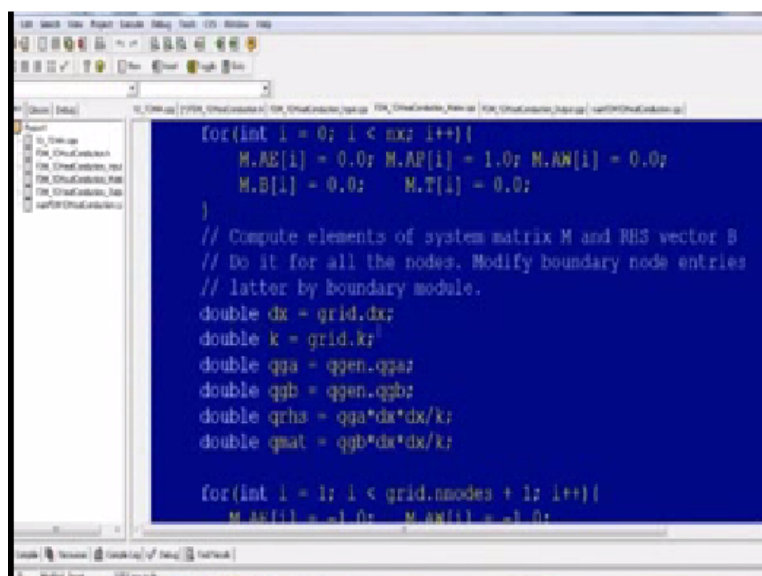
M.AE = new double [nx]; M.AP = new double [nx];
M.AW = new double [nx]; M.B = new double [nx];
M.T = new double [nx];
for(int i = 0; i < nx; i++){
    M.AE[i] = 0.0; M.AP[i] = 1.0; M.AW[i] = 0.0;
    M.B[i] = 0.0; M.T[i] = 0.0;
}

// Compute elements of system matrix M and RHS vector B
// Do it for all the nodes. Modify boundary node entries
// latter by boundary module.
double dx = grid.dx;
double k = grid.k;
double qqa = qgen.qqa;

```

Our next task is to compute the elements of systemMatrix M and right-hand side vector. First we will do it for all the nodes assuming all the nodes are interior nodes, that is what happens in our cell-centred formulation, all the nodes are interior nodes. Even in normal formulation, it does not hurt much. We have got 2 additional nodes to repeat the task and we would modify the boundary node entries later by calling appropriate boundary modification module.

(Refer Slide Time: 46:33)



```

for(int i = 0; i < nx; i++){
    M.AE[i] = 0.0; M.AP[i] = 1.0; M.AW[i] = 0.0;
    M.B[i] = 0.0; M.T[i] = 0.0;
}

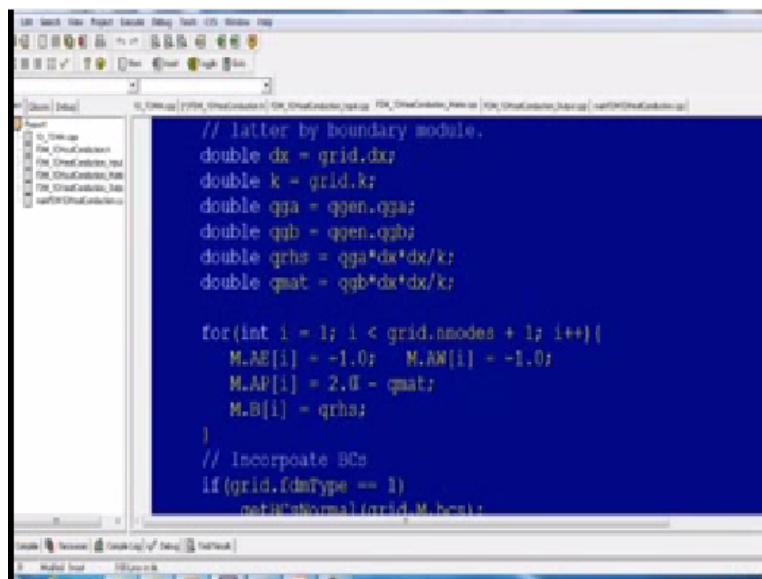
// Compute elements of system matrix M and RHS vector B
// Do it for all the nodes. Modify boundary node entries
// latter by boundary module.
double dx = grid.dx;
double k = grid.k;
double qqa = qgen.qqa;
double qqb = qgen.qqb;
double qrhs = qqa*dx*dx/k;
double qmat = qqb*dx*dx/k;

for(int i = 1; i < grid.nnodes + 1; i++){
    M.AE[i] = -1.0; M.AW[i] = -1.0;

```

For the sake of simplicity, we have introduced some local variables, double dx which is hold screen.dx, k grid.k, qga qgen.qga, qgb qgen.qgb. The contribution which would be generated to rhs, qrhs is  $qga \cdot dx \cdot dx / k$ . Remember there is no exponential operator, so we cannot have  $dx^2$  to represent  $dx^2$ . You have to type  $dx \cdot dx$  in C or C++ language. Similarly, contribution for the linear coefficient of the coefficient of temperature and source term that will also has to be incorporated in AP, so that contribution qmat, qmat is  $qgb \cdot dx \cdot dx / k$ .

**(Refer Slide Time: 47:21)**



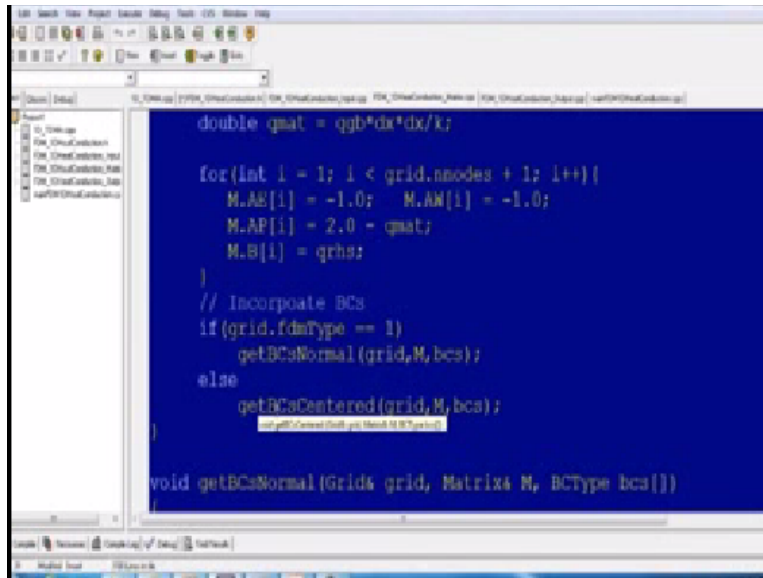
```
// latter by boundary module.
double dx = grid.dx;
double k = grid.k;
double qga = qgen.qga;
double qgb = qgen.qgb;
double qrhs = qga*dx*dx/k;
double qmat = qgb*dx*dx/k;

for(int i = 1; i < grid.nnodes + 1; i++){
    M.AE[i] = -1.0;    M.AW[i] = -1.0;
    M.AP[i] = 2.0 - qmat;
    M.Bi[i] = qrhs;
}

// Incorporate BCs
if(grid.fdmType == 1)
    getBCNormal(grid.M.bcns);
```

Next, let us generate all the entries. So  $i=1, i < \text{grid.nnodes}+1$ , so that we go from 1 to n nodes,  $i++$ . AE and AW entries are -1. Now  $M.AP_i$  is  $2.0 - qmat$  and  $M.Bi = qrhs$ . So that is our right-hand side.

**(Refer Slide Time: 47:51)**



```
double qmat = qgb*dx*dx/x;  
  
for(int i = 1; i < grid.nnodes + 1; i++){  
    M.AB[i] = -1.0;    M.AM[i] = -1.0;  
    M.AP[i] = 2.0 - qmat;  
    M.B[i] = qrhs;  
}  
  
// Incorporate BCs  
if(grid.fdmType == 1)  
    getBCsNormal(grid,M,bcs);  
else  
    getBCsCentered(grid,M,bcs);  
}  
  
void getBCsNormal(Grid& grid, Matrix& M, BCType bcs[])  
{  
    // getBCsNormal: get BCs for Normal FD Formulation  
}
```

And next we need to incorporate the effect of boundary conditions and we will call 2 separate functions or modules to do the task for us. So if `grid.fdmType=1`, that is we are dealing with normal finite difference formulation, let us call a function call `getBCsNormal(grid,M,bcs)` else we will assume a default implementation to be cell-centred formulation. So `getBCsCentered` pass as input `grid`, `M` and `bcs`.

Now we will discuss these 2 functions in detail in our next lecture. We will look at the pseudo-code first and then how do we translate them in appropriate functions and thereafter we will build up our code further and solve 2 example problems in the next lecture.