Writing your own linear algebra matrix library in C

Andrei Ciobanu

January 20, 2021

1 The library

Now, let's say you are one of the few Engineering/Computer Science students that are passionate about *linear algebra*, *numerical analysis* and writing code in a low-level language. Or you are just curious about what's behind the `lu(A)` method in Matlab. Or you are passionate about A.I., and you know you cannot learn A.I. algorithms without a good foundation in linear algebra.

I believe the best exercise you can do is to try to write your (own) Matrix library in a low-level programming language (like C, C++ or even D).

This tutorial is precisely this, a step-by-step explanation of writing a C Matrix library that implements the "basic" and "not-so-basic" numerical analysis algorithms that will allow us in the end to solve linear systems of equations.

All code in this tutorial is available on GitHub in the repository called `neat-matrix-library`.

To clone it (using GitHub CLI):

```
gh repo clone nomemory/neat-matrix-library
```

The code is not meant to be "efficient", but relatively easy to follow and understand.

The tutorial assumes that the reader can write C code, understand pointers and dynamic memory allocation, and is familiar with the C standard library.

2 The data: `nml_matrix`

The first step is to model the data our library will work with: "matrices". So we are going to define our first struct, called `nml_mat` which models a matrix:

```
typedef struct nml_mat_s {
  unsigned int num_rows;
  unsigned int num_cols;
  double **data;
  int is_square;
} nml_mat;
```

The properties of this `struct` have self-explanatory names:

- `unsigned int num_rows` – represents the number of rows of the matrix. 0 is not an acceptable value
- `unsigned int num_cols` – represents the number of columns of the matrix. 0 is not an acceptable value
- `double **data` – is "multi-dimensional array" that stores data in rows and columns
- `int is_square` – is a "boolean" value that determines if the matrix is square (has the same number of rows and columns) or not.

From a performance perspective, it's better to keep the matrix elements in a `double*` using the conversion:

```
data[i][j] <=> array[i * m + j]
```

To better understand how to store multi-dimensional arrays in linear storage please refer to *this StackOverflow question*, or *read the wikipedia article* on the topic.

Even if it might sound like a "controversial" decision, for the sake of simplicity, we will use the double ** multi-dimensional storage.

3 Allocating/deallocating memory for the tt nml_mat matrix

Unlike "higher-level" programming languages (Java, Python, etc), that manage memory allocation for you, in C, you need to explicitly ask for memory and explicitly free the memory once you no longer need it.

In this regard, the next step is to create "constructor-like" and "destructor-like" functions for the `nml_mat struct` defined above. There's an unwritten rule that says: "Every `malloc()` has its personal `free()`".

```
// Constructor-like
// Allocates memory for a new matrix
// All elements in the matrix are 0.0
nml_mat *nml_mat_new(unsigned int num_rows, unsigned int num_cols);

// Destructor-like
// De-allocates the memory for the matrix
void nml_mat_free(nml_mat *matrix);
```

Implementing the `nml_mat_new()` is quite straightforward:

```
nml_mat *nml_mat_new(unsigned int num_rows, unsigned int num_cols) {
  if (num_rows == 0) {
    NML_ERROR(INVALID_ROWS);
    return NULL;
  }
  if (num_cols == 0) {
    NML_ERROR(INVALID_COLS);
    return NULL;
  }
  nml_mat *m = calloc(1, sizeof(*m));
  NP_CHECK(m);
  m->num_rows = num_rows;
  m->num_cols = num_cols;
  m->is_square = (num_rows == num_cols) ? 1 : 0;
  m->data = calloc(m->num_rows, sizeof(*m->data));
  NP_CHECK(m->data);
  int i;
  for(i = 0; i < m->num_rows; ++i) {
    m->data[i] = calloc(m->num_cols, sizeof(**m->data));
    NP_CHECK(m->data[i]);
  }
  return m;
}
```

Notes:

- `NML_ERROR`, `NP_CHECK` are macros defined in `nml_util.h`.
- `NML_ERROR()` or `NML_FERROR()` are logging utils, that helps us print error message on stderr;
- `NP_CHECK` checks if the newly allocated memory chunk is not `NULL`. In case it's `NULL` it aborts the program.

Explanation:

1. First step is to check if `num_rows == 0` or `num_cols == 0`. If they are, we consider the input as invalid, and we print on stderr an error. Afterwards `NULL` is returned;
2. This line: `nml_mat *m = calloc(1, sizeof(*m))` allocates memory for 1 (one) `nml_mat structure`;
3. For a multi-dimensional array (`double**`), we allocate memory in two steps:
   - `m->data = calloc(m->num_rows, sizeof(*m->data))` - this allocates memory for the column array;

  - ◦ Then, we allocate memory for each row. By using `calloc()` the data is initialized with 0.0.

Freeing the matrix is even more straightforward. The implementation for `nml_mat_free()`:

```
void nml_mat_free(nml_mat *matrix) {
  int i;
  for(i = 0; i < matrix->num_rows; ++i) {
    free(matrix->data[i]);
  }
  free(matrix->data);
  free(matrix);
}
```

It's important to note, that given the multidimensional nature of `double**` data, we need to:

- de-allocate each row individually: `free(matrix->data[i]);`
- then the column vector: `free(matrix->data);`
- and lastly the actual struct: `free(matrix)`.

At this point, it's a good idea to add more methods to help the potential use of the library to create various `nml_mat structs`, with various properties.

Method	Description
`nml_mat_rnd()`	A method to create a random matrix.
`nml_mat_sqr()`	A method to create a square matrix with elements 0.0.
`nml_mat_eye()`	A method to create an identity matrix.
`nml_mat_cp()`	A method to copy the content of a matrix into another matrix.
`nml_mat_fromfile()`	A method to read the matrix from a FILE.

```
#include <algorithm>
...

using namespace TNT;
...

Array1D<double> v(10);

// 1st way
for (Array1D<double>::iterator it = x.begin(); it != v.end(); it++) {
    std::cout << x << "\n";
}
// 2nd way
{auto const &x: v) {
    std::cout << x << "\n";
}
```

Also, trying to achieve this efficiently, without a significant lack of performance.

4 Approach

The class hierarchy followed to reach a reasonable design of an iterator that meets the requirement demanded by the C++11 standard (`http://www.cplusplus.com/reference/iterator/`), was

```
                +----------------------+
                |    basic_iterator    |
                +----------------------+
                            |
            ----------------+----------------
           |                                 |
+----------------------+         +----------------------+
|   input_iterator     |         |    output_iterator   |
+----------------------+         +----------------------+
           |                                 |
           |                     +----------------------+
           |                     |   forward_iterator   |
           |                     +----------------------+
           |                                 |
+----------------------+         +------------------------+
| const_bidirectional_ |         | bidirectional_iterator |
| iterator             |         +------------------------+
+----------------------+                     |
           |                                 |
+----------------------+         +--------------------------+
| const_random_access_ |         | random_access_iterator   |
| iterator             |         +--------------------------+
+----------------------+
```

Some guidance to the build these classes was also taken from `https://www.internalpointers.com/post/writing-custom-iterators-modern-cpp`.

5 Implementation

The `basic_iterator` is the parent class for all iterators, and it defines the minimum methods and properties any iterator has to accomplish. This class encapsulates an internal, protected, member what is a regular pointer to the target data type.

The `input_iterator` is supposed to be only able to read from the array, so it retrieves a constant reference to the pointed element (i.e., by expressions like `*it`). The output_iterator is able to read/write, so it retrieves normal references.

Also, the `basic_iterator` have to be *copy-constructible*, *copy-assignable*, *destructible* and *swappable*. Hence, we have to define the copy-constructor, the assignment operator, the default destructor, and the `swap` method:

```
/** Base class to all the TNT iterators */
template<class T>
class basic_iterator
{
public:
  // data types:
  typedef  T            value_type;
  typedef  T            element_type;
  typedef  T &          reference;
  typedef  T *          pointer;
  typedef  T const &    const_reference;
  typedef  T const *    const_pointer;

  // Empty/null constructor
  basic_iterator() : ptr_(NULL) {}
  // Cast a C pointer into an iterator (pointer-constructor)
  basic_iterator(pointer ptr) : ptr_(ptr) {}
  // Copy constructor
  basic_iterator(const basic_iterator &it) : ptr_(it.ptr_) {}
  // Destructor
  ~basic_iterator() {}

  // cast a regular pointer to iterator
  void from(pointer ptr);

  // assignment operator: set the left-hand side variable to be equal to the one in the righ
  basic_iterator &operator=(const basic_iterator &other);
  // exchange the content between two iterators (this is used by the assignment operator)
  void swap(basic_iterator &other);
  // equality/inequality operator
  bool operator==(const basic_iterator &other) const;
  bool operator==(const pointer &ptr) const;
  bool operator!=(const basic_iterator &other) const;
  bool operator!=(const pointer &ptr) const;

protected:
  pointer ptr_;    // internal pointer
};// end of basic_iterator interface
```

It is also required that the `iterator` class define some public datatypes, like for example, `value_type`, which is later used by the standard library to read the *iterator_traits*.

The `forward_iterator` is a specialization of `output_iterator` that allows increment operator `(++)`, both in prefix and postfix versions. The `bidirectional_iterator` extends the bidirectional one, to incorporate the decrement operator `(--)` in both versions.

The `random_access_iterator` is the most complete kind of iterator, and it also is the default class for `Array1D<T>::iterator`. That is, when we invoke an iterator for an `Array1D<T>`, it will be a `random_access_iterator`. This class allows arbitrary increments/decrements by arithmerical operators `(+)`, `(+=)`, `(-)`, `(-=)`.

Finally, the `const_forward_iterator`, `const_bidirectional_iterator` and `const_random_iterator` are similar to their non-const versions, but descending from `input_iterator` as this class is limited to read-only accesses. The `const_random_access_iterator` is the default class for the `Array1D<T>::const_iterator`.

It is also worth noting that the non-const versions of each iterator has to be cast-able to the respective const version (but not vice versa).

```
class forward_iterator {
  ...
  // explicit cast to const_forward_iterator
  explicit operator const_forward_iterator<T> () const;
  ...
}
```

6 The copy-constructor and the assignment operator (=)

In the sentences `it2(it)`, and `it2 = it`, both methods the copy-constructor and the `operator=` are making a copy of the argument object `it`. So, to avoid duplicity of code, we can make a method dependent of the another. In this approach, it was chosen to make the assignment operator dependent of the copy-constructor, that is, the `operator=()` is set to call the copy-contructor.

```
template<class T>
basic_iterator<T> &basic_iterator<T>::operator=(const basic_iterator<T> &other) {
  basic_iterator<T> aux(other);
  this->swap(aux);
  return *this;
}
```

Basically, this code creates a new copy of `other` in `aux`, then exchange the content of both, `*this` and `aux`, and return `*this`. The object pointed by `this` will contain a copy of `other` (this is what we was looking for), and the old content of `*this` will be stored in `aux`, which will be consequently destroyed after returning.

This implementation illustrates the need for including a `swap` method in the definition of iterators.

7 Examples of use

7.1 Raw iterators

The class iterator was developed with a minimalist philosophy, to solely work over a raw sequence of data of the same type, e.g., a C array. To this purpose, a constructor is available to cast a regular pointer to an iterator (with the `Array1D` class, the `begin()` method would be available).

```
#include <iostream>
int v[10];

for (int i = 0; i < 10; i++) {
    v[i] = i+1;    // v = {1, 2, 3, ..., 10}
}

// example 1
TNT::forward_iterator<int> it = &(v[0]);    // or, ‘v‘ instead of ‘&(v[0])‘
std::cout << *it << std::endl;              // 1;
++it;
std::cout << *it << std::endl;              // 2;
++it;
std::cout << *it << std::endl;              // 3;

// example 2
TNT::random_access_iterator<int> it2 = v;
std::cout << *it2 << std::endl;              // 1;
it2 += 9;
std::cout << *it2 << std::endl;              // 10;
it2 = it2 - 5;
std::cout << *it2 << std::endl;              // 5;
```

7.2 Iterators with `Array1D`

The class `TNT::Array1D<T>` was conveniently modified (see file ‘`tnt_array1d_.h`’) to incorporate public data types for iterators:

```
/* iterators */
typedef TNT::random_access_iterator<value_type> iterator;
typedef TNT::const_random_access_iterator<value_type> const_iterator;
//
```

```
typedef TNT::input_iterator<value_type>         input_iterator;
typedef TNT::output_iterator<value_type>        output_iterator;
typedef TNT::forward_iterator<value_type>       forward_iterator;
typedef TNT::bidirectional_iterator<value_type> bidirectional_iterator;
typedef TNT::random_access_iterator<value_type> random_access_iterator;
//
typedef TNT::const_forward_iterator<value_type>       const_forward_iterator;
typedef TNT::const_bidirectional_iterator<value_type> const_bidirectional_iterator;
typedef TNT::const_random_access_iterator<value_type> const_random_access_iterator;
```

The data type for `Array1D<T>::iterator` is `random_access_iterator<T>`, and the one for `Array1D<T>::const_iterator` is `const_random_access_iterator<T>`, to be conformant with the C++ standard.

The class `Array1D<T>` also now incorporates methods `begin()`, `end()`, `cbegin()` and `cend()` with the same meaning as the standard ones for the `vector` class in STL. The non-const versions retrieve `Array1D<T>::iterator`, and the const versions retrieve `Array1D<T>::const_iterator`

Example:

```
#include <iostream>
using namespace TNT;

Array1D<int> v(10);    // TNT array

for (int i = 0; i < 10; i++) {
    v[i] = i+1;    // v = {1, 2, 3, ..., 10}
}

// example 1
TNT::Array1D<int>::iterator it = v.begin(); // points to v[0]
std::cout << *it << std::endl;              // 1;
++it;
std::cout << *it << std::endl;              // 2;
++it;
std::cout << *it << std::endl;              // 3;

// example 2
TNT::Array1D<int>::iterator it2 = v.end() - 1; // points to v[9]
std::cout << *it2 << std::endl;              // 10;
it2 -= 2;
std::cout << *it2 << std::endl;              // 8;
it2 -= 3;
std::cout << *it2 << std::endl;              // 5;
```

```
// example 3: C++ iterators style
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << std::endl;           // will print 1 to 10
}
for (auto it = v.end() - 1; it != v.begin() - 1; --it) {
    std::cout << *it << std::endl;           // will print 10 to 1
}
```

7.3 TNT arrays, plus iterators, plus STL

Having created an `iterator` class, accomplishing the requirements of C++11 standard, this should be enough to work along with any of the algorithmic procedures in the STL.
Allowing, for example, to further develop template algorithms for handling Data Structures, etc., and this way extending the `TNT` library.

Example:

```
#include <iostream>
#include <algorithm>
using namespace TNT;

Array1D<int> v(10);    // TNT array

// example 1: looping over array
for (auto &x : v) {
    std::cout << x << std::endl;           // will print 1 to 10
}
for (auto &x : v) {
    x * = 2;                               // doubling elements in v
}

// example 2: looping array, using std::for_each
std::for_each(v.begin(), v.end(), [](int x) {std::cout << x << std::endl;} );

// example 3: populating array, using std::fill
std::fill(v.begin(), v.end(), 3);

// example 4: doubling values, using std::transform
std::for_each(v.begin(), v.end(), v.begin(), [](int x) {return 2*x;} );

// example 5: sorting array, using std::sort
std::sort(v.begin(), v.end());
```

8 Testing

Test programs were designed to check a variety of features about the defined classes. All the content is into the folder `'testing'`. The file `'iterator.cpp'` is to test the class `iterator` merely, that is, over regular C arrays. The file `'Array1D.cpp'` is to test the iterators applied to the `TNT` arrays. Here, we can comprobate several types of constructors, assignment operator, increments and decrements, as well as the behavior of the `TNT::Array1D::iterator` working together with the STL algorithms.

A `makefile` was built to automate the process of compilation. So, first move to the testing directory

```
~$ cd testing
```

and then

```
~$ make
```

or (if you have garbage files from an old compilation)

```
~$ make clean; make
```

will produce two executables: '`iterator`' and '`Array1D`'. Once done this, the commands

```
~$ ./iterator
~$ ./Array1D
```

are to run each of these tests. Note that you can define two constants, `N_ELEM` and `MAX_NUMBER` to control the number of (random) elements in the datasets, and the range of those.
These constants are default to `N_ELEM=10`, and `MAX_NUMBER=20`, but you can change them by editing the makefile, or directly as for example

```
~$ g++ -o iterator iterator.cpp -DN_ELEM=100
```

References

[1] The `TNT` home page.
`https://math.nist.gov/tnt/index.html`.

[2] Documentation for `TNT::Array1D`
`https://math.nist.gov/tnt/tnt_doxygen/class_TNT__Array1D.html`

[3] An article describing how to write your own `iterator` class.
`https://www.internalpointers.com/post/writing-custom-iterators-modern-cpp`

[4] A post with an overview of C++ iterators.
`https://www.learncpp.com/cpp-tutorial/stl-iterators-overview/`

[5] The standard reference for C++ iterators.
`http://www.cplusplus.com/reference/iterator/`