Lecture 13 - Initial Steps to Setting up a New Project on Github

Prof. Brendan Kochunas

October 14 2020

Contents

1 Outline

1.1 Create Skeleton `SparseMatrix` Class

1.2 Create Dummy Constructor/Destructor

1.3 Create Skeleton Continuous Integration (CI) Testing

1.4 Setup Simple CI

2 Developing Skeleton SparseMatrix Class

1. Write out the following code:

```
namespace  SpMV
{

template <class fp_type>
class SparseMatrix
{
    private:
        int _nrows = -1;
        int _ncols = -1;
        int _nnz   = -1;

    public:
        //This is the constructor
        SparseMatrix(int nrows, int ncols);
        //This is the destructor
       ~SparseMatrix();
};

}

```

Listing 1: Initial `sparsemat.hpp`

```
#include "sparsemat.hpp"

```

Listing 2: `sparsemat.cpp`

   This includes the basic attributes and defines the interfaces for the constructor and destructor. It does not define the constructor and destructor operations. We choose to namespace our class because we might reasonably assume that our library or someone that choose to use our library might also define a class called `SparseMatrix`. We follow some typical conventions for naming. Our private variables are preceded by an underscore. The constructor and destructor names are special. We do not use the template variable `fp_type` yet, but we will (I think).

2. Next consider if `int` is the best data type for our sizes. In fact `size_t` is better. It is better here because it is a platform independent type meant to represent the size of an object/variable etc. `size_t` should be used for the attributes and constructor args. To use `size_t` we must include the header `cstddef`.

3. Add the `using namespace std;` to prevent using `std::` everywhere. `using namespace` is like `import` in Python or `USE` in Fortran 90 (or greater).

4. Lastly, we might improve the constructor interface by using `const`. Using `const` here basically means that the constructor will not modify the arguments. They are "read only" in the constructor. This is equivalent to Fortran's `INTENT(IN)` clause on dummy argument variable declaration.

```cpp
#include <cstddef>

using namespace std;

namespace  SpMV
{

template <class fp_type>
class SparseMatrix
{
    private:
        size_t _nrows = -1;
        size_t _ncols = -1;
        size_t _nnz   = -1;

    public:
        //This is the constructor
        SparseMatrix(const size_t nrows, const size_t ncols);
        //This is the destructor
       ~SparseMatrix();
};

}
```

Listing 3: Final Code `sparsematrix.hpp`

3 Create Dummy Constructor/Destructor

Previously, in the header file we only defined what the interface to the constructor and destructor should look like. The header file does not define their actual implementations (although it could). By defining these interfaces in the `class`, we accomplish two things:

1. We establish a "contract" that *any implementation* of the class or any subclass *must* provide this routine in this form.

2. Like with any function routine defined in a header file, it allows the C++ language compile a file that uses this interface (e.g. in the C++ BLAS example we had to define the routine interface to dgemm so we could compile the source, and then link it appropriately).

For these dummy/stub routines, we simply put in print statements so that when we use them later, we'll be able to follow the the call sequence of the program. Calls to the constructor and destructor can be a little difficult to understand initially because they do not look like other routine calls. These routines, in a sense, are called "implicitly".

```
#include <iostream>
#include "sparsemat.hpp"

using namespace std;
using namespace SpMV;

//Defines the Constructor Operation
template <class fp_type>
SparseMatrix<fp_type>::SparseMatrix(const size_t nrows, const
    size_t ncols)
{
    cout << "Called SpMV::SparseMatrix<fp_type> constructor!" <<
    endl;
    _nrows = nrows;
    _ncols = ncols;
}



//Defines the Destructor Operation
template <class fp_type>
SparseMatrix<fp_type>::~SparseMatrix()
{
    _nrows = -1;
    _ncols = -1;
    cout << "Called SpMV::SparseMatrix<fp_type> destructor!" <<
    endl;
}

}
```

Listing 4: Stub Routines `sparsematrix.cpp`

4 Setup Simple CI

Travis-CI is one of a few free services for CI testing. It's very easy to have it integrated with a Github project.

1. The first step is to just create an empty `.travis.yml` file in your main or master branch.

2. If you'd like, in the previous step you can specify an os and language.

```
language: cpp
os: linux
script:
  - echo "Hello World!"

```

Listing 5: Most Basic travis-ci config

3. A good next step, is to learn something about you CI environment. This can be accomplished with a script like the following:

```
language: cpp
os: linux
script:
  - echo $PWD
  - ls -ahl
  - printenv
# you could add further steps to check for programs you need

```

Listing 6: Most Basic travis-ci config

4. The final form that we need for our project includes configuring with cmake and making sure there's a valid c++ compiler.

```
language: cpp
os: linux

script:
  - echo $PWD && ls -ahl && mkdir ../bld && cd ../bld && echo
    $PWD
  - cmake ${TRAVIS_BUILD_DIR}
  - make

```

As one should expect the configuration for travis-ci can become quite complex. It is recommended you consult their documentation (which is so so) for further info. Some useful links:

- Instructions on connecting travis-ci to a github repo
- Supported languages
- Supported environments
- Concepts for the "Job Lifecycle" of a CI script
  - A CI script defines a *build*
  - A *build* is a sequence of *stages*
  - A *stage* is composed of *job*s (which run in parallel)
  - A *job* is a sequence of *phases*

A full reference for the config is here: `https://config.travis-ci.com/` You can also test configs on this page: `https://config.travis-ci.com/explore`