**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 | Validation of Input data is a strong defense against buffer overflow, malformed string inputs, or other malformed data types. In fact, Secure Coding in C and C++ (Second Edition) says the best protection against bufferoverflow is input validation. This is because malformed input has the ability to crash a system which can lead to buffer overflow exploits. To ensure valid input Data in C/C++ standards, Refer to SEI CERT C++ Rule 07. Input Output (FIO) which includes compliant solutions with fputs() or fprints() for example. |
| 1. Heed Compiler Warnings | In assessment of buffer overflow within a project, always heed compiler warnings on build. Static analysis via Clang-Tidy and tools like Cppcheck are valid tools that can help pinpoint build errors, but as a first line of defense always enable and heed compiler warnings. Visual Studio C++ compiler can help identify uninitialized variables, forgetting a function’s return value, printf and scanf format string mismatches, and functions being used before declaration (among others). In addition, enabling basic runtime checks (/RTCs) within the C/C++ options will allow compiler generated runtime checks for buffer overflows with some minor overhead. |
| 1. Architect and Design for Security Policies | Incorporating Security into the Software Development Life Cycle from the onset of the project is a valid strategy. This all starts with user education, principle of least privelege, and network segregation. Secure Software architecture design might consider dividing the system into distinct, intercommunicating subsystems, each with an appropriate set of priveleges. This would allow unpriveleged process to communicate with another process that has retained elevated priveleges, in order to perform security-critical operations. Ensure the system has data sanitation, input validation, and deny by default appropriately in place. Make the algorithms public to allow comment and dissection for security purposes. Research NIST, OWASP, CERT for recent vulnerabilities. Read academic journals for recent research in the field. |
| 1. Keep It Simple | This principle equates to clarity, cohesion, tidiness, and overall efficiency of your implementations. To reduce complexity, give variables descriptive names, give methods purposeful names, use comments within code for explanations. Ensure each Class has a single responsibility, and remove any redundant instances, methods, processes, global states, or behaviors that are not in use. If efficiency is maintained after these rules, then KISS has been applied. |
| 1. Default Deny | Deny permissions by default, only granting access when the protection scheme deems that permission is authorized. This could be in regards to a firewall policy, blocking all inbound and outbound traffic that has not been permitted expressly by that policy. Deny by default could also be in regards to SQL Server – which is deny by default, allowing only added users with EXECUTE permissions. Deny by default could also be in regards to the port, which could be changed to a nonstandard port, as well as Fail2Ban which denies all attempts at entering incorrect passwords and temp-bans the user, and lastly deny by default can be the file permissions themselves, which on unix based systems have a CHMOD, CHOWN with directories, and files having either root or user permission to read or write. |
| 1. Adhere to the Principle of Least Privilege | A User account should have least privelege – for example, limit an average user from any capability beyond adding a record to a database, and do not allow rampant Root user accounts. MYSQL employs principle of least privelege when several user accounts are each designated toward their own unique DB task and priveleges. This minimizes attack vector surfaces and thereby limits malware propogation and users with destructive intentions. In C code, the const keyword states to the compiler that no change of the class members will be done during the execution of the f() function, implicitly making **this** parameter within f() constant, thereby forcing read-only priveleges when overriding the function. This principle could also mean – do not pass more parameters to a method than needed. |
| 1. Sanitize Data Sent to Other Systems | If a program allowed only valid inputs, and the program logic correctly anticipates and handles every possible combination of valid inputs, then vulnerabilities can be eliminated. The problem is avoiding strings which allow commands to be run, as well as data that is too large to input, or data which causes buffer overflow. Two approaches would be black-listing and white-listing. Utilizing a deny by default approach would be to whitelist, which is a list in which only valid inputs are accepted, allowing fail-safe behavior on unexpected inputs. This is preferred method as we cannot always predict every invalid value to exclude via a black-list. Data sanitation could also include an parameterizing an SQL query to avoid SQL injection. |
| 1. Practice Defense in Depth | Defense in Depth can extend across departments or teams by providing multiple independent layer of defense. For example, IT maintenance people might implement firewalls, intrusion detection, antivirus/antimalware, VPN or even Virtual Machine. Server maintenance might implement port blocking and tighter security configuration settings and logging. As a programmatic level this could involve protecting against XML and SQL injection attacks or validating all user input, and implementing secure encryption protocols. This will prevent costly security vulnerabilities later in the project’s life. Use reliable, well-tested community libraries and resources. Implement hardened code from crytpo libraries (tested, and assured) in place of rolling your own solution. |
| 1. Use Effective Quality Assurance Techniques | Utilize static analysis tools such as Tidy-Clang which is a linter built in to recent versions of Visual Studio C++ compiler, that allows diagnosing and fixing programming errors. Use cppcheck as a second opinion on compiled code, checking for undefined behaviors such as dead pointers and out of bounds checking. For further checking, integrate **Coverity Scan** into your continuous integration workflow (Travis-CI -> Build -> Check -> Github). Utilize techniques such as Exceptions when some situation occurrs that prevents the function from fulfilling its work. However, if the function can recover and handle its problem, it should not throw an exception. Utilize assertions to test for conditions during development that should never be true, assuming all correct code. Failing assertions usually indicate deprecated code that needs to be fixed before build. Lastly, as with all QA, ensure the project adheres to the user requirements every step of the way, possibly via SCRUM or AGILE approaches. |
| 1. Adopt a Secure Coding Standard | As apart of both Quality Assurance and SDLC (software development lifecycle), a secure coding standard should be adopted for the project at hand. In C++, the secure standard to adhere to is SEI CERT C++ Coding Standard which contains compliant examples of declarations / initializations, expressions, integers, characters / strings, memory management, input/output, exceptions / errorhandling, object oriented approaches, and concurrency (among others). These secure principles can act as guidelines and can be incorporated into the continuous integration process. |

### 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 Value** | [STD-001-CPP] | Proper ranging of integer data types must be adhered to so as to prevent wrapping (integer overflow) between signed and unsigned int conversions at their MAX and MIN. As an example, INT30-C requires that integer values not be allowed to wrap if they are used in any pointer arithmetic, including array indexing, as a length or size of an object (for example, the size of a variable-length array), as the bound of access to an array (for example, a loop counter), in function arguments of type size\_t or rsize\_t (for example, as an argument to a memory allocation function), or in security-critical code. Truncation Error results from conversion of a value in an unsigned integer type to a narrower width, as data is lost if the value cannot be represented in the new type. |

| **Noncompliant Code** |
| --- |
| Truncation can occur when a value is too small to represent the result, and conversions can result in values out of range in the resulting type. |
| 1 unsigned long int ul = ULONG\_MAX;  2 signed char sc;  3 sc = (signed char)ul; /\* cast eliminates warning \*/ |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. |
| 1 unsigned long int ul = ULONG\_MAX;  2 signed char sc;  3 if (ul <= SCHAR\_MAX) {  4 sc = (signed char)ul; /\* use cast to eliminate warning \*/  5 }  6 else {  7 /\* handle error condition \*/  8 } |

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

| **Principles(s):** 1) Validate Input Data to catch these mistakes from the onset, and 2) Use Effective Quality Assurance Techniques to mitigate these problems via analysis tools. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |
| CodeSonar | 6.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Parasoft C/C++test | 2021.1 | CERT\_C-INT30-a  CERT\_C-INT30-b  CERT\_C-INT30-c | Avoid integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| Polyspace Bug Finder | R2021a | CERT C: Rule INT30-C | Checks for:  Unsigned integer overflow  Unsigned integer constant overflow  Rule partially covered. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-002-CPP] | As a rule of thumb when choosing a data type, Unsigned integer values should represent values that can’t become negative, signed integer values should be used for values that can become negative. In general, use the smallest signed or unsigned type that can fully represent the range of possible values for the given variable, as this conserves memory. For example:  short total = strlen(argv[1])+ 1; allows negative sizes and may not have adequate range, whereas:  size\_t total = strlen(argv[1])+ 1; utilizes size\_t which is more precise.  Relevant rule: INT32-C. Ensure that operations on signed integers do not result in overflow |

| **Noncompliant Code** |
| --- |
| This code fails to consider that the unsigned integer value will wrap around. (infinite loop) |
| 1 char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  2 size\_t cnt = /\* initialize \*/;  3  4 for (unsigned int i = cnt-2; i >= 0; i--) {  5 a[i] += a[i+1];  6 } |

| **Compliant Code** |
| --- |
| As size\_t is an unsigned type, this behavior is well defined by the standard to be modulo. |
| 1 char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  2 size\_t cnt = /\* initialize \*/;  3  4 for (size\_t i = cnt-2; i != SIZE\_MAX; i--) {  5 a[i] += a[i+1];  6 } |

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

| **Principles(s):** 1) Validate Input Data, 2) Heed compiler warnings, 9) Effective Quality assurance techniques. By properly validating we can prevent out of bound memory errors, by heeding compiler warnings we can catch before production, and by using QA techniques we can usee analsysis in the pipeline to catch this kind of error. 3) Architect for design specifically education on this issue would prevent it and ensure the team are using types correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Helix QAC | 2021.2 | C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863  C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863 | Implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| TrustInSoft Analyzer | 1.38 | Signed\_overflow | Exhaustively verified |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Incorrect string sizes and neglected buffer boundaries can lead to buffer overflows and runtime errors. Never copy data from some unbounded source such as stdin to a fixed-length array. For string literals,  STR30-C. Do not attempt to modify string literals  STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator”  In addition, keep in mind UTF-decoders can be a security hole if an attacker sends the decoder an octect sequence not permitted by UTF-8 syntax: ““MSC10-C. Character encoding—UTF-8-related issues”  A strong and secure model for string handling is “callee allocates, callee frees” from C++ standard class template std:basic\_string, in which callee allocates and frees storage. C++ std:basic\_string template does not require null terminator, preventing costly mistakes. This is the most secure model and if using C++ should be adhered to. Never use gets(). |

| **Noncompliant Code** |
| --- |
| C++: If a user inputs more than 11 characters, it will result in an out-of-bounds write |
| 1 #include <iostream>  2  3 int main(void) {  4 char buf[12];  5  6 std::cin >> buf;  7 std::cout << "echo: " << buf << '\n';  8 } |

| **Compliant Code** |
| --- |
| Eliminates the overflow in the previous example by setting the field width member to the size of  the character array buf. |
| 1 #include <iostream>  2  3 int main(void) {  4 char buf[12];  5  6 std::cin.width(12);  7 std::cin >> buf;  8 std::cout << "echo: " << buf << '\n';  9 }06 string str;  07 string::iterator i;  08 for (i = str.begin(); i != str.end(); ++i) {  09 cout << \*i  10 } |

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

| **Principles(s):** 1) Validate input with appropriate string function, 2) Heed compiler warnings of overflow, 3) Sanitize data to prevent string attacks, and 4) QA Techniques to catch these errors on build in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Attack vectors for strings include command-line arguments, environmental variables, console input, text files, and network connections. These are all vectors in which an attacker can launch strategic SQL injection-based attacks in order to cause overflow. According to Secure Coding in C and C++, “String concatenation is the primary point of entry for script injection.” Test the size / data type of input and enforce appropriate limits, rejecting any entries that contain binary data, escape sequences, and comment characters. Review all code that calls EXECUTE, EXEC, or sp\_executesql. Always use Parameterized Queries in SQL.  C++ operator >> extracts characters and stores them in successive elements of the array pointed to by str, ending extraction when the next element is either a valid white space or a null character or EOF is reached.  By limiting the extraction operation to a certain number of characters, buffer overflow can be prevented: do this by setting the field width (ios\_base::width or setw()) to a value greater than 0.  Related SEI Rule:  STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| Unfiltered code is vulnerable to SQL injection through user input |
| SqlDataAdapter myCommand =  new SqlDataAdapter("LoginStoredProcedure '" +  Login.Text + "'", conn); |

| **Compliant Code** |
| --- |
| Using the Parameters collection with Dynamic SQL: |
| SqlDataAdapter myCommand = new SqlDataAdapter(  "SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id", conn);  SQLParameter parm = myCommand.SelectCommand.Parameters.Add("@au\_id",  SqlDbType.VarChar, 11);  Parm.Value = Login.Text; |

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

| **Principles(s):** 1) Input validation, ensure to use query parameterization, 2) Architect and design for security policies meaning code with knowledge of possible SQL injection attacks in mind, 3) Practice defense in depth, using a layered defensive strategy to prevent such attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | A stack buffer overflow can be caused by stack smashing, wherein a potential security bug is exploited and the stack buffer is filled with data supplied from an untrusted user. The user corrupts the stack in a way that allows code injection into the program at runtime, allowing the user to take control. In addition, dereferencing of null or invalid pointers can lead to undefined behavior and segmentation faults. In addition, pointer subterfuge exploits can overwrite function pointers to transfer control to attacker-supplied shellcode. Mitigations include eliminating buffer overflow conditions and storing encrypted pointers. Some general rules are: do not write to an area that’s been freed, and do not free a memory chunk multiple times (no double free). These rules are from the CERT Standard for C++ security guidelines:  MEM50-CPP. Do not access freed memory  EXP53-CPP. Do not read uninitialized memory  MEM52-CPP. Detect and handle memory allocation errors  MEM31-C. Free dynamically allocated memory when no longer needed  The last rule implies that you can catch a **std::bad\_alloc** exception if sufficient memory cannot be allocated.  In addition, follow the CERT C++ Coding Standard [SEI 2012b], “MEM08-CPP. Use new and delete rather than raw memory allocation and deallocation” to avoid subtle memory errors cropping up. |

| **Noncompliant Code** |
| --- |
| Vulnerable code shows that s is dereferenced after it has been deallocated. If this were to result in “write after free”, the vulnerability can be exploited to run arbitrary code (using the vulnerable process permissions). |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| The solution is to wait until the memory is no longer required to deallocate. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** Architect for security involves trainig the team in writing secure memory allocation routines, Adopt a Secure Coding standard to ensure this is consistent across the project, Quality Assurance Techniques to test for these types of vulnerabilities on each scan in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Polyspace Bug Finder | R2021a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| Helix QAC | 2021.2 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | As a rule of thumb, use assert statements to handle errors that test for conditions that should never be true, if all code is correct. As this situation does not represent a condition that the program has to recover from at run-time, exceptions should not be used. As these types of errors indicate code needing to be fixed, there is little point to handle them with exceptions.  The purpose of an assert statement is to be there, for a programmer to see them. They are especially useful at catching deprecated code errors. When assertions are used, a bad input causes immediate halt.  Especially useful is testing for error conditions at some point in the code that should have already been handled. However, do not use assertions as a substitute for error-handling code. For more information, MSC11-C. Incorporate diagnostic tests using assertions. |

| **Noncompliant Code** |
| --- |
| By relying only on the assertion to handle the error condition, any error code returned by this myGraphRoutine will be unhandled come final release. |
| myErr = myGraphRoutine(a, b);  /\* No Code to handle errors \*/  ASSERT(!myErr); // Don't do this!  \_ASSERT(!myErr); // Don't do this, either! |

| **Compliant Code** |
| --- |
| In this example, a graphic routine returns an error code or zero for success. If the error-handling code works properly, the error should be handled and myErr reset to zero before the assertion is reached. If myErr has another value, the assertion fails, the program halts, and the Assertion Failed Dialog Box appears. |
| myErr = myGraphRoutine(a, b);  /\* Code to handle errors and  reset myErr if successful \*/  ASSERT(!myErr); -- MFC version  \_ASSERT(!myErr); -- CRT version |

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

| **Principles(s):** Architect for security involves trainig the team in writing secure memory allocation routines, Adopt a Secure Coding standard to ensure this is consistent across the project, Quality Assurance Techniques to test for these types of vulnerabilities on each scan in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2021.1 | CERT\_C-MSC11-a | CERT\_C-MSC11-a |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Rule of thumb (when to throw an exception): An exception should be thrown when a situation has happened that prevents the called function from fulfilling its work. Note, that if the called function can still recover from the problem and provide the user with some services promised, then it has handled its own problem and does not need to throw an exception.  In regard to APIs, if some API encounters an error it cannot recover from, it should throw the exception back to the client code calling the Api. This allows the client code to catch said exception, and allows the client to decide if it wants to retry the conneciton after waiting some specified time, or else try some other network endpoint.  Use exceptions to check for errors that might occur such as input validation on parameters of public functions.  Always throw an exception by value and catch by reference or const reference if possible — this ensures the compiler takes care of memory management for the exception object. Lastly, do not catch what you cannot handle.  Relevant SEI Cert C++ Standards:  ERR61-CPP. Catch exceptions by lvalue reference  ERR51-CPP. Handle all exceptions  ERR57-CPP. Do not leak resources when handling exceptions  ERR56-CPP. Guarantee exception safety:  “The strong exception safety guarantee is a property of an operation such that, in addition to satisfying the basic exception safety guarantee, if the operation terminates by raising an exception, it has no observable effects on program state.” |

| **Noncompliant Code** |
| --- |
| As per SEI Cert C++ Coding Standard:  This example shows a *flawed copy assignment operator*.  The **implicit invariants** of the class: 1) the array member is a valid (possibly null) pointer and 2) the nElems member stores the number of elements in the array pointed to by array.  The function *deallocates array* and assigns the element counter, nElems, before *allocating a new block of memory* for the copy. As a result, if the new expression throws an exception, the function will have *modified the state of both member variables* in a way that **violates the implicit invariants** of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior. |
| #include <cstring>    class IntArray {  int \*array;  std::size\_t nElems;  public:  // ...    ~IntArray() {  delete[] array;  }      IntArray(const IntArray& that); // nontrivial copy constructor  IntArray& operator=(const IntArray &rhs) {  if (this != &rhs) {  delete[] array;  array = nullptr;  nElems = rhs.nElems;  if (nElems) {  array = new int[nElems];  std::memcpy(array, rhs.array, nElems \* sizeof(\*array));  }  }  return \*this;  }    // ...  }; |

| **Compliant Code** |
| --- |
| The copy assignment operator provides the *strong exception safety guarantee*. The function allocates new storage for the copy *before* changing the state of the object. Only after the allocation succeeds does the function proceed to *change the state* of the object. In addition, by copying the array to the newly allocated storage *before deallocating the existing array*, the function avoids the test for self-assignment, which improves the performance of the code in the common case [Sutter 2004]. |
| #include <cstring>    class IntArray {  int \*array;  std::size\_t nElems;  public:  // ...    ~IntArray() {  delete[] array;  }    IntArray(const IntArray& that); // nontrivial copy constructor    IntArray& operator=(const IntArray &rhs) {  int \*tmp = nullptr;  if (rhs.nElems) {  tmp = new int[rhs.nElems];  std::memcpy(tmp, rhs.array, rhs.nElems \* sizeof(\*array));  }  delete[] array;  array = tmp;  nElems = rhs.nElems;  return \*this;  }    // ...  }; |

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

| **Principles(s):** Architect for security involves training the team is using assertions in development codebase, Adopt a Secure Coding standard to ensure this is consistent across the project, Quality Assurance Techniques to run such assertions on each scan in the CI/CD pipeline, thereby regression testing securely. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++4035, C++4036, C++4037 |  |
| Parasoft C/C++test | 2021.1 | 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 | R2021a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| API Vulnerability Assessment | [STD-008-CPP] | If using any APIs outside of the C++ standard library (or sometimes within it), remain up to date on possible explots and vulnerabilities within those libraries, if they exist. If the library is open-sourced, this allows a large number of coders to review and point out vulnerabilities or create exploits against such code. As per Wired, “fuzzing is the usually automated process of finding hackable software bugs by randomly feeding different permutations of data into a target program until one of those permutations reveals a vulnerability.” One such article on Security Boulevard describes uncovering vulnerabilities in Open Source libraries. A simple set of standards for evaluating a library is as follows, taking FreeImage as the example:   1. Easy to build and run target (make or otherwise) --> Yes, uses a makefile 2. Target language --> C / C++ 3. In which ways does this target accept input? --> parses image files and can also parse image data from a memory buffer 4. Which input format is expected for the target? --> Very many, including PSD parser which might be susceptible to fuzzing 5. Which projects are using this library? --> Distros: Debian, Ubuntu. Rdepend projects: ArrayFire, CEGui, Gazebo Simulator, OpenCASCADE, FreeCAD, and OGRE. 6. Who has fuzzed this target before --> None public 7. What quality assurance mechanisms are in place for the target? --> A Test suite exists, but they have no tests for the PSD parser. “The project is hosted in a CVS repository on SourceForge, last released on July 31, 2018, and development appears to either have been halted or is very inactive.”   https://securityboulevard.com/2020/01/uncovering-vulnerabilities-in-open-source-libraries/ |

| **Noncompliant Code** |
| --- |
| Vulnerabilities assessed in image library:  CVE-2019-13499: A heap buffer overflow caused by a negative-size memcpy/memset in psdParser::UnpackRLE in the psdThumbnail component of FreeImage 3.18.0 allows an attacker to cause a denial of service via a crafted PSD file.  CVE-2019-13500: A heap buffer overflow in psdThumbnail::Read in the psdThumbnail component of FreeImage 3.18.0 allows an attacker to cause a denial of service or execute arbitrary code via a crafted PSD file.  CVE-2019-13501: A heap buffer overflow in psdParser::ReadImageLine in the psdParser component of FreeImage 3.18.0 allows an attacker to cause a denial of service or execute arbitrary code via a crafted PSD file. |
| @@ -1320,24 +1330,26 @@ void psdParser::UnpackRLE(BYTE\* line, const BYTE\* rle\_line, BYTE\* line\_end, unsi    // (len + 1) bytes of data are copied  ++len;  + // ensure we don't write beyound eol  + size = line + len > line\_end ? line\_end - line : len;    - // assert we don't write beyound eol  - memcpy(line, rle\_line, line + len > line\_end ? line\_end - line : len);  - line += len;  - rle\_line += len;  - srcSize -= len;  + memcpy(line, rle\_line, size);  + line += size;  + rle\_line += size;  + srcSize -= size; |

| **Compliant Code** |
| --- |
| Target for Libfuzzer which helped uncover these vulnerabilities: |
| #include "FreeImage.h"  extern "C" int LLVMFuzzerTestOneInput(const uint8\_t \*data, size\_t size) {  FIMEMORY \*mem;  FIBITMAP \*dib;  int length, height, bpp, y;  FreeImage\_Initialise(true);  mem = FreeImage\_OpenMemory(const\_cast<uint8\_t \*>(data), size);  if (!mem)  return 0;  dib = FreeImage\_LoadFromMemory(FIF\_PSD, mem, PSD\_DEFAULT);  if (!dib)  return 0;  bpp = FreeImage\_GetBPP(dib);  length = FreeImage\_GetWidth(dib);  height = FreeImage\_GetHeight(dib);  for (y = 0; y < height; y++)  FreeImage\_GetScanLine(dib, y);  FreeImage\_Unload(dib);  FreeImage\_CloseMemory(mem);  FreeImage\_DeInitialise();  return 0;  } |

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

| **Principles(s):** Architect for security involves training the team in vulnerability API assessment, Adopt a Secure Coding standard new softwares do not contain such vulnerabilities, Quality Assurance Techniques to run pentesting tools in the CI/CD pipeline, adopt a coding standard to allow only certain secure open source software libraries in the project. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OpenVAS | 21.04 | Open Vulnerability Assessment Scanner. |  |
| OpenSCAP | 1.3.5 | https://www.open-scap.org |  |
| Nmap |  | https://www.nmap.org |  |
| Wireshark |  | https://www.wireshark.org |  |
| Metasploit |  | https://www.metasploit.com/ |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-009-CPP] | Object Oriented Programming is not inherently more secure than procedural or functional programming, it’s all in the hands of the programmer. Of particular note are securely invoking virtual functions and self-copying objects. Rule 9 of SEI Cert C++ Coding Standards involves proper Object Oriented Programming guidelines:  OOP50-CPP. Do not invoke virtual functions from constructors or destructors  Page:  OOP51-CPP. Do not slice derived objects  Page:  OOP52-CPP. Do not delete a polymorphic object without a virtual destructor  Page:  OOP53-CPP. Write constructor member initializers in the canonical order  Page:  OOP54-CPP. Gracefully handle self-copy assignment  Page:  OOP55-CPP. Do not use pointer-to-member operators to access nonexistent members  Page:  OOP56-CPP. Honor replacement handler requirements  Page:  OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions  Page:  OOP58-CPP. Copy operations must not mutate the source object |

| **Noncompliant Code** |
| --- |
| The copy operations for A *mutate the source* operand by resetting its member variable m to 0.  When std::fill() is called, the first element copied will have the original value of **obj.m**, *12*, at which point obj.m is set to 0. The subsequent nine copies will all retain the value 0. |
| #include <algorithm>  #include <vector>    class A {  mutable int m;    public:  A() : m(0) {}  explicit A(int m) : m(m) {}    A(const A &other) : m(other.m) {  other.m = 0;  }    A& operator=(const A &other) {  if (&other != this) {  m = other.m;  other.m = 0;  }  return \*this;  }    int get\_m() const { return m; }  };    void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

| **Compliant Code** |
| --- |
| the copy operations for A *no longer mutate the source operand*, ensuring that the vector contains equivalent copies of obj. Instead, A has been *given move operations* that perform the mutation when it is safe to do so. |
| #include <algorithm>  #include <vector>    class A {  int m;    public:  A() : m(0) {}  explicit A(int m) : m(m) {}    A(const A &other) : m(other.m) {}  A(A &&other) : m(other.m) { other.m = 0; }    A& operator=(const A &other) {  if (&other != this) {  m = other.m;  }  return \*this;  }    A& operator=(A &&other) {  m = other.m;  other.m = 0;  return \*this;  }    int get\_m() const { return m; }  };    void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

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

| **Principles(s):** Architect for security involves training the team in Secure OOP, Adopt a Secure Coding means adhering to the SEI Cert OOP compliant examples, KISS means prefer straightforward solutions over overly-complex ones, and adhere to principle of least privelege by using encapsulation and scope appropriately. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | overflow\_upon\_dereference  invalid\_function\_pointer |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP55 |  |
| Helix QAC | 2021.2 | C++2810, C++2811, C++2812, C++2813, C++2814 |  |
| Parasoft C/C++test | 2021.1 | 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 |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Expressions** | [STD-010-CPP] | Of particular note are concerns pertaining to reading unintialized memory or relying on the value of a moved-from object.  As local, automatic variables assume unexpected values if they are read before they are initialized, which can lead to undefined behavior.  EXP35-C. Do not modify objects with temporary lifetime  EXP63-CPP. Do not rely on the value of a moved-from object  EXP57-CPP. Do not cast or delete pointers to incomplete classes  EXP53-CPP. Do not read uninitialized memory  EXP51-CPP. Do not delete an array through a pointer of the incorrect type  EXP54-CPP. Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| An uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

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

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

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

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<> |
| CodeSonar | |  |  | | --- | --- | |  | 6.1p0 | | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | CERT\_CPP-EXP54-c    Do not use resources that have been freed  The address of an object with automatic storage shall not be returned from a function  The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist |
| Polyspace Bug Finder | R2021a | CERT C++: EXP54-CPP | Checks for:  Non-initialized variable or pointer  Use of previously freed pointer  Pointer or reference to stack variable leaving scope  Accessing object with temporary lifetime  Rule partially covered. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation can be created upon Build, by automating manual processes into a CI/CD pipeline, encourages team workflow, can use Docker for container instances, GitLab for versioning, Jenkins for CI.

Verify and test in SecOps by automating virtualized container deployment. Implement automated security tests and regression tests in QA.

Monitor and detection. Automate static application security tests into nightly builds on key sections of code. Embed dynamic application security testing into SDLC to look for vulnerabilities in real time.

Utilize tools such as OWASP Dependency-Check to check code dependency vulnerabilities.

### 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 | Likely | High | P9 | L2 |
| [STD-002-CPP] | High | Likely | High | P9 | 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 | Probable | Medium | P4 | L3 |
| [STD-008-CPP] | High | Probable | Medium | P4 | L3 |
| [STD-009-CPP] | High | Probable | High | P6 | L2 |
| [STD-010-CPP] | High | Probable | High | P6 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | We shall encrypt data at rest using full-disk encryption at the server level, as well as database encryption using MySQL Server, and provide a backup strategy. |
| Encryption at flight | Use up to date, secure libraries, use Public Key infrastructure for end-to-end protection on message bodies or attachments, use Managed File Transfer or SSH with expiration date on the link, password access, utilize Data leak prevention mechanisms built into cloud services, |
| Encryption in use | Utilize identity management mechanisms to confirm user roles and identity, allow conditional access to the tools functionality based on the user roles and other parameters. Use IRM digital rights management, to apply persistent protection to documentation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Process of identifying a user, using valid credentials of user and password. Control how a user is authenticated using a secured local database or external AWS server, prefer to use tried and trusted protocol. |
| Authorization | After the user has been authenticated, authorization shall be used to determine which resources and functionality the user is allowed to access and which operations can be performed. |
| Accounting | Monitor and log any user events while they are logging in/out or utilizing the resources, as well as user uptime or any other configured parameter. |

**\***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 |  |
| 1.5 | 10/09/2021 | Updated All Policies | Victor Feight | Professor Alhweiti |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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