SmartCar C++ Coding Style Guideline

Author: Peter Tse (mcreng)

This guideline is written in reference to *C++ Coding Standards and Style Guide* by NASA in 2005.

Introduction

This guideline is aimed to provide a recommendation in C++ code writing which is

- Organized
- Easy to read
- Easy to understand
- Maintainable
- Extendable
- Efficient

Names

In general, choose names that are meaningful and readable.

- If abbreviations can be used. do so. The abbreviations should be all in the same case.

```
class PIDController;
BTComm btComm;
```

- Avoid underscores

Struct/Class Names

- Capitalize the first letter of each word.

```
class FeatureExtraction;
struct Edge;
```

Method/Function Names

- Capitalize the first letter of each word. The function name should start with a verb if possible.

```
FindOneLeftEdge();
```

  Some examples of prefix verbs.

Verbs	Meaning
Is/Has/Can	Asking questions about something and return bool type
Set	Setters
Get	Getters
Init	Inititialization
Calc/Find/Compute	Computation
Print	Print

- The name of the class should not be duplicated in a method name.

```
Edge Push(); // not Edge PushEdge();
```

Namespace Names

- Each subpart of your project should be contained in one namespace, with the first letter of each word capitalized.

```
namespace Algorithm;
namespace Utility;
namespace ControlSys;
```

Variable Names

- Variables should be in camelCase, i.e. the first letter of the each word capitalized except the first.

```
Edge leftEdge;
```

- Variable names should be concise at its purpose. If it is not possible, add a comment along with it.

Method/Function Parameters

- Parameter names can have the same name as its type, but with the first letter not capitalized,

```
void genPath(Feature feature);
```

- The parameter name can also be the initial of its type.

```
void genPath(Feature f);
```

Pointers

- Names of pointers should start with `p`.
- Place the `*` operator with the name instead of type.

```
Motor *pMotor = &motor;
```

Reference

- Place the `&` operator with the name instead of type.

```
Coord findFirstCorner(const Edge &edge);
```

- For operator overloads, put the `&` with the type.

```
Coord& operator+=(const Coord &coord);
```

Type Names

- Type name should have the first letter of each word capitalized.

```
typedef uint16_t Byte;
```

Enum Struct Names

- The `enum struct` name should follow the one in *Class Name*.
- The members of the `enum struct` should start with `k` and follow its name with first letter of each word capitalized.

```
enum struct Feature {
  kStraight = 0,
  kRoundroad,
  kCrossroad
};
```

- If the `enum struct` is used with the purpose of flag, name it with the word 'Type'.

```
enum struct RoundaboutStatusType {
  kDetectedEntry = 0,
  kInside,
  kDetectedExit,
  kOutside
};
```

The variable with that type then should have the name camelCase without the word 'Type'.

```
RoundaboutStatusType roundaboutStatus = RoudaboutStatusType::kOutside;
```

Constant Names

- Constant names should be in all CAPS with underscores between words.

```

const uint16_t MAX_NUMBER_OF_EDGE_ENTRY = 200;
```

C++ File Names

- All header files should end with `.h` type.
- All source files should end with `.cpp` type.
- File names of both header and source should match and are put in `\inc` and `\src` folders respectively.

Variables and Constants

- Declarations of temporary variables should just be above the scope it is used.

```
uint16_t blackCount = 0;
for (uint16_t i = 0; i < leftEdge.size(); i++)
  blackCount += leftEdge.at(i)
```

- Beware of the placement of variables, prevent frequent construction and destruction for temporary variables.
- Avoid the use of global variables, rather provide them through the use of class API/namespace.
- Always provide an initialized value for variables.
  - Use `nullptr` instead of `NULL` for initialization of pointers.
- Avoid `#define`, use `const` instead.
- If constant references can be used instead of pointers, use them.

Formatting

Variables

- It is preferable to declare variables with similar purpose in the same line, one per line if not applicable.

```
int leftCount = 0, rightCount = 0;
```

Indentation

- Use 4 spaces instead of a tab for indentation since indentation maybe different for different editors and environments,

Space

- Put one space after a comma/semicolon.

```
pow(2, 3);
for (i = 0; i < n; i++);
```

- Put one space around `=`.

```
c1 = c2;
```

- Put space between keyword and parentheses.

```
if ( ... );
while ( ... );
```

- Put space between parentheses and braces.

```
for (i = 0; i < n; i++) {
  ...
}
```

- No space between function name parentheses.

```
x = pow(2, 3);
```

- No space between unary/primary operators and the operands.

```
p->m;
s.m;
a[i];
a(i);
++i;
-n;
*p;
&r;
```

Blank Lines

- Use blank lines to separate different sections of your code to make it more understandable.

Method/Function Arguments

- If the arguments are too long to be put in one single line, you may line the arguments up with the first argument.

```
Clamp(servoBounds.kLeft,
      pidController.Calc(error),
      servoBounds.kRight)
```

Scopes

- Indent statements if they are in a scope.

```
while (condition) {
  statement;
}
```

Control/Loop

- Same rule with *Scopes*.

- If the inner statement contains only one line, you may write the whole control in one line, or two line with indentation (without braces).

```
if (condition) statement;
//OR
if (condition)
  statement;
```

- It is recommended to use explicit comparisons.

```
if (leftEdge.size() != 0);
//instead of
if (leftEdge.size());
```

Conditional Statements

- Put space around conditional operators.

```
x = (a > b) ? a : b;
```

- Align the `?` and `:` operators in new lines if the statement is too long to be put in one line.

```
(condition)
  ? statement1
  : statement2;
```

Switch

- Always have `default` case, which is put after all other cases and should have `break;` as well for consistency. If it should not be triggered, write a comment to specify it.
- If certain cases are meant to not have `break;`, specify them with a comment.
- You may have a scope declared inside certain case.

```
switch (expression) {
  case a:
    statement;
    break;
  case b: // fall through
    statement;
  case c:
    {
      statement;
      break;
    }
  default:
    // not handled
    break;
}
```

Statements

- Prevent the use of `goto`. Only use it if you believe the control loop would look better with `goto` instead of use of flags.
- You may use `? :` if you believe the statements involved are not too complex.
- Use `constexpr` if you wish the compiler resolve the expression before compilation.

```
constexpr const float SERVO_MODEL_CONST = 120 / 0.5 + 0.6 * std::sin(0.2);
```

Functions

- Use `inline` keyword if the functions are very short.

```
template <class T>
inline T max(T a, T b) { return (a > b) ? a : b; }
```

- Use boolean functions if applicable.

```
bool FindOneLeftEdge() {
  // find one left edge, return false if failed
  return true;
}
```

Documentations

- Always write documentations for class interface (declaration) and function prototypes.

```
/**
 * PIDController Class
 *
 * <brief description>
 */
class PIDController {
public:
  /**
   * @brief Constructor
   * @param kP P constant
   * @param kI I constant
   * @param kD D constant
   */
  PIDController(float kP, float kD, float kD) { ... }

  /**
   * @brief Next control value getter
   * @return Control value
   */
  float getNextVal() { return ...; }
}
```

- Always write the meaning of each constants, specify the unit if necessary.

```
const uint16_t MAX_DISTANCE 400; // Max distance the sensor can detect, in km
```

Classes

- Class declaration should be purely prototypes and attribute declarations.
  - Any implementations should be put outside the class declaration
  - Use `inline` if appropriate
- Sections of `public`, `protected` and `private` should be declared in said order.
- The parameters in class constructors and the member attributes should have different names. Use `m_` to indicate the variable is a member attribute.

```
class Motor {
public:
  Motor(int pow) : m_pow(pow) {}
private:
  int m_pow;
}
```

- Inherited class should have the name of the base class as part of its name.

```
class AlternateMotor : public Motor;
```

- Abstract class should have the function-to-be-overridden declared as pure virtual function, and the inherited class should override the function with `override` keyword.

```
class Motor {
private:
  virtual void OnSetPower(uint16_t power) = 0;
};

class AlternateMotor : public Motor {
private:
  void OnSetPower(uint16_t power) override;
};
```

Templates

- Generic type should have name `T`, `U`, `V`, etc.
- For safety concern, you can include library `<type_traits>` to make sure only certain types are usable.

```
#includfe <type_traits>

template <class T, class = typename std::enable_if<std::is_arithmetic<T>::value>::type>
...
```

Files

Headers

- Preprocessor directive (`#ifndef - #define - #endif`) should be used in every header.

Includes

- Included libraries should be arranged from top to bottom, low-level to high-level.
- Included C++ libraries should be put around `<>` brackets.
- Do not include C libraries, include their C++ counterparts.
- Included libsccc/self-made libraries should be put around `" "`.

```
#include <cmath> // not "math.h"
#include <string>

#include "libsc/motor.h" // libsccc

#include "BTComm.h" // self-made
```