**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 | Accept nothing by default. Validate type, range, length, format, and encoding on every external input (UI, files, network, env) before use. Reject or sanitize anything that doesn’t match strict whitelists. This prevents common flaws like injection and buffer overflows |
| 1. Heed Compiler Warnings | Treat warnings as errors and use the highest warning levels plus static analysis. Compilers and analyzers surface real defects (e.g., use-after-free, narrowing, UB) early, when they’re cheapest to fix. |
| 1. Architect and Design for Security Policies | Bake security in at design time: model threats, define trust boundaries, and choose patterns that enforce authentication, authorization, and auditing (AAA) alongside data-protection controls (at rest, in flight, in use). |
| 1. Keep It Simple | Prefer simple, auditable designs and idiomatic C++. Complexity breeds misconfiguration and mistakes; smaller APIs and fewer states reduce attack surface and ease code review and testing. |
| 1. Default Deny | Block by default; allow by exception. Apply least privilege to processes, files, sockets, and API inputs so mistaken configurations don’t silently grant broad access. |
| 1. Adhere to the Principle of Least Privilege | Run with the minimum permissions needed, isolate secrets, and scope tokens narrowly. In code, minimize object capabilities and avoid exposing raw pointers/handles unnecessarily. |
| 1. Sanitize Data Sent to Other Systems | Normalize and encode all outbound data appropriate to the sink (e.g., SQL parameters, HTML escaping, JSON encoding). Never concatenate untrusted data into commands or documents. |
| 1. Practice Defense in Depth | Layer controls (validation, parametrization, RBAC, logging, rate-limits, sandboxing). If one layer fails, subsequent layers contain impact and support detection and response. |
| 1. Use Effective Quality Assurance Techniques | Combine unit/integration tests, fuzzing, static/dynamic analysis, and code review. Security defects are quality defects—automate checks in CI to prevent regressions. |
| 1. Adopt a Secure Coding Standard | Follow a written, agreed-upon C/C++ standard (e.g., safety rules for memory, exceptions, I/O). Standards make secure practices consistent and enforceable in reviews and tooling. |

### 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] | DCL60-CPP — Obey the one-definition rule (ODR) |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule)  **Noncompliant Code** |
| --- |
| Two separate translation units each define a class with the same name but conflicting definitions. |
| // a.cpp  **struct** S {  **int** a;  };    // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| Use of a header file to introduce the object into both translation units. |
| // S.h  struct S {  **int** a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** (2) Heed Compiler Warnings / (10) Adopt a Secure Coding Standard — Following DCL60-CPP ensures that each entity in the program has only one definition across all translation units. Violating this rule can cause inconsistent object layouts, leading to undefined behavior. Compilers often provide ODR-related warnings, so treating them as errors helps identify and correct inconsistencies early. By also adopting a secure coding standard such as SEI CERT, teams ensure consistent enforcement of best practices that maintain portability, predictability, and reliability across projects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | type-compatibility  definition-duplicate  undefined-extern  undefined-extern-pure-virtual  external-file-spreading  type-file-spreading | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL60 |  |
| CodeSonar | 9.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| Helix QAC | 2025.2 | C++1067, C++1509, C++1510 |  |
| LDRA tool suite | 9.7.1 | 286 S, 287 S | Fully implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL60-a | The One Definition Rule shall not be violated |
| Polyspace Bug Finder | R2025b | CERT C++: DCL60-CPP | Checks for inline constraints not respected (rule partially covered) |
| RuleChecker | 22.10 | type-compatibility definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading type-file-spreading | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | EXP53-CPP - Do not read uninitialized memory |

**Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory)

| **Noncompliant Code** |
| --- |
| An uninitialized local variable is used in an expression for output, which leads to undefined behavior. |
| #include <iostream>    void f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| Object is initialized prior to printing its value. |
| #include <iostream>    void f() {  **int** i = 0;    std::cout << i;  } |

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

| **Principles(s):** (1) Validate Input / (9) Effective Quality Assurance - Reading from uninitialized memory results in unpredictable values and undefined behavior. This issue can cause random crashes or data leaks. Initializing variables upon declaration and using static analysis tools helps ensure predictable program execution and data integrity. Regular testing and input validation prevent uninitialized values from entering the program flow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | uninitialized-read | Partially checked |
| Clang | 3.9 | -Wuninitialized clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| CodeSonar | 9.1p0 | LANG.STRUCT.RPL LANG.MEM.UVAR | Return pointer to local Uninitialized variable |
| Helix QAC | 2025.2 | DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978 |  |
| Klocwork | 2025.2 | UNINIT.CTOR.MIGHT  UNINIT.CTOR.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST  UNINIT.STACK.ARRAY.MIGHT  UNINIT.STACK.ARRAY.MUST  UNINIT.STACK.ARRAY.PARTIAL.MUST  UNINIT.STACK.MIGHT  UNINIT.STACK.MUST |  |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2025b | CERT C++: EXP53-CPP | Checks for:  Non-initialized variable  Non-initialized pointer  Rule partially covered. |
| PVS-Studio | 7.39 | V546, V573, V614, V670, V679, V730, V788, V1007, V1050 |  |
| RuleChecker | 22.10 | uninitialized-read | Partially checked |
| Security Reviewer - Static Reviewer | 6.02 | C54 C55 C56 C57 C58 C59 C60 C61 C62 C63 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | STR51-CPP — Do not attempt to create a std::string from a null pointer |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer)  **Noncompliant Code** |
| --- |
| A std::string is constructed using the value returned by std::getenv(). If the requested environment variable is missing or an error occurs, std::getenv() may return a null pointer, causing undefined behavior when passed directly to the std::string constructor. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    void f() {    const **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");    if (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** (4) Keep It Simple / (1) Validate Input - Constructing a string from a null pointer results in undefined behavior and potential application crashes. The program should always verify the validity of pointers before use. By validating input and simplifying memory handling logic, developers reduce complexity and prevent runtime failures. This ensures robust handling of external or user-provided data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 9.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2025.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2025.2 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2025b | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |
| Security Reviewer - Static Reviewer | 6.02 | shiftTooManyBits | Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | IDS00-J - Prevent SQL injection |

**Source:** [**https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection**](https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection)

| **Noncompliant Code** |
| --- |
| Unvalidated data could be exploited to change the intended behavior of the query. |
| uName = getRequestString("username");  uPass = getRequestString("userpassword");  sql = “SELECT \* FROM Users WHERE Name = " + uName + " AND Pass = " +  uPass + ” |

| **Compliant Code** |
| --- |
| The main defense against SQL injection is input sanitization and validation, most often enforced through parameterized queries and stored procedures. |
| PreparedStatement pStmt = PreparedStatement();  std::cin >> username;  std::cin >> userpassword;  sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username,  userpassword}; |

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

| **Principles(s):** (7) Sanitize Data Sent to Other Systems / (5) Default Deny - SQL injection vulnerabilities allow attackers to manipulate database queries by inserting malicious input. Using parameterized queries, escaping input, and employing a default deny policy ensures that untrusted input cannot alter database logic. Sanitizing data before transmission reduces exposure to injection-based attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors (see Chapter 8) |
| CodeSonar | 9.0p0 | JAVA.IO.INJ.SQL | SQL injection |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Fortify | 1.0 | HTTP\_Response\_Splitting  SQL\_Injection\_\_Persistence  SQL\_Injection | Implemented |
| Klocwork | 2025.2 | SV.DATA.DB  SV.SQL  SV.SQL.DBSOURCE | Implemented |
| Parasoft Jtest | 2024.2 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SonarQube | 9.9 | S2077  S3649 | [Executing SQL queries is security-sensitive](https://rules.sonarsource.com/java/RSPEC-2077)  [SQL queries should not be vulnerable to injection attacks](https://rules.sonarsource.com/java/RSPEC-3649) |
| SpotBugs | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE  SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | MEM50-CPP - Do not access freed memory |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)  **Noncompliant Code** |
| --- |
| The pointer s is used after it’s been freed. If that use writes to the freed memory, an attacker could exploit it to run code with the program’s privileges. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| The program does not free the allocated memory until it is no longer necessary. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

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

| **Principles(s):** (8) Defense in Depth / (10) Adopt a Secure Coding Standard - Accessing memory after it has been freed leads to undefined behavior and may be exploited for arbitrary code execution. Implementing multiple layers of safety, such as smart pointers and resource ownership models, prevents such vulnerabilities. Adhering to SEI CERT coding standards ensures consistent memory safety practices across all codebases. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 9.1p0 | ALLOC.UAF | Use after free |
| Compass/ROSE |  |  |  |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Helix QAC | 2025.2 | C++4303, C++4304 |  |
| Klocwork | 2025.2 | UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST |  |
| LDRA tool suite | 9.71 | 483 S, 484 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ |  |  |  |
| Polyspace Bug Finder | R2025b | CERT C++: MEM50-CPP | Checks for:  Pointer access out of bounds  Deallocation of previously deallocated pointer  Use of previously freed pointer  Rule partially covered. |
| PVS-Studio | 7.39 | V586, V774 |  |
| Security Reviewer - Static Reviewer | 6.02 | CPP\_12  CPP\_14  CPP\_15 | Fully implemented |
| Splint | 5.0 |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | DCL03-C - Use a static assertion to test the value of a constant expression |

**Source:** [**https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression**](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression)

| **Noncompliant Code** |
| --- |
| Uses assert() to validate an essential property of a memory-mapped structure to ensure correct execution. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| Preprocessor conditional statements can be used with constant expressions. |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** (2) Heed Compiler Warnings / (3) Architect and Design for Security Policies - Using static assertions ensures that assumptions about constants, array sizes, or configurations are validated during compilation. This prevents runtime errors and ensures secure architectural design. By catching issues at compile time, developers can fix logic flaws before deployment, improving long-term system reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 9.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |
| Security Reviewer - Static Reviewer | 6.02 | C13  C14  C15  C52  C129  C130  C132  C133  C135  C154  C155 |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | ERR50-CPP - Do not abruptly terminate the program |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR50-CPP.+Do+not+abruptly+terminate+the+program**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR50-CPP.+Do+not+abruptly+terminate+the+program)  **Noncompliant Code** |
| --- |
| A call to f(), set as an exit handler via std::at\_exit(), could lead to std::terminate() if throwing\_func() raises an exception. |
| #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.    throwing\_func();  }    **int** main() {  **if** (0 != std::**atexit**(f)) {      // Handle error    }    // ...  } |

| **Compliant Code** |
| --- |
| All exceptions raised by throwing\_func() are caught within f(), and none are rethrown. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):** (8) Defense in Depth / (9) Effective Quality Assurance - Abrupt program termination bypasses destructors and cleanup routines, leading to potential resource leaks or data corruption. Graceful error handling with structured exception control ensures system stability. This approach upholds data integrity, supports layered fault recovery, and aligns with defense-in-depth design strategies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stdlib-use | Partially checked |
| CodeSonar | 9.1p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| Helix QAC | 2025.2 | C++5014 |  |
| Klocwork | 2025.2 | MISRA.TERMINATE  CERT.ERR.ABRUPT\_TERM |  |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced Enforcement |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR50-a  CERT\_CPP-ERR50-b  CERT\_CPP-ERR50-c  CERT\_CPP-ERR50-d  CERT\_CPP-ERR50-e  CERT\_CPP-ERR50-f  CERT\_CPP-ERR50-g  CERT\_CPP-ERR50-h  CERT\_CPP-ERR50-i  CERT\_CPP-ERR50-j  CERT\_CPP-ERR50-k  CERT\_CPP-ERR50-l  CERT\_CPP-ERR50-m  CERT\_CPP-ERR50-n | The execution of a function registered with 'std::atexit()' or 'std::at\_quick\_exit()' should not exit via an exception  Never allow an exception to be thrown from a destructor, deallocation, and swap  Do not throw from within destructor  There should be at least one exception handler to catch all otherwise unhandled exceptions  An empty throw shall only be used in the compound-statement of a catch handler  Exceptions shall be raised only after start-up and before termination of the program  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point  Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)  Function called in global or namespace scope shall not throw unhandled exceptions  Always catch exceptions  Properly define exit handlers  The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  Avoid throwing exceptions from functions that are declared not to throw  The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |
| Polyspace Bug Finder | R2025b | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function (rule partially covered) |
| PVS-Studio | 7.39 | V667, V2014 |  |
| RuleChecker | 22.10 | stdlib-use | Partially checked |
| SonarQube C/C++ Plugin | 4.10 | S990 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-LLL] | OOP53-CPP - Write constructor member initializers in the canonical order |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order)  **Noncompliant Code** |
| --- |
| The constructor C::C() uses a member initializer list that initializes someVal first, followed by dependsOnSomeVal, which relies on someVal. Because the order of member declarations does not align with the initializer list, dependsOnSomeVal receives an unspecified value when reading someVal. |
| **class** C {  **int** dependsOnSomeVal;  **int** someVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| Modify the class member declaration order so the constructor’s initializer list reflects the intended dependency sequence. |
| **class** C {  **int** someVal;  **int** dependsOnSomeVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** (4) Keep It Simple / (9) Effective Quality Assurance - C++ initializes members in their declaration order, not in the order listed in the constructor. Following canonical order avoids undefined behavior from using uninitialized members. Keeping initialization simple and consistent improves code readability, debugging, and reliability during object construction. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | initializer-list-order | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP53 |  |
| Clang | 3.9 | -Wreorder |  |
| CodeSonar | 9.1p0 | LANG.STRUCT.INIT.OOMI | Out of Order Member Initializers |
| Helix QAC | 2025.2 | C++4053 |  |
| Klocwork | 2025.2 | CERT.OOP.CTOR.INIT\_ORDER |  |
| LDRA tool suite | 9.7.1 | 206 S | Fully implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-OOP53-a | List members in an initialization list in the order in which they are declared |
| Polyspace Bug Finder | R2025b | CERT C++: OOP53-CPP | Checks for members not initialized in canonical order (rule fully covered) |
| RuleChecker | 22.10 | initializer-list-order | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | S3229 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | CTR53-CPP - Use valid iterator ranges |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges)  **Noncompliant Code** |
| --- |
| With each iteration, std::for\_each() increments the starting iterator and tests it against the ending iterator. The loop proceeds while they differ. Attempting to increment the past-the-end iterator leads to undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| Iterators are passed to std::for\_each() in the correct order. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):** (9) Effective Quality Assurance / (4) Keep It Simple - Iterators must refer to valid ranges to prevent out-of-bounds memory access and undefined behavior. Adhering to this standard simplifies algorithm implementation and enhances safety during iteration. Using range-based loops and performing boundary checks ensures predictable program behavior |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| CodeSonar | 9.1p0 | LANG.MEM.BO | Buffer Overrun |
| Helix QAC | 2025.2 | C++3802 |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |
| Polyspace Bug Finder | R2025b | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered). |
| PVS-Studio | 7.39 | V539, V662, V789 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-010-CPP] | EXP54-CPP - Do not access an object outside of its lifetime |

| **Source:** [**https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP54-CPP.+Do+not+access+an+object+outside+of+its+lifetime**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP54-CPP.+Do+not+access+an+object+outside+of+its+lifetime)  **Noncompliant Code** |
| --- |
| Using a pointer to access a non-static member function of an object prior to the start of its lifetime causes undefined behavior. |
| struct S {  void mem\_fn();  };    void f() {  S \*s;  s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| Allocation for the pointer takes place before calling S::mem\_fn(). |
| struct S {    void mem\_fn();  };    void f() {    S \*s = new S;    s->mem\_fn();    delete s;  } |

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

| **Principles(s):** (8) Defense in Depth / (10) Adopt a Secure Coding Standard - Accessing objects before construction or after destruction results in undefined behavior and can be exploited for memory corruption or code execution. Enforcing lifetime management through RAII (Resource Acquisition Is Initialization) ensures object safety. This layered approach to memory management prevents logical and security errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | return-reference-local  dangling\_pointer\_use | Partially checked |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<> |
| CodeSonar | 9.1p0 | IO.UAC  ALLOC.UAF | Use after close Use after free |
| Helix QAC | 2025.2 | C++4003, C++4026  DF2812, DF2813, DF2814, DF2930, DF2931, DF2932, DF2933, DF2934, |  |
| Klocwork | 2025.2 | CL.FFM.ASSIGN  CL.FFM.COPY  LOCRET.ARG  LOCRET.GLOB  LOCRET.RET  UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST  UNINIT.STACK.ARRAY.MIGHT  UNINIT.STACK.ARRAY.MUST  UNINIT.STACK.ARRAY.PARTIAL.MUST  UNINIT.STACK.MIGHT  UNINIT.STACK.MUST |  |
| LDRA tool suite | 9.7.1 | 42 D, 53 D, 77 D, 1 J, 71 S, 565 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | Do not use resources that have been freed The address of an object with automatic storage shall not be returned from a function The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2025b | [CERT C++: EXP54-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp54cpp.html) | Checks for:  Non-initialized variable or pointer  Use of previously freed pointer  Pointer or reference to stack variable leaving scope  Accessing object with temporary lifetime  Rule partially covered. |
| PVS-Studio | 7.39 | V758, V1041, V1099 |  |
| RuleChecker | 22.10 | return-reference-local | Partially checked |

### 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.

Automation will be used for the enforcement and compliance of the standards defined in this policy. The DevSecOps diagram illustrates how security is embedded across the software development lifecycle—from planning through production—ensuring that automation is integrated at every stage to enforce coding standards, run security scans, and monitor for vulnerabilities.

In the Assess and Plan phase, automation tools can continuously analyze source repositories for dependencies and configuration risks. Automated backlog prioritization tools ensure that high-severity vulnerabilities are addressed early in the sprint cycle. Threat modeling software and compliance checkers validate alignment with Green Pace’s secure development policy before any coding begins.

During the Design and Build stages, integrated static application security testing (SAST) and dependency scanning tools (such as SonarQube, Fortify, or GitHub Advanced Security) automatically review code for SEI CERT compliance, unsafe functions, and memory management flaws. Code is committed only after passing automated security and quality gates, enforcing standards like DCL60-CPP, MEM50-CPP, and STR51-CPP.

In the Verify and Test phase, automated test suites perform unit, integration, and fuzz testing, combined with dynamic analysis (DAST) and configuration scanning to detect runtime vulnerabilities. Continuous Integration (CI) pipelines ensure that these tests run on every code commit, enforcing compliance without manual intervention.

As software transitions into Production, automation continues with health checks, configuration validation, and penetration test scheduling to confirm deployment security. The Monitor and Detect stage employs SIEM (Security Information and Event Management) systems and intrusion detection tools to continuously log and correlate security events in real time.

Finally, in the Respond and Maintain/Stabilize stages, automated rollback scripts, container immutability, and baseline assessments ensure systems can quickly recover after incidents. These automated mechanisms support continuous compliance with the Green Pace Secure Development Policy, reducing human error while maintaining defense-in-depth across the entire DevOps pipeline.

### 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 | Unlikely | High | Medium (6) | 2 |
| STD-002-CPP | High | Probable | Medium | High (12) | 1 |
| STD-003-CPP | High | Likely | Medium | High (18) | 1 |
| STD-004-CPP | High | Likely | Medium | High (18) | 1 |
| STD-005-CPP | High | Likely | Medium | Medium (9) | 2 |
| STD-006-CPP | Low | Unlikely | High | Low (3) | 3 |
| STD-007-CPP | Low | Probable | Medium | Low (2) | 3 |
| STD-008-CPP | Medium | Unlikely | Medium | Medium (6) | 2 |
| STD-009-CPP | High | Probable | High | Medium (6) | 2 |
| STD-010-CPP | High | Probable | High | Medium (6) | 2 |

### 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 | Protects stored data in databases, logs, and files using AES-256 or stronger encryption. Prevents unauthorized access from stolen disks or backups. Applies to all production storage and persistent media. |
| Encryption in flight | Secures data transmission between clients, APIs, and services using TLS 1.3 or higher. Prevents man-in-the-middle attacks and packet sniffing. Applies to all internal and external network communications. |
| Encryption in use | Protects data while being processed in memory through techniques like hardware enclaves (Intel SGX) and tokenization. Applies to sensitive workloads handling PII or financial data to limit exposure during runtime. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies user identity through multi-factor methods (SAML SSO, TOTP). Ensures only authorized users access systems. Applied to all login portals and API tokens. |
| Authorization | Grants least-privilege access based on roles and attributes (RBAC/ABAC). Controls what authenticated users can do. Applied to every microservice and database query. |
| Accounting | Logs all access and administrative changes. Provides evidence for compliance and incident response. Covers logins, data modifications, user additions, and file access activities. |

**\***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 | 09/21/2025 | Module 3 Milestone | Jessica Ramirez | [Insert text.] |
| 1.2 | 10/12/2025 | Module 6 Milestone | Jessica Ramirez | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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