12- User-Defined Material Model

In this version 9.0 of Phase2 (RS^2), users can define their own constitutive model and integrate the model into the program by using a dynamic-linking library (dll). The name of the dll should be Phase2UDM.dll and needs to be placed in the installation folder of the program. The program will then detect and load the dll. One or more materials can be included in the dll. An example dll including two materials is included in the UDM folder in the Phase2 9.0 installation folder. The details on how to create a dll are provided in the next section.

The dll should be written in C++ language although other languages can be used. However, the compatibility will not be warranted.

12.1- Programme structure

12.1.1- Pre-requisite

All the header files included in the UDM folder must be included in the dll in order to compile. (Matrix.h, MatUserDefined.h, Phase2UDMx64.h and Realtype.h). The dll also needs to be linked with the Phase2UDMx64.lib (for 64 bit version) or Phase2UDMx32.lib (for 32 bit version).

Note: The static library files were compiled with the release configuration. If you compile with the debug configuration, you may not get the correct values for debugging.

The dll consists of classes for each material and several export functions. The material classes are inherited from the base class name MatUserDefined. The export functions are created to provide the information of the material before the calculation actually carried out.

12.1.2- Export functions

The following functions must be provided in the dll. Function format and functionality will be explained together. Please note that those examples are taken from the example cpp file included in the installation.

1. Return number of materials in the dll:

```
extern "C" __declspec(dllexport) int GetNumberOfMaterial()
{
	return 2;
}
```

This function returns the number of materials in the dll. Since we have two materials, it should return 2.

2. Return names of the materials in the dll:

```
extern "C" __declspec(dllexport) char* GetMaterialName(int id)
{
	char *matName;
	if (id == 0) {matName = "Elastic";}
	else if (id == 1) {matName = "Mohr";}
	else
```

```
                matName = "No material was found";
        return matName;
}
```

This function returns the name of the material based on the input *id*. In this example the function will return Elastic if id equals to 0 and Mohr if id equal to 1.

Note: The user should provide the id starting from 0 as the program uses C index style.

3. Return number of parameters in the constitutive model:

```
extern "C" __declspec(dllexport) int GetNumberOfConstitutiveParameter(char*
matName)
{
        int nVars = 0;
        if (strcmp (matName,"Elastic") == 0) {nVars = 2;}
        else if (strcmp (matName,"Mohr") == 0) {nVars = 5;}
        return nVars;
}
```

This function returns the number of the parameters of the materials based on the input *matName*. The user should return the number of parameters used in the material. In the Elastic material, there are only two parameters (Young modulus and Poisson's ratio) and in the Mohr material, there are five parameters (Young modulus, Poisson's ratio, friction angle, cohesion and dilation angle).

4. Return the name of the parameters in the constitutive model

```
extern "C" __declspec(dllexport) char* GetNameOfConstitutiveParameter(char* matName,
int paramId)
{
    char* paramName;
    if (matName == "Elastic") {
            switch (paramId){
                    case 0:
                            paramName = "Young_modulus";
                            break;
                    case 1:
                            paramName = "Poisson_ratio";
                            break;
                    default:
                            paramName = "No param found";
                            break;
            }
    }
    else if (matName == "Mohr") {
            switch (paramId){
            case 0:
                    paramName = "Young_modulus";
                    break;
            case 1:
                    paramName = "Poisson_ratio";
                    break;
            case 2:
                    paramName = "Cohesion";
                    break;
```

```
            case 3:
                    paramName = "Friction_Angle";
                    break;
            case 4:
                    paramName = "Dilation_Angle";
                    break;
            default:
                    paramName = "No param found";
                    break;
            }
    }
    else
            paramName = "No material was found";
    return paramName;
}
```

This function returns the name of the parameters of the materials based on the two input *matName* and *paramId*. *matName* is one of the name of the material that the user provided in the *GetMaterialName()* function. Similar to other id, *paramId* also needs to be started at 0 as C index style.

5. Return a pointer to the material class

```
extern "C" __declspec(dllexport) MatUserDefined* GetMaterial(char* names)
{
    MatUserDefined* mat = NULL;
    if (strcmp (names,"Elastic") == 0){
            mat    = new Elastic();
    }
    if (strcmp (names,"Mohr") == 0){
            mat    = new Mohr();
    }
    return mat;
}
```

This function returns the pointer of the class based on the input id and names of the materials. The input *id* is the internal variable of Phase2 and should not be modified by the user. The variable *names* is one of the material names that are provided by the user.

12.1.3- Class structures

The general ideas are that the programs will provide the incremental strains, previous stresses and strains, state variables and failure information and the user will provide the current stresses, state variables and failure information for every iteration. Those procedures will be carried out at the class scope.

Besides that main function, the user also needs to initialize specific parameters for the material such as the number of state variables, the material matrix is symmetric or not or the tangential stiffness is required for the material. Those parameters will be declared at the material constructors.

All constitutive models must inherit from the *MatUserDefined* (material) class. The implementations are in the Phase2UDMx64.lib/Phase2UDMx32.lib. In addition to the material classes, the library also includes classes to provide the basic math operators on vector and matrix (Matrix.h)

Generally, the following functions are needed to be defined in the class:

1. ***Elastic();***

   There are parameters inherit from the main class that needed to be declared for the materials:

   `bool isTangential:` The material is calculated using the tangential stiffness or not. If set to true, the user needs to provide the plastic stiffness matrix.

   `int nStateVars:` number of the state variables. For example, the Mohr-Coulomb model may exhibit hardening or softening behaviour. In those cases, the failure envelope will depend on strain. Generally a scalar 'equivalent plastic strain' is used. Thus, the nStateVars should be set to 1 in those cases.

   An example implementation is:

```
Elastic::Elastic() : CMatUserDefined()
{
	isTangential = false;
	nStateVars = 1;
}
```

2. ***~Elastic(void);*** Destructor of the class

3. ***Matrix Elastic::ComputeElasticMatrix(Vector stress, Vector strain, Vector& stateVariables, int& failureType)***

   This returns the elastic stiffness matrix. The input matrix Emat has the size of 4x4. The `Matrix` class and `Vector` class is defined in *Matrix.h*

   The others input is a reference to stresses and strains and state variables. This is necessary because the elastic stiffness matrix may depend on stress or some other state variable stored on the gauss point.

   The size of the vector stateVariables is ***nStateVars*** as declared in the constructors.

   Storage for a matrix is also provided with the gauss point. This is basically for speed, if the user wishes to store some stiffness matrix instead of calculating it each time it is needed through `ComputePlasticMatrix.`

   An example implementation is:

```
Matrix Elastic::ComputeElasticMatrix(Vector stress, Vector strain, Vector&
stateVariables, int& failureType)
{
	Matrix D(4,4);
	D = 0.;

	assert( abs((1 + _dNu) * (1 - 2*_dNu)) > 10e-15 );
	assert( abs(1 - _dNu) > 10e-15 );
	assert( abs(2 * (1 - _dNu)) > 10e-15 );
	double coefficient = _dE * (1 - _dNu)    / ((1 + _dNu) * (1 - 2*_dNu));
	D(0, 0) = coefficient;
	D(0, 1) = coefficient * _dNu / (1 - _dNu);
	D(1, 1) = coefficient;
	D(1, 0) = D(0, 1);
	D(2, 2) = coefficient * (1 - 2 * _dNu)   / (2 * (1 - _dNu));
	D(3, 3) = coefficient;
	D(0, 3) = D(1, 3) = D(3, 0) = D(3, 1) = D(0, 1);
	return D;
}
```

4. ***Matrix Elastic::ComputePlasticMatrix(Vector stress, Vector strain, Vector& stateVariables,int& failureType)***

   Similar to `ComputeElasticMatrix()` but it will be called when the int& ***failureType*** is set to failure (1: Shear, 2: Tension, 3: Shear and tension, 4: critical failure)

5. ***void Elastic::UpdateStress(Vector& stress, Vector& strain, Vector& stateVariables, Vector& dStrain, int& failureType)***

   This is the main function that contains the relationship between the stresses and strains of the material. For the case of a classical elasto-perfect plastic Mohr coulomb material model this will include the following steps:

   - Work out the elastic stress by taking the existing input stresses from the program and adding the elastic stiffness matrix times the input strain increment
   - Check if stress is past the yield surface
   - If not past yield, set the stress to the elastic stress and leave the function
   - If past yield, bring stress back to the yield surface and set stresses to this value
   - Set failure variables if failure has occurred.

   An example implementation is:

```
void Elastic::UpdateStress(Vector& stress, Vector& strain, Vector& stateVariables,
Vector& dStrain, int& failureType)

{
	if (stateVariables.size()==0) stateVariables.resize(1);
	Matrix D = ComputeElasticMatrix(stress, strain, stateVariables,
failureType);
```

```
        size_t sz = dStrain.size();
        Vector ds = D * dStrain;
        stress += ds;
    }
```

6. void Elastic::SetConstitutiveParameters(Vector& parameters)

This function is the interface that allows the main program to set the private constitutive model parameters in your class. The input is a vector of double-precision numbers (The `Vector` class is defined in the library – *Matrix.h*). The implementation of the function should use these values to set your constitutive model variables. It is very important that the order of the parameter output in the `GetNameOfConstitutiveParameter()` in section ***Export Functions*** matches the order of variables in this function.

An example implementation is:

```
void Elastic::SetConstitutiveParameters(Vector& parameters)
{
    _dE = parameters[0];
    _dNu = parameters[1];
}
```

7. void DeleteMaterial(void);

This is necessary so that the material object can delete itself when it is no longer in use to prevent memory leaks.

8. MatUserDefined* Elastic::MakeCopy()

This function returns a pointer to a new created memory allocated for the user defned material model.

An example implementation is:

```
MatUserDefined* Elastic::MakeCopy()
{
    MatUserDefined* newMat = new (Elastic)(*this);
    return newMat;

}
```

9. double Elastic::GetElasticModulus()

Return the elastic young modulus of the material, used to calculate the parameters needed for some types of simulations.

An example implementation is:

```
double Elastic::GetElasticModulus(){return _dE;}
```

10. double Elastic::GetBulkModulus()

Return the elastic bulk modulus of the material, used to calculate the parameters needed for some types of simulations.

An example implementation is:

```
double Elastic::GetBulkModulus(){return _dE/(3*(1-2*_dNu));}
```

11. double Elastic::GetShearModulus()

Return the shear modulus of the material, used to calculate the parameters needed for some types of simulations.

An example implementation is:

```
double Elastic::GetShearModulus(){return _dE/(2*(1.+_dNu));}
```

12. double Elastic::GetPoissonRatio()

Return the Poisson's ratio of the material, used to calculate the parameters needed for some types of simulations.

An example implementation is:

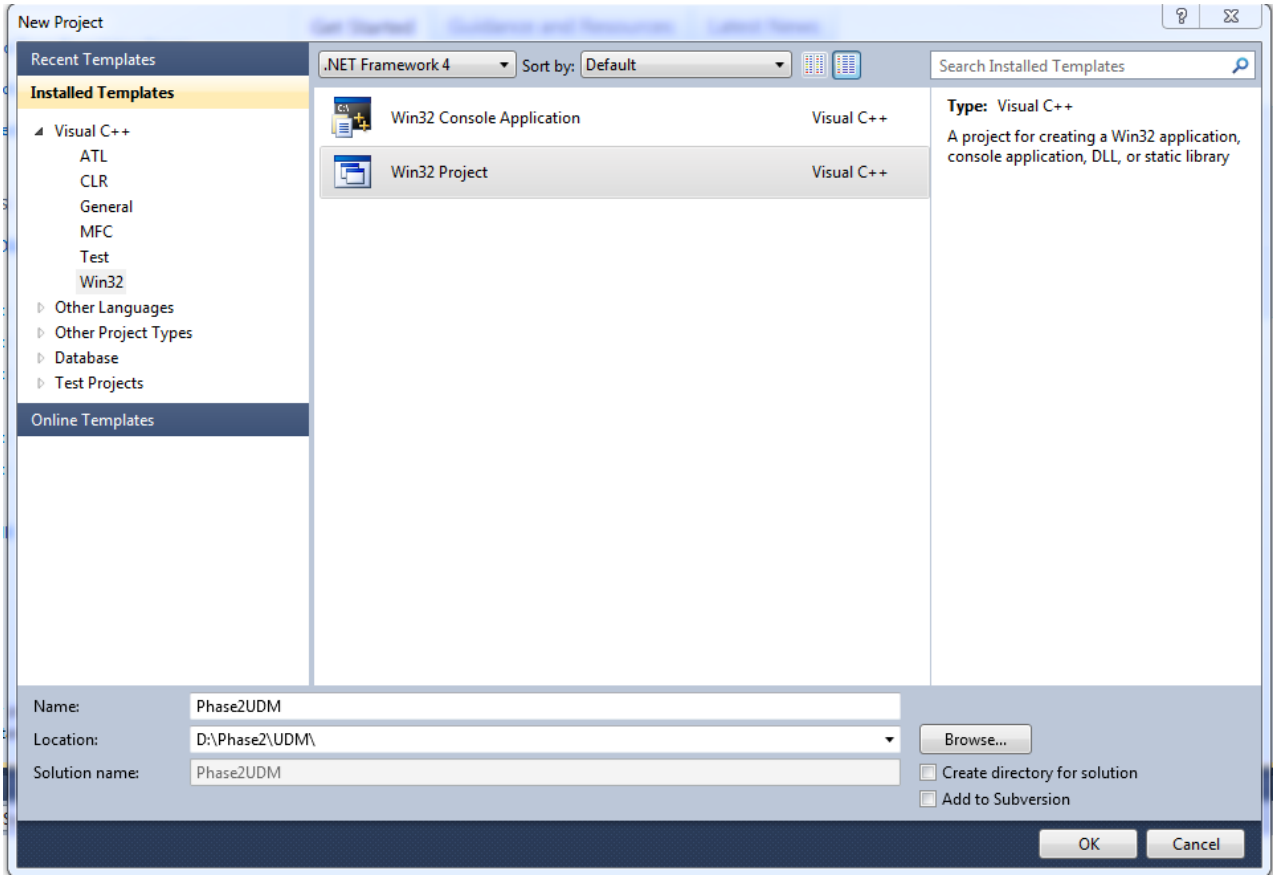
```
double Elastic::GetPoissonRatio(){return _dNu;}
```

for predefined math functions from the Phase2 library refer to the Matrix.h for the implementation and its functionality.

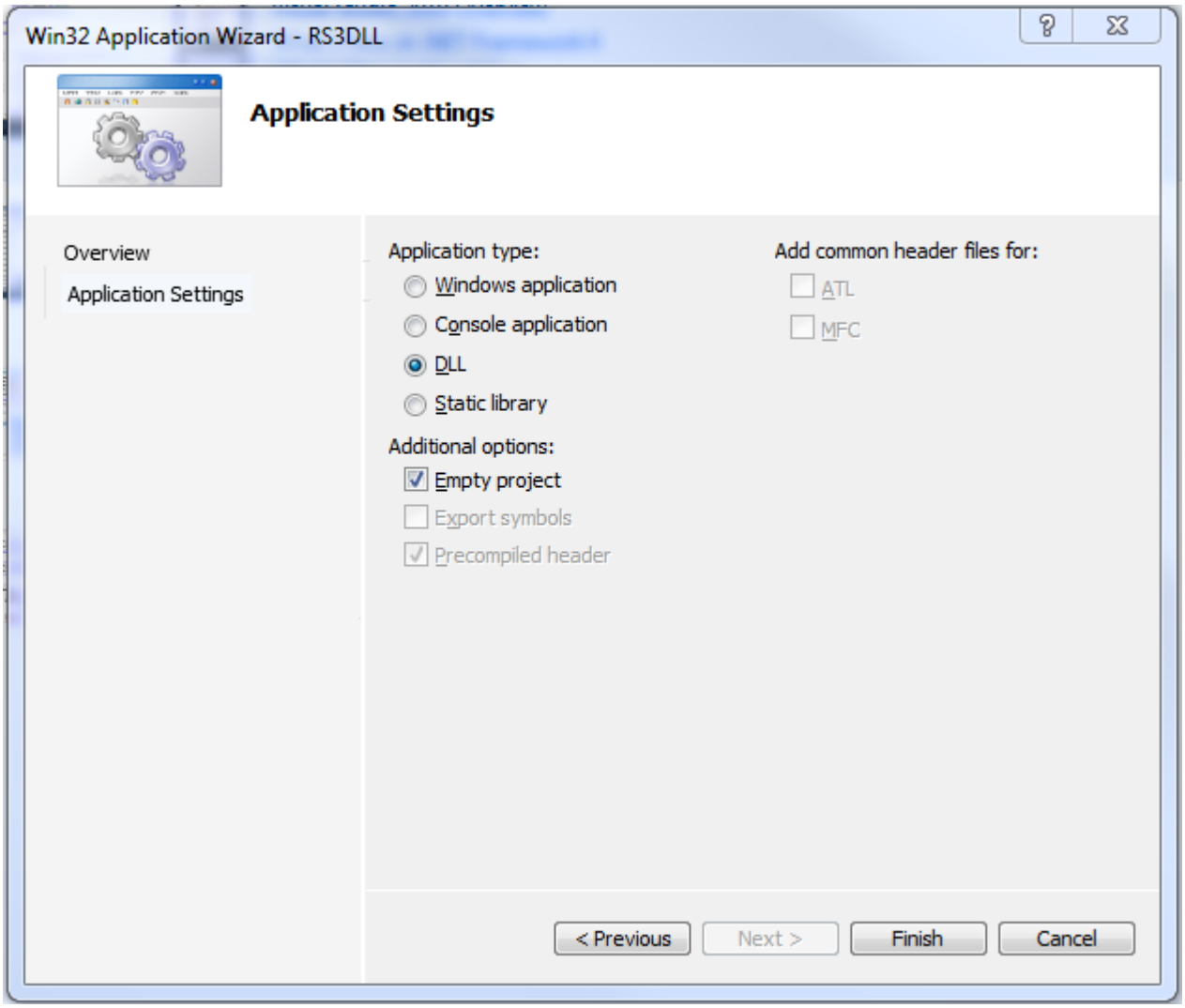
12.2- Walkthrough for creating the dll file

12.2.1- Create a dll project with Microsoft Visual Studio 2010

Select New Project. Under Visual C++ Projects, type in Phase2UDM in the name and select OK.



Click on Application Settings and under Application type: select DLL. Under Additional Options: select Empty project. Click Finish.

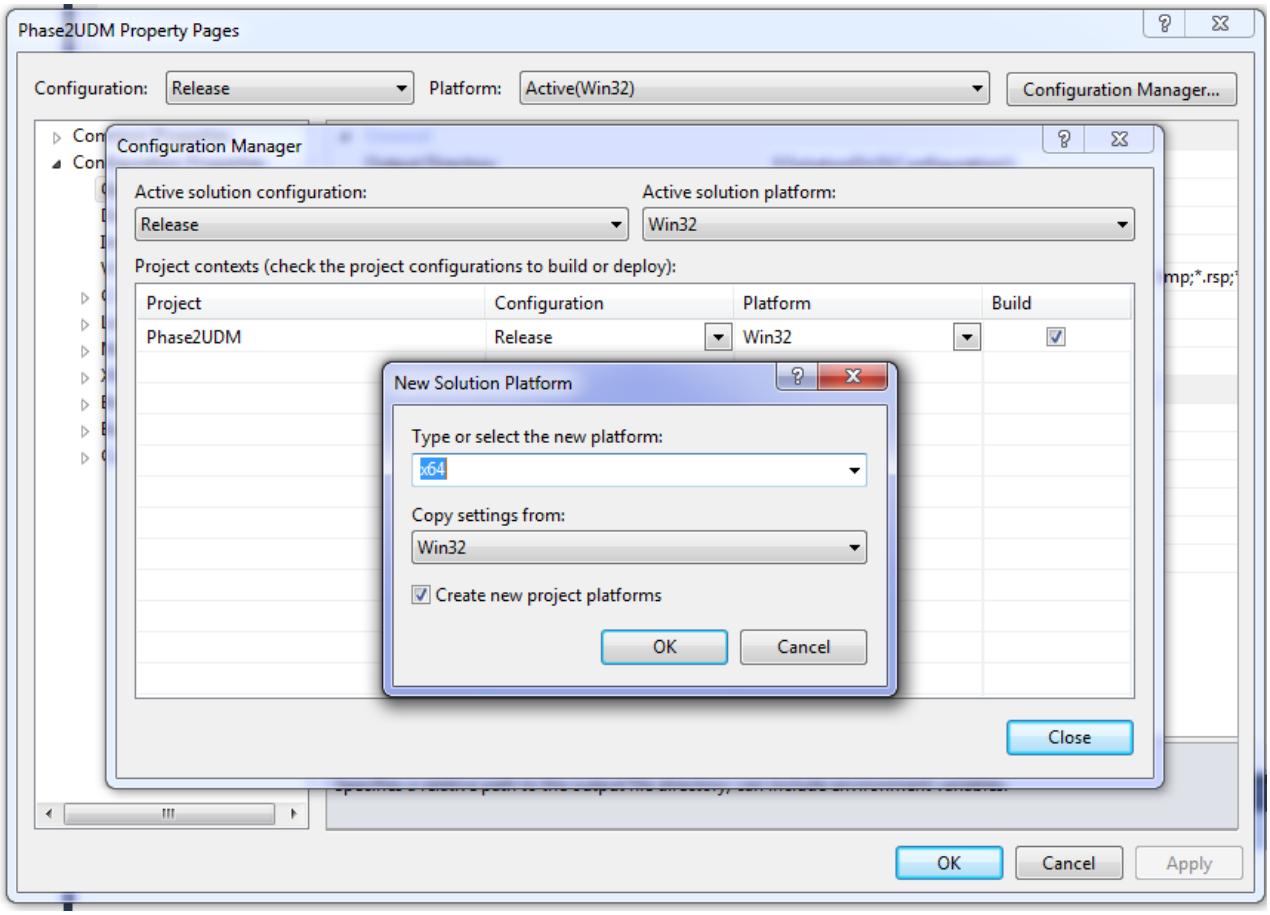


12.2.2- Using the Phase2 library

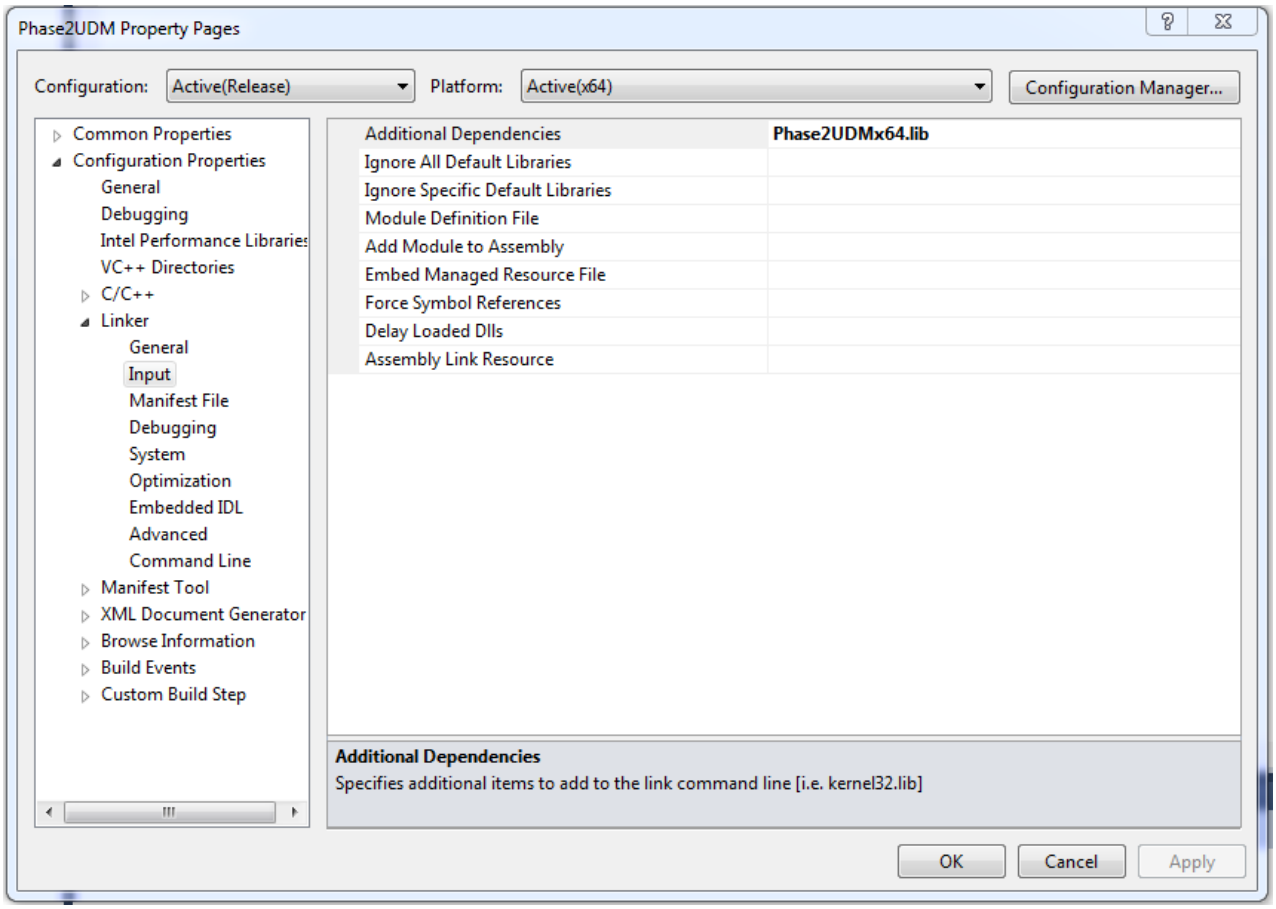
Now add the following files to the project: {Matrix.h; MatUserDefined.h and RealType.h }. Those files are included in the UDM folder in the installation folder of Phase2 version 9.0.

Do this by putting these files in the same directory as your project file (.vcxproj) and then in Visual Studio right click on the project name on the left and choose Add->Existing Item. Choose those files and click Open.

Depending on the version of your Phase2 9.0, we need to compile a corresponding version of the dll (x32 or x64). Right click on the project name on the left and choose Properties. Go to Configuration Manager on the top right of the dialog. Click on Platform->New. Pick x64 (or x32) from the drop down menu and click OK.



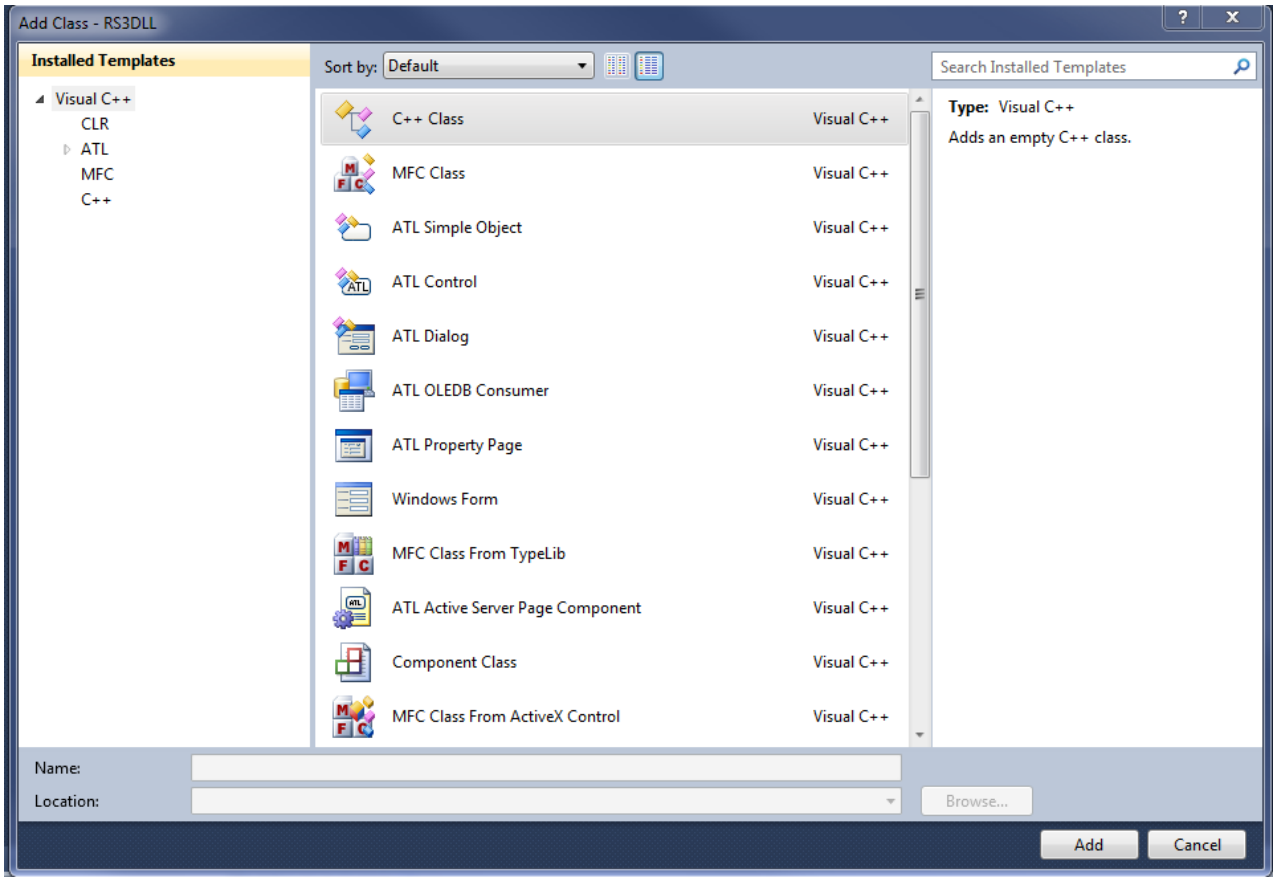
The implementation of these classes is in the file. The file also needs to be in the project folder. Right click on the project on the left and select Linker → Input. Beside Configuration at the top of the window, select All Configurations from the pull-down menu. Now in the box next to Additional Dependencies type Phase2UDMx64.lib (Phase2UDMx32.lib).



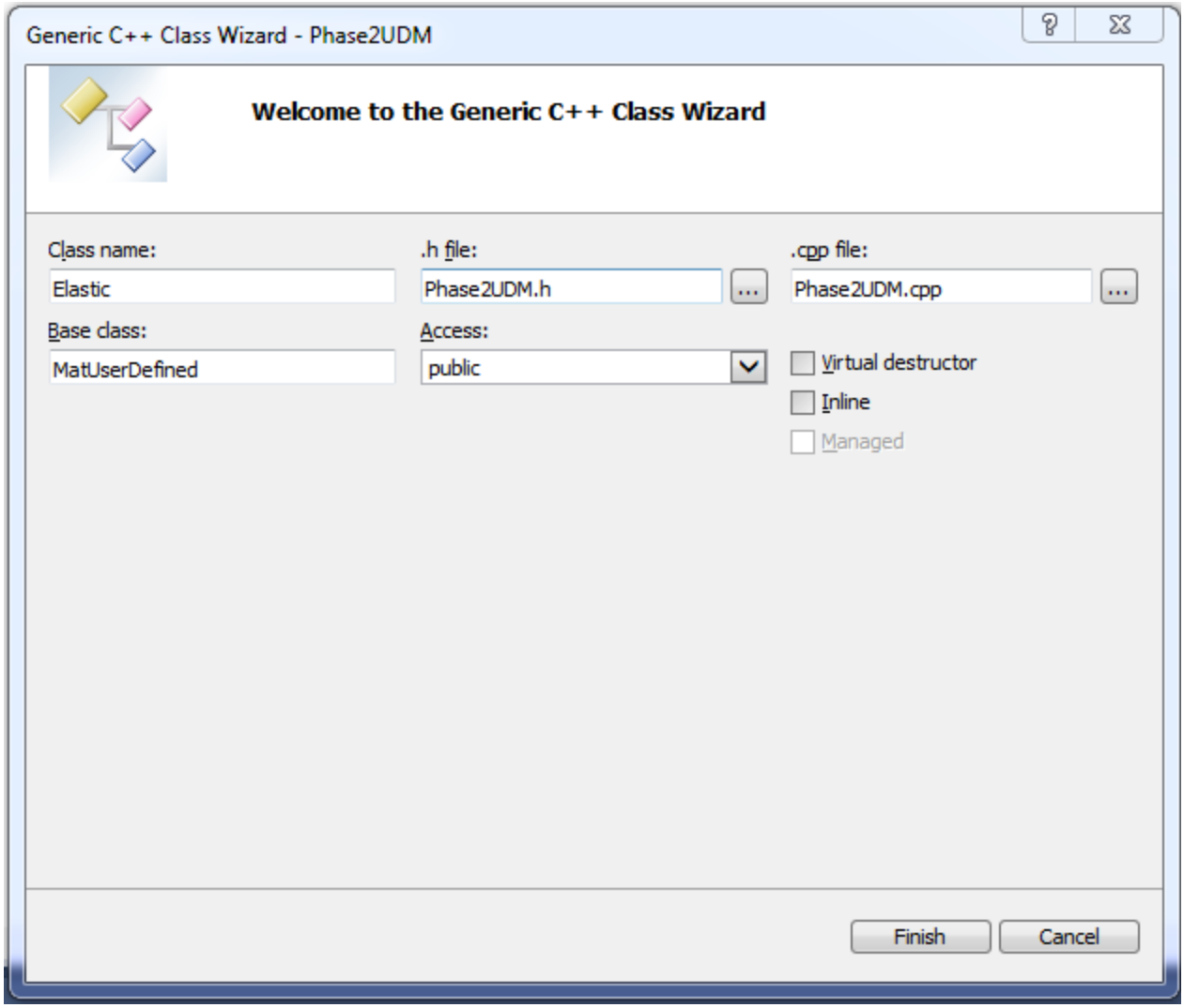
IMPORTANT NOTE: Phase2UDMx64.lib (or Phase2UDMx32.lib) was compiled with the release configuration. If you compile with the debug configuration, you may experience errors.

12.2.3- Phase2UDM.dll implementation

Right click on the project on the left and select Add -> Class



Type Elastic in the Class name and MatUserDefined in the Base class name and click Finish.



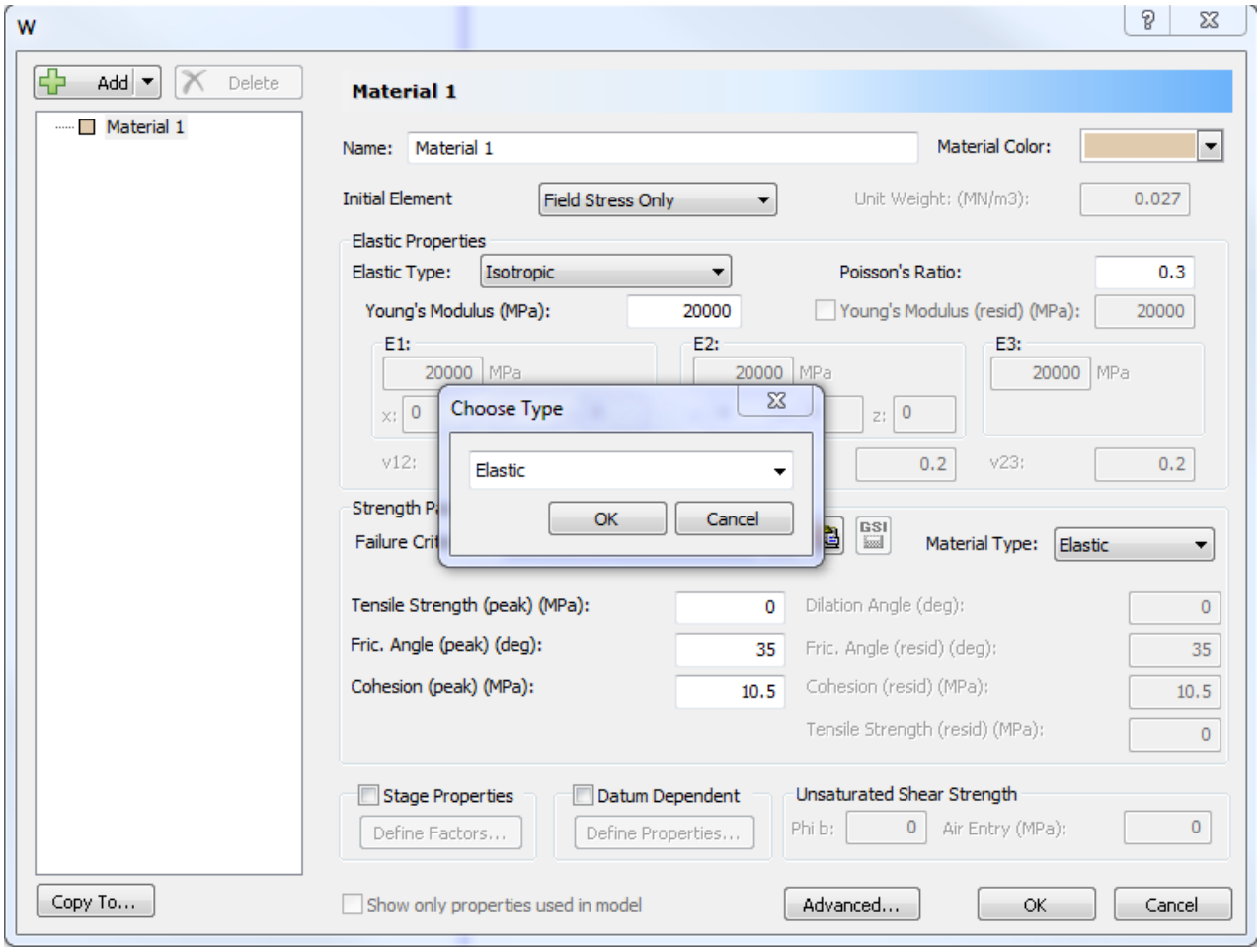
In the header file should include all of the implementation of the Export Function sections. The implementation mentioned in the Class structure should also be included in the header and source files. Now copy the content of the Phase2UDM.h and Phase2UDM.cpp included with the installation into the Phase2UDM.h and Phase2UDM.cpp in the project.

12.2.4- Compilation

Now go to Build->Build Solution. You need to select 64 bit or 32 bit platform depending on the version of Phase2 that you have. Assuming that you have included the linker, you should now be able to compile the program. The result will be a file with extension .dll. As mentioned above, the program may not work properly if you compile in debug mode.

12.2.5- Running

Put your dll file in the Phase2 installation folder. Now open the modeller. Go to Materials & Staging tab, Properties->Define Materials -> Add -> Add User Defined Material. You should see all of the materials that you included in the dlls.



Congratulation! You have created your first user defined model for Phase2 9.0.