**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## 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 | To minimize vulnerabilities, it's important to validate incoming data from both untrusted and trusted sources to ensure no malicious data enters the system. |
| 1. Heed Compiler Warnings | Compile and test the code extensively to ensure it remains efficient. While warnings might seem minor, they could indicate serious issues, so it's important to check them thoroughly. |
| 1. Architect and Design for Security Policies | If you develop your code with security as a priority while ensuring it meets its intended purpose, you can save significant time later by avoiding the need to add safeguards or fix vulnerabilities once they are discovered. |
| 1. Keep It Simple | Everything, including the security of the code, is made simpler by keeping it as simple and uncomplicated as possible. This is a basic life lesson, and it applies particularly to security: the easier something is to comprehend, the easier it is to safeguard. |
| 1. Default Deny | Setting denial as the default is beneficial because it blocks unauthorized access while ensuring that those who need access follow the correct procedures to obtain it. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege aligns well with default denial, as it ensures that only individuals who require higher access will follow the correct procedures to obtain it. During this process, it is clearly established that only the necessary access is granted, thereby reducing potential risks by limiting the amount of available information. |
| 1. Sanitize Data Sent to Other Systems | Make sure that any sensitive information or possible vulnerabilities are removed from all data before sending it to other systems. Sending such data could result in exploits that could have serious repercussions. |
| 1. Practice Defense in Depth | A system that could otherwise be susceptible to breaches and disruptions can be protected by employing several varied layers of defense, as most systems require more than one layer of security. |
| 1. Use Effective Quality Assurance Techniques | In all processes, quality assurance is essential. Since its goal is to find problems in any process or design, it can be very unpopular, but it is still necessary for the result. It is better for team members to find any issues than for a customer or bad actor to find them, which could have more serious repercussions. |
| 1. Adopt a Secure Coding Standard | Establishing a secure coding standard complements other principles like designing code with security policies in mind and implementing effective quality assurance techniques. Keeping these concepts at the forefront helps save time and prevents future issues, which could be more costly to fix later on. |

### 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** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Defining C-style variadic functions may introduce vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The purpose of this function is to read values until a 0 is discovered; however, issues may arise if a 0 is not found after two arguments. |
| #include <cstdarg>  Int add(int one, int two, …) {  Int result = one + two;  va\_list args;  va\_start(args, two);  while (int value = va\_arg(args, int)  result = result + value;  }  Va\_end(args);  Return result;  } |
|  |

| **Compliant Code** |
| --- |
| An add statement included in this function helps to prevent the aforementioned problems. |
| #include <type\_traits>  template<typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type\* = nullptr>  int add(Arg one, Arg two) {  return one + two;  }  template<typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type\* = nullptr>  int add(Arg one, Ts... rest) {  return one + add(rest...);  } |

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

| **Principles(s):** When working with functions that take a variable number of arguments, defensive programming is essential because it makes sure that improper or unexpected usage doesn't result in vulnerabilities. If care is not taken to validate the number and types of arguments used, C-style variadic functions can result in security risks like buffer overflows. Template overloading and type traits are used in the given compliant example to enforce more stringent type checking and guard against abuse. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 5.4p0 | LANG.STRUCT.ELLIPSIS | Ellipsis |
| Astree | 20.10 | Function-ellipsis | Checked fully |
| RuleChecker | 22.10 | CERT\_CPP-INT50-a | Checked fully |
| Clang | 3.9 | Cert-dc150-cpp | Checked By clang-tidy- |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Incorrectly defining a reserved identifier can lead to problems because it won't be reserved. |

| **Noncompliant Code** |
| --- |
| Unpredictable behavior results from the disregard for naming norms. |
| #ifndef \_MY\_HEADER\_H\_  #define \_MY\_HEADER\_H\_  // Contents of <my\_header.h>  #endif // \_MY\_HEADER\_H\_ |

| **Compliant Code** |
| --- |
| The issue is fixed by eliminating the underscores at the start and finish. |
| #ifndef MY\_HEADER\_H  #define MY\_HEADER\_H  // Contents of <my\_header.h>  #endif // MY\_HEADER\_H |

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

| **Principles(s):** Unintended conflicts or undefinable behavior that could result from defining reserved identifiers are avoided by following naming conventions and proper declaration practices. This standard guarantees that code stays portable and is less prone to errors while also preventing namespace pollution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -wreserved-id-macro  -wuser-defined-literals | Default and -wall do not enable the -wreseved-id-macro flag; instead, -weverything does. Not all instances of this rule, like redefining reserved names, are detected by this flag. |
| Axivion Bauhaus Suite | 6.9.0 | CertC++DCL51 |  |
| Astree | 20.10 | Reserved-identifier | Partially checked |
| CodeSonar | 7.4p0 | ALLOC.UAF | Use After Free |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | To avoid undefined behavior, do not qualify a reference type with volatile or const. Unexpected consequences may arise if a compiler fails to deliver a critical error message, which is the optimum scenario. |

| **Noncompliant Code** |
| --- |
| Instead of addressing a const-qualified char, a reference to a char is inadvertently turned const-qualified. |
| #include <iostream>  void f(char c) {  char &const p = c;  p = 'p';  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| Get rid of the const qualifier in order to avoid this problem. |
| #include <iostream>  void augmentChar(char c) {  char &resource = c;  resource = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):** Incorrect use of volatile or const can result in compiler errors or warnings as well as undefined behaviors. By requiring proper modifier usage, this principle makes sure that the code is robust and improves maintainability and reliability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2020a | CERT C++: DCL52-CPP | Checks for:   * Const-qualified reference types * Modification of const-qualified reference types   Rule Fully covered |
| Axivion Bauhaus Suite | 6.9.0 | CertC++DCL52 |  |
| PRQA QA -C++ | 4.4 | 0014 |  |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL52-a | Never use the terms "const" or "volatile" to describe a reference type. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Don't make statements that have more than one interpretation. Make sure your code is unambiguous and straightforward. |

| **Noncompliant Code** |
| --- |
| This statement can be seen as either declaring an object called m and building it by default, or as creating an anonymous object and calling its single-argument converting constructor. |
| #include <mutex>  static std: : mutex m;  static int combined\_item;  void increase\_by\_55() {  std: :individual\_key<std: :mutex>(m):  combined\_item += 55;  } |

| **Compliant Code** |
| --- |
| The relevant conversion constructor is called, and the lock is given an identifier. |
| #include <mutex>  Static std: :mutex m;  Static int combined\_item;  void increase\_by\_55() {  std: :individual\_key<std: :mutex> key(m);  combined\_item += 55 |

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

| **Principles(s):** Code ambiguity may cause compilers to interpret it differently, which could result in unexpected behaviors or vulnerabilities. By making data manipulation transparent and explicit, this principle actively lowers the risk of errors and potential exploits like SQL injection by guaranteeing clear and simple code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PRQA QA -C++ | 4.4 | 2502,2510 |  |
| LDRA tool suite | 9.7.1 | 296 S | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b | Functions should always be declared at file scope.  An identifier declared in a global or namespace scope cannot be concealed by an identifier declared in a local or function prototype scope. |
| Polyspace Bug Finder | R2020a | CERT C++: DCL53-CPP | Verifies whether any declarations are interchangeable with:   * Unnamed object or function parameter declaration * Function and object declaration   Rule fully covered |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Ensure that allocation and deallocation functions are overloaded together within the same scope. Not doing so may result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| There is no deallocation function defined, despite the allocation being globally overloaded. |
| #include <Windows.h>  #include <new>  void\* operator new(std::size\_t size) noexcept(false) {  static HANDLE h = ::HeapCreate(0, 0, 0); // A private, expandable heap.  if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }  // There is no global delete operator defined. |

| **Compliant Code** |
| --- |
| The overload issue should be avoided because the deallocation is indicated. |
| #include <Windows.h>  #include <new>  class HeapAllocator {  static HANDLE heapHandle;  static bool isInitialized;  public:  static void\* allocate(std::size\_t size) noexcept(false) {  if (!isInitialized) {  heapHandle = ::HeapCreate(0, 0, 0); // Create a private, expandable heap.  isInitialized = true;  }  if (heapHandle) {  return ::HeapAlloc(heapHandle, 0, size);  }  throw std::bad\_alloc();  }  static void deallocate(void\* pointer) noexcept {  // Implementation for deallocation  }  };  If (h) {  (void): :HeapFree(h, 0, ptr);  }  }  };  HANDLE HeapAllocator: :h = nullptr;  Bool HeapAllocator: :init = false;  void \*operator new(std: :size\_t\_size) noexcept(false) {  return HeapAllocator: :alloc(size);  }  void operator delete(void\*ptr) noexcept {  return HeapAllocator: :dealloc(ptr);  } |

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

| **Principles(s):** In order to stop undesired behaviors like memory leaks or corruption, memory resources must be managed properly. In order to maintain heap integrity and stop resource leaks, this principle mandates that memory allocation and deallocation must be paired operations within the same scope. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Low | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2020a | CERT C++: DCL54-CPP | Looks for discrepancies between the operator delete and the overloaded operator new (rule fully covered). |
| Astree | 20.10 | New-delete-pairwise | Partially checked |
| Clang | 3.9 | Misc-new-delete-overloads | Checked with clang-tidy- |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL54-a | Always offer delete and new together |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Make sure that when you move a class object beyond a trust border, no information is disclosed. Verifying the data being transferred is crucial to averting any possible problems. |

| **Noncompliant Code** |
| --- |
| Sensitive information may be included in the data in this example when it is transferred using any means. |
| #include <cstddef>  struct test {  int a;  char b;  int c;  };  // Securely transfer bytes to user space  Extern int transfer\_to\_user(void\*dest, void\*src, std: :size\_t size);  void operate(void\*user\_buff) {  test arg{1, 2, 3};  transfer\_to\_user(user\_buff, &arg, sizeof(arg));  } |

| **Compliant Code** |
| --- |
| This should help prevent these problems by serializing the structured data before transferring it. |
| #include <cstddef>  #include <cstring>  struct test {  int a;  char b;  int c;  };  // Securely transfer bytes to user space.  extern int transfer\_to\_user(void \*dest, void \*src, std::size\_t size);  void do\_stuff(void \*usr\_buff) {  test arg = {1, 2, 3};  // Buffer might be larger than necessary.  unsigned char buffer[sizeof(arg)];  std::size\_t offset = 0;  }  std::memcpy(buf + offset, &arg.a, sizeof(arg.a));  offset += sizeof(arg.a);  std::memcpy(buf + offset, &arg.b, sizeof(arg.b));  offset += sizeof(arg.b);  std::memcpy(buf + offset, &arg.c, sizeof(arg.c));  offset += sizeof(arg.c);  transfer\_to\_user(usr\_buff, buf, offset /\* size of data copied \*/);  } |

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

| **Principles(s):** This principle focuses on making sure that sensitive information doesn't leak when data is moved across trust boundaries. The code reduces the risk of unintentionally exposing data as it departs the protection of its original context by requiring serialization and validation of the data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL55-a | A function that can copy data to the user space shouldn't be given a pointer to a structure. |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully Implemented |
| Axivion Bauhaus Suite | 2020.2 | CERT\_CPP-DCL55-a |  |
| LDRA tool suite | 9.7.1 | 44 S | Fully Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When initializing static objects, avoid loops. The behavior will be uncertain if a function is invoked again while a static object is being initialized. |

| **Noncompliant Code** |
| --- |
| This attempts to use caching to build a factorial function; however, recursion is used to initialize the static array cache, which results in undefinable behavior. |
| #include <stdexcept>  int num(int i) noexcept(false) {  if (i < 0) {  // Factorials for negative numbers are not defined.  throw std::domain\_error("i must be >= 0");  }  static const int collection[] = {  num(0), num(1), num2), num(3), num(4), num(5),  num (6), num(7), num(8), num(9), num(10), num(11),  num(12), num(13), num(14), num(15), num(16)  };  if (i < (sizeof(collection) / sizeof(int))) {  return collection[i];  }  return i > 0 ? i \* fact(i - 1) : 1;  } |

| **Compliant Code** |
| --- |
| Because the static cache is not being used, the problem occurs. |
| #include <stdexcept>  int fact(int i) noexcept(false) {  if (i < 0) {  // Factorials for negative numbers are not defined.  throw std::domain\_error("i must be >= 0");  }  // Utilize a collection that initializes lazily.  static int collection<sup><a data-citation-index="17" class="tw-cursor-pointer !tw-text-blue-50">[17]</a></sup>;  if (i < (sizeof(collection) / sizeof(int))) {  if (collection[i] == 0) {  // Additional logic for caching would go here.  }  }  }  collection[i] = (i > 0) ? (i \* fact(i - 1)) : 1;  }  return collection[i];  }  Return i > 0 ? I \* fact(I – 1) : 1;  } |

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

| **Principles(s):** In order to guarantee safe and reliable initialization, this standard places a strong emphasis on avoiding loops when initializing static objects. Static object initialization in C++ can be challenging since undefined behavior may result from accessing an object before it has been fully initialized. In order to avoid recursion or other risky accesses during initialization, the initialization logic is encapsulated. This principle guarantees consistent and secure behavior of program initialization logic by avoiding loops that could result in re-entry into the initialization code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-err58-cpp | Checked byclang-tidy |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL56-a | Use local static objects instead of non-local ones to prevent initialization order issues across translation units. |
| LDRA tool suite | 9.7.1 | 6 D | Enhanced Enforcement |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | Avoid defining an unnamed namespace in a header file, as it can lead to problems. |

| **Noncompliant Code** |
| --- |
| Because the variable is declared in an unnamed namespace, every translation unit uses a different instance. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE  namespace {  int v;  }  #endif // A\_HEADER\_FILE  // a.cpp  #include "a.h"  #include <iostream>  void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }  // b.cpp  #include "a.h"  #include <iostream>  void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }  int main() {  extern void f();  f(); // Prints v, sets it to 42  g(); // Prints v, sets it to 100  f();  g();  } |

| **Compliant Code** |
| --- |
| One translation unit creates the variable, yet it is available to all and yields the intended result. |
| / / a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE  extern int v;  #endif // A\_HEADER\_FILE  // a.cpp  #include "a.h"  #include  int v; // Definition of global variable v  void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }  // b.cpp  #include "a.h"  #include <iostream>  void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }  int main() {  extern void f();  f(); // Prints v, puts the num at 55  g(); // Prints v, puts the num at 200  f(); // Prints v, puts the num at 55  g(); // Prints v, puts the num at 200  } |

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

| **Principles(s):** In order to avoid multiple definitions of the same entity across translation units, this standard recommends against defining unnamed namespaces in header files. Every entity (like a variable or function) must have a single definition when linking, according to the C++ One Definition Rule (ODR) principle. Because unnamed namespaces allow for internal linkage, every translation unit will have a unique instance, which could result in conflicts or unexpected behavior. Code becomes less prone to linkage errors and more predictable when this principle is followed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tool Suite | 9.7.1 | 286 S, 512 S | Fully Implemented |
| Astree | 20.10 | Unnamed-namespace-header | Fully checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++DCL59 |  |
| Clang | 3.9 | Cert-dcl59-cpp | Checked by clang-tidy |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | Avoid allowing exceptions to be thrown from destructors or deallocation functions. |

| **Noncompliant Code** |
| --- |
| Unpredictable behavior could result from an exception raised by the class's destructor. |
| #include <stdexcept>  class S {  bool has\_error() const;  public:  ~S() noexcept(false) {  // Regular processing  if (has\_error()) {  throw std::logic\_error("Something bad");  }  }  }; |

| **Compliant Code** |
| --- |
| Any exceptions will be caught and removed as a result. |
| class SomeClass {  Bad bad\_member;  public:  ~SomeClass() {  try {  // ...  } catch(...) {  // Handle exceptions from noncompliant destructors  // of member objects or base class subobjects.  // NOTE: If a destructor's try-block ends without handling an exception,  // the caught exception is automatically rethrown. An explicit  // return statement will stop this from occurring.  return;  }  }  }; |

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

| **Principles(s):** The main goal of this standard is to forbid throwing exceptions in deallocation or destructor functions. The idea has to do with exception safety, which is essential to C++'s dependable and strong resource management. Program termination, the loss of important diagnostic data, and potential resource leaks can result from a destructor throwing an exception while another exception is propagating. By making sure that destructors finish without propagating exceptions, the compliant code example demonstrates how to catch and handle such exceptions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL57-a | Never permit a destructor, deallocation, or swap to throw an exception.  Always catch exceptions |
| LDRA tool suite | 9.7.1 | 453 S | Partially implemented |
| Axivion Bauhaus Suite | 6.9.0 | CertC++DCL57 |  |
| Astree | 20.10 | Destructor-without-noexcept  Delete-without-noexcept | Fully Checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | Avoid altering the standard namespaces. Adding new declarations within the namespace can lead to unpredictable behavior if not used properly. |

| **Noncompliant Code** |
| --- |
| Unpredictable behavior results from the inclusion of x in the namespace. |
| Namespace std {  Int x;  } |

| **Compliant Code** |
| --- |
| Undefined behavior is avoided by using a non-reserved name. |
| Namespace nonstd {  Int x;  } |

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

| **Principles(s):** Standard namespaces such as std should not be altered, according to this standard. The namespace integrity principle guarantees that libraries and language constructs continue to behave consistently and predictably. Adding definitions to standard namespaces can cause conflicts and ambiguous behavior, especially when the language or third-party libraries are updated in the future. Developers maintain the integrity and expected behavior of these fundamental constructs by not changing standard namespaces. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PQQA QA-C++ | 4.4 | 4032, 4035, 4631 |  |
| Polyspace Bug Finder | R2020a | CERT C++: DCL58-CPP | Verifies whether standard namespaces have been modified. (rule fully covered) |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL58-a | The standard namespaces "std" and "posix" should not be altered. |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL58 |  |

### 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 | High | Probable | Medium | P12 | L1 |
| STD-002-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-003-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-004-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-005-CPP | Low | Probable | Low | P6 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-0010-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-008-CPP | Low | Likely | Medium | P6 | L2 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |

### 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 | Encryption at rest is a technique used to safeguard data stored on devices. The remaining data may be kept in databases or hard drives, but as long as it is encrypted in those locations, it all falls into the same category. By including this encryption technique in the policy, data is protected against potential breaches. Because encryption at rest techniques are being used, hackers might be able to access the data but not see what it contains. To put it another way, the hacker cannot read the data and finds it useless. Data that is not encrypted at rest is vulnerable to exploitation in the event of a system data breach. |
| Encryption in flight | The use of encryption at The simplest definition of flight encryption in flight is the protection of data during its journey from point A to point B. Information and data are always changing. It moves from the user to the network or the other way around. Encrypting data while it is in flight is crucial because it can be intercepted while it is in motion. By doing this, data will be protected from unauthorized access. Prior to traveling, the data is encrypted, and after it arrives at its destination, it is decrypted. By implementing this policy, the potential harm from a data breach is reduced. |
| Encryption in use | Data being used (manipulated) in software applications can be restricted from being accessed by anyone other than the intended user by implementing safety measures. While data is at risk when it is in an active state, hackers can avoid eavesdropping by encrypting data while it is in the computer's memory. In order to prevent data from being vulnerable to exploitation, this policy is applicable and crucial. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of confirming that the person planning to log in to the system has the right login credentials is known as authentication. This restricts system access according to the admin's permission. Systems can employ a variety of authentication techniques, including two-factor authentication, biometric features, and physical key cards. This policy's security features, which restrict access to authorized personnel, make it applicable. |
| Authorization | Limiting what a user can do while on the network is the main goal of authorization. Restricting users' access to the system lessens the chance that malware or vulnerabilities will be introduced. The best course of action when applying authorization is to adhere to the maxim "less is best." In other words, it will be easier to stop attacks if users have less access. |
| Accounting | Accounting is defined as recording the activities of users within the system. When suspicious activity is discovered, the administrator can view a backlog of what has been done while users are on the network. This technique can also be used to notify the administrator of any unauthorized changes or attempts to access them. |

**\***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 | 1/20/25 | Added the Ten Core Security Principles and Ten Coding Standards | Darius Quick | N/A |
| 1.2 | 2/10/25 | Added the Risk Assessments | Darius Quick | N/A |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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