1 Document for C++ style guide

1.1 First rule

Create minimal working example for all the functions being used.

1.2 Space

- No space for constructor

```
SeqString seq ("AAA");
```

↓

```
SeqString seq("AAA");
```

- Add space for one-line function body

```
Obj dosomething() {return a;}
```

↓

```
Obj dosomething() { return a; }
```

- Constructors with member initializer lists

```
Ref(const SeqStringPtr ref) : ref_(ref) {}
```

↓

```
Ref(const SeqStringPtr ref): ref_(ref) {}
```

- Constructor with empty function body

```
Ref(const SeqStringPtr ref): ref_(ref)
{}
```

↓

```
Ref(const SeqStringPtr ref): ref_(ref) {}
```

- Space between statements

Spaces are used to separate logical blocks of the codes.

In some cases if the code is simple enough and they logically belong to the same block, there is no empty line needed.

```
SeqString query_seq(string("ATGC"));

std::cout << query_seq[2] << std::endl;
```

↓

```
SeqString query_seq(string("ATGC"));
std::cout << query_seq[2] << std::endl;
```

- Remove unnecessary space after `set -v` in bash. Otherwise, it will result an extra space to be printed

```
set -v

ls
```

↓

```
set -v
ls
```

1.3 Variable

- Variable name in bash script should be written in lowercase by default

```
./main.exe "$tmpdir/Index"
```

↓

```
./main.exe "$tmpdir/index"
```

- Variable should only be declared if it will be used multiple times

```
string seq_temp = "ATG"
SeqString seq(seq_temp);
```

↓

```
SeqString seq((string("ATG")));
```

- When passing a large `struct` or `class` to a function, pass by value will make a copy of the argument into the function parameter. Pass by reference solves this issue.

```
void fun(MyClass largeObj)
```

↓

```
void fun(MyClass& largeObj)
```

- Function Parameters passed by reference should be marked as `const` if they are not changed in the function

```
void fun(const class& obj)
```

- Declare class member function to be `const` if no class member changed during the function call

```
void fun1() const;

void fun2() const { doingsomething(); }
```

1.4 Test case

- The folder name of test case must be the same as the function name currently testing on

`/main/xxx/main.cpp` uses to test function named `yyy()`. The folder name should be changed to `/main/yyy/main.cpp`

If you have constructor overloading, you need to organize its test cases in the following way.

```
constructor/parameter_name_version1
constructor/parameter_name_version2
```

- There should always be a test case in `main/` to demonstrate the basic usage of a class. The test case here should only contain one function that is used most frequently.
- When you test a function with parameters, the parameter names you use in test case should be the same in the function declaration

```
//example.hpp
void fun(int name_in_hpp1, int name_in_hpp2)

//test case for example.hpp
int name_in_hpp1;
int name_in_hpp2;
fun(name_in_hpp1, name_in_hpp2);
```

1.5 Others

1.5.1 Code order

Code should be ordered in the way to make the distance between variable definition and usage short

```
0: int  a;                                  //distance 0
1: int  b;                                  //distance 0
2: obj1 c(b);                               //distance 2 - 1 = 1
3: obj2 d(c, a);                    //distance (3 - 2) + (3 - 0) = 4
//Total distance of this block is 5
```

↓

```
0: int  b;                                    //distance 0
1: obj1 c(b);                                 //distance 1 - 0 = 1
2: int  a;                                    //distance 0
3: obj2 d(c, a);                     //distance (3 - 1) + (3 - 2) = 3
//Total distance of this block is 4
```

1.5.2 Print

Never print any unnecessary information

1.5.2.1 print format

In general, using `cout` is a bad idea. This can be easily written as the following, which is more readable and takes less to type.

```
printf("%d plus %d.\n", a, b);
```

Unless there is something really simple (e.g., `cout << x << endl;`), in which `cout` is shorter to type, use `printf()` instead.

1.5.3 Format of library including

Assume you are creating your own class `MyClass`, you should use following include format in `MyClass.cpp`:

```
#include <MyClass.hpp>
```

Use the following format include format in the test case `main.cpp` of `MyClass`, so that you can use `ctrl-W f` in vim to open the file if needed.

```
#include "../relative_path_to_MyClass/MyClass.hpp"
```

1.5.4 Reduce dependency

Only includes a library when it is really needed, e.g. if the function used in `.cpp` file, do not includes library in `.hpp` file.

1.5.5 Combine functions

- If a free function is always used for one class, the free function should be replaced as the member function in that class
- If function `fun_b()` is always used after `fun_a()`, `fun_a()` and `fun_b()` should be combined into one function

1.5.6 Class function

One class should only do one thing. If one class is doing multiple things, it should be split into multiple classes.

1.5.7 shebang & modline

For each bash file, remember to put shebang at very beginning.

```
#!/usr/bin/env bash
```

For each cpp, hpp, bash file, remember to add modline.

```
#vim: set noexpandtab tabstop=2:
```

1.5.8 Code not useful for now

- If you don't need certain files for now, you can simply move them to `backup/`. The convention of `mpp` Makefile is that the files in backup will not be compiled.
- If you don't need certain lines in the file, use the following format.

```
#if 0
// FIXME
// Seems not useful for now

Code..not..useful..for..now

#endif
```

1.5.9 Convention for `main()`

Use the following convention for c++ `main()`

```
int main(int agrc, char *argv[])
{
    code..in..main()
}
```