**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 validation is like an air traffic controller helping flights to land. In order to land safely, each aircraft (input data) must adhere to a predetermined flight route (data format), preserve a constant altitude (data type), and stay within a certain speed range (value range). If an aircraft deviates from these criteria, it might lead to possible risks like crashes or off-course landings, similar to how unvalidated inputs can lead to system vulnerabilities like SQL injection or buffer overflows. Enforcing stringent input validation ensures that only secure, well-defined data enters the system, providing an essential line of defense against a wide variety of digital threats, similar to how strictly adhering to the rules of the air traffic controller ensures the safety and orderly management of the planes. It's the first line of defense against cyberattacks; this automated checking keeps your system's processes honest and dependable. |
| 1. Heed Compiler Warnings | Similar to cracks in a building's foundation, compiler warnings may be compared to faulty code. Although superficial cracks may not pose an immediate hazard to the building, they do serve as warning signs of underlying structural issues. If left unchecked, these fractures might widen and weaken the structure of the building, necessitating more extensive and difficult repairs. Comparably, when a compiler raises a warning, it's pointing out little "cracks" in your code that could cause security flaws, erratic behavior, or decreased performance if left unchecked. In the same way that a responsible building crew would check for and fix such flaws to preserve the structure, conscientious programmers would swiftly address compiler warnings to guarantee the resilience and security of the software they are developing. The goal of this kind of maintenance is to prevent the escalation of small issues into catastrophic breakdowns of the system. |
| 1. Architect and Design for Security Policies | Just as contemporary automobiles have safety measures built in from the start, so too should software security regulations be considered throughout the first phases of development. You wouldn't add airbags, anti-lock brakes, or seatbelts to a car that has already been built, and you shouldn't add security features to a software system that has already been built either. The creation of safety features in cars includes planning for them, engineering them, and testing them from the first plans to the finished product. Like physical structures, software needs security built up from the ground up. Just as a well-designed automobile will keep its passengers safe in the case of an accident, so too will this method guarantee that the final product is fundamentally secure and able to safeguard users from dangers. Putting security first from the very beginning of the software development process results in a more robust and resilient system, in the same way as putting safety first during the design phase of a car results in a vehicle that is better prepared to keep its occupants safe. |
| 1. Keep It Simple | The concept of simplicity in system design and coding is similar to the meticulous engineering and planning involved in constructing a highway. In order to facilitate travel and ensure the safety of its users, a highway should have unambiguous markings, regular entrances and exits, and distinct lanes. Simplicity in software also enhances readability and predictability. It gets rid of extraneous details and complications that might cause confusion and mistakes. A complicated codebase may include security flaws and bugs that are difficult to find and address, just as a highway with an excessive number of unclear intersections and poorly signposted routes may cause accidents and traffic jams. The smooth and safe flow of data "traffic" across the system is ensured by minimizing complexity, which improves the code's maintainability and makes it simpler for developers to do in-depth security audits. Minimalism is promoted by this philosophy, which favors simple, well-understood processes over complex, rarely-needed ones. More safe and dependable systems may be built using this method since it simplifies the development process and lessens the likelihood of security flaws. |
| 1. Default Deny | Implementing a default-deny approach in access controls mirrors the stringent security measures employed at a highly secure government facility. Envision an establishment wherein entry is strictly regulated, with all individuals being denied access by default. An authorized visitor must produce identification, pass a security check, and be given a time and date to enter the facility. If you don't meet all of these strict requirements, you won't be let in. This method is used to safeguard the building by allowing access only to authorized individuals who have a legitimate reason to be there. A comparable function is performed by a default-deny policy in software systems, which ensures that only trusted processes and users are able to access the system's resources. |
| 1. Adhere to the Principle of Least Privilege | In cybersecurity, least privilege is like a hotel's key card system. Each visitor is given a key card that allows them access to their accommodation and shared facilities like the pool and fitness center, but not to any other guest rooms or restricted areas. A manager, on the other hand, may have a master key card that unlocks all rooms but may still be locked out of the hotel's main computer system or the safe in the accounting room. The potential for harm to guests is greatly reduced by this approach, since the finder of a lost key card will be unable to use it to get access to any room other than the one associated with that card. Likewise, in a software system, granting the least amount of privilege required implies that if a user's credentials are exposed, the breach's effect is restricted to just what that person can access, avoiding full-scale system penetration. The idea is to provide security and reduce risk by limiting access to just what is necessary for each user. |
| 1. Sanitize Data Sent to Other Systems | Similar to the thorough examination and cleaning procedure that international mail takes before crossing borders, sanitizing data transported between systems is essential. Each package is checked for illicit drugs or dangerous chemicals that might be a threat to the nation it is being sent to. Contents are checked against stringent standards, and any potential threats are eliminated. The same goes for data, which is "inspected" before transmission to make sure it doesn't include any harmful code or information that shouldn't be publicized. Data sanitization techniques ensure the safe and secure transfer of digital information across networks in the same way that international customs officials try to stop the flow of illegal goods across borders. |
| 1. Practice Defense in Depth | The concept of defense in depth in cybersecurity is comparable to the integrated approach of modern military strategy, where various branches of the military collaborate to establish a multi-layered defense. The Navy is responsible for maritime security, the Air Force for aerial dominance, the Army for ground mobilization, and the Marines for quick reaction in times of crisis. This well-rounded strategy guarantees that if an opponent were to attack from one direction, they would encounter opposition from various sources, each with unique skills and strategies. Confronted with a multitude of obstacles and diverse defensive strategies, the adversary is presented with the formidable task of mounting a successful assault. In the same way, a multi-layered defensive plan in cyber security makes use of many different tools and methods. Firewalls are like the Navy in that they regulate traffic and prevent intruders from getting in. Like the Air Force, intrusion detection systems keep a watchful eye out for any dangers from above. The function of malware scanners is analogous to that of the Army operating on the ground since they are tasked with locating and neutralizing threats that have breached the perimeter. The specialized teams that proactively fortify defenses and react to breaches are comparable to secure coding methods and routine audits. Because of this concerted defense, the system's most precious resources are protected against a wide variety of dangers, and it can continue to function normally even if a single line of protection is breached. |
| 1. Use Effective Quality Assurance Techniques | Implementing rigorous quality assurance practices in software development is similar to the thorough pre-flight checks conducted on an aircraft before it takes off. Developers and QA specialists examine software using code reviews, automated testing, and vulnerability scanning, same as pilots and engineers analyze and test every system, from the engines to the electronics. In this way, a code review may be compared to a pilot's preflight check of the plane. The pilot is looking for any obvious problems with the aircrafts outside that may compromise flying performance. A code review is a process used in software development in which the program's creators check the program's source code for bugs and security vulnerabilities. Automated testing is similar to the avionics system inspections, in which the plane's internal diagnostics undergo a battery of tests to ensure everything is in working order before takeoff. Automated testing ensures that every software functions as intended and communicates with one another correctly. Security checks on people and their luggage have parallels in vulnerability scanning. Similar to how security guards check passengers' bags for harmful items, vulnerability scanners search for security flaws in the system that might be used by malicious actors. These quality assurance strategies work in tandem to identify and eliminate potential threats long before an airplane ever takes to the air. They assure that the software, much like an airplane, is dependable, secure, and ready for deployment, having been tested and inspected to satisfy the strictest criteria of reliability. |
| 1. Adopt a Secure Coding Standard | Comparable to the use of a set of building regulations in the construction sector, using a secure coding standard throughout all development activities helps ensure the safety of the final product. Secure coding standards provide a consistent framework for developers to follow, ensuring the security and stability of software applications, much in the same way that building codes are a standardized set of guidelines that all construction projects must follow to ensure structural integrity, fire prevention, and safety. A secure coding standard may, for instance, mandate input validation to forestall SQL injection attacks, just as a building must have a fire escape plan to guarantee safety. Secure coding standards necessitate encryption of sensitive data in the same way that building rules mandate the use of certain materials to resist earthquakes. Building rules specify the criteria for electrical systems, plumbing, and load-bearing buildings, coding standards such as OWASP's Top Ten and the SEI CERT Coding Standards serve as the blueprint, outlining the dos and don'ts of coding techniques. Just as a structure created to code serves its purpose and is safe for its inhabitants, so too can software developed in accordance with these plans guarantee the safety and security of its users from the ground up. |

### 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 | Never assign an enumeration value that is outside of its valid range.   * Assigning an enumeration value that is outside of its valid range in programming is similar to a bank teller trying to deposit money into a non-existent account number. Applying incorrect enumeration values in software may lead to undefined behavior, which in turn causes program crashes, security vulnerabilities, or inaccurate data, much as this can create transactional mistakes and confusion in banking. |

| **Noncompliant Code** |
| --- |
| The variable AccountStatus is converted to an integer in the wrong way by the function updateAccountStatus. Like a bank teller attempting to access account number three while the system only knows how to handle the numbers 0 for Active, 1 for Inactive, and 2 for Closed. There may be mistakes and discrepancies caused by the system trying to handle this incorrect account number. |
| enum AccountStatus { Active, Inactive, Closed };  void updateAccountStatus(int accountNumber, AccountStatus newStatus) {  AccountStatus status = static\_cast<AccountStatus>(accountNumber);  // Noncompliant    if (status == Closed) {  std::cout << "Account is closed. Cannot update status."  << std::endl;  } else {  // Process status update  std::cout << "Updating account status." << std::endl;  }  }  int main() {  updateAccountStatus(3, Closed);  // Invalid: 3 is not a valid AccountStatus  } |

| **Compliant Code** |
| --- |
| When using the conforming version, updateAccountStatus will accept an AccountStatus enumeration appropriately, much as a bank teller would when processing a transaction for an existing, legitimate account. This safeguards the system's integrity and stability by verifying the operation's validity and removing the possibility of processing an invalid account status. |
| enum AccountStatus { Active, Inactive, Closed };  void updateAccountStatus(AccountStatus status) {  if (status == Closed) {  std::cout << "Account is closed. Cannot update status."  << std::endl;  } else {  // Correctly process status update  std::cout << "Updating account status to " << status << std::endl;  }  }  int main() {  updateAccountStatus(Active);  // Valid: Active is a valid AccountStatus  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * It is of the utmost importance to utilize only correct enumeration values in banking systems, since these numbers could reflect many sorts of transactions, account statuses, etc. Invalid or altered transaction types might lead to mistakes or security holes, which is why this check is in place. 2. **Heed Compiler Warnings**:    * Possible enumeration misassignments are often the source of compiler warnings. To avoid logic mistakes that might impact account statuses or financial transactions, it is necessary to address these alerts in the context of banking software. 3. **Architect and Design for Security Policies**:    * Security standards mandating the use of legitimate enumeration ranges should be included into banking software architecture from the ground up. In doing so, we protect the system against inconsistencies and vulnerabilities caused by inconsistent treatment of enumerated data. 4. **Keep It Simple**:    * It is crucial to use enumerations within their valid ranges in banking applications to keep the code simple and intelligible. Complexity may lead to mistakes in financial processing, therefore this is vital. 5. **Default Deny**:    * For financial software, this concept means that by default, any enumeration value that is outside of the specified range should be refused. This helps to prevent security vulnerabilities or unexpected behavior. 6. **Adhere to the Principle of Least Privilege**:    * Applying this technique in banking applications helps to minimize the likelihood of abuse or mistakes by ensuring that enumeration variables only retain data that are essential for their intended usage. 7. **Practice Defense in Depth**:    * To further protect banking software against exploits using out-of-range numbers, a defense-in-depth strategy might include checking for enumeration value ranges at several points during the processing of transactions. 8. **Use Effective Quality Assurance Techniques**:    * Verifying accurate enumeration use is an important part of quality assurance in the banking industry. The application's enumeration values may be checked by automated tests to ensure they are within the anticipated range. 9. **Adopt a Secure Coding Standard**:    * The creation of software for financial institutions must adhere to a safe coding standard that specifies how enumeration values must be used. Errors or security holes caused by enumeration values that are too high or too low are less likely to occur when all developers adhere to established practices. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | Level 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Cast-integer-to-enum | Partially checked |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | Values assigned to an expression with an enum underlying type must match the enumerators of the enumeration. |
| Klocwork | 2023 | KW\_ENUM\_RANGE | One static code analysis tool that offers real-time feedback on code security and dependability is Klocwork. It interfaces with the CI/CD pipeline and IDEs. Incorrect assignments to enum variables may be caught by the KW\_ENUM\_RANGE check, which helps to avoid code integrity concerns.  <https://www.perforce.com/p/kw/try-klocwork-static-code-analyzer?utm_leadsource=cpc-googleadwords&utm_source=googleadwords&utm_medium=cpc&utm_campaign=KlocworkNABrand&utm_adgroup=Klocwork-BrandTerms-NA&gad_source=1&gclid=Cj0KCQiA67CrBhC1ARIsACKAa8RW2DP5sS0DPQpDwNevpA2NdToL6MHknOk_77okEtA5ktnRuKGCE40aAnKnEALw_wcB> |
| Coverity | 2023.1 | ENUM\_RANGE | One technique for finding software vulnerabilities and weaknesses is Coverity, which is part of the static application security testing (SAST) suite. An enumeration type variable's value being set beyond its range could cause undefined behavior or security problems; the ENUM\_RANGE checker can identify this.  <https://scan.coverity.com/> |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Prevent Use of Uninitialized Values   * In the same way that a bank teller must verify the opening balance of an account before processing any transactions for the day to avoid financial mistakes, programmers must initialize variables before using them to avoid ambiguity and possible security breaches. |

| **Noncompliant Code** |
| --- |
| Because the accountBalance is not initialized before being utilized in arithmetic operations, the ultimate balance is uncertain. Like a bank teller dealing with an uncertain initial amount, this might lead to inaccurate information being shown or processed. |
| #include <iostream>  #include <vector>  int main() {  int accountBalance;  // Noncompliant: accountBalance is not initialized    // Simulated transactions that might occur  std::vector<int> transactions = {100, -50, 25, -10};    for (int transaction : transactions) {  accountBalance += transaction;  //accountBalance is used in an operation,  //but it holds an unknown value  }    std::cout << "The final account balance is: " << accountBalance  << std::endl;  // The output is undefined    return 0;  } |

| **Compliant Code** |
| --- |
| Each transaction is processed against the known value of the accountBalance, which begins at zero, and the outcome is a clearly defined ultimate balance. In the field of accounting, precision and transparency are paramount, and this reflects that. |
| #include <iostream>  #include <vector>  int main() {  int accountBalance = 0;  // Compliant: accountBalance is initialized to a known value  // Simulated transactions that might occur  std::vector<int> transactions = {100, -50, 25, -10};    for (int transaction : transactions) {  accountBalance += transaction;  // accountBalance is correctly updated with each transaction  }    std::cout << "The final account balance is: " << accountBalance  << std::endl; // The output is well-defined.    return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * Data processing and transaction validation is of the utmost importance in financial applications. In order to avoid vulnerabilities or problems, such as erroneous transaction values caused by uninitialized variables, a kind of internal input validation is to initialize variables. This ensures that the data variables held before processing are legitimate and anticipated. 2. **Heed Compiler Warnings**:    * It is common for compilers to be able to identify potential uninitialized variable use. Paying great attention to and fixing these alerts is critical in banking software development for avoiding any operational and security issues linked to uninitialized variables. 3. **Architect and Design for Security Policies**:    * Including clear instructions for variable initialization in security rules is a must when building banking system architecture. By doing this, any weaknesses that can allow for illegal access to or alteration of private financial data are avoided. 4. **Keep It Simple**:    * Clean, uncomplicated code is promoted by the simplicity concept in financial application development. Complexity and the possibility of mistakes leading to major financial consequences or data breaches are both reduced with correctly initialized variables. 5. **Default Deny**:    * The creation of financial software should adhere to a "default deny" policy, which states that variables should not be utilized unless they are manually populated. This method lowers the chance of handling deals or doing estimates based on data that is wrong or hard to guess. 6. **Adhere to the Principle of Least Privilege**:    * This guiding concept guarantees that all processes and users in financial systems have minimal access. As a coder, you should make sure that variables aren't storing any sensitive or random data since they aren't initialized, and that they only have access to the data they require for their intended function. 7. **Practice Defense in Depth**:    * Several checks and validations, such as making sure all variables are initialized, may be part of defense in depth for banking applications. This protects against the possible abuse of uninitialized variables by adding another level of protection. 8. **Use Effective Quality Assurance Techniques**:    * There should be stringent checks for uninitialized variables in banking software's quality assurance. This is crucial because mistakes like this may cause people to lose money, have their accounts overdrawn, or be susceptible to fraud. 9. **Adopt a Secure Coding Standard**:    * Building software for financial institutions requires the adoption and strict adherence to a safe coding standard that specifies how variables should be initialized. This lessens the likelihood of security flaws by making sure the development team consistently applies recommended practices. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | Level 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | VariableInitilizationCheck | Uninitialized variable checks are only one part of SonarQube's extensive code quality and security inspection.  <https://www.sonarsource.com/products/sonarqube/whats-new/sonarqube-9-8/> |
| Coverity Scan | 2023.2 | UNINIT\_CTOR | Static code analysis is provided by Coverity, which finds possible vulnerabilities such as uninitialized variables, untrusted loop boundaries, and more.  <https://scan.coverity.com/> |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Fortify Static Code Analyzer | 22.1 | InitializationCheck | A collection of tools called Fortify SCA can do static application security testing (SAST) and find security flaws in code.  <https://www.devopsschool.com/blog/what-is-fortify-and-how-it-works-an-overview-and-its-use-cases/> |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensure Sufficient Storage for String Data   * In programming, a string has to be allotted enough memory to hold all of its characters plus the null terminator, much like a bank vault needs to be big enough to safely keep all of its contents. In a financial context, this prevents buffer overflows, which are like leaving important possessions outside the vault because there isn't enough room inside, which might lead to theft. |

| **Noncompliant Code** |
| --- |
| Not properly checking the length before copying a string into an allocated buffer can result in a buffer overflow, which is like overstuffing a vault with more valuables than it can handle securely. |
| #include <cstring>  #include <iostream>  void storeAccountName(char \*dest, const char \*accountName) {  // Noncompliant: strcpy does not check for buffer size  std::strcpy(dest, accountName);  }  int main() {  char accountName[10];  // The name is too long for the accountName buffer,  // leading to buffer overflow  storeAccountName(accountName, "Hope Exnicious");  std::cout << "Account name stored: " << accountName << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Like planning a vault to safely store all objects, using strncpy to copy the string guarantees that the boundaries of the destination buffer are maintained. It prevents overflow by writing only as much data as may be securely stored. |
| #include <cstring>  #include <iostream>  void storeAccountName(char \*dest, size\_t destSize, const char \*accountName) {  // Compliant: strncpy checks for buffer size and avoids overflow  std::strncpy(dest, accountName, destSize - 1);  dest[destSize - 1] = '\0'; // Ensure null termination  }  int main() {  char accountName[25];  // The buffer is now sufficiently sized for the accountName  storeAccountName(accountName, sizeof(accountName),  "Hope Exnicious");  std::cout << "Account name stored: " << accountName << std::endl;  return 0;  } |

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

| **Principles(s):**  Validate Input Data:   * A thorough validation of the length and content of all input data, including strings that will be transferred to buffers, is required. This prevents buffer overflows, which may happen while processing client information, account names, and transaction details routinely, and illegal command execution. This is especially important in banking applications.   Heed Compiler Warnings:   * Compilers have the ability to detect and alert developers about possible problems related to manipulating strings. Bank developers need to pay attention to these warnings so that security holes that could be used with string operations don't happen.   Architect and Design for Security Policies:   * Critical in banking, where the integrity of transactional data is crucial, design and architecture should include regulations that avoid string-related vulnerabilities.   Keep It Simple:   * The likelihood of mistake reduction is directly proportional to the simplicity of the method used to handle strings. Simple coding principles lessen the likelihood of complicated security flaws in financial institutions.   Default Deny:   * The standard practice when dealing with strings is to prevent operations that might overflow their allocated buffers. This implies that any feature in banking software that does not guarantee string safety need to be avoided.   Adhere to the Principle of Least Privilege:   * Code should only have the rights it needs to do its action, and nothing more. In the financial industry, where client data must be protected at all costs, this implies that functions should only access the amount of memory they require for string operations and no more.   Sanitize Data Sent to Other Systems:   * When banking systems talk to each other, it's important to clean up all the data, even strings, so that the receiving systems don't have to deal with buffer overflow risks.   Practice Defense in Depth:   * In order to prevent a buffer from overflowing, it is recommended to include many levels of checks. This is particularly relevant in the financial sector, as several validation points might detect mistakes that a single point could overlook.   Use Effective Quality Assurance Techniques:   * Bugs in string processing and buffer overflows should be tested as part of quality assurance. When it comes to banking, this includes making sure all features that deal with account and transaction data are thoroughly tested.   Adopt a Secure Coding Standard:   * The purpose of a secure coding standard is to provide best practices for handling strings in a secure manner. The integrity of financial data and protection against exploitation depend on the banking sector adopting such standards. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | Level 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.8 | bufferOverflow | Cppcheck is a program that analyzes C and C++ programs statically. Bugs like buffer overflows are among the many types of faults it may find in your code. Because it is open-source, it is compatible with CI systems.  <http://cppcheck.sourceforge.net/> |
| Clang Static Analyzer | 14.0.0 | CStringChecker | The Clang Static Analyzer’s comprehensive analysis capabilities, which include checks for possible string manipulation concerns, have made it a well-known component of the LLVM project.  <https://clang-analyzer.llvm.org/> |
| Coverity Scan | 2023.1 | STRING\_OVERFLOW | Bugs like buffer overflows and security holes may be found with the help of Coverity's thorough static code analysis.  <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html> |
| Visual Studio Code Analysis | 2022 or 2019 | C6001 | Buffer overflow checks are only one of several C++-specific code analysis features offered by this tool, which is an integral part of the Visual Studio integrated development environment.  <https://visualstudio.microsoft.com/> |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Protect Against SQL Injection via Smart Pointer Management   * Although smart pointer management may not have a direct connection to SQL Injection, following smart pointer rules can indirectly help prevent SQL Injection. One way to ensure good resource management and avoid injection attacks is to make sure that smart pointers uniquely own database connections and prepared statement objects. Handling dynamic resources with smart pointers is akin to relying on a single clerk to manage your bank account. The same way that having many clerks handle the same account might lead to mistakes and confusion, having multiple smart pointers assigned the same dynamically created object could cause duplicate deletion and unpredictable functionality. It is similar to having a transparent, responsible chain of custody for financial transactions in that it ensures unique ownership of dynamic resources using smart pointers, hence preventing operational errors and upholding transactional authenticity. |

| **Noncompliant Code** |
| --- |
| The BankAccount object is managed by two unique\_ptr objects. This is a potential issue with the use of smart pointers that may lead to the unintended deletion of the BankAccount object. |
| #include <memory>  #include <iostream>  class BankAccount {  public:  BankAccount(int accountId) : id(accountId) {}  void display() const { std::cout << "Account ID: " << id  << std::endl; }  private:  int id;  };  int main() {  BankAccount\* account = new BankAccount(12345);  std::unique\_ptr<BankAccount> clerkOne(account);  std::unique\_ptr<BankAccount> clerkTwo(account);  // Noncompliant: Both clerks should not own the same account  clerkOne->display();  // Undefined behavior: clerkTwo may also try to manage  // or delete the account  return 0;  } |

| **Compliant Code** |
| --- |
| ClerkOne is in charge of the BankAccount at first, while clerkTwo later acquires ownership. Limiting account access to a single clerk (smart pointer) at a time prevents accidental duplication of effort and guarantees efficient use of resources. |
| #include <memory>  #include <iostream>  class BankAccount {  public:  BankAccount(int accountId) : id(accountId) {}  void display() const { std::cout << "Account ID: " << id  << std::endl; }  private:  int id;  };  int main() {  std::unique\_ptr<BankAccount> clerkOne  = std::make\_unique<BankAccount>(12345);  std::unique\_ptr<BankAccount> clerkTwo = std::move(clerkOne);  // Compliant: Ownership transferred to clerkTwo  if (!clerkOne) {  std::cout << "Clerk One no longer manages the account."  << std::endl;  }  clerkTwo->display(); // Safe: clerkTwo uniquely owns the account  return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * As a general rule, you should check the validity of data, including pointers, before using them. By effectively using smart pointers, data corruption may be prevented in banking software by ensuring that resources, such as database connections, are legitimate and uniquely owned. 2. **Practice Defense in Depth**:    * To prevent memory leaks or hanging pointers, which might be abused in a banking application, many safety checks are implemented, one of which is smart pointer usage. 3. **Adhere to the Principle of Least Privilege**:    * Objects should only be accessed by functions that really require them. To minimize the possibility of unauthorized access, smart pointers should be used effectively to guarantee that only the program portions that really require resources may access and manage them. 4. **Keep It Simple**:    * A crucial component of the intricate banking software environment is the correct use of smart pointers, which simplifies memory management and reduces the chance of mistakes while managing dynamic resources. 5. **Default Deny**:    * A resource is not considered to be securely accessible unless it is held by a smart pointer explicitly. For the safe management of financial data, this delicate approach is essential. 6. **Use Effective Quality Assurance Techniques**:    * In order to avoid flaws that might cause major financial mistakes, it is crucial to implement comprehensive QA procedures in banking apps. This includes checking for the correct use of smart pointers. 7. **Adopt a Secure Coding Standard**:    * When creating safe and reliable banking software, it is crucial to adhere to a secure coding standard that addresses smart pointer best practices. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | Level 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.8 | smartPointerUsage | Critical to the management of software resources in financial institutions, Cppcheck can identify a variety of patterns of smart pointer usage.  <http://cppcheck.sourceforge.net/> |
| Clang Static Analyzer | 14.0.0 | SmartPtrChecker | Important to avoid in financial applications, this tool may detect typical C++ smart pointer issues, such as double-free errors.  <https://clang-analyzer.llvm.org/> |
| Studio Code Analysis | 2022 or 2019 | CppCoreCheckSmartPointers | This Visual Studio add-in verifies that your C++ code follows standard procedures for smart pointer use.  <https://visualstudio.microsoft.com/> |
| Coverity | 2023.1 | USE\_AFTER\_FREE | Accessing freed memory is a key problem in banking systems, and Coverity's static analysis may uncover incorrect usage of smart pointers.  <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html> |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Avoid Access After Free   * Accessing memory that has been freed is similar to a bank teller attempting to access a closed account. Memory that has been freed up should not be accessed or altered in the same way that a closed account should not be. Much to a bank experiencing a security crisis as a consequence of mismanaged accounts, such an effort might cause the software to behave in an unanticipated way, perhaps damaging its memory or leading to security breaches. |

| **Noncompliant Code** |
| --- |
| The software makes an erroneous attempt to access the show function after removing the account, which requests the destructor and frees the memory. Similar to a bank mistake that occurs while trying to access a closed account, this undefined behavior might cause catastrophic software issues. |
| #include <iostream>  class BankAccount {  public:  BankAccount(int accountId) : id(accountId) {  std::cout << "Bank account " << id << " opened." << std::endl;  }  ~BankAccount() {  std::cout << "Bank account " << id << " closed." << std::endl;  }  void display() const {  std::cout << "Displaying account ID: " << id << std::endl;  }  private:  int id;  };  int main() {  BankAccount\* account = new BankAccount(123);  delete account;  account->display();  // Noncompliant: Accessing memory after it has been freed  return 0;  } |

| **Compliant Code** |
| --- |
| Prior to account deletion, the display method is invoked. The right and secure method to handle dynamic memory is to call delete so that the reference is no longer utilized. In the same way that a bank would correctly archive records of closed accounts, this guarantees that the application operates as intended. |
| #include <iostream>  class BankAccount {  public:  BankAccount(int accountId) : id(accountId) {  std::cout << "Bank account " << id << " opened." << std::endl;  }  ~BankAccount() {  std::cout << "Bank account " << id << " closed." << std::endl;  }  void display() const {  std::cout << "Displaying account ID: " << id << std::endl;  }  private:  int id;  };  int main() {  BankAccount\* account = new BankAccount(123);  account->display();  delete account;  // After this point, account is no longer valid and  // should not be used.  // No further access to account  return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * For security and integrity reasons, it is as important to check that memory pointers are legitimate and not leading to freed memory as it is to check the legitimacy of transaction requests in banking. 2. **Practice Defense in Depth**:    * Layers of security may be added by using several checks, such as smart pointers and validating pointers. This is analogous to the many verification stages used in financial transactions. 3. **Adhere to the Principle of Least Privilege**:    * Similarly to how limiting employee access to just the essential financial information may lessen the danger of accessing invalid memory locations, restricting a function's memory access to just what it requires can do the same. 4. **Keep It Simple**:    * Just as plain financial procedures limit the danger of transactional mistakes in banking, simple and unambiguous administration of dynamic memory minimizes the probability of errors. 5. **Default Deny**:    * Similar to how banks reject transactions until all checks are passed, any memory access to unallocated or released memory should be rejected by default. 6. **Use Effective Quality Assurance Techniques**:    * Thorough auditing is required in financial operations, and rigorous testing is vital to discover memory usage. 7. **Adopt a Secure Coding Standard**:    * Banking software must adhere to a standard that prohibits access to freed memory in order to be secure and reliable. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | Medium | Level 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.8 | MemoryLeak | The dependability and safety of financial software relies on Cppcheck's ability to identify memory leaks and possible access to freed memory.  <http://cppcheck.sourceforge.net/> |
| Valgrind | 3.18.1 | Memcheck | Banking software relies on Valgrind's Memcheck tool, which is essential for finding memory leaks and access to freed memory in C/C++ programs.  <https://valgrind.org/> |
| AddressSanitizer | 14.0.0 | AddressSanitizer | To assist avoid critical vulnerabilities in financial applications, AddressSanitizer, a quick memory error detector, may identify usage after free.  <https://clang.llvm.org/docs/AddressSanitizer.html> |
| Visual Studio Code Analysis | 2022 or 2019 | C6001 | <https://visualstudio.microsoft.com/> |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Resource Management with Assertions   * Removing assertions in the production build, a common practice, can potentially result in resource leaks. You shouldn't manage program logic or resources using assertions; they should only be used for debugging. By comparing it to a bank, it's easy to see how keeping track of transactions in a temporary ledger without first verifying them in the permanent record might lead to financial inconsistencies if the two aren't correctly transferred. |

| **Noncompliant Code** |
| --- |
| The code in question utilizes an assertion to verify whether the BankAccount pointer is nullptr. This check would not be present in the production environment since assertions are removed from the produced code in release mode (with NDEBUG set). Because to the lack of appropriate exception handling, resources allocated inside BankAccount (maybe a file or database connection) would not be released if processAccount is invoked with a nullptr, resulting in resource leaks. |
| #include <cassert>  #include <iostream>  class BankAccount {  public:  BankAccount() {  // Resource allocation such as opening a file or  // database connection  std::cout << "BankAccount constructed  and resources allocated.\n";  }  ~BankAccount() {  // Resource deallocation  std::cout << "BankAccount destructed and  resources deallocated.\n";  }  };  void processAccount(BankAccount\* account) {  assert(account != nullptr && "Account cannot be null");  // Noncompliant: Asserts may be disabled in production  // Account processing logic  std::cout << "Processing account.\n";  }  int main() {  BankAccount\* myAccount = nullptr;  // Simulating an error condition  // The assertion here is intended to catch the error, but  // will not be present in a release build  processAccount(myAccount);  delete myAccount;  // In the case of an error condition, this could lead to  // undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| Even in production, the check for a nullptr is always done since the assertion is replaced with an exception throw in the conforming code. You may catch and handle the std::invalid\_argument exception that is issued if processAccount gets a nullptr. This ensures that whatever resources that BankAccount has assigned may be safely released upon destruction, preventing any possible loss of resources. |
| #include <stdexcept>  #include <iostream>  class BankAccount {  public:  BankAccount() {  // Resource allocation such as opening a file or  // database connection  std::cout << "BankAccount constructed and  resources allocated.\n";  }  ~BankAccount() {  // Resource deallocation  std::cout << "BankAccount destructed and  resources deallocated.\n";  }  };  void processAccount(BankAccount\* account) {  if (account == nullptr) {  throw std::invalid\_argument("Account cannot be null");  }  // Account processing logic  std::cout << "Processing account.\n";  }  int main() {  try {  BankAccount\* myAccount = new BankAccount();  // Properly allocating a new account  processAccount(myAccount);  // Safe processing  delete myAccount;  // Safe deallocation of resources  } catch (const std::invalid\_argument& e) {  std::cerr << e.what() << std::endl;  // Additional error handling logic  }  return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * For security and integrity reasons, it is as important to check that memory pointers are legitimate and not leading to freed memory as it is to check the legitimacy of transaction requests in banking. 2. **Practice Defense in Depth**:    * Similar to how there are several verification processes in financial transactions, using many checks, such as validating pointers and creating smart pointers, provides layers of security. 3. **Adhere to the Principle of Least Privilege**:    * Similarly to how limiting employee access to just the essential financial information may lessen the danger of accessing invalid memory locations, restricting a function's memory access to just what it requires can do the same. 4. **Keep It Simple**:    * Errors are less likely to occur when dynamic memory is carefully and clearly managed, just as they are when banking transactions are kept simple. 5. **Default Deny**:    * By default, no memory access to freed or unallocated memory should be allowed. This is similar to how banks prevent transactions until all checks are passed. 6. **Use Effective Quality Assurance Techniques**:    * Just as comprehensive auditing is vital in financial operations, rigorous testing to detect memory usage is critical. 7. **Adopt a Secure Coding Standard**:    * The security and dependability of financial software depends on following a standard that prohibits access to freed memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Low | High | Level 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.8 | ResourceLeak | Important for banking software exception handling, Cppcheck checks for possible resource leaks.  <http://cppcheck.sourceforge.net/> |
| Visual Studio Code Analysis | 2022 or 2019 | C6211 | Banking apps can be more reliably built with the help of Visual Studio's code analysis tool, which can find resource leaks like those that could happen in exception handling.  <https://visualstudio.microsoft.com/> |
| Valgrind | 3.18.1 | Memcheck | The efficient administration of banking software relies on the detection of memory leaks, a prevalent resource leak problem, and Valgrind's Memcheck utility does just that.  <https://valgrind.org/> |
| Coverity | 2023.1 | RESOURCE\_LEAK | Critical for financial software, coverity may detect resource leaks in complicated C++ programs, such as those resulting from incorrect exception handling.  <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html> |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle Pre-Main Exceptions Safely   * Dealing with exceptions that arise during the initialization phase is comparable to a bank conducting a thorough system check before welcoming customers through its doors. The program must handle exceptions that occur before the primary function to avoid crashes at startup, much as the bank must ensure all systems are working to prevent service interruptions. For banking software, this may include making sure the connection to the database is secure before processing any transactions. |

| **Noncompliant Code** |
| --- |
| A DatabaseConnection object is made as a static global object. The program will terminate because the try-catch block in main will not catch an error that occurs during its startup (for example, while trying to connect to the database). |
| #include <stdexcept>  #include <iostream>  class DatabaseConnection {  public:  DatabaseConnection() {  throw std::runtime\_error("Database connection failed.");  }  };  // Global static object  DatabaseConnection globalDatabaseConnection;  int main() {  // The globalDatabaseConnection constructor has thrown  // before reaching this point.    // As a result, the program will terminate, and the following  // code will not execute.    std::cout << "Main function started, but this  message will never be displayed." << std::endl;    // The code intended for transaction processing is not  // reached due to the exception.  return 0; // This return statement is also never reached.  } |

| **Compliant Code** |
| --- |
| An encapsulating method that uses the DatabaseConnection and returns a local, static object is used. If an error is encountered, this pattern will catch it in the main function's call to getDatabaseConnection() and handle it correctly. |
| #include <stdexcept>  #include <iostream>  class DatabaseConnection {  public:  DatabaseConnection() {  // If the connection attempt fails, an exception is thrown  // For this example, assume the connection is always successful  std::cout << "Database connection successfully established."  << std::endl;  }  };  int main() {  try {  DatabaseConnection dbConnection;  // DatabaseConnection object is instantiated here    std::cout << "Main function started." << std::endl;    // Example transaction processing code  std::cout << "Processing transactions..." << std::endl;    // Simulate successful transaction processing  std::cout << "Transactions processed successfully."  << std::endl;  } catch (const std::runtime\_error& e) {  std::cerr << "Failed to establish database connection: "  << e.what() << std::endl;  return 1; // Return an error code to indicate failure  }  // Normal program termination  std::cout << "Program terminated successfully." << std::endl;  return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * In the context of exception handling, this approach means making sure that all possible exception situations are evaluated and handled, particularly during the startup process of financial software. Normally, it pertains to user input. 2. **Practice Defense in Depth**:    * To make banking applications more resilient and prevent unexpected crashes or behavior, it is recommended to include many levels of exception handling, particularly for actions before main(). 3. **Adhere to the Principle of Least Privilege**:    * The likelihood of exceptions that might affect the stability of banking software is decreased by limiting actions prior to main() to the bare minimum of functionality that is required. 4. **Keep It Simple**:    * Conforming to the banking idea of uncomplicated financial processing, minimizing complicated procedures prior to main() helps to decrease the likelihood of exceptions. 5. **Default Deny**:    * It's comparable to how resolving all exceptions avoids unexpected program states; both may be used to prevent illegal transactions in financial systems. 6. **Use Effective Quality Assurance Techniques**:    * Maintaining service dependability in banking applications requires robust testing to manage exceptions, especially during the early execution process. 7. **Adopt a Secure Coding Standard**:    * Building trustworthy financial software is easier when developers adhere to a coding standard that stresses thorough exception handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | High | High | Level 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | exceptionSafety | For banking software to be resilient against exceptions throughout all execution stages, Cppcheck can discover possible safety concerns.  <http://cppcheck.sourceforge.net/> |
| Visual Code Studio | 2022 or 2019 | CppCoreCheckExceptions | Important for the first phases of program execution in banking applications, this tool verifies C++ code for exception safety.  <https://visualstudio.microsoft.com/> |
| Clang Static Analyzer | 14.0.0 | ExceptionSafety | Improving the dependability of financial applications, this tool may detect exception handling flaws, including those that may occur before main().  <https://clang-analyzer.llvm.org/> |
| Polyspace Bug Finder | R2023b | CERT C++:ERR58-CPP | Checks for exceptions raised during program startup (rule fully covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Execution Boundary Safety | [STD-008-CPP] | Execution Boundary Safety   * This coding standard emphasizes the importance of avoiding the transfer of nonstandard-layout type objects across execution boundaries within the framework of a banking application. When data moves from one execution context to another, such an OS's user mode to its kernel mode or between processes, this is called an execution boundary. If you don't follow this rule, you could end up with unpredictable behavior, memory loss, or security holes. This is especially dangerous in a banking system. |

| **Noncompliant Code** |
| --- |
| In the code provided, there is a BankAccount class that does not meet the standard-layout requirement. The processTransaction function receives this nonstandard-layout object as an argument across execution boundaries. An increase in security concerns and erratic behavior may result from this breach. |
| #include <iostream>  class BankAccount {  int accountNumber;  double balance;  public:  BankAccount(int number, double initialBalance)  : accountNumber(number), balance(initialBalance) {}  void deposit(double amount) {  balance += amount;  }  void withdraw(double amount) {  if (balance >= amount) {  balance -= amount;  } else {  std::cout << "Insufficient funds." << std::endl;  }  }  double getBalance() const {  return balance;  }  };  void processTransaction(const BankAccount& account, double amount) {  // Processing a transaction  }  int main() {  BankAccount userAccount(12345, 1000.0);  processTransaction(userAccount, 200.0);  // Passing a nonstandard-layout object (BankAccount) across  // execution boundaries  return 0;  } |

| **Compliant Code** |
| --- |
| The code constructs a class called BankAccount, which is a representation of a bank account. This class has properties like a balance, methods to add and remove cash, and a place to store the account number. Additionally, it has a transaction processing method called processTransaction. Initiating a transaction in the main function entails creating an object of a BankAccount, assigning it a number and a starting amount, and then running processTransaction on it. Good coding principles are followed by the code by enclosing data inside the class, providing member methods for operations, and ensuring that the processTransaction function does not alter the account using const-correctness. |
| #include <iostream>  class BankAccount {  int accountNumber;  double balance;  public:  BankAccount(int number, double initialBalance)  : accountNumber(number), balance(initialBalance) {}    // Other member functions...  void deposit(double amount) {  balance += amount;  }    void withdraw(double amount) {  if (balance >= amount) {  balance -= amount;  } else {  std::cout << "Insufficient balance for withdrawal."  << std::endl;  }  }    double getBalance() const {  return balance;  }  };  void processTransaction(const BankAccount& account, double amount) {  // Processing a transaction  // For this example, we'll just simulate a transaction  // by depositing the amount into the account.  account.deposit(amount);  }  int main() {  BankAccount userAccount(12345, 1000.0);    // Performing a transaction (deposit)  double transactionAmount = 200.0;  processTransaction(userAccount, transactionAmount);    // Checking the updated balance  std::cout << "Updated balance: $" << userAccount.getBalance()  << std::endl;    return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data**:    * Just as it is essential to check financial transactions at each stage, data integrity validation while crossing execution boundaries is vital in banking software. By checking that objects follow a predefined structure before sending them across borders, we can keep data consistent and stop undefined behavior. 2. **Practice Defense in Depth**:    * Multiple checks, like type validation and memory safety checks, protect against data abuse or security holes that could happen when objects are passed between banking systems. 3. **Adhere to the Principle of Least Privilege**:    * Control what objects may be accessed and what can be transmitted across execution boundaries. The banking analogy here would be to authorize just the minimum set of rights required to complete a transaction or access certain data. 4. **Keep It Simple**:    * There is less room for mistakes when objects' interactions across execution boundaries are simplified, just as there is less room for error when banking transactions are simplified. 5. **Default Deny**:    * Passing sophisticated, nonstandard-layout objects across execution boundaries should be disabled by default, much as a bank could reject transactions that don't match their standards. 6. **Use Effective Quality Assurance Techniques**:    * Similar to how full audits and compliance checks are critical in financial operations, rigorous testing is needed to verify that only objects with a standard layout are transmitted across execution boundaries. 7. **Adopt a Secure Coding Standard**:    * When creating safe banking software, it is essential to use a coding standard that highlights the need of safe object passing techniques. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Low | High | Level 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | objectLayoutCheck | Important for the security of financial applications, Cppcheck may examine object structures to guarantee they are safe to transfer over execution boundaries.  <http://cppcheck.sourceforge.net/> |
| Clang Static Analyzer | 14.0.0 | ObjectBoundaryCheck | Ensuring the stability of banking applications relies on this tool's ability to check for possible errors when objects are transferred over execution boundaries.  <https://clang-analyzer.llvm.org/> |
| Visual Studio Code Analysis | 2022 or 2019 | CppObjectBoundary | To improve the safety of financial applications, this analysis tool is included into Visual Studio and helps make sure that objects that cross borders follow standard design.  <https://visualstudio.microsoft.com/> |
| Coverity | 2023.1 | ExecutionBoundaryChecker | For financial software to run safely, coverage is essential for detecting any issues with passing objects over execution boundaries.  <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html> |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object-Oriented Programming | [STD-009-CPP] | No Virtual Calls in Constructors/Destructors   * Because virtual functions may access derived class sections that have not been initialized or deleted yet, using them in constructors or destructors might cause undefined behavior. Because of the reliance on inheritance in banking software and other complicated class hierarchies, this might cause subtle errors. |

| **Noncompliant Code** |
| --- |
| In the given code example, a virtual function auditTrail() is invoked within the constructor of a BankAccount class. This can result in a scenario where the auditTrail() function of the derived class is invoked prior to the complete construction of the derived class. |
| class BankAccount {  public:  BankAccount() {  // Initialization code for the bank account  auditTrail();  // Noncompliant: Virtual function call in constructor  }  virtual void auditTrail() const {  std::cout << "General BankAccount audit trail." << std::endl;  }  };  class SavingsAccount : public BankAccount {  public:  double interestRate;  SavingsAccount(double rate) : BankAccount(), interestRate(rate) {  // The derived class's constructor body is executed  // after the base class's constructor  }  void auditTrail() const override {  std::cout << "SavingsAccount audit trail. Interest rate: "  << interestRate << std::endl;  }  }; |

| **Compliant Code** |
| --- |
| In the given code example, the constructor does not invoke the virtual function. Rather, it is called after the object is completely built, guaranteeing that the appropriate function is run for the object's dynamic type. |
| class BankAccount {  public:  BankAccount() {  // Initialization code for the bank account  }  virtual void auditTrail() const {  std::cout << "General BankAccount audit trail." << std::endl;  }  void initialize() {  // Additional initialization code as necessary  auditTrail();  // Compliant: Called after construction is complete  }  };  class SavingsAccount : public BankAccount {  public:  double interestRate;  SavingsAccount(double rate) : BankAccount(), interestRate(rate) {  // The derived class's constructor body is executed  // after the base class's constructor  }  void auditTrail() const override {  std::cout << "SavingsAccount audit trail. Interest rate: "  << interestRate << std::endl;  }  };  // Usage:  // SavingsAccount \*account = new SavingsAccount(0.05);  // account->initialize();  // Properly calls SavingsAccount::auditTrail after full construction |

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

| **Principles(s):**   1. **Validate Input Data**:    * The production and destruction processes of objects are guaranteed to be intact by this concept. Just as processing unvalidated cash transactions may lead to mistakes or fraud, using virtual parameters in constructors or destructors in banking software might cause unexpected consequences. 2. **Practice Defense in Depth**:    * Using several verification levels in banking activities, avoiding virtual calls in constructors and destructors provides a layer of safety to prevent unwanted activity during object construction and destruction. 3. **Adhere to the Principle of Least Privilege**:    * Following the principle of minimal access in financial systems, constructors and destructors should accomplish the bare minimum to put up and break down an object, without incurring complicated structures or relationships. 4. **Keep It Simple**:    * One way to simplify object lifecycle management and reduce the possibility for mistakes in complicated banking systems is to avoid calling virtual functions in constructors and destructors. 5. **Default Deny**:    * Like with conservative banking processes, where operations are limited until specifically authorized, virtual function calls should not be part of constructors and destructors by default. 6. **Use Effective Quality Assurance Techniques**:    * It is just as important to conduct thorough audits of financial systems as it is to test constructors and destructors to detect and avoid the abuse of virtual functions. 7. **Adopt a Secure Coding Standard**:    * Bank software must adhere to a coding standard that forbids virtual calls in constructors and destructors if it is to remain stable and dependable. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Medium | Medium | Level 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.8 | virtualFunctionCallInContructor | One way to make sure that banking software uses safe object lifecycle management is by using Cppcheck to find virtual function calls in constructors and destructors.  <http://cppcheck.sourceforge.net/> |
| SonarQube | 9.8 | VirtualCallinConstructorOrDestructor | Many object-oriented programming difficulties may be found in C++ code using SonarQube. One of these concerns is the dangerous habit of using virtual functions in constructors and destructors. This is of the utmost importance for banking software, since the stability and security of the application depend on class structures and the established nature of constructors and destructors.  <https://www.sonarqube.org/> |
| Clang-Tidy | 14.0.0 | Cppcoreguidelines-special-member-functions | Clang-Tidy contains tests for the C++ Core Guidelines, which are essential for banking applications and cover the proper usage of special member functions.  <https://clang.llvm.org/extra/clang-tidy/> |
| Visual Studio Code Analysis | 2022 or 2019 | CppCoreCheckSpecialRules | The use of virtual functions in constructors and destructors is one of the fundamental criteria that this Visual Studio tool checks for in C++ code.  <https://visualstudio.microsoft.com/> |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | [STD-010-CPP] | Thread-Safe Resource Access   * Data races, caused by undefined behavior and possible security risks, may occur when many processes attempt to access shared resources at the same time without synchronization. To avoid race situations and keep data intact in a banking system where several transactions may access and change shared financial data, thread-safe operations are essential. |

| **Noncompliant Code** |
| --- |
| A data race may occur if several threads attempt to invoke the makeTransaction method at the same time because it changes a global shared variable without guaranteeing exclusive access to it. |
| // Global shared balance for a banking application  double sharedBalance = 10000.00;  void makeTransaction(double amount) {  sharedBalance += amount;  // Noncompliant: Direct access to shared resource  // without synchronization  }  // Usage in a concurrent context might look like this  std::thread t1(makeTransaction, 500.00);  std::thread t2(makeTransaction, -200.00);  t1.join();  t2.join(); |

| **Compliant Code** |
| --- |
| A mutex is introduced by the complying code to synchronize access to the sharedBalance. In order to avoid deadlocks, the lock\_guard makes sure that the mutex is immediately released when the scope is vacated. This design technique ensures that sharedBalance may be updated without causing any thread safety issues. |
| #include <mutex>  // Global shared balance for a banking application  double sharedBalance = 10000.00;  std::mutex balanceMutex;  // Mutex to protect sharedBalance  void makeTransaction(double amount) {  std::lock\_guard<std::mutex> guard(balanceMutex);  // Lock the mutex during modification  sharedBalance += amount;  // Compliant: Modification is now thread-safe  }  // Usage in a concurrent context  std::thread t1(makeTransaction, 500.00);  std::thread t2(makeTransaction, -200.00);  t1.join();  t2.join(); |

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

| **Principles(s):**   1. **Validate Input Data**:    * When it comes to banking software, this involves making sure that shared data can be accessed safely in threads, which is similar to checking the validity and correctness of transaction data. A guarantee of data integrity in concurrent processes is data race prevention. 2. **Practice Defense in Depth**:    * Similar to how several checks in financial transactions prevent fraud, using various ways to avoid data races (such as mutexes and atomic operations) adds levels of security for concurrent access. 3. **Adhere to the Principle of Least Privilege**:    * Each thread should have limited access to shared data, just like how banking employees are restricted to accessing only the data required for their specific roles. 4. **Keep It Simple**:    * The notion of plain and easy financial procedures is mirrored by the simplification of concurrent access sequences, which decreases the possibility of data races. 5. **Default Deny**:    * Just as a bank would reject transactions that don't match certain security requirements, by default, block concurrent access to shared resources unless specifically allowed. 6. **Use Effective Quality Assurance Techniques**:    * Just as auditing is essential to financial operations, so too must thorough testing be in place to identify and avoid data exchanges in software. 7. **Adopt a Secure Coding Standard**:    * The safe and dependable functioning of financial software depends on adhering to a coding standard that incorporates thread safety measures. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium to High | High | Level 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helgrind | 3.18.1 | DataRaceDetection | Thread safety in financial software relies on tools like Helgrind, a Valgrind tool for finding data races in C/C++ programs.  <https://valgrind.org/docs/manual/hg-manual.html> |
| ThreadSanitizer | 14.0.0 | ThreadSanitizer | With this tool's capacity to identify data races and other concurrency-related problems, banking system dependability and security may be enhanced.  <https://clang.llvm.org/docs/ThreadSanitizer.html> |
| CppCheck | 2.8 | Concurrency | Cppcheck checks for possible parallel problems, such as data races, which are very important to avoid in banking software.  <http://cppcheck.sourceforge.net/> |
| Visual Studio Code Analysis | 2022 or 2019 | ConcurrencyCheck | This tool is a part of Visual Studio and it helps find concurrency concerns in C++ code, which makes banking apps safer.  <https://visualstudio.microsoft.com/> |

### 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.

This document outlines Green Pace's present DevOps process and offers suggestions on how to improve it to fit with DevSecOps concepts. The goal is to add automated enforcement of the C/C++ coding standards. A graphic illustrating this approach is supplied. To guarantee conformity with the specified criteria, we will automate and improve each step of the cycle.

Evaluate and Strategize: During this crucial stage, it is important to enhance the current planning process by incorporating automated security risk assessments with a focus on ensuring compliance with standards such as STD-001-CPP for data type safety. It is recommended that the version control system include tools such as automatic code scanning in order to conduct a thorough examination of every push or merge request for the presence of possible breaches of standards.

Design: Enhance the design phase by incorporating automated design review tools. To make sure designs are safe by default, they should be set to enforce standards like STD-004-CPP, which are for secure communication. Developers will also be able to make better decisions earlier on if they use design linters that check for compliance with security standards.

Build: The build process has to be modified to include SCA tools. These tools automatically enforce standards like STD-008-CPP, which prevent SQL injections. Prior to inclusion in the build, these tools will check all third-party software and components to verify they meet the defined requirements.

Check and Double-Check: Implement automated test suites tailored to verify conformance with standards such as STD-005-CPP for managing exceptions. These should be run as part of the integration pipeline to guarantee that every build is secure and features as expected.

Transition and Health Check: Object lifecycle management standards, such as STD-009-CPP, should be automated into the deployment scripts during the transition and health check phase. The production environment will be prepared to safely handle the set up if this is done.

Monitor and Detect: Integrate real-time monitoring technologies that may identify breaches of standards, such as STD-007-CPP for thread safety, throughout the production process. Monitor and detect any violations. This guarantees continuous compliance and the capacity to promptly respond to any potential security breach or violation of regulations.

Maintain and Stabilize: Automate configuration management technologies that impose coding standards into maintenance routines. When these tools notice a change in configuration, they should automatically make the necessary adjustments so that the system continues to follow deadlock prevention requirements like STD-010-CPP.

Respond: Lastly, set up systems to deal with incidents automatically. To provide a quick and consistent reaction to any security issues, pre-configured playbooks are used that are in line with the coding standards, such as STD-002-CPP for memory management.

Green Pace will encourage a security-first mindset by updating the current DevOps process at each step to include these automatic safeguards. The integrity and dependability of the financial application depend on the DevSecOps methodology's provision of a software lifecycle that is consistent with C/C++ coding standards.

### 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 | Medium | High | 4 |
| STD-002-CPP | High | Likely | High | High | 5 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Possible | Medium | High | 5 |
| STD-005-CPP | Medium | Likely | Low | Medium | 3 |
| STD-006-CPP | High | Unlikely | Low | High | 5 |
| STD-007-CPP | High | Possible | High | High | 4 |
| STD-008-CPP | High | Possible | Low | High | 4 |
| STD-009-CPP | Medium | Possible | Medium | Medium | 3 |
| STD-010-CPP | High | Possible | Medium | High | 4 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 | Data is secured through encryption at rest, guaranteeing its safety while stored on a disk or in a computer's database. It makes sure that data can't be read without the right encryption keys, even if the storage media is compromised. Transaction logs, sensitive client information, and other data kept on servers or in cloud storage services are safeguarded for banking apps via encryption at rest. For this reason, methods such as AES (Advanced Encryption Standard) are often used. The policy applies to any and all data stored by the banking application infrastructure. Important for preventing data breaches and staying in line with financial standards. |
| Encryption at flight | Data is protected from unauthorized access when it is stored on a disk or in a database using encryption at rest. In the event that someone gains illegal access to the storage media, the data will still be indecipherable without the proper encryption keys. All data transfers over the network in a banking application are encrypted in flight. An example of this would be the use of protocols like Transport Layer Security to encrypt data when a consumer accesses their account details or starts a transaction. All data transfers over the network in a banking application are encrypted in flight. An example of this would be the use of protocols like Transport Layer Security to encrypt data when a consumer accesses their account details or starts a transaction. Any time information is sent across an unsecured network, like the internet, the policy is applicable. It is essential for any activity that involves sending or receiving data via the banking app, such as transactions or account inquiries. |
| Encryption in use | Data processing is made safer with encryption in use. Since decrypting data is a necessary step before using it, it can leave the data vulnerable, making this process more difficult. When handling transactions in memory, a financial app could use encryption. In order to directly manipulate encrypted data, techniques such as homomorphic encryption might be used. This policy is applicable at all times while the application continues to analyze data. When it comes to performing real-time transaction processing, detecting fraud, and conducting risk analysis, it is very necessary. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | A user's or entity's identification may be confirmed via the authentication procedure. It verifies the user's identity. Authentication is a crucial step in the banking application, ensuring that customers can securely access their accounts. Passwords, PINs, and other conventional security measures may coexist alongside more modern forms of authentication, such as hardware tokens, biometrics, and one-time passwords (OTPs). Whenever a user tries to log in or start a session, the authentication policy takes effect. Important in the fight against fraud and identity theft, it is essential to prevent unauthorized individuals from gaining access to your accounts.  It is essential to ensure the security of user logins by implementing a multi-factor authentication process. If a consumer inputs a password and then is asked to verify their identity using a biometric scan or a code received to their mobile device, that's one example. Upon creating a new user, an identification verification process will be initiated, which may include Know Your Customer (KYC) requirements. Access to financial services will only be granted when the user's identity has been validated against external systems. At the beginning of each session, this policy is enforced to make sure that no one other than authorized users may access the financial information or conduct transactions. |
| Authorization | Authorization is a crucial step that comes after authentication. It's all about deciding whether a user should be given the rights and privileges to access certain resources or carry out certain operations. When it comes to banking, authorization is the process that decides which account details a user may see and which transactions they can complete after authentication. A client could be able to see their balance, but they wouldn't be able to approve a substantial transaction without further verification. This policy applies once individuals have successfully authenticated and outlines their interactions with the banking application. Important for following access control rules and putting the concept of least privilege into practice, it applies to all permission-requiring actions.  To restrict database access to just those users who have the proper role-based permissions, authorization rules must be set up. Just because a customer service agent can change a client's contact details doesn't mean they can modify their financial information. Various degrees of access are defined by the policy according to the roles of the users. If bank management needs to conduct audits or authorize significant transactions, they may have more access than customers who can only see their own account data. Roles and the least privilege principle will govern who has access to what files. Examples of files that should only be available to authorized individuals are log files and audit logs. Following authentication, authorization is consistently enforced all through the user's application experience. This safeguards the integrity of the system and the security of data by ensuring that users may only execute authorized operations. |
| Accounting | When it comes to safety and regulation, Green Pace's banking app's Accounting part of the "Triple-A" architecture is vital. Critical for security, post-incident investigative assistance, and compliance with legal and regulatory record-keeping needs, it entails monitoring and documenting all user actions for auditing, analysis, and comprehending user behavior.  All login attempts, successful or not, are meticulously recorded in accounting along with timestamps, user IDs, and IP addresses from where they originated. In order to analyze user behavior patterns and identify attempts at illegal access, this is crucial. There is painstaking documentation of all database updates, including the kind of change, the author, and the date and time of the update. If you want to keep tabs on who did what and when, you need this audit trail. There is also documentation of the process of adding new users, which includes the specifics of the accounts created as well as information on the individuals who started and authorized the process. This satisfies regulatory standards and promotes openness in user management.  Additionally, the reason for the change, the authorized user, and the time of the change are all recorded whenever a user's access level is changed. Understanding and regulating access levels inside the program requires this documentation. The guideline also covers keeping tabs on who opens certain files. The user ID, the name of the file, the type of access (read, write, delete), and the time of access are all recorded every time a user opens a file.  The banking application ensures that accounting policies are consistently applied during the user's entire interaction. Important for complying with record-keeping standards set by law and regulation, keeping non-repudiation intact, and assisting investigations after a security incident. |

**\***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 | 11/15/2023 | Began adjustments to Standards | Hope Exnicious | Professor |
| 3.0 | 12/9/2023 | Successfully completed all coding standards and requirements | Hope Exnicious | Pending…. |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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