**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 | This principle involves verifying the data that is entered into a system to ensure it is of the expected type, format, and length. This helps prevent potential security vulnerabilities such as injection attacks that could be caused by invalid or malicious data. |
| 1. Heed Compiler Warnings | This principle involves paying attention to any warnings or errors reported by the compiler during the software development process. These warnings can provide insight into potential security issues and should be addressed promptly. |
| 1. Architect and Design for Security Policies | This principle involves incorporating security policies and requirements into the design of a system from the outset. This includes identifying potential security threats and implementing appropriate measures to mitigate those threats. |
| 1. Keep It Simple | This principle involves designing software with simplicity in mind, which can reduce the likelihood of errors and vulnerabilities. Complex software can be more difficult to secure and can increase the likelihood of bugs and vulnerabilities. |
| 1. Default Deny | This principle involves configuring systems to deny access by default, and only granting access to authorized users or processes. This helps prevent unauthorized access to systems and data. |
| 1. Adhere to the Principle of Least Privilege | This principle involves granting users or processes only the minimum level of access necessary to perform their required functions. This helps reduce the potential impact of a security breach or vulnerability. |
| 1. Sanitize Data Sent to Other Systems | This principle involves ensuring that any data sent to other systems is properly sanitized and validated to prevent the introduction of vulnerabilities or malicious code. |
| 1. Practice Defense in Depth | This principle involves implementing multiple layers of security measures, such as firewalls, access controls, and intrusion detection systems, to protect against potential threats. |
| 1. Use Effective Quality Assurance Techniques | This principle involves incorporating quality assurance practices into the software development process to ensure that code is tested thoroughly for security vulnerabilities and other issues. |
| 1. Adopt a Secure Coding Standard | This principle involves using a set of coding standards or guidelines that incorporate security best practices, such as avoiding insecure coding practices, to help reduce the likelihood of security vulnerabilities. |

### 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** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Enumerations in C++ come in two forms: scoped enumerations in which the underlying type is fixed and unscoped enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. The range of valid enumeration values for an enumeration type is defined by the C++ Standard, [dcl.enum], in paragraph 8 [ISO/IEC 14882-2020]:  For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, the values of the enumeration are the values representable by a hypothetical integer type with minimal width M such that all enumerators can be represented. The width of the smallest bit-field large enough to hold all the values of the enumeration type is M. It is possible to define an enumeration that has values not defined by any of its enumerators. If the enumerator-list is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.  The C++ Standard, [expr.static.cast], paragraph 10, states the following:  A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the enumeration type has a fixed underlying type, the value is first converted to that type by integral conversion, if necessary, and then to the enumeration type. If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values (9.7.1), and otherwise, the behavior is undefined. A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (7.3.10), and subsequently to the enumeration type.  To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. |

| **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. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| **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 checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** It is possible for unspecified values to result in a buffer overflow, leading to the execution of arbitrary code by an attacker. However, because enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will result in data integrity violations rather than arbitrary code execution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | cast-integer-to-enum | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 |  |
| Helix QAC | 2023.1 | C++3013 |  |
| Parasoft C/C++test | 2022.2 | 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 | 7.24 | V1016 |  |
| RuleChecker | 22.10 | cast-integer-to-enum | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use valid references, pointers, and iterators to reference elements of a basic\_string** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Since std::basic\_string is a container of characters, this rule is a specific instance of CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container. As a container, it supports iterators just like other containers in the Standard Template Library. However, the std::basic\_string template class has unusual invalidation semantics. The C++ Standard, [string.require], paragraph 5 [ISO/IEC 14882-2014], states the following:  References, pointers, and iterators referring to the elements of a basic\_string sequence may be invalidated by the following uses of that basic\_string object:  As an argument to any standard library function taking a reference to non-const basic\_string as an argument.  Calling non-const member functions, except operator[], at, front, back, begin, rbegin, end, and rend.  Examples of standard library functions taking a reference to non-const std::basic\_string are std::swap(), ::operator>>(basic\_istream &, string &), and std::getline().  Do not use an invalidated reference, pointer, or iterator because doing so results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example copies input into a std::string, replacing semicolon (;) characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

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

| **Principles(s):** Using an invalid reference, pointer, or iterator to a string object could allow an attacker to run arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2023.1 | DF4746, DF4747, DF4748, DF4749 |  |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2023a | CERT C++: STR52-CPP | Checks for use of invalid string iterator (rule partially covered). |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The std::basic\_string type uses the traits design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Specifically, the std::basic\_string class is paired with std::char\_traits to create the std::string, std::wstring, std::u16string, and std::u32string classes. The std::char\_traits class is explicitly specialized to provide policy-based implementation details to the std::basic\_string type. One such implementation detail is the std::char\_traits::length() function, which is frequently used to determine the number of characters in a null-terminated string. According to the C++ Standard, [char.traits.require], Table 62 [ISO/IEC 14882-2014], passing a null pointer to this function is undefined behavior because it would result in dereferencing a null pointer. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <iostream>    **void** f(**char** c) {  **char** &p = c;    p = 'p';    std::cout << c << std::endl;  } |

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

| **Principles(s):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low. |
| --- |

**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 | 7.3p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2023.1 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2023.1 | 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 | 2022.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace bug | R2023a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Do not store an already-owned pointer value in an unrelated smart pointer** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Smart pointers such as std::unique\_ptr and std::shared\_ptr encode pointer ownership semantics as part of the type system. They wrap a pointer value, provide pointer-like semantics through operator \*() and operator->() member functions, and control the lifetime of the pointer they manage. When a smart pointer is constructed from a pointer value, that value is said to be owned by the smart pointer.  Calling std::unique\_ptr::release() will relinquish ownership of the managed pointer value. Destruction of, move assignment of, or calling std::unique\_ptr::reset() on a std::unique\_ptr object will also relinquish ownership of the managed pointer value, but results in destruction of the managed pointer value. If a call to std::shared\_ptr::unique() returns true, then destruction of or calling std::shared\_ptr::reset() on that std::shared\_ptr object will relinquish ownership of the managed pointer value but results in destruction of the managed pointer value.  Some smart pointers, such as std::shared\_ptr, allow multiple smart pointer objects to manage the same underlying pointer value. In such cases, the initial smart pointer object owns the pointer value, and subsequent smart pointer objects are related to the original smart pointer. Two smart pointers are related when the initial smart pointer is used in the initialization of the subsequent smart pointer objects. For instance, copying a std::shared\_ptr object to another std::shared\_ptr object via copy assignment creates a relationship between the two smart pointers, whereas creating a std::shared\_ptr object from the managed pointer value of another std::shared\_ptr object does not.  Do not create an unrelated smart pointer object with a pointer value that is owned by another smart pointer object. This includes resetting a smart pointer's managed pointer to an already-owned pointer value, such as by calling reset(). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability. |
| #include <memory>    **void** f() {  **int** \*i = **new** **int**;    std::shared\_ptr<**int**> p1(i);    std::shared\_ptr<**int**> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::shared\_ptr objects are related to one another through copy construction. When the local, automatic variable p2 is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local variable. |
| #include <memory>    **void** f() {    std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();    std::shared\_ptr<**int**> p2(p1);  } |

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

| **Principles(s):** Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | dangling\_pointer\_use |  |
| Axivioin Bauhaus Suite | 7.2.0 | CertC++-MEM56 |  |
| Helix QAC | 2023.1 | DF4721, DF4722, DF4723 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| Polyspace Bug Finder | R2023a | CERT C++: MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |
| PVS-Studio | 7.24 | V1006 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CCP] | The C programming language provides several ways to allocate memory, such as std::malloc(), std::calloc(), and std::realloc(), which can be used by a C++ program. However, the C programming language defines only a single way to free the allocated memory: std::free(). See MEM31-C. Free dynamically allocated memory when no longer needed and MEM34-C. Only free memory allocated dynamically for rules specifically regarding C allocation and deallocation requirements.  The C++ programming language adds additional ways to allocate memory, such as the operators new, new[], and placement new, and allocator objects. Unlike C, C++ provides multiple ways to free dynamically allocated memory, such as the operators delete, delete[](), and deallocation functions on allocator objects. |

| **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):** Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | invalid\_dynamic\_memory\_allocation  dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM51 |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 7.3p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| Helix QAC | 2023.1 | C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264 |  |
| Klocwork | 2023.1 | 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, 483 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |
| Parasoft C/C++test | 2022.2 | 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 | R2023a | CERT C++: MEM51-CPP | Checks for:   * Invalid deletion 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 | 7.24 | V515, V554, V611, V701, V748, V773, V1066 |  |
| SonarQube C/C++ Plugin | 4.10 | S1232 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Avoid information leakage when passing a class object across a trust boundary** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | The C++ Standard, [class.mem], paragraph 13 [ISO/IEC 14882-2014], describes the layout of non-static data members of a non-union class, specifying the following:   * Nonstatic data members of a (non-union) class with the same access control are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified. Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions and virtual base classes.   Further, [class.bit], paragraph 1, in part, states the following:   * Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit.   Thus, padding bits may be present at any location within a class object instance (including at the beginning of the object, in the case of an unnamed bit-field as the first member declared in a class). Unless initialized by zero-initialization, padding bits contain indeterminate values that may contain sensitive information.  When passing a pointer to a class object instance across a trust boundary to a different trusted domain, the programmer must ensure that the padding bits of such an object do not contain sensitive information. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, arg is value-initialized through direct initialization. Because test does not have a user-provided default constructor, the value-initialization is preceded by a zero-initialization that guarantees the padding bits are initialized to 0 before any further initialization occurs. It is akin to using std::memset() to initialize all of the bits in the object to 0. |
| #include <cstddef>    **struct** test {  **int** a;  **char** b;  **int** c;  };    // Safely copy bytes to user space  **extern** **int** copy\_to\_user(**void** \*dest, **void** \*src, std::**size\_t** size);    **void** do\_stuff(**void** \*usr\_buf) {    test arg{};      arg.a = 1;    arg.b = 2;    arg.c = 3;      copy\_to\_user(usr\_buf, &arg, **sizeof**(arg));  } |

| **Compliant Code** |
| --- |
| This compliant solution serializes the structure data before copying it to an untrusted context. |
| #include <cstddef>  #include <cstring>    **struct** test {  **int** a;  **char** b;  **int** c;  };    // Safely copy bytes to user space.  **extern** **int** copy\_to\_user(**void** \*dest, **void** \*src, std::**size\_t** size);    **void** do\_stuff(**void** \*usr\_buf) {    test arg{1, 2, 3};    // May be larger than strictly needed.    unsigned **char** buf[**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);      copy\_to\_user(usr\_buf, buf, offset /\* size of info copied \*/);  } |

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

| **Principles(s):** Padding bits might inadvertently contain sensitive data such as pointers to kernel data structures or passwords. A pointer to such a structure could be passed to other functions, causing information leakage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL55 |  |
| CodeSonar | 7.3p0 | MISC.PADDING.POTB | Padding Passed Across a Trust Boundary |
| Helix QAC | 2023.1 | DF4941, DF4942, DF4943 |  |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-DCL55-a | A pointer to a structure should not be passed to a function that can copy data to the user space |
| Polyspace Bug Finder | R2023a | CERT C++: DCL55-CPP | Checks for information leakage due to structure padding (rule partially covered) |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Honor exception specifications** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | The C++ Standard, [except.spec], paragraph 8 [ISO/IEC 14882-2014], states the following:   * A function is said to allow an exception of type E if the constant-expression in its noexcept-specification evaluates to false or its dynamic-exception-specification contains a type T for which a handler of type T would be a match (15.3) for an exception of type E.   If a function throws an exception other than one allowed by its exception-specification, it can lead to an implementation-defined termination of the program ([except.spec], paragraph 9).  If a function declared with a dynamic-exception-specification throws an exception of a type that would not match the exception-specification, the function std::unexpected() is called. The behavior of this function can be overridden but, by default, causes an exception of std::bad\_exception to be thrown. Unless std::bad\_exception is listed in the exception-specification, the function std::terminate() will be called.  Similarly, if a function declared with a noexcept-specification throws an exception of a type that would cause the noexcept-specification to evaluate to false, the function std::terminate() will be called.  Calling std::terminate() leads to implementation-defined termination of the program. To prevent abnormal termination of the program, any function that declares an exception-specification should restrict itself, as well as any functions it calls, to throwing only allowed exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) noexcept(**true**) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) {    v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):** Throwing unexpected exceptions disrupts control flow and can cause premature termination and denial of service. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | unhandled-throw-noexcept | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR55 |  |
| CodeSonar | 7.3p0 | LANG.STRUCT.EXCP.THROW | Use of throw |
| Helix QAC | 2023.1 | C++4035, C++4036, C++4632 |  |
| LDRA tool suite | 9.7.1 | 56 D | Partially implemented |
| Parasoft C/C++Test | 2022.2 | CERT\_CPP-ERR55-a | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |
| Polyspace Bug Finder | R2023a | CERT C++: ERR55-CPP | Checks for noexcept functions exiting with exception (rule fully covered) |
| PRQA QA-C++ | 4.4 | 4035, 4036, 4632 |  |
| RuleChecker | 22.10 | unhandled-throw-noexcept | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input Output | [STD-008-CPP] | A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.  Note that std::basic\_ifstream<T>, std::basic\_ofstream<T>, and std::basic\_fstream<T> all maintain an internal reference to a std::basic\_filebuf<T> object on which open() and close() are called as needed. Properly managing an object of one of these types (by not leaking the object) is sufficient to ensure compliance with this rule. Often, the best solution is to use the stream object by value semantics instead of via dynamic memory allocation, ensuring compliance with MEM51-CPP. Properly deallocate dynamically allocated resources. However, that is still insufficient for situations in which destructors are not automatically called. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {    std::fstream file(fileName);  **if** (!file.is\_open()) {      // Handle error  **return**;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {    {      std::fstream file(fileName);  **if** (!file.is\_open()) {        // Handle error  **return**;      }    } // file is closed properly here when it is destroyed    std::terminate();  } |

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

| **Principles(s):** Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Supported, but no explicit checker |
| CodeSonar | 7.3p0 | ALLOC.LEAK | Leak |
| Compass/ROSE |  |  |  |
| Coverity | 2017.07 | RESOURCE\_LEAK (partial) | Partially implemented |
| Helix QAC | 2023.1 | DF2701, DF2702, DF2703 |  |
| Klocwork | 2023.1 | RH.LEAK |  |
| LDRA tool suite | 9.7.1 | 49 D | Partially implemented |
| Parasoft C/C++test | 2022.2 | CERT\_C-FIO42-a | Ensure resources are freed |
| PC-lint Plus | 1.4 | 429 | Partially supported |
| Polyspace Bug Finder | R2023a | CERT C: Rule FIO42-C | Checks for resource leak (rule partially covered) |
| PRQA QA-C | 9.7 | 2701, 2702, 2703 |  |
| PRQA QA-C++ | 4.4 | 2701, 2702, 2703 |  |
| SonarQube C/C++ Plugin | 3.11 | S2095 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not use std::rand() for generating pseudorandom numbers** |
| --- | --- | --- |
| Miscellaneous | [STD-009-CPP] | Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.  The C Standard rand() function, exposed through the C++ standard library through <cstdlib> as std::rand(), makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of std::rand() have a comparatively short cycle, and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs. |

| **Noncompliant Code** |
| --- |
| The following noncompliant code generates an ID with a numeric part produced by calling the rand() function. The IDs produced are predictable and have limited randomness. Further, depending on the value of RAND\_MAX, the resulting value can have modulo bias. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    id += std::to\_string(std::**rand**() % 10000);    // ...  } |

| **Compliant Code** |
| --- |
| The C++ standard library provides mechanisms for fine-grained control over pseudorandom number generation. It breaks random number generation into two parts: one is the algorithm responsible for providing random values (the engine), and the other is responsible for distribution of the random values via a density function (the distribution). The distribution object is not strictly required, but it works to ensure that values are properly distributed within a given range instead of improperly distributed due to bias issues. This compliant solution uses the Mersenne Twister algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the noncompliant code example. |
| #include <random>  #include <string>    **void** f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    std::uniform\_int\_distribution<**int**> distribution(0, 10000);    std::random\_device rd;    std::mt19937 engine(rd());    id += std::to\_string(distribution(engine));    // ...  } |

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

| **Principles(s):** Using the std::rand() function could lead to predictable random numbers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | bad-function (AUTOSAR.26.5.1A) | Fully Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC50 |  |
| Clang | 4.0 (prerelease) | cert-msc50-cpp | Checked by clang-tidy |
| CodeSonar | 7.3p0 | BADFUNC.RANDOM.RAND | Use of rand |
| Compass/ROSE |  |  |  |
| ECLAIR | 1.2 | CC2.MSC30 | Fully implemented |
| Helix QAC | 2023.1 | C++5028 |  |
| Klocwork | 2023.1 | CERT.MSC.STD\_RAND\_CALL |  |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-MSC50-a | Do not use the rand() function for generating pseudorandom numbers |
| Polyspace Bug Finder | R2023a | CERT C++: MSC50-CPP | Checks for use of vulnerable pseudo-random number generator (rule partially covered) |
| PRQA QA-C++ | 4.4 | 5028 | Fully implemented |
| RuleChecker | 22.10 | bad-function (AUTOSAR.26.5.1A) | Fully checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not use an additive operator on an iterator if the result would overflow** |
| --- | --- | --- |
| Containers | [STD-nnn-LLL] | Expressions that have an integral type can be added to or subtracted from a pointer, resulting in a value of the pointer type. If the resulting pointer is not a valid member of the container, or one past the last element of the container, the behavior of the additive operator is undefined. The C++ Standard, [expr.add], paragraph 5 [ISO/IEC 14882-2014], in part, states the following:   * If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.   Because iterators are a generalization of pointers, the same constraints apply to additive operators with random access iterators. Specifically, the C++ Standard, [iterator.requirements.general], paragraph 5, states the following:   * Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called past-the-end values. Values of an iterator i for which the expression \*i is defined are called dereferenceable. The library never assumes that past-the-end values are dereferenceable.   Do not allow an expression of integral type to add to or subtract from a pointer or random access iterator when the resulting value would overflow the bounds of the container. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a random access iterator from a std::vector is used in an additive expression, but the resulting value could be outside the bounds of the container rather than a past-the-end value. |
| #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {  **for** (auto i = c.begin(), e = i + 20; i != e; ++i) {      std::cout << \*i << std::endl;    }  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the programmer's intention was to process up to 20 items in the container. Instead of assuming all containers will have 20 or more elements, the size of the container is used to determine the upper bound on the addition. |
| #include <algorithm>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {  **const** std::vector<**int**>::size\_type maxSize = 20;  **for** (auto i = c.begin(), e = i + std::min(maxSize, c.size()); i != e; ++i) {      // ...    }  } |

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

| **Principles(s):** If adding or subtracting an integer to a pointer results in a reference to an element outside the array or one past the last element of the array object, the behavior is undefined but frequently leads to a buffer overflow or buffer underrun, which can often be exploited to run arbitrary code. Iterators and standard template library containers exhibit the same behavior and caveats as pointers and arrays. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.1 | DF3526, DF3527, DF3528, DF3529, DF3530, DF3531, DF3532, DF3533, DF3534 |  |
| LDRA tool suite | 9.7.1 | 567 S | Enhanced Enforcement |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-CTR55-a | Do not add or subtract a constant with a value greater than one from an iterator |
| Polyspace Bug Finder | R2023a | CERT C++: CTR55-CPP | Checks for possible iterator overflows (rule partially covered). |

### 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.

To automate enforcement of the standards defined in the policy, it is recommended to modify the existing DevOps process by incorporating DevSecOps practices. This involves integrating security into every stage of the development and deployment 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | High | Probable | High | P6 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-009-CPP | Medium | Unlikely | Low | P6 | L2 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 encrypting data while it is stored in a database, file system, or other storage medium. This is done to prevent unauthorized access to the data in case the storage medium is stolen, lost, or hacked. Encryption at rest involves using cryptographic algorithms to convert plain text data into a cipher text, which can only be decrypted with the correct decryption key. The goal of encryption at rest is to protect data confidentiality and integrity even if the storage medium is compromised. |
| Encryption at flight | Encryption at flight refers to the process of encrypting data while it is being transmitted over a network or between systems. This is done to prevent unauthorized access or interception of the data during transmission. It involves using encryption protocols such as TLS/SSL to ensure secure communication between the client and the server. The goal of encryption at flight is to ensure that data remains confidential, integral, and available only to authorized users. |
| Encryption in use | Encryption in use, also known as runtime encryption, is a technique that involves encrypting data while it is being processed or used by an application or system. This is done to protect sensitive data from unauthorized access by applications, system administrators, or other users. Encryption in use is typically used in scenarios where sensitive data needs to be processed or analyzed while maintaining its confidentiality and integrity. This technique involves encrypting the data while it is being processed, and only decrypting it when it is needed, using specialized hardware or software solutions. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the process of verifying the identity of a user or system. It involves confirming that a user is who they claim to be by using methods such as passwords, biometric identification, or multifactor authentication. |
| Authorization | Authorization refers to the process of granting or denying access to resources based on the user's identity and permissions. It involves determining what resources a user is allowed to access and what actions they are permitted to perform on those resources. |
| Accounting | Accounting refers to the process of tracking and auditing user activity to ensure compliance with security policies and regulations. It involves logging and monitoring user activity and generating reports to help detect and investigate security breaches. |

**\***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 |  |  | Emmanuel Nieves |  |
| 1.2 | 4/15/2023 | Final Revision | Emmanuel Nieves |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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