Simple matrix class

I have built a 3D matrix class in C++, mainly for data I/O, for a project I am working on. The code works ok, but I would like you to help me improve it.

1. Are there any bad programming practices I am doing?
2. Is there any code that could be simplified?
3. Is there something that would give me fastest access to the data stored in the object? 4. Is there something that could lead me to error in the future?

Grid3d.h:

```
#ifndef _GRID3D_
#define _GRID3D_

#include <iostream>
#include <cmath>
#include <iomanip>  // setprecision()
#include <cassert>  // assert()

using namespace std;

class Grid3d
{
private:
    int L;
    int M;
    int N;
    double *** G;

public:
    //Constructors
    Grid3d(int,int,int);
    Grid3d();
    Grid3d(const Grid3d &);
    ~Grid3d();

    //Operators
    double & operator()(int,int,int);
    Grid3d & operator=(Grid3d);

    //Access
    int get_L();
    int get_M();
    int get_N();
    double get(int,int,int);
    void clean();

    void swap(Grid3d &);
    friend std::ostream & operator<< (std::ostream &,Grid3d &);
};

#endif
```

Grid3d.cc:

```
#include "Grid3d.h"

//Constructor
Grid3d::Grid3d(int L,int M,int N)
    :L(L), M(M), N(N)
{
    int i,j,k;
    G = new double ** [L];
    for (i=0;i<L;i++){
        G[i] = new double * [M];
        for (j=0;j<M;j++){
            G[i][j] = new double [N];
            for (k=0;k<N;k++){
                G[i][j][k] = NAN;
            }
        }
    }
}

//Empty constructor
Grid3d::Grid3d()
    :L(0), M(0), N(0)
{
    G = NULL;
}

//Copy constructor
Grid3d::Grid3d(const Grid3d &A)
    :L(A.L), M(A.M), N(A.N)
{
    G = new double ** [L];
    int i,j,k;
    for (i=0;i<L;i++){
        G[i] = new double * [M];
        for (j=0;j<M;j++){
            G[i][j] = new double [N];
            for (k=0;k<N;k++){
                G[i][j][k] = A.G[i][j][k];
            }
        }
    }
}

//Destructor
Grid3d::~Grid3d()
{
    // Free memory
    for (int i=0;i<L;i++){
        for (int j=0;j<M;j++){
            delete [] G[i][j];
            G[i][j] = NULL;
        }
        delete [] G[i];
        G[i] = NULL;
    }
    delete G;
    G = NULL;
}

//-------------------------------------------------------------------
// Operators
//-------------------------------------------------------------------

//Access operator ():
double& Grid3d::operator()(int i,int j,int k)
{
    assert(i >= 0 && i < L);
    assert(j >= 0 && j < M);
    assert(k >= 0 && k < N);
    return G[i][j][k];
}

//Assignment operator
Grid3d& Grid3d::operator = (Grid3d A)
{
    swap(A);
    return *this;
}

//-------------------------------------------------------------------
// Access
//-------------------------------------------------------------------

int Grid3d::get_L()
{
    return L;
}

int Grid3d::get_M()
{
    return M;
}

int Grid3d::get_N()
{
    return N;
}

double Grid3d::get(int i,int j,int k)
{
    return G[i][j][k];
}

void Grid3d::clean()
{
    int i,j,k;
    for (i=0;i<L;i++){
        for (j=0;j<M;j++){
            for (k=0;k<N;k++){
                G[i][j][k] = NAN;
            }
        }
    }
}

//------------------------------------------------------------------
// Others
//------------------------------------------------------------------

//Swap
void Grid3d::swap(Grid3d & A)
{
    std::swap(L, A.L);
    std::swap(M, A.M);
    std::swap(N, A.N);
    std::swap(G, A.G);
}

//cout
std::ostream & operator << (std::ostream & os , Grid3d & g)
{
    int L = g.get_L();
    int M = g.get_M();
    int N = g.get_N();

    for (int k=0;k<N;k++){
        cout << "k = " << k << endl;
        for (int i=0;i<L;i++){
            for (int j=0;j<M;j++){
                cout.width(10);
                os << setprecision(3) << g.G[i][j][k] << " ";
            }
            os << endl;
        }
        os << endl;
    }
    return os;
}
```

This is a sample function in which I am using these objects to evaluate the curl of a vector field in a point:

```
double System::curl_r(Grid3d &Ath,Grid3d &Aph,int i,int j,int k)
{
    double curl_rA = ( (SIN[j+1]*Aph(i,j+1,k)-SIN[j-1]*Aph(i,j-1,k))/dth - (Ath(i,j,k+1)-
Ath(i,j,k-1))/dph )/(2.0e0*R[i]*SIN[j]);
    return curl_rA;
}
```

For example:

```
Grid3d Er  = Grid3d(Nr,Nth,Nph);
Grid3d Eth = Grid3d(Nr,Nth,Nph);
Grid3d Eph = Grid3d(Nr,Nth,Nph);
Grid3d curlE_r = Grid3d(Nr,Nth,Nph);
...
curlE_r(i,j,k) = curl_r(Eth,Eph,i,j,k);
```

c++ matrix

edited Apr 25 '14 at 4:04
Jamal ♦
23.1k 6 77 169

asked Jun 19 '13 at 21:47
user1995432
26 1 4

1 I just noticed something, and as you're unlikely to read through a giant answer answer again, I figured I'd make a comment. Your `delete G` ; should be `delete[] G;` since you allocated the contents of `G` with `new[]` . – Corbin Jun 20 '13 at 20:29

4 Answers

We can improve this code.

1: First of all, I urge you to **change your underlying data structure.** `double *** G;` hurts my eyes. Consider using for example `std::vector<std::vector<std::vector<double> > >` instead. (If you are using C++11, you can drop the spaces. Yay!) Using `std::vector` has a plethora of advantages. The two most important are that it will be easier to get the code to work correctly, the other is readability.

If for some masochistic reason you don't want to use `std::vector` or a similar type, at least use arrays instead of raw pointers.

2: `L` , `M` , `N` and `G` are horrific variable names. Consider using for example `width` , `length` , `depth` and `matrix` -- or something along those lines -- instead. The same goes for your getter functions.

3: Your output stream operator (`operator<<`) doesn't use any private members of `Grid3d` (which is good!), so it should not be declared as a `friend` of the class.

4: You can safely use `std::swap()` directly on your object.

5: Your class is using *destructive assignment*. This *could* be okay if it's a well thought-out design decision. In almost all cases, however, it is better to make a (deep) copy of your object.

6: Your class is *not* exception-safe. Much of the reason for this is that it fiddles with raw pointers. In C++, we try to avoid pointers to owned resources whenever we can. Putting your `double` s into a `std::vector` will take care of a lot of this.

7: Use more whitespace. `for (i=0;i<L;i++){` is not very easy on the eye. `for (i = 0; i < L; i++) {` is much better.

8: ~~You probably want to **use** `operator[]` **instead of** `operator()` **for indexing operations.** When in doubt, do as the built-ins do, and arrays use `foo[42]` for indexing.~~ Update: For multiple indices, it's perfectly okay to use `operator()` .

9: Use as few `#include` s as possible in your header file. Move what you can into your implementation file. After removing the `friend` declaration (as discussed in point 3), you can move *all* your `#include` s to your implementation file.

10: Don't ever have `using` statements or `using` directives in a header file. Any file `#include` ing your header will have its global namespace polluted. You don't want that. Use full namespace specifiers in header files. You are already doing this, so the `using` directive can safely be removed.

There are some other issues, but these are the most important ones. They should give you something to work on. I suggest you create a new thread with your improved code after incorporating these changes, for further feedback.

edited Jun 20 '13 at 18:04

answered Jun 20 '13 at 0:13
Lstor
2,390 1 8 29

1 Agreed, except for **8**. It's fairly common practice to overload `operator()` when you need to index something with multiple indices. – Yuushi Jun 20 '13 at 1:03

@Yuushi Oh, right, multiple indices. You are completely correct, I'll edit my answer. – Lstor Jun 20 '13 at 1:05

thanks a lot, i will follow your advices – user1995432 Jun 21 '13 at 2:22

There can be efficiency considerations with a contiguous memory block. So potentially std::vector<double> to do the memory management and then do the appropriate calculations in `operator()` or `operator[]` – Loki Astari Jun 22 '13 at 18:19

I have a few things to add to Lstor's review (and a few things to repeat).

`double***` makes me cry. That's just waiting for some weird exception safety problem or a memory leak. As Lstor said, use a vector (thoug technically `std::vector< std::vector< std::vector<double> > >`, not `std::vector<std::vector<double> >` as he said).

Or, if you really want to use raw allocation, at least use `std::unique_ptr` so you have exception safety.

Fun bonus: if you use vectors, you actually won't need your swap (which you don't need now), destructor or `operator=` as the default ones will work.

Your dimensions should be `std::size_t` s rather than `int` s.

It doesn't apply quite as much here since meaningful names are rather hard to come up with, but in general, you should always use named parameters in your class declarations.

`Grid3d(int,int,int);` doesn't exactly tell me much about what I'm passing to the constructor. `Grid3d(int L, int M, int N);` tells me more. `Grid3d(int width, int height, int depth);` tells me a lot more (though it brings up a rather difficult problem of what you would name a fourth dimension if you were to have one)

`clean` is a bad name. I wouldn't expect clean to mean set everything to `NAN`. I would name it `reset` or `nullify` or something. Or, better yet, have a method called `setAll(double val)` that sets the entire matrix. That way the ambiguity of what you're setting it to is gone (and you don't have to duplicate your constructor loop and your clear loops).

Declare variables as tightly scoped as possible.

```
void Grid3d::clean()
{
    for (int i=0; i < L; i++){
        for (int j=0; j < M; j++){
            for (int k=0; k < N; k++){
                G[i][j][k] = NAN;
            }
        }
    }
}
```

(And once again, those ints should be `std::size_t`.)

You should have a const getter too. Not having one destroys all possibilities of const correctness with usage of your class. For example, it would be nice if `operator<<` took a const `Grid3d` instead of a mutable one. Or what about `curl_r`? If I saw the declaration of that, I would wonder why in the world calculating the curl changes the vectors.

```
T v = ...;
return v;
```

Should usually just be `return ...;`

But in this case, your statement should probably be broken into more digestable pieces:

```
const double part1 = ...; //With a better name than part1, ideally
const double part2 = ...;
const double part2 = ...;
return (part1 * part2) + part3;
```

(This in no way resembles your formula, but I have depressingly little memory of what the different parts mean, so I have no idea how to meaningfully break it apart).

I like to declare variables that I don't plan on changing as `const`. That way, 4 lines later, when I decide I want to reuse `width` (or whatever), I get an error instead of then having the wrong value 2 more lines down when I think I'm using the original width again. Then again, there is definitely a potential to over do it :).

This one is mostly personal preference, but I find spaced out arithmetic operators a million times easier to read. `4+3*2` vs `4 + 3 * 2` (though I would also parenthesize that).

(This one is 80% personal preference.)

I find middle-reference style awkward. I like either `T& t` or `T &t` (personally, I hugely prefer `T& t` as I see the reference being part of the type, not part of the variable).

`std::ostream & operator << (std::ostream & os , Grid3d & g)` Is weird to read for me. I would use: `std::ostream& operator<<(std::ostream& os , Grid3d& g)` .

(And I would actually use `std::ostream& operator<<(std::ostream& os , const Grid3d& g)` .

Being a bit pedantic, `clean` could use the class' getter rather than direct access to G. That way if you change something about G, you only have to alter it in one place.

(If you're worried about performance, `get` will likely be inlined anyway meaning it will have no overhead. You would of course want to profile though if performance is an issue.)

edited Jun 20 '13 at 19:23 answered Jun 20 '13 at 8:41
Corbin
8,236 2 13 43

3 `inline` is basically a compiler suggestion. The compiler is free to do its own cost/benefit analysis and completely ignore it. For something as simple as a getter/setter, however, on any kind of reasonable optimization level, it's almost 100% guaranteed to be inlined automatically anyway. – Yuushi Jun 20 '13 at 10:48

1 +1. Good points! Also, like @Yuushi points out, `inline` is just a hint to the compiler, which is free to do what it wants. Defining a function in the class definition is the same as declaring it `inline` . – Lstor Jun 20 '13 at 18:05

Ah, seems I need to do some reading on inline. I've been procrastinating it for years now :). – Corbin Jun 20 '13 at 19:24

Quickie style issues

Several of these were mentioned in the other answers and some of them weren't.

- You should generally not write code in the global namespace. There are a number of good reasons for this, but this margin is too small to contain them. The programmers stackexchange has some of them.
- You usually shouldn't call `std::swap` directly, though here it is harmless. You should get in the habit of swapping like this: `using std::swap; swap(foo, bar);` It won't cause a different `swap` to be used here, but you shouldn't try to keep in your head which namespace `swap` comes from for each type you use (see this stackoverflow question for more discussion on `swap` and argument-dependent lookup). I discuss `swap` a bit more later.
- Dimensions should generally be passed as `size_t` , not `int` .
- You should not put `using` statements in a header unless inside a function (or sometimes class) definition. You should *especially* not put `using` statements in the global namespace in a header (and usually not in a source file -- put `using` statements in a namespace to prevent symbol clashes and ODR violations during linking).
- You should name the arguments in your header file. Names are the primary form of API documentation and you've got none of that at all in this header file. As LStor mentioned, the names you do have are not great, especially `L` , `get_L` , etc.
- What is `clean` supposed to be doing? Are you trying to pollute your surrounding data with NANs in order to detect use of an uninitialized grid? Consider `clear` instead with 0s instead of NANs, or maybe `setAll` as Corbin suggests.

Underlying data structure

I (mostly) disagree with the other answers about `double***` . This is exactly the sort of class that should use a contiguous array as its underlying data structure. You get exception safety by deleting the memory in your destructor (that, after all, is all that smart pointers do!). However, you're losing any efficiency gain of using the raw data structure by creating an array for each row. Instead, use a `double*` as your underlying memory like so:

```
class Grid3d {
  private:
    size_t L, M, N;
    double* buffer;
  public:
    Grid3d(size_t L, size_t M, size_t N)
      : L(L), M(M), N(N), buffer(new double[L * M * N]) {}
    ~Grid3d() { delete [] buffer; }
    /* ... */
};
```

This has the following advantages over your `double***` approach and over the `vector`-based approaches suggested in the other answers.

- The buffer is allocated with a single allocation, which:
  - reduces heap fragmentation
  - reduces the number of system calls (so is faster)
  - eliminates the possibility of partial allocation, which genuinely was an exception-safety issue with the previous implementation.
- Element accesses will require only one memory lookup, instead of 3.

This approach is in line with Herb Sutter's excellent guideline:

> Don't use explicit new, delete, and owning * pointers, except in rare cases encapsulated inside the implementation of a low-level data structure.

This is one of those low-level data structures.

Allow your class to be used correctly when const.

It is a good general guideline to declare variables const whenever possible. If you didn't wish to make your class assignable and swappable, I'd suggest making its dimensions const. As it is, your users have no way to access the data in the grid when the `Grid3d` is passed to functions by constant reference, for example (as the `Grid3d` arguments should be passed to `System::curl_r` in your example). Here's a solution:

Add a private function to share the index calculation logic:

```
size_t compute_index(size_t i, size_t j, size_t k) const {
    return i * (M * N) + j * N + k;  // untested
}
```

Replace `get(int, int, int)` by `get(int, int, int) const`:

```
double Grid3d::get(size_t i, size_t j, size_t k) const {
    /* add bounds checking */
    return buffer[compute_index(i, j, k)];
}
```

Add `const` and non-`const` versions of `operator()`:

```
double& Grid3d::operator()(size_t i, size_t j, size_t k) {
    return buffer[compute_index(i, j, k)];
}
double Grid3d::operator()(size_t i, size_t j, size_t k) const {
    return buffer[compute_index(i, j, k)];
}
```

Swap correctness

`std::swap` actually correctly swaps your type (as LStor points out) because its member variables are all built-in types (this is sufficient but not necessary). You needn't provide a `swap` method at all, but if you do you can write it as

```
void Grid3d::swap(Grid3d& other) {
  using std::swap;
  swap(*this, other);
}
```

Using the `using std::swap;` construction as I have is unnecessary because you know that you actually will be using `std::swap`, not doing ADL. However, as mentioned above, it's good to get in the habit of doing this because it's absolutely necessary when the arguments to `swap` are of unknown type, as when writing a template (`begin` and `end` are the other two functions which merit this treatment).

The assignment operator

You'll want to provide two assignment operators:

```
Grid3d& Grid3d::operator=(Grid3d&& other) {
  swap(other);
  return *this;
}
```

```
Grid3d& Grid3d::operator=(Grid3d other) {
  swap(other);
  return *this;
}
```

The move-assignment operator prevents an unnecessary copy in expressions like `grid1 = std::move(grid2)` .

There are, of course, other areas of improvement, but this should be enough to get you started.

edited Jun 21 '13 at 2:52

answered Jun 20 '13 at 15:07
ruds
2,158 4 16

Many good points. However, except if writing an industrial-strength library for numeric computations, your suggestion about a `double*` is simply premature optimization. – Lstor Jun 20 '13 at 18:26

2 You're right that a `double*` would have much better performance than a `double***` . One of us probably should have said that sooner. 3 levels of indirection is going to murder the cache, not to mention all the load/store. It could still easily be a std::vector<double> instead of double* though. And, this is *not* a low level data structure. Using a single dimensional vector is going to offer the same performance as a single dimensional array on all but the worst compilers. – Corbin Jun 20 '13 at 19:28

His assignment operator is fine by the way. It's just the classic copy and swap idiom (which is typically done as a simple method of getting a strong exception guarantee -- something your operator= does not offer). – Corbin Jun 20 '13 at 19:37

@Corbin With a double*** that gets deleted by the constructor, it is absolutely not fine. It goes like this. Start with the code Grid3d a(10, 10, 10); Grid3d b; b = a; When b = a is computed, a is copied (and in particular, a.G is copied). Now the copy and b are swapped. b.G == a.G. When a and b go out of scope, they'll both delete G. However, if Grid3d were implemented with a vector, you'd be right; copy and swap would be fine. – ruds Jun 20 '13 at 19:57

@Lstor It's really not a premature optimization. You should not sacrifice code clarity in return for unnecessary optimization, that is true. But a vector<vector<vector<double>>> is actually far less clear than a double*, and insisting on it is premature pessimization. Whether you use a vector or a double* here is a matter of taste, I suppose, but it's not completely clear either way. After all, this buffer is not logically a vector. – ruds Jun 20 '13 at 20:01

A few points mostly regarding efficiency:

- As @ruds said, use a contiguous array. I would use a `vector<double>` instead of the `double *` he suggests, as this simplifies many member functions, at a trivial extra overhead.
- Try to decouple the *size* of the leading dimension of your array (i.e. the one whose elements you store contiguously) from its *allocation*. This allows you to e.g. store 35*10*10 array in a structure with 36*10*10 elements. If you want to make use of the vector processing units (SSE3, Velocity Engine, etc.) that many modern processors have, having structures aligned to 2- or 4-element boundaries can bring tremendous performance benefits. For this reason, numeric libraries like BLAS and LAPACK tend to have separate parameters for the size and allocation of the leading dimension.
- If your problem is suitable for `float` instead of `double` , that is also a tremendous performance win.

answered Jun 20 '13 at 17:58
microtherion
396 1 5