编码风格和命名规范文档

一、总则

- **一致性**：整个项目需保持一致的编码风格，便于代码阅读和维护。
- **可读性**：代码应尽量清晰易读，避免使用难以理解的缩写和复杂的逻辑。
- **自注释**：尽量通过清晰的命名和结构让代码自解释，减少对注释的依赖，但对复杂逻辑应详细注释。

二、文件命名

- **头文件**：使用`.h`后缀，如：`myclass.h`
- **源文件**：使用`.cpp`或`.cc`后缀，如：`myclass.cpp`和`myclass.cc`
- **文件名格式**：采用小写字母命名方式，例如：`myclass.h`

三、类命名

- **类名**：类名应采用大驼峰命名法，每个单词的首字母大写，例如：`MyClass`
- **命名空间**：使用类似类名的命名规则，例如：`MyNamespace`

四、变量命名

- **成员变量**：成员变量使用小驼峰命名法，前缀加上`m_`，例如：`m_memberVariable`
- **局部变量**：使用小驼峰命名法，例如：`localVariable`
- **全局变量**：应避免使用全局变量，如果必须，全局变量使用全大写加下划线分隔，例如：`GLOBAL_VARIABLE`

五、函数命名

- **成员函数**：成员函数应采用小驼峰命名法，例如：`void doSomething()`
- **普通函数**：普通函数应采用小驼峰命名法，例如：`int CalculateSum(int a, int b)`

六、命名规范

- **常量**：
  - 使用全大写并用下划线分隔单词，例如：`const int MAX_SIZE = 100;`
- **宏定义**：
  - 使用全大写并用下划线分隔单词，前缀加上项目名称缩写，例如：`#define PROJECT_NAME_MAX_LENGTH 255`

七、注释规范

- **文件注释**：每个文件开头应包含文件功能的简短描述和作者信息。

```
/**
 * @file customclass.h
 * @brief Short description of the file's purpose
 * @author Your Name
 * @date 2024-06-17
 */
```

- **类注释**：类定义前应包含类功能的简短描述。

```
/**
 * @class CustomClass
 * @brief Short description of the class
 */
class CustomClass {};
```

- **函数注释**：函数声明和定义前应包含函数功能的简短描述，参数和返回值的解释。

```
/**
 * @brief Calculates the sum of two integers
 * @param a The first integer
 * @param b The second integer
 * @return The sum of a and b
 */
int calculateSum(int a, int b);
```

- **行内注释**：对于复杂逻辑，可以在行尾或行上方加上简短说明。

```
int result = a + b; // Calculate the sum
```

八、排版风格

- **缩进**：统一使用4个空格进行缩进，不使用制表符(tab)。
- **空格和空行**：适当地使用空行来分隔代码块，增强可读性。关键字与括号间留一个空格，例如：`if (condition) {`

```
if (condition) {
    doSomething();
}
else {
    doSomethingElse();
}
```

- **大括号**：尽量保持统一，大括号放在控制语句同一行的末尾，并在下一行开始新的代码块。

```
for (int i = 0; i < 10; ++i) {
    // Loop body
}
```

九、特殊规则

- **智能指针**：尽量使用Qt提供的智能指针(如`QSharedPointer`、`QScopedPointer`等)代替普通指针，避免内存泄漏。
- **信号和槽**：使用`connect`函数连接信号与槽时，尽量使用成员函数指针形式。

```
connect(sender, &SenderClass::signalName, receiver, &ReceiverClass::slotName);
```

十、代码检查

- **代码审查**：定期进行代码审查，确保遵守编码规范，并发现和解决潜在问题。
- **静态分析工具**：使用Qt Creator内置的静态分析工具或其他代码分析工具，发现和修复代码中的潜在问题。

附录：Qt代码示例

```
#ifndef CUSTOMCLASS_H
#define CUSTOMCLASS_H

/**
 * @file customclass.h
 * @brief Example class demonstrating Qt coding standards
 * @author Your Name
 * @date 2024-06-17
 */

#include <QString>

/**
 * @class CustomClass
 * @brief This class provides an example following Qt coding standards.
 */
class CustomClass {
public:
    /**
     * @brief Constructor for CustomClass.
     */
    MyClass();

    /**
     * @brief Calculates the sum of two integers.
```

```
     * @param a The first integer.
     * @param b The second integer.
     * @return The sum of a and b.
     */
    int CalculateSum(int a, int b) const;

private:
    QString m_name; // Name of the instance
};

#endif // CUSTOMCLASS_H
```

```
#include "customclass.h"

/**
 * @brief Constructor implementation
 */
CustomClass::CustomClass() : m_name("Default") { // Member initializer list
}

/**
 * @brief Calculates the sum of two integers.
 */
int CustomClass::CalculateSum(int a, int b) const {
    // Simple sum calculation
    return a + b;
}
```

通过遵循上述的编码风格和命名规范，能够确保团队协作效率，提升代码的可读性和可维护性。同时，规范的代码风格也能够减少代码审核中出现的问题，提升整体的开发质量。如果有其他特定的规范需求，可以根据项目实际情况进行适当调整。