

Laboration 1

Introduction: matrices

Introduction

This is an introductory laboration that *does not* have to be handed in. I still recommend that you do it, though, to get started using C++. If you are short of time, you of course shouldn't feel that you have to complete the entire laboration.

The task is to create a class for matrices that can read from and write to files, check index bounds, and do simple arithmetic operations. The purpose of the laboration is twofold. You will learn how to write a class specification and how to implement this specification. The other purpose is to learn how to organise the programming work, what files are being used, how the program `make` is used, and how to compile programs.

While completing the laboration, it is recommended that you answer the questions in this text. The answer can generally be found in the course literature or in the lecture notes. If there is something you don't understand or cannot answer yourself, ask the reaching assistants or the lecturer.

Preparation

Always load the course specific module before beginning work:

```
module load oopk_e00
```

Make a new directory there you can put your files and make it the current directory. In the laboration you will complete some code fragments that can be found in the course directory. Copy these files and a file for `make`:

```
cp $OOPKHOME/lab1/* .
```

A first look at the files

Start by load the file `mat.hh` into Emacs. This file contains the interface for the matrix class you are about to create. Most of the member functions (methods) are commented out. You are going to remove the comments one by one and implement the corresponding functions in the file `mat.cc`.

- *The beginning of `mat.hh` reads*

```
#ifndef MAT_HH
#define MAT_HH
```

What's the use of this construction?

In the class `Matrix` only one of the methods is declared initially: `Matrix()`, the constructor. The constructor is in the `public` part and is therefore accessible from outside the class (e.g., from the application program). There are also three variables that are members of the class: `rows_`, `cols_` och `elements_`. Since there are in the `private` part, they can only be accessed by the class's methods. The underscore at the end of the names is a convention we use in the course for private member variables.

The idea is that `rows_` contains the number of rows in the matrix, and that `cols_` contains the number of columns. `elements_`, which has the type `vector<double>` (a class from the standard library) will contain the matrix elements. By using a predefined class for the elements you don't have to bother with memory allocation and other related issues. The only thing you have to do is to figure out the mapping between positions in the matrix and indices for the `elements_` vector. The mapping from matrix coordinates to vector coordinates can be done in several ways, and it is up to you to decide how. Also decide if the matrix elements are to be indexed from (0, 0) or (1,1).

The methods in the class should be implemented in the file `mat.cc`. Initially, only a constructor is implemented.

- *Implement the functions (methods) `rows()` and `cols()`. When you are done you should be able to*

compile and run the program in the file `lab1.cc`.

- What is a member function that returns the value of a member variable called?
- What does the program in `lab1.cc` do?

Compile by running `make`. If there are no error messages when `make` is finished you can run the program by typing `./a.out`.

- Did the program behave as expected?
- What file did `make` use during the compilation and linking?
- Study this file. What line would you need to change if you wanted to compile additional source files?
- Implement both variants of `get_element`. The functions should return the value of the matrix element (i, j) , and a reference to the element, respectively.

The functions should also check that `i` and `j` are within allowed limits. Use the macro `assert` for this check. If you write

```
assert(condition);
```

in the source code, the condition will be evaluated. If the result is `false` the program will be aborted.

Output method

To simplify testing of the rest of the methods it is handy to be able to print the matrix' contents from the code. A 3x2 matrix can, for example, be printed like

```
3 2
11 12
21 22
31 32
```

The integers on the first line tell the matrix dimensions. The rest of the lines contain the matrix elements (floating point numbers). The method `print` takes a reference to an object of the type `ostream` as a parameter. Since `cout` (standard output) is such an object, you can call `print` like

```
m1.print(cout);
```

if `m1` is a `Matrix`.

- Implement the method `print`. Add a few lines in the main program that calls the method and verify the functionality.

Constructors

Each time a new object is created, also if only a temporary object, one of the class's constructors is being used. The constructor is used to initialize the object.

- The matrix' default constructor (the constructor that is called without parameters) in `mat.cc` makes matrices of fixed size, which is not very useful. Therefore, write a constructor that takes two integer parameters and creates a matrix of the corresponding size. Add some code to the main program to test the new constructor.
- Also write a constructor that takes the number of rows, the number of columns, and an array of values as parameters. Test this one as well.

In many contexts, a constructor that creates a copy of an existing object of the same type is needed. Constructors doing this are usually called copy constructors. If it is a member of the class `X`, it has a parameter of the type `const X&`. The corresponding argument is the object to be copied:

```
Matrix m1(3, 2);
Matrix m2(m1); // make a copy of m1
```

If you don't write a copy constructor, the compiler will automatically create a constructor that performs "standard copying." The question is:

- *Do we need to implement a copy constructor in this case, and if so why?*

Operators

It is a bit of a hassle having to type

```
m1.get_element(2, 1)
```

each time a matrix element is accessed. More convenient would be to write `m1(2, 1)`. This is possible to accomplish if we rename `get_element(int i, int j)` to `operator()(int i, int j)`. Even if the name is a bit peculiar, the method can be called as usual:

```
double d=m1.operator()(2, 1);
```

The point, though, is that the compiler also accepts the more compact syntax

```
double d=m1(2, 1);
```

- *Implement both variants of `operator()` that can be found in the class specification. Add some test code to main and try it.*

const deklarations

As you can see in the class specification the word `const` follows some of the member functions. It means that those functions promise not to alter the object (`*this`) in any way when called. The the application programmer has declared that an object can not be altered (by declaring it as a `const` object), the compiler ensures that only `const` declared member function are allowed to act on the object.

Matrix addition

You are now about to write two functions to add matrices. The first, `add_to_mat`, changes the matrix elements by adding values from another matrix. The other function, `add`, creates a new matrix that contains the sum of both parameters:

```
Matrix m1(...);
Matrix m2(...);
m1.add_to_mat(m2); // Adderar värden från m2 till elementen i m1.
Matrix m3=m1.add(m2); // Varken m1 eller m2 påverkas.
```

- *Implement `add_to_mat` and try it.*
- *Implement `add`. Can you use `add_to_mat` which is implemented already?*
- *To achive a more convenient syntax, implement `operator+=` and `operator+` so that it is possible to write*

```
Matrix m1(...);
Matrix m2(...);
m1+=m2; // Adds values from m2 to the elements in m1.
Matrix m3=m1+m2; // Neither m1 nor m2 is affected.
```

Notice that `operator+=` returns the object (`*this`). This is to allow for the construction

```
m1 += m2 += m3;
```

which works like for the built-in types.

Assignments

When the code

```
Matrix m1(...);
Matrix m2(...);
m2=m1;
```

is executed, `m2` is assigned the values from `m1`. Notice that `m2` exists already, and is initialized by its cons-

structor when it is being assigned to. The assignments *alters* the value of `m2`. This is taken care of by the assignment operator `operator=`. Like for the copy constructor, the assignment operators is generated automatically by the compiler if it is absent.

- *Is the automatically generated assignment operator sufficient in this case? Why?*

The destructor

Each time an object is removed its destructor is called. The purpose is to clean up any memory or other resources being held by the object. The destructor has to release all such resources allocated by the object. For example, the destructor normally has to `delete` everything that the constructor allocated by `new`. The destructor should not, however, release the memory occupied by the object itself. (The destructor does in fact not know if the object was allocated on the stack or on the heap.) The destructor should merely undo whatever is done in the constructor.

- *Do we need a destructor for `Matrix`?*

Write a destructor that just outputs one line of text to `cerr`, e.g., "destructor called", and try the program again.

- *How many times was the destructor called? Was this what you had expected?*

Reading from a file

The function `read` in the class specification reads matrix elements from a file (e.g., `cin`) and puts them into an existing matrix:

```
Matrix m1;
m1.read(cin);
m1.print(cout);
```

`read` assumes that the input data is given in the format used by `print` when printing. The file `testdata` contains an example. Consider that `m1` in the example above already exists and has some fixed size when `read` is called. The size given by the file could very well be larger than that. Make sure that the matrix size is set correctly in this case. Also consider that the data file may be written in an illegal format and that the `read` operation may fail. Check that you can read the size and all elements before changing the size of the matrix.

- *Implement the `read` method. Test it using the file `testdata`.*

Of course, it is more convenient to be able to use the syntax

```
Matrix m1, m2;
cin >> m1 >> m2;
cout << m1 << m2 << m1+m2;
```

for reading and writing matrices. To be able to do this, we need to define the functions `operator<<` and `operator>>`. The natural thing to do would have been to add them to the class `Matrix`. This does not work, however, which can be seen by examining the following expressions:

```
m1=m2; // corresponds to m1.operator=(m2)
cout << m1; // corresponds to cout.operator<<(m1)
```

As can be seen, the object to the left of `=` and `<<` respectively has to belong to the same class as the function. This means that `operator<<` in this case has to be added to the class `ostream`. This is, however, a system class that we cannot change. The solution is, therefore, to write `operator<<` as an ordinary function, outside of all classes. It is declared as

```
ostream& operator<<(ostream& os, const Matrix& m);
```

The function should return the parameter `os`, so that several calls can be made using only one expression (as in the example above).

- *Implement `operator<<` and `operator>>`. Use `print` and `read` that you have already implemented.*
- *Why does `operator<<`, but not `operator>>`, take a constant reference to a matrix as a param-*

ter?

Template

We have so far assumed that the matrix elements are of the type `double`. Sometimes, matrices of other types are needed, e.g., for the types `long double` and `complex<double>`.

- *Rewrite the matrix class into a template class. It should still have the same interface..*