Clean code principles

1. The Basics:
   - Clean code should be easy to read and modify.
   - Clean code should have a clear purpose.

   Example:

```
// Good code
int calculateSum(int a, int b) {
  return a + b;
}

// Bad code
int f(int x) { return x * x; }
```

2. Naming:
   - Good naming makes code easier to read and understand.

   Example:

```
// Good code
int calculateSum(int firstNumber, int secondNumber) {
  return firstNumber + secondNumber;
}

// Bad code
int f(int x, int y) { return x + y; }
```

3. Functions:
   - Functions should be short and do one thing only.
   - Avoid global variables and complex conditions.
   - Use early returns instead of nested if statements.

   Example:

```
// Good code
bool isValidNumber(int number) {
  return number >= 0 && number <= 100;
}

// Bad code
bool checkNumber(int number) {
  if (number >= 0 && number <= 100) {
    return true;
  } else {
    return false;
  }
}
```

4. Comments:
   - Use comments to explain why code is written in a certain way, not what it does.
   - Avoid using comments to hide bad code.

   Example:

```
// Good code
// Calculates the sum of two numbers
int calculateSum(int firstNumber, int secondNumber) {
  return firstNumber + secondNumber;
}

// Bad code
// This is a function that adds two numbers
int f(int x, int y) { return x + y; }
```

5. Organization:
   - Organize code in a logical way.
   - Avoid repeating code and use shared functions.

   Example:

```
// Good code
void printName(string name) {
  cout << "Name: " << name << endl;
}

void printAge(int age) {
  cout << "Age: " << age << endl;
}

// Bad code
void printNameAndAge(string name, int age) {
  cout << "Name: " << name << endl;
  cout << "Age: " << age << endl;
}
```

6. Testing:
   - Write unit tests for functions.
   - Use automated testing to prevent breaking the system.
   - Test all possible paths in the system.

   Example:

```
// Good code
int calculateSum(int firstNumber, int secondNumber) {
  return firstNumber + secondNumber;
}

TEST(CalculateSumTest, TestPositiveNumbers) {
  EXPECT_EQ(calculateSum(5, 10), 15);
}

// Bad code
int f(int x, int y) { return x + y; }
```

7. Error Handling:
   - Handle errors and exceptions responsibly.
   - Use clear and understandable error messages.

   Example:

```
// Good code
void openFile(string fileName) {
  ifstream file(fileName);
  if (!file.is_open()) {
    throw runtime_error("Failed to open file.");
  }
}

// Bad code
void openFile(string fileName) {
  ifstream file(fileName);
  if (!file.is_open()) {
    cout << "Error opening file." << endl;
  }
}
```

8. Maintenance:
   - Think about maintenance when writing code.
   - Keep code easy to maintain and modify.
   - Document code changes and remove unused code.

   Example:

```
// Good code
class Rectangle {
public:
  Rectangle(int width, int height) : width_(width), height_(height)
{}
  int area() const { return width_ * height_; }
private:
  int width_;
  int height_;
};

// Bad code
int calculateArea(int w, int h) { return w * h; }
```

9. Design:
   - Design the system before writing code.
   - Keep the design simple and easy to understand.
   - Design the system to be extensible and modifiable.

   Example:

```
// Good code
class Shape {
public:
  virtual int area() const = 0;
};

class Rectangle : public Shape {
public:
  Rectangle(int width, int height) : width_(width), height_(height)
{}
  int area() const override { return width_ * height_; }
private:
  int width_;
  int height_;
};

// Bad code
int calculateArea(int w, int h) { return w * h; }
```

10. DRY (Don't Repeat Yourself):
    - Avoid duplicating code.
    - Use functions or classes to encapsulate common functionality.

    Example:

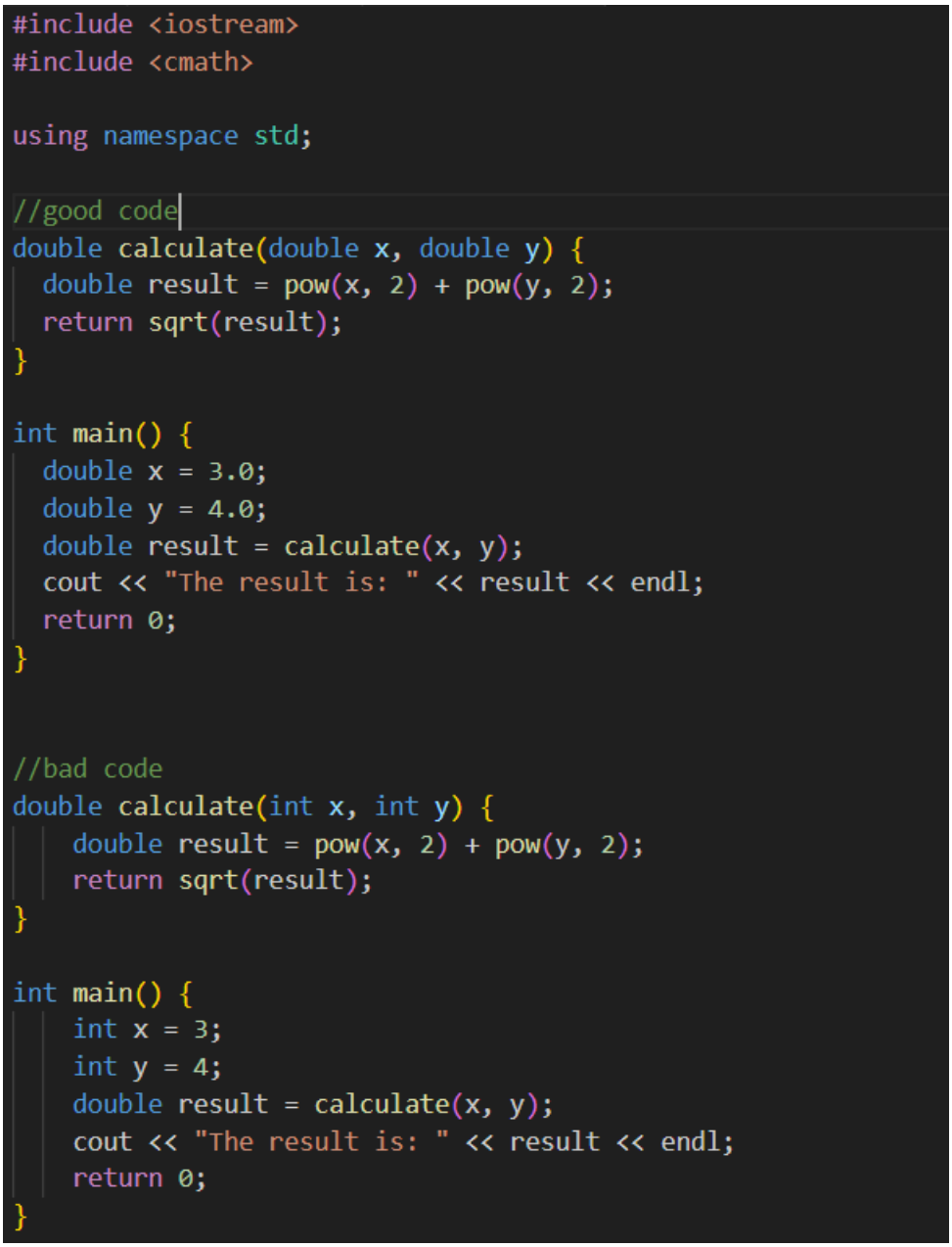
```
// Good code
int calculateSum(int a, int b) {
  return a + b;
}

int calculateProduct(int a, int b) {
  return a * b;
}

// Bad code
int main() {
  int a = 5, b = 10;
  int sum = a + b;
  int product = a * b;
}
```

11. Tools:
    - Use appropriate tools to aid in writing clean code.
    - Utilize available tools to help with code quality and identifying potential issues.

    Example:



```cpp
#include <iostream>
#include <cmath>

using namespace std;

//good code
double calculate(double x, double y) {
  double result = pow(x, 2) + pow(y, 2);
  return sqrt(result);
}

int main() {
  double x = 3.0;
  double y = 4.0;
  double result = calculate(x, y);
  cout << "The result is: " << result << endl;
  return 0;
}


//bad code
double calculate(int x, int y) {
    double result = pow(x, 2) + pow(y, 2);
    return sqrt(result);
}

int main() {
    int x = 3;
    int y = 4;
    double result = calculate(x, y);
    cout << "The result is: " << result << endl;
    return 0;
}
```

12. Composition over Inheritance:
    - Prefer composition over inheritance to avoid tight coupling between classes.
    - Use inheritance only when it makes sense and when it's necessary.

    Example:

```cpp
// Good code
class Engine {
public:
  virtual void start() = 0;
  virtual void stop() = 0;
};

class ElectricEngine : public Engine {
public:
  void start() override { /* start electric engine */ }
  void stop() override { /* stop electric engine */ }
};

class Car {
public:
  Car(Engine* engine) : engine_(engine) {}
  void start() { engine_->start(); }
  void stop() { engine_->stop(); }
private:
  Engine* engine_;
};

// Bad code
class Car {
public:
  void start() { /* start car */ }
  void stop() { /* stop car */ }
};

class ElectricCar : public Car {
public:
  void start() override { /* start electric car */ }
  void stop() override { /* stop electric car */ }
};
```

13. Single Responsibility Principle:
    - A class should have only one reason to change.
    - Avoid creating classes that do too much.

    Example:

```cpp
// Good code
class Logger {
public:
  void log(std::string message) { /* log message */ }
};

class Calculator {
public:
  int add(int a, int b) {
    logger_.log("Adding numbers");
    return a + b;
  }
private:
  Logger logger_;
};

// Bad code
class Calculator {
public:
  int add(int a, int b) {
    /* log message */
    return a + b;
  }
};
```