**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 | Input must be validated from any untrusted data sources. With correct input validation a large amount of software vulnerabilities can be eliminated. It is important to be cautious of a lot of external data sources. These sources can include networks interfaces, user controlled files, command line arguments, and environmental variables to name a few external data source to be suspicious of. It is important to ensure that data is as expected and valid. It is important to ensure that data will not cause any injections or overflows that could affect the functioning of the system or reveal any sensitive information. |
| 1. Heed Compiler Warnings | It is important to never ignore any warnings. When compiling your code use the highest warning level that is available to you. Modifying your code to eliminate warnings is vital. Non-standard practices and deprecated code may cause issues and risks later. This is also true even if the code functions fine as is. |
| 1. Architect and Design for Security Policies | During the design process it is imperative to follow security policies from the very start of developing software. This is important because it always keeps safety in mind from beginning to end. Policies must be developed to create rules that limit any potential threats. |
| 1. Keep It Simple | Simple methods are easier to monitor, fix, and detect bugs. Make a choice to use simple solutions and ideas over more complex ones. The more complex it is could open the door to the program using more memory, costing more to maintain, and more time to fix or trace bugs. |
| 1. Default Deny | It is important to always by default deny any access unless credentials are met, and different privilege levels are accepted and confirmed. Conditionals should be set to pass by default until or unless a check fails. The opposite should be done, and everything should be denied until it passes. Whitelisting could be used to grant access to the user, but it is key we do not trust just the user or a device. |
| 1. Adhere to the Principle of Least Privilege | Levels of access should only be given out of necessity, if a user has no need to gain access to it, then it should be admin-level only. If there is no need for it to be public, it should be made private. Even if the admin does not need access to it to complete their job duties, do not make it accessible. |
| 1. Sanitize Data Sent to Other Systems | Copies of data that are unnecessary when sending data to other systems should never be left behind. This is just opening the door for a possibility for the data to get in the wrong hands. Do not just delete sent data to the recycling bin. It is important to ensure the data is destroyed entirely. The use of data masking or data encryption is also important if the data is not destroyed. |
| 1. Practice Defense in Depth | Multiple layers of defense should be utilized so that if one layer ends up failing or turns out to be inadequate there is another layer of defense that will help to prevent a security flaw from becoming a vulnerability. This will also help to limit the consequences if it becomes a successful exploit. It is important to include protection different levels such as the network, physical, data and application levels. |
| 1. Use Effective Quality Assurance Techniques | The use of good quality assurance techniques is effective in helping to identify and eliminate vulnerabilities. This includes testing, code reviews, and utilizing static and dynamic analysis tools. There should also be a plan in place to handle complaints, reports, and fix bugs in a timely manner. |
| 1. Adopt a Secure Coding Standard | It is important to follow a secure coding standard. For example, SEI CERT C++ guidelines. The use of this will help ensure consistent use of security best practices throughout development. Failure to do this can have negative impacts on the program and the organization. |

Resource: <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

### 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]. | Use valid references, pointers, and iterators to reference elements of a basic\_string.  **Rationalization-** This is important because the use of an invalidated reference, iterator, or pointer will result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this example the input is copied into “std::string” with this any semicolons are replaced with spaces. This is an issue because the iterator loc will be invalid after the first call to “insert()”. Any following 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 : ' ');    }  } |
| In this example, after the call to “replace()” data is invalid and also the use in “g()” is an undefined behavior. |
| #include <iostream>  #include <string>    extern void g(const char \*);    void f(std::string &exampleString) {    const char \*data = exampleString.data();    // ...    exampleString.replace(0, 2, "bb");    // ...    g(data);  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| Compliant Solution (std::string::insert()) In this example with each call to “insert()” the iterator loc will be updated. With this the invalidated iterator will not be accessed. The end of the loop is where the updated iterator will be incremented. |
| #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 : ' ');    }  } |
| Compliant Solution (std::replace()) In this example a standard algorithm is used to perform the replacement. Whenever possible, it is preferred to use a generic algorithm over creating your own solution. |
| #include <algorithm>  #include <string>    void f(const std::string &input) {    std::string email{input};    std::replace(email.begin(), email.end(), ';', ' ');  } |
| In this example when the modification from “replace()” has been completed the pointer to “exampleString” internal buffer will be generated. |
| #include <iostream>  #include <string>    extern void g(const char \*);    void f(std::string &exampleString) {    // ...    exampleString.replace(0, 2, "bb");    // ...    g(exampleString.data());  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **3 -** Architect and Design for Security Policies  Ensure string pointers and iterators are validated. If they are created and not used properly, they can lead to undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.3p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2024.4 | DF4746, DF4747, DF4748, DF4749 |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2024a | CERT C++: STR52-CPP | Checks for use of invalid string iterator (rule partially covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CLG] | Use correct integer precisions  **Rationalization-** If an integer’s size is mistaken for its precision this can permit invalid precision arguments to the operations which will then result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this example there is a function that creates 2 raised to the power of the function’s argument. It is important to ensure that an expression is not shifted by a negative number of bits, greater than or equal to the amount of bits existing in the operand. |
| #include <limits.h>    unsigned int pow2(unsigned int exp) {    if (exp >= sizeof(unsigned int) \* CHAR\_BIT) {      /\* Handle error \*/    }    return 1 << exp;  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example “popcount()” function is used. This function counts the bits that are set on unsigned integers. This allows the code to decide the precision integer types whether unsigned or signed. |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  size\_t popcount(uintmax\_t num) {    size\_t precision = 0;    while (num != 0) {      if (num % 2 == 1) {        precision++;      }      num >>= 1;    }    return precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **1 –** ValidateInput Data  Making sure that input data is correct will help prevent any issues compiling integer information. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported: Astrée reports overflows due to insufficient precision |
| CodeSonar | 8.3p0 | LANG.ARITH.BIGSHIFT | Shift Amount Exceeds Bit Width |
| CppCheck Premium | 24.11.0 | premium-cert-int35-c |  |
| Helix QAC | 2024.4 | C0582  C++3115 |  |
| Parasoft C/C++test | 2024.2 | CERT\_C-INT35-a | Use correct integer precisions when checking the right hand operand of the shift operator |
| Polyspace  Bug Finder | R2024a | CERT C: Rule INT35-C | Checks for situations when integer precisions are exceeded(rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CLG] | Do not attempt to modify string literals  **Rationalization-** If string literals are modified it can possibly lead denial-of-service attacks and even worse abnormal program termination. |

| **Noncompliant Code** |
| --- |
| In this example there is an undefined behavior that will be caused due to the attempt to modify the string literal. |
| char \*str  = "string literal";  str[0] = 'S'; |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example the code makes a copy of the string literal. This is done in the space that is allocated to str the character array. This make modifying the string that is stored in str safe. |
| char str[] = "string literal";  str[0] = 'S'; |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **10 -** Use Effective Quality Assurance Techniques  Not following this standard can lead to denial-of-service attacks and abnormal program termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | String-literal-modification  Write-to-string-literal | Fully checked |
| Axivion Bauhaus Suit | 7.2.0 | CertC-STR30 | Fully implemented |
| Compass/ROSE |  |  | Can delete simple violations of this rule |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to “char\*” |
| Helix QAC | 2024.4 | C0556, C0752, C0753, C0754  C++3063, C++3064, C++3605, C++3606,  C++3607 |  |
| Klocwork | 2024.4 | CERT.STR.ARG.CONST\_TO\_NONCONST  CERT.STR.ASSIGN.CONST\_TO\_NONCONST |  |
| LDRA tool suite | 9.7.1 | 157S | Partially Implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-STR30a  CERT\_C-STR30b | A string literal shall not be modified  Do not modify string literals |
| PC-lint Plus | 1.4 | 489, 1776 | Partially supported |
| Polyspace Bug Finder | R2024a | CERT C: rule STR30-C | Checks for writing to const qualified object (rule fully covered) |
| PVS-Studio | 7.35 | V675 |  |
| RuleChecker | 24.04 | String-literal-modfication | Partially checked |
| Splint | 3.1.1 |  |  |
| TrustinSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Exclude user input from format strings  **Rationalization-** If user input is included with format specifiers this can open up the ability for an attacker to view the contents of the stack, crash a vulnerable process, view memory content and much worse. |

| **Noncompliant Code** |
| --- |
| In this example there is an issue of untrusted data which comes from an unauthenticated user. The “incorrect\_password()” function in this code is called when identification and authorization is being completed to output an error message is the password is incorrect or a user is not found. This function will accept the users name as string that is referenced by “user” in the code block. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {    int ret;    /\* User names are restricted to 256 or fewer characters \*/    static const char msg\_format[] = "%s cannot be authenticated.\n";    size\_t len = strlen(user) + sizeof(msg\_format);    char \*msg = (char \*)malloc(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }    fprintf(stderr, msg);    free(msg);  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example the problem is fixed with replacing “fprintf()” with a call to “fput()”. This will result in the msg being outputted directly to stderr. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {    int ret;    /\* User names are restricted to 256 or fewer characters \*/    static const char msg\_format[] = "%s cannot be authenticated.\n";    size\_t len = strlen(user) + sizeof(msg\_format);    char \*msg = (char \*)malloc(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }    fputs(msg, stderr);    free(msg);  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **1 -** ValidateInput Data  This standard focuses on preventing user input from format strings which can have major consequences. The adoption of this standard should help to mitigate this risk. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus | 7.2.0 | CertC-FIO30 | Partially Implemented |
| CodeSonar | 8.3p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Compass/ROSE |  |  |  |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| Cppcheck Premium | 24.11.0 | Premium-cert-fio30-c |  |
| GCC | 4.3.5 |  | Can detect violations of this rule when the -**Wformat-security** flag is used |
| Helix QAC | 2024.4 | DF4916, DF4917, DF4918 |  |
| Klocwork | 2024.4 | SV.FMTSTR.GENERIC  SV.TAINTED.FMTSTR |  |
| LDRA tool suite | 9.7.1 | 86D | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-FIO30-a  CERT\_C-FIO30-b  CERT\_C-FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant  Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable  Never use unfiltered data from an untrusted user as the format parameter |
| PC-lint Plus | 1.4 | 592 | Partially supported: reports non-literal format strings |
| Polyspace Bug Finder | R2024a | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |
| PVS-Studio | 7.35 | V618 |  |
| Splint | 3.1.1 |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory  **Rationalization-** Writing memory that has already been freed may lead to execution of arbitrary code. This code will have permissions of the vulnerable process. On the other end reading freed memory may cause denial-of-service attacks and abnormal program termination. |

| **Noncompliant Code** |
| --- |
| In this example “s” after being deallocated will be dereferenced. This could open possible vulnerabilities is the access results in a write-after-free which can be exploited allowing for arbitrary to be run with permissions linked to the vulnerable process. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example deallocation of dynamically allocated memory is not done until it’s no longer required. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

**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.]  **9** - Use Effective Quality Assurance Techniques  **10** - Adopt a Secure Coding Standard  Any attempts to access free memory may result in unplanned outcomes. The use of effective quality assurance techniques and the adoption of a secure coding standard would assist in recognizing and preventing any potential issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++MEM50 |  |
| Clang | 3.9 | clang-analyzer-  cplusplus.NewDelete  clang-analyzer-  alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.3p0 | ALLOC.UAF | Use after free |
| Compass/ROSE |  |  |  |
| Coverity | V7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Helix QAC | 2024.4 | C++4303, C++4304 |  |
| Klocwork | 2024.4 | 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 | 2024.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024a | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |
| PVS-Studio | 7.35 | V586, V774 |  |
| Splint | 5.0 |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Avoid cycles during initialization of static objects  **Rationalization-** Cycles during initialization of static objects can lead to denial of service or a hacker having the ability to cause a crash. |

| **Noncompliant Code** |
| --- |
| In this example this noncompliant code tries to apply a factorial function utilizing caching. Due to the static local array cache being initialized which involves recursion the functions behavior will be undefined. This is true even as the recursion isn’t infinite. |
| #include <stdexcept>    int fact(int i) noexcept(false) {    if (i < 0) {      // Negative factorials are undefined.      throw std::domain\_error("i must be >= 0");    }      static const int cache[] = {      fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),      fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),      fact(12), fact(13), fact(14), fact(15), fact(16)    };      if (i < (sizeof(cache) / sizeof(int))) {      return cache[i];    }      return i > 0 ? i \* fact(i - 1) : 1;  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example the initialization of the static local array cache is avoided. Instead, zero-initialization is used to determine if each member in the array has an assigned value yet. If a value has not been assigned it will recursively compute its value. |
| #include <stdexcept>    int fact(int i) noexcept(false) {     if (i < 0) {      // Negative factorials are undefined.      throw std::domain\_error("i must be >= 0");    }      // Use the lazy-initialized cache.    static int cache[17];    if (i < (sizeof(cache) / sizeof(int))) {      if (0 == cache[i]) {        cache[i] = i > 0 ? i \* fact(i - 1) : 1;      }      return cache[i];    }      return i > 0 ? i \* fact(i - 1) : 1;  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **10 -** Adopt a Secure Coding Standard  Cycles during initialization of static objects can lead to vulnerably and should be avoided adopting a secure coding standard would help to recognize and prevent coding in such a way. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.3p0 | LANG.STRUCT.INIT.CYCLE  LANG.STRUCT.INIT.UNORDERED | Initialize Cycle  Unordered Initialization |
| Helix QAC | 2024.4 | C++1552, C++1554, C++1704 |  |
| LDRA tool suite | 9.7.1 | 6 D | Enhance Enforcement |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL56-a | Avoid initialization order problems across translation units by replacing non-local static objects with local static objects |
| Polyspace Bug Finder | R2024a | CERT C++: DCL56-CPP | Checks for:   * Recursive initialization of static variables * Undetermined initialization order of global variables   Rule fully covered. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions  **Rationalization-** Handling all exceptions is critical to developing secure and reliable code. |

| **Noncompliant Code** |
| --- |
| In this example both “f()” or “main()” catch any exceptions that are thrown by “throwing\_func()”. With this no matching handler is found for the exception thrown resulting in “std::terminate()” to be called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example all exceptions are handled by the main entry point. This ensures the stack is unwound up to the main function. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **8** - Practice Defense in Depth  **9** - Use Effective Quality Assurance Techniques  Handling all exceptions will prevent the program from crashing and unhandled exceptions can lead to vulnerabilities.  The practice of defense in depth and the use of effective quality assurance techniques will help prevent this. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 8.3p0 | LANG.STRUCT.UCTCH | Unreachable catch |
| Helix QAC | 2024.4 | C++4035, C++4036, C++4037 |  |
| Klocwork | 2024.4 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially checked |
| Parasoft  C/C++test | 2024.2 | CERT\_CPP\_ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming (OOP) | [STD-008-CPP] | Do not use pointer-to-member operators to access nonexistent members  **Rationalization-** Using a pointer-to-member expression with the dynamic type of the first operand doesn’t include the member to which the second operand refers, including the utilization of a null pointer-to-member value as the second operand can result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In the example “D::g” is used to obtain a pointer to member object. Then it is upcast to a “B::\*”. The pointer-to-member is well defined when an object is called on whose dynamic type is “D”. The problem here is the underlying object is “B” this will result in undefined behavior. |
| struct B {    virtual ~B() = default;  };    struct D : B {    virtual ~D() = default;    virtual void g() { /\* ... \*/ }  };    void f() {    B \*b = new B;      // ...      void (B::\*gptr)() = static\_cast<void(B::\*)()>(&D::g);    (b->\*gptr)();    delete b;  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example the upcast has been removed which renders the initial code not well formed and accentuating the original problem that B::g() doesn’t exist. This solution supposes the programmer’s intent was to utilize the proper dynamic type regarding the underlying object. |
| struct B {    virtual ~B() = default;  };    struct D : B {    virtual ~D() = default;    virtual void g() { /\* ... \*/ }  };    void f() {    B \*b = new D; // Corrected the dynamic object type.      // ...    void (D::\*gptr)() = &D::g; // Moved static\_cast to the next line.    (static\_cast<D \*>(b)->\*gptr)();    delete b;  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **4 -** Keep It Simple    Keeping it simple and straight forward is important to the standard as a pointer to member expression that is coded incorrectly can cause issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference  invalid\_function\_pointer |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP55 |  |
| CodeSonar | 8.3p0 | LANG.MEM.UVAR | Uninitialized Variable |
| Helix QAC | 2024.4 | DF2810, DF2811, DF2812, DF2813, DF2814 |  |
| Klocwork | 2024.4 | CERT>OOP.PTR\_MEMBER.NO\_MEMBER |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-OOP55-a | A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024a | CERT C++: OOP55-CPP | Checks for pointers to member accessing non existent class members (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | [STD-009-CPP] | Detect errors when converting a string to a number  **Rationalization-** Although it can be uncommon for a violation of this standard to result in a security vulnerability it is more probable to occur in security sensitive code which can easily result in misinterpreted or lost data. |

| **Noncompliant Code** |
| --- |
| In this example there are multiple numeric values that are converted from the standard input stream. When the text that is received from the standard input stream can’t be changed into a numeric value that is represented by an “int” the value that is stored in the variables ”i” and “j” could be unexpected. |
| #include <iostream>    void f() {    int i, j;    std::cin >> i >> j;    // ...  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example exceptions are enabled allowing any conversion failure to result in an exception to be thrown. However, this approach is not able to differentiate between which values are the valid or invalid. This results in the program assuming all values are invalid. In this compliant code both “badbit” and “failbit” flags have been set to guarantee that conversion errors and loss of integrity that happen with the stream will be treated as exceptions. |
| #include <iostream>    void f() {    int i, j;      std::cin.exceptions(std::istream::failbit | std::istream::badbit);    try {      std::cin >> i >> j;      // ...    } catch (std::istream::failure &E) {      // Handle error    }  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **3 -** Architect and Design for Security Policies  This standard is most probable when dealing with security sensitive code and this principle would help to better recognize the most vulnerable code. |
| --- |

**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++-ERR62 |  |
| Clang | 3.9 | cert-err34-c | Checked by clang-tidy; only identifies use of unsafe C Standard Library functions corresponding to ERR34-C |
| CodeSonar | 8.3p0 | BADFUNC.ATOF  BADFUNC.ATOI  BADFUNC.ATOL  BADFUNC.ATOLL | Use of atof  Use of atoi  Use of atol  Use of atoll |
| Helix QAC | 2024.4 | C++3161 |  |
| Klocwork | 2024.4 | CERT.ERR.CONV.STR\_TO\_NUM |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR62-a | The ‘atof’, ‘atoi’, ‘atol’, and ‘atoll’ functions from the ‘stdlib.h’ or ‘cstdlib’ library should not be used |
| Polyspace Bug Finder | R2024a | CERT C++: ERR62-CPP | Checks for unvalidated string-to-number conversion (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | [STD-010-CPP] | Close files when they are no longer needed  **Rationalization-** The failure to close files could permit attackers to deplete system resources which can increase the risk that data that has been written into the in-memory file buffers not to be flushed on the occasion of an abnormal program termination. |

| **Noncompliant Code** |
| --- |
| In this example an object file of “std::fstream” is constructed. In this the constructor for “std::fstream” will call “std::basic\_filebuf<T>open()”, and the “std::terminate\_handler” called by “std::terminate()” is “std::abort()” which doesn’t call any destructors. With this the underlying “std::basicfilebuf<T>” object is not correctly 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();  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

| **Compliant Code** |
| --- |
| In this example the call to “std::fstream::close()” is made before “std::terminate()”. This assures that file resources are appropriately 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.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

Resource: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

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

| **Principles(s):**  **3** - Architect and Design for Security Policies  **10** - Adopt a Secure Coding Standard  When files are not closed, they can result in the corruption of data or resources leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.3p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2024.4 | DF4786, DF4787, DF4788 |  |
| Klocwork | 2024.4 | RH.LEAK |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024a | CERT C++: FIO51-CPP | Checks for resource leak (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.

In this security policy a critical component of ensuring and enforcing compliance regarding the standards outlined is automation. Green Pace can attain a scalable, robust, and a continuous security stance by inserting different security processes and checks into the DevSecOps. To achieve this, automation will be integrated at different times within the development lifecycle, while leveraging the DevSecOps and Defense-in-Depth diagrams. First, the technical security debt should be addressed. In the planning phase, potential security vulnerabilities will be detected early with the use automation tools. Static analysis tools such as SonarQube or Cppcheck will help identify the potential for memory leaks, SQL injection vulnerabilities and improper exception handling. Automated tools will help to assess any potential attack vectors due the systems architecture one such too to be used is the Microsoft Threat Modeling Tool.

A secure architecture must be built, and this starts with architecture validation where automation tools will be leveraged to validate compliance utilizing predefined secure coding principles and will command secure design patterns. This will ensure devotion to design in depth and the principle of least privilege. To comply with Green Pace’s security standards automation of validations will be within CI/CD pipelines. Automated dependency checkers such as OWASP Dependency-Check will be utilized while in the build phase to make sure frameworks and libraries that are being utilized are free from any well-known vulnerabilities. The use of strict compiler warnings will force developers to address potential coding issues before they move on. Test will be a key part of the process The use of automated static application security testing (SAST) tools, dynamic application security testing (DAST) will be leveraged. This will help identify violations of secure coding standards, unhandled exceptions, and improper synchronization to name a few.

During pre-production integration testing and software signing will be utilized. All components, including third party libraries will be validated utilizing automated integration testing tools ensure everything works securely with each other. Different tools like GPG will help to verify the integrity of the different build prior to deployment to assist in preventing tampering. During production run time self-protection (RASP) will be implemented the application will continuously be monitored by automation in real time trying to identify potential exploits. The exploits can include improper exception handling and data races. Network monitoring will be key and tools like Nagios and Splunk can be leveraged to automatically give alerts when suspicious activity happens in the network infrastructure or the application. Another area automation will be a key factor is with incident response. Security Orchestration, Automation, and Response (SOAR) platforms will be used to simplify the responses to breaches or identify vulnerabilities. This will help to make sure any incidents are addressed promptly. Finally, to continually maintain and adapt the use of tools such as ThreadSanitizer and Valgrind will be utilized to do regular automated scans. This will help to make sure that there is compliance with secure coding practices. Configuration management tools will be utilized to help create a continuous secure environment. The ability to embed automation into every different phase of the DevSecOps pipeline will ensure Green Pace is consistently following secure coding standards. With this, vulnerabilities will be reduced, and compliance will be met with the companies’ defense-in-depth strategy.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Probable | High | Medium (**P8**) | 2 (**L2**) |
| STD-002-CLG | Low | Unlikely | Medium | Low (**P2**) | 3 (**L3**) |
| STD-003-CLG | Low | Likely | Low | Medium (**P9**) | 2 (**L2**) |
| STD-004-CLG | High | Likely | Medium | High (**P10**) | 1 (**L1**) |
| STD-005-CPP | High | Likely | Medium | High (**P18**) | 1 (**L1**) |
| STD-006-CPP | Low | Unlikely | Medium | Low (**P2**) | 3 (**L3**) |
| STD-007-CPP | Low | Probable | Medium | Low (**P4**) | 3 (**L3**) |
| STD-008-CPP | High | Probable | High | Medium (**P6**) | 2 (**L2**) |
| STD-009-CPP | Medium | Unlikely | Medium | Low (**P4**) | 3 (**L3**) |
| STD-010-CPP | Medium | Unlikely | Medium | Low (**P4**) | 3 (**L3**) |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is focused on the application of encryption strategies for inactive and stored data. This includes information in hard drives, databases, and any other long term storage solutions. The use of encryption algorithms can be utilized to mitigate any unauthorized access to malicious actors. Any data that will be kept for an indeterminate or substantial amount of time needs to follow the encryption at rest policy. This means all locally collected files, backup information, or database contents must be encrypted utilizing the AES-256 encryption standard. This will help to prevent any unauthorized access in the event that the data at rest had been compromised. |
| Encryption in flight | Encryption at flight is focused on the utilization of encryption strategies for any data that is going to be transmitted over a network for a particular location to another. This can include any public communication channels or requests and or responses between a server and a client. Data in flight can more insecure that data at rest because it is exposed to the internet which make it a possibility for it to be modified and or intercepted. All information that that is going to be moving internally or externally source and destination must follow the encryption in flight policy. This means we must use standards such as AES-256 and also relevant protocols like SSL/TLS to encrypt and protect sensitive information. We must apply these protocols to APIs, email, web applications among other services. |
| Encryption in use | Encryption in use is focused on the utilization of different encryption strategies to protect data this actively being used. This pertains to data that is being processed, changes, or read. Data at this point is most vulnerable. To prevent malicious actor’s encryption strategies must be utilized to safeguard and promote resiliency. Encryption in use policy must be applied to all information that is going to accessed by any users or systems. Principle of least privilege must be enforced to help protect the vulnerable data. This will help mistakes not have such an impact or corrupt the data. The use of the correct authentication standards with the addition of robust encryption protocols to decrypt data while it is needed and then later to encrypt it is imperative to assist in protect the data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a concept that focuses on the validation of whether the user is who they say they are. Protect systems should only be accessed be verified applications or users and authentication ensures this. The use of Multi-Factor Authentication can be utilized to confirm any access request. Other best practices can be followed such as the use of strong passwords can reduce the risk of malicious actors having the ability to be authenticated like a trusted user. The authentication policy must be applied by taking the passwords that will be used to log into the system and securely hashing them. This should also extend to potentially vulnerable or protected resources helping to prevent any unauthorized access due to any overlooked weak spots. To also help prevent the risk of attacks and minimize the surface area, time windows should be enforced when inactivity is detected on user logins. |
| Authorization | Authorization sets the different levels of access the users have to the system. This helps to prevent access user from have access to the entire system they may be authenticated, but that does not mean they will have the highest level of access. The principle of least privilege should be enforced or that users should only be granted the least amount of access that is required. We can use the principle of Default Deny to better help protect actions and resources that may have not been clearly accounted for. The use of role-based permissions must be applied to all users that engage with sensitive resources and systems. The ability to add new users must be regulated due to the fact the pose a risk of unwarranted outcomes if mistreated. |
| Accounting | Accounting revolves around the logging and keeping track of user activity and or the processes within the system. Some examples of this are logging changes made to database items and keeping track of the users’ activity when a protect resources has been accessed. These examples and forms of tracking and logging can be gathered into reports to promote efficient incident tracking. Accounting must be applied to all interactions and transactions with any resources or sensitive data within the system. Unusual activity or activities will be easier to catch if we are prudent about the logs and metrics that we are tracking. With this any incidents that arise can be adequately accessed to determine at which point the failure occurred. |

**\***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 |  |
| 2.0 | 01/23/2024 | Milestone 1 Update | Dylan Coulter |  |
| 3.0 | [Insert text.] | Project 1 Update | Dylan Coulter |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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