

# FASTlib Code Style guide

## From Fastlab

Jump to: [navigation](#), [search](#)

--Created and maintained by Nick, after discussing the ideas with Garry and Ryan

## Contents

[\[hide\]](#)

- [1 Examples and tips on the C++ style guide](#)
- [2 C++ DO NOT USE](#)
- [3 Fastlib Magic](#)
- [4 Naming variables, classes, files, constants, etc](#)
- [5 Declarations inside a class](#)
- [6 Functions and Arguments](#)
- [7 Indentation](#)

[\[edit\]](#)

## Examples and tips on the C++ style guide

The original [Coding Style Page](#) was written by Garry and you should read it first. Here you will find more examples.

[\[edit\]](#)

## C++ DO NOT USE

In general we avoid some complexities of C++. We also avoid some of the automations of C++ that generate bugs, just because the programmer is not always aware of the mechanism of C++ and things that happen in the background. Fastlib doesn't use:

1. Inheritance (we prefer templates)
2. Exceptions (if something goes wrong we prefer the program to log and die with the FATAL macro included in fastlib/fastlib.h)
3. Operator overloading, except for some basic classes where they are useful, ie operator[] is useful for matrices, vectors. We prefer to explicitly use Add rather than +.
4. Streams. We prefer the C stdio functions. Prefer the NONFATAL macro in fastlib/fastlib.h to show things on the screen
5. We prefer std::string rather than char\*
6. Never use

```
using namespace name_space;  
where name_space might be la, std  
Always be explicit std::string, std::vector
```

We like but it is not obligatory

1. To use the const qualifier whenever applies

2. Use the keyword `explicit` before the constructors, so that the constructors are called when they should

[\[edit\]](#)

## Fastlib Magic

Use always the fundamental Fastlib types, that are contained in the `fastlib/fastlib.h` header:

```
example:
index_t instead of int
Vector instead of std::vector<double>
```

[\[edit\]](#)

## Naming variables, classes, files, constants, etc

In general try to use long and self explanatory names, for example, `columns` instead of `cols`, `CoverTree`, instead of `CovTree`. - **Classes, Structs** should always start with a capital letter, if the name is a concatenation of two words then every new word should start with a capital letter:

```
example:
class File;
class FileManager;
```

Suggestion: Class names should be nouns (objects)i.e `Manager`, `Tree`, `Classifier`, `EMTrainer`, `SupportVectorMachine`

- **Variables** should always be in small letters. If the variable is a concatenation of more than one words they should be separated by underscore `_`:

```
example:
float alpha;
index_t binary_tree;
```

- **Private Member variables** should always have a trailing underscore `_`

```
example:
class NicksFavoriteExampleClass {
public:
    float you_should_avoid_public_variables;
private:
    string this_is_a_nice_variable_;
}
```

- **Constants** follow the java style constant rules. They are capitalized separated with underscores.

```
example:
constant double UNIVERSAL_FASTLIB_CONSTANT;
and when inside a class
class FastlibClass {
private:
    static constant index_t BLOCK_SIZE_;
};
```

- **Functions** always start with a capital and follow the name convention for Classes. For private member functions add a trailing underscore `_`

```
example:
public:
```

```
void MyFunction();
index_t BuldKdTreeDepthFirst();
private:
void AnotherPrivateFunction_();
```

Suggestion: Use verbs as names for the functions. Words that declare actions.

```
example:
void PruneTree();
index_t EMTrain();
void Build BreadthFirst();
```

**-Files** are always named in small letters and follow the variables naming rules. For example for the class SparseMatrix the definitions will be in the sparse\_matrix.h and sparse\_matrix.cc. If you are writing templated function/classes or inline functions where the definitions should be in the inclusion file use the following scheme:

```
example:
declaration file sparse_matrix.h:
template<typename PRECISION>
class SparseMatrix {
public:
void Init();
};

definition file sparse_matrix_impl.h
template<typename PRECISION>

void SparseMatrix<PRECISION>::Init() {
printf("Do the appropriate Initializations\n");
}

unit test file sparse_matrix_test.cc
```

[\[edit\]](#)

## Declarations inside a class

In this section we present the order of declarations inside a class. First we declare the public and then the private variables/functions. Inside each block (public and private)

1. Constants
2. Typedefs
3. Member Variables
4. Member functions
  1. Constructor
  2. Copy Constructor
  3. Destructor
  4. Init
  5. Copy
  6. Destruct
  7. Member Functions
  8. Accessors do not use get\_private\_variable, instead use the name of the variable private\_variable()
  9. Mutators set\_private\_variable

Note that accessors and mutators are the only exceptions in the naming of functions since they do not start with a capital letter and they use underscore

```

example:
class FastlibClass {
public:
    static const double ALEX_GRAY_SSECRET_CONSTANT =6.66;
    typedef std::vector<std::string> Document;
    struct Feet {
        bool left;
        bool right;
    };
    double AlexSpeed;
    FastlibClass();
    FastlibClass(const FastlibClass &other);
    ~FastlibClass();
    void Init(std::string text);
    void Copy(const FastlibClass &other);
    void Destruct();
    void Funcl();
    std::string file();
    void set_file(std::string &file);

private:
    std::string file_;
    index_t iterations_;
    void PrivateFuncl_();
};

```

Every class should have a trivial constructor that does nothing. It should also have at least one Init function that does all the initializations and one Destruct that does the destructions.

[\[edit\]](#)

## Functions and Arguments

In general we prefer void functions or functions that return `success_t` that indicate success or not of the function. The return structures are always last in the argument list and they are declared as pointers not references.

```

examples:
success_t ProcessFastlibClasses(FastlibClass &class1,
                               FastlibClass &class2,
                               FastlibClass *return_class);

```

```

Usage:
FastlibClass c1,c2,c3;
ProcessFastlibClasses(c1, c2, &c3);

```

The reason why we always use pointers instead of references for the return arguments is because we want to make clear that it is a return argument. Also note that we also avoid default values:

```

example:
NEVER do that
void NeuralNetworkTrain(Netowrk &net, double tolerance=1e-6)
Instead do this
void NeuralNetworkTrain(Netowrk &net, double tolerance);
void NeuralNetworkTrain(Netowrk &net) {
    NeuralNetworkTrain(net, 1e-6);
}

```

We avoid default values because they can cause bugs. This is a general problem with C++. Many things are done automatically implicitly and often cause bugs because the programmer is not aware of them. In our code we often have many parameters that have to be set like the example of the neural network. This is the way you should solve this problem

```

example:
class NeuralNetwork {
public:
    void Init() {
        // Put here all the default initializations for the network
        // tolerance, activation function etc...
    }
    // This will do the training
    void Train();
    // if you want to change the parameters do that with mutators
    // before calling Train()
    void set_tolerance(double tolerance);
    void set_max_num_of_iterations(index_t iterations);
};

```

[\[edit\]](#)

## Indentation

-Leave one space for the label public:, private:. Then leave another space for a member function/variable. Leave one space between public and private functions.

```

example:
class MonteCarloSampler {
public:
    void Init();

private:
    void RandomGenerate();
};

```

-When you define functions in the cc file leave a space between function definitions

example:  
Inside the l2e\_classifier.h

```

class L2EClassifier{
public:
    void Init();
    void Destruct();
};

```

inside the l2e\_classifier.cc

```

void L2EClassifier::Init() {
    printf("Hi there\n");
    printf("I am hi\n");
}

```

```

void L2EClassifier::Destruct() {
    printf("Where are you\n");
    printf("I am super hi\n");
}

```

-Never exceed 80 characters per line. If you need to more space break the line in the two following ways:

1. continue in the following line leaving 4 spaces
2. Or try to align it to look nice

```

example:
a=b.MySuperLongAmazingFunction(vector1,

```

```
vector2,  
&output_vector,  
precision);
```

```
or  
a=b.MySuperLongAmazingFunction(vector1,  
vector2,&output_vector,precision);
```

-Comments. Always use // instead of /\* \*/. Leave a space after //

```
example:  
// This is a super function  
// It has amazing arguments
```

Note Avoiding /\* \*/ gives you the advantage of commenting out code during development so that you can find bugs

Retrieved from "[http://wiki.cc.gatech.edu/fastlab/index.php/FASTlib\\_Code\\_Style\\_guide](http://wiki.cc.gatech.edu/fastlab/index.php/FASTlib_Code_Style_guide)"

## Views

- [Article](#)
- [Discussion](#)
- [Edit](#)
- [History](#)
- [Move](#)
- [Watch](#)

## Personal tools

- [Ravi](#)
- [My talk](#)
- [Preferences](#)
- [My watchlist](#)
- [My contributions](#)
- [Log out](#)

## Navigation

- [Main Page](#)
- [Community portal](#)
- [Current events](#)
- [Recent changes](#)
- [Random page](#)
- [Help](#)
- [Donations](#)

## Search

## Toolbox

- [What links here](#)
- [Related changes](#)

- [Upload file](#)
- [Special pages](#)
- [Printable version](#)
- [Permanent link](#)



- This page was last modified 03:12, 19 January 2008.
- This page has been accessed 86 times.
- [Privacy policy](#)
- [About Fastlab](#)
- [Disclaimers](#)