**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 | Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user-controlled files |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Create a software architecture and design your software to implement and enforce security policies. |
| 1. Keep It Simple | Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex. |
| 1. Default Deny | Base access decisions on permission rather than exclusion. This means that, by default, access is denied, and the protection scheme identifies conditions under which access is permitted |
| 1. Adhere to the Principle of Least Privilege | Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem. |
| 1. Practice Defense in Depth | Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment. |
| 1. Use Effective Quality Assurance Techniques | Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions. |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for your target development language and platform. |

### 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 | Do not cast to an out-of-range enumeration value |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);      if (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a scoped enumeration, which has a fixed underlying int type by default, allowing any value from the parameter to be converted into a valid enumeration value. It does not restrict the converted value to one for which there is a specific enumerator value, but it could do so as shown in the previous compliant solution. |
| [Compliant code block; code should be indented using 12-point Courier New font.] |

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

| enum class EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);  } |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ - INT50 |  |
| CodeSonar | 6.2p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2022.1 | C++3013 |  |
| Parasfot C/C++ Test | 2021.2 | CERT\_CPP – INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PRQA QA-C++ | 4.4 | 3013 |  |
| PVS-Studio | 7.18 | V1016 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Guarantee that container indices and iterators are within the valid range. |

| **Noncompliant Code** |
| --- |
| his noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Overflow\_upon\_dereference |  |
| CodeSonar | 6.2p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TO  LANG.MEM.TU  LANG.MEM.TBA  LANG.STRUCT.PBB  LANG.STRUCT.PPE | Buffer overrun  Buffer underrun  Type overrun  Type underrun  Tainted buffer access  Pointer before beginning of object  Pointer past end of object |
| Helix QAC | 2022.1 | C++2891, C++3139, C++3140 |  |
| Klocwork | 2022.1 | ABV.ANY\_SIZE\_ARRAY  ABV.GENERAL  ABV.STAACK  ABV.TAINTED  SV.TAINTED.ALLOC\_SIZE  SV.TAINTED.CALL.INDEX\_ACCESS  SV.TAINTED.CALL.LOOP\_BOUND  SV.TAINTED.INDEX\_ACCESS |  |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 4176 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X | Partially implemented |
| Polyspace Bug Finder | R2021b | CERT C++:CTR50-CPP | Checks for   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered |
| PRQA QA-C++ | 4.4 | 2891, 3139, 3140 |  |
| PVS-Studio | 7.18 | V781 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    void f() {    char buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No Space for null terminator  Buffer overrun  Type overrun |
| Helix QAC | 2022.1 | C++2835, C++2836, C++2839, C++5216 |  |
| Klocwork | 2022.1 | NNTS.MIGHT  NNTS.TAINTED  NNTS.MUST  SV.UNBOUND\_STRING\_INPUT.CIN |  |
| LDRA tool suite | 9.7.1 | 489 S, 66X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-C  CERT\_CPP-STR50-E  CERT\_CPP-STR50-F  CERT\_CPP-STR50-G | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the ‘char’ buffer to store input from ‘std::cin’ |
| Polyspace Bug Finder | R2021b | CERT C++:STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * destination buffer overflow in string manipulation   Rule partially covered |
| SonarQube C/C++ Plugin | 4.10 | S3519 |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #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):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | 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 | 6.2p0 | ALLOC.UAF | User 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 WAC | 2022.1 | C++4303, C++4304 |  |
| Klocwork | 2022.1 | 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 | 9.7.1 | 483 S, 484 S | Partially implemented |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-MEM50-a | Do not use resources that have been free |
| Parasoft Bug Finder | R2021b | CERT C++:MEM50-CPP | Checks for:   * Pointer out of bounds * Deallocation of previously deallocated pointer * Use of previously free pointer |
| PRQA QA-C++ | 4.4 | 4304, 4304 |  |
| PVS-studio | 7.18 | V586, V774 |  |
| Splint | 5.0 |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the local variable space is passed as the expression to the placement new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined behavior due to ::operator delete() attempting to free memory that was not returned by ::operator new(). |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      delete s1;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. This is one of the few times when explicitly invoking a destructor is warranted. |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      s1->~S();  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Invalid\_dynamic\_memory\_allocation  Dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM51 |  |
| Clang | 3.9 | Clang-analyzer-cplusplus.NewDeleteLeaks-Wmistmatched-new-delete  Clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy. But does not catch all violations of this rule |
| CodeSonar | 6.2p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TIM | Free non-heap variable  Double free  Type mismatch |
| Klocwork | 2022.1 | CL.FFM.ASSIGN  CL.FFM.COPY  CL FMM  CL.SHALLOW.ASSIGN  CL.SHALLOW.COPY  FMM.MIGHT.FMM.MUST  FNH.MIGHT  FNH.MUST  FUM.GEN.MIGHT  FUM.GEN.MUST  UNINIT.CTOR.MIGHT  UNINIT.CTOR.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST |  |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resources |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2021b | CERT C++:MEM51-CPP | Checks for:   * Invalid detection of pointer * Invalid free of pointer * Deallocation of previously deallocated pointer   Rule partially covered |
| PRQA QA-C++ | 4.4 | 2110, 2111, 2112, 2113,2118  3337,3339.4262,4263,4264 |  |
| PVS-Studio | 7.18 | V515,V554,V611,V701,V773,V1066 |  |
| SonarQube C/C++ Plugin | 4.10 | S1232 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Do not declare more than one variable per declaration |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a programmer or code reviewer might mistakenly believe that the two variables src and c are declared as char \*. In fact, src has a type of char \*, whereas c has a type of char. |
| char \*src = 0, c = 0; |

| **Compliant Code** |
| --- |
| In this compliant solution, each variable is declared on a separate line: |
| char \*src;    /\* Source string \*/  char c;       /\* Character being tested \*/ |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL04 |  |
| CodeSonar | 6.2p0 | LANG.STRUCT.DECL.ML | Multiple Declarations on Line |
| ÉCLAIR | 1.2 | CCS2.DCL04 | Fully implemented |
| LDRA tool suite | 9.7.1 | 579 S | Fully implemented |
| Parasoft C/C++ test | 2021.1 | CERT\_C-DCL04-a | Each variable should be declared in a separate declaration statement |
| PC-lint Plus | 1.4 | 9146 | Partially supported: exceptions not supported |
| SonarQube C/C++ Plugin | 3.11 | SingleDeclarationPerStatement |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| Helix QAC | 2022.1 | C++4035,C++4036,C++4037 |  |
| Klocwork | 2022.1 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2021b | CERT C++:ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| PRQA QA-C++ | 4.4 | 4035, 4036, 4037 |  |
| RuleChecker | 20.10 | Main-function-catch-a;;  Early-catch-all | Partially covered |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | Do not invoke virtual functions from constructors or destructors |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| struct B {    B() { seize(); }    virtual ~B() { release(); }    protected:    virtual void seize();    virtual void release();  };    struct D : B {    virtual ~D() = default;    protected:    void seize() override {      B::seize();      // Get derived resources...    }      void release() override {      // Release derived resources...      B::release();    }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources. |
| class B {    void seize\_mine();    void release\_mine();    public:    B() { seize\_mine(); }    virtual ~B() { release\_mine(); }    protected:    virtual void seize() { seize\_mine(); }    virtual void release() { release\_mine(); }  };    class D : public B {    void seize\_mine();    void release\_mine();    public:    D() { seize\_mine(); }    virtual ~D() { release\_mine(); }    protected:    void seize() override {      B::seize();      seize\_mine();    }      void release() override {      release\_mine();      B::release();    }  }; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Virtual-call-in-constructor  Invalid\_function\_pointer | Fully checked |
| Clang | 3.9 | Clang-analyzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP50 |  |
| CodeSonar | 6.2p0 | LANG.STRUCT.VCALL\_IN\_CTOR  LANG.STRUCT.VCALL\_IN\_DTOR | Virtual Call in Constructor  Virtual Call in Destructor |
| Helix QAC | 2022.1 | C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282 |  |
| LDRA tool suite | 9.7.1 | 467 S, 92 D | Fully implemented |
| Parasoft C/C++ | 2021.2 | CERT\_CPP-OPP50-a  CERT\_CPP-OPP50-b  CERT\_CPP-OPP50-c  CERT\_CPP-OPP50-d | Avoid calling virtual functions from constructors  Avoid calling virtual functions from destructors  Do not invoke class’s virtual functions from any of its constructors  Do not involve class’s virtual functions from its desctructors |
| Polyspace Bug Finder | R2021b | CERT C++:OOP50-CPP | Checks for virtual function from constructors and destructors (rule fully covered) |
| PRQA QA-C++ | 4.4 | 4260, 4261, 4272, 4274, 4275, 4276, 4277, 4278, 4279, 4280, 4281, 4282 |  |
| PVS-Studio | 7.18 | V1052 |  |
| RuleChecker | 20.10 | Virtual-call-in-constructor | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | S1699 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-009-CPP | DO not refer to an atomic variable twice in an expression |

| **Noncompliant Code** |
| --- |
| This noncompliant code example declares a shared atomic\_bool flag variable and provides a toggle\_flag() method that negates the current value of flag: |
| #include <stdatomic.h>  #include <stdbool.h>    static atomic\_bool flag = ATOMIC\_VAR\_INIT(false);    void init\_flag(void) {    atomic\_init(&flag, false);  }    void toggle\_flag(void) {    bool temp\_flag = atomic\_load(&flag);    temp\_flag = !temp\_flag;    atomic\_store(&flag, temp\_flag);  }    bool get\_flag(void) {    return atomic\_load(&flag);  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a compare-and-exchange to guarantee that the correct value is stored in flag. All updates are visible to other threads. The call to atomic\_compare\_exchange\_weak() is in a loop in conformance with CON41-C |
| #include <stdatomic.h>  #include <stdbool.h>    static atomic\_bool flag = ATOMIC\_VAR\_INIT(false);    void init\_flag(void) {  atomic\_init(&flag, false);  }    void toggle\_flag(void) {  bool old\_flag = atomic\_load(&flag);  bool new\_flag;  do {  new\_flag = !old\_flag;  } while (!atomic\_compare\_exchange\_weak(&flag, &old\_flag, new\_flag));  }    bool get\_flag(void) {  return atomic\_load(&flag);  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Multiple-atomic-accesses | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-CON40 |  |
| CodeSonar | 6.2p0 | CONCURRENCY.MAA | Multiple Accesses of Atomic |
| Coverity | 2017.07 | EVALUATION\_ORDER(partial)  MISRA 2012 Rule 13.2  VOLATILE\_ATOICITY (possible) | Implemented |
| Helix QAC | 2022.1 | C1114, C1115, C1116  C++3171, C++4150 |  |
| Klocwork | 2022.1 | CERT.CONC.ATOMIC\_TWICE\_EXPR |  |
| Parasoft C/C++ test | 2021.2 | CERT\_CON40-a | Do not refer to an atomic variable twice in an expression |
| Polyspace Bug Finder | R2021a | CERT C: Rule CON40-C | Checks for:   * Atomic variable accesses twice in an expression * Atomic load and store sequence not atomic |
| PRQA QA-C | 9.7 | 1114, 1115, 1116 |  |
| RuleChecler | 20.10 | Multiple-atomic-accesses | Partially checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-010-CPP | Arguments to character-handling functions must be representable as an unsigned char |

| **Noncompliant Code** |
| --- |
| On implementations where plain char is signed, this code example is noncompliant because the parameter to isspace(), \*t, is defined as a const char \*, and this value might not be representable as an unsigned char: |
| #include <ctype.h>  #include <string.h>    size\_t count\_preceding\_whitespace(const char \*s) {    const char \*t = s;    size\_t length = strlen(s) + 1;    while (isspace(\*t) && (t - s < length)) {      ++t;    }    return t - s;  } |

| **Compliant Code** |
| --- |
| This compliant solution casts the character to unsigned char before passing it as an argument to the isspace() function: |
| #include <ctype.h>  #include <string.h>    size\_t count\_preceding\_whitespace(const char \*s) {    const char \*t = s;    size\_t length = strlen(s) + 1;    while (isspace((unsigned char)\*t) && (t - s < length)) {      ++t;    }    return t - s;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Ctype-limits | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR37 | Fully implemented |
| CodeSonar | 6.2p0 | MISC.NEGCHAR | Negative character value |
| Compass/ROSE |  |  | Could not detect violations of this rule by seeing if the argument to a character handling function (listed above) is not an unsigned char |
| ÉCLAIR | 1.2 | CCS.STR37 | Fully implemented |
| Helix QAC | 2022.1 | C4413, C4414  C++3051 |  |
| Klocwork | 2022.1 | AUTOSAR.STDLIB.CCTYPE.UCHAR  MISRA.ETYPE.ASSIGN.2012 |  |
| LDRA tool suite | 9.7.1 | 663 S | Fully implemented |
| Parasoft C/C+++test | 2021.2 | CERT\_C-STR37-a | Do not pass incorrect values to ctype.h library functions |
| Polyspace Bug Funder | R2021a | CERT C:Rule STR37-C | Checks for invalid use of standard library integer routine (rule fully covered) |
| PRQA QA-C | 9.7 | 4413, 4414 | Fully implemented |
| PRQA QA-C++ | 4.4 | 3051 |  |
| RuleChecker | 20.10 | Ctype-limits | Partially checked |
| TrustInSoft Analyzer | 1.38 | Valid\_char | Partially verified |

### 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 | Unlikely | Medium | 4 | 3 |
| STD-002-CPP | High | Likely | High | 9 | 2 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CPP | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Unlikely | Low | 3 | 3 |
| STD-007-CPP | Low | Probably | Medium | 4 | 3 |
| STD-008-CPP | Low | Unlikely | Medium | 2 | 3 |
| STD-009-CPP | Medium | Probably | Medium | 8 | 2 |
| STD-010-CPP | Low | Unlikely | Low | 3 | 3 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 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 in rest | Encryption at rest is the process of securely encrypting data that is not accessed on a server through a program or application. Encryption at rest is typically data that is secured behind a firewall or some sort of anti-virus software. |
| Encryption at flight | Unlike data at rest, data in flight is data that is transmitted from a server to a program that could be on a phone or PC. Encryption at flight is the process of securing the data being transmitted as it makes its way to the users PC or mobile device. Emails are secured for many purposes; they hold company and/or private information that only the user and the end user are supposed to see. By encrypting the data as it is being sent from A to B, you are encrypting in flight. |
| Encryption in use | Encryption in use is the security protocol for protecting data as it is being transmitted. For instance, logging into a bank application requires multiple levels of security, typically a 2 factor authentication is in use. By not securing a persons login information, and then what that information is after logging in, a user is at risk of having their bank information stolen. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of providing information to prove who you are. This is typically done by use of username and password, and then an additional 2 factor authentication or token of some kind. |
| Authorization | The granting of specific privileges once a user has been authenticated. These privileges can be changed by an admin by granting more or less access to specific areas of the network. The admin is also the one to add and grant new users access. |
| Accounting | Keeping a record of user login information, and a history of what the user has done once they have been authenticated. Accounting is able to verify which files were accessed by users and at what time this was done. |

**\***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
  + - 3 - keeping a history of design and security changes of the system
    - 4 – keep the history logs simple as can be so there is no confusion
    - 5 – keep track of denials to prevent future intrusions
    - 6 – prevents those without proper authorization from gaining access
    - 8 – logs should not be easily accessible to just anyone
    - 10 - keeps everything simple and everyone aware
* Firewall logs
  + - 1 – if firewall is accessed then it is good to know by whom
    - 5 – keep track of denials to prevent future intrusions
    - 8 – logs should not be easily accessible to just anyone
    - 10 – keeps everything simple and everyone aware
* Anti-malware logs
  + - 4 – keep the history logs simple as can be so there is no confusion
    - 5 – keep track of denials to prevent future intrusions
    - 6 – prevents those without proper authorization from gaining access
    - 8 – logs should not be easily access to just anyone
    - 10 – keeps everything simple and everyone aware

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 |  |
| 2.0 | 04/09/2021 | Project One | Samuel Powers | Trevor Hodde |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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