Coding Guidlines

Git Branches

branches allow us to set up our own work enviroments!

how-github-works

Git-CheatSheet

our branching:

master(Daniel)
├── Wolf(Constanca)
├── Metropolis(Justus)
└── Observables(Mateo)

Most important comands

list all branches: `git branch -a`
make a new banch:`git branch my_branch`
switch to a differen branch: `git checkout my_branch`

to push your branch `git push -u origin <branch-name>`

How to get updates form main branch:

`git pull origin main` : updates the main

`git merge main` : get uptates into your current branch

merge as often a possible!!

Pushing Rules

Push as often as possible!!!

only push/sync when the program still compiles!

Daniel will get notified and will merge your work into the main branch

Programs

don't be affraid to change files like in .vscode or CMakeLists in your own branch. Set up your own enviroments!

Daniel will hole heatingly ignore them

1. Make a new program by adding a new .c++ into the programs folder
2. Target the Porgam inside of CMakeList

```
# programs
add_executable(Heisenberg ${SRC_FILES} ./programs/my_program.c++)
```

3. Make sure the ./.vscode/lauch.json file is debuging your program

```
"program": "${workspaceFolder}/build/My_Program",
```

Naming Convetions

```
// write everything big on data types
class AppleJuice;

// write just the first word small on variables/instances
type_t appleJuice;

// write erything small and with _ as spaces in functions
type_t apple_juice(type_t const& t);
```

in general use:

- constexpr: as much as possible, evaluates value by preprocessor.
- const: as much as possible.
- inline: on smaller functions, avoids function call

alwas have an is-, can- or has- for bools

```
bool isOver9000 = true;
bool isDone = false;
bool hasSolution = false;
```

always comment your functions like this

1. What does the function do
2. describe the argument
3. what does the function return?
4. if it can throw an exeption!

try to always use `const&` in your function arguments! otherwise it will copy the entire data stucture!

```
/*
wolf algorithm for the Heisenber3D model

/ @brief
```

```
/ @param lattice our 3d lattice, where to perform the simulation on
/ @return if the procedure succeds
/ @exception may fail
*/
bool wolf(Lattice3d<Spin> & lattice);
```

Our Types

write symbols and types that are often into Base.h++. do never write: `using namespace std;` or any other auto inclusion of a namespace. only use `using std::something` for the things you regulary need to use.

`using flt = double` - is more convienent and faster to wite type.

`Spin` : Our Spin class that works with all representations

`Lattice3D` : A 3d Lattice that can handle different boundry conditions

Exeptions

make use of exceptions, better than asserts!

example:

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

/*
divides a by b = a/b

- a: nenner
- b: zähler
- returns: a/b
- can throw!
*/
flt division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

/*
MainFunction

- no arguments
- returns: 0=Succes, 1=Failure
*/
int main () {
   int x = 50;
   int y = 0;
```

```
    flt z = 0;

    try {
       z = division(x, y);
       cout << z << endl;
    } catch (const char* msg) {
      cerr << msg << endl;
    }

    return 0;
}
```

see: https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

for void function, always return a bool if it had succedet!

example:

```
/*
- i :  pointer to allocated memory
returns: if i is not Null instead of void
- can throw
*/
bool increment_by_one(int* i){
    if(i == nullptr){
        throw std::runtime_error("Nullpointer in increment by one");
        return false;
    }
    ++(*i);
    return true;
}
```

try to avoid unnessesary nesting of if statements by negating it:

```
// never do unnesesarry nesting with if statements
// it leads to spaggetti code
int bad(){
    if(isOver9000){
        if(isDone){
            if(hasSolution){
                // ...
                // a lot of code
                // ...
                return true;
            }
            throw std::domain_error("");
        }
        throw std::runtime_error("");
    }
    throw std::logic_error("");
```

```
    return false;
}

// try to negate the if statements instead
bool good(){
    if(!isOver9000){
        throw std::logic_error("");
        return false;
    }
    if(!isDone){
        throw std::runtime_error("");
        return false;
    }
    if(!hasSolution){
        throw std::domain_error("");
        return false;
    }
    // ...
    // a lot of code
    // ...
    return true;
}
```

Ruler lenth and padding

use a ruler of 70 Symbols

|--|

extend lines with a double tab

example:

```
flt my_superlong_function(Array<flt> const& x, Array<flt> const& y,
        const& Array<flt> z)
{
    // ...
    // code
    // ...
}
```

or use a consitent padding

```
flt my_superlong_function(  Array<flt> const& x,
                            Array<flt> const& y,
                            const& Array<flt> z){
    // ...
    // code
```

```
        // ...
    }
```

if you have a lot of nested loops you can just tap it back at some point:

```
for(uint i = 0; i < L; ++i){
    for(uint j = 0; j < L; ++j){
        for(uint k = 0; k < L; ++k){
            for(uint l = 0; l < L; ++l){
                for(uint m = 0; m < L; ++m){
for(uint n = 0; n < L; ++n){
    for(uint o = 0; o < L; ++o){
        // ...
        // code
        // ..
    }
}
                }
            }
        }
    }
}
```