**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

[C/C++ Ten Coding Standards 3](#_3dy6vkm)

[Coding Standard 1 4](#_1t3h5sf)

[Coding Standard 2 5](#_4d34og8)

[Coding Standard 3 6](#_2s8eyo1)

[Coding Standard 4 7](#_17dp8vu)

[Coding Standard 5 8](#_3rdcrjn)

[Coding Standard 6 9](#_26in1rg)

[Coding Standard 7 10](#_lnxbz9)

[Coding Standard 8 11](#_35nkun2)

[Coding Standard 9 13](#_1ksv4uv)

[Coding Standard 10 14](#_44sinio)

[Defense-in-Depth Illustration 15](#_2jxsxqh)

[Project One 15](#_z337ya)

[1.](#_3j2qqm3) Revise the C/C++ Standards 15

[2.](#_1y810tw) Risk Assessment 15

[3.](#_4i7ojhp) Automated Detection 15

[4.](#_2xcytpi) Automation 15

[5.](#_1ci93xb) Summary of Risk Assessments 16

[6.](#_3whwml4) Create Policies for Encryption and Triple A 16

[7.](#_2bn6wsx) Map the Principles 17

[Audit Controls and Management 18](#_qsh70q)

[Enforcement 18](#_3as4poj)

[Exceptions Process 18](#_1pxezwc)

[Distribution 19](#_49x2ik5)

[Policy Change Control 19](#_2p2csry)

[Policy Version History 19](#_147n2zr)

[Appendix A Lookups 19](#_3o7alnk)

[Approved C/C++ Language Acronyms 19](#_23ckvvd)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Don't blindly trust information coming from users or external sources. Check all inputs to make sure they are in the expected format and don't contain malicious code. |
| 1. Heed Compiler Warnings | Compilers often flag potential security issues in your code. Ignoring these warnings can leave vulnerabilities for attackers to exploit. |
| 1. Architect and Design for Security Policies | Security should be considered from the very beginning of any project. The architecture and design should reflect the security policies you want to enforce. |
| 1. Keep It Simple | Complex systems are harder to secure. Strive for clean, well-organized code that is easier to understand and maintain. |
| 1. Default Deny | By default, deny access to everything and only grant permissions explicitly. This reduces the attack surface and makes it harder for unauthorized users to gain access. |
| 1. Adhere to the Principle of Least Privilege | Users and systems should only have the minimum permissions needed to perform their tasks. This limits the potential damage if a security breach occurs. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to another system, make sure it is free of malicious code or unexpected formatting that could cause security issues. |
| 1. Practice Defense in Depth | Security shouldn't rely on a single layer of defense. Implement multiple security measures to create a layered approach that makes it more difficult for attackers to succeed. |
| 1. Use Effective Quality Assurance Techniques | Thorough testing helps identify and fix vulnerabilities before they can be exploited. |
| 1. Adopt a Secure Coding Standard | Following a well-established secure coding standard helps developers write code that is less vulnerable to security breaches. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Variable and Data Type Selection and Use** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | * Choose the data type with the smallest range that can accurately represent the intended values. * Declare data types for all variables and function parameters. * Choose variable names that reflect the data type and its purpose. * Avoid unnecessary conversions between data types, especially when losing precision. * If using non-standard data types (e.g., large integers), document the reason and potential limitations. |

| **Noncompliant Code** |
| --- |
| * value variable uses a generic type and variable name * temp variable is unclear, as it could be one of several units. * largeNumber has the possibility to overflow, and is likely better as a long long. * Implicit type conversions can result in loss of precision, such as float -> int |
| int value; // Could be for age, price, or something else  double temp; // What kind of temperature is it (Celsius, Fahrenheit)?  int largeNumber = 10000000000; // Might be better as a long long  // Loss of precision  int averageRating = (float)totalRating / numRatings; |

| **Compliant Code** |
| --- |
| * Appropriate Data types are used for different data * Variable names are descriptive * Function parameters have explicit type declarations |
| int age = 30;  double pi = 3.14159;  bool isRegistered = true;  const std::string customerName = "John Doe";  unsigned int orderNumber = 12345;  float calculateRectArea(float length, float width) {  // ...  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Adhere to the Principle of Least Privilege (Principle 6): By choosing the smallest data type, you minimize the memory footprint and potential overflow vulnerabilities. * Keep It Simple (Principle 4): Using clear and appropriate data types improves code readability and reduces the chance of errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | Application |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | [Insert text.] | Static Code Analyzer | Analyzes code for various issues, including unused variables, type mismatches, and potential narrowing conversions. |
| CPP Lint | [Insert text.] | Static Code Analyzer | Identifies stylistic and potential functional problems in C++ code. |
| Clang-Tidy | Included in Clang compiler | Static Code Analyzer | Offers a set of linters that flag coding issues based on custom checks or predefined configurations. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Literal, Magic Number, and Value Range Use** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | * Employ clear and descriptive values for constants and literals. * Avoid using unexplained numeric constants. If a specific value is crucial, define it as a named constant with clear documentation (comments or header files). * When dealing with a set of predefined values, consider using enums for improved readability and type safety. * Anticipate and handle potential invalid or unexpected data values gracefully. Implement error handling or validation checks where necessary. * If specific ranges or limitations apply to data values (e.g., valid temperature range), document them clearly for future reference (comments or header files). |

| **Noncompliant Code** |
| --- |
| * Literal values are unclear * Magic number are used without explanation * Hardcoded values are used instead of enums * Invalid input is not checked |
| int status = 1; // What does 1 represent?  int quantity\_limit = 10; // No context  float discount\_applied\_if\_gt = 0.1f; // Unclear purpose  int payment\_type = 2; // Error-prone if new options emerge (e.g., CHECK)  int calculate\_area(int length, int width) {  // No check for negative values  int area = length \* width;  return area;  } |

| **Compliant Code** |
| --- |
| * Literal values have descriptive names and optional further clarification comments * Magic numbers are documented when necessary * Enums are used for predefined values * Invalid input is handled gracefully |
| #include <limits> // for numeric\_limits  const int MAX\_ITEM\_QUANTITY = 10;  const double GRAVITY\_CONSTANT = 9.81; // m/s^2  static const float DISCOUNT\_THRESHOLD = 0.1f; // Documented discount activation point  enum class PaymentMethod {  CREDIT\_CARD,  DEBIT\_CARD,  CASH  };  PaymentMethod payment\_type = CASH;  int calculate\_area(int length, int width) {  if (length < 0 || width < 0) {  throw std::invalid\_argument(  "Length and/or Width cannot be negative"  );  }  int area = length \* width;  return area;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Validate Input Data (Principle 1): Documenting expected data value ranges helps developers anticipate and handle invalid inputs. * Keep It Simple (Principle 4): Employing clear and descriptive constants enhances code understanding and reduces the likelihood of errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | Application |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | [Insert text.] | Code Quality & Security Platform | Provides various reports and metrics, including code smells related to magic numbers and potential hardcoded values. |
| Code Climate | [Insert text.] | Code Review & Quality Platform | Identifies issues like unclear variable names and missing documentation for constants. |
| Coverity Static Analysis | [Insert text.] | Static Code Analyzer | Detects coding defects, including use of undefined variables and potential buffer overflows. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | * Always validate external input strings before processing to prevent unexpected characters, buffer overflows, or potential code injection. * Choose the correct encoding (e.g., UTF-8) based on the expected character set to avoid encoding errors and data corruption. * Be mindful of null terminators (C-style strings) and ensure proper memory management to avoid dangling pointers and undefined behavior. * Utilize the C++ Standard Library string functions (std::string, std::stringstream, etc.) for safe and efficient string manipulation. * Before performing operations like concatenation or accessing characters, verify the string's length and capacity to prevent buffer overflows or out-of-bounds access. * Clearly document any assumptions made about the format or content of strings (e.g., expected characters, maximum length) for future reference. |

| **Noncompliant Code** |
| --- |
| * User input is not validated * C-style strings are used without proper null termination * Strings are concatenated without checking capacity |
| char username[20];  std::cout << "Enter your username: ";  std::cin >> username; // Potential buffer overflow  char message[] = "Hello, world!";  int message\_length = strlen(message); // Assumes null termination  std::string greeting = "Welcome, ";  greeting += username; // Might overflow if username is too long |

| **Compliant Code** |
| --- |
| * std::string is used over C-style strings * User input is validated, both in variable constraints and runtime/business constrains * String functions are used for safer variable accessing |
| #include <iostream>  #include <string>  #include <cctype> // for std::isalnum  // Function to validate user input (basic example)  bool validate\_username(const std::string& username) {  if (username.empty()) {  return false; // Empty username is invalid  }  for (char c : username) {  if (!std::isalnum(c) && c != '\_') {  return false; // Only alphanumeric and underscore allowed  }  }  return true;  }  int main() {  std::string name;  std::cout << "Enter your name: ";  std::getline(std::cin, name);  if (validate\_username(name)) {  std::cout << "Hello, " << name << std::endl;  } else {  std::cerr << "Invalid username. Please use only letters, numbers, or underscores." << std::endl;  }  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Validate Input Data (Principle 1): String validation protects against potential code injection attacks (SQL injection, XSS). * Practice Defense in Depth (Principle 8): Multiple techniques (e.g., validation, character encoding) strengthen string security. * Use Effective Quality Assurance Techniques (Principle 9): Verifying string length and capacity helps prevent buffer overflows during testing. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | System |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer (ASan) | Included in compiler toolchains | Runtime Analyzer | Detects buffer overflows, heap-use-after-free, and other memory-related errors. |
| UndefinedBehavior Sanitizer (UBSan) | [Insert text.] | Runtime Analyzer | Identifies potential undefined behavior, including invalid string operations. |
| Valgrind | [Insert text.] | Memory Error Detection Suite | Offers tools like Memcheck for identifying memory leaks, invalid memory access, and use of uninitialized memory. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Secure Database Access** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | * Always construct database queries using parameterized queries. Bind user input as separate parameters to the query, preventing malicious code from being interpreted as part of the SQL statement. * Thoroughly validate all user input before incorporating it into a query. Sanitize user input to remove potentially harmful characters or code. * Utilize prepared statements (if supported by your database driver) to pre-compile the SQL statement structure, further preventing SQL injection attempts. * Avoid building SQL queries dynamically from strings. This increases the risk of SQL injection if user input is directly inserted. * Implement robust error handling for database operations. Analyze error messages to identify potential injection attempts. |

| **Noncompliant Code** |
| --- |
| * User input is not validated or sanitized and is directly inserted into query string * Query is not parameterized, allowing malicious code to be inserted |
| // Building SQL string dynamically (prone to injection)  std::string username;  std::cout << "Enter your name: ";  std::getline(std::cin, username);  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| * User input is sanitized prior to being parameterized by removing any non alphanumeric or underscore characters * Query is parameterized and prepared |
| #include <cppconn/driver.h>  #include <cppconn/exception.h>  #include <cppconn/statement.h>  // Function to execute a parameterized query  void execute\_query(const std::string& sql, const std::string& username) {  try {  sql::Driver\* driver = get\_driver\_instance();  std::unique\_ptr<sql::Connection> con(driver->connect("localhost", "user", "password"));  std::unique\_ptr<sql::Statement> stmt(con->createStatement());  stmt->prepareSQL(sql);  stmt->setString(1, username); // Bind username as parameter  stmt->execute();  } catch (const sql::SQLException& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  }  int main() {  std::string name;  std::cout << "Enter your name: ";  std::getline(std::cin, name);  // Sanitize user input (example)  name.erase(std::remove\_if(name.begin(), name.end(), [](char c) { return !std::isalnum(c) && c != '\_'; }), name.end());  std::string query = "SELECT \* FROM users WHERE username = ?";  execute\_query(query, name);  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Validate Input Data (Principle 1): Sanitizing user input prevents malicious code from being injected into SQL statements. * Practice Defense in Depth (Principle 8): Parameterized queries and prepared statements add layers of protection against SQL injection. * Use Effective Quality Assurance Techniques (Principle 9): Robust error handling helps identify potential injection attempts. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Medium | Medium | High | System |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQLMap | [Insert text.] | Security Scanner | Automated penetration testing tool that can identify and exploit SQL injection vulnerabilities. |
| Acunetix | [Insert text.] | Web Application Security Scanner | Scans web applications for various vulnerabilities, including SQL injection. |
| OWASP ZAP | [Insert text.] | Web Application Security Scanner | Open-source tool that can be used to identify SQL injection vulnerabilities in web applications. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Management and Protection** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | * Favor smart pointers (e.g., std::unique\_ptr, std::shared\_ptr) for memory management. Smart pointers automatically handle deallocation, reducing the risk of memory leaks. * Manage resources like memory using RAII principles. Acquire resources (e.g., memory) within a constructor and release them automatically in the destructor. * Minimize the use of raw pointers (new, delete) as they require manual memory management. Use smart pointers or containers from the Standard Library when appropriate. * Always check array indices before accessing elements to prevent out-of-bounds access and potential buffer overflows. * Utilize container classes (e.g., std::vector, std::string) from the Standard Library for dynamic data management. These classes handle memory allocation and deallocation internally. * Gracefully handle situations where memory allocation might fail using functions like new with a null check (if (ptr == nullptr) …). |

| **Noncompliant Code** |
| --- |
| * Raw pointers are used without proper management * Allocation failures are not checked for * Array bounds are not checked before accession. |
| int\* data = new int(10);  int\* buffer = new int[100]; // Might fail if memory is unavailable  int my\_array[5];  my\_array[10] = 20; // Accessing beyond allocated memory |

| **Compliant Code** |
| --- |
| * Smart pointers are used for object management * Vectors are used for dynamic array management |
| #include <memory>  #include <vector>  std::unique\_ptr<int> create\_number(int value) {  return std::make\_unique<int>(value);  }  int main() {  std::unique\_ptr<int> number = create\_number(42);  std::cout << "The number is: " << \*number << std::endl;  // No need to manually deallocate memory (handled by smart pointer)  std::vector<int> data = {1, 2, 3, 4};  for (int element : data) {  std::cout << element << " ";  }  std::cout << std::endl;  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Practice Defense in Depth (Principle 8): Smart pointers and RAII principles provide automatic memory management, reducing the risk of memory leaks and vulnerabilities. * Keep It Simple (Principle 4): Utilizing container classes simplifies memory management and reduces the chance of errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | System |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer (ASan) | Included in compiler toolchains | Runtime Analyzer | Detects memory errors like use-after-free and heap buffer overflows. |
| Clang-Tidy | Included in Clang compiler | Static Code Analyzer | Offers memory-related checks, such as potential leaks and use of raw pointers without proper management. |
| Valgrind | [Insert text.] | Memory Error Detection Suite | Offers Memcheck for identifying memory leaks, invalid memory access, and use of uninitialized memory. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Effective Use of Assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | * Use assertions to verify assumptions made about program state at specific points in the code. These invariants should hold true unless there's a bug. * Assertions should be clear, concise, and easy to understand. Include informative messages to aid in debugging if an assertion fails. * Place assertions at logical checkpoints within the code where the expected state can be verified. * Don't assert conditions that are already checked for normal program flow. * In production builds, consider disabling assertions to optimize performance, as they are primarily for development and testing. |

| **Noncompliant Code** |
| --- |
| * Assertion messages are unclear * Redundant assertions are used |
| int result = calculate\_area(length, width);  assert(result > 0); // What does a negative result indicate?  bool is\_even(int number) {  if (number % 2 == 0) {  return true;  } else {  assert(false); // Unnecessary assertion, already checked in the if condition  }  } |

| **Compliant Code** |
| --- |
| * Assertions have clear messages * Assertions are used only when needed |
| #include <cassert>  // Function to check if a number is positive  bool is\_positive(int number) {  assert(number >= 0); // Assert the expected input condition  return number > 0;  }  int main() {  int value = 5;  if (is\_positive(value)) {  std::cout << "The value is positive." << std::endl;  } else {  assert(false && "Unexpected negative input received."); // Fail assertion with message for debugging  }  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Keep It Simple (Principle 4): Clear and concise assertions improve code clarity and aid debugging. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Medium | Low | Low | Development |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | [Insert text.] | Static Code Analyzer | Offers checks for unused assertions, which might indicate potential logic errors. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exception Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | * Use exceptions to handle errors that are unexpected or outside the normal program flow. Don't use them for regular control flow. * Throw specific exception types that clearly indicate the nature of the error. Utilize the C++ Standard Library exceptions (std::exception, std::logic\_error, etc.) when appropriate. * When catching exceptions, use catch by reference (catch (const std::exception& e)) to avoid unnecessary copies. * In a catch block, handle the exception gracefully. This may involve logging the error, cleaning up resources, notifying the user, or re-throwing the exception for higher-level handling. * Choose the appropriate exception guarantee for your functions (noexcept, strong guarantee, basic guarantee). This helps callers understand the exception behavior. |

| **Noncompliant Code** |
| --- |
| * Generic exceptions are used for normal program flow * Exceptions are caught by value, potentially creating an unnecessary copy |
| // Using exceptions for normal program flow control  int divide(int a, int b) {  if (b == 0) {  throw std::runtime\_error("Division by zero"); // Not an exceptional condition  }  return a / b;  }  // Catching by value (potentially unnecessary copy)  try {  // ... code  } catch (std::exception e) { // Catch by value  std::cerr << "Error: " << e.what() << std::endl;  } |

| **Compliant Code** |
| --- |
| * Custom exception is used over generic exceptions * Exceptions are caught by reference |
| #include <iostream>  #include <stdexcept>  // Custom exception for invalid input  class InvalidInputException : public std::invalid\_argument {  public:  InvalidInputException(const std::string& message) : std::invalid\_argument(message) {}  };  int divide(int numerator, int denominator) {  if (denominator == 0) {  throw InvalidInputException("Division by zero!");  }  return numerator / denominator;  }  int main() {  try {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  } catch (const InvalidInputException& e) {  std::cerr << "Error: " << e.what() << std::endl;  } catch (const std::exception& e) {  // Catch other potential exceptions here  std::cerr << "Unexpected error: " << e.what() << std::endl;  }  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Keep It Simple (Principle 4): Specific exception types enhance code readability and maintainability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | Application |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPP Lint | [Insert text.] | Static Code Analyzer | Can identify issues like missing exception specifications or exception handlers. |
| Clang-Tidy | Included in Clang compiler | Static Code Analyzer | Offers checks for exception-related issues, such as catching exceptions by value instead of reference. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Robust Input/Output** |
| --- | --- | --- |
| Input/Output | STD-008-CPP | * Favor std::endl over \n`: Use std::endl for newlines to ensure proper flushing of output buffers, especially when mixing formatted and unformatted output with cout. * Always check the return value of file operations like fopen, ifstream::is\_open(), etc., to handle potential file access errors gracefully. * Utilize stream manipulators like std::setw, std::setprecision, and std::fixed to format output for better readability and control over data presentation. * For complex data handling or persistence needs, explore alternative I/O mechanisms like file streams (fstream), istringstream/ostringstream for string manipulation, or custom I/O abstractions. |

| **Noncompliant Code** |
| --- |
| There are several issues: The file openness is never validated; The file is never closed; The potentially unopened file is attempted to be read from; The output is unbuffered, with \n being used directly. |
| #include <iostream>  #include <fstream>  int main() {  std::ifstream file("data.txt");  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  std::cout << "Line 1";  std::cout << "\nLine 2"; // Might not print "Line 2" immediately  } |

| **Compliant Code** |
| --- |
| Non-compliant issues are addressed: The file openness is validated before attempted to read; The file is closed when done; std::endl is used over \n; |
| #include <iostream>  #include <fstream>  int main() {  std::ifstream input\_file("data.txt");  if (!input\_file.is\_open()) {  std::cerr << "Error: Could not open file!" << std::endl;  return 1;  }  std::string line;  while (std::getline(input\_file, line)) {  std::cout << line << std::endl;  }  input\_file.close();  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Validate Input Data (Principle 1): Checking file operation return values helps handle potential access errors that could lead to security issues. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | Application |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPP Lint | [Insert text.] | Static Code Analyzer | Can identify potential issues related to file operations, such as missing error handling for fopen or fstream. |
| Clang-Tidy | Included in Clang compiler | Static Code Analyzer | Offers checks for exception-related issues, such as catching exceptions by value instead of reference. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Effective Object-Oriented Design** |
| --- | --- | --- |
| OOP Design | STD-009-CPP | * Prefer composition (has-a relationship) over inheritance (is-a relationship) to achieve code flexibility and avoid tight coupling between classes. * Ensure each class has a single, well-defined responsibility to enhance code clarity and maintainability. * Strive to design classes that are open for extension but closed for modification. Utilize techniques like interfaces and abstract classes. * Define public interfaces for a class exposing its functionalities, while keeping implementation details encapsulated within the private section. * Ensure data members are private and accessed through member functions (getters and setters) to control access and maintain data integrity. |

| **Noncompliant Code** |
| --- |
| * SRP is violated as the Shape class is responsible for the shape itself, and the drawing functionality. * Inheritance is used over composition, creating a tight coupling |
| // Violation of SRP (Shape class has drawing responsibility)  class Shape {  private:  double width\_;  double height\_;  public:  Shape(double width, double height) : width\_(width), height\_(height) {}  double getArea() const { return width\_ \* height\_; }  void draw() const { // Drawing logic here (not SRP compliant) }  };  class Rectangle : public Shape {  public:  Rectangle(double width, double height) : Shape(width, height) {}  };  class Square : public Rectangle {  public:  Square(double side\_length) : Rectangle(side\_length, side\_length) {}  }; |

| **Compliant Code** |
| --- |
| * The Shape class is only responsible for being a shape * Abstract base Shape class provides unique implementations for similar functionality |
| class Shape {  public:  virtual double getArea() const = 0;  };  class Circle : public Shape {  private:  double radius\_;  public:  Circle(double radius) : radius\_(radius) {}  double getArea() const override { return 3.14159 \* radius\_ \* radius\_; }  };  class Square : public Shape {  private:  double side\_length\_;  public:  Square(double side\_length) : side\_length\_(side\_length) {}  double getArea() const override { return side\_length\_ \* side\_length\_; }  }; |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Keep It Simple (Principle 4): Well-defined class responsibilities and clear interfaces make code easier to understand and maintain. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | Application |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Understand for C++ | [Insert text.] | Code Analysis Tool | Provides visualizations and reports to help understand class relationships and potential design issues. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Safe and Efficient Concurrency** |
| --- | --- | --- |
| Concurrency | STD-010-CPP | * Strive to design tasks that operate on independent data sets to maximize parallelization potential. * Ensure shared data is accessed and modified in a thread-safe manner using synchronization mechanisms like mutexes, atomics, or lock-free data structures. * Carefully design synchronization to avoid situations where multiple threads wait indefinitely for each other's resources. * Be mindful of scenarios where threads keep attempting to acquire locks but never succeed, leading to wasted CPU cycles. * For managing a pool of reusable threads, utilize thread pool abstractions offered by the C++ standard library or external libraries. |

| **Noncompliant Code** |
| --- |
| * Shared data is not accessed and modified in a thread-safe manner * Two threads are declared explicitly, performing the same function. A thread pool or simple array/vector group of threads is more clear and flexible. |
| #include <iostream>  #include <thread>  int shared\_counter = 0;  void increment\_counter(int n) {  for (int i = 0; i < n; ++i) {  shared\_counter++; // Data race - potential for incorrect value  }  }  int main() {  std::thread thread1(increment\_counter, 10000);  std::thread thread2(increment\_counter, 10000);  thread1.join();  thread2.join();  std::cout << "Final counter value (may be incorrect): " << shared\_counter << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| * Shared data is accessed and modified in a thread-safe manner by the use of a mutex * Similar threads are created in a vector group, providing a single entry variable for all threads of a task |
| #include <iostream>  #include <mutex>  #include <thread>  #include <vector>  std::mutex mtx; // Mutex for thread safety  int shared\_counter = 0;  void increment\_counter(int n) {  for (int i = 0; i < n; ++i) {  std::lock\_guard<std::mutex> lock(mtx); // Acquire mutex for critical section  shared\_counter++;  }  }  int main() {  std::vector<std::thread> threads;  int num\_threads = 4;  int increments\_per\_thread = 10000;  // Launch threads to increment the counter  for (int i = 0; i < num\_threads; ++i) {  threads.emplace\_back(increment\_counter, increments\_per\_thread);  }  // Wait for all threads to finish  for (auto& thread : threads) {  thread.join();  }  std::cout << "Final counter value: " << shared\_counter << std::endl;  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   * Practice Defense in Depth (Principle 8): Synchronization mechanisms ensure thread-safe access to shared data, preventing race conditions and data corruption.Risk Assessment table that read |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Low (for simple cases)  High (for complex cases) | Medium | High | System |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer (TSan) | Included in compiler toolchains | Runtime Analyzer | Detects data races, deadlocks, and other thread-related errors. |
| Helgrind | Included in Valgrind suite | Thread Interaction Error Detection | Identifies errors like data races and potential deadlocks in multithreaded applications. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Insert your written explanations here.]

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Low | Low | Medium | 2 |
| STD-002-CPP | Medium | Low | Low | Medium | 2 |
| STD-003-CPP | High | Medium | Medium | High | 4 |
| STD-004-CPP | Critical | Medium | Medium | High | 4 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Low | Medium | Low | Low | 1 |
| STD-007-CPP | Medium | Low | Low | Medium | 3 |
| STD-008-CPP | Medium | Low | Low | Medium | 3 |
| STD-009-CPP | Medium | Low | Medium | Medium | 3 |
| STD-010-CPP | High | Low - simple  High - complex | Medium | High | 4 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | * How it's used: This method scrambles data on storage devices like hard drives, solid-state drives, or backup media. The data becomes gibberish without a decryption key. * Why and When the Policy Applies: This is crucial for protecting sensitive information like financial records, personal data, or intellectual property. Even if a device is lost, stolen, or accessed by unauthorized personnel, the encrypted data remains unreadable. It's a core security policy for organizations dealing with sensitive data or complying with regulations. |
| Encryption in flight | * How it's used: This encrypts data while it's being transmitted over a network, like sending an email with an attachment or accessing online banking information. The data is scrambled during transmission and decrypted upon reaching its destination. * Why and When the Policy Applies: This protects data from eavesdroppers or attackers who might intercept communication channels. It's essential for online transactions, sending confidential emails, or using public Wi-Fi networks. |
| Encryption in use | * How it's used: This encrypts data while it's actively being processed or accessed. Imagine a program that needs to decrypt a credit card number for verification purposes, but then keeps it encrypted in memory while processing the transaction. * Why and When the Policy Applies: This adds an extra layer of security for highly sensitive data, even within a trusted system. It's particularly important for organizations that handle financial information or need to comply with stringent data privacy regulations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | * How it's used: This process verifies the identity of a user or device attempting to access a network resource. It's like checking someone's ID before letting them enter a building. Common methods include usernames and passwords, multi-factor authentication (MFA), or digital certificates. * Why and When the Policy Applies: Authentication is the foundation of network security. It ensures only authorized users and devices can access specific resources, preventing unauthorized access and potential breaches. This policy applies to all network connections, regardless of user type or device. |
| Authorization | * How it's used: This process determines what level of access a verified user or device has after authentication. Imagine someone with an ID card being granted access to specific floors in a building based on their role. Authorization controls what actions users can perform on the network (e.g., read, edit, delete files). * Why and When the Policy Applies: Authorization minimizes damage from compromised accounts. Even if someone gains unauthorized access, authorization limits their ability to cause harm by restricting access to sensitive data or functionalities. This policy applies to all authenticated users and devices, ensuring granular control over network access. |
| Accounting | * How it's used: This process tracks and records user and device activity on the network. It's like keeping logs of who entered the building, when, and what areas they accessed. Accounting data can be used for auditing purposes, identifying suspicious activity, or troubleshooting network issues. * Why and When the Policy Applies: Accounting provides valuable insights into network usage and potential security threats. It helps identify unusual access patterns that might indicate unauthorized activity or compromised accounts. This policy is crucial for organizations with compliance requirements or needing to monitor network activity for security reasons. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/26/2024 | Milestone 3: Create 10 standards | Dennis James Stelmach |  |
| 1.2 | 06/16/2024 | Project One | Dennis James Stelmach |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |