Lecture 13 - High Level Design and C++

Prof. Brendan Kochunas

October 13 2022

Contents

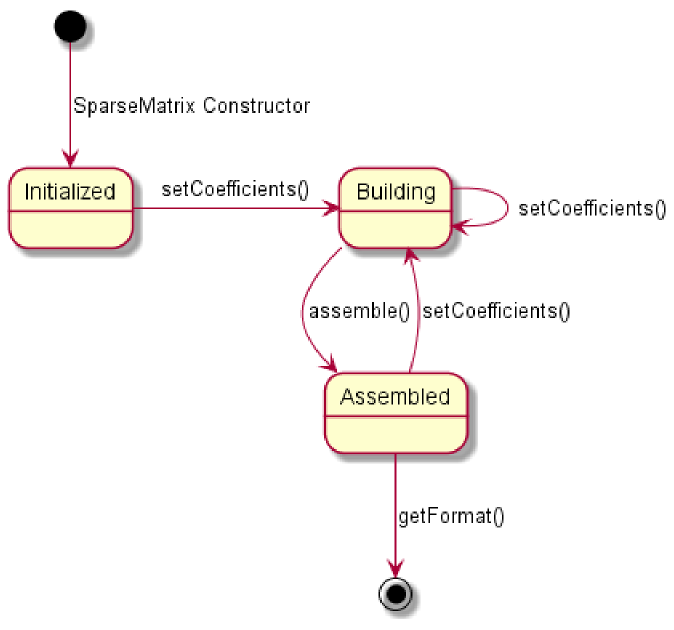


Figure 1: SparseMatrix State Diagram

1 Where we left off from Lecture 12...

Towards the end of this lecture we showed the following diagram as the design for the "state" of the class.

Then, we finished with adding "skeletons" for the additional methods/interfaces. The final version of the code from Lecture 12 is below.

```
#ifndef _SPMV570_
#define _SPMV570_

#include <iostream>
#include <cstddef>

using namespace std;

namespace SpMV
{
    class SparseMatrix
    {
    private:
        size_t _nrows = 0;
        size_t _ncols = 0;
        size_t _nnz   = 0;

    public:
        SparseMatrix(const int nrows, const int ncols);
        ~SparseMatrix();

        void setCoefficient(const size_t row, const size_t col, const double aij);
        void assembleStorage();
        SparseMatrix getFormat();
    };

    SparseMatrix::SparseMatrix(const int nrows, const int ncols) :
        _nrows(nrows), _ncols(ncols)
    {
        cout << "Hello from SparseMatrix Constructor!" << endl;
    }

```

```cpp
    SparseMatrix::~SparseMatrix()
    {
        cout << "Hello from SparseMatrix Destructor!" << endl;
        cout << "this->_ncols=" << this->_ncols << endl;
        cout << "this->_nrows=" << this->_nrows << endl;
        cout << "this->_nnz  =" << this->_nnz   << endl;
    }

    void SparseMatrix::setCoefficient(const size_t row, const size_t col, const double aij)
    {
        cout << "Hello from SparseMatrix::setCoefficient!" << endl;
    }

    void SparseMatrix::assembleStorage()
    {
        cout << "Hello from SparseMatrix::assembleStorage!" << endl;
    }
    SparseMatrix SparseMatrix::getFormat()
    {
        cout << "Hello from SparseMatrix::getFormat!" << endl;

        return SparseMatrix(this->_ncols,this->_nrows);
    }

}

#endif
```

Listing 1: SparseMatrix class with all methods

```cpp
#include <iostream>
#include "SpMV.hpp"

int main(int argc, char* argv[])
{
    std::cout << "Hello World!" << std::endl;

    SpMV::SparseMatrix* ptr_A = new SpMV::SparseMatrix(1,1);

    // New scoping unit. This means variables defined in here, stay here.
    {
        SpMV::SparseMatrix A = SpMV::SparseMatrix(2,2);

        A.assembleStorage();
        A.setCoefficient(5,6, 1.0);

        SpMV::SparseMatrix B = A.getFormat();
    }

    delete(ptr_A);
    ptr_A = NULL;

    return 0;
}
```

Listing 2: Program using our SparseMatrix Class

1.1 Enumerations

In our design of the states for the `class SparseMatrix`, we identified 3 states: `initialized`, `building`, `assembled`. I will assert that it is helpful to define a fourth `unknown` as a catch all (but, strictly, we don't need this right now).

In defining certain states, we may expect to add bits of code like:

```
if( this->state  == something)
{
    //do something
} else if (this->state == somethingElse) {
    //do something else
} else {
    //do another thing
}
```

We *could* define the `state` attribute in the code as simple integer, use values like 1, 2, 3, and 4 for to indicate `initialized`, `building`, `assembled`, and `unknown`. Then our code becomes something like:

```
// ...

    private:
        MatrixState _state

// ...

if( this->_state  == 1)
{
    //do something for initialized
} else if (this->_state == 2) {
    //do something for building
} else {
    //do another thing
}
```

However, this approach is not so good because for anyone reading the code, to understand it, they would first have to understand that you–the original programmer–intended the values 1 and 2 to have a particular meaning. So, a slightly better way to write this code is:

```
const int INITIALIZED=1;
const int BUILDING=2;
const int ASSEMBLED=3;

// ...

    private:
        int _state;

// ...

if( this->state  == INITIALIZED)
{
    //do something for initialized
} else if (this->state == BUILDING) {
    //do something for building
} else {
    //do another thing
}
```

Now for anyone reading the code it is more obvious what branch of code corresponds to which state. Here `const int` means we are defining an integer variable with a constant value. If we try to change it, the compiler will get mad at us. But, we can still improve upon this. Here the values have been hard coded as parameters (another word for constant variables), but we still arbitrarily chose the values of 1, 2, 3, 4. What if someone else uses 1 through 4 to define `const int` variables to mean other things? Or what if we define an `initialized` for a vector class or linear system class and for whatever reason, this was done by another program and they used a different value for `initialized`? Hopefully, you can see how this would create problems.

In fact, the appropriate way to define values for `initialized`, `building`, `assembled`, and `unknown` is to understand that these are *enumerations*. Enumerations (or enumerated types) are basically the solution for representing a "finite set of things" (here we have states of our class) numerically. The way we define enumerated types is:

```


enum MatrixState {undefined,initialized,building,assembled};

// ...

    private:
        MatrixState _state

// ...

if( this->_state  == initialized)
{
    //do something for initialized
} else if (this->_state == building) {
    //do something for building
} else {
    //do another thing
}
```

In this case, the compiler will choose numerical values represent the enumerations of the `MatrixState`. This is a little safer, as the compiler will be smart enough to pick values that will be unique and will not interfere with other enumerations in a large program–and it will understand that the possible values only apply to variables declared to be the enumerated `MatrixState` type. This also prevents programmers from accidentally comparing values of one declared enumerated type with another.

Finally, putting it all together, our updated `SparseMatrix` class looks like:

```
namespace SpMV
{
        enum MatrixState {undefined,initialized,building,assembled};

        class SparseMatrix
        {
        private:
            size_t _nrows = 0;
            size_t _ncols = 0;
            size_t _nnz   = 0;

            MatrixState _state = undefined;

        public:
            SparseMatrix(const size_t nrows, const size_t ncols);
            ~SparseMatrix();

            void setCoefficient(const size_t row, const size_t col, const double aij);
            void assembleStorage();
            SparseMatrix getFormat();
        };
}
```

Listing 3: SparseMatrix class with enumerated states

1.2 Design-By-Contract for MatrixState

There are likely several ways you can implement or use design-by-contract in C++. Certainly, there are folks that have written their own implementation. For us, we will simply rely on the `assert()` function from the C standard library, to use the `assert()` function in C++ code, you need to add `#include <cassert>` to the top of your source file.

The types of assertions we will make are:

```cpp


enum MatrixState {undefined,initialized,building,assembled};

// ...

    private:
        MatrixState _state

// ...

    assert(this->_state  == initialized); //or whichever state value we desire

// ...
```

Based on fig. 1, we should implement the following assertions

```cpp
// ...

    SparseMatrix::SparseMatrix(const size_t nrows, const size_t ncols) :
        _nrows(nrows), _ncols(ncols)
    {
        assert(this->_state == undefined);
        //...
    }

// ...
```

```cpp
// ...

    void SparseMatrix::setCoefficient(const size_t row, const size_t col, const double aij)
    {
       assert(this->_state != undefined);
       // ...
    }

// ...
```

```cpp
// ...

    void SparseMatrix::assembleStorage()
    {
        assert(this->_state == building);
        // ...
    }

// ...
```

```cpp
// ...

    SparseMatrix SparseMatrix::getFormat()
    {
        assert(this->_state == assembled);
        // ...
    }

// ...
```

2 Creating a Class Hierarchy

In this section we'll show how to

1. Convert the `SparseMatrix` class to an abstract class
2. Declare a subclass, `SparseMatrix_COO` of the base class

We'll work from part of previous design that includes the class hierarchy. and the following set of methods

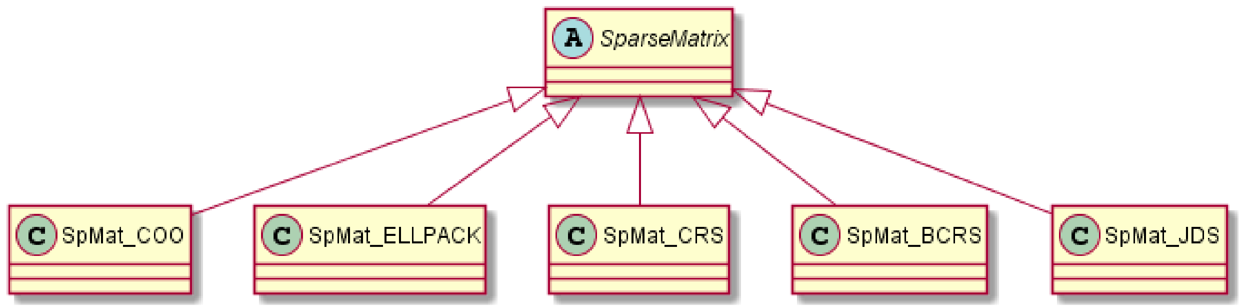
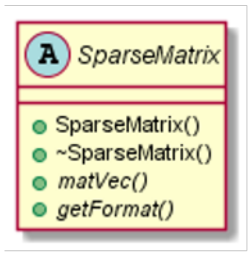


Figure 2: SparseMatrix Class Hierarchy

on our `SparseMatrix` class



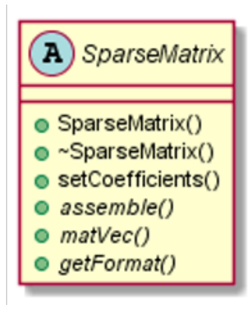


Figure 3: Abstract SparseMatrix class with methods

Declaring an abstract class

In C++, the that an Abstract Class is defined is to merely declare, one of the methods as a *virtual function* (or an abastract method). This is a method without an implementation.

Based on fig. 3, which methods should be abstract?

This results in the following updated class definition.

```
namespace SpMV
{
        enum MatrixState {undefined,initialized,building,assembled};

        class SparseMatrix
        {
        protected:
            size_t _nrows = 0;
            size_t _ncols = 0;
            size_t _nnz   = 0;

            MatrixState _state = undefined;

        public:
            SparseMatrix(const size_t nrows, const size_t ncols);
            virtual ~SparseMatrix();

            void setCoefficient(const size_t row, const size_t col, const double aij);
            virtual void assembleStorage() =0;
            virtual SparseMatrix getFormat(/* args */)=0;
            /* return args?*/ void matVec(/* args */)=0;
        };
}
```

Listing 4: Abstract SparseMatrix Class

The key here is that any function that is to be potentially overridden by the subclass needs to be declared `virtual`. Note here we also made the private attributes `protected` in the base class, which so that they are accessible in the subclass.

Notice that now, our main program is way broken. We'll fix it after declaring our subclass.

Declaring a Subclass

The basic subclass declaration syntax is applied in the class declaration like: `class SparseMatrix_COO : public SparseMatrix`. Here the ":" basically corresponds to UML arrow for a generalization. The public keyword is the *access specifier*. It declares which members of the base class are made public by the subclass.

- With `public`, public members of the base class remain public.
- With `protected`, protected and public members of the base class are protected in the subclass.
- With `private`, private, protected, and public members of the base class are private in the subclass.

```cpp
#include "SpMV.hpp"

namespace SpMV
{
    class SparseMatrix_COO : public SparseMatrix
    {

        private:
                int*    _I = nullptr;
                int*    _J = nullptr;
            double* _val = nullptr;

        public:
            SparseMatrix_COO(const size_t nrows, const size_t ncols) : SparseMatrix(nrows, ncols) {};
            void assembleStorage();
            void getFormat() override;
    };

    void SparseMatrix_COO::assembleStorage()
    {
        cout << "Hello from SparseMatrix_COO::assembleStorage()" << endl;
        cout << "this->_nnz=" << this->_nnz << endl; //Not possible unless baseclass declares _nnz protected
    }

    void SparseMatrix_COO::getFormat()
    {
        cout << "Hello from SparseMatrix_COO::getFormat()" << endl;
    }
}
```

Listing 5: SparseMatrix_COO Subclass

Here we have also mapped the base class constructor to the subclass.

Now we have to update our main program.

```cpp
#include <iostream>
#include <cassert>
#include "SpMV.hpp"
#include "SparseMatrix_COO.hpp"

int main(int argc, char* argv[])
{
    std::cout << "Hello World! " << std::endl;

    //SpMV::SparseMatrix* ptr_A = new SpMV::SparseMatrix(1,2);
    SpMV::SparseMatrix* ptr_A;
    ptr_A = new SpMV::SparseMatrix_COO(1,2);
    //This is a scoping unit, what's defined here, stays here.
    {
        SpMV::SparseMatrix_COO A = SpMV::SparseMatrix_COO(3,4);
        A.assembleStorage();
        A.setCoefficient(5,7,10.0);
        //std::cout << A._nnz << std::endl;
    }

    ptr_A->getFormat();
    delete(ptr_A);
    ptr_A = nullptr;

    assert(ptr_A == nullptr);

    return 0;
}
```

Listing 6: Program using our SparseMatrix and SparseMatrix_COO Classes

3 Templating

The basic idea of templating is that you want to apply the same code (e.g. functions, algorithms, etc.) to different data types. Recall that the high-level languages are strongly typed–meaning the specific data type of each variable is checked for correct usage. As a result, if one wants to write our class for a `double` and a `float` without templating; you would have to effectively write your code for the `double`, copy and paste it and do a find replace of `double` with `float`.

This approach works, but its not ideal. Its not ideal because you have essentially doubled the amount of code you to maintain. If you change something in your implementation on `double`, you have to remember to make the same change for your implementation for `float`. Hopefully its obvious that this situation does not scale.

This problem is exactly the problem that is solved by templating. In that sense, it is kind of like “meta-programming”. It is an extremely powerful feature of the C++ language that does not really exist in C or Fortran. In addition to just the syntax for doing this, the C++ standard also defines the C++ Standard Template Library (STL) with a whole host of capabilities that you can bring to bear on not just intrinsic types, but any classes you might define. Learning the STL can be just as challenging and important as learning the C++ language syntax.

To make use of templating in our code, we first have to modify our class declaration to declare it as a templatable type.

```cpp

//Declares the class as templatable, and fp_type as the template parameter
template <class fp_type>
class SparseMatrix
{

// ...

public:
    SparseMatrix(const size_t nrows, const size_t ncols);
    virtual ~SparseMatrix();

    //Template parameter replaces the use of double here
    void setCoefficient(const size_t row, const size_t col, const fp_type aij);


    virtual void assembleStorage() =0;
    virtual SparseMatrix getFormat(/* args */)=0;
    /* return args?*/ void matVec(/* args */)=0;
};
```

Listing 7: Templated SparseMatrix Class

We also have to make several other changes to our class. Now, every method implementation needs to be redefined to include the template parameter.

So before we had:

```cpp
// ...

    SparseMatrix::SparseMatrix(const size_t nrows, const size_t ncols) :
        _nrows(nrows), _ncols(ncols)
    {
        assert(this->_state == undefined);
        //...
    }

// ...
```

And this now becomes:

```cpp
// ...

    template <class fp_type>
    SparseMatrix<fp_type>::SparseMatrix(const size_t nrows, const size_t ncols) :
        _nrows(nrows), _ncols(ncols)
    {
        assert(this->_state == undefined);
        //...
    }

// ...
```

Effectively we add the line `template <class fp_type>` before the implementation of each method and we modify the class specifier in the function declaration to include the template parameter(s). So, `SparseMatrix::SparseMatrix(...)` becomes `SparseMatrix<fp_type>::SparseMatrix(...)`. Or more generally `ClassName::someMethod(...)` becomes `ClassName<template_parameter>::someMethod(...)`.

For any subclasses that we also want to be templated, we apply similar modifications. We must also make a few other modifications in specifying the inheritance and for how the constructor is defined to use the constructor on the base class.

If the subclass is to be templatable like the base class (as is the case for our `SparseMatrix_COO`) we would modify the class declaration to be:

```
template <class fp_type>
class SparseMatrix_COO : public SparseMatrix<fp_type>
{
    // ...
};
```

And the overloaded constructor is also modified to reference the templated base class like: `SparseMatrix_COO`) we would modify the class declaration to be:

```
template <class fp_type>
class SparseMatrix_COO : public SparseMatrix<fp_type>
{
    // ...

    public:
        SparseMatrix_COO(const size_t nrows, const size_t ncols) :
            SparseMatrix<fp_type>::SparseMatrix(nrows, ncols) {};
};
```

4 A Final Working Implementation

```cpp
#ifndef _SPMV570_
#define _SPMV570_

#include <iostream>
#include <cstddef>
#include <cassert>

using namespace std;

namespace SpMV
{
    enum MatrixState {undefined,initialized,building,assembled};

    template <class fp_type>
    class SparseMatrix
    {
    protected:
        size_t _nrows = 0;
        size_t _ncols = 0;
        size_t _nnz   = 0;

        MatrixState _state = undefined;

    public:
        SparseMatrix(const int nrows, const int ncols);
        virtual ~SparseMatrix();

        void setCoefficient(const size_t row, const size_t col, const fp_type aij);
        virtual void assembleStorage() =0;
        virtual void getFormat();
    };

    template <class fp_type>
    SparseMatrix<fp_type>::SparseMatrix(const int nrows, const int ncols) :
        _nrows(nrows), _ncols(ncols), _nnz(0), _state(initialized)
    {
        cout << "Hello from SparseMatrix Constructor!" << endl;
        assert(this->_state == initialized);
    }

    template <class fp_type>
    SparseMatrix<fp_type>::~SparseMatrix()
    {
        cout << "Hello from SparseMatrix Destructor!" << endl;
        cout << "this->_ncols=" << this->_ncols << endl;
        cout << "this->_nrows=" << this->_nrows << endl;
        cout << "this->_nnz  =" << this->_nnz   << endl;
        this->_state = undefined;
        assert(this->_state == undefined);
    }

    template <class fp_type>
    void SparseMatrix<fp_type>::setCoefficient(const size_t row, const size_t col, const
    fp_type aij)
    {
        assert(this->_state != undefined);
        cout << "Hello from SparseMatrix::setCoefficient!" << endl;

        this->_state = building;
        assert(this->_state == building);
    }

    template <class fp_type>
    void SparseMatrix<fp_type>::getFormat()
    {
        assert(this->_state == assembled);
        cout << "Hello from SparseMatrix::getFormat!" << endl;

        //return new SparseMatrix(this->_ncols,this->_nrows);
    }

}

#endif
```
Listing 8: SparseMatrix.hpp

```
#ifndef __SPMV570_COO__
#define __SPMV570_COO__

#include "SparseMatrix.hpp"

namespace SpMV
{
    template <class fp_type>
    class SparseMatrix_COO : public SparseMatrix<fp_type>
    {

        //using SparseMatrix<fp_type>::SparseMatrix;

        private:
            int* I = nullptr;
            int* J = nullptr;
            double* val = nullptr;

        public:
            SparseMatrix_COO(const size_t nrows, const size_t ncols) : SparseMatrix<fp_type
    >::SparseMatrix(nrows,ncols)
            {
                cout << "Hello From SparseMartix_COO" << endl;
            };
            void assembleStorage();
            /*Some return type*/ void getFormat(/*some args*/);

    };

    template <class fp_type>
    void SparseMatrix_COO<fp_type>::assembleStorage()
    {
        assert(this->_state == building);
        cout << "Hello from SparseMatrix_COO::assembleStorage!" << endl;

        this->_state = assembled;
        assert(this->_state == assembled);
    }

    template <class fp_type>
    void SparseMatrix_COO<fp_type>::getFormat()
    {
        assert(this->_state == assembled);
        cout << "Hello from SparseMatrix_COO::getFormat!" << endl;

        //return new SparseMatrix(this->_ncols,this->_nrows);
    }
}

#endif
```
Listing 9: SparseMatrix_COO.hpp

```cpp
#include <iostream>
#include "SparseMatrix.hpp"
#include "SparseMatrix_COO.hpp"

template class SpMV::SparseMatrix<float>;  //Forces binary code to be generated for float
    data type
template class SpMV::SparseMatrix<double>; //Forces binary code to be generated for double
    data type


int main(int argc, char* argv[])
{
    std::cout << "Hello World!" << std::endl;

    SpMV::SparseMatrix<double>* ptr_A = nullptr;
    SpMV::SparseMatrix_COO<double>* ptr_B = nullptr;

    ptr_A = new SpMV::SparseMatrix_COO<double>(5,8);

    ptr_B = (SpMV::SparseMatrix_COO<double>*)ptr_A;

    // New scoping unit. This means variables defined in here, stay here.
    {
        SpMV::SparseMatrix_COO<double> A = SpMV::SparseMatrix_COO<double>(2,2);

        //A.assembleStorage();
        //A.setCoefficient(5,6, 1.0);

        //SpMV::SparseMatrix B = A.getFormat();
    }

    delete(ptr_A);
    ptr_A = NULL;

    return 0;
}
```

Listing 10: main.cpp