

C++ Code analysis in Visual Studio

Visual Studio provides tools to analyze and improve C++ code quality.

Analyze C and C++ code

OVERVIEW

[Code analysis for C/C++ overview](#)

QUICKSTART

[Code Analysis for C/C++ quick start](#)

TUTORIAL

[Analyze C/C++ code for defects walkthrough](#)

Code analysis reference

OVERVIEW

[C++ Core Guidelines ↗](#)

REFERENCE

[C++ Core Guidelines warnings](#)

[C++ code analysis warnings](#)

Use SAL annotations to reduce defects

OVERVIEW

[Understanding SAL](#)

GET STARTED

[SAL examples](#)

 **HOW-TO GUIDE**

[Annotate function parameters and return values](#)

[Annotate function behavior](#)

[Annotate structs and classes](#)

[Annotate locking behavior](#)

[Specify when and where annotations apply](#)

 **REFERENCE**

[SAL annotation intrinsics](#)

Code analysis for C/C++ overview

Article • 08/03/2021

The C/C++ Code Analysis tool provides information about possible defects in your C/C++ source code. Common coding errors reported by the tool include buffer overruns, uninitialized memory, null pointer dereferences, and memory and resource leaks. The tool can also run checks against the [C++ Core Guidelines](#).

IDE (integrated development environment) integration

The code analysis tool is fully integrated within the Visual Studio IDE.

During the build process, any warnings generated for the source code appear in the Error List. You can navigate to source code that caused the warning, and you can view additional information about the cause and possible solutions of the issue.

Command line support

You can also use the analysis tool from the command line, as shown in the following example:

```
Windows Command Prompt
C:\>cl /analyze Sample.cpp
```

Visual Studio 2017 version 15.7 and later: You can run the tool from the command line with any build system including CMake.

#pragma support

You can use the `#pragma` directive to treat warnings as errors; enable or disable warnings, and suppress warnings for individual lines of code. For more information, see [Pragma directives and the _pragma and _Pragma keywords](#).

Annotation support

Annotations improve the accuracy of the code analysis. Annotations provide additional information about pre- and post- conditions on function parameters and return types. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Run analysis tool as part of check-in policy

You might want to require that all source code check-ins satisfy certain policies. In particular, you want to make sure that analysis was run as a step of the most recent local build. For more information about enabling a code analysis check-in policy, see [Creating and Using Code Analysis Check-In Policies](#).

Team Build integration

You can use the integrated features of the build system to run code analysis tool as a step of the Azure DevOps build process. For more information, see [Azure Pipelines](#).

See also

- [Quickstart: Code analysis for C/C++](#)
- [Walkthrough: Analyze C/C++ Code for Defects](#)
- [Code Analysis for C/C++ Warnings](#)
- [Use the C++ Core Guidelines checkers](#)
- [C++ Core Guidelines Checker Reference](#)
- [Use rule sets to specify the C++ rules to run](#)
- [Analyze Driver Quality by Using Code Analysis Tools](#)
- [Code Analysis for Drivers Warnings](#)

Quickstart: Code analysis for C/C++

Article • 10/04/2022

You can improve the quality of your application by running code analysis regularly on C or C++ code. Code analysis can help you find common problems and violations of good programming practice. And, it finds defects that are difficult to discover through testing. Its warnings differ from compiler errors and warnings: It searches for specific code patterns that are known to cause problems. That is, code that's valid, but could still create issues, either for you or for other people who use your code.

Configure rule sets for a project

1. In **Solution Explorer**, open the shortcut menu for the project name and then choose **Properties**.
2. Optionally, in the **Configuration** and **Platform** lists, choose the build configuration and target platform.
3. To run code analysis every time the project is built using the selected configuration, select the **Enable Code Analysis on Build** check box. You can also run code analysis manually by opening the **Analyze** menu and then choosing **Run Code Analysis on ProjectName** or **Run Code Analysis on File**.
4. Choose the **rule set** that you want to use or create a **custom rule set**. If using LLVM/clang-cl, see [Using Clang-Tidy in Visual Studio](#) to configure Clang-Tidy analysis options.

Standard C/C++ rule sets

Visual Studio includes these standard sets of rules for native code:

Rule Set	Description
C++ Core Check Arithmetic Rules	These rules enforce checks related to arithmetic operations from the C++ Core Guidelines .
C++ Core Check Bounds Rules	These rules enforce the Bounds profile of the C++ Core Guidelines .

Rule Set	Description
C++ Core Check Class Rules	These rules enforce checks related to classes from the C++ Core Guidelines .
C++ Core Check Concurrency Rules	These rules enforce checks related to concurrency from the C++ Core Guidelines .
C++ Core Check Const Rules	These rules enforce const-related checks from the C++ Core Guidelines .
C++ Core Check Declaration Rules	These rules enforce checks related to declarations from the C++ Core Guidelines .
C++ Core Check Enum Rules	These rules enforce enum-related checks from the C++ Core Guidelines .
C++ Core Check Experimental Rules	These rules collect some experimental checks. Eventually, we expect these checks to be moved to other rulesets or removed completely.
C++ Core Check Function Rules	These rules enforce checks related to functions from the C++ Core Guidelines .
C++ Core Check GSL Rules	These rules enforce checks related to the Guidelines Support Library from the C++ Core Guidelines .
C++ Core Check Lifetime Rules	These rules enforce the Lifetime profile of the C++ Core Guidelines .
C++ Core Check Owner Pointer Rules	These rules enforce resource-management checks related to owner<T> from the C++ Core Guidelines .
C++ Core Check Raw Pointer Rules	These rules enforce resource-management checks related to raw pointers from the C++ Core Guidelines .

Rule Set	Description
C++ Core Check Rules	These rules enforce a subset of the checks from the C++ Core Guidelines . Use this ruleset to include all of the C++ Core Check rules except the Enum and Experimental rulesets.
C++ Core Check Shared Pointer Rules	These rules enforce resource-management checks related to types with shared pointer semantics from the C++ Core Guidelines .
C++ Core Check STL Rules	These rules enforce checks related to the C++ Standard Library from the C++ Core Guidelines .
C++ Core Check Style Rules	These rules enforce checks related to use of expressions and statements from the C++ Core Guidelines .
C++ Core Check Type Rules	These rules enforce the Type profile of the C++ Core Guidelines .
C++ Core Check Unique Pointer Rules	These rules enforce resource-management checks related to types with unique pointer semantics from the C++ Core Guidelines .
Concurrency Check Rules	These rules enforce a set of Win32 concurrency pattern checks in C++.
Concurrency Rules	Adds concurrency rules from C++ Core Guidelines to Concurrency Check Rules .
Microsoft Native Minimum Rules	These rules focus on the most critical problems in your native code, including potential security holes and application crashes. We recommend you include this rule set in any custom rule set you create for your native projects.
Microsoft Native Recommended Rules	These rules focus on the most critical and common problems in your native code. These problems include potential security holes and application crashes. We recommend you include this rule set in any custom rule set you create for your native projects. This ruleset is designed to work with Visual Studio Professional edition and higher. It includes all the rules in Microsoft Native Minimum Rules .

Visual Studio includes these standard sets of rules for managed code:

Rule Set	Description
----------	-------------

Rule Set	Description
Microsoft Basic Correctness Rules	These rules focus on logic errors and common mistakes made in the usage of framework APIs. Include this rule set to expand on the list of warnings reported by the minimum recommended rules.
Microsoft Basic Design Guideline Rules	These rules focus on enforcing best practices to make your code easy to understand and use. Include this rule set if your project includes library code or if you want to enforce best practices for easily maintainable code.
Microsoft Extended Correctness Rules	These rules expand on the basic correctness rules to maximize the reported logic and framework usage errors. Extra emphasis is placed on specific scenarios such as COM interop and mobile applications. Consider including this rule set if one of these scenarios applies to your project or to find more problems in your project.
Microsoft Extended Design Guideline Rules	These rules expand on the basic design guideline rules to maximize the reported usability and maintainability issues. Extra emphasis is placed on naming guidelines. Consider including this rule set if your project includes library code or if you want to enforce the highest standards for writing maintainable code.
Microsoft Globalization Rules	These rules focus on problems that prevent data in your application from displaying correctly when used in different languages, locales, and cultures. Include this rule set if your application is localized and/or globalized.
Microsoft Managed Minimum Rules	These rules focus on the most critical problems in your code for which Code Analysis is the most accurate. These rules are small in number and they're intended only to run in limited Visual Studio editions. Use MinimumRecommendedRules.ruleset with other Visual Studio editions.
Microsoft Managed Recommended Rules	These rules focus on the most critical problems in your code. These problems include potential security holes, application crashes, and other important logic and design errors. We recommend you include this rule set in any custom rule set you create for your projects.
Microsoft Mixed (C++ /CLR) Minimum Rules	These rules focus on the most critical problems in your C++ projects that support the Common Language Runtime. These problems include potential security holes, application crashes, and other important logic and design errors. We recommend you include this rule set in any custom rule set you create for your C++ projects that support the Common Language Runtime.
Microsoft Mixed (C++ /CLR) Recommended Rules	These rules focus on the most common and critical problems in your C++ projects that support the Common Language Runtime. These problems include potential security holes, application crashes, and other important logic and design errors. This ruleset is designed for use in the Visual Studio Professional edition and higher.

Rule Set	Description
Microsoft Security Rules	This rule set contains all Microsoft security rules. Include this rule set to maximize the number of potential security issues that are reported.

To include every rule:

Rule Set	Description
Microsoft All Rules	This rule set contains all rules. Running this rule set may result in a large number of warnings being reported. Use this rule set to get a comprehensive picture of all issues in your code. It can help you decide which of the more focused rule sets are most appropriate to run for your projects.

Run code analysis

On the **Code Analysis** page of the Project Properties dialog, you can configure code analysis to run each time you build your project. You can also run code analysis manually.

To run code analysis on a solution:

- In the **Build** menu, choose **Run Code Analysis on Solution**.

To run code analysis on a project:

1. In the Solution Explorer, select the name of the project.
2. In the **Build** menu, choose **Run Code Analysis on Project Name**.

To run code analysis on a file:

1. In the Solution Explorer, select the name of the file.
2. In the **Build** menu, choose **Run Code Analysis on File** or press **Ctrl+Shift+Alt+F7**.

The project or solution is compiled and code analysis runs. Results appear in the Error List window.

Analyze and resolve code analysis warnings

The Error List window lists the code analysis warnings found. The results are displayed in a table. If more information is available about a particular warning, the first column contains an expansion control. Choose it to expand the display for additional

information about the issue. When possible, code analysis displays the line numbers and analysis logic that led to the warning.

For detailed information about the warning, including possible solutions to the issue, choose the warning ID in the Code column to display its corresponding online help article.

Double-click a warning to move the cursor to the line of code that caused the warning in the Visual Studio code editor. Or, press Enter on the selected warning.

After you understand the problem, you can resolve it in your code. Then, rerun code analysis to make sure that the warning no longer appears in the Error List.

Create work items for code analysis warnings

You can use the work item tracking feature to log bugs from within Visual Studio. To use this feature, you must connect to an instance of Azure DevOps Server (formerly, Team Foundation Server).

To create a work item for one or more C/C++ code warnings

1. In the Error List, expand and select the warnings
2. On the shortcut menu for the warnings, choose **Create Work Item**, and then choose the work item type.
3. Visual Studio creates a single work item for the selected warnings and displays the work item in a document window of the IDE.
4. Add any additional information, and then choose **Save Work Item**.

Search and filter code analysis results

You can search long lists of warning messages and you can filter warnings in multi-project solutions.

- **To filter warnings by title or warning ID:** Enter the keyword in the Search Error List box.
- **To filter warnings by severity:** By default, code analysis messages are assigned a severity of **Warning**. You can assign the severity of one or more messages as **Error** in a custom rule set. On the **Severity** column of the **Error List**, choose the drop-

down arrow and then the filter icon. Choose **Warning** or **Error** to display only the messages that are assigned the respective severity. Choose **Select All** to display all messages.

See also

- [Code analysis for C/C++](#)

Build reliable and secure C++ programs

Article • 09/28/2023

The United States government publication [NISTIR 8397: Guidelines on Minimum Standards for Developer Verification of Software](#) contains excellent guidance on how to build reliable and secure software in any programming language.

This document follows the same structure as NISTIR 8397. Each section:

- summarizes how to use Microsoft developer products for C++ and other languages to meet that section's security needs, and
- provides guidance to get the most value in each area.

2.1 Threat modeling

Summary

Threat modeling is a valuable process, especially when applied in a way that scales to meet your development needs and that reduces noise.

Recommendations

Threat modeling should be one part of your dynamic Security Development Lifecycle (SDL). We suggest that for your product as a whole, for a specific feature, or for a major design or implementation change:

- Have a solid, dynamic SDL that allows for early engagement with developer teams and rightsizing of approach.
- Apply threat modeling in a targeted way. Apply threat modeling to all features, but tactically start with exposed, complex or critical features. Do apply it regularly instead as part of a top-down product review.
- Apply threat modeling early (as with all security requirements), when there's still opportunity to change the design. Also, threat models serve as an input to other processes, such as attack surface reduction or designing for security. Threat models that are created later are at best "surveys" for pen test (penetration testing) or areas that need security testing such as fuzzing. After you create a baseline threat model, plan to continue iterating on it as the attack surface changes.
- Use asset inventory and compliance to appropriately track what makes up a product, and track security artifacts (including threat models) along with the assets they apply to. This approach enables better automated risk assessment and focusing of security efforts on the specific components or features that change.

- In Azure, the Microsoft Threat Modeling Tool was updated in 2022 for Azure development. For more information, see [Microsoft Threat Modeling Tool overview - Azure](#)

Supporting factors and practices

To properly apply threat modeling and avoid underuse/overuse, we have found that the following core concepts need to be addressed first.

Development approach

First, understand the team's development approach. For teams with agile development workflows that push dozens of changes to production daily, it's not practical or reasonable to require a threat model update for every functional change. Instead, from the start when writing a feature's functional requirements, consider including a security requirements questionnaire. The questionnaire should focus on specific questions about the feature to determine what future aspects of your SDL apply. For example:

- Does the feature make a major change in design of how we provide customer isolation in a multi-tenant environment? If so, consider performing a full threat model.
- Does a new feature allow file uploads? If so, perhaps what's more appropriate is a web application security assessment.
- Is this change primarily just a functional UI change? If so, perhaps nothing is needed beyond your traditional automated tooling.

The security questionnaire results inform which SDL techniques to tie to which unit of development. It also informs development partners of the feature's SDL timelines, so they can collaborate at the right times.

Product inventory

Second, maintain a strong asset inventory of the products you're tasked with assessing. Products are growing in complexity. It's common to write software for connected devices that have:

- sensors (such as passenger rail and vehicles),
- bus-based networks that talk to other components in the vehicle (such as CANBUS or PROFIBUS),
- wireless/cellular/Bluetooth for communication with customer devices and cloud back ends,
- machine learning in the cloud feeding back into the device or a fleet management application,
- and more.

In such complex products, threat modeling is critical. Having a strong asset inventory lets you view the entire product stack to see the complete picture, and to see the key places that need to be evaluated for how a new or changed feature impacts product security.

Granularity and integration

Establish systems to measure compliance using clear metrics.

- Regularly measure compliance for feature level development. Feature compliance generally should be measured with higher frequency and smaller granularity, sometimes even on the developer's system or at code commit/merge time.
- Periodically evaluate security for the broader product in which a feature or component is being consumed. Broader evaluations typically are done with lower frequency and broader granularity, such as at module or system testing time.

Scale

Keep a proper asset inventory system that captures and preserves security artifacts and the output of threat model reviews. Having a clear inventory lets you evaluate review outputs for patterns, and make intelligent decisions on how to refine the product security program regularly.

Try to combine requirements-phase security questionnaires, threat modeling results, security assessment results, and results from automated tools. Combining them enables you to automate a viewpoint of relative risk of a given product, ideally as a "dashboard," to inform your security teams what to focus on to get the best value out of the threat modeling.

2.2 Automated testing

Summary

Automated tests are an important way to ensure the quality and safety of your code. They're an integral piece in supporting other areas mentioned in this document, such as Threat Modeling. When paired with other secure coding practices, they help to protect against bugs and vulnerabilities being introduced into the codebase.

Key attributes

Tests should be reliable, consistent, and isolated. These tests should cover as much of the code as possible. All new features and bug fixes should have corresponding tests to ensure the long-term security and reliability of the code when possible. Run automated

tests regularly and in as many environments as possible, to ensure they get run and that they cover all areas:

- The first place they should run is on the machine that is making the changes. Running tests is most easily done within the IDE that is being used for editing, or as a script on the command line as the developer makes the changes.
- The next place they should run is as part of the pull request commit/merge process.
- The last place to run tests is as part of a Continuous Integration and Continuous Deployment (CI/CD) pipeline, or on your release candidate builds.

The scope of the tests should increase at each step, with the last step providing full coverage for anything the other steps may miss.

Continuous use and maintenance

Test reliability is an important part of maintaining the effectiveness of the testing suite. Test failures should be assigned and investigated, with potential security issues getting high priority and getting updated within a prompt and predetermined timeframe. Ignoring test failures shouldn't be a common practice, but should require strong justification and approval. Test failures due to issues within the test suite itself should be treated the same as other failures, to prevent a lapse in coverage in which product issues could be missed.

Kinds of tests, especially unit tests

There are several types of automated tests, and while not all are applicable to all applications, a good test suite contains a selection of several different types. Code Based Test Cases such as unit tests are the most common and most integral, being applicable to all applications and intentionally covering as many code paths as possible for correctness. These tests should be small, quick, and not affect the state of the machine, so that the full suite of tests can be run quickly and often. If possible, run tests on many machines that have different hardware setups to catch problems that aren't reproducible on a single type of machine.

Visual Studio

Visual Studio Test Explorer natively supports many of the most popular C++ testing frameworks, and has options to install extensions for more frameworks. This flexibility is helpful for running a subset of tests covering the code you're working on, and makes it easy to debug test failures as they arise. Visual Studio also makes it easy to set up new test suites for existing projects, and provides helpful tools such as CodeLens to make it easier to manage these tests. For more information about writing, running, and

managing C/C++ tests with Visual Studio, see [Write unit tests for C/C++ - Visual Studio \(Windows\)](#).

In Azure and GitHub CI/CD

Tests that do deeper verification and take longer to run, such as static analysis, component detection, and so on, are good candidates for pull request testing or continuous integration testing. Azure DevOps and GitHub [Actions](#) make it easy to run validations automatically and block code checkins if a validation fails. Automated enforcement helps ensure that all code being checked in is secure based on these more rigorous checks being run. Azure Pipelines and Azure DevOps Build Validation are described here:

- [Git branch policies and settings - Azure Repos](#)
- [Defining the mergeability of pull requests | GitHub Docs](#)

2.3 Code-based, or static, analysis

Summary Static code/binary analysis should be enabled by default, to be secure by default. Static analysis analyzes a program for required safety and security policies at the time it's being built, not at execution time when an exploit can occur on the customer's machine. Static analysis can analyze the program in source code form or in compiled executable form.

Recommendations Microsoft recommends that you:

- Enable static analysis for all C++ programs, for both the input source code (before compilation) and the executable binaries (after compilation). "Enable" might mean to run analysis during each build on the developer's machine, or as a separate build to inspect the code later or as a checkin requirement.
- Incorporate static analysis into CI pipelines as a form of testing.
- Static analysis by definition comes with false positives, and be prepared to incorporate that fact into your quality feedback loop. Be quick to enable all low-false-positive warnings up front. Then be proactive to gradually increase the number of rules for which your code base compiles warning-clean as you regularly add more rules that flag important bugs at the expense of gradually higher false positives (initially, before the code base has been cleaned for those rules too).
- Always use the latest supported versions of Visual Studio, and set up your engineering environment to quickly consume the latest patch releases as soon as they become available, without delaying to the next development stage/cycle.

Key tools Be aware of and use the following:

- [Code analysis documentation - C++ and .NET](#)
- [/analyze - Visual C++ compiler](#)
- [/W4 and /WX - Visual C++ compiler](#)
- [Use the C++ Core Guidelines Checkers](#)
- [CodeQL | GitHub ↗](#)
- [BinSkim user guide | GitHub ↗](#)
- See also (Windows only): [SAL annotations](#)

Notes:

- `/analyze` enables static analysis of C++ code at compile time to identify critical security and reliability code vulnerabilities. It should be enabled throughout a C++ program's entire development timeline. Start by enabling at least the "Microsoft Native Recommended" by default as a minimum baseline. Then consult the documentation for how to specify more rules, especially the C++ Core Guidelines rules, as required by your engineering policies. The source code Static Analysis capability is available in both the Visual C++ IDE and in the command-line Build Tools.
- `/W4` and `/WX` should be enabled wherever possible, to ensure you compile your code cleanly at high warning levels (`w4`) and treat warnings as errors that must be fixed (`wx`). These options enable finding uninitialized data errors that other static analysis tools can't check, because the errors only become visible after the compiler back-end performs interprocedural analysis and inlining.
- BinSkim binary analysis ensures that projects enable a broad range of security features. BinSkim generates PDBs and other outputs that make it easier to verify chain-of-custody and to respond efficiently to security issues. Microsoft recommends running the BinSkim tool to analyze all executable binaries (`.sys`, `.dll` or `.exe`) produced for or consumed by your programs. The BinSkim User Guide includes a list of supported security standards. Microsoft recommends that you fix all issues reported as "errors" by the BinSkim tool. Issues reported as "warnings" should be evaluated selectively, because resolving them can have performance implications or may not be necessary.

In Azure and GitHub CI/CD Microsoft recommends always enabling source code and binary static analysis in release CI/CD scenarios. Run source analysis immediately on the local developer's machine, or at least for every commit or pull request, to catch source bugs as early as possible and minimize overall costs. Binary level bugs tend to be introduced more slowly, so it may be sufficient to run binary analysis in less frequent prerelease CI/CD scenarios (such as nightly or weekly builds).

2.4 Review for hardcoded secrets

Summary

Don't hardcode secrets within software. You can find and remove secrets from the source code efficiently by using reliable tools that can scan your entire source code base. Once you find secrets, move them to a safe place following the guideline for secure storage and use of secrets.

Problem

"Secrets" means entities that establish identity and provide access to resources, or that are used to sign or encrypt sensitive data. Examples include passwords, storage keys, connection strings, and private keys. It's tempting to keep secrets in the software product so they can be readily obtained when needed by the software. However, these hardcoded secrets can lead to severe or catastrophic security incidents as they're easily discovered and can be used to compromise your service and data.

Prevention

Secrets hardcoded in source code (as plain text or encrypted blob) are a security vulnerability. Here are general guidelines on how to avoid secrets in the source code:

- Use a precheckin tool to scan and catch potential hardcoded secrets in code prior submitting to source control.
- Don't put clear text credentials in source code or configuration files.
- Don't store clear text credentials in SharePoint, OneNote, file shares, and so on. Or share them via email, IM, and so on.
- Don't encrypt a secret with an easily discoverable decryption key. For example, don't store a PFX file along with a file that contains its password.
- Don't encrypt a secret with a weak decryption. For example, don't encrypt a PFX file with a weak or common password.
- Avoid putting encrypted credentials in source code. Instead, use placeholders in your source, and let your deployment system replace them with secrets from approved stores.
- Apply the same principles to secrets in environments such as testing, staging, and so on, as you do in production deployments. Adversaries often target nonproduction systems as they're less well managed, then use them to pivot into production.
- Don't share secrets between deployments (for example, testing, staging, production).

While not directly related to hardcoded secrets, also remember securing secrets for your test, development, and production:

- Rotate your secrets periodically and whenever they may have been exposed. Having a demonstrated ability to rotate/redeploy secrets is evidence of a secure system. More notably, the absence of this capability is even stronger evidence of an inevitable vulnerability.
- Don't give in to the common developer rationale that "my test credentials don't create risk." In practice, they nearly always do.
- Consider moving away from secrets (for example, passwords, bearer keys) entirely in preference of RBAC/identity-driven solutions as a good engineering solution that can sidestep secret mismanagement entirely.

Detection

Legacy components of your product may contain hidden hardcoded secrets in their source code. Sometimes secrets from developers' desktop machines can creep into remote branch and merge into the release branch, leaking secrets unintentionally. To discover secrets that might be hiding in your source code, you can use tools that can scan your code for hardcoded secrets:

- [Microsoft sarif-pattern-matcher tool ↗](#)

Remediation

When credentials are found in your source code, the immediate urgent need is to invalidate the exposed key and perform a risk analysis based on exposure. Even if your system needs to stay running, you can enable a secret manager for remediation using these steps:

1. If remediation allows switching over to managed identities, or requires dropping in a secret manager such as Azure Key Vault (AKV), do that first. Then redeploy with the updated identity or key.
2. Invalidate the exposed secret.
3. Perform auditing/risk assessment of potential damage due to compromise.

To safeguard cryptographic keys and other secrets used by cloud apps and services, use [Azure Key Vault ↗](#) with an appropriate access policy.

If an exposure compromises certain customer data/PII, it may require other compliance/reporting requirements.

Remove the now-invalidated secrets from your source code, and replace them with alternative methods that don't expose the secrets directly in your source code. Look for

opportunities to eliminate secrets where possible by using tools like Azure AD. You can update your authentication methods to take advantage of managed identities via Azure Active Directory (AAD). Only use approved stores to store and manage secrets such as Azure Key Vault (AKV). For more information, see:

- [Azure AD: Implementing autorotation using Azure Active Directory \(AAD\)](#) ↗

Azure DevOps (AzDO)

AzDO users can scan their code through GitHub Advanced Security for Azure DevOps (GHAzDO). GHAzDO also allows users to prevent secret exposures by enabling Push Protection on their repositories, catching potential exposures before they're ever leaked. For more information on how to detect hardcoded secrets in code in Azure DevOps, see *Secret Scanning for Github Advanced Security for Azure DevOps* in each of the following links:

- [GitHub advanced security for Azure DevOps](#) ↗
- [Secret Scanning for GitHub Advanced Security for Azure DevOps](#)
- [Microsoft Defender for DevOps Preview](#) ↗

In GitHub

Secret scanning is available on GitHub.com in two forms:

- *Secret scanning alerts for partners.* Runs automatically on all public repositories. Any strings that match patterns that were provided by secret scanning partners are reported directly to the relevant partner.
- *Secret scanning alerts for users.* You can enable and configure extra scanning for repositories owned by organizations that use GitHub Enterprise Cloud and have a license for GitHub Advanced Security. These tools also support private and internal repositories.

GitHub provides known patterns of secrets for partners and users that can be configured to meet your needs. For more information, please see:

- [Secret scanning patterns](#) ↗
- [About secret scanning in GitHub](#) ↗

Note

GitHub Advanced Security for Azure DevOps brings the same secret scanning, dependency scanning and CodeQL code scanning solutions already available for

GitHub users and natively integrates them into Azure DevOps to protect your Azure Repos and Pipelines.

Additional resources

- [Credential Scanning | Microsoft Code With Engineering Playbook ↗](#).
- [detect-secrets: Credential scanning tool | GitHub ↗](#) - an aptly named module for detecting secrets within a code base.
- [Running detect-secrets in Azure DevOps pipelines ↗](#).
- [Git-secrets | GitHub awslabs ↗](#) - prevents you from committing passwords and other sensitive information to a git repository.
- [Secrets Management | Microsoft Code with Engineering Playbook ↗](#) - provides general guidelines on how secrets should be managed.

2.5 Run with language- and OS-provided checks and protection

Summary

Binary hardening is done by applying compile-time security controls. These include mitigations that:

- prevent exploitable vulnerabilities in code,
- enable runtime detections that trigger security defenses on exploitation, and
- enable data production and archiving to help limit the damage caused by security incidents.

Binary consumers must opt into Windows security features to gain the full benefit of hardening.

Microsoft provides a set of facilities specific to C++ projects to help developers write and ship safer and more secure code. C++ developers should also adhere to security standards common to languages that generate executable code. Microsoft maintains BinSkim, a public OSS binary checker that helps enforce use of many protections described in this section. For more information about BinSkim, see [Binskim user guide | GitHub ↗](#)

Binary-level controls differ according to where they're applied in the engineering process. You should distinguish among compiler and linker options that: are strictly compile time, alter code generation with run-time overhead, and alter code generation to achieve compatibility with OS protections.

Developer settings should prefer to enable as much static analysis as possible, enable production of private data to accelerate debugging, and so on. Release builds should be tuned to an appropriate combination of security, performance, and other code generation concerns. Release processes must be configured to properly generate and manage public vs. privately consumed build data (for example, public vs. private symbols).

Stay current: Always use up-to-date compilers and tools

Compile all code with current toolsets to benefit from up-to-date language support, static analysis, code generation and security controls. Because compilers impact every generated component, the potential for regression on tool update is relatively high. Using outdated compilers creates a particular risk for corrective action while responding to a security incident, because teams may not have enough time to upgrade compilers. Microsoft recommends that teams develop the facility to regularly refresh and test compiler updates.

Use secure development methods, language versions, frameworks/APIs

Code should utilize development methodologies, language versions, framework, APIs, and so on, that minimize risk by fostering safety and simplicity in C++, including:

- See [C++ Core Guidelines' Guideline Support Library \(GSL\)](#) for guidance to write modern, safe, and consistent C++ code that follows best practices and avoids common pitfalls.
- See [Microsoft GSL implementation](#) for functions and types that the C++ Core Guidelines suggest you use.
- Resource-safe C++ containers, C runtime library (CRT) memory overflow protections: Prefer `std::vector` and `std::string`, which are resource-safe. If you must use C data, use the [secure versions of CRT functions](#), which are designed to help prevent memory corruption due to buffer misuse and undefined language behaviors.
- The [SafeInt library](#) protects against integer overflow in mathematical and comparison operations.

Consume secure dependencies

Binaries shouldn't link to insecure libraries and dependencies. Development teams should track all external dependencies and resolve CVEs/identified security vulnerabilities in these components by updating to more secure versions when subject to those vulnerabilities.

Maximize code provenance guarantees and efficiency of security response

Compilation should enable strong code provenance guarantees to help detect and prevent introduction of backdoors and other malicious code. The resulting data, also critical to debugging and investigation, should be archived for all software releases to drive efficient security response if they're compromised. The following compiler switches generate information that is critical to a security response:

- [/ZH:SHA_SHA256 in Visual C++](#) - Ensures that a cryptographically secure algorithm is used to generate all PDB source file hashes.
- [/Zi, /ZI \(Debug Information Format\) in Visual C++](#) - In addition to publishing stripped symbols for collecting crash data and other public use scenarios, ensure that builds produce and archive private PDBs for all released binaries. Binary analysis tools require full symbols to verify whether many security mitigations were enabled at compile-time. Private symbols are critical in security response, and lower debugging and investigation costs when engineers are racing to assess and limit damage when an exploit happens.
- [/SOURCELINK in Visual C++ Linker - Include Sourcelink file in PDB](#): Source link is a language- and source-control agnostic system providing source debugging for binaries. Source debugging greatly increases the efficiency the range of prerelease security validations and post-release incident response.

Enable compiler errors to prevent issues at code authoring time

Compilation should enable security-relevant compiler checks as breaking errors, for example:

- [/sdl in Visual C++ - Enable additional security checks](#) ↗ elevates many security-relevant warnings into errors and enables advanced secure code-generation features.
- [BinSkim BA2007.EnableCriticalCompilerWarnings | GitHub](#) ↗ maintains a list of Microsoft-recommended C/C++ compiler warnings that should always be enabled and elevated to errors.

Mark binaries as compatible with OS runtime security mitigations

Compiler and linker settings should opt into code generation features that detect and mitigate malicious code execution, including:

- Stack corruption prevention
 - [/SAFESEH - Image has safe exception handlers](#) ↗ - Produces a table of the image's safe exception handlers for x86 binaries.
 - [/GS - Buffer Security Check](#) ↗ - Detects some buffer overruns that overwrite return addresses, exception handler addresses or certain types of parameters.
- Position independent code execution

- [/DYNAMICBASE - Use Address Space Layout Randomization](#) - Generates executable images that can be randomly rebased at load time.
- [/HIGHENTROPVA and /LARGEADDRESSAWARE - Support 64-bit ASLR, and Handle large addresses](#) - Enables use of entire 64-bit address space for image rebasing.
- Code flow integrity
 - [/guard:cf - Enable Control Flow Guard](#) - Inserts runtime verifications for indirect call targets.
 - [/CETCOMPAT - CET shadow stack compatible](#) - Marks an executable image as compatible with Microsoft's implementation of Intel's [Control-flow Enforcement Technology \(CET\)](#) Shadow Stack feature.
 - [/guard:ehcont - Enable EH continuation metadata](#) - Generates a list of safe relative virtual addresses (RVA) of all exception handling continuation targets.
- Data execution prevention
 - [/NXCOMPAT - Compatible with Data Execution Prevention](#) - Marks a 32-bit executable image as compatible with the [Windows Data Execution Prevention \(DEP\)](#) feature. 64-bit builds are compatible with DEP by default.)

Prevent sensitive information disclosure

Compiler settings should opt into sensitive information discovery prevention. In recent years, researchers have uncovered unintended information leakage that originates with hardware features such as speculative execution.

At the software level, confidential data may be transmitted to attackers if unexpectedly leaked. Failure to zero-initialize buffers and other buffer misuse may leak private confidential data to attackers that call trusted API. This class of problem best handled by enabling extra static analysis and using secure resource containers as described previously.

- [/Qspectre - Mitigate speculative execution side-channel attacks](#) - Inserts barrier instructions that help prevent the disclosure of sensitive data produced by speculative execution. These mitigations should be enabled for code that stores sensitive data in memory and operates across a trust boundary. Microsoft always recommends measuring performance impact against appropriate benchmarks when enabling Spectre-mitigations due to the possibility of introducing runtime checks in performance-critical blocks or loops. These code paths can disable mitigations via the [spectre\(nomitigation\)](#) `declspec` modifier. Projects that enable `/Qspectre`` should also link to libraries that are also compiled with these mitigations, including the Microsoft runtime libraries.

2.6 Black box test cases

Summary

Black box tests don't rely on knowing the tested component's inner workings. Black box tests are designed to test the end-to-end functionality of the features in the product at any layer or level. Black box tests can be functional tests, UI tests, performance tests, and integration tests. Black box tests are valuable for measuring general reliability and functional correctness, and ensuring that the product behaves as expected.

Relation to other sections

These types of requirement-based tests are useful for validating the assumptions made in the Threat Model and covering potential threats as brought up in that section. These tests are useful for testing the integration between separate components of the product, especially ones that are across trust boundaries as described in the threat model. Black box test cases are also useful for testing all kinds of edge cases for user input validation. Testing known edge cases and error cases are both useful. Fuzzing is also useful to test less obvious cases.

Automation and regression

Run these tests regularly, and compare the results to previous runs to catch breaking changes or performance regressions. Also, running these tests on many different machines and installation setups can help cover any issues that may arise from different architectures or setup changes.

Crash dumps

These tests help find issues with reliability, being able to test many different scenarios that may run into crashes, hangs, deadlocks, and so on. By collecting crash dumps as part of test failures, you can import the dumps directly into Visual Studio to further investigate what parts of the code are hitting these issues. If you run functional tests from within Visual Studio, you can easily replicate and debug failures by seeing exactly where inside the black box the test fails, and you can test fixes quickly.

To get started with debugging tests, see [Debug unit tests with Test Explorer - Visual Studio \(Windows\)](#)

In Azure

Azure DevOps can also help manage and validate these tests with the use of Test Plans. These tests can be used to ensure approval with manual validation, and to run

automated tests associated with product requirements. More information on Azure Test Plans and using them to run automated testing can be found here:

- [What is Azure Test Plans? Manual, exploratory, and automated test tools. - Azure Test Plans](#)
- [Run automated tests from test plans - Azure Test Plans](#)

2.7 Code-based test cases

Summary

Code-based test cases are an integral part of maintaining the security and reliability of your product. These tests should be small and fast and shouldn't have an impact on each other so they can be run in parallel. Code-based tests are easy for developers to run locally on their development machine anytime they make changes to the code without worrying about slowing down their development cycle.

Types, and relation to other sections

Common types of code-based test cases include:

- unit tests,
- parameterized tests to cover functions with multiple input types,
- component tests to keep each test component separate, and
- mock testing to validate parts of the code that communicate with other services, without expanding the scope of the test to include those services themselves.

These tests are based on the internal code that is written, whereas black box tests are based on the external functional requirements of the product.

Goal

Through these tests, the goal is to achieve a high level of test coverage over your code. You should actively track this coverage and where gaps exist. As you add more tests that exercise more code paths, the overall confidence in your code's security and reliability increases.

Visual Studio

The test explorer tools in Visual Studio make it easy to run these tests frequently and get feedback on pass/fail rates and failure locations quickly. Many of the test frameworks also support CodeLens features to see the test status at the location of the test itself, making adding and maintaining the suite of tests easier. The Test Explorer also makes

managing these tests easy, allowing for test groups, custom test playlists, filtering, sorting, searching, and more.

For more information, see:

- [Unit testing fundamentals - Visual Studio \(Windows\)](#) - an introduction and overview
- [Run unit tests with Test Explorer - Visual Studio \(Windows\)](#) - a deeper look at what's available to help manage the potentially large set of unit tests with the Test Explorer

Visual Studio also comes with tools for tracking the code coverage. These tools enable you to ensure that code changes you make are covered by existing tests, or to add new tests to cover new and untested code paths. The tools also show the code coverage percentage to ensure it's maintained above a target level for confidence in overall code quality.

For information about these tools, see [Code coverage testing - Visual Studio \(Windows\)](#)

In Azure

Azure DevOps can also help in tracking code coverage results for the whole product as part of the build pipeline process. For more information, see [Review code coverage - Azure Pipelines](#).

2.8 Historical test cases

Summary

Historical test cases, also known as regression test cases, prevent old issues from resurfacing again and increase the overall test coverage of the product. You should ensure that when a bug is fixed the project also adds a corresponding test case. Over time, as fixes are made, the overall robustness of the testing suite will keep improving, giving better assurances of reliability and security.

Key qualities, and relation to other sections

Since they test for bug regressions, these tests should be quick and easy to run, so they can run alongside the [Code Based Test Cases] and contribute to the overall code coverage of the product. Along with this, using real examples from customers to inspire new test cases is a great way to improve coverage and quality of tests.

Visual Studio

Visual Studio lets you easily add tests to the suite while making the changes to fix the bug, and quickly run the tests and code coverage to ensure all new cases get considered. Referencing the bug ID from your issue tracking system in your code where you write the test is a good way to connect regression tests to the corresponding issues. Prefer to use Azure DevOps boards and test plans together with Visual Studio:

- to associate tests, test cases, and issues; and
- to track of all aspects of an issue and its corresponding tests.

For more information, see:

- [Associate automated tests with test cases - Azure Test Plans](#)
- [Link work items to other objects - Azure DevOps](#)

Eventually, integrating these tests into the unit testing area that is supposed to cover the code section helps keep the test suite organized and easier to manage. You can use the Test Explorer's test grouping to effectively track the tests that belong together. For more information, see [Run unit tests with Test Explorer - Visual Studio \(Windows\)](#)

2.9 Fuzzing

Summary Fuzzing (also known as fuzz testing) is an automated software testing technique that involves providing invalid, unexpected, or random data as input to a program. The program is then monitored for exceptions such as crashes, failing built-in or compiler injected code assertions and potential memory leaks.

Guidance

Use fuzzing on all software that may process untrusted inputs that an attacker could control. If you're building a new application and its associated test suite, include fuzzing for key modules as early as possible. Running fuzzing for the first time on a piece of software nearly always uncovers actual vulnerabilities that were previously unknown. Once you start fuzzing, never stop.

Relation to other sections

When fuzzing reports a failure, it always naturally provides a reproducible test case that demonstrates the bug. This test case can be reproduced, resolved, and then added to the Historical Test Cases.

When using both sanitizers such as [Address Sanitizer \(ASan\)](#) and fuzzing:

- First run your normal tests with sanitizers enabled to see if there are issues, then once the code is sanitizer-clean start fuzzing.

- For C or C++, there are compilers that automate injection of runtime assertions and meta-data that enable ASan. When compiled for ASan, the resulting binaries link with a runtime library that can precisely diagnose [15+ categories of memory safety errors](#) with zero false positives. For C or C++ when you have source, use [LibFuzzer](#), which requires ASan to be enabled first.
- For libraries written in Java, C#, Python, Rust, and so on, use the [AFL++ framework](#).

Key qualities

- Fuzzing finds vulnerabilities often missed by static program analysis, exhaustive feature testing and manual code inspection.
- Fuzzing is an effective way to find security and reliability bugs in software, so much so that the [Microsoft Security Development Lifecycle](#) requires fuzzing at every untrusted interface of every product (see also Threat Modeling).
- Always use fuzzing for software that may process untrusted inputs.
- Fuzzing is effective for standalone applications with large data parsers.

Azure and GitHub CI/CD

Modify your build(s) to support continuous creation of executables that use LibFuzzer or AFL++. You can add extra computing resources required for fuzzing at services like OSS-Fuzz or OneFuzz.

2.10 Web Application Scanning

Summary

Within the scope of Microsoft Visual C++ on Windows, Microsoft recommends:

- Prefer TypeScript, JavaScript, and ASP.NET for web applications.
- Don't write web extensions in C++. Microsoft has deprecated ActiveX.
- When code is compiled to Emscripten/WASM, it's no longer C++ and other tools apply.
- Microsoft provides [RESTler, a stateful REST API fuzzer](#).

Overview and key qualities

A web application scanner explores a web application by crawling through its web pages and examines it for security vulnerabilities. This crawl involves the automatic generation of malicious inputs and evaluation of the application's responses. Critically, web application scanning must cover/support:

- Catalogs all web apps in your network, including new and unknown ones, and scales from a handful of apps to thousands.
- Deep scanning for software versions, SOAP and REST API services and APIs used by mobile devices.
- Insertion of security primitives into application development and deployment in DevOps environments. These primitives work with the crawler.
- Malware detection.

2.11 Check Included Software Components

Summary

Handle your C++ code the same as code written in other programming languages, and apply any Software Composition Analysis (SCA) and Origin Analysis (OA) tooling adopted by your company to your C++ code. Workflows and security scanning should be designed as part of CI/CD (continuous integration and continuous delivery) systems.

Upstream defense

To mitigate the risk of attacks on upstream dependencies, third party sources/components should be stored on an enterprise-controlled asset, against which SCA and OA tools are run.

- Tools should scan and alert when vulnerabilities are identified (including public databases) such as: [Home](#) | [CVE ↗](#)
- Run static analysis on all software components included in your application/repo to identify vulnerable code patterns.

Dependency defense

Perform and maintain an audit of dependencies to validate that all such occurrences are accounted for and covered by your SCA and OA tools.

- Components should be regularly audited and updated to the latest verified versions.
- Package feed dependencies.
- SCA/OA tools cover and audit all package dependencies that come from a single feed.

SBOM

Produce an SBOM (software bill of materials) with your product listing all dependencies such as:

- origin (for example, URL (Uniform Resource Locator))
- version
- consistency (for example, SHA-256 source hash), and other means for validating consistency such as deterministic builds.
- Require and audit SBOM files in software dependencies or produced as part of a build including OSS (open-source software).
- Microsoft is standardizing on and recommends [SPDX \(Software Package Data Exchange\) version 2.2 or later | Linux Foundation](#) as the SBOM document format.
- Build determinism can be used to independently produce bit-wise identical binaries and provide independent verifications of integrity:
 - First-party or third-party attestation of reproducibility
 - Other techniques such as binary signing via a trusted certificate source can also provide some assurances of binary integrity.

Additional resources

Microsoft solutions include the following guidance and products:

- [Microsoft Supply Chain Platform | Microsoft](#)
- [Secure your software supply chain | GitHub Security](#)
- [vcpkg | vcpkg.io](#) - vcpkg private registries allow redirection of OSS acquisition to Enterprise-controlled resources for acquiring sources for a dependency, to minimize risk of upstream or over-the-wire attacks.

Walkthrough: Analyzing C/C++ code for defects

Article • 08/03/2021

This walkthrough demonstrates how to analyze C/C++ code for potential code defects. It uses the code analysis tools for C/C++ code.

In this walkthrough, you'll:

- Run code analysis on native code.
- Analyze code defect warnings.
- Treat warning as an error.
- Annotate source code to improve code defect analysis.

Prerequisites

- A copy of the [CppDemo Sample](#).
- Basic understanding of C/C++.

Run code analysis on native code

To run code defect analysis on native code

1. Open the CppDemo solution in Visual Studio.

The CppDemo solution now populates **Solution Explorer**.

2. On the **Build** menu, choose **Rebuild Solution**.

The solution builds without any errors or warnings.

3. In **Solution Explorer**, select the CodeDefects project.

4. On the **Project** menu, choose **Properties**.

The **CodeDefects Property Pages** dialog box is displayed.

5. Select the **Code Analysis** property page.

6. Change the **Enable Code Analysis on Build** property to **Yes**. Choose **OK** to save your changes.

7. Rebuild the CodeDefects project.

Code analysis warnings are displayed in the **Error List** window.

To analyze code defect warnings

1. On the **View** menu, choose **Error List**.

This menu item may not be visible. It depends on the developer profile that you chose in Visual Studio. You might have to point to **Other Windows** on the **View** menu, and then choose **Error List**.

2. In the **Error List** window, double-click the following warning:

C6230: Implicit cast between semantically different types: using `HRESULT` in a Boolean context.

The code editor displays the line that caused the warning inside the function `bool ProcessDomain()`. This warning indicates that a `HRESULT` is being used in an 'if' statement where a Boolean result is expected. It's typically a mistake, because when the `S_OK` `HRESULT` is returned from a function it indicates success, but when converted into a boolean value it evaluates to `false`.

3. Correct this warning by using the `SUCCEEDED` macro, which converts to `true` when a `HRESULT` return value indicates success. Your code should resemble the following code:

C++

```
if (SUCCEEDED(ReadUserAccount()))
```

4. In the **Error List**, double-click the following warning:

C6282: Incorrect operator: assignment of constant in Boolean context. Consider using '`==`' instead.

5. Correct this warning by testing for equality. Your code should look similar to the following code:

C++

```
if ((len == ACCOUNT_DOMAIN_LEN) || (g_userAccount[len] != L'\\'))
```

6. Correct the remaining C6001 warnings in the **Error List** by initializing `i` and `j` to 0.

7. Rebuild the CodeDefects project.

The project builds without any warnings or errors.

Correct source code annotation warnings

To enable the source code annotation warnings in annotation.c

1. In Solution Explorer, select the Annotations project.

2. On the **Project** menu, choose **Properties**.

The **Annotations Property Pages** dialog box is displayed.

3. Select the **Code Analysis** property page.

4. Change the **Enable Code Analysis on Build** property to **Yes**. Choose **OK** to save your changes.

To correct the source code annotation warnings in annotation.c

1. Rebuild the Annotations project.

2. On the **Build** menu, choose **Run Code Analysis on Annotations**.

3. In the **Error List**, double-click the following warning:

C6011: Dereferencing NULL pointer 'newNode'.

This warning indicates failure by the caller to check the return value. In this case, a call to `AllocateNode` might return a NULL value. See the annotations.h header file for the function declaration for `AllocateNode`.

4. The cursor is on the location in the annotations.cpp file where the warning occurred.

5. To correct this warning, use an 'if' statement to test the return value. Your code should resemble the following code:

C++

```
LinkedList* newNode = AllocateNode();
if (nullptr != newNode)
{
    newNode->data = value;
    newNode->next = 0;
    node->next = newNode;
}
```

6. Rebuild the Annotations project.

The project builds without any warnings or errors.

Use source code annotation to discover more issues

To use source code annotation

1. Annotate formal parameters and return value of the function `AddTail` to indicate the pointer values may be null:

C++

```
_Ret_maybenull_ LinkedList* AddTail(_Maybenull_ LinkedList* node, int
value)
```

2. On the **Build** menu, choose **Run Code Analysis on Solution**.

3. In the **Error List**, double-click the following warning:

C6011: Dereferencing NULL pointer 'node'.

This warning indicates that the node passed into the function might be null.

4. To correct this warning, use an 'if' statement at the beginning of the function to test the passed in value. Your code should resemble the following code:

C++

```
if (nullptr == node)
{
    return nullptr;
}
```

5. On the **Build** menu, choose **Run Code Analysis on Solution**.

The project now builds without any warnings or errors.

See also

[Walkthrough: Analyzing Managed Code for Code Defects](#)

[Code analysis for C/C++](#)

Sample C++ project for code analysis

Article • 08/03/2021

The following procedures show you how to create the sample for [Walkthrough: Analyze C/C++ code for defects](#). The procedures create:

- A Visual Studio solution named *CppDemo*.
- A static library project named *CodeDefects*.
- A static library project named *Annotations*.

The procedures also provide the code for the header and *.cpp* files for the static libraries.

Create the CppDemo solution and the CodeDefects project

1. Open Visual Studio and select **Create a new project**
2. In the **Create a new project** dialog, change the language filter to **C++**.
3. Select **Windows Desktop Wizard** and choose the **Next** button.
4. On the **Configure your new project** page, in the **Project name** text box, enter *CodeDefects*.
5. In the **Solution name** text box, enter *CppDemo*.
6. Choose **Create**.
7. In the **Windows Desktop Project** dialog, change the **Application type** to **Static Library (.lib)**.
8. Under **Additional options**, select **Empty project**.
9. Choose **OK** to create the solution and project.

Add the header and source file to the CodeDefects project

1. In Solution Explorer, expand **CodeDefects**.
2. Right-click to open the context menu for **Header Files**. Choose **Add > New Item**.

3. In the Add New Item dialog box, select **Visual C++ > Code**, and then select **Header File (.h)**.

4. In the **Name** edit box, enter *Bug.h*, and then choose the **Add** button.

5. In the edit window for *Bug.h*, select and delete the contents.

6. Copy the following code and paste it into the *Bug.h* file in the editor.

```
C++  
  
#pragma once  
  
#include <windows.h>  
  
// Function prototypes  
bool CheckDomain(wchar_t const *);  
HRESULT ReadUserAccount();  
  
// These constants define the common sizes of the  
// user account information throughout the program  
const int USER_ACCOUNT_LEN = 256;  
const int ACCOUNT_DOMAIN_LEN = 128;
```

7. In Solution Explorer, right-click to open the context menu for **Source Files**. Choose **Add > New Item**.

8. In the Add New Item dialog box, select **C++ File (.cpp)**.

9. In the **Name** edit box, enter *Bug.cpp*, and then choose the **Add** button.

10. Copy the following code and paste it into the *Bug.cpp* file in the editor.

```
C++  
  
#include "Bug.h"  
  
// the user account  
wchar_t g_userAccount[USER_ACCOUNT_LEN] = { L"domain\\user" };  
int len = 0;  
  
bool CheckDomain(wchar_t const* domain)  
{  
    return (wcsnlen_s(domain, USER_ACCOUNT_LEN) > 0);  
}  
  
HRESULT ReadUserAccount()  
{  
    return S_OK;  
}
```

```

bool ProcessDomain()
{
    wchar_t* domain = new wchar_t[ACCOUNT_DOMAIN_LEN];
    // ReadUserAccount gets a 'domain\user' input from
    // the user into the global 'g_userAccount'
    if (ReadUserAccount())
    {
        // Copies part of the string prior to the '\'
        // character onto the 'domain' buffer
        for (len = 0; (len < ACCOUNT_DOMAIN_LEN) && (g_userAccount[len]
!= L'\0'); len++)
        {
            if (g_userAccount[len] == L'\\')
            {
                // Stops copying on the domain and user separator ('\'')
                break;
            }
            domain[len] = g_userAccount[len];
        }
        if ((len = ACCOUNT_DOMAIN_LEN) || (g_userAccount[len] !=
L'\\'))
        {
            // '\' was not found. Invalid domain\user string.
            delete[] domain;
            return false;
        }
        else
        {
            domain[len] = L'\0';
        }
        // Process domain string
        bool result = CheckDomain(domain);

        delete[] domain;
        return result;
    }
    return false;
}

int path_dependent(int n)
{
    int i;
    int j;
    if (n == 0)
        i = 1;
    else
        j = 1;
    return i + j;
}

```

11. On the menu bar, choose **File > Save All**.

Add the Annotations project and configure it as a static library

1. In Solution Explorer, right-click **CppDemo** to open the context menu. Choose **Add > New Project**.
2. In the **Add a new project** dialog box, select **Windows Desktop Wizard**, and then choose the **Next** button.
3. On the **Configure your new project** page, in the **Project name** text box, enter *Annotations*, and then choose **Create**.
4. In the **Windows Desktop Project** dialog, change the **Application type** to **Static Library (.lib)**.
5. Under **Additional options**, select **Empty project**.
6. Choose **OK** to create the project.

Add the header file and source file to the Annotations project

1. In Solution Explorer, expand **Annotations**.
2. Right-click to open the context menu for **Header Files** under **Annotations**. Choose **Add > New Item**.
3. In the **Add New Item** dialog box, select **Visual C++ > Code**, and then select **Header File (.h)**.
4. In the **Name** edit box, enter *annotations.h*, and then choose the **Add** button.
5. In the edit window for *annotations.h*, select and delete the contents.
6. Copy the following code and paste it into the *annotations.h* file in the editor.

```
C++  
  
#pragma once  
#include <sal.h>  
  
struct LinkedList  
{  
    struct LinkedList* next;  
    int data;
```

```
};

typedef struct LinkedList LinkedList;

_Ret_maybenull_ LinkedList* AllocateNode();
```

7. In Solution Explorer, right-click to open the context menu for **Source Files** under **Annotations**. Choose **Add > New Item**.
8. In the **Add New Item** dialog box, select **C++ File (.cpp)**.
9. In the **Name** edit box, enter *annotations.cpp*, and then choose the **Add** button.
10. Copy the following code and paste it into the *annotations.cpp* file in the editor.

```
C++

#include "annotations.h"
#include <malloc.h>

_Ret_maybenull_ LinkedList* AllocateNode()
{
    LinkedList* result = static_cast<LinkedList*>
(malloc(sizeof(LinkedList)));
    return result;
}

LinkedList* AddTail(LinkedList* node, int value)
{
    // finds the last node
    while (node->next != nullptr)
    {
        node = node->next;
    }

    // appends the new node
    LinkedList* newNode = AllocateNode();
    newNode->data = value;
    newNode->next = 0;
    node->next = newNode;

    return newNode;
}
```

11. On the menu bar, choose **File > Save All**.

The solution is now complete and should build without errors.

Use the C++ Core Guidelines checkers

Article • 06/21/2023

The C++ Core Guidelines are a portable set of guidelines, rules, and best practices about coding in C++ created by C++ experts and designers. Visual Studio currently supports a subset of these rules as part of its code analysis tools for C++. The core guideline checkers are installed by default in Visual Studio 2017 and Visual Studio 2019. They're available as a NuGet package for Visual Studio 2015.

The C++ Core Guidelines Project

Created by Bjarne Stroustrup and others, the C++ Core Guidelines are a guide to using modern C++ safely and effectively. The Guidelines emphasize static type safety and resource safety. They identify ways to eliminate or minimize the most error-prone parts of the language. They also suggest how to make your code simpler, more reliable, and have better performance. These guidelines are maintained by the Standard C++ Foundation. To learn more, see the documentation, [C++ Core Guidelines](#), and access the C++ Core Guidelines documentation project files on [GitHub](#).

Enable the C++ Core Check guidelines in Code Analysis

A subset of C++ Core Check rules is included in the Microsoft Native Recommended rule set. It's the ruleset that runs by default when Microsoft code analysis is enabled.

To enable code analysis on your project:

1. Open the **Property Pages** dialog for your project.
2. Select the **Configuration Properties > Code Analysis** property page.
3. Set the **Enable Code Analysis on Build** and **Enable Microsoft Code Analysis** properties.

You can also choose to run all the supported C++ Core Check rules, or select your own subset to run:

To enable additional Core Check rules

1. Open the **Property Pages** dialog for your project.
2. Select the **Configuration Properties > Code Analysis > Microsoft** property page.
3. Open the **Active Rules** dropdown list and select **Choose multiple rule sets**.
4. In the **Add or Remove Rule Sets** dialog, choose which rule sets you want to include.

Examples

Here's an example of some of the issues that the C++ Core Check rules can find:

```
C++  
  
// CoreCheckExample.cpp  
// Add CppCoreCheck package and enable code analysis in build for warnings.  
  
int main()  
{  
    int arr[10];           // warning C26494  
    int* p = arr;          // warning C26485  
  
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1  
    {  
        int* q = p + 1;     // warning C26481 (suppressed)  
        p = q++;            // warning C26481 (suppressed)  
    }  
  
    return 0;  
}
```

This example demonstrates a few of the warnings that the C++ Core Check rules can find:

- C26494 is rule Type.5: Always initialize an object.
- C26485 is rule Bounds.3: No array-to-pointer decay.
- C26481 is rule Bounds.1: Don't use pointer arithmetic. Use `span` instead.

Install and enable the C++ Core Check code analysis rulesets, then compile this code. Code analysis outputs the first two warnings, and suppresses the third. Here's the build output from the example code in Visual Studio 2015:

Output

```
1>----- Build started: Project: CoreCheckExample, Configuration: Debug  
Win32 -----  
1> CoreCheckExample.cpp  
1> CoreCheckExample.vcxproj -> C:\Users\username\documents\visual studio  
2015\Projects\CoreCheckExample\Debug\CoreCheckExample.exe  
1> CoreCheckExample.vcxproj -> C:\Users\username\documents\visual studio  
2015\Projects\CoreCheckExample\Debug\CoreCheckExample.pdb (Full PDB)  
c:\users\username\documents\visual studio  
2015\projects\corecheckexample\corecheckexample\corecheckexample.cpp(6):  
warning C26494: Variable 'arr' is uninitialized. Always initialize an  
object. (type.5: http://go.microsoft.com/fwlink/?LinkID=620421)  
c:\users\username\documents\visual studio  
2015\projects\corecheckexample\corecheckexample\corecheckexample.cpp(7):  
warning C26485: Expression 'arr': No array to pointer decay. (bounds.3:  
http://go.microsoft.com/fwlink/?LinkID=620415)  
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

The C++ Core Guidelines are there to help you write better and safer code. However, you may find an instance where a rule or a profile shouldn't be applied. It's easy to suppress it directly in the code. You can use the `[[gsl::suppress]]` attribute to keep C++ Core Check from detecting and reporting any violation of a rule in the following code block. You can mark individual statements to suppress specific rules. You can even suppress the entire bounds profile by writing `[[gsl::suppress(bounds)]]` without including a specific rule number.

Supported rule sets

As new rules are added to the C++ Core Guidelines Checker, the number of warnings that are produced for pre-existing code may increase. You can use predefined rule sets to filter which kinds of rules to enable. You'll find reference articles for most rules under [Visual Studio C++ Core Check Reference](#).

- **Arithmetic Rules:** Rules to detect arithmetic [overflow](#), [signed-unsigned operations](#), and [bit manipulation](#).^{15.6}
- **Bounds Rules:** Enforce the [Bounds profile of the C++ Core Guidelines](#).^{15.3}
- **Class Rules:** A few rules that focus on proper use of special member functions and virtual specifications. They're a subset of the checks recommended for [classes and class hierarchies](#).^{15.5}
- **Concurrency Rules:** A single rule, which catches bad guard object declarations. For more information, see [guidelines related to concurrency](#).^{15.5}
- **Const Rules:** Enforce [const-related checks from the C++ Core Guidelines](#).^{15.3}

- **Declaration Rules:** A couple of rules from the [interfaces guidelines](#)^{15.5} that focus on how global variables are declared.^{15.5}
- **Enum Rules:** These rules enforce [enum-related checks from the C++ Core Guidelines](#)^{16.3}.
- **Experimental Rules** These are experimental C++ Core Check rules that are useful but not ready for everyday use. Try them out and [provide feedback](#)^{16.0}.
- **Function Rules:** Two checks that help with adoption of the `noexcept` specifier. They're part of the guidelines for [clear function design and implementation](#)^{15.5}.
- **GSL Rules:** These rules enforce checks related to the [Guidelines Support Library from the C++ Core Guidelines](#)^{15.7}.
- **Lifetime Rules:** These rules enforce the [Lifetime profile of the C++ Core Guidelines](#)^{15.7}.
- **Owner Pointer Rules:** Enforce [resource-management checks related to owner<T>](#) from the [C++ Core Guidelines](#)^{15.3}.
- **Raw Pointer Rules:** Enforce [resource-management checks related to raw pointers](#) from the [C++ Core Guidelines](#)^{15.3}.
- **Shared pointer Rules:** It's part of [resource management](#)^{15.5} guidelines enforcement. We added a few rules specific to how shared pointers are passed into functions or used locally.
- **STL Rules:** These rules enforce checks related to the [C++ Standard Library \(STL\)](#) from the [C++ Core Guidelines](#)^{15.7}.
- **Style Rules:** One simple but important check, which bans use of [goto](#)^{15.5}. It's the first step to improve your coding style and use of expressions and statements in C++.
- **Type Rules:** Enforce the [Type profile of the C++ Core Guidelines](#)^{15.3}.
- **Unique Pointer Rules:** Enforce [resource-management checks related to types with unique pointer semantics](#) from the [C++ Core Guidelines](#)^{15.3}.
- **C++ Core Check Rules:** This rule set contains all the currently implemented checks from the [C++ Core Guidelines](#), except for the Experimental rules.

^{15.3} These rules first appeared in Visual Studio 2017 version 15.3

^{15.5} These rules first appeared in Visual Studio 2017 version 15.5

^{15.6} These rules first appeared in Visual Studio 2017 version 15.6

15.7 These rules first appeared in Visual Studio 2017 version 15.7

16.0 These rules first appeared in Visual Studio 2019 version 16.0

16.3 These rules first appeared in Visual Studio 2019 version 16.3

You can choose to limit warnings to just one or a few of the groups. The **Native Minimum** and **Native Recommended** rule sets include C++ Core Check rules and other PREfast checks.

To see the available rule sets, open the **Project Properties** dialog. In the **Property Pages** dialog box, select the **Configuration Properties > Code Analysis > Microsoft** property page. Then, open the dropdown in the **Active Rules** combo-box to see the available rule sets. To build a custom combination of rule sets, select **Choose multiple rule sets**. The **Add or Remove Rule Sets** dialog lists the rules you can choose from. For more information about using Rule Sets in Visual Studio, see [Use rule sets to specify the C++ rules to run](#).

Macros

The C++ Core Guidelines Checker comes with a header file, which defines macros that make it easier to suppress entire categories of warnings in code:

C++

```
ALL_CPPCORECHECK_WARNINGS  
CPPCORECHECK_TYPE_WARNINGS  
CPPCORECHECK_RAW_POINTER_WARNINGS  
CPPCORECHECK_CONST_WARNINGS  
CPPCORECHECK_OWNER_POINTER_WARNINGS  
CPPCORECHECK_UNIQUE_POINTER_WARNINGS  
CPPCORECHECK_BOUNDS_WARNINGS
```

These macros correspond to the rule sets and expand into a space-separated list of warning numbers. By using the appropriate pragma constructs, you can configure the effective set of rules that is interesting for a project or a section of code. In the following example, code analysis warns only about missing constant modifiers:

C++

```
#include <CppCoreCheck\Warnings.h>  
#pragma warning(disable: ALL_CPPCORECHECK_WARNINGS)  
#pragma warning(default: CPPCORECHECK_CONST_WARNINGS)
```

Attributes

The Microsoft C++ compiler has limited support for the `[[gsl::suppress]]` attribute. It can be used to suppress warnings on expression and block statements inside functions.

C++

```
// Suppress only warnings from the 'r.11' rule in expression.  
[[gsl::suppress(r.11)]] new int;  
  
// Suppress all warnings from the 'r' rule group (resource management) in  
block.  
[[gsl::suppress(r)]]  
{  
    new int;  
}  
  
// Suppress only one specific warning number.  
// For declarations, you may need to use the surrounding block.  
// Macros are not expanded inside of attributes.  
// Use plain numbers instead of macros from the warnings.h.  
[[gsl::suppress(26400)]]  
{  
    int *p = new int;  
}
```

Suppress analysis by using command-line options

Instead of #pragmas, you can use command-line options in the file's property page to suppress warnings for a project or a single file. For example, to disable the warning C26400 for a file:

1. Right-click the file in **Solution Explorer** and choose **Properties**.
2. In the **Property Pages** dialog box, select the **Configuration Properties > C/C++ > Command Line** property page.
3. In the **Additional Options** edit box, add `/wd26400`.

You can use the command-line option to temporarily disable all code analysis for a file by specifying `/analyze-`. You'll see warning *D9025 overriding '/analyze' with '/analyze-'*, which reminds you to re-enable code analysis later.

Enable the C++ Core Guidelines Checker on specific project files

Sometimes it's useful to do focused code analysis and still use the Visual Studio IDE. Try the following sample scenario for large projects. It can save build time and make it easier to filter results:

1. In the command shell, set the `esp.extension` environment variable.
2. To inherit this variable, open Visual Studio from the command shell.
3. Load your project and open its properties.
4. Enable code analysis, pick the appropriate rule sets, but don't enable code analysis extensions.
5. Go to the file you want to analyze with the C++ Core Guidelines Checker and open its properties.
6. Choose **Configuration Properties > C/C++ > Command Line > Additional Options** and add `/analyze:plugin EspXEngine.dll`
7. Disable the use of precompiled header (**Configuration Properties > C/C++ > Precompiled Headers**). It's necessary because the extensions engine may attempt to read its internal information from the precompiled header (PCH). If the PCH was compiled with default project options, it won't be compatible.
8. Rebuild the project. The common PREFast checks should run on all files. Because the C++ Core Guidelines Checker isn't enabled by default, it should only run on the file that's configured to use it.

How to use the C++ Core Guidelines Checker outside of Visual Studio

You can use the C++ Core Guidelines checks in automated builds.

MSBuild

The Native Code Analysis checker (PREfast) is integrated into MSBuild environment by custom targets files. You can use project properties to enable it, and add the C++ Core Guidelines Checker (which is based on PREfast):

XML

```
<PropertyGroup>
  <EnableCppCoreCheck>true</EnableCppCoreCheck>
  <CodeAnalysisRuleSet>CppCoreCheckRules.ruleset</CodeAnalysisRuleSet>
```

```
<RunCodeAnalysis>true</RunCodeAnalysis>
</PropertyGroup>
```

Make sure you add these properties before the import of the `Microsoft.Cpp.targets` file. You can pick specific rule sets or create a custom rule set. Or, use the default rule set that includes other PREfast checks.

You can run the C++ Core Checker only on specified files. Use the same approach as [described earlier](#), but use MSBuild files. The environment variables can be set by using the `BuildMacro` item:

XML

```
<ItemGroup>
  <BuildMacro Include="Esp_Extensions">
    <EnvironmentVariable>true</EnvironmentVariable>
    <Value>CppCoreCheck.dll</Value>
  </BuildMacro>
</ItemGroup>
```

If you don't want to modify the project file, you can pass properties on the command line:

Windows Command Prompt

```
msbuild /p:EnableCppCoreCheck=true /p:RunCodeAnalysis=true
/p:CodeAnalysisRuleSet=CppCoreCheckRules.ruleset ...
```

Non-MSBuild projects

If you use a build system that doesn't rely on MSBuild, you can still run the checker. To use it, you need to get familiar with some internals of the Code Analysis engine configuration. We don't guarantee support for these internals in future versions of Visual Studio.

Code Analysis requires a few environment variables and compiler command-line options. We recommend you use the **Native Tools Command Prompt** environment so you don't have to search for specific paths for the compiler, include directories, and so on.

- **Environment variables**
 - `set esp.extensions=cppcorecheck.dll` This tells the engine to load the C++ Core Guidelines module.

- Since Visual Studio 2019 we no longer recommend setting the `esp.annotationbuildlevel` environment variable because setting it can result in false positives. If seeing unexpected results, remove this variable from your environment.
 - `set caexcludepath=%include%` We highly recommend that you disable warnings that fire on standard headers. You can add more paths here, for example the path to the common headers in your project.
- Command-line options
 - `/analyze` Enables code analysis (consider also using `/analyze:only` and `/analyze:quiet`).
 - `/analyze:plugin EspXEngine.dll` This option loads the Code Analysis Extensions engine into the PREfast. This engine, in turn, loads the C++ Core Guidelines Checker.

Use the Guideline Support Library

The Guideline Support Library (GSL) is designed to help you follow the Core Guidelines. The GSL includes definitions that let you replace error-prone constructs with safer alternatives. For example, you can replace a `T*, length` pair of parameters with the `span<T>` type. The GSL project is available on GitHub at <https://github.com/Microsoft/GSL>. The library is open-source, so you can view the sources, make comments, or contribute. You can also use the [vcpkg](#) package manager to download and install the library locally.

See also

- [Visual Studio C++ Core Check Reference](#)

How to: Set Code Analysis Properties for C/C++ Projects

Article • 08/03/2021

You can configure which rules the code analysis tool uses to analyze the code in each configuration of your project. In addition, you can direct code analysis to suppress warnings from code that was generated and added to your project by a third-party tool.

Code Analysis Property Page

The **Code Analysis** property page contains all code analysis configuration settings for an MSBuild project. To open the code analysis property page for a project in **Solution Explorer**, right-click the project and then click **Properties**. Next, expand **Configuration Properties** and select the **Code Analysis** tab.

Project Configuration and Platform

The **Configuration** list and **Platform** list at the top of the window lets you apply different code analysis settings to different project configuration and platform combinations. For example, you can direct code analysis to apply one set of rules to your project for debug builds and a different set for release builds.

Enabling Code Analysis

You can enable code analysis for your project by toggling the **Enable Microsoft Code Analysis** and **Enable Clang-Tidy** options, and further configure if it runs on build by selecting **Enable Code Analysis on Build**. In combination with the **Configuration** list, you could, for example, decide to disable Code Analysis for debug builds and enable it for release builds.

Code analysis is designed to help you improve the quality of your code and avoid common pitfalls. Therefore, consider carefully whether to disable code analysis. It is usually better to disable rule sets, individual rules, or individual checks that you do not want applied to your project.

CMake configuration

In CMake projects, change the value of the `enableMicrosoftCodeAnalysis` and `enableClangTidyCodeAnalysis` keys within `cMakeSettings.json` to enable or disable code analysis. See [Using Clang-Tidy in Visual Studio](#) for more information.

See also

- [Analyzing Managed Code Quality](#)
- [Code Analysis for C/C++ Warnings](#)
- [Use rule sets to specify the C++ rules to run](#)
- [Using Clang-Tidy](#)

Use Rule Sets to Specify the C++ Rules to Run

Article • 11/07/2022

In Visual Studio, you can create and modify a custom *rule set* to meet specific project needs associated with code analysis. The default rule sets are stored in `%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Rule Sets`.

Visual Studio 2017 version 15.7 and later: You can create custom rule sets using any text editor and apply them in command line builds no matter what build system you're using. For more information, see [/analyze:ruleset](#).

To create a custom C++ rule set in Visual Studio, a C/C++ project must be open in the Visual Studio IDE. You then open a standard rule set in the rule set editor and then add or remove specific rules and optionally change the action that occurs when code analysis determines a rule has been violated.

To create a new custom rule set, you save it by using a new file name. The custom rule set is automatically assigned to the project.

To create a custom rule from a single existing rule set

1. In Solution Explorer, open the shortcut menu for the project and then choose **Properties**.
2. In the **Property Pages** dialog box, select the **Configuration Properties > Code Analysis > Microsoft** property page.
3. In the **Active Rules** drop-down list, do one of the following:
 - Choose the rule set that you want to customize.
 - or -
 - Choose **<Browse...>** to specify an existing rule set that isn't in the list.
4. Choose **Open** to display the rules in the rule set editor.

To modify a rule set in the rule set editor

- To change the display name of the rule set, on the **View** menu, choose **Properties Window**. Enter the display name in the **Name** box. Notice that the display name can differ from the file name.
- To add all the rules of the group to a custom rule set, select the check box of the group. To remove all the rules of the group, clear the check box.
- To add a specific rule to the custom rule set, select the check box of the rule. To remove the rule from the rule set, clear the check box.
- To change the action taken when a rule is violated in a code analysis, choose the **Action** field for the rule and then choose one of the following values:

Warning - generates a warning.

Error - generates an error.

Info - generates a message.

None - disables the rule. This action is the same as removing the rule from the rule set.

To group, filter, or change the fields in the rule set editor by using the rule set editor toolbar

- To expand the rules in all groups, choose **Expand All**.
- To collapse the rules in all groups, choose **Collapse All**.
- To change the field that rules are grouped by, choose the field from the **Group By** list. To display the rules ungrouped, choose **<None>**.
- To add or remove fields in rule columns, choose **Column Options**.
- To hide rules that don't apply to the current solution, choose **Hide rules that do not apply to the current solution**.
- To switch between showing and hiding rules that are assigned the Error action, choose **Show rules that can generate Code Analysis errors**.
- To switch between showing and hiding rules that are assigned the Warning action, choose **Show rules that can generate Code Analysis warnings**.
- To switch between showing and hiding rules that are assigned the **None** action, choose **Show rules that are not enabled**.

- To add or remove Microsoft default rule sets to the current rule set, choose **Add** or **remove child rule sets**.

To create a rule set in a text editor

You can create a custom rule set in a text editor, store it in any location with a `.ruleset` extension, and apply in with the `/analyze:ruleset` compiler option.

The following example shows a basic rule set file that you can use as a starting point:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="New Rule Set" Description="New rules to apply."
ToolsVersion="10.0">
    <Rules AnalyzerId="Microsoft.Analyzers.NativeCodeAnalysis"
RuleNamespace="Microsoft.Rules.Native">
        <Rule Id="C6001" Action="Warning" />
        <Rule Id="C26494" Action="Warning" />
    </Rules>
</RuleSet>
```

Ruleset schema

The following ruleset schema describes the XML schema of a ruleset file. The ruleset schema is stored in `%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Schemas\RuleSet.xsd`. You can use it to author your own rulesets programmatically or to validate whether your custom rulesets adhere to the correct format. For more information, see [How to: Create an XML document based on an XSD schema](#).

XML

```
<?xml version="1.0" encoding="utf-8"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">

    <xss:annotation>
        <xss:documentation xml:lang="en">
            Visual Studio Code Analysis Rule Set Schema Definition Language.
            Copyright (c) Microsoft Corporation. All rights reserved.
        </xss:documentation>
    </xss:annotation>

    <!-- Every time this file changes, be sure to change the Validate method
for the corresponding object in the code -->

    <xss:element name="RuleSet" type="TRuleSet">
```

```

</xs:element>

<xs:complexType name="TLocalization">
  <xs:all>
    <xs:element name="Name" type="TName" minOccurs="0" maxOccurs="1" />
    <xs:element name="Description" type="TDescription" minOccurs="0"
maxOccurs="1" />
  </xs:all>
  <xs:attribute name="ResourceAssembly" type="TNonEmptyString"
use="required" />
  <xs:attribute name="ResourceBaseName" type="TNonEmptyString"
use="required" />
</xs:complexType>

<xs:complexType name="TRuleHintPaths">
  <xs:sequence>
    <xs:element name="Path" type="TNonEmptyString" minOccurs="0"
maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TName">
  <xs:attribute name="Resource" type="TNonEmptyString" use="required" />
</xs:complexType>

<xs:complexType name="TDescription">
  <xs:attribute name="Resource" type="TNonEmptyString" use="required" />
</xs:complexType>

<xs:complexType name="TInclude">
  <xs:attribute name="Path" type="TNonEmptyString" use="required" />
  <xs:attribute name="Action" type="TIncludeAction" use="required" />
</xs:complexType>

<xs:complexType name="TIncludeAll">
  <xs:attribute name="Action" type="TIncludeAllAction" use="required" />
</xs:complexType>

<xs:complexType name="TRule">
  <xs:attribute name="Id" type="TNonEmptyString" use="required" />
  <xs:attribute name="Action" type="TRuleAction" use="required" />
</xs:complexType>

<xs:complexType name="TRules">
  <xs:sequence>
    <xs:element name="Rule" type="TRule" minOccurs="0"
maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="AnalyzerId" type="TNonEmptyString" use="required" />
  <xs:attribute name="RuleNamespace" type="TNonEmptyString" use="required"
/>
</xs:complexType>

<xs:complexType name="TRuleSet">
  <xs:sequence minOccurs="0" maxOccurs="1">

```

```

        <xs:element name="Localization" type="TLocalization" minOccurs="0"
maxOccurs="1" />
        <xs:element name="RuleHintPaths" type="TRuleHintPaths" minOccurs="0"
maxOccurs="1" />
        <xs:element name="IncludeAll" type="TIncludeAll" minOccurs="0"
maxOccurs="1" />
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Include" type="TInclude" minOccurs="0"
maxOccurs="unbounded" />
            <xs:element name="Rules" type="TRules" minOccurs="0"
maxOccurs="unbounded">
                <xs:unique name="UniqueRuleName">
                    <xs:selector xpath="Rule" />
                    <xs:field xpath="@Id" />
                </xs:unique>
            </xs:element>
        </xs:choice>
    </xs:sequence>
    <xs:attribute name="Name" type="TNonEmptyString" use="required" />
    <xs:attribute name="Description" type="xs:string" use="optional" />
    <xs:attribute name="ToolsVersion" type="TNonEmptyString" use="required"
/>
</xs:complexType>

<xs:simpleType name="TRuleAction">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Error"/>
        <xs:enumeration value="Warning"/>
        <xs:enumeration value="Info"/>
        <xs:enumeration value="Hidden"/>
        <xs:enumeration value="None"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TIncludeAction">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Error"/>
        <xs:enumeration value="Warning"/>
        <xs:enumeration value="Info"/>
        <xs:enumeration value="Hidden"/>
        <xs:enumeration value="None"/>
        <xs:enumeration value="Default"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TIncludeAllAction">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Error"/>
        <xs:enumeration value="Warning"/>
        <xs:enumeration value="Info"/>
        <xs:enumeration value="Hidden"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TNonEmptyString">

```

```

<xs:restriction base="xs:string">
  <xs:minLength value="1" />
</xs:restriction>
</xs:simpleType>

</xs:schema>

```

Schema element details:

Schema element	Description
TLocalization	Localization information including name of the ruleset file, description of the ruleset file, name of the resource assembly containing the localized resource, and base name of the localized resource
TRuleHintPaths	File paths used as hints to search for ruleset files
TName	Name of the current ruleset file
TDescription	Description of the current ruleset file
TInclude	Path to an included ruleset with rule action
TIncludeAll	Rule action for all rules
TRule	Rule ID with rule action
TRules	Collection of one or more rules
TRuleSet	Ruleset file format consisting of localization information, rule hint paths, include all information, include information, rules information, name, description, and tools version information
TRuleAction	Enumeration describing a rule action such as an error, warning, info, hidden, or none
TIncludeAction	Enumeration describing a rule action such as an error, warning, info, hidden, none, or default
TIncludeAllAction	Enumeration describing a rule action such as an error, warning, info, or hidden

To see an example of a ruleset, see [To create a rule set in a text editor](#), or any of the default rulesets stored in `%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Rule Sets`.

Using Clang-Tidy in Visual Studio

Article • 05/24/2024

Code Analysis natively supports [Clang-Tidy](#) for both MSBuild and CMake projects, whether using Clang or MSVC toolsets. Clang-Tidy checks can run as part of background code analysis. They appear as in-editor warnings (squiggles), and display in the Error List.

Clang-Tidy support is available starting in Visual Studio 2019 version 16.4. It's included automatically when you choose a C++ workload in the Visual Studio Installer.

Clang-Tidy is the default analysis tool when using the LLVM/clang-cl toolset, available in both MSBuild and CMake. You can configure it when using an MSVC toolset to run alongside, or to replace, the standard Code Analysis experience. If you use the clang-cl toolset, Microsoft Code Analysis is unavailable.

Clang-Tidy runs after successful compilation. You may need to resolve source code errors to get Clang-Tidy results.

MSBuild

You can configure Clang-Tidy to run as part of both Code Analysis and build under the **Code Analysis > General** page in the Project Properties window. Options to configure the tool can be found under the Clang-Tidy submenu.

For more information, see [How to: Set Code Analysis Properties for C/C++ Projects](#).

CMake

In CMake projects, you can configure Clang-Tidy checks within `CMakeSettings.json` or `CMakePresets.json`.

Clang-Tidy recognizes the following keys:

- `enableMicrosoftCodeAnalysis`: Enables Microsoft Code Analysis
- `enableClangTidyCodeAnalysis`: Enables Clang-Tidy analysis
- `clangTidyChecks`: Clang-Tidy configuration. A comma-separated list of checks to enable or disable. A leading `-` disables the check. For example, `cert-oop58-cpp, -cppcoreguidelines-no-malloc, google-runtime-int` enables `cert-oop58-cpp` and

`google-runtime-int`, but disables `cppcoreguidelines-no-malloc`. For a list of Clang-Tidy checks, see the [Clang-Tidy documentation](#).

If neither of the "enable" options are specified, Visual Studio selects the analysis tool matching the Platform Toolset used.

CMake settings

To edit your Clang-Tidy settings, open your CMake settings, and select **Edit JSON** in the CMake Project Settings Editor. You can use the keys above to fill out your Clang-Tidy specifications in the CMake Settings JSON file.

An example CMake settings implementation looks like this:

```
JSON

{
  "configurations": [
    {
      "name": "x64-debug",
      "generator": "Ninja",
      ...
      "clangTidyChecks": "llvm/include-order, -modernize-use-override",
      "enableMicrosoftCodeAnalysis": true,
      "enableClangTidyCodeAnalysis": true
    }
  ]
}
```

CMake presets

The same keys can be used in your CMake presets via the `vendor` object.

An example CMake preset implementation looks like this:

```
JSON

"configurePreset": [
  { "name": "base",
    ...
    "vendor": {
      "microsoft.com/VisualStudioSettings/CMake/1.0": {
        "clangTidyChecks": "llvm/include-order, -modernize-use-override",
        "enableMicrosoftCodeAnalysis": true,
        "enableClangTidyCodeAnalysis": true
      }
    }
}
```

```
}
```

```
]
```

Warning display

Clang-Tidy runs result in warnings displayed in the Error List, and as in-editor squiggles underneath relevant sections of code. To sort and organize Clang-Tidy warnings, use the **Category** column in the **Error List** window. You can configure in-editor warnings by toggling the **Disable Code Analysis Squiggles** setting under **Tools > Options**.

Clang-Tidy configuration

By default, Clang-Tidy does not set any checks when enabled. To see the list of checks in the command-line version, run `clang-tidy -list-checks` in a developer command prompt. You can configure the checks that Clang-Tidy runs inside Visual Studio. In the project Property Pages dialog, open the **Configuration Properties > Code Analysis > Clang-Tidy** page. Enter checks to run in the **Clang-Tidy Checks** property. A good default set is `clang-analyzer-*`. This property value is provided to the `--checks` argument of the tool. Any further configuration can be included in custom `.clang-tidy` files. For more information, see the [Clang-Tidy documentation on LLVM.org ↗](#).

Clang-Tidy Tool Directory

If you'd like to have custom rules built into your clang-tidy executable and run it in Microsoft Visual Studio, you can change the path to the executable that Visual Studio runs. In the project Property Pages dialog, open the **Configuration Properties > Code Analysis > Clang-Tidy** page. Manually type in the path or **Browse** and select the path under the **Clang-Tidy Tool Directory** property. The new executable is used once the change is saved, and the app is recompiled.

See also

- [Clang/LLVM support for MSBuild projects ↗](#)
- [Clang/LLVM support for CMake projects ↗](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Using SAL Annotations to Reduce C/C++ Code Defects

Article • 08/03/2021

SAL is the Microsoft source code annotation language. By using source code annotations, you can make the intent behind your code explicit. These annotations also enable automated static analysis tools to analyze your code more accurately, with significantly fewer false positives and false negatives.

The articles in this section of the documentation discuss aspects of SAL, provide reference for SAL syntax, and give examples of its use.

- [Understanding SAL](#)

Provides information and examples that show the core SAL annotations.

- [Annotating Function Parameters and Return Values](#)

Lists the SAL annotations for functions and function parameters.

- [Annotating Function Behavior](#)

Lists the SAL annotations for functions and function behavior.

- [Annotating Structs and Classes](#)

Lists the SAL annotations for structures and classes.

- [Annotating Locking Behavior](#)

Explains how to use SAL annotations with lock mechanisms.

- [Specifying When and Where an Annotation Applies](#)

Lists the SAL annotations that specify the condition or scope (placement) of other SAL annotations.

- [Intrinsic Functions](#)

Lists the intrinsic SAL annotations.

- [Best Practices and Examples](#)

Provides examples that show how to use SAL annotations. Also explains common pitfalls.

See Also

[SAL 2.0 Annotations for Windows Drivers](#)

Understanding SAL

Article • 11/07/2022

The Microsoft source-code annotation language (SAL) provides a set of annotations that you can use to describe how a function uses its parameters, the assumptions that it makes about them, and the guarantees that it makes when it finishes. The annotations are defined in the header file `<sal.h>`. Visual Studio code analysis for C++ uses SAL annotations to modify its analysis of functions. For more information about SAL 2.0 for Windows driver development, see [SAL 2.0 Annotations for Windows Drivers](#).

Natively, C and C++ provide only limited ways for developers to consistently express intent and invariance. By using SAL annotations, you can describe your functions in greater detail so that developers who are consuming them can better understand how to use them.

What Is SAL and Why Should You Use It?

Simply stated, SAL is an inexpensive way to let the compiler check your code for you.

SAL Makes Code More Valuable

SAL can help you make your code design more understandable, both for humans and for code analysis tools. Consider this example that shows the C runtime function `memcpy`:

C++

```
void * memcpy(
    void *dest,
    const void *src,
    size_t count
);
```

Can you tell what this function does? When a function is implemented or called, certain properties must be maintained to ensure program correctness. Just by looking at a declaration such as the one in the example, you don't know what they are. Without SAL annotations, you'd have to rely on documentation or code comments. Here's what the documentation for `memcpy` says:

"`memcpy` copies *count* bytes from *src* to *dest*; `wmemcpy` copies *count* wide characters (two bytes). If the source and destination overlap, the behavior of `memcpy` is undefined. Use `memmove` to handle overlapping regions."

Important: Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns."

The documentation contains a couple of bits of information that suggest that your code has to maintain certain properties to ensure program correctness:

- `memcpy` copies the `count` of bytes from the source buffer to the destination buffer.
- The destination buffer must be at least as large as the source buffer.

However, the compiler can't read the documentation or informal comments. It doesn't know that there is a relationship between the two buffers and `count`, and it also can't effectively guess about a relationship. SAL could provide more clarity about the properties and implementation of the function, as shown here:

C++

```
void * memcpy(
    _Out_writes_bytes_all_(count) void *dest,
    _In_reads_bytes_(count) const void *src,
    size_t count
);
```

Notice that these annotations resemble the information in the documentation, but they are more concise and they follow a semantic pattern. When you read this code, you can quickly understand the properties of this function and how to avoid buffer overrun security issues. Even better, the semantic patterns that SAL provides can improve the efficiency and effectiveness of automated code analysis tools in the early discovery of potential bugs. Imagine that someone writes this buggy implementation of `wmemcpy`:

C++

```
wchar_t * wmemcpy(
    _Out_writes_all_(count) wchar_t *dest,
    _In_reads_(count) const wchar_t *src,
    size_t count)
{
    size_t i;
    for (i = 0; i <= count; i++) { // BUG: off-by-one error
        dest[i] = src[i];
    }
    return dest;
}
```

This implementation contains a common off-by-one error. Fortunately, the code author included the SAL buffer size annotation—a code analysis tool could catch the bug by

analyzing this function alone.

SAL Basics

SAL defines four basic kinds of parameters, which are categorized by usage pattern.

Category	Parameter Annotation	Description
Input to called function	<code>_In_</code>	Data is passed to the called function, and is treated as read-only.
Input to called function, and output to caller	<code>_Inout_</code>	Usable data is passed into the function and potentially is modified.
Output to caller	<code>_Out_</code>	The caller only provides space for the called function to write to. The called function writes data into that space.
Output of pointer to caller	<code>_Outptr_</code>	Like Output to caller . The value that's returned by the called function is a pointer.

These four basic annotations can be made more explicit in various ways. By default, annotated pointer parameters are assumed to be required—they must be non-NULL for the function to succeed. The most commonly used variation of the basic annotations indicates that a pointer parameter is optional—if it's NULL, the function can still succeed in doing its work.

This table shows how to distinguish between required and optional parameters:

	Parameters are required	Parameters are optional
Input to called function	<code>_In_</code>	<code>_In_opt_</code>
Input to called function, and output to caller	<code>_Inout_</code>	<code>_Inout_opt_</code>
Output to caller	<code>_Out_</code>	<code>_Out_opt_</code>
Output of pointer to caller	<code>_Outptr_</code>	<code>_Outptr_opt_</code>

These annotations help identify possible uninitialized values and invalid null pointer uses in a formal and accurate manner. Passing NULL to a required parameter might cause a crash, or it might cause a "failed" error code to be returned. Either way, the function cannot succeed in doing its job.

SAL Examples

This section shows code examples for the basic SAL annotations.

Using the Visual Studio Code Analysis Tool to Find Defects

In the examples, the Visual Studio Code Analysis tool is used together with SAL annotations to find code defects. Here's how to do that.

To use Visual Studio code analysis tools and SAL

1. In Visual Studio, open a C++ project that contains SAL annotations.
2. On the menu bar, choose **Build, Run Code Analysis on Solution**.

Consider the `_In_` example in this section. If you run code analysis on it, this warning is displayed:

C6387 Invalid Parameter Value 'pInt' could be '0': this does not adhere to the specification for the function 'InCallee'.

Example: The `_In_` Annotation

The `_In_` annotation indicates that:

- The parameter must be valid and will not be modified.
- The function will only read from the single-element buffer.
- The caller must provide the buffer and initialize it.
- `_In_` specifies "read-only". A common mistake is to apply `_In_` to a parameter that should have the `_Inout_` annotation instead.
- `_In_` is allowed but ignored by the analyzer on non-pointer scalars.

C++

```
void InCallee(_In_ int *pInt)
{
    int i = *pInt;
}
```

```

void GoodInCaller()
{
    int *pInt = new int;
    *pInt = 5;

    InCallee(pInt);
    delete pInt;
}

void BadInCaller()
{
    int *pInt = NULL;
    InCallee(pInt); // pInt should not be NULL
}

```

If you use Visual Studio Code Analysis on this example, it validates that the callers pass a non-Null pointer to an initialized buffer for `pInt`. In this case, `pInt` pointer cannot be NULL.

Example: The `_In_opt_` Annotation

`_In_opt_` is the same as `_In_`, except that the input parameter is allowed to be NULL and, therefore, the function should check for this.

C++

```

void GoodInOptCallee(_In_opt_ int *pInt)
{
    if(pInt != NULL) {
        int i = *pInt;
    }
}

void BadInOptCallee(_In_opt_ int *pInt)
{
    int i = *pInt; // Dereferencing NULL pointer 'pInt'
}

void InOptCaller()
{
    int *pInt = NULL;
    GoodInOptCallee(pInt);
    BadInOptCallee(pInt);
}

```

Visual Studio Code Analysis validates that the function checks for NULL before it accesses the buffer.

Example: The `_Out_` Annotation

`_Out_` supports a common scenario in which a non-NULL pointer that points to an element buffer is passed in and the function initializes the element. The caller doesn't have to initialize the buffer before the call; the called function promises to initialize it before it returns.

C++

```
void GoodOutCallee(_Out_ int *pInt)
{
    *pInt = 5;
}

void BadOutCallee(_Out_ int *pInt)
{
    // Did not initialize pInt buffer before returning!
}

void OutCaller()
{
    int *pInt = new int;
    GoodOutCallee(pInt);
    BadOutCallee(pInt);
    delete pInt;
}
```

Visual Studio Code Analysis Tool validates that the caller passes a non-NULL pointer to a buffer for `pInt` and that the buffer is initialized by the function before it returns.

Example: The `_Out_opt_` Annotation

`_Out_opt_` is the same as `_Out_`, except that the parameter is allowed to be NULL and, therefore, the function should check for this.

C++

```
void GoodOutOptCallee(_Out_opt_ int *pInt)
{
    if (pInt != NULL) {
        *pInt = 5;
    }
}

void BadOutOptCallee(_Out_opt_ int *pInt)
{
    *pInt = 5; // Dereferencing NULL pointer 'pInt'
}
```

```
void OutOptCaller()
{
    int *pInt = NULL;
    GoodOutOptCallee(pInt);
    BadOutOptCallee(pInt);
}
```

Visual Studio Code Analysis validates that this function checks for `NULL` before `pInt` is dereferenced, and if `pInt` is not `NULL`, that the buffer is initialized by the function before it returns.

Example: The `_Inout_` Annotation

`_Inout_` is used to annotate a pointer parameter that may be changed by the function. The pointer must point to valid initialized data before the call, and even if it changes, it must still have a valid value on return. The annotation specifies that the function may freely read from and write to the one-element buffer. The caller must provide the buffer and initialize it.

ⓘ Note

Like `_Out_`, `_Inout_` must apply to a modifiable value.

C++

```
void InOutCallee(_Inout_ int *pInt)
{
    int i = *pInt;
    *pInt = 6;
}

void InOutCaller()
{
    int *pInt = new int;
    *pInt = 5;
    InOutCallee(pInt);
    delete pInt;
}

void BadInOutCaller()
{
    int *pInt = NULL;
    InOutCallee(pInt); // 'pInt' should not be NULL
}
```

Visual Studio Code Analysis validates that callers pass a non-NULL pointer to an initialized buffer for `pInt`, and that, before return, `pInt` is still non-NULL and the buffer is initialized.

Example: The `_Inout_opt_` Annotation

`_Inout_opt_` is the same as `_Inout_`, except that the input parameter is allowed to be NULL and, therefore, the function should check for this.

C++

```
void GoodInOutOptCallee(_Inout_opt_ int *pInt)
{
    if(pInt != NULL) {
        int i = *pInt;
        *pInt = 6;
    }
}

void BadInOutOptCallee(_Inout_opt_ int *pInt)
{
    int i = *pInt; // Dereferencing NULL pointer 'pInt'
    *pInt = 6;
}

void InOutOptCaller()
{
    int *pInt = NULL;
    GoodInOutOptCallee(pInt);
    BadInOutOptCallee(pInt);
}
```

Visual Studio Code Analysis validates that this function checks for NULL before it accesses the buffer, and if `pInt` is not NULL, that the buffer is initialized by the function before it returns.

Example: The `_Outptr_` Annotation

`_Outptr_` is used to annotate a parameter that's intended to return a pointer. The parameter itself should not be NULL, and the called function returns a non-NULL pointer in it and that pointer points to initialized data.

C++

```
void GoodOutPtrCallee(_Outptr_ int **pInt)
{
    int *pInt2 = new int;
```

```

*pInt2 = 5;

*pInt = pInt2;
}

void BadOutPtrCallee(_Outptr_ int **pInt)
{
    int *pInt2 = new int;
    // Did not initialize pInt buffer before returning!
    *pInt = pInt2;
}

void OutPtrCaller()
{
    int *pInt = NULL;
    GoodOutPtrCallee(&pInt);
    BadOutPtrCallee(&pInt);
}

```

Visual Studio Code Analysis validates that the caller passes a non-NULL pointer for `*pInt`, and that the buffer is initialized by the function before it returns.

Example: The `_Outptr_opt_` Annotation

`_Outptr_opt_` is the same as `_Outptr_`, except that the parameter is optional—the caller can pass in a NULL pointer for the parameter.

C++

```

void GoodOutPtrOptCallee(_Outptr_opt_ int **pInt)
{
    int *pInt2 = new int;
    *pInt2 = 6;

    if(pInt != NULL) {
        *pInt = pInt2;
    }
}

void BadOutPtrOptCallee(_Outptr_opt_ int **pInt)
{
    int *pInt2 = new int;
    *pInt2 = 6;
    *pInt = pInt2; // Dereferencing NULL pointer 'pInt'
}

void OutPtrOptCaller()
{
    int **ppInt = NULL;
    GoodOutPtrOptCallee(ppInt);
}

```

```
    BadOutPtrOptCallee(ppInt);  
}
```

Visual Studio Code Analysis validates that this function checks for NULL before `*pInt` is dereferenced, and that the buffer is initialized by the function before it returns.

Example: The `_Success_` Annotation in Combination with `_Out_`

Annotations can be applied to most objects. In particular, you can annotate a whole function. One of the most obvious characteristics of a function is that it can succeed or fail. But like the association between a buffer and its size, C/C++ cannot express function success or failure. By using the `_Success_` annotation, you can say what success for a function looks like. The parameter to the `_Success_` annotation is just an expression that when it is true indicates that the function has succeeded. The expression can be anything that the annotation parser can handle. The effects of the annotations after the function returns are only applicable when the function succeeds. This example shows how `_Success_` interacts with `_out_` to do the right thing. You can use the keyword `return` to represent the return value.

C++

```
_Success_(return != false) // Can also be stated as _Success_(return)  
bool GetValue(_Out_ int *pInt, bool flag)  
{  
    if(flag) {  
        *pInt = 5;  
        return true;  
    } else {  
        return false;  
    }  
}
```

The `_out_` annotation causes Visual Studio Code Analysis to validate that the caller passes a non-NULL pointer to a buffer for `pInt`, and that the buffer is initialized by the function before it returns.

SAL Best Practice

Adding Annotations to Existing Code

SAL is a powerful technology that can help you improve the security and reliability of your code. After you learn SAL, you can apply the new skill to your daily work. In new code, you can use SAL-based specifications by design throughout; in older code, you can add annotations incrementally and thereby increase the benefits every time you update.

Microsoft public headers are already annotated. Therefore, we suggest that in your projects you first annotate leaf node functions and functions that call Win32 APIs to get the most benefit.

When Do I Annotate?

Here are some guidelines:

- Annotate all pointer parameters.
- Annotate value-range annotations so that Code Analysis can ensure buffer and pointer safety.
- Annotate locking rules and locking side effects. For more information, see [Annotating Locking Behavior](#).
- Annotate driver properties and other domain-specific properties.

Or you can annotate all parameters to make your intent clear throughout and to make it easy to check that annotations have been done.

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Best Practices and Examples](#)

Annotating function parameters and return values

Article • 08/03/2021

This article describes typical uses of annotations for simple function parameters—scalars, and pointers to structures and classes—and most kinds of buffers. This article also shows common usage patterns for annotations. For additional annotations that are related to functions, see [Annotating function behavior](#).

Pointer parameters

For the annotations in the following table, when a pointer parameter is annotated, the analyzer reports an error if the pointer is null. This annotation applies to pointers and to any data item that's pointed to.

Annotations and descriptions

- `_In_`

Annotates input parameters that are scalars, structures, pointers to structures and the like. Explicitly may be used on simple scalars. The parameter must be valid in pre-state and won't be modified.
- `_Out_`

Annotates output parameters that are scalars, structures, pointers to structures and the like. Don't apply this annotation to an object that can't return a value—for example, a scalar that's passed by value. The parameter doesn't have to be valid in pre-state but must be valid in post-state.
- `_Inout_`

Annotates a parameter that will be changed by the function. It must be valid in both pre-state and post-state, but is assumed to have different values before and after the call. Must apply to a modifiable value.
- `_In_z_`

A pointer to a null-terminated string that's used as input. The string must be valid in pre-state. Variants of `PSTR`, which already have the correct annotations, are preferred.

- `_Inout_z_`

A pointer to a null-terminated character array that will be modified. It must be valid before and after the call, but it's assumed the value has changed. The null terminator may be moved, but only the elements up to the original null terminator may be accessed.

- `_In_reads_(s)`

`_In_reads_bytes_(s)`

A pointer to an array, which is read by the function. The array is of size `s` elements, all of which must be valid.

The `_bytes_` variant gives the size in bytes instead of elements. Use this variant only when the size can't be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_In_reads_z_(s)`

A pointer to an array that is null-terminated and has a known size. The elements up to the null terminator—or `s` if there isn't a null terminator—must be valid in pre-state. If the size is known in bytes, scale `s` by the element size.

- `_In_reads_or_z_(s)`

A pointer to an array that is null-terminated or has a known size, or both. The elements up to the null terminator—or `s` if there isn't a null terminator—must be valid in pre-state. If the size is known in bytes, scale `s` by the element size. (Used for the `strn` family.)

- `_Out_writes_(s)`

`_Out_writes_bytes_(s)`

A pointer to an array of `s` elements (resp. bytes) that will be written by the function. The array elements don't have to be valid in pre-state, and the number of elements that are valid in post-state is unspecified. If there are annotations on the parameter type, they're applied in post-state. For example, consider the following code.

C++

```
typedef _Null_terminated_ wchar_t *PWSTR;
void MyStringCopy(_Out_writes_(size) PWSTR p1, _In_ size_t size, _In_
```

```
PWSTR p2);
```

In this example, the caller provides a buffer of `size` elements for `p1`. `MyStringCopy` makes some of those elements valid. More importantly, the `_Null_terminated_` annotation on `PWSTR` means that `p1` is null-terminated in post-state. In this way, the number of valid elements is still well-defined, but a specific element count isn't required.

The `_bytes_` variant gives the size in bytes instead of elements. Use this variant only when the size can't be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_Out_writes_z_(s)`

A pointer to an array of `s` elements. The elements don't have to be valid in pre-state. In post-state, the elements up through the null terminator—which must be present—must be valid. If the size is known in bytes, scale `s` by the element size.

- `_Inout_updates_(s)`

```
_Inout_updates_bytes_(s)
```

A pointer to an array, which is both read and written to in the function. It's of size `s` elements, and valid in pre-state and post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this variant only when the size can't be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_Inout_updates_z_(s)`

A pointer to an array that's null-terminated and has a known size. The elements up through the null terminator—which must be present—must be valid in both pre-state and post-state. The value in the post-state is presumed to be different from the value in the pre-state; that includes the location of the null terminator. If the size is known in bytes, scale `s` by the element size.

- `_Out_writes_to_(s,c)`

```
_Out_writes_bytes_to_(s,c)
```

```
_Out_writes_all_(s)
```

```
_Out_writes_bytes_all_(s)
```

A pointer to an array of `s` elements. The elements don't have to be valid in pre-state. In post-state, the elements up to the `c`-th element must be valid. The `_bytes_` variant can be used if the size is known in bytes rather than number of elements.

For example:

C++

```
void *memcpy(_Out_writes_bytes_all_(s) char *p1, _In_reads_bytes_(s)
char *p2, _In_ int s);
void *wordcpy(_Out_writes_all_(s) DWORD *p1, _In_reads_(s) DWORD *p2,
_In_ int s);
```

- `_Inout_updates_to_(s,c)`

`_Inout_updates_bytes_to_(s,c)`

A pointer to an array, which is both read and written by the function. It's of size `s` elements, all of which must be valid in pre-state, and `c` elements must be valid in post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this variant only when the size can't be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_Inout_updates_all_(s)`

`_Inout_updates_bytes_all_(s)`

A pointer to an array, which is both read and written by the function of size `s` elements. Defined as equivalent to:

```
_Inout_updates_to_(_Old_(s), _Old_(s)) _Inout_updates_bytes_to_(_Old_(s),
_Old_(s))
```

In other words, every element that exists in the buffer up to `s` in the pre-state is valid in the pre-state and post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this variant only when the size can't be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_In_reads_to_ptr_(p)`

A pointer to an array for which `p - _curr_` (that is, `p` minus `_curr_`) is a valid expression. The elements before `p` must be valid in pre-state.

For example:

C++

```
int ReadAllElements(_In_reads_to_ptr_(EndOfArray) const int *Array,  
const int *EndOfArray);
```

- `_In_reads_to_ptr_z_(p)`

A pointer to a null-terminated array for which expression `p - _curr_` (that is, `p` minus `_curr_`) is a valid expression. The elements before `p` must be valid in pre-state.

- `_Out_writes_to_ptr_(p)`

A pointer to an array for which `p - _curr_` (that is, `p` minus `_curr_`) is a valid expression. The elements before `p` don't have to be valid in pre-state and must be valid in post-state.

- `_Out_writes_to_ptr_z_(p)`

A pointer to a null-terminated array for which `p - _curr_` (that is, `p` minus `_curr_`) is a valid expression. The elements before `p` don't have to be valid in pre-state and must be valid in post-state.

Optional pointer parameters

When a pointer parameter annotation includes `_opt_`, it indicates that the parameter may be null. Otherwise, the annotation behaves the same as the version that doesn't include `_opt_`. Here is a list of the `_opt_` variants of the pointer parameter annotations:

`_In_opt_`
`_Out_opt_`
`_Inout_opt_`
`_In_opt_z_`
`_Inout_opt_z_`
`_In_reads_opt_`

```
_In_reads_bytes_opt_
_In_reads_opt_z_

_Out_writes_opt_
_Out_writes_opt_z_
_Inout_updates_opt_
_Inout_updates_bytes_opt_
_Inout_updates_opt_z_
_Out_writes_to_opt_
_Out_writes_bytes_to_opt_
_Out_writes_all_opt_
_Out_writes_bytes_all_opt_

_Inout_updates_to_opt_
_Inout_updates_bytes_to_opt_
_Inout_updates_all_opt_
_Inout_updates_bytes_all_opt_
_In_reads_to_ptr_opt_
_In_reads_to_ptr_opt_z_
_Out_writes_to_ptr_opt_
_Out_writes_to_ptr_opt_z_
```

Output pointer parameters

Output pointer parameters require special notation to disambiguate nullness on the parameter and the pointed-to location.

Annotations and descriptions

- `_Outptr_`

Parameter can't be null, and in the post-state the pointed-to location can't be null and must be valid.

- `_Outptr_opt_`

Parameter may be null, but in the post-state the pointed-to location can't be null and must be valid.

- `_Outptr_result_maybenull_`

Parameter can't be null, and in the post-state the pointed-to location can be null.

- `_Outptr_opt_result_maybenull_`

Parameter may be null, and in the post-state the pointed-to location can be null.

In the following table, additional substrings are inserted into the annotation name to further qualify the meaning of the annotation. The various substrings are `_z`, `_COM_`, `_buffer_`, `_bytebuffer_`, and `_to_`.

ⓘ Important

If the interface that you are annotating is COM, use the COM form of these annotations. Don't use the COM annotations with any other type interface.

- `_Outptr_result_z_`

`_Outptr_opt_result_z_`

`_Outptr_result_maybenull_z_`

`_Outptr_opt_result_maybenull_z_`

The returned pointer has the `_Null_terminated_` annotation.

- `_COM_Outptr_`

`_COM_Outptr_opt_`

`_COM_Outptr_result_maybenull_`

`_COM_Outptr_opt_result_maybenull_`

The returned pointer has COM semantics, which is why it carries an `_On_failure_` post-condition that the returned pointer is null.

- `_Outptr_result_buffer_(s)`

`_Outptr_result_bytebuffer_(s)`

`_Outptr_opt_result_buffer_(s)`

`_Outptr_opt_result_bytebuffer_(s)`

The returned pointer points to a valid buffer of size `s` elements or bytes.

- `_Outptr_result_buffer_to_(s, c)`
- `_Outptr_result_bytbuffer_to_(s, c)`
- `_Outptr_opt_result_buffer_to_(s,c)`
- `_Outptr_opt_result_bytbuffer_to_(s,c)`

The returned pointer points to a buffer of size `s` elements or bytes, of which the first `c` are valid.

Certain interface conventions presume that output parameters are nullified on failure. Except for explicitly COM code, the forms in the following table are preferred. For COM code, use the corresponding COM forms that are listed in the previous section.

- `_Result_nullonfailure_`

Modifies other annotations. The result is set to null if the function fails.

- `_Result_zeroonfailure_`

Modifies other annotations. The result is set to zero if the function fails.

- `_Outptr_result_nullonfailure_`

The returned pointer points to a valid buffer if the function succeeds, or null if the function fails. This annotation is for a non-optional parameter.

- `_Outptr_opt_result_nullonfailure_`

The returned pointer points to a valid buffer if the function succeeds, or null if the function fails. This annotation is for an optional parameter.

- `_Outref_result_nullonfailure_`

The returned pointer points to a valid buffer if the function succeeds, or null if the function fails. This annotation is for a reference parameter.

Output reference parameters

A common use of the reference parameter is for output parameters. For simple output reference parameters such as `int&`, `_out_` provides the correct semantics. However, when the output value is a pointer such as `int * &`, the equivalent pointer annotations like `_Outptr_ int **` don't provide the correct semantics. To concisely express the

semantics of output reference parameters for pointer types, use these composite annotations:

Annotations and descriptions

- `_Outref_`

Result must be valid in post-state and can't be null.

- `_Outref_result_maybenull_`

Result must be valid in post-state, but may be null in post-state.

- `_Outref_result_buffer_(s)`

Result must be valid in post-state and can't be null. Points to valid buffer of size `s` elements.

- `_Outref_result_bytebuffer_(s)`

Result must be valid in post-state and can't be null. Points to valid buffer of size `s` bytes.

- `_Outref_result_buffer_to_(s, c)`

Result must be valid in post-state and can't be null. Points to buffer of `s` elements, of which the first `c` are valid.

- `_Outref_result_bytebuffer_to_(s, c)`

Result must be valid in post-state and can't be null. Points to buffer of `s` bytes of which the first `c` are valid.

- `_Outref_result_buffer_all_(s)`

Result must be valid in post-state and can't be null. Points to valid buffer of size `s` valid elements.

- `_Outref_result_bytebuffer_all_(s)`

Result must be valid in post-state and can't be null. Points to valid buffer of `s` bytes of valid elements.

- `_Outref_result_buffer_maybenull_(s)`

Result must be valid in post-state, but may be null in post-state. Points to valid buffer of size `s` elements.

- `_Outref_result_bytebuffer_maybenull_(s)`

Result must be valid in post-state, but may be null in post-state. Points to valid buffer of size `s` bytes.

- `_Outref_result_buffer_to_maybenull_(s, c)`

Result must be valid in post-state, but may be null in post-state. Points to buffer of `s` elements, of which the first `c` are valid.

- `_Outref_result_bytebuffer_to_maybenull_(s,c)`

Result must be valid in post-state, but may be null in post state. Points to buffer of `s` bytes of which the first `c` are valid.

- `_Outref_result_buffer_all_maybenull_(s)`

Result must be valid in post-state, but may be null in post state. Points to valid buffer of size `s` valid elements.

- `_Outref_result_bytebuffer_all_maybenull_(s)`

Result must be valid in post-state, but may be null in post state. Points to valid buffer of `s` bytes of valid elements.

Return values

The return value of a function resembles an `_Out_` parameter but is at a different level of de-reference, and you don't have to consider the concept of the pointer to the result. For the following annotations, the return value is the annotated object—a scalar, a pointer to a struct, or a pointer to a buffer. These annotations have the same semantics as the corresponding `_Out_` annotation.

```
_Ret_z_
_Ret_writes_(s)
_Ret_writes_bytes_(s)
_Ret_writes_z_(s)
_Ret_writes_to_(s,c)
_Ret_writes_maybenull_(s)
```

```
_Ret_writes_to_maybenull_(s)
_Ret_writes_maybenull_z_(s)

_Ret_maybenull_
_Ret_maybenull_z_
_Ret_null_
_Ret_notnull_
_Ret_writes_bytes_to_
_Ret_writes_bytes_maybenull_
_Ret_writes_bytes_to_maybenull_
```

Format string parameters

- `_Printf_format_string_` Indicates that the parameter is a format string for use in a `printf` expression.

Example

C++

```
int MyPrintF(_Printf_format_string_ const wchar_t* format, ...)
{
    va_list args;
    va_start(args, format);
    int ret = vwprintf(format, args);
    va_end(args);
    return ret;
}
```

- `_Scanf_format_string_` Indicates that the parameter is a format string for use in a `scanf` expression.

Example

C++

```
int MyScanF(_Scanf_format_string_ const wchar_t* format, ...)
{
    va_list args;
    va_start(args, format);
    int ret = vwscanf(format, args);
    va_end(args);
    return ret;
}
```

- `_Scanf_s_format_string_` Indicates that the parameter is a format string for use in a `scanf_s` expression.

Example

C++

```
int MyScanF_s(_Scanf_s_format_string_ const wchar_t* format, ...)
{
    va_list args;
    va_start(args, format);
    int ret = vwscanf_s(format, args);
    va_end(args);
    return ret;
}
```

Other common annotations

Annotations and descriptions

- `_In_range_(low, hi)`

`_Out_range_(low, hi)`

`_Ret_range_(low, hi)`

`_Deref_in_range_(low, hi)`

`_Deref_out_range_(low, hi)`

`_Deref inout_range_(low, hi)`

`_Field_range_(low, hi)`

The parameter, field, or result is in the range (inclusive) from `low` to `hi`. Equivalent to `_Satisfies_(_Curr_ >= low && _Curr_ <= hi)` that is applied to the annotated object together with the appropriate pre-state or post-state conditions.

ⓘ Important

Although the names contain "in" and "out", the semantics of `_In_` and `_Out_` do **not** apply to these annotations.

- `_Pre_equal_to_(expr)`

```
_Post_equal_to_(expr)
```

The annotated value is exactly `expr`. Equivalent to `_Satisfies_(_Curr_ == expr)` that is applied to the annotated object together with the appropriate pre-state or post-state conditions.

- `_Struct_size_bytes_(size)`

Applies to a struct or class declaration. Indicates that a valid object of that type may be larger than the declared type, with the number of bytes being given by `size`. For example:

```
typedef _Struct_size_bytes_(nSize) struct MyStruct { size_t nSize; ... };
```

The buffer size in bytes of a parameter `pM` of type `MyStruct *` is then taken to be:

```
min(pM->nSize, sizeof(MyStruct))
```

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

Annotating function behavior

Article • 08/03/2021

In addition to annotating [function parameters and return values](#), you can annotate properties of the whole function.

Function annotations

The following annotations apply to the function as a whole and describe how it behaves or what it expects to be true.

Annotation	Description
<code>_Called_from_function_class_(name)</code>	<p>Not intended to stand alone; instead, it is a predicate to be used with the <code>_When_</code> annotation. For more information, see Specifying When and Where an Annotation Applies.</p> <p>The <code>name</code> parameter is an arbitrary string that also appears in a <code>_Function_class_</code> annotation in the declaration of some functions. <code>_Called_from_function_class_</code> returns nonzero if the function that is currently being analyzed is annotated by using <code>_Function_class_</code> that has the same value of <code>name</code>; otherwise, it returns zero.</p>
<code>_Check_return_</code>	Annotates a return value and states that the caller should inspect it. The checker reports an error if the function is called in a void context.
<code>_Function_class_(name)</code>	The <code>name</code> parameter is an arbitrary string that is designated by the user. It exists in a namespace that is distinct from other namespaces. A function, function pointer, or—most usefully—a function pointer type may be designated as belonging to one or more function classes.
<code>_Raises_SEH_exception_</code>	Annotates a function that always raises a structured exception handler (SEH) exception, subject to <code>_When_</code> and <code>_On_failure_</code> conditions. For more information, see Specifying When and Where an Annotation Applies .
<code>_Maybe_raises_SEH_exception_</code>	Annotates a function that may optionally raise an SEH exception, subject to <code>_When_</code> and <code>_On_failure_</code> conditions.

Annotation	Description
<code>_Must_inspect_result_</code>	Annotates any output value, including the return value, parameters, and globals. The analyzer reports an error if the value in the annotated object is not subsequently inspected. "Inspection" includes whether it is used in a conditional expression, is assigned to an output parameter or global, or is passed as a parameter. For return values, <code>_Must_inspect_result_</code> implies <code>_Check_return_</code> .
<code>_Use_decl_annotations_</code>	May be used on a function definition (also known as a function body) in place of the list of annotations in the header. When <code>_Use_decl_annotations_</code> is used, the annotations that appear on an in-scope header for the same function are used as if they are also present in the definition that has the <code>_Use_decl_annotations_</code> annotation.

Success/failure annotations

A function can fail, and when it does, its results may be incomplete or differ from the results when the function succeeds. The annotations in the following list provide ways to express the failure behavior. To use these annotations, you must enable them to determine success; therefore, a `_Success_` annotation is required. Notice that `NTSTATUS` and `HRESULT` already have a `_Success_` annotation built into them; however, if you specify your own `_Success_` annotation on `NTSTATUS` or `HRESULT`, it overrides the built-in annotation.

Annotation	Description
<code>_Always_(anno_list)</code>	Equivalent to <code>anno_list _On_failure_(anno_list)</code> ; that is, the annotations in <code>anno_list</code> apply whether or not the function succeeds.
<code>_On_failure_(anno_list)</code>	To be used only when <code>_Success_</code> is also used to annotate the function—either explicitly, or implicitly through <code>_Return_type_success_</code> on a typedef. When the <code>_On_failure_</code> annotation is present on a function parameter or return value, each annotation in <code>anno_list</code> (anno) behaves as if it were coded as <code>_When_(!expr, anno)</code> , where <code>expr</code> is the parameter to the required <code>_Success_</code> annotation. This means that the implied application of <code>_Success_</code> to all post-conditions does not apply for <code>_On_failure_</code> .

Annotation	Description
<code>_Return_type_success_(expr)</code>	May be applied to a typedef. Indicates that all functions that return that type and do not explicitly have <code>_Success_</code> are annotated as if they had <code>_Success_(expr)</code> . <code>_Return_type_success_</code> cannot be used on a function or a function pointer typedef.
<code>_Success_(expr)</code>	<p><code>expr</code> is an expression that yields an rvalue. When the <code>_Success_</code> annotation is present on a function declaration or definition, each annotation (<code>anno</code>) on the function and in post-condition behaves as if it were coded as <code>_When_(expr, anno)</code>. The <code>_Success_</code> annotation may be used only on a function, not on its parameters or return type. There can be at most one <code>_Success_</code> annotation on a function, and it cannot be in any <code>_When_</code>, <code>_At_</code>, or <code>_Group_</code>. For more information, see Specifying When and Where an Annotation Applies.</p>

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

Annotating Structs and Classes

Article • 10/17/2022

You can annotate struct and class members by using annotations that act like invariants—they are presumed to be true at any function call or function entry/exit that involves the enclosing structure as a parameter or a result value.

Struct and Class Annotations

- `_Field_range_(low, high)`

The field is in the range (inclusive) from `low` to `high`. Equivalent to

`_Satisfies_(_Curr_ >= low && _Curr_ <= high)` applied to the annotated object by using the appropriate pre or post conditions.

- `_Field_size_(size)`, `_Field_size_opt_(size)`, `_Field_size_bytes_(size)`,
`_Field_size_bytes_opt_(size)`

A field that has a writable size in elements (or bytes) as specified by `size`.

- `_Field_size_part_(size, count)`, `_Field_size_part_opt_(size, count)`,
`_Field_size_bytes_part_(size, count)`, `_Field_size_bytes_part_opt_(size, count)`

A field that has a writable size in elements (or bytes) as specified by `size`, and the `count` of those elements (bytes) that are readable.

- `_Field_size_full_(size)`, `_Field_size_full_opt_(size)`,
`_Field_size_bytes_full_(size)`, `_Field_size_bytes_full_opt_(size)`

A field that has both readable and writable size in elements (or bytes) as specified by `size`.

- `_Field_z_`

A field that has a null-terminated string.

- `_Struct_size_bytes_(size)`

Applies to struct or class declaration. Indicates that a valid object of that type may be larger than the declared type, with the number of bytes being specified by `size`. For example:

C++

```
typedef _Struct_size_bytes_(nSize)
struct MyStruct {
    size_t nSize;
    ...
};
```

The buffer size in bytes of a parameter `pM` of type `MyStruct *` is then taken to be:

C++

```
min(pM->nSize, sizeof(MyStruct))
```

Example

C++

```
#include <sal.h>

// This _Struct_size_bytes_ is equivalent to what below _Field_size_ means.
_Struct_size_bytes_(__builtin_offsetof(MyBuffer, buffer) + bufferSize *
sizeof(int))
struct MyBuffer
{
    static int MaxBufferSize;

    _Field_z_
    const char* name;

    int firstField;

    // ... other fields

    _Field_range_(1, MaxBufferSize)
    int bufferSize;

    _Field_size_(bufferSize)          // Preferred way - easier to read and
    maintain.
    int buffer[]; // Using C99 Flexible array member
};
```

Notes for this example:

- `_Field_z_` is equivalent to `_Null_terminated_`. `_Field_z_` for the `name` field specifies that the `name` field is a null-terminated string.

- `_Field_range_` for `bufferSize` specifies that the value of `bufferSize` should be within 1 and `MaxBufferSize` (both inclusive).
- The end results of the `_Struct_size_bytes_` and `_Field_size_` annotations are equivalent. For structures or classes that have a similar layout, `_Field_size_` is easier to read and maintain, because it has fewer references and calculations than the equivalent `_Struct_size_bytes_` annotation. `_Field_size_` doesn't require conversion to the byte size. If byte size is the only option, for example, for a void pointer field, `_Field_size_bytes_` can be used. If both `_Struct_size_bytes_` and `_Field_size_` exist, both will be available to tools. It is up to the tool what to do if the two annotations disagree.

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

Annotating Locking Behavior

Article • 08/25/2022

To avoid concurrency bugs in your multithreaded program, always follow an appropriate locking discipline and use SAL annotations.

Concurrency bugs are notoriously hard to reproduce, diagnose, and debug because they're non-deterministic. Reasoning about thread interleaving is difficult at best, and becomes impractical when you're designing a body of code that has more than a few threads. Therefore, it's good practice to follow a locking discipline in your multithreaded programs. For example, obeying a lock order while acquiring multiple locks helps avoid deadlocks, and acquiring the proper guarding lock before accessing a shared resource helps prevent race conditions.

Unfortunately, seemingly simple locking rules can be surprisingly hard to follow in practice. A fundamental limitation in today's programming languages and compilers is that they do not directly support the specification and analysis of concurrency requirements. Programmers have to rely on informal code comments to express their intentions about how they use locks.

Concurrency SAL annotations are designed to help you specify locking side effects, locking responsibility, data guardianship, lock order hierarchy, and other expected locking behavior. By making implicit rules explicit, SAL concurrency annotations provide a consistent way for you to document how your code uses locking rules. Concurrency annotations also enhance the ability of code analysis tools to find race conditions, deadlocks, mismatched synchronization operations, and other subtle concurrency errors.

General Guidelines

By using annotations, you can state the contracts that are implied by function definitions between implementations (callees) and clients (callers). You can also express invariants and other properties of the program that can further improve analysis.

SAL supports many different kinds of locking primitives—for example, critical sections, mutexes, spin locks, and other resource objects. Many concurrency annotations take a lock expression as a parameter. By convention, a lock is denoted by the path expression of the underlying lock object.

Some thread ownership rules to keep in mind:

- Spin locks are uncounted locks that have clear thread ownership.

- Mutexes and critical sections are counted locks that have clear thread ownership.
- Semaphores and events are counted locks that don't have clear thread ownership.

Locking Annotations

The following table lists the locking annotations.

Annotation	Description
<code>_Acquires_exclusive_lock_(expr)</code>	Annotates a function and indicates that in post state the function increments by one the exclusive lock count of the lock object that's named by <code>expr</code> .
<code>_Acquires_lock_(expr)</code>	Annotates a function and indicates that in post state the function increments by one the lock count of the lock object that's named by <code>expr</code> .
<code>_Acquires_nonreentrant_lock_(expr)</code>	The lock that's named by <code>expr</code> is acquired. An error is reported if the lock is already held.
<code>_Acquires_shared_lock_(expr)</code>	Annotates a function and indicates that in post state the function increments by one the shared lock count of the lock object that's named by <code>expr</code> .
<code>_Create_lock_level_(name)</code>	A statement that declares the symbol <code>name</code> to be a lock level so that it may be used in the annotations <code>_Has_Lock_level_</code> and <code>_Lock_level_order_</code> .

Annotation	Description
<code>_Has_lock_kind_(kind)</code>	<p>Annotates any object to refine the type information of a resource object. Sometimes a common type is used for different kinds of resources and the overloaded type isn't sufficient to distinguish the semantic requirements among various resources. Here's a list of pre-defined <code>kind</code> parameters:</p> <ul style="list-style-type: none"> <code>_Lock_kind_mutex_</code> Lock kind ID for mutexes. <code>_Lock_kind_event_</code> Lock kind ID for events. <code>_Lock_kind_semaphore_</code> Lock kind ID for semaphores. <code>_Lock_kind_spin_lock_</code> Lock kind ID for spin locks. <code>_Lock_kind_critical_section_</code> Lock kind ID for critical sections.
<code>_Has_lock_level_(name)</code>	<p>Annotates a lock object and gives it the lock level of <code>name</code>.</p>
<code>_Lock_level_order_(name1, name2)</code>	<p>A statement that gives the lock ordering between <code>name1</code> and <code>name2</code>. Locks that have level <code>name1</code> must be acquired before locks that have level <code>name2</code>.</p>
<code>_Post_same_lock_(expr1, expr2)</code>	<p>Annotates a function and indicates that in post state the two locks, <code>expr1</code> and <code>expr2</code>, are treated as if they're the same lock object.</p>
<code>_Releases_exclusive_lock_(expr)</code>	<p>Annotates a function and indicates that in post state the function decrements by one the exclusive lock count of the lock object that's named by <code>expr</code>.</p>
<code>_Releases_lock_(expr)</code>	<p>Annotates a function and indicates that in post state the function decrements by one the lock count of the lock object that's named by <code>expr</code>.</p>
<code>_Releases_nonreentrant_lock_(expr)</code>	<p>The lock that's named by <code>expr</code> is released. An error is reported if the lock isn't currently held.</p>
<code>_Releases_shared_lock_(expr)</code>	<p>Annotates a function and indicates that in post state the function decrements by one the shared lock count of the lock object that's named by <code>expr</code>.</p>

Annotation	Description
<code>_Requires_lock_held_(expr)</code>	Annotates a function and indicates that in pre state the lock count of the object that's named by <code>expr</code> is at least one.
<code>_Requires_lock_not_held_(expr)</code>	Annotates a function and indicates that in pre state the lock count of the object that's named by <code>expr</code> is zero.
<code>_Requires_no_locks_held_</code>	Annotates a function and indicates that the lock counts of all locks that are known to the checker are zero.
<code>_Requires_shared_lock_held_(expr)</code>	Annotates a function and indicates that in pre state the shared lock count of the object that's named by <code>expr</code> is at least one.
<code>_Requires_exclusive_lock_held_(expr)</code>	Annotates a function and indicates that in pre state the exclusive lock count of the object that's named by <code>expr</code> is at least one.

SAL Intrinsics For Unexposed Locking Objects

Certain lock objects aren't exposed by the implementation of the associated locking functions. The following table lists SAL intrinsic variables that enable annotations on functions that operate on those unexposed lock objects.

Annotation	Description
<code>_Global_cancel_spin_lock_</code>	Describes the cancel spin lock.
<code>_Global_critical_region_</code>	Describes the critical region.
<code>_Global_interlock_</code>	Describes interlocked operations.
<code>_Global_priority_region_</code>	Describes the priority region.

Shared Data Access Annotations

The following table lists the annotations for shared data access.

Annotation	Description
<code>_Guarded_by_(expr)</code>	Annotates a variable and indicates that whenever the variable is accessed, the lock count of the lock object that's named by <code>expr</code> is at least one.

Annotation	Description
<code>_Interlocked_</code>	Annotates a variable and is equivalent to <code>_Guarded_by_(_Global_interlock_)</code> .
<code>_Interlocked_operand_</code>	The annotated function parameter is the target operand of one of the various Interlocked functions. Those operands must have specific additional properties.
<code>_Write_guarded_by_(expr)</code>	Annotates a variable and indicates that whenever the variable is modified, the lock count of the lock object that's named by <code>expr</code> is at least one.

Smart Lock and RAII Annotations

Smart locks typically wrap native locks and manage their lifetime. The following table lists annotations that can be used with smart locks and RAII coding patterns with support for `move` semantics.

Annotation	Description
<code>_Analysis_assume_smart_lock_acquired_(lock)</code>	Tells the analyzer to assume that a smart lock has been acquired. This annotation expects a reference lock type as its parameter.
<code>_Analysis_assume_smart_lock_released_(lock)</code>	Tells the analyzer to assume that a smart lock has been released. This annotation expects a reference lock type as its parameter.
<code>_Moves_lock_(target, source)</code>	Describes a <code>move constructor</code> operation, which transfers lock state from the <code>source</code> object to the <code>target</code> . The <code>target</code> is considered a newly constructed object, so any state it had before is lost and replaced by the <code>source</code> state. The <code>source</code> is also reset to a clean state with no lock counts or aliasing target, but aliases pointing to it remain unchanged.
<code>_Replaces_lock_(target, source)</code>	Describes <code>move assignment operator</code> semantics where the target lock is released before transferring the state from the source. You can regard it as a combination of <code>_Moves_lock_(target, source)</code> preceded by a <code>_Releases_lock_(target)</code> .

Annotation	Description
<code>_Swaps_locks_(left, right)</code>	Describes the standard <code>swap</code> behavior, which assumes that objects <code>left</code> and <code>right</code> exchange their state. The state exchanged includes lock count and aliasing target, if present. Aliases that point to the <code>left</code> and <code>right</code> objects remain unchanged.
<code>_Detaches_lock_(detached, lock)</code>	Describes a scenario in which a lock wrapper type allows dissociation with its contained resource. It's similar to how <code>std::unique_ptr</code> works with its internal pointer: it allows programmers to extract the pointer and leave its smart pointer container in a clean state. Similar logic is supported by <code>std::unique_lock</code> and can be implemented in custom lock wrappers. The detached lock retains its state (lock count and aliasing target, if any), while the wrapper is reset to contain zero lock count and no aliasing target, while retaining its own aliases. There's no operation on lock counts (releasing and acquiring). This annotation behaves exactly as <code>_Moves_lock_</code> except that the detached argument should be <code>return</code> rather than <code>this</code> .

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

Specifying When and Where an Annotation Applies

Article • 08/03/2021

When an annotation is conditional, it may require other annotations to specify that to the analyzer. For example, if a function has a variable that can be either synchronous or asynchronous, the function behaves as follows: In the synchronous case it always eventually succeeds, but in the asynchronous case it reports an error if it can't succeed immediately. When the function is called synchronously, checking the result value provides no value to the code analyzer because it would not have returned. However, when the function is called asynchronously and the function result is not checked, a serious error could occur. This example illustrates a situation in which you could use the `_When_` annotation—described later in this article—to enable checking.

Structural Annotations

To control when and where annotations apply, use the following structural annotations.

Annotation	Description
<code>_At_(expr, anno-list)</code>	<code>expr</code> is an expression that yields an lvalue. The annotations in <code>anno-list</code> are applied to the object that is named by <code>expr</code> . For each annotation in <code>anno-list</code> , <code>expr</code> is interpreted in pre-condition if the annotation is interpreted in pre-condition, and in post-condition if the annotation is interpreted in post-condition.
<code>_At_buffer_(expr, iter, elem-count, anno-list)</code>	<code>expr</code> is an expression that yields an lvalue. The annotations in <code>anno-list</code> are applied to the object that is named by <code>expr</code> . For each annotation in <code>anno-list</code> , <code>expr</code> is interpreted in pre-condition if the annotation is interpreted in precondition, and in post-condition if the annotation is interpreted in post-condition. <code>iter</code> is the name of a variable that is scoped to the annotation (inclusive of <code>anno-list</code>). <code>iter</code> has an implicit type <code>long</code> . Identically named variables in any enclosing scope are hidden from evaluation. <code>elem-count</code> is an expression that evaluates to an integer.
<code>_Group_(anno-list)</code>	The annotations in <code>anno-list</code> are all considered to have any qualifier that applies to the group annotation that is applied to each annotation.

Annotation	Description
<code>_When_(expr, anno-list)</code>	<p><code>expr</code> is an expression that can be converted to <code>bool</code>. When it is non-zero (<code>true</code>), the annotations that are specified in <code>anno-list</code> are considered applicable.</p>

By default, for each annotation in `anno-list`, `expr` is interpreted as using the input values if the annotation is a precondition, and as using the output values if the annotation is a post-condition. To override the default, you can use the `_Old_` intrinsic when you evaluate a post-condition to indicate that input values should be used. **Note:** Different annotations might be enabled as a consequence of using `_When_` if a mutable value—for example, `*pLength`—is involved because the evaluated result of `expr` in precondition may differ from its evaluated result in post-condition.

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

Intrinsic Functions

Article • 08/03/2021

An expression in SAL can be a C/C++ expression provided that it is an expression that does not have side effects—for example, `++`, `--`, and function calls all have side effects in this context. However, SAL does provide some function-like objects and some reserved symbols that can be used in SAL expressions. These are referred to as *intrinsic functions*.

General Purpose

The following intrinsic function annotations provide general utility for SAL.

Annotation	Description
<code>_Curr_</code>	A synonym for the object that is currently being annotated. When the <code>_At_</code> annotation is in use, <code>_Curr_</code> is the same as the first parameter to <code>_At_</code> . Otherwise, it is the parameter or the entire function/return value with which the annotation is lexically associated.
<code>_Inexpressible_(expr)</code>	Expresses a situation where the size of a buffer is too complex to represent by using an annotation expression—for example, when it is computed by scanning an input data set and then counting selected members.
<code>_Nullterm_length_(param)</code>	<code>param</code> is the number of elements in the buffer up to but not including a null terminator. It may be applied to any buffer of non-aggregate, non-void type.
<code>_Old_(expr)</code>	When it is evaluated in precondition, <code>_Old_</code> returns the input value <code>expr</code> . When it is evaluated in post-condition, it returns the value <code>expr</code> as it would have been evaluated in precondition.
<code>_Param_(n)</code>	The <code>n</code> th parameter to a function, counting from 1 to <code>n</code> , and <code>n</code> is a literal integral constant. If the parameter is named, this annotation is identical to accessing the parameter by name. Note: <code>n</code> may refer to the positional parameters that are defined by an ellipsis, or may be used in function prototypes where names are not used.
<code>return</code>	The C/C++ reserved keyword <code>return</code> can be used in a SAL expression to indicate the return value of a function. The value is only available in post state; it is a syntax error to use it in pre state.

String Specific

The following intrinsic function annotations enable manipulation of strings. All four of these functions serve the same purpose: to return the number of elements of the type that is found before a null terminator. The differences are the kinds of data in the elements that are referred to. Note that if you want to specify the length of a null-terminated buffer that is not composed of characters, use the `_Nullterm_length_(param)` annotation from the previous section.

Annotation	Description
<code>_String_length_(param)</code>	<code>param</code> is the number of elements in the string up to but not including a null terminator. This annotation is reserved for string-of-character types.
<code>strlen(param)</code>	<code>param</code> is the number of elements in the string up to but not including a null terminator. This annotation is reserved for use on character arrays and resembles the C Runtime function strlen() .
<code>wcslen(param)</code>	<code>param</code> is the number of elements in the string up to (but not including) a null terminator. This annotation is reserved for use on wide character arrays and resembles the C Runtime function wcslen() .

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Best Practices and Examples](#)

Best practices and examples (SAL)

Article • 03/31/2023

Here are some ways to get the most out of the Source Code Annotation Language (SAL) and avoid some common problems.

In

If the function is supposed to write to the element, use `_Inout_` instead of `_In_`. This is relevant in cases of automated conversion from older macros to SAL. Prior to SAL, many programmers used macros as comments—macros that were named `IN`, `OUT`, `IN_OUT`, or variants of these names. Although we recommend that you convert these macros to SAL, we also urge you to be careful when you convert them because the code might have changed since the original prototype was written and the old macro might no longer reflect what the code does. Be especially careful about the `OPTIONAL` comment macro because it's frequently placed incorrectly—for example, on the wrong side of a comma.

C++

```
#include <sal.h>

// Incorrect
void Func1(_In_ int *p1)
{
    if (p1 == NULL)
        return;

    *p1 = 1;
}

// Correct
// _Out_opt_ because the function tolerates NULL as a valid argument, i.e.
// no error is returned. If the function didn't check p1 for NULL, then
// _Out_ would be the better choice
void Func2(_Out_opt_ PCHAR p1)
{
    if (p1 == NULL)
        return;

    *p1 = 1;
}
```

opt

If the caller isn't allowed to pass in a null pointer, use `_In_` or `_Out_` instead of `_In_opt_` or `_Out_opt_`. This applies even to a function that checks its parameters and returns an error if it's `NULL` when it shouldn't be. Although having a function check its parameter for unexpected `NULL` and return gracefully is a good defensive coding practice, it doesn't mean that the parameter annotation can be of an optional type (`_*Xxx*_opt_`).

C++

```
#include <sal.h>

// Incorrect
void Func1(_Out_opt_ int *p1)
{
    *p = 1;
}

// Correct
void Func2(_Out_ int *p1)
{
    *p = 1;
}
```

_Pre_defensive_ and **_Post_defensive_**

If a function appears at a trust boundary, we recommend that you use the `_Pre_defensive_` annotation. The "defensive" modifier modifies certain annotations to indicate that, at the point of call, the interface should be checked strictly, but in the implementation body it should assume that incorrect parameters might be passed. In that case, `_In_ _Pre_defensive_` is preferred at a trust boundary to indicate that although a caller gets an error if it attempts to pass `NULL`, the function body is analyzed as if the parameter might be `NULL`, and any attempts to dereference the pointer without first checking it for `NULL` are flagged. A `_Post_defensive_` annotation is also available, for use in callbacks where the trusted party is assumed to be the caller and the untrusted code is the called code.

_Out_writes_

The following example demonstrates a common misuse of `_Out_writes_`.

C++

```
#include <sal.h>
```

```
// Incorrect
void Func1(_Out_writes_(size) CHAR *pb,
           DWORD size
);
```

The annotation `_Out_writes_` signifies that you have a buffer. It has `cb` bytes allocated, with the first byte initialized on exit. This annotation isn't strictly wrong and it's helpful to express the allocated size. However, it doesn't tell how many elements the function initializes.

The next example shows three correct ways to fully specify the exact size of the initialized portion of the buffer.

C++

```
#include <sal.h>

// Correct
void Func1(_Out_writes_to_(size, *pCount) CHAR *pb,
           DWORD size,
           PDWORD pCount
);

void Func2(_Out_writes_all_(size) CHAR *pb,
           DWORD size
);

void Func3(_Out_writes_(size) PSTR pb,
           DWORD size
);
```

Out PSTR

The use of `_Out_ PSTR` is almost always wrong. This combination is interpreted as having an output parameter that points to a character buffer and the buffer is null-terminated.

C++

```
#include <sal.h>

// Incorrect
void Func1(_Out_ PSTR pFileName, size_t n);

// Correct
void Func2(_Out_writes_(n) PSTR wszFileName, size_t n);
```

An annotation like `_In_ PCSTR` is common and useful. It points to an input string that has null termination because the precondition of `_In_` allows the recognition of a null-terminated string.

`_In_ WCHAR* p`

`_In_ WCHAR* p` says that there's an input pointer `p` that points to one character. However, in most cases, this is probably not the specification that is intended. Instead, what is probably intended is the specification of a null-terminated array; to do that, use `_In_ PWSTR`.

C++

```
#include <sal.h>

// Incorrect
void Func1(_In_ WCHAR* wszFileName);

// Correct
void Func2(_In_ PWSTR wszFileName);
```

Missing the proper specification of null termination is common. Use the appropriate `STR` version to replace the type, as shown in the following example.

C++

```
#include <sal.h>
#include <string.h>

// Incorrect
BOOL StrEquals1(_In_ PCHAR p1, _In_ PCHAR p2)
{
    return strcmp(p1, p2) == 0;
}

// Correct
BOOL StrEquals2(_In_ PSTR p1, _In_ PSTR p2)
{
    return strcmp(p1, p2) == 0;
}
```

`_Out_range_`

If the parameter is a pointer and you want to express the range of the value of the element that's pointed to by the pointer, use `_Deref_out_range_` instead of `_Out_range_`.

In the following example, the range of *pcbFilled is expressed, not pcbFilled.

C++

```
#include <sal.h>

// Incorrect
void Func1(
    _Out_writes_bytes_to_(cbSize, *pcbFilled) BYTE *pb,
    DWORD cbSize,
    _Out_range_(0, cbSize) DWORD *pcbFilled
);

// Correct
void Func2(
    _Out_writes_bytes_to_(cbSize, *pcbFilled) BYTE *pb,
    DWORD cbSize,
    _Deref_out_range_(0, cbSize) _Out_ DWORD *pcbFilled
);
```

`_Deref_out_range_(0, cbSize)` isn't strictly required for some tools because it can be inferred from `_Out_writes_to_(cbSize,*pcbFilled)`, but it's shown here for completeness.

Wrong context in `_When_`

Another common mistake is to use post-state evaluation for preconditions. In the following example, `_Requires_lock_held_` is a precondition.

C++

```
#include <sal.h>

// Incorrect
_When_(return == 0, _Requires_lock_held_(p->cs))
int Func1(_In_ MyData *p, int flag);

// Correct
_When_(flag == 0, _Requires_lock_held_(p->cs))
int Func2(_In_ MyData *p, int flag);
```

The expression `return` refers to a post-state value that isn't available in pre-state.

TRUE in `_Success_`

If the function succeeds when the return value is nonzero, use `return != 0` as the success condition instead of `return == TRUE`. Nonzero doesn't necessarily mean equivalence to the actual value that the compiler provides for `TRUE`. The parameter to `_Success_` is an expression, and the following expressions are evaluated as equivalent: `return != 0`, `return != false`, `return != FALSE`, and `return` with no parameters or comparisons.

C++

```
// Incorrect
_Success_(return == TRUE) _Acquires_lock_(*lpCriticalSection)
BOOL WINAPI TryEnterCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);

// Correct
_Success_(return != 0) _Acquires_lock_(*lpCriticalSection)
BOOL WINAPI TryEnterCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);
```

Reference variable

For a reference variable, the previous version of SAL used the implied pointer as the annotation target and required the addition of a `_deref` to annotations that attached to a reference variable. This version uses the object itself and doesn't require `_Deref_`.

C++

```
#include <sal.h>

// Incorrect
void Func1(
    _Out_writes_bytes_all_(cbSize) BYTE *pb,
    _Deref_ _Out_range_(0, 2) _Out_ DWORD &cbSize
);

// Correct
void Func2(
    _Out_writes_bytes_all_(cbSize) BYTE *pb,
    _Out_range_(0, 2) _Out_ DWORD &cbSize
);
```

Annotations on return values

The following example shows a common problem in return value annotations.

C++

```
#include <sal.h>

// Incorrect
_Out_opt_ void *MightReturnNullPtr1();

// Correct
_Ret_maybenull_ void *MightReturnNullPtr2();
```

In this example, `_out_opt_` says that the pointer might be `NULL` as part of the precondition. However, preconditions can't be applied to the return value. In this case, the correct annotation is `_Ret_maybenull_`.

See also

[Using SAL annotations to reduce C/C++ code defects](#)

[Understanding SAL](#)

[Annotating function parameters and return values](#)

[Annotating function behavior](#)

[Annotating structs and classes](#)

[Annotating locking behavior](#)

[Specifying when and where an annotation applies](#)

[Intrinsic functions](#)

How to specify additional code information by using `_Analysis_assume_`

Article • 10/24/2023

You can provide hints to the code analysis tool for C/C++ code that help the analysis process and reduce warnings. To provide additional information, use the following function macro:

```
_Analysis_assume_( expr )
```

`expr` - any expression that is assumed to evaluate to true.

The code analysis tool assumes that the condition represented by the expression `expr` is true at the point where the function appears. And, it remains true until `expr` is altered, for example, by assignment to a variable.

ⓘ Note

`_Analysis_assume_` does not impact code optimization. Outside the code analysis tool, `_Analysis_assume_` is defined as a no-op.

Example

The following code uses `_Analysis_assume_` to correct the code analysis warning [C6388](#):

C++

```
#include <windows.h>
#include <sal.h>

// Requires pc to be null.
void f(_Pre_null_ char* pc);

// Calls free and sets ch to null.
void FreeAndNull(char** ch);

void test()
{
    char* pc = (char*)malloc(5);
    FreeAndNull(&pc);
    _Analysis_assume_(pc == NULL);
```

```
f(pc);  
}
```

`_Analysis_assume_` should be used as a last resort. We should first try to make the contracts of the functions more precise. In this case we could improve the contract of `FreeAndNull` instead of using `_Analysis_assume_`:

C++

```
#include <windows.h>  
#include <sal.h>  
  
// Requires pc to be null.  
void f(_Pre_null_ char* pc);  
  
// Calls free and sets ch to null.  
_At_(*ch, _Post_null_)  
void FreeAndNull(char** ch);  
  
void test()  
{  
    char* pc = (char*)malloc(5);  
    FreeAndNull(&pc);  
    f(pc);  
}
```

See also

- [_assume](#)

C++ Core Guidelines checker reference

Article • 10/07/2022

This section lists C++ Core Guidelines Checker warnings. For information about Code Analysis, see [/analyze \(Code Analysis\)](#) and [Quick Start: Code Analysis for C/C++](#).

ⓘ Note

Some warnings belong to more than one group, and not all warnings have a complete reference topic.

OWNER_POINTER Group

[C26402 DONT_HEAP_ALLOCATE_MOVABLE_RESULT](#)

Return a scoped object instead of a heap-allocated if it has a move constructor. See [C++ Core Guidelines R.3 ↗](#).

[C26403 RESET_OR_DELETE_OWNER](#)

Reset or explicitly delete an owner<T> pointer '*variable*'. See [C++ Core Guidelines R.3 ↗](#).

[C26404 DONT_DELETE_INVALID](#)

Do not delete an owner<T> that may be in invalid state. See [C++ Core Guidelines R.3 ↗](#).

[C26405 DONT_ASSIGN_TO_VALID](#)

Do not assign to an owner<T> that may be in valid state. See [C++ Core Guidelines R.3 ↗](#).

[C26406 DONT_ASSIGN_RAW_TO_OWNER](#)

Do not assign a raw pointer to an owner<T>. See [C++ Core Guidelines R.3 ↗](#).

[C26407 DONT_HEAP_ALLOCATE_UNNECESSARILY](#)

Prefer scoped objects, don't heap-allocate unnecessarily. See [C++ Core Guidelines R.5 ↗](#).

[C26429 USE_NOTNULL](#)

Symbol '*symbol*' is never tested for nullness, it can be marked as `not_null`. See [C++ Core Guidelines F.23 ↗](#).

[C26430 TEST_ON_ALL_PATHS](#)

Symbol '*symbol*' is not tested for nullness on all paths. See [C++ Core Guidelines F.23 ↗](#).

C26431 DONT_TEST_NOTNULL

The type of expression '*expr*' is already `gsl::not_null`. Do not test it for nullness. See [C++ Core Guidelines F.23](#).

RAW_POINTER Group

C26400 NO_RAW_POINTER_ASSIGNMENT

Do not assign the result of an allocation or a function call with an `owner<T>` return value to a raw pointer; use `owner<T>` instead. See [C++ Core Guidelines I.11](#).

C26401 DONT_DELETE_NON_OWNER

Do not delete a raw pointer that is not an `owner<T>`. See [C++ Core Guidelines I.11](#).

C26402 DONT_HEAP_ALLOCATE_MOVABLE_RESULT

Return a scoped object instead of a heap-allocated if it has a move constructor. See [C++ Core Guidelines R.3](#).

C26408 NO_MALLOC_FREE

Avoid `malloc()` and `free()`, prefer the `nothrow` version of `new` with `delete`. See [C++ Core Guidelines R.10](#).

C26409 NO_NEW_DELETE

Avoid calling `new` and `delete` explicitly, use `std::make_unique<T>` instead. See [C++ Core Guidelines R.11](#).

C26429 USE_NOTNULL

Symbol '*symbol*' is never tested for nullness, it can be marked as `not_null`. See [C++ Core Guidelines F.23](#).

C26430 TEST_ON_ALL_PATHS

Symbol '*symbol*' is not tested for nullness on all paths. See [C++ Core Guidelines F.23](#).

C26431 DONT_TEST_NOTNULL

The type of expression '*expr*' is already `gsl::not_null`. Do not test it for nullness. See [C++ Core Guidelines F.23](#).

C26481 NO_POINTER_ARITHMETIC

Don't use pointer arithmetic. Use `span` instead. See [C++ Core Guidelines Bounds.1](#).

C26485 NO_ARRAY_TO_POINTER_DECAY

Expression '*expr*': No array to pointer decay. See [C++ Core Guidelines Bounds.3](#).

UNIQUE_POINTER Group

C26410 NO_REF_TO_CONST_UNIQUE_PTR

The parameter '*parameter*' is a reference to `const` unique pointer, use `const T*` or `const T&` instead. See [C++ Core Guidelines R.32](#).

C26411 NO_REF_TO_UNIQUE_PTR

The parameter '*parameter*' is a reference to unique pointer and it is never reassigned or reset, use `T*` or `T&` instead. See [C++ Core Guidelines R.33](#).

C26414 RESET_LOCAL_SMART_PTR

Move, copy, reassign, or reset a local smart pointer '*symbol*'. See [C++ Core Guidelines R.5](#).

C26415 SMART_PTR_NOT_NEEDED

Smart pointer parameter '*symbol*' is used only to access contained pointer. Use `T*` or `T&` instead. See [C++ Core Guidelines R.30](#).

SHARED_POINTER Group

C26414 RESET_LOCAL_SMART_PTR

Move, copy, reassign, or reset a local smart pointer '*symbol*'. See [C++ Core Guidelines R.5](#).

C26415 SMART_PTR_NOT_NEEDED

Smart pointer parameter '*symbol*' is used only to access contained pointer. Use `T*` or `T&` instead. See [C++ Core Guidelines R.30](#).

C26416 NO_RVALUE_REF_SHARED_PTR

Shared pointer parameter '*symbol*' is passed by rvalue reference. Pass by value instead. See [C++ Core Guidelines R.34](#).

C26417 NO_LVALUE_REF_SHARED_PTR

Shared pointer parameter '*symbol*' is passed by reference and not reset or reassigned. Use `T*` or `T&` instead. See [C++ Core Guidelines R.35](#).

C26418 NO_VALUE_OR_CONST_REF_SHARED_PTR

Shared pointer parameter '*symbol*' is not copied or moved. Use `T*` or `T&` instead. See [C++ Core Guidelines R.36](#).

DECLARATION Group

C26426 NO_GLOBAL_INIT_CALLS

Global initializer calls a non-constexpr function '*symbol*'. See [C++ Core Guidelines](#)

I.22 ↴ .

C26427 NO_GLOBAL_INIT_EXTERNNS

Global initializer accesses extern object '*symbol*'. See C++ Core Guidelines I.22 ↴ .

C26444 NO_UNNAMED_RAIIL_OBJECTS

Avoid unnamed objects with custom construction and destruction. See ES.84: Don't (try to) declare a local variable with no name ↴ .

CLASS Group

C26432 DEFINE_OR_DELETE_SPECIAL_OPS

If you define or delete any default operation in the type '*symbol*', define or delete them all. See C++ Core Guidelines C.21 ↴ .

C26433 OVERRIDE_EXPLICITLY

Function '*symbol*' should be marked with 'override'. See C.128: Virtual functions should specify exactly one of virtual, override, or final ↴ .

C26434 DONT_HIDE_METHODS

Function '*symbol_1*' hides a non-virtual function '*symbol_2*'. See C++ Core Guidelines C.128 ↴ .

C26435 SINGLE_VIRTUAL_SPECIFICATION

Function '*symbol*' should specify exactly one of 'virtual', 'override', or 'final'. See C.128: Virtual functions should specify exactly one of virtual, override, or final ↴ .

C26436 NEED_VIRTUAL_DTOR

The type '*symbol*' with a virtual function needs either public virtual or protected non-virtual destructor. See C++ Core Guidelines C.35 ↴ .

C26443 NO_EXPLICIT_DTOR_OVERRIDE

Overriding destructor should not use explicit 'override' or 'virtual' specifiers. See C.128: Virtual functions should specify exactly one of virtual, override, or final ↴ .

STYLE Group

C26438 NO_GOTO

Avoid goto. See C++ Core Guidelines ES.76 ↴ .

FUNCTION Group

C26439 SPECIAL_NOEXCEPT

This kind of function may not throw. Declare it `noexcept`. See C++ Core Guidelines F.6 [↗](#).

C26440 DECLARE_NOEXCEPT

Function '*symbol*' can be declared `noexcept`. See C++ Core Guidelines F.6 [↗](#).

C26447 DONT_THROW_IN_NOEXCEPT

The function is declared `noexcept` but calls a function which may throw exceptions. See C++ Core Guidelines: F.6: If your function may not throw, declare it `noexcept` [↗](#).

CONCURRENCY Group

C26441 NO_UNNAMED GUARDS

Guard objects must be named. See C++ Core Guidelines cp.44 [↗](#).

CONST Group

C26460 USE_CONST_REFERENCE_ARGUMENTS

The reference argument '*argument*' for function '*function*' can be marked as `const`. See C++ Core Guidelines con.3 [↗](#).

C26461 USE_CONST_POINTER_ARGUMENTS:

The pointer argument '*argument*' for function '*function*' can be marked as a pointer to `const`. See C++ Core Guidelines con.3 [↗](#).

C26462 USE_CONST_POINTER_FOR_VARIABLE

The value pointed to by '*variable*' is assigned only once, mark it as a pointer to `const`. See C++ Core Guidelines con.4 [↗](#).

C26463 USE_CONST_FOR_ELEMENTS

The elements of array '*array*' are assigned only once, mark elements `const`. See C++ Core Guidelines con.4 [↗](#).

C26464 USE_CONST_POINTER_FOR_ELEMENTS

The values pointed to by elements of array '*array*' are assigned only once, mark elements as pointer to `const`. See C++ Core Guidelines con.4 [↗](#).

C26496 USE_CONST_FOR_VARIABLE

The variable '*variable*' is assigned only once, mark it as `const`. See C++ Core Guidelines con.4 [↗](#).

C26497 USE_CONSTEXPR_FOR_FUNCTION

This function *function* could be marked `constexpr` if compile-time evaluation is desired. See [C++ Core Guidelines F.4](#).

C26498 USE_CONSTEXPR_FOR_FUNCTIONCALL

This function call *function* can use `constexpr` if compile-time evaluation is desired. See [C++ Core Guidelines con.5](#).

TYPE Group

C26437 DONT_SLICE

Do not slice. See [C++ Core Guidelines ES.63](#).

C26465 NO_CONST_CAST_UNNECESSARY

Don't use `const_cast` to cast away `const`. `const_cast` is not required; constness or volatility is not being removed by this conversion. See [C++ Core Guidelines Type.3](#).

C26466 NO_STATIC_DOWNCASE_POLYMORPHIC

Don't use `static_cast` downcasts. A cast from a polymorphic type should use `dynamic_cast`. See [C++ Core Guidelines Type.2](#).

C26471 NO_REINTERPRET_CAST_FROM_VOID_PTR

Don't use `reinterpret_cast`. A cast from `void*` can use `static_cast`. See [C++ Core Guidelines Type.1](#).

C26472 NO_CASTS_FOR_ARITHMETIC_CONVERSION

Don't use a `static_cast` for arithmetic conversions. Use brace initialization, `gsl::narrow_cast`, or `gsl::narrow`. See [C++ Core Guidelines Type.1](#).

C26473 NO_IDENTITY_CAST

Don't cast between pointer types where the source type and the target type are the same. See [C++ Core Guidelines Type.1](#).

C26474 NO_IMPLICIT_CAST

Don't cast between pointer types when the conversion could be implicit. See [C++ Core Guidelines Type.1](#).

C26475 NO_FUNCTION_STYLE_CASTS

Do not use function style C-casts. See [C++ Core Guidelines ES.49](#).

C26490 NO_REINTERPRET_CAST

Don't use `reinterpret_cast`. See [C++ Core Guidelines Type.1](#).

C26491 NO_STATIC_DOWNCAST

Don't use `static_cast` downcasts. See [C++ Core Guidelines Type.2 ↗](#).

C26492 NO_CONST_CAST

Don't use `const_cast` to cast away `const`. See [C++ Core Guidelines Type.3 ↗](#).

C26493 NO_CSTYLE_CAST

Don't use C-style casts. See [C++ Core Guidelines Type.4 ↗](#).

C26494 VAR_USE_BEFORE_INIT

Variable '*variable*' is uninitialized. Always initialize an object. See [C++ Core Guidelines Type.5 ↗](#).

C26495 MEMBER_UNINIT

Variable '*variable*' is uninitialized. Always initialize a member variable. See [C++ Core Guidelines Type.6 ↗](#).

BOUNDS Group

C26446 USE GSL_AT

Prefer to use `gsl::at()` instead of unchecked subscript operator. See [C++ Core Guidelines: Bounds.4: Don't use standard-library functions and types that are not bounds-checked ↗](#).

C26481 NO_POINTER_ARITHMETIC

Don't use pointer arithmetic. Use span instead. See [C++ Core Guidelines Bounds.1 ↗](#)

C26482 NO_DYNAMIC_ARRAY_INDEXING

Only index into arrays using constant expressions. See [C++ Core Guidelines Bounds.2 ↗](#)

C26483 STATIC_INDEX_OUT_OF_RANGE

Value *value* is outside the bounds (0, *bound*) of variable '*variable*'. Only index into arrays using constant expressions that are within bounds of the array. See [C++ Core Guidelines Bounds.2 ↗](#)

C26485 NO_ARRAY_TO_POINTER_DECAY

Expression '*expr*': No array to pointer decay. See [C++ Core Guidelines Bounds.3 ↗](#)

GSL Group

C26445 NO_SPAN_REF

A reference to `gsl::span` or `std::string_view` may be an indication of a lifetime issue.

See C++ Core Guidelines [GSL.view: Views](#)

C26446 USE_GSL_AT

Prefer to use `gsl::at()` instead of unchecked subscript operator. See [C++ Core Guidelines: Bounds.4: Don't use standard-library functions and types that are not bounds-checked](#).

C26448 USE_GSL_FINALLY

Consider using `gsl::finally` if final action is intended. See [C++ Core Guidelines: GSL.util: Utilities](#).

C26449 NO_SPAN_FROM_TEMPORARY

`gsl::span` or `std::string_view` created from a temporary will be invalid when the temporary is invalidated. See [C++ Core Guidelines: GSL.view: Views](#).

Deprecated Warnings

The following warnings are present in an early experimental rule set of the core guidelines checker, but are now deprecated and can be safely ignored. The warnings are superseded by warnings from the list above.

- 26412 DEREF_INVALID_POINTER
- 26413 DEREF_NULLPTR
- 26420 ASSIGN_NONOWNER_TO_EXPLICIT_OWNER
- 26421 ASSIGN_VALID_OWNER
- 26422 VALID_OWNER_LEAVE_SCOPE
- 26423 ALLOCATION_NOT_ASSIGNED_TO_OWNER
- 26424 VALID_ALLOCATION_LEAVE_SCOPE
- 26425 ASSIGNING_TO_STATIC
- 26499 NO_LIFETIME_TRACKING

See also

[Using the C++ Core Guidelines Checkers](#)

Warning C26400

Article • 10/07/2022

Do not assign the result of an allocation or a function call with an `owner<T>` return value to a raw pointer, use `owner<T>` instead (i.11)

Remarks

This check helps to enforce the *rule I.11: Never transfer ownership by a raw pointer (`T*`), which is a subset of the rule R.3: *A raw pointer (a `T*`) is non-owning.* Specifically, it warns on any call to `operator new`, which saves its result in a variable of raw pointer type. It also warns on calls to functions that return `gsl::owner<T>` if their results are assigned to raw pointers. The idea is that you should clearly state ownership of memory resources. For more information, see the [C++ Core Guidelines](#).

The easiest way to fix this warning is to use `auto` declaration if the resource is assigned immediately at the variable declaration. If this fix isn't possible, then we suggest that you use the type `gsl::owner<T>`. The `auto` declarations initialized with operator `new` are "owners" because we assume that the result of any allocation is implicitly an owner pointer. We transfer this assumption to the `auto` variable and treat it as `owner<T>`.

If this check flags a call to a function that returns `owner<T>`, it may be an indication of a legitimate bug in the code. Basically, it points to a place where the code leaks an explicit notion of ownership (and maybe the resource itself).

This rule currently checks only local variables. If you assign an allocation to a formal parameter, global variable, class member, and so on, it's not flagged. Appropriate coverage of such scenarios is planned for future work.

Code analysis name: NO_RAW_POINTER_ASSIGNMENT

Example 1: Simple allocation

C++

```
char *buffer = nullptr;
if (useCache)
    buffer = GetCache();
else
    buffer = new char[bufferSize]; // C26400
```

Example 2: Simple allocation (fixed with `gsl::owner<T>`)

C++

```
gsl::owner<char*> buffer = nullptr;
if (useCache)
    buffer = GetCache();
else
    buffer = new char[bufferSize]; // OK
```

Example 3: Simple allocation (fixed with `auto`)

C++

```
auto buffer = useCache ? GetCache() : new char[bufferSize]; // OK
```

Warning C26401

Article • 10/07/2022

Do not delete a raw pointer that is not an `owner<T>` (i.11)

Remarks

This check detects code where moving to `owner<T>` can be a good option for the first stage of refactoring. Like C26400, it enforces rules I.11 and R.3, but focuses on the "release" portion of the pointer lifetime. It warns on any call to operator `delete` if its target isn't an `owner<T>` or an implicitly assumed owner. For more information about `auto` declarations, see [C26400](#). This check includes expressions that refer to global variables, formal parameters, and so on.

Warnings C26400 and C26401 always occur with [C26409](#), but they're more appropriate for scenarios where immediate migration to smart pointers isn't feasible. In such cases, the `owner<T>` concept can be adopted first, and C26409 may be temporarily suppressed.

Code analysis name: `DONT_DELETE_NON_OWNER`

See also

[C++ Core Guidelines I.11 ↗](#)

Examples

C++

```
struct myStruct {};  
  
myStruct* createMyStruct();  
void function()  
{  
    myStruct* pMyStruct = createMyStruct();  
    // ...  
    delete pMyStruct; // C26401. Do not delete a raw pointer that is not an  
    owner<T>  
}
```

See that C26401 is removed if ownership of the pointer is indicated by `gsl::owner`.

C++

```
#include <gsl/pointers>
struct myStruct {};

gsl::owner<myStruct*> createMyStruct();
void function()
{
    gsl::owner<myStruct*> pMyStruct = createMyStruct();
    // ...
    delete pMyStruct; // no warning.
}
```

There's a C++ idiom that triggers this warning: `delete this`. The warning is intentional, because the C++ Core Guidelines discourage this pattern. You can suppress the warning by using the `gsl::suppress` attribute, as shown in this example:

C++

```
class MyReferenceCountingObject final
{
public:
    void AddRef();
    void Release() noexcept
    {
        ref_count_--;
        if (ref_count_ == 0)
        {
            [[gsl::suppress(i.11)]]
            delete this;
        }
    }
private:
    unsigned int ref_count_{1};
};
```

Warning C26402

Article • 10/07/2022

Return a scoped object instead of a heap-allocated if it has a move constructor
(r.3).

Remarks

To avoid confusion about whether a pointer owns an object, a function that returns a movable object should allocate it on the stack. It should then return the object by value instead of returning a heap-allocated object. If pointer semantics are required, return a smart pointer instead of a raw pointer. For more information, see [C++ Core Guidelines R.3](#): *Warn if a function returns an object that was allocated within the function but has a move constructor. Suggest considering returning it by value instead.*

Example

This example shows a `bad_example` function that raises warning C26409. It also shows how function `good_example` doesn't cause this issue.

C++

```
// C26402.cpp

struct S
{
    S() = default;
    S(S&& s) = default;
};

S* bad_example()
{
    S* s = new S(); // C26409, avoid explicitly calling new.
    // ...
    return s; // C26402
}

// Prefer returning objects with move constructors by value instead of
// unnecessarily heap-allocating the object.
S good_example() noexcept
{
    S s;
    // ...
    return s;
}
```


Warning C26403

Article • 10/07/2022

Reset or explicitly delete an `owner<T>` pointer '*variable*' (r.3)

Owner pointers are like unique pointers: they own a resource exclusively, and manage release of the resource, or its transfers to other owners. This check validates that a local owner pointer properly maintains its resource through all execution paths in a function. If the resource wasn't transferred to another owner, or wasn't explicitly released, the checker warns, and points to the declaration of the pointer variable.

For more information, see the [C++ Core Guidelines](#).

Remarks

- Currently this check doesn't give the exact path that fails to release the resource. This behavior may be improved in future releases. It may be difficult to find exact location for a fix. The better approach is to try to replace plain pointers in complex functions with unique pointers to avoid any risks.
- The check may discard an over-complicated function in order to not block code analysis. Generally, the complexity of functions should be maintained under some reasonable threshold. We may consider adding a local complexity check to the C++ Core Guidelines module if there's clear demand for it. This limitation is applicable to other rules that are sensitive to data flow.
- The warning may fire on clearly false positive cases where memory is deleted only after the null check of a pointer. These false positives are the result of a current limitation of the tool's API, but it may be improved in future.

Code analysis name: `RESET_OR_DELETE_OWNER`

Example

Missing cleanup during error handling:

C++

```
gsl::owner<int*> sequence = GetRandomSequence(); // C26403  
try
```

```
{  
    StartSimulation(sequence);  
}  
catch (const std::exception& e)  
{  
    if (KnownException(e))  
        return; // Skipping the path which deletes the owner.  
  
    ReportException(e);  
}  
  
delete [] sequence;
```

Warning C26404

Article • 10/07/2022

| Do not delete an `owner<T>` which may be in invalid state (r.3)

Remarks

Once an owner pointer releases or transfers its resource, it gets into an "invalid" state. Deleting such a pointer may lead to immediate memory corruption due to double delete, or to an access violation when the deleted resource is accessed from another owner pointer.

Code analysis name: `DONT_DELETE_INVALID`

Example 1

Deleting an owner after transferring its value:

```
C++

gsl::owner<State*> validState = nullptr;
gsl::owner<State*> state = ReadState();
validState = state;
if (!IsValid(state))
    delete state; // C26404
```

Example 2

Deleting an uninitialized owner:

```
C++

gsl::owner<Message*> message;
if (popLast)
    message = ReleaseMessage();
delete message; // C26404
```

Warning C26405

Article • 10/07/2022

Do not assign to an `owner<T>` which may be in valid state (r.3)

Remarks

If an owner pointer already points to a valid memory buffer, it must not be assigned to another value without releasing its current resource first. Such assignment may lead to a resource leak even if the resource address is copied into some raw pointer (because raw pointers shouldn't release resources). For more information, see the [C++ Core Guidelines](#).

Code analysis name: `DONT_ASSIGN_TO_VALID`

Example 1

Overwriting an owner in a loop:

C++

```
gsl::owner<Shape*> shape = nullptr;
while (shape = NextShape()) // C26405
    Process(shape) ? delete shape : 0;
```

Warning C26406

Article • 10/07/2022

Do not assign a raw pointer to an `owner<T>` (r.3)

This warning enforces R.3 from the C++ Core Guidelines. For more information, see [C++ Core Guidelines R.3 ↗](#).

Remarks

Owners are initialized from allocations or from other owners. This warning occurs when you assign a value from a raw pointer to an owner pointer. Raw pointers don't guarantee ownership transfer; the original owner may still hold the resource and attempt to release it. It's okay to assign a value from an owner to a raw pointer. Raw pointers are valid clients to access resources, but not to manage them.

Code analysis name: `DONT_ASSIGN_RAW_TO_OWNER`

Example

Using address of object:

This sample attempts to assign ownership of the address of `defaultSocket` to owner pointer `socket`:

C++

```
gsl::owner<Socket*> socket = defaultSocket ? &defaultSocket : new Socket();
// C26406
```

Warning C26407

Article • 10/07/2022

Prefer scoped objects, don't heap-allocate unnecessarily (r.5)

To avoid unnecessary use of pointers, we try to detect common patterns of local allocations. For example, we detect when the result of a call to operator `new` is stored in a local variable and later explicitly deleted. This check supports the [C++ Core Guidelines rule R.5](#): *Prefer scoped objects, don't heap-allocate unnecessarily*. To fix the issue, use an RAII type instead of a raw pointer, and allow it to deal with resources. Obviously, it isn't necessary to create a wrapper type to allocate a single object. Instead, a local variable of the object's type would work better.

Remarks

- To reduce the number of warnings, code analysis only detects this pattern for owner pointers. So, it's necessary to mark owners properly first. We can easily extend this analysis to cover raw pointers if we receive feedback on the Visual Studio C++ [Developer Community](#) from customers in support of such scenarios.
- The *scoped object* term may be a bit misleading. In general, we suggest you use either a local variable whose lifetime is automatically managed, or a smart object that efficiently manages dynamic resources. Smart objects can do heap allocations, but it's not explicit in the code.
- If the warning fires on array allocation, which is often needed for dynamic buffers, you can fix it by using standard containers, or `std::unique_pointer<T[]>`.
- The pattern is detected only for local variables. We don't warn in cases where an allocation is assigned to, say, a global variable and then deleted in the same function.

Code analysis name: `DONT_HEAP_ALLOCATE_UNNECESSARILY`

Example 1: Unnecessary object allocation on heap

C++

```
auto tracer = new Tracer();
ScanObjects(tracer);
delete tracer; // C26407
```

Example 2: Unnecessary object allocation on heap (fixed with local object)

C++

```
Tracer tracer; // OK
ScanObjects(&tracer);
```

Example 3: Unnecessary buffer allocation on heap

C++

```
auto value = new char[maxValueSize];
if (ReadSetting(name, value, maxValueSize))
    CheckValue(value);
delete[] value; // C26407
```

Example 4: Unnecessary buffer allocation on the heap (fixed with container)

C++

```
auto value = std::vector<char>(maxValueSize); // OK
if (ReadSetting(name, value.data(), maxValueSize))
    CheckValue(value.data());
```

Warning C26408

Article • 10/07/2022

Avoid `malloc()` and `free()`, prefer the `nothrow` version of `new` with `delete` (r.10)

This warning flags places where `malloc` or `free` is invoked explicitly in accordance to R.10: Avoid `malloc` and `free`. One potential fix for such warnings would be to use `std::make_unique` to avoid explicit creation and destruction of objects. If such a fix isn't acceptable, operator `new` and `delete` should be preferred. In some cases, if exceptions aren't welcome, `malloc` and `free` can be replaced with the `nothrow` version of operators `new` and `delete`.

Remarks

- To detect `malloc()`, we check if a call invokes a global function named `malloc` or `std::malloc`. The function must return a pointer to `void` and accept one parameter of unsigned integral type.
- To detect `free()`, we check global functions named `free` or `std::free` that return no result and accept one parameter, which is a pointer to `void`.

Code analysis name: `NO_MALLOC_FREE`

See also

[C++ Core Guidelines R.10 ↗](#)

Example

C++

```
#include <new>

struct myStruct {};

void function_malloc_free() {
    myStruct* ms = static_cast<myStruct*>(malloc(sizeof(myStruct))); // C26408
    free(ms); // C26408
}
```

```
void function_nothrow_new_delete() {
    myStruct* ms = new(std::nothrow) myStruct;
    operator delete (ms, std::nothrow);
}
```

Warning C26409

Article • 10/07/2022

Avoid calling `new` and `delete` explicitly, use `std::make_unique<T>` instead (r.11).

Even if code is clean of calls to `malloc` and `free`, we still suggest that you consider better options than explicit use of operators `new` and `delete`.

C++ Core Guidelines:

R.11: Avoid calling new and delete explicitly

The ultimate fix is to use smart pointers and appropriate factory functions, such as `std::make_unique`.

Remarks

- The checker warns on calls to any kind of operator `new` or `delete`: scalar, vector, overloaded versions (global and class-specific), and placement versions. The placement `new` case may require some clarifications in the Core Guidelines for suggested fixes, and may be omitted in the future.

Code analysis name: NO_NEW_DELETE

Examples

This example shows C26409 is raised for explicit `new` and `delete`. Consider using smart pointer factory functions such as `std::make_unique` instead.

```
C++  
  
void f(int i)  
{  
    int* arr = new int[i]{}; // C26409, warning is issued for all new calls  
    delete[] arr;           // C26409, warning is issued for all delete  
    calls  
  
    auto unique = std::make_unique<int[]>(i); // prefer using smart pointers  
    over new and delete  
}
```

There's a C++ idiom that triggers this warning: `delete this`. The warning is intentional, because the C++ Core Guidelines discourage this pattern. You can suppress the warning by using the `gsl::suppress` attribute, as shown in this example:

C++

```
class MyReferenceCountingObject final
{
public:
    void AddRef();
    void Release() noexcept
    {
        ref_count_--;
        if (ref_count_ == 0)
        {
            [[gsl::suppress(i.11)]]
            delete this;
        }
    }
private:
    unsigned int ref_count_{1};
};
```

Warning C26410

Article • 10/07/2022

The parameter '*parameter*' is a reference to const unique pointer, use `const T*` or `const T&` instead (r.32)

Generally, references to const unique pointer are meaningless. They can safely be replaced by a raw reference or a pointer. This warning enforces [C++ Core Guidelines rule R.32](#).

Remarks

- Unique pointer checks have rather broad criteria to identify smart pointers. The [C++ Core Guidelines rule R.31](#): *If you have non-std smart pointers, follow the basic pattern from std describes the unique pointer and shared pointer concepts.* The heuristic is simple, but may lead to surprises: a smart pointer type is any type that defines either `operator->` or `operator*`. A copy-able type (shared pointer) must have either a public copy constructor or an overloaded assignment operator that deals with a non-RValue reference parameter.
- Template code may produce noisy warnings. Keep in mind that templates can be instantiated with various type parameters with different levels of indirection, including references. Some warnings may not be obvious and fixes may require some rework of templates (for example, explicit removal of reference indirection). If template code is intentionally generic, the warning can be suppressed.

Code analysis name: `NO_REF_TO_CONST_UNIQUE_PTR`

Example

Unnecessary reference:

C++

```
std::vector<std::unique_ptr<Tree>> roots = GetRoots();
std::for_each(
    roots.begin(),
    roots.end(),
    [](&const auto &root) { Rebalance(root.get()); }); // C26410
```

Warning C26411

Article • 10/07/2022

The parameter '*parameter*' is a reference to unique pointer and it is never reassigned or reset, use `T*` or `T&` instead (r.33)

When you pass a unique pointer to a function by reference, it implies that its resource may be released or transferred inside the function. If the function uses its parameter only to access the resource, it's safe to pass a raw pointer or a reference. For more information, see [C++ Core Guidelines rule R.33 ↗](#): *Take a unique_ptr<widget>& parameter to express that a function reseats the widget.*

Remarks

- The limitations from the warning [C26410](#) are also applicable here.
- The heuristic to detect `release` or `reset` access to the unique pointer is naive. We only detect calls to assignment operators and to functions named `reset` (case-insensitive). Obviously, this detection doesn't cover all possible cases of smart pointer modifications. (For example, it doesn't detect `std::swap`, or any special non-`const` function in a custom smart pointer). We expect this warning may produce many false positives on custom types, and in some scenarios dealing with standard unique pointers. We expect to improve the heuristic as we implement more checks focused on smart pointers.
- The fact that smart pointers are often templates brings an interesting limitation. The compiler isn't required to process template code in templates if it's not used. In code that makes limited use of smart pointer interfaces, the checker may produce unexpected results. The checker can't properly identify semantics of the template type, because some functions may never get used. For the standard `std::unique_ptr`, this limitation is mitigated by recognizing the type's name. This analysis may be extended in the future to cover more well-known smart pointers.
- Lambda expressions that do implicit capture-by-reference may lead to surprising warnings about references to unique pointers. Currently, all captured reference parameters in lambdas are reported, regardless of whether they're reset or not. A future release may extend the heuristic to correlate lambda fields and lambda parameters.

Code analysis name: `NO_REF_TO_UNIQUE_PTR`

Example: Unnecessary reference

C++

```
void TraceValid(std::unique_ptr<Slot> &slot)    // C26411
{
    if (!IsDamaged(slot.get()))
        std::cout << *slot.get();
}

void ReleaseValid(std::unique_ptr<Slot> &slot) // OK
{
    if (!IsDamaged(slot.get()))
        slot.reset(nullptr);
}
```

Warning C26414

Article • 10/07/2022

"Move, copy, reassign or reset a local smart pointer."

C++ Core Guidelines:

[R.5: Prefer scoped objects, don't heap-allocate unnecessarily](#) ↗

Smart pointers are convenient for dynamic resource management, but they're not always necessary. For example, it may be easier and more efficient to manage a local dynamic buffer by using a standard container. You may not need dynamic allocation at all for single objects, for example, if they never outlive their creator function. They can be replaced with local variables. Smart pointers become handy when a scenario requires a change of ownership. For example, when you reassign a dynamic resource multiple times, or in multiple paths. They're also useful for resources obtained from external code. And, when smart pointers are used to extend the lifetime of a resource.

Remarks

This check recognizes both the standard `std::unique_ptr` and `std::shared_ptr` templates, and user-defined types that are likely intended to be smart pointers. Such types are expected to define the following operations:

- overloaded dereference or member access operators that are public and not marked as deleted;
- a public destructor that isn't deleted or defaulted. That includes destructors explicitly defined as empty.

The type `Microsoft::WRL::ComPtr` behaves as a shared pointer, but it's often used in specific scenarios that are affected by the COM lifetime management. To avoid excessive noise this type is filtered out.

This check looks for explicit local allocations assigned to smart pointers, to identify if scoped variables could work as an alternative. Both direct calls to operator `new`, and special functions like `std::make_unique` and `std::make_shared`, are interpreted as direct allocations.

Code analysis name: `RESET_LOCAL_SMART_PTR`

Example

Dynamic buffer:

C++

```
void unpack_and_send(const frame &f)
{
    auto buffer = std::make_unique<char[]>(f.size()); // C26414
    f.unpack(buffer.get());
    // ...
}
```

Dynamic buffer replaced by container:

C++

```
void unpack_and_send(const frame &f)
{
    auto buffer = std::vector<char>(f.size());
    f.unpack(buffer.data());
    // ...
}
```

Warning C26415

Article • 10/07/2022

Smart pointer parameter is used only to access contained pointer. Use T* or T& instead.

C++ Core Guidelines: [R.30 ↗](#): Take smart pointers as parameters only to explicitly express lifetime semantics

Using a smart pointer type to pass data to a function indicates that the target function needs to manage the lifetime of the contained object. However, say the function only uses the smart pointer to access the contained object and never actually calls any code that may lead to its deallocation (that is, never affects its lifetime). Then there's usually no need to complicate the interface with smart pointers. A plain pointer or reference to the contained object is preferred.

Remarks

This check covers most scenarios that also cause C26410, C26415, C26417, and C26418. It's better to clean up SMART_PTR_NOT_NEEDED first and then switch to edge cases for shared or unique pointers. For more focused cleanup, this warning can be disabled.

In addition to the standard std::unique_pointer and std::shared_pointer templates, this check recognizes user-defined types that are likely intended to be smart pointers. Such types are expected to define the following operations:

- Overloaded dereference or member access operators that are public and not marked as deleted.
- Public destructor that's not deleted or defaulted, including destructors that are explicitly defined empty.

Interpretation of the operations that can affect the lifetime of contained objects is broad and includes:

- Any function that accepts a pointer or reference parameter to a non-constant smart pointer
- Copy or move constructors or assignment operators
- Non-constant functions

Examples

Cumbersome lifetime management.

C++

```
bool set_initial_message(
    const std::unique_ptr<message> &m) // C26415, also C26410
NO_REF_TO_CONST_UNIQUE_PTR
{
    if (!m || initial_message_)
        return false;

    initial_message_.reset(m.get());
    return true;
}

void pass_message(const message_info &info)
{
    auto m = std::make_unique<message>(info);
    const auto release = set_initial_message(m);
    // ...
    if (release)
        m.release();
}
```

Cumbersome lifetime management - reworked.

C++

```
void set_initial_message(std::shared_ptr<message> m) noexcept
{
    if (m && !initial_message_)
        initial_message_ = std::move(m);
}

void pass_message(const message_info &info)
{
    auto m = std::make_shared<message>(info);
    set_initial_message(m);
    // ...
}
```

Warning C26416

Article • 10/07/2022

Shared pointer parameter is passed by rvalue reference. Pass by value instead.

C++ Core Guidelines: [R.34 ↗](#): Take a `shared_ptr<widget>` parameter to express that a function is part owner

Passing a shared pointer by rvalue reference is rarely necessary. Unless it's an implementation of move semantics for a shared pointer type itself, shared pointer objects can be safely passed by value. Using rvalue reference may be also an indication that unique pointer is more appropriate since it clearly transfers unique ownership from caller to callee.

Remarks

- This check recognizes `std::shared_pointer` and user-defined types that are likely to behave like shared pointers. The following traits are expected for user-defined shared pointers:
 - overloaded dereference or member access operators (public and non-deleted);
 - a copy constructor or copy assignment operator (public and non-deleted);
 - a public destructor that isn't deleted or defaulted. Empty destructors are still counted as user-defined.

Examples

Questionable constructor optimization:

C++

```
action::action(std::shared_ptr<transaction> &&t) noexcept // C26416
    : transaction_(std::move(t))
{}

action::action(std::shared_ptr<transaction> &t) noexcept // also C26417
LVALUE_REF_SHARED_PTR
    : transaction_(t)
{}
```

Questionable constructor optimization - simplified:

C++

```
action::action(std::shared_ptr<transaction> t) noexcept
    : transaction_(std::move(t))
{}
```

Warning C26417

Article • 10/07/2022

Shared pointer parameter is passed by reference and not reset or reassigned. Use T* or T& instead.

C++ Core Guidelines: [R.35 ↗](#): Take a `shared_ptr<widget>&` parameter to express that a function might reseat the shared pointer

Passing shared pointers by reference may be useful in scenarios where called code updates the target of the smart pointer object, and its caller expects to see such updates. Using a reference solely to reduce costs of passing a shared pointer is questionable. If called code only accesses the target object and never manages its lifetime, it's safer to pass a raw pointer or reference, rather than to expose resource management details.

Remarks

- This check recognizes `std::shared_pointer` and user-defined types that are likely to behave like shared pointers. The following traits are expected for user-defined shared pointers:
 - overloaded dereference or member access operators (public and non-deleted);
 - a copy constructor or copy assignment operator (public and non-deleted);
 - a public destructor that isn't deleted or defaulted. Empty destructors are still counted as user-defined.
- The action of resetting or reassigning is interpreted in a more generic way:
 - any call to a non-constant function on a shared pointer can potentially reset the pointer;
 - any call to a function that accepts a reference to a non-constant shared pointer can potentially reset or reassign that pointer.

Examples

unnecessary interface complication

C++

```
bool unregister(std::shared_ptr<event> &e) // C26417, also C26415
SMART_PTR_NOT_NEEDED
{
    return e && events_.erase(e->id());
}

void renew(std::shared_ptr<event> &e)
{
    if (unregister(e))
        e = std::make_shared<event>(e->id());
    // ...
}
```

unnecessary interface complication - simplified

C++

```
bool unregister(const event *e)
{
    return e && events_.erase(e->id());
}

void renew(std::shared_ptr<event> &e)
{
    if (unregister(e.get()))
        e = std::make_shared<event>(e->id());
    // ...
}
```

Warning C26418

Article • 10/07/2022

Shared pointer parameter is not copied or moved. Use T* or T& instead.

C++ Core Guidelines: [R.36 ↗](#): Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object

If a shared pointer parameter is passed by value or by reference to a constant object, the function is expected to take control of the target object's lifetime without affecting the caller. The code should either copy or move the shared pointer parameter to another shared pointer object, or pass it along to other code by invoking functions that accept shared pointers. Otherwise, a plain pointer or reference may be feasible.

Remarks

- This check recognizes `std::shared_pointer` and user-defined types that are likely to behave like shared pointers. The following traits are expected for user-defined shared pointers:
 - overloaded dereference or member access operators (public and non-deleted);
 - a copy constructor or copy assignment operator (public and non-deleted);
 - a public destructor that isn't deleted or defaulted. Empty destructors are still counted as user-defined.

Examples

unnecessary interface complication

C++

```
template<class T>
std::string to_string(const std::shared_ptr<T> &e) // C26418, also C26415
SMART_PTR_NOT_NEEDED
{
    return !e ? null_string : e->to_string();
}
```

unnecessary interface complication - simplified

C++

```
template<class T>
std::string to_string(const T *e)
{
    return !e ? null_string : e->to_string();
}
```

Warning C26426

Article • 10/07/2022

Global initializer calls a non-constexpr function 'symbol' (i.22)

C++ Core Guidelines

[I.22 ↗](#): Avoid complex initialization of global objects

The order of execution of initializers for global objects may be inconsistent or undefined, which can lead to issues that are hard to reproduce and investigate. To avoid such problems, global initializers shouldn't depend on external code that's executed at run time, and that may depend on data that's not yet initialized. This rule flags cases where global objects call functions to obtain their initial values.

Remarks

- The rule ignores calls to `constexpr` functions or intrinsic functions on the assumption that these calls either will be calculated at compile time or guarantee predictable execution at run time.
- Calls to inline functions are still flagged, since the checker doesn't attempt to analyze their implementation.
- This rule can be noisy in many common scenarios where a variable of a user-defined type (or a standard container) is initialized globally. It's often due to calls to constructors and destructors. It's still a valid warning, since it points to places where unpredictable behavior may exist or where future changes in external code may introduce instability.
- Static class members are considered global, so their initializers are also checked.

Code analysis name: `NO_GLOBAL_INIT_CALLS`

Examples

External version check:

C++

```
// api.cpp
int api_version = API_DEFAULT_VERSION; // Assume it can change at run time,
hence non-const.
int get_api_version() noexcept {
    return api_version;
}

// client.cpp
int get_api_version() noexcept;
bool is_legacy_mode = get_api_version() <= API_LEGACY_VERSION; // C26426,
also stale value
```

External version check made more reliable:

C++

```
// api.cpp
int& api_version() noexcept {
    static auto value = API_DEFAULT_VERSION;
    return value;
}
int get_api_version() noexcept {
    return api_version();
}

// client.cpp
int get_api_version() noexcept;
bool is_legacy_mode() noexcept {
    return get_api_version() <= API_LEGACY_VERSION;
}
```

Warning C26427

Article • 10/07/2022

Global initializer accesses extern object 'symbol' (i.22)

C++ Core Guidelines: [I.22 ↗](#): Avoid complex initialization of global objects

Global objects may be initialized in an inconsistent or undefined order, which means that interdependency between them is risky and should be avoided. This guideline is applicable when initializers refer to another object that's considered to be `extern`.

Remarks

An object is deemed `extern` if it conforms to the following rules:

- it's a global variable marked with `extern` specifier or it's a static member of a class;
- it's not in an anonymous namespace;
- it's not marked as `const`;
- Static class members are considered global, so their initializers are also checked.

Code analysis name: `NO_GLOBAL_INIT_EXTERNS`

Example

External version check:

```
C++  
  
// api.cpp  
int api_version = API_DEFAULT_VERSION; // Assume it can change at run time,  
hence non-const.  
  
// client.cpp  
extern int api_version;  
bool is_legacy_mode = api_version <= API_LEGACY_VERSION; // C26427, also  
stale value
```

External version check made more reliable:

```
C++
```

```
// api.cpp
int api_version = API_DEFAULT_VERSION; // Assume it can change at run time,
hence non-const.

// client.cpp
extern int api_version;
bool is_legacy_mode() noexcept
{
    return api_version <= API_LEGACY_VERSION;
}
```

Warning C26429

Article • 10/07/2022

Symbol is never tested for nullness, it can be marked as `gsl::not_null`.

C++ Core Guidelines: [F.23](#): Use a `not_null<T>` to indicate that "null" isn't a valid value

It's a common practice to use asserts to enforce assumptions about the validity of pointer values. The problem is, asserts don't expose assumptions through the interface (such as in return types or parameters). Asserts are also harder to maintain and keep in sync with other code changes. The recommendation is to use `gsl::not_null` from the Guidelines Support Library to mark resources that should never have a null value. The rule `USE_NONNULL` helps to identify places that omit checks for null and hence can be updated to use `gsl::not_null`.

Remarks

The logic of the rule requires code to dereference a pointer variable so that a null check (or enforcement of a non-null value) would be justified. So, warnings are emitted only if pointers are dereferenced and never tested for null.

The current implementation handles only plain pointers (or their aliases) and doesn't detect smart pointers, even though `gsl::not_null` can be applied to smart pointers as well.

A variable is marked as checked for null when it's used in the following contexts:

- as a symbol expression in a branch condition, for example, `if (p) { ... }`;
- non-bitwise logical operations;
- comparison operations where one operand is a constant expression that evaluates to zero.

The rule doesn't have full dataflow tracking. It can produce incorrect results in cases where indirect checks are used (such as when an intermediate variable holds a null value and is later used in a comparison).

Code analysis name: `USE_NONNULL`

Example

Hidden expectation:

C++

```
using client_collection = gsl::span<client*>;
// ...
void keep_alive(const connection *connection)    // C26429
{
    const client_collection clients = connection->get_clients();
    for (ptrdiff_t i = 0; i < clients.size(); i++)
    {
        auto client = clients[i];                // C26429
        client->send_heartbeat();
        // ...
    }
}
```

Hidden expectation clarified by `gsl::not_null`:

C++

```
using client_collection = gsl::span<gsl::not_null<client*>>;
// ...
void keep_alive(gsl::not_null<const connection*> connection)
{
    const client_collection clients = connection->get_clients();
    for (ptrdiff_t i = 0; i < clients.size(); i++)
    {
        auto client = clients[i];
        client->send_heartbeat();
        // ...
    }
}
```

Warning C26430

Article • 06/27/2024

Symbol is not tested for nullness on all paths.

C++ Core Guidelines: [F.23 ↗](#): Use a `not_null<T>` to indicate that "null" isn't a valid value

If code ever checks pointer variables for null, it should do so consistently and validate pointers on all paths. Sometimes overaggressive checking for null is still better than the possibility of a hard crash in one of the complicated branches. Ideally, such code should be refactored to be less complex (by splitting it into multiple functions), and to rely on markers like `gsl::not_null`. These markers allow the code to isolate parts of the algorithm that can make safe assumptions about valid pointer values. The rule `TEST_ON_ALL_PATHS` helps to find places where null checks are inconsistent (meaning assumptions may require review). Or, it finds actual bugs where a potential null value can bypass null checks in some of the code paths.

Remarks

This rule expects that code dereferences a pointer variable so that a null check (or enforcement of a non-null value) would be justified. If there's no dereference, the rule is suspended.

The current implementation handles only plain pointers (or their aliases) and doesn't detect smart pointers, even though null checks are applicable to smart pointers as well.

A variable is marked as checked for null when it's used in the following contexts:

- as a symbol expression in a branch condition, for example, in `if (p) { ... }`;
- in non-bitwise logical operations;
- in comparison operations where one operand is a constant expression that evaluates to zero.

Implicit null checks are assumed when a pointer value is assigned from:

- an allocation performed with throwing `operator new`;
- a pointer obtained from a type marked with `gsl::not_null`.

Example

inconsistent testing reveals logic error

C++

```
void merge_states(const state *left, const state *right) // C26430
{
    if (*left && *right)
        converge(left, right);
    else
    {
        // ...
        if (!left && !right) // Logic error!
            discard(left, right);
    }
}
```

inconsistent testing reveals logic error - corrected

C++

```
void merge_states(gsl::not_null<const state *> left, gsl::not_null<const state *> right)
{
    if (*left && *right)
        converge(left, right);
    else
    {
        // ...
        if (*left && *right)
            discard(left, right);
    }
}
```

Heuristics

When ensuring that a dereference of a pointer isn't null, this rule doesn't require *every* dereference to have a prior null check. Instead, it requires a null check before *first* dereference of the pointer. The following function doesn't trigger C26430:

C++

```
void f(int* p)
{
    if (p)
        *p = 1;
    *p = 2;
}
```

The following function generates C26430 because there's a path to assign `*p` without a null check:

C++

```
void f(bool b, int* p)
{
    if (b && p)
        *p = 1;
    *p = 2;
}
```

Rules [C26822](#) and [C26823](#) apply to dereferencing a (possibly) null pointer.

This rule doesn't do full data flow tracking. It can produce incorrect results in cases where indirect checks are used, such as when an intermediate variable holds a null value and is later used in a comparison.

See also

[C26822](#)

[C26823](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Warning C26431

Article • 10/07/2022

The type of expression 'expr' is already `gsl::not_null`. Do not test it for nullness
(f.23)

C++ Core Guidelines: [F.23 ↗](#): Use a `not_null<T>` to indicate that "null" isn't a valid value

The marker type `gsl::not_null` from the Guidelines Support Library is used to clearly indicate values that are never null pointers. It causes a hard failure if the assumption doesn't hold at run time. So, obviously, there's no need to check for null if an expression evaluates to a result of type `gsl::not_null`.

Remarks

Since `gsl::not_null` itself is a thin pointer wrapper class, the rule actually tracks temporary variables that hold results from calls to the overloaded conversion operator (which returns contained pointer objects). Such logic makes this rule applicable to expressions that involve variables and eventually have a result of the `gsl::not_null` type. However, it currently skips expressions that contain function calls returning `gsl::not_null`.

The current heuristic for null checks detects the following contexts:

- a symbol expression in a branch condition, for example `if (p) { ... }`;
- non-bitwise logical operations;
- comparison operations where one operand is a constant expression that evaluates to zero.

Code analysis name: `DONT_TEST_NOTNULL`

Example

Unnecessary null checks reveal questionable logic:

C++

```
class type {
public:
    template<class T> bool is() const;
    template<class T> gsl::not_null<const T*> as() const;
```

```

//...
};

class alias_type : public type {
public:
    gsl::not_null<const type*> get_underlying_type() const;
    gsl::not_null<const type*> get_root_type() const
    {
        const auto ut = get_underlying_type();
        if (ut) // C26431
        {
            const auto uat = ut->as<alias_type>();
            if (uat) // C26431, also
incorrect use of API!
                return uat->get_root_type();

            return ut;
        }

        return this; // Alias to nothing?
    Actually, dead code!
    }
    //...
};

```

Unnecessary null checks reveal questionable logic, reworked:

C++

```

//...
gsl::not_null<const type*> get_root_type() const
{
    const auto ut = get_underlying_type();
    if (ut->is<alias_type>())
        return ut->as<alias_type>()->get_root_type();

    return ut;
}
//...

```

Warning C26432

Article • 10/07/2022

If you define or delete any default operation in the type 'type-name', define or delete them all (c.21).

C++ Core Guidelines:

[C.21: If you define or =delete any default operation, define or =delete them all ↗](#)

Special operations such as constructors are assumed to alter the behavior of types so they rely more on language mechanisms to automatically enforce specific scenarios. The canonical example is resource management. If you explicitly define, default, or delete any of these special operations, it signals you want to avoid any special handling of a type. It's inconsistent to leave the other operations unspecified, that is, implicitly defined as deleted by the compiler.

Remarks

This check implements the *rule of five*, which treats the following operations as special:

- copy constructors,
- move constructors,
- copy assignment operators,
- move assignment operators, and
- destructors.

The rule doesn't check if operations are defined in the same way. It's okay to mix deleted and defaulted operations with explicitly defined ones. However, you must specify all of them if you specify any of them.

Access levels aren't important and can also be mixed.

The warning flags the first non-static function definition of a type, once per type.

Example

In this example, `warning::S` defines only a default constructor and a destructor. The `no_warning::S` declaration defines or deletes all five special member functions.

```
// C26432.cpp
namespace warning
{
    struct S
    {
        S() noexcept { ++_count; }
        ~S() { --_count; } // C26432 because only the constructor and
                           destructor are explicitly defined.
        static unsigned _count;
    };
    unsigned S::_count = 0;
}

namespace no_warning
{
    struct S
    {
        S() noexcept { _count++; }
        S(const S&) = delete;
        S(S&&) = delete;
        S& operator=(const S&) = delete;
        S& operator=(S&&) = delete;
        ~S() { --_count; }
        static unsigned _count;
    };
    unsigned S::_count = 0;
}
```

Warning C26433

Article • 10/07/2022

Function should be marked with `override`

C++ Core Guidelines

C.128: Virtual functions should specify exactly one of virtual, override, or final ↗

It's not required by the compiler to clearly state that a virtual function overrides its base. Not specifying `override` can cause subtle issues during maintenance if the virtual specification ever changes in the class hierarchy. It also lowers readability and makes an interface's polymorphic behavior less obvious. If a function is clearly marked as `override`, the compiler can check the consistency of the interface, and help to spot issues before they manifest themselves at run time.

Notes

This rule isn't applicable to destructors. Destructors have their own virtuality specifics.

The rule doesn't flag functions explicitly marked as `final`, which is itself a special variety of virtual specifier.

Warnings show up on function definitions, not declarations. It may be confusing, since definitions don't have virtual specifiers, but the warning is still correct.

Code analysis name: `OVERRIDE_EXPLICITLY`

Example: Implicit overriding

C++

```
class Shape {
public:
    virtual void Draw() = 0;
    // ...
};

class Ellipse : public Shape {
public:
    void Draw() { // C26433
        //...
    }
}
```

```
    }  
};
```

See also

C.128: Virtual functions should specify exactly one of virtual, override, or final ↗

Warning C26434

Article • 10/07/2022

Function 'derived::function' hides a non-virtual function 'base::function' (c.128).

C++ Core Guidelines

[C.128: Virtual functions should specify exactly one of virtual, override, or final ↗](#)

Remarks

When you introduce a function that has the same name as a non-virtual function in a base class, you may get unexpected behavior. It's like introducing a variable name that conflicts with a name from an outer scope. For example, you may have intended to override a base class function. If the signatures of the functions don't match, the override you intended may turn into an overload instead. In general, name hiding is dangerous and error-prone.

In the Core Guidelines checks:

- Only non-overriding functions in the current class are checked.
- Only non-virtual functions of base classes are considered.
- No signature matching is performed. Warnings are emitted if unqualified names match.

Example

This example demonstrates how a derived class can hide non-virtual functions, and how virtual functions allow both overloads and overrides:

C++

```
// C26434.cpp
struct Base
{
    virtual ~Base() = default;
    virtual void is_virtual() noexcept {}
    void not_virtual() noexcept {}
};

struct Derived : Base
{
```

```
    void is_virtual() noexcept override {}           // Okay, override existing
function
    virtual void is_virtual(int i) noexcept {} // Add a virtual overload
for function
    void not_virtual() noexcept {}           // C26434, hides a non-
virtual function
    virtual void not_virtual(int i) noexcept {} // C26434, and parameters
ignored
};
```

Warning C26435

Article • 11/06/2024

The virtual function 'symbol' should specify exactly one of 'virtual', 'override', or 'final' (c.128)

C++ Core Guidelines

[C.128: Virtual functions should specify exactly one of virtual, override, or final ↗](#)

To improve readability, the kind of virtual behavior should be stated clearly and without unnecessary redundancy. Even though multiple virtual specifiers can be used simultaneously, it's better to specify one at a time to emphasize the most important aspect of virtual behavior. The following order of importance is apparent:

- plain virtual function;
- virtual function that explicitly overrides its base;
- virtual function that overrides its base and provides the final implementation in the current inheritance chain.

Notes

- This rule skips destructors since they have special rules regarding virtuality.
- Warnings show up on function definitions, not declarations. It may be confusing, since definitions don't have virtual specifiers, but the warning is still appropriate.

Code analysis name: `SINGLE_VIRTUAL_SPECIFICATION`

Example: Redundant specifier

C++

```
class Ellipse : public Shape {
public:
    void Draw() override {
        //...
    }
};

class Circle : public Ellipse {
public:
    void Draw() override final { // C26435, only 'final' is necessary.
```

```
//...
}
    virtual void DrawCircumference() final { // C26435, should be neither
'virtual' nor 'final'.
    //...
}
};
```

See also

C.128: Virtual functions should specify exactly one of virtual, override, or final ↗

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback ↗ | Get help at Microsoft Q&A

Warning C26436

Article • 10/07/2022

The type 'symbol' with a virtual function needs either public virtual or protected non-virtual destructor (c.35)

C++ Core Guidelines: C.35 : A base class destructor should be either public and virtual, or protected and nonvirtual

If a class defines a virtual function it becomes polymorphic, which implies that derived classes can change its behavior including resource management and destruction logic. Because client code may call polymorphic types via pointers to base classes, there's no way a client can explicitly choose which behavior is appropriate without downcasting. To make sure that resources are managed consistently and destruction occurs according to the actual type's rules, you should define a public virtual destructor. If the type hierarchy is designed to disallow client code to destroy objects directly, destructors should be defined as protected non-virtual.

Remarks

- The warning shows up on the first virtual function definition of a type (it can be a virtual destructor if it isn't public), once per type.
- Since a definition can be placed separately from a declaration, it may not always have any of the virtual specifiers. But the warning is still valid: it checks the actual 'virtuality' of a function.

Code analysis name: NEED_VIRTUAL_DTOR

Example

C++

```
namespace no_destructor
{
    struct base {
        virtual void foo() {} // C26436, see remarks to understand the
placement of the warning.
    };
}
```

The warning doesn't appear when the base class has either a virtual public destructor or a protected non-virtual destructor.

C++

```
namespace virtual_destructor
{
    struct base {
        virtual ~base();
        virtual void foo() {}
    };
}
namespace protected_destructor
{
    struct base {
        virtual void foo() {}
protected:
    ~base() {}
};
}
```

Warning C26437

Article • 06/26/2023

Do not slice.

C++ Core Guidelines: [ES.63: Don't slice ↗](#)

The language allows [slicing ↗](#) and can be viewed as a special case of a dangerous implicit cast. Even if it's done intentionally and doesn't lead to immediate issues, it's still highly discouraged. It makes code harder to change, by forcing extra requirements on related data types. It's especially true if types are polymorphic or involve resource management.

Remarks

This rule warns not only on explicit assignments, but also on implicit slicing. Implicit slicing happens when a result gets returned from the current function, or when data gets passed to other functions.

The rule also flags cases where an assignment doesn't involve real data slicing (for example, if types are empty or don't make any dangerous data manipulations). Such warnings should still be fixed to prevent any undesirable regressions if data types or behaviors change in the future.

Example

In the next code example, we read `id_ex`, but the caller of the function will only get a slice of the object:

```
C++  
  
struct id {  
    int value;  
};  
  
struct id_ex : id {  
    int extension;  
};  
  
bool read_id(stream &s, id &v) {  
    id_ex tmp{};  
    if (!s.read(tmp.value) || !s.read(tmp.extension))  
        return false;
```

```
v = tmp; // C26437
return true;
}
```

To fix the issue, update the function to use the correct types:

C++

```
// ...
bool read_id(stream &s, id_ex &v) {
// ...
```

Warning C26438

Article • 10/07/2022

Avoid `goto` (es.76)

C++ Core Guidelines:

[ES.76 ↗](#): Avoid goto

The use of `goto` is widely considered a dangerous and error-prone practice. It's acceptable only in generated code, such as in a parser generated from a grammar. With modern C++ features and utilities provided by the Guidelines Support Library, it should be easy to avoid `goto` altogether.

Remarks

- This rule warns on any occurrence of `goto`, even if it happens in dead code, except template code that's never used and so is ignored by the compiler.
- Warnings can multiply when a macro contains `goto`. Current reporting mechanisms point to all instances where such a macro gets expanded. It can often be fixed in one place by changing the macro, or avoiding its use in favor of more maintainable mechanisms.

Code analysis name: `NO_GOTO`

Example

'goto clean-up' in macro

```
C++  
  
#define ENSURE(E, L) if (!(E)) goto L;  
  
void poll(connection &c)  
{  
    ENSURE(c.open(), end); // C26438  
  
    while (c.wait())  
    {  
        connection::header h{};  
        connection::signature s{};  
        ENSURE(c.read_header(h), end); // C26438  
        ENSURE(c.read_signature(s), end); // C26438  
        // ...  
    }  
}
```

```
    }

end:
    c.close();
}
```

'goto clean-up' in macro, replaced with `gsl::finally`

C++

```
void poll(connection &c)
{
    auto end = gsl::finally([&c] { c.close(); });

    if (!c.open())
        return;

    while (c.wait())
    {
        connection::header h{};
        connection::signature s{};
        if(!c.read_header(h))
            return;
        if(!c.read_signature(s))
            return;
        // ...
    }
}
```

Warning C26439

Article • 06/26/2023

This kind of function may not throw. Declare it 'noexcept'.

C++ Core Guidelines F.6 [↗](#): If your function must not throw, declare it `noexcept`

Some operations should never throw exceptions. Their implementations should be reliable and should handle possible errors conditions gracefully. They shouldn't use exceptions to indicate failure. This rule flags cases where such operations aren't explicitly marked as `noexcept`, which means that they may throw exceptions and consumers can't make assumptions about its reliability.

It's important for these functions to be reliable as they're often used as building blocks to implement functions with [exception safety guarantees](#) [↗](#). A move constructor that throws will force Standard Template Library (STL) containers to fall back to copy operations, reducing runtime performance.

Code analysis name: `SPECIAL_NOEXCEPT`

Remarks

- Special kinds of operations:
 - destructors;
 - move constructors and move assignment operators;
 - standard functions with move semantics: `std::move` and `std::swap`.
- Nonstandard and outdated specifiers like `throw()` or `declspec(nothrow)` aren't equivalent to `noexcept`.
- Explicit specifiers `noexcept(false)` and `noexcept(true)` are respected appropriately.

Example

The tool warns on all functions except the destructor because they're missing `noexcept`.

C++

```
struct S
{
```

```
~S() {}

S(S&& s) /*impl*/ // C26439, This kind of function may not throw.
Declare it 'noexcept' (f.6)
S& operator=(S&& s) /*impl*/ // C26439, This kind of function may not
throw. Declare it 'noexcept' (f.6)

S(const S& s) /*impl*/ // C26440, This function can be declared
'noexcept'
S& operator=(const S& s) /*impl*/ // C26440, This function can be
declared 'noexcept'
};
```

With `noexcept` decorating the same structure, all warnings are removed.

C++

```
struct S
{
    ~S() {}

    S(S&& s) noexcept /*impl*/
    S& operator=(S&& s) noexcept /*impl*/

    S(const S& s) noexcept /*impl*/
    S& operator=(const S& s) noexcept /*impl*/
};
```

See also

[C26455](#)

[C++ Core Guidelines F.6 ↗](#)

Warning C26440

Article • 10/07/2022

Function can be declared 'noexcept'.

[C++ Core Guidelines F.6 ↴](#): If your function may not throw, declare it `noexcept`

If code isn't supposed to cause any exceptions, it should be marked by using the `noexcept` specifier. This annotation helps to simplify error handling on the client code side, and enables the compiler to do more optimizations.

Remarks

- A function is considered non-throwing if:
 - it has no explicit `throw` statements;
 - function calls in its body, if any, invoke only functions that are unlikely to throw: `constexpr` or functions marked with any exception specification that entails non-throwing behavior (including some non-standard specifications).
- Non-standard and outdated specifiers like `throw()` or `_declspec(nothrow)` aren't equivalent to `noexcept`.
- Explicit specifiers `noexcept(false)` and `noexcept(true)` are respected appropriately.
- Functions marked as `constexpr` aren't supposed to cause exceptions and aren't analyzed.
- The rule also applies to lambda expressions.
- The logic doesn't consider recursive calls as potentially non-throwing. This logic may change in the future.

Example

All functions except the destructor will warn because they're missing noexcept.

C++

```
struct S
{
    S() {} // C26455, Default constructor may not throw. Declare it
    'noexcept'
    ~S() {}
```

```
S(S&& s) /*impl*/ // C26439, This kind of function may not throw.  
Declare it 'noexcept' (f.6)  
S& operator=(S&& s) /*impl*/ // C26439, This kind of function may not  
throw. Declare it 'noexcept' (f.6)  
  
S(const S& s) /*impl*/ // C26440, This function can be declared  
'noexcept'  
S& operator=(const S& s) /*impl*/ // C26440, This function can be  
declared 'noexcept'  
};
```

With noexcept decorating the same structure, all warnings are removed.

C++

```
struct S  
{  
    S() noexcept {}  
    ~S() {}  
  
    S(S&& s) noexcept /*impl*/  
    S& operator=(S&& s) noexcept /*impl*/  
  
    S(const S& s) noexcept /*impl*/  
    S& operator=(const S& s) noexcept /*impl*/  
};
```

Warning C26441

Article • 06/05/2023

Guard objects must be named (cp.44)

C++ Core Guidelines

[CP.44 ↗](#): Remember to name your `lock_guard`s and `unique_lock`s

Remarks

The standard library provides locks to help control concurrent access to resources during their lifetime. When you declare a lock object without a name, the compiler creates a temporary object that's immediately destructed rather than one that lives to the end of the enclosing scope. So, failure to assign a lock object to a variable is a mistake that effectively disables the locking mechanism (because temporary variables are transient). This rule catches simple cases of such unintended behavior.

This diagnostic only analyzes the standard lock types `std::scoped_lock`, `std::unique_lock`, and `std::lock_guard`. Warning [C26444](#) covers other unnamed RAII types.

The analyzer only analyzes simple calls to constructors. More complex initializer expressions may lead to inaccurate results in the form of missed warnings. The analyzer ignores locks passed as arguments to function calls or returned from function calls. It's unable to determine if those locks are deliberately trying to protect that function call or if their [lifetime should be extended ↗](#). To provide similar protection for types returned by a function call, annotate them with `[[nodiscard]]`. You can also annotate constructors with `[[nodiscard]]` to avoid unnamed objects of that type:

C++

```
struct X { [[nodiscard]] X(); };

void f() {
    X{}; // warning C4834
}
```

The analyzer ignores locks created as temporaries but assigned to named references to extend their lifetime.

Example

In this example, the name of the scoped lock is missing.

C++

```
void print_diagnostic(std::string_view text)
{
    auto stream = get_diagnostic_stream();
    if (stream)
    {
        std::lock_guard<std::mutex>{ diagnostic_mutex_ }; // C26441
        write_line(stream, text);
    }
}
```

To fix the error, give a name to the lock, which extends its lifetime.

C++

```
void print_diagnostic(std::string_view text)
{
    auto stream = get_diagnostic_stream();
    if (stream)
    {
        std::lock_guard<std::mutex> lock{ diagnostic_mutex_ };
        write_line(stream, text);
    }
}
```

See also

[C26444](#)

Warning C26443

Article • 10/07/2022

Overriding destructor should not use explicit 'override' or 'virtual' specifiers.

This warning was removed in Visual Studio 16.8 to reflect [changes to C.128 in the C++ Core Guidelines](#).

C++ Core Guidelines

[C.128: Virtual functions should specify exactly one of virtual, override, or final](#).

The current consensus on the Core Guidelines is to exclude destructors from the 'override explicitly' recommendation.

Notes

- The rule flags overriding destructors that explicitly use 'virtual' or 'override' specifiers.
- Destructors can still use the 'final' specifier because of its special semantics.
- Warnings show up on function definitions, not declarations. It may be confusing, since definitions don't have virtual specifiers, but the warning is still appropriate.

Code analysis name: NO_EXPLICIT_DTOR_OVERRIDE

Example: Explicit 'override'

C++

```
class Transaction {
public:
    virtual ~Transaction();
    // ...
};

class DistributedTransaction : public Transaction {
public:
    ~DistributedTransaction() override { // C26443
        // ...
    }
};
```

See also

[C.128: Virtual functions should specify exactly one of virtual, override, or final ↗](#)

Warning C26444

Article • 06/05/2023

Don't try to declare a local variable with no name (es.84).

C++ Core Guidelines

[ES.84: Don't \(try to\) declare a local variable with no name](#) ↗

An unnamed variable declaration creates a temporary object that is discarded at the end of the statement. Such temporary objects with nontrivial behavior may point to either inefficient code that allocates and immediately throws away resources or to the code that unintentionally ignores nonprimitive data. Sometimes it may also indicate plainly wrong declaration.

Remarks

- This rule detects types with a hand-written destructor or a compiler-generated destructor that transitively calls a hand-written destructor.
- This rule can flag code that invokes a nontrivial constructor of a RAII type.
- The logic skips temporaries if they're used in higher-level expressions. One example is temporaries that are passed as arguments or used to invoke a function.

Code analysis name: NO_UNNAMED_RAIID_OBJECTS

Examples

C++

```
struct A { A(int i); ~A(); };
void Foo()
{
    A{42}; // warning C26444: Don't try to declare a local variable with no
            // name (es.84).
}
```

To fix the issue, convert the temporary object to a local.

C++

```
struct A { A(int i); ~A(); };
void Foo()
{
    A guard{42}; // OK.
}
```

See also

[C26441](#)

[ES.84: Don't \(try to\) declare a local variable with no name](#) ↗

Warning C26445

Article • 10/07/2022

Do not assign `gsl::span` or `std::string_view` to a reference. They are cheap to construct and are not owners of the underlying data. (gsl.view)

A reference to `gsl::span` or `std::string_view` may be an indication of a lifetime issue.

C++ Core Guidelines

[GSL.view: Views ↗](#)

This rule catches subtle lifetime issues that may occur in code migrated from standard containers to new span and view types. Such types can be considered as "references to buffers." Using a reference to a span or view creates an extra layer of indirection. Such indirection is often unnecessary and can be confusing for maintainers. Spans are cheap to copy and can be returned by value from function calls. Obviously, such call results should never be referenced.

Remarks

- The rule detects references to `gsl::span<>`, `gsl::basic_string_span<>`, and `std::basic_string_view<>` (including aliases to instantiations).
- Currently warnings are emitted only on declarations and return statements. This rule may be extended in future to also flag function parameters.
- The implementation of this rule is lightweight doesn't attempt to trace actual lifetimes. Using of references may still make sense in some scenarios. In such cases, false positives can safely be suppressed.

Code analysis name: `NO_SPAN_REF`

Example

Reference to a temporary:

C++

```
// Old API - uses string reference to avoid data copy.  
const std::string& get_working_directory() noexcept;
```

```
// New API - after migration to C++17 it uses string view.  
std::string_view get_working_directory() noexcept;  
  
// ...  
// Client code which places an explicit reference in a declaration with auto  
specifier.  
const auto &wd = get_working_directory(); // C26445 after API update.
```

Warning C26446

Article • 10/07/2022

Prefer to use `gsl::at()` instead of unchecked subscript operator (bounds.4).

C++ Core Guidelines: [Bounds.4: Don't use standard-library functions and types that are not bounds-checked ↗](#).

Remarks

The Bounds profile of the C++ Core Guidelines tries to eliminate unsafe manipulations of memory. It helps you avoid the use of raw pointers and unchecked operations. One way to perform uniform range-checked access to buffers is to use the `gsl::at()` utility from the Guidelines Support Library. It's also a good practice to rely on standard implementations of `at()` available in STL containers.

This rule helps to find places where potentially unchecked access is performed via calls to `operator[]`. In most cases, you can replace such calls by using `gsl::at()`.

- Access to arrays of known size is flagged when a non-constant index is used in a subscript operator. Constant indices are handled by [C26483 STATIC_INDEX_OUT_OF_RANGE](#).
- The logic to warn on overloaded `operator[]` calls is more complex:
 - If the index is non-integral, the call is ignored. This also handles indexing in standard maps, since parameters in such operators are passed by reference.
 - If the operator is marked as non-throwing (by using `noexcept`, `throw()`, or `_declspec(nothrow)`), the call is flagged. We assume that if the subscript operator never throws exceptions, it either doesn't perform range checks or these checks are obscure.
 - If the operator isn't marked as non-throwing, it may be flagged if it comes from an STL container that also defines a conventional `at()` member function. Such functions are detected by simple name matching.
 - The rule doesn't warn on calls to standard `at()` functions. These functions are safe; replacing them with `gsl::at()` wouldn't bring much value.
- Indexing into `std::basic_string_view<>` is unsafe, so a warning is issued. Replace the standard `string_view` by using `gsl::basic_string_span<>`, which is always bounds-checked.
- The implementation doesn't consider range checks that user code may have somewhere in loops or branches. Here, accuracy is traded for performance. In

general, you can often replace explicit range checks by using more reliable iterators or more concise enhanced `for`-loops.

Example

This example demonstrates how the `gsl::at` function can replace an indexed reference:

C++

```
// C26446.cpp
#include <vector>
#include <gsl/gsl_util>
#include <iostream>

void fn()
{
    std::vector<int> v{1, 2, 3, 4, 5};

    // Normal bracket operators do not prevent you from accessing memory out
    // of bounds.
    std::cout << v[5] << '\n'; // C26446, prefer using gsl::at instead of
    // using operator[].

    // gsl::at prevents accessing memory out of bounds and invokes
    // std::terminate on access.
    std::cout << gsl::at(v, 5) << '\n';
}
```

Warning C26447

Article • 10/07/2022

The function is declared `noexcept` but calls function *function_name* that may throw exceptions (f.6).

C++ Core Guidelines:

F.6: If your function may not throw, declare it noexcept ↴.

Remarks

This rule amends another rule, [C26440 DECLARE_NOEXCEPT](#), which tries to find functions that are good candidates to mark as `noexcept`. In this case, the idea is that once you mark some function as `noexcept`, it must keep its contract by not invoking other code that may throw exceptions.

- The Microsoft C++ compiler already handles straightforward violations like `throw` statements in the function body (see [C4297](#)).
- The rule focuses only on function calls. It flags targets that aren't `constexpr` and that can potentially throw exceptions. In other words, they aren't marked explicitly as non-throwing by using `noexcept`, `__declspec(nothrow)`, or `throw()`.
- The compiler-generated target functions are skipped to reduce noise since exception specifications aren't always provided by the compiler.
- The checker also skips special kinds of target functions we expect you to implement as `noexcept`; this rule is enforced by [C26439 SPECIAL_NOEXCEPT](#).

Example

C++

```
#include <vector>
#include <string>
#include <iostream>

std::vector<std::string> collect(std::istream& is) noexcept
{
    std::vector<std::string> res;
    for (std::string s; is >> s;) // C26447, `operator bool()` can throw,
        std::string's allocator can throw
        res.push_back(s);           // C26447, `push_back` can throw
```

```
    return res;  
}
```

You can fix these warnings by removing `noexcept` from the function signature.

See also

[C26440 DECLARE_NOEXCEPT](#)

Warning C26448

Article • 10/07/2022

Consider using `gsl::finally` if final action is intended (gsl.util)

C++ Core Guidelines: [GSL.util: Utilities](#) ↗

The Guidelines Support Library provides a convenient utility to implement the *final action* concept. Since the C++ language doesn't support `try-finally` constructs, it became common to implement custom cleanup types that would invoke arbitrary actions on destruction. The `gsl::finally` utility is implemented in this way and provides a more uniform way to perform final actions across a code base.

There are also cases where final actions are performed in an old-fashioned C-style way by using `goto` statements (which is discouraged by [C26438 NO_GOTO](#)). It's hard to detect the exact intention in code that heavily uses `goto`, but some heuristics can help to find better candidates for cleanup.

Remarks

- This rule is lightweight and uses label names to guess about opportunities to use final action objects.
- Label names that can raise a warning contain words like "end", "final", "clean", and so on.
- Warnings appear at the `goto` statements. You may see verbose output on some occasions, but the output may help in prioritizing code, depending on its complexity.
- This rule always goes in pair with [C26438 NO_GOTO](#). Depending on the priorities, one of these rules can be disabled.

Code analysis name: `USE_GSL_FINALLY`

Example

Cleanup with multiple `goto` statements:

C++

```
void poll(connection_info info)
{
```

```

connection c = {};
if (!c.open(info))
    return;

while (c.wait())
{
    connection::header h{};
    connection::signature s{};
    if (!c.read_header(h))
        goto end;           // C26448 and C26438
    if (!c.read_signature(s))
        goto end;           // C26448 and C26438
    // ...
}

end:
    c.close();
}

```

Cleanup with multiple goto statements replaced by `gsl::finally`:

C++

```

void poll(connection_info info)
{
    connection c = {};
    if (!c.open(info))
        return;

    auto end = gsl::finally([&c] { c.close(); });
    while (c.wait())
    {
        connection::header h{};
        connection::signature s{};
        if (!c.read_header(h))
            return;
        if (!c.read_signature(s))
            return;
        // ...
    }
}

```

Warning C26449

Article • 06/05/2023

`gsl::span` or `std::string_view` created from a temporary will be invalid when the temporary is invalidated (`gsl.view`)

C++ Core Guidelines: [GSL.view: Views ↗](#).

Spans and views are convenient and lightweight types that allow you to reference memory buffers. But they must be used carefully: while their interface looks similar to standard containers, their behavior is more like the behavior of pointers and references. They don't own data and must never be constructed from temporary buffers. This check focuses on cases where source data is temporary, while a span or view isn't. This rule can help to avoid subtle but dangerous mistakes made when legacy code gets modernized and adopts spans or views. There's another check that handles a slightly different scenario involving span references: [C26445 NO_SPAN_REF](#).

Consider using [C26815](#) and [C26816](#). Those warnings are more general versions of this warning.

Remarks

- This rule warns on places where constructors get invoked for spans or views and the source data buffer belongs to a temporary object created in the same statement. This check includes:
 - implicit conversions in return statements;
 - implicit conversions in ternary operators;
 - explicit conversions in `static_cast` expressions;
 - function calls that return containers by value.
- Temporaries created for function call arguments aren't flagged. It's safe to pass spans from such temporaries if target functions don't retain data pointers in external variables.
- If spans or views are themselves temporaries, the rule skips them.
- Data tracking in the checker has certain limitations; therefore complex scenarios involving multiple or obscure reassignments may not be handled.

Code analysis name: `NO_SPAN_FROM_TEMPORARY`

Example

Subtle difference in result types:

C++

```
// Returns a predefined collection. Keeps data alive.  
gsl::span<const sequence_item> get_seed_sequence() noexcept;  
  
// Returns a generated collection. Doesn't own new data.  
std::vector<sequence_item> get_next_sequence(gsl::span<const  
sequence_item>);  
  
void run_batch()  
{  
    auto sequence = get_seed_sequence();  
    while (send(sequence))  
    {  
        sequence = get_next_sequence(sequence); // C26449  
        // ...  
    }  
}
```

To fix the issue, make sure the view is created from an object that lives at least as long as the view itself. Sometimes a solution can be achieved by copying the data, other times some APIs need to be redesigned to share a reference to an object that lives long enough instead of returning a temporary copy.

See also

[C26815](#)

[C26816](#)

Warning C26450

Article • 06/05/2023

Arithmetic overflow: 'operator' operation causes overflow at compile time. Use a wider type to store the operands (io.1)

Remarks

This warning indicates that an arithmetic operation was provably lossy at compile time. It can be asserted when the operands are all compile-time constants. Currently, we check left shift, multiplication, addition, and subtraction operations for such overflows.

Warning [C4307](#) is a similar check in the Microsoft C++ compiler.

Code analysis name: `RESULT_OF_ARITHMETIC_OPERATION_PROVABLY_LOSSY`

Examples

C++

```
int multiply()
{
    const int a = INT_MAX;
    const int b = 2;
    int c = a * b; // C26450 reported here
    return c;
}
```

To correct this warning, use the following code.

C++

```
long long multiply()
{
    const int a = INT_MAX;
    const int b = 2;
    long long c = static_cast<long long>(a) * b; // OK
    return c;
}
```

See also

[26451](#)

[26452](#)

[26453](#)

[26454](#)

[ES.103: Don't overflow ↗](#)

Warning C26451

Article • 06/05/2023

Arithmetic overflow: Using operator '*operator*' on a *size-a* byte value and then casting the result to a *size-b* byte value. Cast the value to the wider type before calling operator '*operator*' to avoid overflow (io.2)

This warning indicates incorrect behavior that results from integral promotion rules and types larger than the ones in which arithmetic is typically performed.

Remarks

Code analysis detects when an integral value gets shifted left, multiplied, added, or subtracted, and the result gets cast to a wider integral type. If the operation overflows the narrower integral type, then data is lost. You can prevent this loss by casting the value to a wider type before the arithmetic operation.

Code analysis name: RESULT_OF_ARITHMETIC_OPERATION_CAST_TO_LARGER_SIZE

Examples

The following code generates this warning:

```
C++  
  
void leftshift(int i) noexcept  
{  
    unsigned long long x;  
    x = i << 31; // C26451 reported here  
  
    // code  
}
```

To correct this warning, use the following code:

```
C++  
  
void leftshift(int i) noexcept  
{  
    unsigned long long x;  
    x = static_cast<unsigned long long>(i) << 31; // OK
```

```
// code  
}
```

See also

[26450](#)

[26452](#)

[26453](#)

[26454](#)

[ES.103: Don't overflow ↗](#)

Warning C26452

Article • 06/05/2023

Arithmetic overflow: Left shift count is negative or greater than or equal to the operand size, which is undefined behavior (io.3)

Remarks

This warning indicates the shift count is negative, or greater than or equal to the number of bits in the shifted operand. Either case results in undefined behavior.

Warning [C4293](#) is a similar check in the Microsoft C++ compiler.

Code analysis name: `SHIFT_COUNT_NEGATIVE_OR_TOO_BIG`

Example

C++

```
unsigned long long combine(unsigned lo, unsigned hi)
{
    return (hi << 32) | lo; // C26252 here
}
```

To correct this warning, use the following code:

C++

```
unsigned long long combine(unsigned lo, unsigned hi)
{
    return (static_cast<unsigned __int64>(hi) << 32) | lo; // OK
}
```

See also

[26450](#)

[26451](#)

[26453](#)

[26454](#)

[ES.101: Use unsigned types for bit manipulation ↴](#)

[ES.102: Use signed types for arithmetic ↴](#)

Warning C26453

Article • 06/05/2023

Arithmetic overflow: Left shift of a negative signed number is undefined behavior
(io.4)

Remarks

This warning indicates the code left shifts a negative signed integral value, which is nonportable and triggers implementation defined behavior.

Code analysis name: LEFTSHIFT_NEGATIVE_SIGNED_NUMBER

Example

C++

```
void leftshift(int shiftCount)
{
    const auto result = -1 << shiftCount; // C26453 reported here

    // code
}
```

To correct this warning, use the following code:

C++

```
void leftshift(int shiftCount)
{
    const auto result = ~0u << shiftCount; // OK

    // code
}
```

See also

[26450](#)

[26451](#)

[26452](#)

[26454](#)

ES.101: Use unsigned types for bit manipulation ↗

ES.102: Use signed types for arithmetic ↗

Warning C26454

Article • 06/05/2023

Arithmetic overflow: 'operator' operation produces a negative unsigned result at compile time

Remarks

This warning indicates that the subtraction operation produces a negative result that was evaluated in an unsigned context, which can result in unintended overflows.

Code analysis name: RESULT_OF_ARITHMETIC_OPERATION_NEGATIVE_UNSIGNED

Example

C++

```
unsigned int negativeunsigned()
{
    const unsigned int x = 1u - 2u; // C26454 reported here
    return x;
}
```

To correct this warning, use the following code:

C++

```
unsigned int negativeunsigned()
{
    const unsigned int x = 4294967295; // OK
    return x;
}
```

See also

[26450](#)

[26451](#)

[26452](#)

[26453](#)

[ES.106: Don't try to avoid negative values by using unsigned ↗](#)

Warning C26455

Article • 06/26/2023

Default constructor should not throw. Declare it '`noexcept`' (f.6)

The C++ Core Guidelines suggest that default constructors shouldn't do anything that can throw. When the default constructor can throw, all code that relies on a properly instantiated object may also throw.

Remarks

Consider the default constructors of the STL types, like `std::vector`. In these implementations, the default constructors initialize internal state without making allocations. In the `std::vector` case, the size is set to 0 and the internal pointer is set to `nullptr`. The same pattern should be followed for all default constructors.

Code analysis name: `DEFAULT_CTOR_NOEXCEPT`

See also

[C26439](#)

[C++ Core Guidelines F.6 ↗](#)

Warning C26456

Article • 10/07/2022

Operator '*symbol_1*' hides a non-virtual operator '*symbol_2*' (c.128)

Remarks

Hiding base methods that aren't virtual is error prone and makes code harder to read.

Code analysis name: DONT_HIDE_OPERATORS

See also

[C++ Core Guideline c.128 ↗](#)

Warning C26457

Article • 10/07/2022

(`void`) should not be used to ignore return values, use '`std::ignore =`' instead
(es.48)

Remarks

Excerpt from the [C++ Core Guideline ES.48](#):

Never cast to (`void`) to ignore a `[[nodiscard]]` return value. If you deliberately want to discard such a result, first think hard about whether that is really a good idea (there is usually a good reason the author of the function or of the return type used `[[nodiscard]]` in the first place). If you still think it's appropriate and your code reviewer agrees, use `std::ignore =` to turn off the warning which is simple, portable, and easy to grep.

Code analysis name: `USE_STD_IGNORE_INSTEAD_OF_VOID_CAST`

Example

Use `std::ignore` instead of cast to `void`:

```
C++  
  
struct S{};  
[[nodiscard]] S gets();  
  
void function() {  
    (void) gets(); // C26457  
    std::ignore = gets(); // OK  
}
```

Warning C26459

Article • 04/12/2024

You called an STL function '%function%' with a raw pointer parameter at position '%position%' that may be unsafe - this relies on the caller to check that the passed values are correct. Consider wrapping your range in a `gsl::span` and pass as a span iterator (stl.1)

Remarks

Out of bound writes are one of the leading causes of remote code execution vulnerabilities. One remedy is to use bounds checked data structures like `gsl::span`. This warning identifies cases where Standard Template Library (STL) algorithms operate on raw pointers as output ranges. Raw pointers aren't bounds checked. To prevent vulnerabilities, use `gsl::span` instead.

Code analysis name: NO_RAW_POINTER_IN_STL_RANGE_CHECKED

Example

The following code demonstrates undefined behavior because there isn't any bounds checking and `copy_if` writes beyond the provided storage.

C++

```
void f()
{
    std::vector<int> myints = { 10, 20, 30, 40, 50, 60, 70 };
    int mydestinationArr[7] = { 10, 20, 80 };

    std::copy_if(myints.begin(), myints.end(), mydestinationArr, [](int i) {
        return !(i<0); });
    // Warning: C26459
}
```

To fix the warning, use `gsl::span` to make sure the output range is bounds checked:

C++

```
void f()
{
    std::vector<int> myints = { 10, 20, 30, 40, 50, 60, 70 };
    int mydestinationArr[7] = { 10, 20, 80 };
    gsl::span<int> mySpan{mydestinationArr};
```

```
    std::copy_if(myints.begin(), myints.end(), mySpan.begin(), [](int i) {
return !(i<0); }); // No warning
}
```

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Warning C26460

Article • 10/07/2022

The reference argument '*argument*' for function '*function*' can be marked as `const` (con.3).

Remarks

Passing an object by reference indicates that the function has the potential modify the object. If that isn't the intent of the function, it's better to mark the argument as a const reference.

Code analysis name: `USE_CONST_REFERENCE_ARGUMENTS`

Example

C++

```
struct MyStruct
{
    void MemberFn1() const;
    void MemberFn2();
};

void Function1_Helper(const MyStruct&);
void Function1(MyStruct& myStruct) // C26460, see comments below.
{
    myStruct.MemberFn1();          // The member function is marked as const
    Function1_Helper(myStruct);   // Function1_Helper takes a const reference
}

void Function2(MyStruct& myStruct)
{
    myStruct.MemberFn2(); // MemberFn2 is non-const and has the potential to
                         // modify data
}
```

See also

[C++ Core Guidelines con.3 ↗](#).

Warning C26461

Article • 10/07/2022

The pointer argument '*argument*' for function '*function*' can be marked as a pointer to `const` (con.3).

Remarks

A function with a `T*` argument has the potential to modify the value of the object. If that isn't the intent of the function, it's better to make the pointer a `const T*` instead.

Code analysis name: `USE_CONST_POINTER_ARGUMENTS`

Example

C++

```
struct MyStruct
{
    void MemberFn1() const;
    void MemberFn2();
};

void Function1_Helper(const MyStruct* myStruct);
void Function1(MyStruct* myStruct) // C26461, neither of the operations on
myStruct would modify the pointer's value.
{
    if (!myStruct)
        return;

    myStruct->MemberFn1();      // The member function is const
    Function1_Helper(myStruct); // Function1_Helper takes a const
}

void Function2(MyStruct* myStruct)
{
    if (!myStruct)
        return;

    myStruct->MemberFn2(); // The member function is non-const, so no C26461
will be issued
}
```

See also

C++ Core Guidelines con.3 ↴ .

Warning C26462

Article • 10/07/2022

The value pointed to by '*variable*' is assigned only once, mark it as a pointer to `const` (con.4).

Remarks

Pointers to variables whose values remain unchanged should be marked as `const`.

Code analysis name: `USE_CONST_POINTER_FOR_VARIABLE`

Example

C++

```
void useVal(int val);

void function1(int* ptr)
{
    int* p = ptr; // C26462, the value pointed to by p is unmodified
    ptr = nullptr;

    useVal(*p);
}
```

See also

[C++ Core Guidelines con.4 ↗](#).

Warning C26463

Article • 10/07/2022

The elements of array '%array%' are assigned only once, mark elements `const` (con.4)

Remarks

- This rule is currently not implemented in CppCoreCheck.

Code analysis name: `USE_CONST_FOR_ELEMENTS`

See also

[C++ Core Guidelines con.4 ↗](#).

Warning C26464

Article • 10/07/2022

The values pointed to by elements of array '*array*' are assigned only once, mark elements as pointer to `const` (con.4).

Remarks

- This rule is currently not implemented in CppCoreCheck.

Code analysis name: `USE_CONST_POINTER_FOR_ELEMENTS`

See also

[C++ Core Guidelines con.4 ↗](#).

Warning C26465

Article • 10/07/2022

Don't use `const_cast` to cast away `const`. `const_cast` is not required; constness or volatility is not being removed by this conversion.

See also

C++ Core Guidelines Type.3 ↗

Example

C++

```
void function(int* const constPtrToInt)
{
    auto p = const_cast<int*>(constPtrToInt); // C26465, const is not being
removed
}
```

Warning C26466

Article • 10/07/2022

Don't use `static_cast` downcasts. A cast from a polymorphic type should use `dynamic_cast`.

See also

[C++ Core Guidelines Type.2 ↗](#)

Example

C++

```
struct Base {
    virtual ~Base();
};

struct Derived : Base {};

void bad(Base* pb)
{
    Derived* test = static_cast<Derived*>(pb); // C26466
}

void good(Base* pb)
{
    if (Derived* pd = dynamic_cast<Derived*>(pb))
    {
        // ... do something with Derived*
    }
    else
    {
        // ... do something with Base*
    }
}
```

Warning C26471

Article • 10/07/2022

Don't use `reinterpret_cast`. A cast from `void*` can use `static_cast` (type.1)

Remarks

Code analysis name: `NO_REINTERPRET_CAST_FROM_VOID_PTR`

Example

C++

```
void function(void* pValue)
{
    {
        int* pointerToInt = reinterpret_cast<int*>(pValue); // C26471, use
static_cast instead
    }
    {
        int* pointerToInt = static_cast<int*>(pValue); // Good
    }
}
```

See also

[C++ Core Guidelines Type.1 ↗](#)

Warning C26472

Article • 10/07/2022

Don't use a `static_cast` for arithmetic conversions. Use brace initialization,

`gsl::narrow_cast`, or `gsl::narrow`.

C++ Core Guidelines: [Type.1 ↗](#): Avoid casts

This rule helps to find places where static casts are used to convert between integral types. These casts are unsafe because the compiler wouldn't warn if any data loss occurs. Brace initializers are better for the cases where constants are used, and a compiler error is desired. There are also utilities from the Guidelines Support Library that help to describe intentions clearly:

- `gsl::narrow` ensures lossless conversion and throws `gsl::narrowing_error` if it's not possible.
- `gsl::narrow_cast` clearly states that conversion can lose data and it's acceptable.

Remarks

- This rule is implemented only for static casts. Using of C-style casts is discouraged.

Code analysis name: `NO_CASTS_FOR_ARITHMETIC_CONVERSION`

Example

Unhandled unexpected data:

C++

```
rgb from_24bit(std::uint32_t v) noexcept {
    return {
        static_cast<std::uint8_t>(v >> 16),           // C26472, what if top
        byte is non-zero?
        static_cast<std::uint8_t>((v >> 8) & 0xFF), // C26472
        static_cast<std::uint8_t>(v & 0xFF)           // C26472
    };
}
```

Unhandled unexpected data, safer version:

C++

```
rgb from_24bit(std::uint32_t v) noexcept {
    return {
        gsl::narrow<std::uint8_t>(v >> 16),
        gsl::narrow_cast<std::uint8_t>((v >> 8) & 0xFF),
        gsl::narrow_cast<std::uint8_t>(v & 0xFF)
    };
}
```

Warning C26473

Article • 10/07/2022

Don't cast between pointer types where the source type and the target type are the same.

C++ Core Guidelines: [Type.1 ↗](#): Avoid casts

This rule helps to remove unnecessary or suspicious casts. Obviously, when a type is converted to itself, such a conversion is ineffective. Yet the fact that the cast is used may indicate a subtle design issue or a potential for regression if types change in future. It's always safer to use as few casts as possible.

Remarks

- This rule is implemented for static casts and reinterpret casts, and checks only pointer types.

Example

dangerously generic lookup

C++

```
gsl::span<server> servers_;
```

```
template<class T>
server* resolve_server(T tag) noexcept {
    auto p = reinterpret_cast<server*>(tag); // C26473, also 26490
NO_REINTERPRET_CAST
    return p >= &(*servers_.begin()) && p < &(*servers_.end()) ? p :
nullptr;
}

void promote(server *s, int index) noexcept {
    auto s0 = resolve_server(s);
    auto s1 = resolve_server(index);
    if (s0 && s1)
        std::swap(s0, s1);
}
```

dangerously generic lookup - reworked

C++

```
// ...
server* resolve_server(server *p) noexcept {
    return p >= &(*servers_.begin()) && p < &(*servers_.end()) ? p :
nullptr;
}

server* resolve_server(ptrdiff_t i) noexcept {
    return !servers_.empty() && i >= 0 && i < servers_.size() ? &servers_[i]
: nullptr;
}
// ...
```

Warning C26474

Article • 10/07/2022

Don't cast between pointer types when the conversion could be implicit.

C++ Core Guidelines:

Type.1 ↗: Avoid casts

In some cases, implicit casts between pointer types are safe and don't require you to write a specific cast expression. This rule finds instances of unnecessary casts you can safely remove.

Remarks

The rule ID should be interpreted as "An implicit cast isn't used where it's acceptable."

This rule is only applicable to pointers. It checks static casts and reinterpret casts.

These cases are acceptable pointer conversions that shouldn't use explicit cast expressions:

- conversion to `nullptr_t`;
- conversion to `void*`;
- conversion from a derived type to its base when invoking a base member function that's not hidden by the derived type.

Example 1

An unnecessary conversion hides a logic error in this example:

C++

```
template<class T>
bool register_buffer(T buffer) {
    auto p = reinterpret_cast<void*>(buffer); // C26474, also 26490
NO_REINTERPRET_CAST
    // To fix, declare buffer as T*, and use this to define p:
    // auto p = buffer;
    return buffers_.insert(p).second;
}

void merge_bytes(std::uint8_t *left, std::uint8_t *right)
{
```

```
if (left && register_buffer(*left)) { // Unintended dereference!
    // ...
    if (right && register_buffer(right)) {
        // ...
    }
}
```

Example 2

This example demonstrates using casts to access base-class member functions:

C++

```
struct struct_1
{
    void foo();
    void bar();
};

struct struct_2 : struct_1
{
    void foo(); // this definition hides struct_1::foo
};

void fn(struct_2* ps2)
{
    static_cast<struct_1*>(ps2)->foo(); // This cast is necessary to access
struct_1::foo
                                // Alternatively, use ps2-
>struct_1::foo();
    static_cast<struct_1*>(ps2)->bar(); // This cast is unnecessary and can
be done implicitly
}
```

Warning C26475

Article • 10/07/2022

Do not use function style C-casts.

C++ Core Guidelines: [ES.49 ↗](#): If you must use a cast, use a named cast

Function-style casts (for example, `int(1.1)`) are another form of C-style casts (like `(int)1.1`), which have questionable safety. Specifically, the compiler doesn't try to check if any data loss can occur either in C-casts or in function casts. In both cases, it's better either to avoid casting or to use a braced initializer if possible. If neither works, static casts may be suitable, but it's still better to use utilities from the Guidelines Support Library:

- `gsl::narrow` ensures lossless conversion and throws `gsl::narrowing_error` if it's not possible.
- `gsl::narrow_cast` clearly states that conversion can lose data and it's acceptable.

Remarks

- This rule fires only for constants of primitive types. The compiler can clearly detect data loss in these cases and emits an error if a braced initializer is used. The cases that would require run-time execution are flagged by [C26493 NO_CSTYLE_CAST](#).
- Default initializers aren't flagged (for example `int()`).

Example

Dangerous conversion example:

C++

```
constexpr auto planck_constant = float( 6.62607004082e-34 ); // C26475
```

Compiler error for dangerous conversion, detecting potential data loss:

C++

```
constexpr auto planck_constant = float{ 6.62607004082e-34 }; // Error C2397
```

To correct the dangerous conversion, use an appropriately sized primitive type:

C++

```
constexpr auto planck_constant = double{ 6.62607004082e-34 };
```

Warning C26476

Article • 10/07/2022

Expression/symbol '*name*' uses a naked union '*union*' with multiple type pointers:
Use variant instead (type.7)

Remarks

`std::variant` provides a type-safe alternative to `union` and should be preferred in modern code.

Code analysis name: `USE_VARIANT`

See also

[C++ Core Guideline C.181 ↗](#)

Warning C26477

Article • 10/07/2022

Use '`nullptr`' rather than 0 or `NULL` (es.47)

Remarks

`nullptr` has a special type `nullptr_t` that allows overloads with special null handling. Using `0` or `NULL` in place of `nullptr` bypasses the type safety and deduction that `nullptr` provides.

Code analysis name: `USE_NULLPTR_NOT_CONSTANT`

See also

[C++ Core Guideline ES.47 ↗](#)

Warning C26478

Article • 10/24/2023

Don't use `std::move` on constant variables. (es.56)

Remarks

This warning is to indicate that the use of `std::move` not consistent with how `std::move` is intended to be used.

Because `const` objects can't be moved, calling `std::move` on them has no effect. This pattern can result in unintended copies.

Code analysis name: `NO_MOVE_OP_ON_CONST`

Example

C++

```
struct node
{
    node* next;
    int id;
};

void foo(const node& n)
{
    const node local = std::move(n); // C26478 reported here
    // ...
}
```

To fix the issue, remove the redundant `std::move`.

See also

[ES.56 - Write `std::move\(\)` only when you need to explicitly move an object to another scope ↗](#)

Warning C26479

Article • 10/24/2023

Don't use std::move to return a local variable. (f.48)

Remarks

The `return` statement is the last use of a local variable, so the compiler uses move semantics to return it whenever possible. Adding a `std::move` is redundant in this scenario. Moreover, redundant `std::moves` can prevent copy elision.

Code analysis name: `NO_MOVE_RET_ON_LOCALS`

Example 1

C++

```
S foo()
{
    S local1{};
    return std::move(local1); // Warning: C26479
}
```

To fix this issue, remove the redundant `std::move`:

C++

```
S foo()
{
    S local1{};
    return local1; // No warning
}
```

See also

[F.48 - Don't return std::move\(local\)](#) ↗ [ES.56 - Write std::move\(\) only when you need to explicitly move an object to another scope](#) ↗

Warning C26481

Article • 10/07/2022

Don't use pointer arithmetic. Use span instead (bounds.1).

Remarks

This check supports the [C++ Core Guidelines](#) rule I.13: *Do not pass an array as a single pointer*. Whenever raw pointers are used in arithmetic operations they should be replaced with safer kinds of buffers, such as `span<T>` or `vector<T>`.

This check is more restrictive than I.13: it doesn't skip `zstring` or `czstring` types.

C26481 and [C26485](#) come from the [Bounds Safety Profile](#) rules. These rules were implemented in the first release of the C++ Core Guidelines Checker. They're applicable to the raw pointers category since they help to avoid unsafe use of raw pointers.

Example

This sample results in a warning for pointer arithmetic.

C++

```
// c26481_bad.cpp
// compile using:
// set Esp.Extensions=CppCoreCheck.dll
// cl /W3 /EHsc /permissive- /analyze /analyze:plugin EspXEngine.dll
// /analyze:ruleset "%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Rule
Sets\CppCoreCheckBoundsRules.ruleset" c26481_bad.cpp

int main() noexcept
{
    int * from_array = new int(10);
    int * later_array = from_array + 1;
    delete[](from_array);
}
```

Warning C26482

Article • 10/07/2022

Only index into arrays using constant expressions.

See also

[C++ Core Guidelines Bounds.2 ↗](#)

Example

C++

```
int getSomeIndex();

void function(int* p, int count)
{
    p[getSomeIndex()] = 0; // C26482, Only index into arrays using constant
                          // expressions
}
```

Warning C26483

Article • 10/07/2022

Value 'value' is outside the bounds (0, 'bound') of variable 'variable'. Only index into arrays using constant expressions that are within bounds of the array (bounds.2).

See also

[C++ Core Guidelines Bounds.2 ↗](#)

Example

C++

```
void function()
{
    std::array<int, 3> arr1 { 1, 2, 3 };
    arr1[3] = 4; // C26483, 3 is outside the bounds of the array

    int arr2[] { 1, 2, 3 };
    arr2[3] = 4; // C26483, 3 is outside the bounds of the array
}
```

Warning C26485

Article • 10/07/2022

Expression 'array-name': No array to pointer decay (bounds.3).

Remarks

Like [C26481](#), this check helps to enforce the [C++ Core Guidelines](#) rule [I.13](#): *Do not pass an array as a single pointer*. The rule detects places where static array type information is lost from decay to a raw pointer. The `zstring` and `czstring` types aren't excluded.

C26481 and C26485 come from the [Bounds Safety Profile](#) rules. These rules were implemented in the first release of the C++ Core Guidelines Checker. They're applicable to the raw pointers category since they help to avoid unsafe use of raw pointers.

Example

This sample results in two warnings for array to pointer decay in the call to `memcpy`.

```
C++  
  
// c26485_bad.cpp  
// compile using:  
// set Esp.Extensions=CppCoreCheck.dll  
// cl /W4 /EHsc /permissive- /analyze /analyze:plugin EspXEngine.dll  
/analyze:ruleset "%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Rule  
Sets\CppCoreCheckBoundsRules.ruleset" c26485_bad.cpp  
#include <cstring>  
constexpr int array_length = 10;  
  
int main() noexcept  
{  
    int const from_array[array_length] = { 4, 3, 2, 1, 0, 9, 8, 7, 6, 5 };  
    int to_array[array_length] = {};  
  
    if (nullptr != memcpy(to_array, from_array, sizeof(int) * array_length))  
        return 0;  
    return 1;  
}
```

To address this issue, avoid calls that take pointer parameters, but don't manage bounds information. Use of such functions is often error-prone. Prefer C++ standard library calls to C runtime library functions. Consider using `gsl::span` or `std::vector` in your own

functions. An explicit cast to the decayed pointer type prevents the warning, but it doesn't prevent buggy code.

Warning C26486

Article • 10/07/2022

Don't pass a pointer that may be invalid (dangling) as a parameter to a function.

C++

```
void use(int*);  
  
void ex1()  
{  
    int* px;  
    {  
        int x;  
        px = &x;  
    }  
  
    use(px); // px is a dangling pointer  
}
```

Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

- Never dereference an invalid (dangling) or known-null pointer
- Never return (either formal return or out parameter) any pointer from a function.
- Never pass an invalid (dangling) pointer to any function.

Code analysis name: LIFETIMES_FUNCTION_PRECONDITION_VIOLATION

See also

- [C++ Core Guidelines Lifetimes Paper ↗](#)

Warning C26487

Article • 10/07/2022

Don't allow a function to return an invalid pointer, either through a formal return statement or through output parameters.

C++

```
int* ex1(int a)
{
    return &a;      // returns a dangling pointer to the stack variable 'a'
}

void ex2(int a, int** out)
{
    *out = &a;      // 'out' contains a dangling pointer to the stack variable
    'a'
}
```

Remarks

The Lifetime guidelines from the C++ Core Guidelines outline a contract that code can follow which enables more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

- Never dereference an invalid (dangling) or known-null pointer.
- Never return, either by formal return statement or out parameter, any dangling pointer from a function.
- Never pass an invalid (dangling) pointer to any function.

An invalid pointer is *dangling* when it points to something that isn't there anymore. For example, any pointer to a local variable or parameter, once it's gone out of scope. Or, a pointer to a resource that's been deleted. Even a pointer to a static can be dangling, if the value gets changed before it can be used. A dangling pointer was once valid; that's what distinguishes it from other kinds of invalid pointers, such as an uninitialized pointer, or `nullptr`.

See also

- [C++ Core Guidelines Lifetimes Paper ↗](#)

Warning C26488

Article • 10/07/2022

Don't dereference a pointer that may be null.

C++

```
void ex1()
{
    int* px = nullptr;

    if (px)      // notice the condition is incorrect
        return;

    *px = 1;      // 'px' known to be null here
}
```

Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

1. Never dereference an invalid (dangling) or known-null pointer
2. Never return (either formal return or out parameter) any pointer from a function.
3. Never pass an invalid (dangling) pointer to any function.

See also

- [C++ Core Guidelines Lifetimes Paper ↗](#)

Warning C26489

Article • 10/07/2022

Don't dereference a pointer that may be invalid.

C++

```
int ex1()
{
    int* px;

    {
        int x = 0;
        px = &x;
    }

    return *px;    // 'px' was invalidated when 'x' went out of scope.
}
```

Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

1. Never dereference an invalid (dangling) or known-null pointer
2. Never return (either formal return or out parameter) any pointer from a function.
3. Never pass an invalid (dangling) pointer to any function.

See also

[C++ Core Guidelines Lifetimes Paper ↗](#)

Warning C26490

Article • 10/07/2022

Don't use `reinterpret_cast`.

See also

[C++ Core Guidelines Type.1 ↗](#).

Example

C++

```
void function(void* ptr)
{
    std::size_t val = reinterpret_cast<std::size_t>(ptr); // C26490, Don't
use reinterpret_cast
}
```

Warning C26491

Article • 10/07/2022

Don't use `static_cast` downcasts. See [C++ Core Guidelines Type.2](#).

Warning C26492

Article • 10/07/2022

Don't use `const_cast` to cast away `const`.

See also

[C++ Core Guidelines Type.3 ↗](#).

Example

C++

```
void function(const int* constIntPtr)
{
    int* intPtr = const_cast<int*>(constIntPtr); // C26492, Do not use
const_cast to cast away const
}
```

Warning C26493

Article • 10/07/2022

Don't use C-style casts.

See also

[C++ Core Guidelines Type.4 ↗](#).

Example

C++

```
void function(const int* const_i)
{
    int* nonconst_i = (int*)const_i; // C26493 Don't use C-style casts
    int i = (int)*const_i; // C26493 Don't use C-style casts
}
```

Warning C26494

Article • 06/05/2023

Variable 'variable' is uninitialized. Always initialize an object.

Remarks

This check requires local variables to be initialized at the declaration or in the following statement.

Example

C++

```
#include <iostream>
void function()
{
    int myVal; // C26494, Variable is uninitialized
    std::cout << myVal; // C6001
}
```

To fix the issue, initialize the variable at the declaration.

C++

```
#include <iostream>
void function()
{
    int myVal{};
    std::cout << myVal;
}
```

See also

[ES.20: Always initialize an object ↗](#)

[C++ Core Guidelines Type.5 ↗](#)

Warning C26495

Article • 06/05/2023

Variable '*variable*' is uninitialized. Always initialize a member variable (type.6).

Remarks

A member variable isn't initialized by a constructor or by an initializer. Make sure all variables are initialized by the end of construction. For more information, see C++ Core Guidelines [Type.6](#) and [C.48](#).

This check is intra-procedural. Whenever there's a function call to a nonconst member function, the check assumes that this member function initializes all of the members. This heuristic can result in missed errors and is in place to avoid false positive results. Moreover, when a member is passed by nonconst reference to a function, the check assumes that the function initializes the member.

Code analysis name: MEMBER_UNINIT

Example

The following sample generates warning C26495 because the member variable `value` isn't initialized when a `MyStruct` object is created.

C++

```
struct MyStruct
{
    int value;
    MyStruct() {} // C26495, MyStruct::value is uninitialized
};
```

To resolve the issue, you can add in-class initialization to all of the member variables.

C++

```
struct MyStruct
{
    int value{}; // empty brace initializer sets value to 0
    MyStruct() {} // no warning, MyStruct::value is set via default member
                  // initialization
};
```

See also

[C26494](#)

Warning C26496

Article • 10/07/2022

The variable '*variable*' is assigned only once, mark it as `const`.

See also

[C++ Core Guidelines con.4 ↗](#).

Example

C++

```
int GetTheNumber();
int GiveMeTheNumber(int);

void function1()
{
    int theNumber = GetTheNumber(); // C26496, 'theNumber' is never assigned
    to again, so it can be marked as const
    std::cout << theNumber << '\n';

    GiveMeTheNumber(theNumber);
    // ...
}
```

Warning C26497

Article • 10/07/2022

This function *function-name* could be marked `constexpr` if compile-time evaluation is desired (f.4).

See also

[C++ Core Guidelines F.4 ↗](#).

Example

C++

```
const int GetTheAnswer(int x) noexcept { return 42 + x; } // Could be marked
constexpr

void function1() noexcept
{
    const int theAnswer = GetTheAnswer(0);
}
```

To reduce code analysis noise on new code, this warning isn't issued if the function has an empty implementation.

C++

```
int function1(){ // no C26497
    return 1;
}
void function2(){}
// no C26497
```

Warning C26498

Article • 06/26/2023

The function '*function*' is `constexpr`, mark variable '*variable*' `constexpr` if compile-time evaluation is desired (con.5)

This rule helps to enforce Con.5 from the [C++ Core Guidelines](#): use `constexpr` for values that can be computed at compile time.

Remarks

The warning is triggered by assigning the result of a `constexpr` function to any non-`constexpr` variable whose value doesn't change after the initial assignment.

Code analysis name: `USE_CONSTEXPR_FOR_FUNCTIONCALL`

Example

This sample code shows where C26498 may appear:

C++

```
constexpr int getMyValue()
{
    return 1;
}

void foo()
{
    constexpr int val0 = getMyValue(); // no C26498
    const int val1 = getMyValue();    // C26498, C26814
    int val2 = getMyValue();         // C26498, value is never modified
    int val3 = getMyValue();         // no C26498, val3 is assigned to
                                    // below.
    val3 = val3 * val2;
}
```

To fix the issues, mark `val1` and `val2` `constexpr`:

C++

```
constexpr int getMyValue()
{
    return 1;
```

```
}
```

```
void foo()
{
    constexpr int val0 = getMyValue(); // OK
    constexpr int val1 = getMyValue(); // OK
    constexpr int val2 = getMyValue(); // OK
    int val3 = getMyValue();           // OK
    val3 = val3 * val2;
}
```

See also

[C26497](#)

[C26814](#)

[Con.5 ↗](#)

Warning C26812

Article • 10/07/2022

The enum type '*type-name*' is unscoped. Prefer 'enum class' over 'enum' (Enum.3)

Remarks

Prefer `enum class` over `enum` to prevent pollution in the global namespace.

Code analysis name: `PreferScopedEnum`

Example

The following example is from the C++ Core Guidelines:

C++

```
void Print_color(int color);

enum Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum Product_info { Red = 0, Purple = 1, Blue = 2 };

Web_color webby = Web_color::blue;

// Clearly at least one of these calls is buggy.
Print_color(webby);           // C26812
Print_color(Product_info::Blue); // C26812
```

These warnings are corrected by declaring the enum as enum class:

C++

```
void Print_color(int color);

enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Product_info { Red = 0, Purple = 1, Blue = 2 };

Web_color webby = Web_color::blue;
Print_color(webby); // Error: cannot convert Web_color to int.
Print_color(Product_info::Red); // Error: cannot convert Product_info to
int.
```

See also

Enum.3 Prefer enum class over enum ↗

Warning C26814

Article • 10/07/2022

The const variable '*variable*' can be computed at compile time. Consider using

`constexpr` (con.5)

Remarks

Use `constexpr` for constants whose value is known at compile time. (Con.5)

Code analysis name: `USE_CONSTEXPR_RATHER_THAN_CONST`

Example

C++

```
const int foo = 1234; // C26814 reported here.
constexpr int getMagicNumber()
{
    return 42;
}

void bar()
{
    const int myval = 3; // C26814 reported here
    const int magicNumber = getMagicNumber(); // C26814 reported here.
}
```

See also

[Con.5 Use `constexpr` for all variables that can be computed at compile time ↗](#)

Warning C26815

Article • 06/05/2023

The pointer is dangling because it points at a temporary instance that was destroyed. (ES.65)

Remarks

The created pointer or view refers to an unnamed temporary object that is destroyed at the end of the statement. The pointer or view will dangle.

This check recognizes views and owners from the C++ Standard Template Library (STL). To teach this check about user-authored types, use the `[[msvc::lifetimebound]]` annotation. The `[[msvc::lifetimebound]]` support is new in MSVC 17.7.

Code analysis name: `LIFETIME_LOCAL_USE_AFTER_FREE_TEMP`

Example

Consider the following code compiled in a C++ version before C++23:

```
C++

std::optional<std::vector<int>> getTempOptVec();

void loop() {
    // Oops, the std::optional value returned by getTempOptVec gets deleted
    // because there is no reference to it.
    for (auto i : *getTempOptVec()) // warning C26815
    {
        // do something interesting
    }
}

void views()
{
    // Oops, the 's' suffix turns the string literal into a temporary
    std::string.
    std::string_view value("This is a std::string"s); // warning C26815
}

struct Y { int& get() [[msvc::lifetimebound]]; };
void f() {
    int& r = Y{}.get(); // warning C26815
}
```

These warnings can be fixed by extending the lifetime of the temporary object.

C++

```
std::optional<std::vector<int>> getTempOptVec();

void loop() {
    // Fixed by extending the lifetime of the std::optional value by giving
    it a name.
    auto temp = getTempOptVec();
    for (auto i : *temp)
    {
        // do something interesting
    }
}

void views()
{
    // Fixed by changing to a constant string literal.
    std::string_view value("This is a string literal");
}

struct Y { int& get() [[msvc::lifetimebound]]; };
void f() {
    Y y{};
    int& r = y.get();
}
```

See also

[C26816](#)

[ES.65: Don't dereference an invalid pointer ↗](#)

Warning C26816

Article • 06/05/2023

The pointer points to memory allocated on the stack (ES.65)

Remarks

The pointer points to a variable that is allocated on the stack. When the variable goes out of scope it's cleaned up, which causes the pointer to be invalid.

This check recognizes views and owners from the C++ Standard Template Library (STL). To teach this check about user-authored types, use the `[[msvc::lifetimebound]]` annotation. The `[[msvc::lifetimebound]]` support is new in MSVC 17.7.

Code analysis name: `LIFETIME_LOCAL_USE_AFTER_FREE_STACK`

Examples

C++

```
// In this example, std::string is being used internally because the
// implementer felt it was easier to
// perform the non-trivial initialization of the value but the function
// returns a C-style string.
const char *danglingRawPtrFromLocal() {
    std::string s;

    // interesting string initialization here

    return s.c_str(); // Oops, The pointer points to memory that will be
                      // cleaned up upon return. Warning C26816.
}

struct Y { int& get() [[msvc::lifetimebound]]; };
int& f() {
    Y y;
    return y.get(); // Warning C26826
}
```

The fix is to extend the life of the value that is used. In this example, we address the warning by returning the `std::string`. It could also be addressed by copying the data to the heap or adding an "out" variable to the function parameter list.

C++

```
std::string danglingRawPtrFromLocal() {
    std::string s;

    // interesting string initialization here

    return s;
}

struct Y { int& get() [[msvc::lifetimebound]]; };
int f() {
    Y y;
    return y.get();
}
```

See also

[C26815](#)

[ES.65: Don't dereference an invalid pointer ↗](#)

Warning C26817

Article • 10/24/2023

Potentially expensive copy of variable *name* in range-for loop. Consider making it a const reference (es.71).

For more information, see [ES.71 notes](#) in the C++ Core Guidelines.

Example

If a range-for loop variable isn't explicitly marked as a reference, it gets a copy of each element iterated over:

C++

```
#include <vector>

class MyComplexType {
    int native_array[1000];
    // ...
};

void expensive_function(std::vector<MyComplexType>& complex_vector_ref)
{
    for (auto item: complex_vector_ref) // Warning: C26817
    {
        // At each iteration, item gets a copy of the next element
        // ...
    }
    for (MyComplexType item: complex_vector_ref)
    {
        // It happens whether you use the auto keyword or the type name
        // ...
    }
}
```

The warning is ignoring some types that are cheap to copy like for scalars (pointers, arithmetic types, and so on).

To fix this issue, if the loop variable isn't mutated anywhere in the loop, make it a const reference:

C++

```
#include <vector>

class MyComplexType {
    int native_array[1000];
    // ...
};

void less_expensive_function(std::vector<MyComplexType>& complex_vector_ref)
{
    for (const auto& item: complex_vector_ref)
    {
        // item no longer gets a copy of each iterated element
        // ...
    }
    for (const MyComplexType& item: complex_vector_ref)
    {
        // item no longer gets a copy of each iterated element
        // ...
    }
}
```

The `const` keyword makes the loop variable immutable. Use of a non-const reference makes it possible to inadvertently use the reference to modify the container's elements. If you need to modify only the local loop variable, the potentially expensive copying is unavoidable.

Warning C26818

Article • 10/07/2022

Switch statement does not cover all cases. Consider adding a 'default' label (es.79).

Remarks

This check covers the missing `default` label in switch statements that switch over a non-enumeration type, such as `int`, `char`, and so on.

For more information, see [ES.79: Use default to handle common cases \(only\)](#) in the C++ Core Guidelines.

Example

This example has a missing `default` label when switching over an `int`.

C++

```
void fn1();
void fn2();

void foo(int a)
{
    switch (a)
    {
        case 0:
            fn1();
            break;
        case 2:
            fn2();
            break;
    }
}
```

Solution

To fix this issue, insert a `default` label to cover all remaining cases.

C++

```
void fn1();
void fn2();
```

```
void default_action();

void foo(int a)
{
    switch (a)
    {
        case 0:
            fn1();
            break;
        case 2:
            fn2();
            break;
        default:
            default_action();
            break;
    }
}
```

If no default action needs to be taken, insert an empty `default` label to indicate that the other cases haven't been forgotten.

C++

```
void fn1();
void fn2();

void foo(int a)
{
    switch (a)
    {
        case 0:
            fn1();
            break;
        case 2:
            fn2();
            break;
        default:
            // do nothing for the rest of the cases
            break;
    }
}
```

Warning C26819

Article • 06/05/2023

Unannotated fallthrough between switch labels (es.78).

Remarks

This check covers implicit fallthrough in switch statements. Implicit fallthrough is when control flow transfers from one switch case directly into a following switch case without the use of the `[[fallthrough]];` statement. This warning is raised when an implicit fallthrough is detected in a switch case containing at least one statement.

For more information, see [ES.78: Don't rely on implicit fallthrough in switch statements](#) in the C++ Core Guidelines.

Example

In this sample, implicit fallthrough occurs from a nonempty `switch case` into a following `case`.

C++

```
void fn1();
void fn2();

void foo(int a)
{
    switch (a)
    {
        case 0:      // implicit fallthrough from case 0 to case 1 is OK
because case 0 is empty
        case 1:
            fn1(); // implicit fallthrough from case 1 into case 2
        case 2:      // Warning C26819.
            fn2();
            break;
        default:
            break;
    }
}
```

To fix this issue, insert a `[[fallthrough]];` statement where the fallthrough occurs.

C++

```
void fn1();
void fn2();

void foo(int a)
{
    switch (a)
    {
        case 0:
        case 1:
            fn1();
            [[fallthrough]];
        // fallthrough is explicit
        case 2:
            fn2();
            break;
        default:
            break;
    }
}
```

Another way to fix the issue is to remove the implicit fallthrough.

C++

```
void fn1();
void fn2();

void foo(int a)
{
    switch (a)
    {
        case 0:
        case 1:
            fn1();
            break; // case 1 no longer falls through into case 2
        case 2:
            fn2();
            break;
        default:
            break;
    }
}
```

See also

[ES.78: Don't rely on implicit fallthrough in switch statements ↗](#)

Warning C26820

Article • 10/24/2023

This is a potentially expensive copy operation. Consider using a reference unless a copy is required (p.9)

For more information, see [P.9: Don't waste time or space ↗](#) in the C++ Core Guidelines.

This check covers nonobvious and easy-to-miss behavior when assigning a reference to a variable marked `auto`. The type of the `auto` variable is resolved to a value rather than a reference, and an implicit copy is made.

Remarks

- This warning isn't raised for scalars, smart pointers, or views. It's also not raised for types whose size isn't more than twice the platform-dependent pointer size.
- This warning isn't raised when the variable gets mutated, as marking it `auto&` would introduce side-effects to the mutation.
- This warning isn't raised when the reference comes from a temporary object, because that results in a dangling reference. For example:

```
C++

std::optional<int> TryGetNumber();
...
const auto& val = TryGetNumber().value();
val++; // Temporary from TryGetNumber() is destroyed and val is now
dangling
```

Example

This sample shows a variable definition that makes a potentially expensive copy when assigned a reference:

```
C++

const Object& MyClass::getRef() { ... }
...
```

```
auto ref = myclass.getRef(); // C26820 (`ref` takes a copy of the returned  
object)
```

To resolve this issue, declare the variable by using `const auto&` instead:

C++

```
const Object& MyClass::getRef() { ... }  
...  
const auto& ref = myclass.getRef(); // OK
```

Code analysis for C/C++ warnings

Article • 08/03/2021

This section lists C/C++ Code Analysis warnings except those that are raised by the [C++ Core Guidelines Checkers](#). For information about Code Analysis, see [/analyze \(Code Analysis\)](#) and [Quick Start: Code Analysis for C/C++](#).

See also

- [Analyzing C/C++ Code Quality by Using Code Analysis](#)
- [Using SAL Annotations to Reduce C/C++ Code Defects](#)

Fatal error C1250

Article • 10/07/2022

Unable to load plug-in '*plugin-name*'.

The Code Analysis tool reports this error when there's an internal error in the plugin, not in the code being analyzed.

Fatal error C1251

Article • 10/07/2022

Unable to load models.

The Code Analysis tool reports this error when there's an internal error in the model file, not in the code being analyzed.

Fatal error C1252

Article • 10/07/2022

Circular or missing dependency between plugins: '*plugin-name*' requires GUID '*globally-unique-identifier*'

The Code Analysis tool reports this error when there's an internal error in the plugin dependencies. It's not caused by an issue in the code being analyzed.

In some cases, it's possible to work around this issue by disabling the **Enable Code Analysis on Build** property. To disable this build property, open the Property pages dialog for your project. In the **Solution Explorer** window, right-click on the project (not the solution) and select **Properties** in the shortcut menu. Set the **Configuration** to **All Configurations** and the **Platform** to **All Platforms**. Open the **Configuration Properties > Code Analysis > General** property page. Modify the **Enable Code Analysis on Build** property to **No**. Choose **OK** to save your changes, and then save your project files. Rebuild your project to verify that the issue no longer occurs.

If the issue continues, you can report it to Microsoft. For more information on how to report an issue, see [How to report a problem with the Visual C++ toolset or documentation](#).

Fatal error C1253

Article • 10/07/2022

Unable to load model file.

The Code Analysis tool reports this error when there's an internal error in the model file, not in the code being analyzed.

Fatal error C1254

Article • 10/07/2022

Plugin version mismatch: '*module*' version '*version-number*' doesn't match the version '*version-number*' of the PREfast driver

The Code Analysis tool reports this error when there's an internal error with the plugin version, not in the code being analyzed.

Fatal error C1255

Article • 10/07/2022

PCH data for plugin '*plugin-name*' has incorrect length.

The Code Analysis tool reports this error when there's an internal error in the tool, not in the code being analyzed.

Fatal error C1256

Article • 10/07/2022

'*plugin-name*': PCH must be both written and read.

The Code Analysis tool reports this error when there's an internal error in the tool, not in the code being analyzed.

Fatal error C1257

Article • 10/07/2022

'Plugin-name': Initialization Failure.

The Code Analysis tool reports this error when there's an internal error in the plugin, not in the code being analyzed.

Fatal error C1258

Article • 10/07/2022

Failed to save XML Log file '*filename*'. *Message*.

The Code Analysis tool reports this error when there's an internal error in the plugin, not in the code being analyzed.

Fatal error C1259

Article • 10/07/2022

A fatal error was issued by a plugin.

The Code Analysis tool reports this error when there's an internal error in the plugin, not in the code being analyzed.

Fatal error C1260

Article • 10/07/2022

The plugins '*plugin-1*' and '*plugin-2*' share the same id. It is not supported to load the same plugin twice.

The Code Analysis tool reports this error when there's an internal error in the plugin, not in the code being analyzed.

Warning C6001

Article • 05/08/2023

Using uninitialized memory '*variable*'.

Remarks

This warning is reported when an uninitialized local variable is used before it's assigned a value. This usage could lead to unpredictable results. You should always initialize variables before use.

Code analysis name: USING_UNINIT_VAR

Example

The following code generates this warning because variable `i` is only initialized if `b` is true:

```
C++

int f( bool b )
{
    int i;
    if ( b )
    {
        i = 0;
    }
    return i; // i is uninitialized if b is false
}
```

To correct this warning, initialize the variable as shown in the following code:

```
C++

int f( bool b )
{
    int i = -1;

    if ( b )
    {
        i = 0;
    }
    return i;
}
```

Heuristics

The following example shows that passing a variable to a function by reference causes the compiler to assume that it's initialized:

C++

```
void init( int& i );

int f( bool b )
{
    int i;

    init(i);

    if ( b )
    {
        i = 0;
    }
    return i; // i is assumed to be initialized because it's passed by
reference to init()
}
```

This supports the pattern of passing a pointer to a variable into an initialization function.

This heuristic can lead to false negatives because many functions expect pointers that point to initialized data. Use [SAL annotations](#), such as `_In_` and `_Out_`, to describe the function's behavior. The following example calls a function that expects its argument to be initialized, so a warning is generated:

C++

```
void use( _In_ int& i );

int f( bool b )
{
    int i;

    use(i); // uninitialized variable warning because of the _In_ annotation
on use()

    if ( b )
    {
        i = 0;
    }
    return i;
}
```

See also

[Compiler Warning \(level 1 and level 4\) C4700](#)

Warning C6011

Article • 06/05/2023

Dereferencing NULL pointer '*pointer-name*'.

Remarks

This warning indicates that your code dereferences a potentially null pointer. If the pointer value is invalid, the result is undefined. To resolve the issue, validate the pointer before use.

Code analysis name: DEREF_NULL_PTR

Example

The following code generates this warning because a call to `malloc` might return null if insufficient memory is available:

```
C++

#include <malloc.h>

void f( )
{
    char *p = ( char * ) malloc( 10 );
    *p = '\0';

    // code ...
    free( p );
}
```

To correct this warning, examine the pointer for a null value as shown in the following code:

```
C++

#include <malloc.h>
void f( )
{
    char *p = ( char * ) malloc( 10 );
    if ( p )
    {
        *p = '\0';
        // code ...
    }
}
```

```
    free( p );
}
}
```

Functions may have parameters annotated by using the `Null` property in a `Pre` condition. Allocate memory inside these functions before you dereference the parameter. The following code generates warning C6011 because an attempt is made to dereference a null pointer (`pc`) inside the function without first allocating memory:

C++

```
#include <sal.h>
using namespace vc_attributes;
void f([Pre(Null=Yes)] char* pc)
{
    *pc='\0'; // warning C6011 - pc is null
    // code ...
}
```

The careless use of `malloc` and `free` leads to memory leaks and exceptions. To minimize these kinds of leaks and exception problems altogether, avoid allocating raw memory yourself. Instead, use the mechanisms provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

Heuristics

A heuristic used to reduce the number of warnings in legacy code assumes that a pointer is non-`NULL` unless there is evidence that it is `NULL`. In the examples we've seen so far, pointers returned by `malloc` or `new` might be `NULL` because allocation might fail. Another characteristic that the analysis engine uses as evidence of nullability is if the program explicitly checks for `NULL`. This is illustrated in the following examples:

C++

```
void f(int* n)
{
    *n = 1; // Does not warn, n is assumed to be non-null
}

void f(int* n)
{
    if (n) {
        (*n)++;
    }
}
```

```
    }
    *n = 1; // Warns because the earlier conditional shows that n might be
    null
}
```

In the second case, the user can fix the warning by moving the `*n = 1` line inside the if block.

See also

- [Using SAL Annotations to reduce code defects](#)
- [NULL](#)
- [Indirection and Address-of Operators](#)
- [malloc](#)
- [free](#)
- [new operator](#)
- [delete operator](#)

Warning C6014

Article • 10/07/2022

Leaking memory '*pointer-name*'.

This warning indicates that the specified pointer points to allocated memory or some other allocated resource that hasn't been freed.

Remarks

The analyzer checks for this condition only when the

`_Analysis_mode_(_Analysis_local_leak_checks_)` SAL annotation is specified. By default, this annotation is specified for Windows kernel mode (driver) code. For more information about SAL annotations, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning is reported for both memory and resource leaks when the resource is commonly *aliased* to another location. Memory is aliased when a pointer to the memory escapes the function by using an `_out_` parameter annotation, global variable, or return value. This warning can be reported on function exit if the argument is annotated that its release is expected.

Code Analysis won't recognize the actual implementation of a memory allocator (involving address arithmetic) and won't recognize that memory is allocated (although many wrappers will be recognized). In this case, the analyzer doesn't recognize that the memory was allocated and issues this warning. To suppress the false positive, use a `#pragma warning(disable: 6014)` directive on the line that precedes the opening brace `{` of the function body.

Code analysis name: `MEMORY_LEAK`

Examples

The following code generates warning C6014:

C++

```
// cl.exe /analyze /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>
#include <string.h>
```

```

_Analysis_mode_(_Analysis_local_leak_checks_)

#define ARRAYSIZE 10
const int TEST_DATA [ARRAYSIZE] = {10,20,30,40,50,60,70,80,90,100};

void f( )
{
    int *p = (int *)malloc(sizeof(int)*ARRAYSIZE);
    if (p) {
        memcpy(p, TEST_DATA, sizeof(int)*ARRAYSIZE);
        // code ...
    }
}

int main( )
{
    f();
}

```

The following code corrects the warning by releasing the memory:

C++

```

// cl.exe /analyze /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>
#include <string.h>

_Analysis_mode_(_Analysis_local_leak_checks_)

#define ARRAYSIZE 10
const int TEST_DATA [ARRAYSIZE] = {10,20,30,40,50,60,70,80,90,100};

void f( )
{
    int *p = (int *)malloc(sizeof(int)*ARRAYSIZE);
    if (p) {
        memcpy(p, TEST_DATA, sizeof(int)*ARRAYSIZE);
        // code ...
        free(p);
    }
}

int main( )
{
    f();
}

```

To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include [shared_ptr](#), [unique_ptr](#), and containers

such as [vector](#). For more information, see [Smart pointers and C++ Standard Library](#).

C++

```
// cl.exe /analyze /EHsc /nologo /W4
#include <sal.h>
#include <memory>

using namespace std;

_Analysis_mode_(_Analysis_local_leak_checks_)

const int ARRSIZE = 10;
const int TEST_DATA [ARRSIZE] = {10,20,30,40,50,60,70,80,90,100};

void f( )
{
    unique_ptr<int[]> p(new int[ARRSIZE]);
    std::copy(begin(TEST_DATA), end(TEST_DATA), p.get());

    // code ...

    // No need for free/delete; unique_ptr
    // cleans up when out of scope.
}

int main( )
{
    f();
}
```

See also

[Warning C6211](#)

Warning C6029

Article • 02/22/2023

Possible buffer overrun in call to '*function*'

Possible buffer overrun in called function due to an unchecked buffer length/size parameter.

Remarks

This warning indicates that code passes an unchecked size to a function that takes a buffer and a size. The code doesn't verify that the data read from some external source is smaller than the buffer size. An attacker might intentionally specify a larger than expected value for the size, which can lead to a buffer overrun. Generally, whenever you read data from an untrusted external source, make sure to verify it for validity. It's appropriate to verify the size to make sure it's in the expected range.

Code analysis name: `USING_TAINTED_DATA`

Example

The following code generates this warning when it calls the annotated function `std::fread` two times. The code uses the first call to determine the length of the data to read in later calls. After the first call, analysis marks `dataSize` as coming from an untrusted source. Therefore, when the code passes the untrusted value to the second `std::fread` call, the analyzer generates this warning. A malicious actor could modify the file and cause the call to `std::fread` to overflow the `buffer` array. In real world code, you should also handle error recovery based on the return value of `std::fread`. For simplicity, these examples intentionally leave out error recovery code.

C++

```
void processData(FILE* file)
{
    const size_t MAX_BUFFER_SIZE = 100;
    uint32_t buffer[MAX_BUFFER_SIZE]{};
    uint8_t dataSize = 0;

    // Read length data from the beginning of the file
    fread(&dataSize, sizeof(uint8_t), 1, file);
    // Read the rest of the data based on the dataSize
```

```
    fread(buffer, sizeof(uint32_t), dataSize, file);
}
```

The fix for the issue depends on the nature of the data and the behavior of the annotated function that triggers the diagnostic. For more information, see the documentation for that function. A straightforward fix is to check the size before the second call to `std::fread`. In the next example, we throw an exception to terminate the function. Most real-world code would instead have an error recovery strategy that's specific to the scenario.

C++

```
void processData(FILE* file)
{
    const size_t MAX_BUFFER_SIZE = 100;
    uint32_t buffer[MAX_BUFFER_SIZE]{};
    uint8_t dataSize = 0;

    fread(&dataSize, sizeof(uint32_t), 1, file);

    if (dataSize > MAX_BUFFER_SIZE)
    {
        throw std::runtime_error("file data unexpected size");
    }

    fread(buffer, sizeof(uint32_t), dataSize, file);
}
```

In `std::fread` and similar functions, the code may need to read large amounts of data. To handle large data, you can allocate the size of the buffer dynamically after the size becomes known. Or, you can call `std::fread` multiple times as needed to read in the rest of the data. If you allocate the buffer dynamically, we recommend you put a limit on the size to avoid introducing an out-of-memory exploit for large values. We don't use this approach in our example because it's already bounded by the size of `uint8_t`.

C++

```
void processDataDynamic(FILE* file)
{
    uint8_t dataSize = 0;
    fread(&dataSize, sizeof(uint8_t), 1, file);

    // Vector with `dataSize` default initialized objects
    std::vector<uint32_t> vecBuffer(dataSize);

    fread(&vecBuffer[0], sizeof(uint32_t), dataSize, file);
}

void processDataMultiple(FILE* file)
```

```
{  
    const size_t MAX_BUFFER_SIZE = 100;  
    uint32_t buffer[MAX_BUFFER_SIZE]{};  
    uint8_t dataSize = 0;  
  
    fread(&dataSize, sizeof(uint32_t), 1, file);  
  
    while( dataSize > 0 )  
    {  
        size_t readSize = dataSize > MAX_BUFFER_SIZE ? MAX_BUFFER_SIZE :  
dataSize;  
        readSize = fread(buffer, sizeof(uint32_t), readSize, file);  
        dataSize = dataSize - readSize;  
        // Process the data in `buffer`...  
    }  
}
```

See also

[Rule sets for C++ code](#)

[Using SAL Annotations to reduce code defects](#)

Warning C6030

Article • 03/22/2023

Use attribute `[[noreturn]]` over `_declspec(noreturn)` in function '*function-name*'

Remarks

This warning suggests using the C++11 standard attribute `[[noreturn]]` in place of the `declspec` variant `_declspec(noreturn)`. The standard attribute provides better cross-platform support because it doesn't rely on language extensions.

This warning is off by default and isn't part of the `All Rules` rule set. To enable this warning, it must be added to the rule set file being used.

This check is available in Visual Studio 2022 version 17.0 and later versions. Code analysis name: `USE_ATTRIBUTE_NORETURN`

Example

The following code generates C6030:

C++

```
_declspec(noreturn) void TerminateApplication();
```

Fix the issue by using the `[[noreturn]]` attribute:

C++

```
[[ noreturn ]] void TerminateApplication();
```

See also

[Use Rule Sets to Specify the C++ Rules to Run](#)

Warning C6031

Article • 04/05/2024

Return value ignored: '*called-function*' could return unexpected value

Remarks

Warning C6031 indicates the caller doesn't check a function's return value for failure. Depending on which function is being called, this defect can lead to seemingly random program misbehavior. That includes crashes and data corruptions in error conditions or low-resource situations.

In general, it isn't safe to assume that calls to functions requiring disk, network, memory, or other resources will succeed. The caller should always check the return value and handle error cases appropriately. Also consider using the `_Must_inspect_result_` annotation, which checks that the value is examined in a useful way.

This warning applies to both C and C++ code.

Code analysis name: `RETVAL_IGNORED_FUNC_COULD_FAIL`

Example

The following code generates warning C6031:

```
C

#include <stdio.h>
int main()
{
    fopen("test.c", "r"); // C4996, C6031 return value ignored
    // code ...
}
```

To correct this warning, check the return value of the function as shown in the following code:

```
C

#include <stdio.h>
int main()
{
    FILE* stream;
```

```
if ((stream = fopen("test.c", "r")) == NULL)
{
    return;
}
// code ...
}
```

The following code uses safe function `fopen_s` to correct this warning:

C

```
#include <stdio.h>
int main()
{
    FILE* stream;
    errno_t err;

    if ((err = fopen_s(&stream, "test.c", "r")) != 0)
    {
        // code ...
    }
}
```

This warning is also generated if the caller ignores the return value of a function annotated with the `_Check_return_` property as shown in the following code.

C++

```
#include <sal.h>
_Check_return_ bool func()
{
    return true;
}

int main()
{
    func();
}
```

To correct the previous warning, check the return value as shown in the following code:

C++

```
#include <sal.h>
_Check_return_ bool func()
{
    return true;
}

int main()
```

```
{  
    if (func())  
    {  
        // code ...  
    }  
}
```

In cases where it's necessary to ignore the return value of a function, assign the returned value to `std::ignore`. Assigning to `std::ignore` clearly indicates developer intent and helps in future code maintenance.

C++

```
#include <tuple>  
#include <ctime>  
#include <cstdlib>  
#include <stdio.h>  
  
int main()  
{  
    std::srand(static_cast<unsigned int>(std::time(nullptr))); // set  
    initial seed value to system clock  
    std::ignore = std::rand(); // Discard the first result as the few random  
    results are always small.  
    // ...  
}
```

See also

[fopen_s, _wfopen_s](#)

[Using SAL Annotations to reduce code defects](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Warning C6053

Article • 10/20/2023

Call to '*function*' may not zero-terminate string '*variable*'.

Remarks

This warning indicates that the specified function has been called in such a way that the resulting string might not be zero-terminated. This defect might cause an exploitable buffer overrun or crash. This warning is also generated if an annotated function expects a null-terminated string, but you pass a non-null-terminated string.

Most C standard library and Win32 string handling functions require and produce zero-terminated strings. A few 'counted string' functions (including `strncpy`, `wcsncpy`, `_mbsncpy`, `_snprintf`, and `snwprintf`) don't produce zero-terminated strings if they exactly fill their buffer. In this case, a subsequent call to a string function that expects a zero-terminated string will go beyond the end of the buffer looking for the zero. The program should make sure that the string ends with a zero. In general, you should pass a length to the 'counted string' function one smaller than the size of the buffer and then explicitly assign zero to the last character in the buffer.

Code analysis name: `MISSING_ZERO_TERMINATION1`

Examples

The following sample code generates this warning:

C++

```
#include <string.h>
#define MAX 15

size_t f( )
{
    char szDest[MAX];
    char *szSource="Hello, World!";

    strncpy(szDest, szSource, MAX);
    return strlen(szDest); // possible crash here
}
```

To correct this warning, zero-terminate the string as shown in the following sample code:

C++

```
#include <string.h>
#define MAX 15

size_t f( )
{
    char szDest[MAX];
    char *szSource="Hello, World!";

    strncpy(szDest, szSource, MAX-1);
    szDest[MAX-1]=0;
    return strlen(szDest);
}
```

The following sample code corrects this warning using safe string manipulation

`strncpy_s` function:

C++

```
#include <string.h>
#define MAX 15

size_t f( )
{
    char szDest[MAX];
    char *szSource= "Hello, World!";

    strncpy_s(szDest, sizeof(szDest), szSource, strlen(szSource));
    return strlen(szDest);
}
```

Heuristics

This warning is sometimes reported on certain idioms guaranteed to be safe in practice. Because of the frequency and potential consequences of this defect, the analysis tool is biased in favor of finding potential issues instead of its typical bias of reducing noise.

See also

- [Using SAL Annotations to reduce code defects](#)
- [`strncpy_s`, `_strncpy_s_l`, `wcsncpy_s`, `_wcsncpy_s_l`, `_mbsncpy_s`, `_mbsncpy_s_l`](#)

Warning C6054

Article • 10/07/2022

String '*variable*' may not be zero-terminated.

Remarks

This warning indicates that a function that requires a zero-terminated string was passed a non-zero terminated string. A function that expects a zero-terminated string could look for the zero beyond the end of the buffer. This defect might cause an exploitable buffer overrun error or crash. The program should make sure the string passed in ends with a zero.

Code analysis name: `MISSING_ZERO_TERMINATION2`

Example

The following code generates this warning:

```
C++  
  
// Warning C6054_bad.cpp  
// Compile using: cl /W4 /EHsc /c /analyze C6054_bad.cpp  
#include <sal.h>  
  
void func( _In_z_ wchar_t* wszStr );  
  
void g()  
{  
    wchar_t wcArray[200] = { 'h', 'e', 'l', 'l', 'o' };  
    func(wcArray); // Warning C6054  
}
```

To correct this warning, null-terminate `wcArray` before calling function `func()` as shown in the following sample code:

```
C++  
  
// C6054_good.cpp  
// Compile using: cl /W4 /EHsc /c /analyze C6054_good.cpp  
#include <sal.h>  
  
void func( _In_z_ wchar_t* wszStr );
```

```
void g( )
{
    wchar_t wcArray[200] = { 'h', 'e', 'l', 'l', 'o', '\0' };
    func(wcArray); // OK
}
```

See also

- [Warning C6053](#)
- [Using SAL Annotations to reduce code defects](#)

Warning C6059

Article • 01/04/2024

Incorrect length parameter in call to '*function*'. Pass the number of remaining characters, not the buffer size of '*variable*'.

Remarks

This warning indicates that a call to a string concatenation function is probably passing an incorrect value for the number of characters to concatenate. This defect might cause an exploitable buffer overrun or crash. A common cause of this defect is passing the buffer size (instead of the remaining number of characters in the buffer) to the string manipulation function.

This warning helps identify the common error of sending the size of the target buffer instead of the size of the data. It does so by detecting when the size used to allocate the buffer is passed, unchanged, to the function putting data in the buffer.

Code analysis name: BAD_CONCATENATION

Example

The following code generates warning C6059:

```
C++  
  
#include <string.h>  
#define MAX 25  
  
void f( )  
{  
    char szTarget[MAX];  
    const char *szState = "Washington";  
    const char *szCity = "Redmond, ";  
  
    strncpy(szTarget, szCity, MAX);  
    szTarget[MAX - 1] = '\0';  
    strncat(szTarget, szState, MAX); // wrong size  
    // code ...  
}
```

To correct this warning, use the correct number of characters to concatenate as shown in the following code:

C++

```
#include <string.h>
#define MAX 25

void f( )
{
    char szTarget[MAX];
    const char *szState ="Washington";
    const char *szCity="Redmond, ";

    strncpy(szTarget, szCity, MAX);
    szTarget[MAX - 1] = '\0';
    strncat(szTarget, szState, MAX - strlen(szTarget)); // correct size
    // code ...
}
```

To correct this warning using the safe string manipulation functions `strncpy_s` and `strncat_s`, see the following code:

C++

```
#include <string.h>

void f( )
{
    const char *szState ="Washington";
    const char *szCity="Redmond, ";

    size_t nTargetSize = strlen(szState) + strlen(szCity) + 1;
    char *szTarget= new char[nTargetSize];

    strncpy_s(szTarget, nTargetSize, szCity, strlen(szCity));
    strncat_s(szTarget, nTargetSize, szState,
              nTargetSize - strlen(szTarget));
    // code ...
    delete [] szTarget;
}
```

Heuristics

This analysis detects when the target buffer size is passed unmodified into the length parameter of the string manipulation function. This warning isn't given if some other value is passed as the length parameter, even if that value is incorrect.

Consider the following code that generates warning C6059:

C++

```

#include <string.h>
#define MAX 25

void f( )
{
    char szTarget[MAX];
    const char *szState ="Washington";
    const char *szCity="Redmond, ";

    strncpy(szTarget, szCity, MAX);
    szTarget[MAX -1] = '\0';
    strncat(szTarget, szState, MAX); // wrong size
    // code ...
}

```

The warning goes away by changing the `MAX` argument to `strncat` to `MAX - 1`, even though the length calculation is still incorrect.

C++

```

#include <string.h>
#define MAX 25

void f( )
{
    char szTarget[MAX];
    const char *szState ="Washington";
    const char *szCity="Redmond, ";

    strncpy(szTarget, szCity, MAX);
    szTarget[MAX -1] = '\0';
    strncat(szTarget, szState, MAX - 1); // wrong size, but no warning
    // code ...
}

```

See also

- `strncpy_s`, `_strncpy_s_l`, `wcsncpy_s`, `_wcsncpy_s_l`, `_mbsncpy_s`, `_mbsncpy_s_l`
- `strncat_s`, `_strncat_s_l`, `wcsncat_s`, `_wcsncat_s_l`, `_mbsncat_s`, `_mbsncat_s_l`

Warning C6063

Article • 03/23/2023

Missing string argument to '*function*' that corresponds to conversion specifier '*number*'.

Remarks

This warning indicates that not enough arguments are being provided to match a format string. At least one of the missing arguments is a string. This defect can cause crashes and buffer overflows (if the called function is of the `sprintf` family), and also potentially incorrect output.

Code analysis name: `MISSING_STRING_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning:

```
C++  
  
#include <stdio.h>  
void f( )  
{  
    char buff[15];  
    sprintf(buff, "%s %s", "Hello, World!");  
}
```

To correct this warning, remove the unused format specifier or provide the required arguments as shown in the following code:

```
C++  
  
#include <stdio.h>  
void f( )  
{  
    char buff[15];  
    sprintf(buff, "%s %s ", "Hello", "World");  
}
```

The following code corrects this warning using safe string manipulation function:

```
C++
```

```
#include <stdio.h>
void f( )
{
    char buff[15];
    sprintf_s( buff, sizeof(buff), "%s", "Hello, World!" );
}
```

See also

[Format specification syntax: printf and wprintf functions](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

Warning C6064

Article • 02/22/2023

Missing integer argument to '*function-name*' corresponding to conversion specifier '*number*'

This warning indicates that the code doesn't provide enough arguments to match a format string and one of the missing arguments is an integer.

Remarks

This defect is likely to cause incorrect output and, in more dangerous cases, can lead to stack overflow.

Code analysis name: MISSING_INTEGER_ARGUMENT_TO_FORMAT_FUNCTION

Example

The following code generates this warning because it uses an incorrect number of arguments in the call to `sprintf_s` and the missing argument is an integer. If the unsafe function `sprintf` was used instead of the safer variant `sprintf_s`, this code would likely cause a stack overflow instead of just an unexpected output:

C++

```
void f()
{
    char buff[8];
    const char *string="Hello";
    sprintf_s(buff, sizeof(buff), "%s %d", string); // Attempts to print
"Hello "
    // followed by a number up to eleven characters long, depending on the
garbage
    // found on the stack. Any number other than a single non-negative digit
can't
    // fit in the 8 char buffer and leave room for the trailing null. If
sprintf
    // had been used instead, it would overflow.
}
```

To correct this warning, specify missing arguments or adjust the format string. In this example, we add the missing integer value.

C++

```
void f()
{
    char buff[8];
    const char *string = "Hello";
    sprintf_s(buff, sizeof(buff), "%s %d", string, strlen(string));
}
```

See also

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[C4473](#)

Warning C6065

Article • 03/23/2023

warning C6065: Missing pointer to '*string type*' argument to '*function*' that corresponds to argument '*number*'

Remarks

This warning indicates that there's a mismatch between the format specifiers in a string and the types of the associated parameters. The format specifier indicates that at least one of the mismatched arguments should be a pointer to a counted string such as `UNICODE_STRING` or `ANSI_STRING` but it is not. This defect can cause crashes, buffer overflows, and potentially incorrect output.

To fix this warning, determine if the format specifier or the argument matches the intended behavior and modify the other to match. When modifying the format specifier for a counted string, it's recommended to explicitly use the size prefix such as `%wZ` or `%hz` rather than `%z` due to compatibility issues between C runtimes (CRT). For more information on CRT compatibility, see the `%z` row in the [Type field characters documentation](#).

Code analysis name: `MISSING_COUNTED_STRING_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning because the value passed to `printf` isn't a pointer:

```
C++  
  
int PrintDiagnostic(UNICODE_STRING u)  
{  
    printf("%wZ", u);  
}
```

In this example, we fix the warning by changing the passed in parameter to be a pointer:

```
C++  
  
int PrintDiagnostic(UNICODE_STRING *u)  
{
```

```
    printf("%wZ", &u);  
}
```

See also

[format specification syntax: printf and wprintf functions](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[UNICODE_STRING](#)

[ANSI_STRING/_STRING](#)

[C4313](#)

Warning C6066

Article • 03/23/2023

Non-pointer passed as parameter(*number*) when pointer is required in call to '*function*'.

Remarks

This warning indicates that the format string specifies that a pointer is required, but a non-pointer is being passed. A pointer is required, for example, when you use a `%n` or `%p` specification for `printf`, or a `%d` for `scanf`. This defect is likely to cause a crash or corruption of some form.

Code analysis name: `NON_POINTER_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning:

```
C++  
  
#include <stdio.h>  
#define MAX 30  
void f( )  
{  
    char buff[MAX];  
    sprintf( buff, "%s %p %d", "Hello, World!", 1, MAX ); //warning C6066  
    // code ...  
}  
  
void g( int i )  
{  
    int result = scanf( "%d", i ); // warning C6066  
    // code ...  
}
```

To correct this warning, the following code passes correct parameters to the `sprintf` and `scanf` functions:

```
C++  
  
#include <stdio.h>  
#define MAX 30
```

```
void f( )
{
    char buff[MAX];
    sprintf( buff, "%s %p %d", "Hello, World!", buff, MAX ); // pass buff
    // code ...
}
void g( int i )
{
    int result = scanf( "%d", &i ); // pass the address of i
    // code ...
}
```

The following code uses safe string manipulation functions `sprintf_s` and `scanf_s` to correct this warning:

C++

```
void f( )
{
    char buff[MAX];
    sprintf_s( buff, sizeof(buff), "%s %p %d", "Hello, World!", buff, MAX );
    // code ...
}
void g( int i )
{
    int result = scanf_s( "%d", &i );
    // code ...
}
```

This warning is typically reported because an integer has been used for a `%p` format instead of a pointer. Using an integer in this instance isn't portable to 64-bit computers.

See also

[Format specification syntax: printf and wprintf functions](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l](#)

[C4313](#)

[C4477](#)

Warning C6067

Article • 03/23/2023

Parameter 'number' in call to 'function' must be the address of the string

Remarks

This warning indicates a mismatch between the format specifier and the function parameter. Even though the warning suggests using the address of the string, you must check the type of parameter a function expects before correcting the problem. For example, a `%s` specification for `printf` requires a string argument, but a `%s` specification in `scanf` requires an address of the string.

This defect is likely to cause a crash or corruption of some form.

Code analysis name: `NON_STRING_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning because an integer is passed instead of a string:

```
C++  
  
#include <stdio.h>  
  
void f_defective()  
{  
    char *str = "Hello, World!";  
    printf("String:\n %s", 1);  
    // code ...  
}
```

To correct the warning, pass a string as a parameter to `printf` as shown in the following code:

```
C++  
  
#include <stdio.h>  
  
void f_corrected()  
{  
    char *str = "Hello, World!";  
}
```

```
    printf("String:\n %s", str);
    // code ...
}
```

The following code generates this warning because an incorrect level of indirection is specified when passing the parameter, buffer, to `scanf`:

C++

```
#include <stdio.h>

void h_defective()
{
    int retval;
    char* buffer = new char(20);
    if (buffer)
    {
        retval = scanf("%s", &buffer); // warning C6067
        // code...
        delete buffer;
    }
}
```

To correct above warnings, pass the correct parameter as shown in the following code:

C++

```
#include <stdio.h>

void h_corrected()
{
    int retval;
    char* buffer = new char(20);
    if (buffer)
    {
        retval = scanf("%s", buffer);
        // code...
        delete buffer;
    }
}
```

The following code uses safe string manipulation functions to correct this warning:

C++

```
#include <stdio.h>

void f_safe()
{
    char buff[20];
```

```
int retVal;

sprintf_s(buff, 20, "%s %s", "Hello", "World!");
printf_s("String:\n    %s %s", "Hello", "World!");
retVal = scanf_s("%s", buff, 20);
}
```

See also

[Format specification syntax: printf and wprintf functions](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[printf, _printf_l, wprintf, _wprintf_l](#)

[scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l](#)

[C4313](#)

[C4477](#)

Warning C6101

Article • 02/22/2023

Returning uninitialized memory '*parameter-name*'.

A successful path through the function doesn't set the `_Out_` annotated parameter.

Remarks

The purpose of this warning is to avoid the use of uninitialized values by callers of the function. The analyzer assumes callers don't initialize any parameters annotated with `_Out_` before the function call, and checks that the function initializes them. The analyzer doesn't emit this warning if the function returns a value that indicates it had an error or wasn't successful. To fix this issue, make sure to initialize the `_Out_` parameter under all successful return paths. The error message contains the line numbers of an example path that doesn't initialize the parameter.

If the initialization behavior is by design, then incorrect or missing SAL annotations are a likely cause for the warning. You can typically resolve these cases in one of two ways: Either change `_Out_` to a more appropriate annotation, or use the `_Success_()` annotation to help define the success/error states of the function. It's important for the static analysis tools to have correct annotations on the function when analyzing the call sites of the function.

Fix by changes to parameter annotations

If the parameter should already be in an initialized state and the function conditionally modifies it, then the `_Inout_` annotation may be more appropriate. If no other high level annotation fits the intended behavior, you can use low level annotations such as `_Pre_null_`, `_Pre_satisfies_()`, and `_Post_satisfies_()` that provide extra flexibility and control over the expected state of the parameter. For more information on parameter annotations, see [Annotating function parameters and return values](#).

Fix by defining successful return paths

The analyzer only emits this warning when the code doesn't initialize an `_Out_` parameter in the success paths of the function. If there's no `_Success_` annotation and no function return type annotation, then it considers all return paths successful. For

more information on `_Success_` and similar annotations, see [Success/Failure annotations](#).

Code analysis name: RETURN_UNINIT_VAR

Example

The following code generates this warning. Because the function returns `void`, the analyzer considers all paths successful. In this case, the correct fix would probably be to adjust the logic of the `if` statement, but in real world code it's typically not as straightforward and the solution depends on the intended behavior of the function.

C++

```
#include <sal.h>
void AlwaysInit(_Out_ int* output, int input) // : warning C6101: Returning
uninitialized memory '*p'.: Lines: 2, 4, 9, 14, 2
{
    if( input > 0 )
    {
        *output = input;
        return;
    }
    else if( input < 0 )
    {
        *output = 0;
        return;
    }
    return; // Oops, input was 0
}
```

To make the solution more interesting, we assume that it isn't valid to initialize `output` when `input` is `0`. One approach is to modify the function return value to a different type, such as `bool`. Then, add a `_Success_` annotation to define the successful return paths.

C++

```
_Success_(return == true)
bool InitNotZero(_Out_ int* output, int input)
{
    if( input > 0 )
    {
        *output = input;
        return true;
    }
    else if( input < 0 )
```

```
{  
    *output = 0;  
    return true;  
}  
return false;  
}
```

If this pattern is common in your codebase, you can add the annotation to the return type. Error codes such as HRESULT from the Windows SDK give you the behavior of the `_Success_` annotation without needing to add it to each function. If you already use an annotated type as a return type and want to override the behavior, then add the annotation to the function, as shown in the previous example.

C++

```
using SuccessWhenTrue = _Success_(return == true) bool;  
  
SuccessWhenTrue InitNotZero(_Out_ int* output, int input)  
{  
    // ...  
}
```

See also

[Rule sets for C++ code](#)

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C6102

Article • 10/07/2022

Using '*variable*' from failed function call at line '*location*'.

This warning is reported instead of [C6001](#) when a variable isn't set because it was marked as an `_Out_` parameter on a prior function call that failed.

The problem might be that the prior called function isn't fully annotated. It may require `_Always_`, `_Outptr_result_nullonfailure_` (`_COM_Outptr_` for COM code), or a related annotation.

See also

- [C6001](#)
- [Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C6103

Article • 10/07/2022

Returning '*variable*' from failed function call at line '*location*'.

A successful path through the function is returning a variable that was used as an `_Out_` parameter to an internal function call that failed.

The problem might be that the called function and the calling function aren't fully annotated. The called function may require `_Always_`, `_Outptr_result_nullonfailure_` (`_COM_Outptr_` for COM code), or a related annotation, and the calling function may require a `_Success_` annotation. Another possible cause for this warning is that the `_Success_` annotation on the called function is incorrect.

See also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C6200

Article • 06/05/2023

Index '*index*' is out of valid index range '*min*' to '*max*' for nonstack buffer '*parameter-name*'

This warning indicates that an integer offset into the specified nonstack array exceeds the maximum bounds of that array, causing undefined behavior and potentially crashes.

Remarks

One common cause of this defect is using the size of an array as an index into the array. Because C/C++ array indexing is zero-based, the maximum legal index into an array is one less than the number of array elements.

Code analysis name: INDEX_EXCEEDS_MAX_NONSTACK

Example

The following code generates this warning. This issue stems from the `for` loop exceeding the index range, attempting to access index 14 (the 15th element) when index 13 (the 14th element) is the last:

```
C++  
  
void f()  
{  
    int* buff = new int[14]; // array of 0..13 elements  
    for (int i = 0; i <= 14; i++) // i exceeds the index  
    {  
        buff[i] = 0; // warning C6200  
    }  
    delete[] buff;  
}
```

To correct both warnings, use correct array size as shown in the following code:

```
C++  
  
void f()  
{  
    int* buff = new int[14]; // array of 0..13 elements  
    for (int i = 0; i < 14; i++) // i == 13 on the final iteration
```

```
{  
    buff[i] = 0; // initialize buffer  
}  
delete[] buff;  
}
```

Heuristics

Code analysis can't always prove whether an array index is in range. This can happen, for example, when the index is computed from a complex expression, including those expressions that call into other functions. In these cases, code analysis may fall back on other clues to determine the range an array index expression may fall into.

For example, consider the following function that uses `rand()` in index calculations as a stand-in for a function call that code analysis can't analyze:

C++

```
#include <stdlib.h>  
  
void f()  
{  
    int* buff = new int[14];  
    for (int i = 1; i < 14; i++)  
    {  
        buff[rand()] = 0;          // no warning, nothing is known about the  
        return value of rand()  
        buff[rand() % 15] = 0;    // warning C6200, rand() % 15 is known to be  
        in the range 0..14 and index 14 is out of bounds  
        buff[rand() % 14] = 0;    // no warning, rand() % 14 is known to be in  
        the range 0..13  
    }  
    delete[] buff;  
}
```

Code analysis doesn't warn with just `rand()` because it doesn't have any information about its return value. On the other hand, `rand() % 15` and `rand() % 14` provide hints as to the range of the return value of `rand()` and code analysis can use that information to determine that the index is out of bounds in the first case but not the second.

Warning C6201

Article • 11/20/2023

Index '*index-name*' is out of valid index range '*minimum*' to '*maximum*' for possibly stack allocated buffer '*variable*'

This warning indicates that an integer offset into the specified stack array exceeds the maximum bounds of that array. It might potentially cause stack overflow errors, undefined behavior, or crashes.

Remarks

One common cause of this defect is using an array's size as an index into the array. Because C/C++ array indexing is zero-based, the maximum legal index into an array is one less than the number of array elements.

Code analysis name: INDEX_EXCEEDS_MAX

Example

The following code generates warning C6201. The `for` loop condition exceeds the valid index range for `buff` when it sets `i` to 14, which is one element past the end:

```
C++

void f()
{
    int buff[14]; // array of 0..13 elements
    for (int i = 0; i <= 14; i++) // i == 14 exceeds the bounds
    {
        buff[i] = 0; // initialize buffer
    }
}
```

To correct the warning, make sure the index stays in bounds. The following code shows the corrected loop condition:

```
C++

void f()
{
    int buff[14]; // array of 0..13 elements
```

```
for (int i = 0; i < 14; i++) // i == 13 on the final iteration
{
    buff[i] = 0; // initialize buffer
}
}
```

Heuristics

This analysis is limited to stack-allocated arrays. It doesn't consider, for example, arrays passed into the function with a Microsoft source code annotation language ([SAL](#))-annotated length.

This analysis can't catch all possible out of bounds indices because not all arithmetic can be precisely analyzed. It's tuned to report cases where it can guarantee an out of bounds index is possible. The absence of a warning doesn't mean the index is guaranteed to be in bounds.

Warning C6211

Article • 10/07/2022

Leaking memory '*pointer*' due to an exception. Consider using a local catch block to clean up memory

This warning indicates that allocated memory isn't freed when an exception is thrown. The statement at the end of the path could throw an exception.

Remarks

The analyzer checks for this condition only when the

`_Analysis_mode_(_Analysis_local_leak_checks_)` SAL annotation is specified. By default, this annotation is specified for Windows kernel mode (driver) code. For more information about SAL annotations, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Code analysis name: `MEMORY_LEAK_EXCEPTION`

Example

The following code generates warning C6211 because an exception could be thrown during the second allocation and thereby leak the first allocation. Or, an exception could be thrown somewhere in the code that's represented by the "`code ...`" comment and thereby leak both allocations.

C++

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>

_Analysis_mode_(_Analysis_local_leak_checks_)
void f( )
{
    char *p1 = new char[10];
    char *p2 = new char[10];

    // code ...

    delete[] p2;
    delete[] p1;
}
```

To use the same allocation functions and correct this problem, add an exception handler:

```
C++  
  
// cl.exe /analyze /c /EHsc /nologo /W4  
#include <sal.h>  
#include <new>  
#include <iostream>  
using namespace std;  
  
_Analysis_mode_( _Analysis_local_leak_checks_ )  
  
void f()  
{  
    char *p1 = nullptr;  
    char *p2 = nullptr;  
  
    try  
    {  
        p1 = new char[10];  
        p2 = new char[10];  
  
        // code ...  
  
        delete [] p2;  
        delete [] p1;  
    }  
    catch (const bad_alloc& ba)  
    {  
        cout << ba.what() << endl;  
        delete [] p2;  
        delete [] p1;  
    }  
    // code ...  
}
```

To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include [shared_ptr](#), [unique_ptr](#), and containers such as [vector](#). For more information, see [Smart pointers](#) and [C++ Standard Library](#).

```
C++  
  
// cl.exe /analyze /c /EHsc /nologo /W4  
#include <sal.h>  
#include <vector>  
#include <memory>  
  
using namespace std;  
  
_Analysis_mode_( _Analysis_local_leak_checks_ )
```

```
void f( )
{
    // use 10-element vectors in place of char[10]
    vector<char> v1;
    vector<char> v2;

    for (int i=0; i<10; ++i) {
        v1.push_back('a');
        v2.push_back('b');
    }
    // code ...

    // use unique_ptr if you still want char[10]
    unique_ptr<char[]> a1(new char[10]);
    unique_ptr<char[]> a2(new char[10]);

    // code ...

    // No need for delete; vector and unique_ptr
    // clean up when out of scope.
}
```

See also

[C++ exception handling](#)

Warning C6214

Article • 10/07/2022

Cast between semantically different integer types: HRESULT to a Boolean type

This warning indicates that an `HRESULT` is being cast to a Boolean type. The success value (`S_OK`) of an `HRESULT` equals 0. However, 0 indicates failure for a Boolean type. Casting an `HRESULT` to a Boolean type and then using it in a test expression will yield an incorrect result.

Remarks

Sometimes, this warning occurs if an `HRESULT` is being stored in a Boolean variable. Any comparison that uses the Boolean variable to test for `HRESULT` success or failure could lead to incorrect results.

Code analysis name: `CAST_HRESULT_TO_BOOL`

Example

The following code generates warning C6214:

C++

```
#include <windows.h>

BOOL f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;
    hr = CoGetMalloc(1, &pMalloc);
    if ((BOOL)hr) // warning C6214
    {
        // success code ...
        return TRUE;
    }
    else
    {
        // failure code ...
        return FALSE;
    }
}
```

To correct this warning, use the following code:

C++

```
#include <windows.h>

BOOL f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (SUCCEEDED(hr))
    {
        // success code ...
        return TRUE;
    }
    else
    {
        // failure code ...
        return FALSE;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

Usually, the `SUCCEEDED` or `FAILED` macro should be used to test the value of an `HRESULT`.

For more information, see one of the following articles:

[SUCCEEDED](#)

[FAILED](#)

To make use of modern C++ memory allocation methodology, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Warning C6215

Article • 10/07/2022

Cast between semantically different integer types: a Boolean type to HRESULT

This warning indicates that a Boolean is being cast to an `HRESULT`. Boolean types indicate success by a non-zero value, whereas success (`S_OK`) in `HRESULT` is indicated by a value of 0. Casting a Boolean type to an `HRESULT` and then using it in a test expression will yield an incorrect result.

Remarks

This warning frequently occurs when a Boolean is used as an argument to `SUCCEEDED` or `FAILED` macro, which explicitly casts their arguments to an `HRESULT`.

Code analysis name: `CAST_BOOL_TO_HRESULT`

Example

The following code generates this warning:

C++

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

void f( REFGUID riid1, REFGUID riid2 )
{
    if (SUCCEEDED( IsEqual( riid1, riid2 ) )) //warning C6215
    {
        // success code ...
    }
    else
    {
        // failure code ...
    }
}
```

Generally, the `SUCCEEDED` or `FAILED` macros should only be applied to `HRESULT`.

To correct this warning, use the following code:

C++

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

void f( REFGUID riid1, REFGUID riid2 )
{
    if (IsEqual( riid1, riid2 ) == TRUE)
    {
        // code for riid1 == riid2
    }
    else
    {
        // code for riid1 != riid2
    }
}
```

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#)

Warning C6216

Article • 10/07/2022

Compiler-inserted cast between semantically different integral types: a Boolean type to HRESULT

A Boolean type is being used as an `HRESULT` without being explicitly cast.

Remarks

Boolean types indicate success by a non-zero value; success (`S_OK`) in `HRESULT` is indicated by a value of 0. A Boolean `false` value used as an `HRESULT` would indicate `S_OK`, which is frequently a mistake.

Code analysis name: `COMPILER_INSERTED_CAST_BOOL_TO_HRESULT`

Example

The following code generates this warning:

C++

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

HRESULT f( REFGUID riid1, REFGUID riid2 )
{
    // Oops, f() should return S_OK when the values are equal but will
    // return E_FAIL instead because IsEqual returns a c-style boolean values
    TRUE or FALSE
    return IsEqual(rivid1, rivid2);
}
```

To correct this warning, either add the appropriate conversion between the two types or add an explicit cast.

C++

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

HRESULT f( REFGUID rivid1, REFGUID rivid2 )
{
```

```
// converting because IsEqual returns a c-style TRUE or FALSE
return IsEqual(riid1, riid2) ? S_OK : E_FAIL;
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#).

Warning C6217

Article • 02/22/2023

Implicit cast between semantically different integer types: testing `HRESULT` with 'not'. Consider using `SUCCEEDED` or `FAILED` macro instead.

Remarks

This warning indicates that the code tests an `HRESULT` with the logical-not (!) operator. A value of 0 (the value defined for `S_OK`) indicates success in an `HRESULT`. However, 0 also indicates failure for a Boolean type. If you test an `HRESULT` with the logical-not operator (!) to determine which code block to run, it can cause incorrect behavior or code that confuses future maintainers.

To verify whether an `HRESULT` is a success or failure, use the `SUCCEEDED` or `FAILED` macros instead.

This warning works for both `HRESULT` and `SCODE` types.

Code analysis name: `TESTING_HRESULT_WITH_NOT`

Example

The following code generates this warning because it uses the logical-not (!) operator to determine success or failure of an `HRESULT` value. In this case, the code executes the wrong code path because an `HRESULT` of 0 indicates success, so (`!hr`) incorrectly runs the failure code:

C++

```
#include <windows.h>
#include <objbase.h>

void f( )
{
    HRESULT hr = CoInitialize(NULL);
    if (!hr)
    {
        // failure code ...
    }
    else
    {
        // success code ...
    }
}
```

```
}
```

To correct this warning, the following code uses a `FAILED` macro to check for failure:

C++

```
#include <windows.h>
#include <objbase.h>

void f( )
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
    {
        // failure code ...
    }
    else
    {
        // success code ...
    }
}
```

See also

[SUCCEEDED macro](#)

[FAILED macro](#)

Warning C6219

Article • 10/07/2022

Implicit cast between semantically different integer types: comparing `HRESULT` to 1 or `TRUE`. Consider using `SUCCEEDED` or `FAILED` macro instead

Remarks

This warning indicates an `HRESULT` is being compared with an explicit, non-`HRESULT` value of one (1). This comparison is likely to lead to incorrect results, because the typical success value of `HRESULT` (`S_OK`) is 0. If you compare this value to a Boolean type, it's implicitly converted to `false`.

Code analysis name: `COMPARING_HRESULT_TO_ONE`

Example

The following code generates this warning because the `CoGetMalloc` returns an `HRESULT`, which then is compared to `TRUE`:

C++

```
#include <windows.h>

void f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;
    hr = CoGetMalloc(1, &pMalloc);

    if (hr == TRUE)
    {
        // success code ...
    }
    else
    {
        // failure code
    }
}
```

Most of the time, this warning is caused by code that compares an `HRESULT` to a Boolean. It's better to use the `SUCCEEDED` or `FAILED` macros to test the value of an `HRESULT`. To correct this warning, use the following code:

C++

```
#include <windows.h>

void f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;
    hr = CoGetMalloc(1, &pMalloc);

    if (SUCCEEDED(hr))
    {
        // success code ...
    }
    else
    {
        // failure code
    }
}
```

For this warning, the `SCODE` type is treated as an `HRESULT`.

The use of `malloc` and `free` (and related dynamic memory APIs) has many pitfalls as a cause of memory leaks and exceptions. To avoid these kinds of leaks and exception problems, use the pointer and container classes provided by the C++ Standard Library. These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

Warning C6220

Article • 10/07/2022

Implicit cast between semantically different integer types: comparing HRESULT to -1.

Consider using `SUCCEEDED` or `FAILED` macro instead

This warning indicates that an `HRESULT` is being compared with an explicit, non-`HRESULT` value of -1, which isn't a well-formed `HRESULT`.

Remarks

A failure in `HRESULT` (`E_FAIL`) isn't represented by a -1. Therefore, an implicit cast of an `HRESULT` to an integer will generate an incorrect value and is likely to lead to the wrong result.

Code analysis name: `COMPARING_HRESULT_TO_MINUS_ONE`

Example

In most cases, warning C6220 is caused by code that mistakenly expects a function to return an integer, and to use -1 as a failure value, but instead the function returns an `HRESULT`. The following code sample generates this warning:

C++

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (hr == -1)
    {
        // failure code ...
        return E_FAIL;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

It's best to use the `SUCCEEDED` or `FAILED` macro to test the value of an `HRESULT`. To correct this warning, use the following code:

C++

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (FAILED(hr))
    {
        // failure code ...
        return E_FAIL;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

Explicit comparison is appropriate to check for specific `HRESULT` values, such as, `E_FAIL`. Otherwise, use the `SUCCEEDED` or `FAILED` macros.

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#).

The use of `malloc` and `free` (and related dynamic memory allocation APIs) has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Warning C6221

Article • 10/07/2022

Implicit cast between semantically different integer types: comparing `HRESULT` to an integer. Consider using `SUCCEEDED` or `FAILED` macros instead

This warning indicates that an `HRESULT` is being compared to an integer other than zero.

Remarks

A success in an `HRESULT` (`S_OK`) is represented by a 0. Therefore, an implicit cast of an `HRESULT` to an integer generates an incorrect value and is likely to lead to the wrong result. The error is often caused by mistakenly expecting a function to return an integer when it actually returns an `HRESULT`.

Code analysis name: `COMPARING_HRESULT_TO_INT`

Example

The following code generates warning C6221 by comparing an `HRESULT` against an integer value:

C++

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (hr == 4)
    {
        // failure code ...
        return S_FALSE;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

To correct this warning, the following code uses the `FAILED` macro:

C++

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (FAILED(hr))
    {
        // failure code ...
        return S_FALSE;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#).

The use of `malloc` and `free` (and related dynamic memory allocation APIs) has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Warning C6225

Article • 10/07/2022

Implicit cast between semantically different integer types: assigning 1 or `TRUE` to `HRESULT`. Consider using `S_FALSE` instead

This warning indicates that an `HRESULT` is being assigned or initialized with a value of an explicit 1. Boolean types indicate success by a non-zero value; success (`S_OK`) in `HRESULT` is indicated by a value of 0.

Remarks

This warning is frequently caused by accidental confusion of Boolean and `HRESULT` types. To indicate success, the symbolic constant `S_OK` should be used.

Code analysis name: `ASSIGNING_ONE_TO_HRESULT`

Example

In the following code, assignment of `HRESULT` generates warning C6225:

```
C++  
  
#include <windows.h>  
  
VOID f( )  
{  
    HRESULT hr;  
    LPMALLOC pMalloc;  
  
    if (SUCCEEDED(CoGetMalloc(1, &pMalloc)))  
    {  
        hr = S_OK;  
    }  
    else  
    {  
        hr = 1;  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
#include <windows.h>  
  
VOID f( )  
{  
    HRESULT hr;  
    LPMALLOC pMalloc;  
  
    if (SUCCEEDED(CoGetMalloc(1, &pMalloc)))  
    {  
        hr = S_OK;  
    }  
    else  
    {  
        hr = S_FALSE;  
    }  
}
```

```
VOID f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    if (SUCCEEDED(CoGetMalloc(1, &pMalloc)))
    {
        hr = S_OK;
    }
    else
    {
        hr = S_FALSE;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

To indicate failure, `E_FAIL`, or another constant, should be used.

For more information, see one of the following articles:

[SUCCEEDED](#)

[FAILED](#)

To make use of modern C++ memory allocation methodology, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Warning C6226

Article • 10/07/2022

Implicit cast between semantically different integer types: assigning -1 to HRESULT.

Consider using E_FAIL instead.

This warning indicates that an `HRESULT` is assigned or initialized to an explicit value of -1.

Remarks

This warning is frequently caused by accidental confusion of integer and `HRESULT` types. To indicate success, use the symbolic constant `S_OK` instead. To indicate failure, use the symbolic constants that start with `E_constant`, such as `E_FAIL`.

For more information, see the [SUCCEEDED](#) and [FAILED](#) macros.

Code analysis name: `ASSIGNING_MINUS_ONE_TO_HRESULT`

Example

The following code generates this warning:

```
C++  
  
#include <windows.h>  
  
VOID f( )  
{  
    HRESULT hr;  
    LPMALLOC pMalloc;  
  
    if (FAILED(CoGetMalloc(1, &pMalloc)))  
    {  
        hr = -1;  
        // code ...  
    }  
    else  
    {  
        // code ...  
    }  
}
```

To correct this warning, use the following code:

C++

```
#include <windows.h>

VOID f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    if (FAILED(CoGetMalloc(1, &pMalloc)))
    {
        hr = E_FAIL;
        // code ...
    }
    else
    {
        // code ...
    }
}
```

For this warning, the `SCODE` type is treated as an `HRESULT`.

The use of `malloc` and `free` (and related dynamic memory APIs) has many pitfalls as a cause of memory leaks and exceptions. To avoid these kinds of leaks and exception problems, use the pointer and container classes provided by the C++ Standard Library. These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

Warning C6230

Article • 10/07/2022

Implicit cast between semantically different integer types: using `HRESULT` in a Boolean context

Remarks

This warning indicates that a bare `HRESULT` is used in a context where a Boolean result is expected, such as an `if` statement. This test is likely to yield incorrect results. For example, the typical success value for `HRESULT` (`S_OK`) is false when it's tested as a Boolean.

Code analysis name: `USING_HRESULT_IN_BOOLEAN_CONTEXT`

Example

The following code generates this warning:

```
C++  
  
#include <windows.h>  
  
VOID f( )  
{  
    LPMALLOC pMalloc;  
    HRESULT hr = CoGetMalloc(1, &pMalloc);  
  
    if (hr)  
    {  
        // code ...  
    }  
    else  
    {  
        // code ...  
    }  
}
```

In most situations, the `SUCCEEDED` or `FAILED` macro should be used to test the value of the `HRESULT`. To correct this warning, use the following code:

```
C++  
  
#include <windows.h>  
  
VOID f( )  
{  
    LPMALLOC pMalloc;  
    HRESULT hr = CoGetMalloc(1, &pMalloc);  
  
    if (SUCCEEDED(hr))  
    {  
        // code ...  
    }  
    else  
    {  
        // code ...  
    }  
}
```

```
#include <windows.h>

VOID f( )
{
    LPMALLOC pMalloc;
    HRESULT hr = CoGetMalloc(1, &pMalloc);

    if (SUCCEEDED(hr))
    {
        // code ...
    }
    else
    {
        // code ...
    }
}
```

For this warning, the `SCODE` type is treated as an `HRESULT`.

The use of `malloc` and `free` (and related dynamic memory APIs) has many pitfalls as a cause of memory leaks and exceptions. To avoid these kinds of leaks and exception problems, use the pointer and container classes provided by the C++ Standard Library. These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

Warning C6235

Article • 10/07/2022

('non-zero constant' || 'expression') is always a non-zero constant

This warning indicates that a non-zero constant value, other than one, was detected on the left side of a logical-or operation that occurs in a test context. The right side of the logical-or operation isn't evaluated because the resulting expression always evaluates to true. This language feature is referred to as "short-circuit evaluation."

Remarks

A non-zero constant value, other than one, suggests that the bitwise-AND operator (&) may have been intended. This warning isn't generated for the common idiom when the non-zero constant is 1, because of its use for selectively enabling code paths. However, it's generated if the non-zero constant evaluates to 1, for example 1+0.

Code analysis name: NONZEROLOGICALOR

Example

The following code generates this warning because INPUT_TYPE is 2:

```
C++  
  
#define INPUT_TYPE 2  
void f(int n)  
{  
    if(INPUT_TYPE || n) //warning C6235 issued  
    {  
        puts("Always gets here");  
    }  
    else  
    {  
        puts("Never gets here");  
    }  
}
```

The following code uses the bitwise-AND (&) operator to correct this warning:

```
C++
```

```
#define INPUT_TYPE 2
void f(int n)
{
    if((INPUT_TYPE & n) == 2)
    {
        puts("bitwise-AND comparison true");
    }
    else
    {
        puts("bitwise-AND comparison false");
    }
}
```

See also

[C Logical Operators](#)

Warning C6236

Article • 10/07/2022

('expression' || 'non-zero constant') is always a non-zero constant

This warning indicates that a non-zero constant value, other than one, was detected on the right side of a logical OR operation that occurs in a test context. Logically, it implies that the test is redundant and can be removed safely. Alternatively, it suggests that the programmer may have intended to use a different operator, for example, the equality (==), bitwise-AND (&) or bitwise-XOR (^) operator, to test for a specific value or flag.

Remarks

This warning isn't generated for the common idiom when the non-zero constant is 1, because of its use for selectively enabling code paths at compile time. However, the warning is generated if the non-zero constant is formed by an expression that evaluates to 1, for example, 1 + 0.

Code analysis name: LOGICALORNONZERO

Example

This code shows how warning C6236 can appear. Because INPUT_TYPE isn't 0, the expression n || INPUT_TYPE is always non-zero, and the else clause is never executed. However, INPUT_TYPE is a constant with a value other than one, which suggests that it's meant as a value for comparison:

C++

```
#define INPUT_TYPE 2
#include <stdio.h>

void f( int n )
{
    if ( n || INPUT_TYPE ) // analysis warning C6236
    {
        puts( "Always gets here" );
    }
    else
    {
        puts( "Never enters here" );
    }
}
```

The following code instead uses a bitwise-AND (`&`) operator to test whether the `INPUT_TYPE` bit of the input parameter `n` is set:

C++

```
#define INPUT_TYPE 2
#include <stdio.h>

void f( int n )
{
    if ( n & INPUT_TYPE ) // no warning
    {
        puts( "Bitwise-AND comparison is true" );
    }
    else
    {
        puts( "Bitwise-AND comparison is false" );
    }
}
```

See also

[Bitwise AND operator: &](#)

Warning C6237

Article • 10/07/2022

('zero' && 'expression') is always zero. 'expression' is never evaluated and may have side effects

This warning indicates that a constant value of zero was detected on the left side of a logical-and operation that occurs in a test context. The resulting expression always evaluates to false. Therefore, the right side of the logical-AND operation isn't evaluated. This language feature is referred to as "short-circuit evaluation."

Remarks

You should examine the right side of the expression carefully: Ensure that any side effects such as assignment, function call, increment, and decrement operations needed for proper functionality aren't affected by the short-circuit evaluation.

The expression (0 && n) produces no side effects and is commonly used to selectively choose code paths.

Code analysis name: ZEROLOGICALANDLOSINGSIDEFFECTS

Example

The following code shows various code samples that generate this warning:

C++

```
#include <stdio.h>
#define INPUT_TYPE 0

int test();

// side effect: n not incremented
void f1( int n )
{
    if(INPUT_TYPE && n++) //warning: 6237
    {
        puts("code path disabled");
    }
    else
    {
        printf_s("%d - n was not incremented",n);
    }
}
```

```

}

// side effect: test() not called
void f2( )
{
    if(INPUT_TYPE && test()) //warning: 6237
    {
        puts("code path disabled");
    }
    else
    {
        puts("test() was not called");
    }
}

//side effect: assignment and function call did not occur
void f3( int n )
{
    if(INPUT_TYPE && ( n=test() ) ) //warning: 6237
    {
        puts("code path disabled");
    }
    else
    {
        printf_s("%d -- n unchanged. test() was not called", n);
    }
}

```

To correct this warning, use the following code:

C++

```

#include <stdio.h>
#define INPUT_TYPE 0
int test();

void f1( int n )
{
    if(INPUT_TYPE)
    {
        if(n++)
        {
            puts("code path disabled");
        }
    }
    else
    {
        puts("n was not incremented");
    }
}

void f2( )
{

```

```
if(INPUT_TYPE)
{
    if( test() )
    {
        puts("code path disabled");
    }
}
else
{
    puts("test() was not called");
}

void f3( int n )
{
    if(INPUT_TYPE)
    {
        n = test();
        if( n )
        {
            puts("code path disabled");
        }
    }
    else
    {
        puts("test() was not called");
    }
}
```

See also

[C Logical Operators](#)

Warning C6239

Article • 10/07/2022

('non-zero constant' && 'expression') always evaluates to the result of 'expression'.

Did you intend to use the bitwise-and operator?

This warning indicates that a non-zero constant value, other than one, was detected on the left side of a logical-AND operation that occurs in a test context. For example, the expression (2 && n) is reduced to (!!n), which is the Boolean value of n.

Remarks

This warning typically indicates an attempt to check a bit mask in which the bitwise-AND (&) operator should be used, and isn't generated if the non-zero constant evaluates to 1 because of its use for selectively choosing code paths.

Code analysis name: NONZEROLOGICALAND

Example

The following code generates this warning:

```
C++  
  
#include <stdio.h>  
#define INPUT_TYPE 2  
void f( int n )  
{  
    if(INPUT_TYPE && n) // warning C6239  
    {  
        puts("boolean value of n is true");  
    }  
    else  
    {  
        puts("boolean value of n is false");  
    }  
}
```

To correct this warning, use bitwise-AND (&) operator as shown in the following code:

```
C++
```

```
#include <stdio.h>
#define INPUT_TYPE 2
void f( int n )
{
    if( ( INPUT_TYPE & n ) )
    {
        puts("bitmask true");
    }
    else
    {
        puts("bitmask false");
    }
}
```

See also

[& Operator](#)

Warning C6240

Article • 10/07/2022

('expression' && 'non-zero constant') always evaluates to the result of 'expression'.

Did you intend to use the bitwise-and operator?

This warning indicates that a non-zero constant value, other than one, was detected on the right side of a logical-and operation that occurs in a test context. For example, the expression `(n && 3)` reduces to `(!!n)`, which is the Boolean value of `n`.

Remarks

This warning typically indicates an attempt to check a bit mask in which the bitwise-AND (`&`) operator should be used. It isn't generated if the non-zero constant evaluates to 1 because of its use for selectively choosing code paths.

Code analysis name: `LOGICALANDNONZERO`

Example

The following code generates this warning:

```
C++  
  
#include <stdio.h>  
#define INPUT_TYPE 2  
  
void f(int n)  
{  
    if (n && INPUT_TYPE)  
    {  
        puts("boolean value of !!n is true");  
    }  
    else  
    {  
        puts("boolean value of !!n is false");  
    }  
}
```

To correct this warning, use bitwise-AND operator as shown in the following code:

```
C++
```

```
#include <stdio.h>
#define INPUT_TYPE 2

void f(int n)
{
    if ( (n & INPUT_TYPE) )
    {
        puts("bitmask true");
    }
    else
    {
        puts("bitmask false");
    }
}
```

See also

[& Operator](#)

Warning C6242

Article • 10/07/2022

A jump out of this try-block forces local unwind. Incurs severe performance penalty

This warning indicates that a jump statement causes control-flow to leave the protected block of a `try-finally` other than by fall-through.

Remarks

Leaving the protected block of a `try-finally` other than by falling through from the last statement requires local unwind to occur. Local unwind typically requires approximately 1000 machine instructions, so it's detrimental to performance.

Use `_leave` to exit the protected block of a `try-finally`.

Code analysis name: `LOCALUNWINDFORCED`

Example

The following code generates this warning:

C++

```
#include <malloc.h>
void DoSomething(char *p); // function can throw exception

int f( )
{
    char *ptr = 0;
    __try
    {
        ptr = (char*) malloc(10);
        if ( !ptr )
        {
            return 0;    //Warning: 6242
        }
        DoSomething( ptr );
    }
    __finally
    {
        free( ptr );
    }
    return 1;
}
```

To correct this warning, use `__leave` as shown in the following code:

C++

```
#include <malloc.h>
void DoSomething(char *p);
int f()
{
    char *ptr = 0;
    int retVal = 0;

    __try
    {
        ptr = (char *) malloc(10);
        if ( !ptr )
        {
            retVal = 0;
            __leave; //No warning
        }
        DoSomething( ptr );
        retVal = 1;
    }
    __finally
    {
        free( ptr );
    }

    return retVal;
}
```

The use of `malloc` and `free` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Library. These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

See also

[try-except statement](#)

[try-finally statement](#)

Warning C6244

Article • 10/07/2022

Local declaration of '*variable*' hides previous declaration at '*line*' of '*file*'

Remarks

This warning indicates that a declaration has the same name as a declaration at an outer scope and hides the previous declaration. You won't be able to refer to the previous declaration from inside the local scope. Any intended use of the previous declaration will end up using the local declaration. This warning only identifies a scope overlap and not lifetime overlap.

Code analysis name: LOCALDECLHIDESGLOBAL

Example

The following code generates this warning:

```
C++  
  
#include <stdlib.h>  
#pragma warning(push)  
  
// disable warning C4101: unreferenced local variable  
#pragma warning(disable: 4101)  
  
int i;  
void f();  
void (*pf)();  
  
void test()  
{  
    // Hide global int with local function pointer  
    void (*i)(); //Warning: 6244  
  
    // Hide global function pointer with an int  
    int pf; //Warning: 6244  
}  
#pragma warning(pop)
```

To correct this warning, use the following sample code:

```
C++
```

```
#include <stdlib.h>
#pragma warning(push)
// disable warning C4101: unreferenced local variable
#pragma warning(disable: 4101)

int g_i;           // modified global variable name
void g_f();        // modified global function name
void (*f_pf)();   // modified global function pointer name

void test()
{
    void (*i)();
    int pf;
}
#pragma warning(pop)
```

When dealing with memory allocation, review code to determine whether an allocation was saved in one variable and freed by another variable.

Warning C6246

Article • 10/07/2022

Local declaration of '*variable*' hides declaration of same name in outer scope.

Additional Information: See previous declaration at '*location*'.

Remarks

This warning indicates that two declarations have the same name at local scope. The name at outer scope is hidden by the declaration at the inner scope. Any intended use of the outer scope declaration will result in the use of local declaration.

Code analysis name: LOCALDECLHIDESLOCAL

Example

The following code generates this warning:

```
C++  
  
#include <stdlib.h>  
#define UPPER_LIMIT 256  
int DoSomething( );  
  
int f( )  
{  
    int i = DoSomething( );  
    if (i > UPPER_LIMIT)  
    {  
        int i;  
        i = rand( );  
    }  
    return i;  
}
```

To correct this warning, use another variable name as shown in the following code:

```
C++  
  
#include <stdlib.h>  
#define UPPER_LIMIT 256  
int DoSomething( );  
  
int f ( )  
{
```

```
int i = DoSomething( );
if (i > UPPER_LIMIT)
{
    int j = rand( );
    return j;
}
else
{
    return i;
}
```

This warning only identifies a scope overlap.

Warning C6248

Article • 10/07/2022

Setting a SECURITY_DESCRIPTOR's DACL to NULL will result in an unprotected object.

Remarks

If the DACL that belongs to the security descriptor of an object is set to NULL, a null DACL is created. A null DACL grants full access to any user who requests it; normal security checking isn't performed with respect to the object. A null DACL shouldn't be confused with an empty DACL. An empty DACL is a properly allocated and initialized DACL that contains no ACEs. An empty DACL grants no access to the object it's assigned to. Objects that have null DACLs can have their security descriptors altered by malicious users, making it so that no one has access to the object. Even in a situation where everyone needs access to an object, only administrators should be able to alter that object's security. If only the creator needs access to an object, a DACL shouldn't be set on the object; the system will choose an appropriate default.

Code analysis name: `CREATINGNULLDACL`

Example

The following code generates this warning because a null DACL is passed to the `SetSecurityDescriptorDacl` function:

C++

```
#include <windows.h>

void f( PSECURITY_DESCRIPTOR pSecurityDescriptor )
{
    if (SetSecurityDescriptorDacl(pSecurityDescriptor,
                                TRUE,           // Dacl Present
                                NULL,          // NULL pointer to DACL
                                FALSE))        // Defaulted
    {
        // Dacl is now applied to an object
    }
}
```

To see a complete example on how to create security descriptor, see [Creating a Security Descriptor for a New Object in C++](#). For more information on creating DACLs, see [Creating a DACL](#).

Warning C6250

Article • 10/07/2022

Calling 'VirtualFree' without the `MEM_RELEASE` flag may free memory but not address descriptors (VADs); results in address space leaks

This warning indicates that a call to `VirtualFree` without the `MEM_RELEASE` flag only decommits the pages, and doesn't release them. To both decommit and release pages, use the `MEM_RELEASE` flag in the call to `VirtualFree`. If any pages in the region are committed, the function first decommits and then releases them. After this operation, the pages are in the free state. If you specify this flag, `dwSize` must be zero, and `lpAddress` must point to the base address returned by the `VirtualAlloc` function when the region was reserved. The function fails if either of these conditions isn't met.

Remarks

You can ignore this warning if your code later frees the address space by calling `VirtualFree` with the `MEM_RELEASE` flag.

For more information, see [VirtualAlloc](#) and [VirtualFree](#).

The use of `VirtualAlloc` and `VirtualFree` has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include [shared_ptr](#), [unique_ptr](#), and containers such as [vector](#). For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Code analysis name: `WIN32UNRELEASEDEVADS`

Example

The following sample code generates warning C6250:

C++

```
#include <windows.h>
#include <stdio.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size
```

```

VOID f( )
{
    LPVOID lpvBase;           // base address of the test memory
    SYSTEM_INFO sSysInfo;     // system information

    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL,                  // system selects address
        PAGELIMIT*dwPageSize,  // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS);

    //
    // code to access memory
    // ...

    if (lpvBase != NULL)
    {
        if (VirtualFree( lpvBase, 0, MEM_DECOMMIT )) // decommit pages
        {
            puts ("MEM_DECOMMIT Succeeded");
        }
        else
        {
            puts("MEM_DECOMMIT failed");
        }
    }
    else
    {
        puts("lpvBase == NULL");
    }
}

```

To correct this warning, use the following sample code:

C++

```

#include <windows.h>
#include <stdio.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( )
{
    LPVOID lpvBase;           // base address of the test memory
    SYSTEM_INFO sSysInfo;     // system information

    GetSystemInfo(&sSysInfo);

```

```
dwPageSize = sSysInfo.dwPageSize;

// Reserve pages in the process's virtual address space
lpvBase = VirtualAlloc(
    NULL,                      // system selects address
    PAGELIMIT*dwPageSize, // size of allocation
    MEM_RESERVE,
    PAGE_NOACCESS);

// code to access memory
// ...

if (lpvBase != NULL)
{
    if (VirtualFree(lpvBase, 0, MEM_RELEASE )) // decommit & release
    {
        // code ...
    }
    else
    {
        puts("MEM_RELEASE failed");
    }
}
else
{
    puts("lpvBase == Null ");
    // code...
}
```

Warning C6255

Article • 10/07/2022

`_alloca` indicates failure by raising a stack overflow exception. Consider using `_malloca` instead

This warning indicates that a call to `_alloca` has been detected outside of local exception handling.

Remarks

`_alloca` should always be called from within the protected range of an exception handler because it can raise a stack overflow exception on failure. If possible, instead of using `_alloca`, consider using `_malloca`, which is a more secure version of `_alloca`.

Code analysis name: UNPROTECTEDUSEOFALLOCA

Example

The following code generates this warning because `_alloca` can generate exception:

C++

```
#include <windows.h>

void f( )
{
    void *p = _alloca(10);
    // code ...
}
```

To correct this warning, use `_malloca` and add exception handler as shown in the following code:

C++

```
#include <windows.h>
#include <malloc.h>

void f( )
{
    void *p;
    int errcode;
```

```
__try
{
    p = _malloca(10);
    // code...
    _freea(p);
}
__except( (GetExceptionCode() == STATUS_STACK_OVERFLOW ) ?
          EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH )
{
    errcode = _resetstkoflw();
    // code ...
}
}
```

See also

[_malloca](#)

Warning C6258

Article • 10/07/2022

Using `TerminateThread` does not allow proper thread clean up.

This warning indicates that a call to `TerminateThread` has been detected.

Remarks

`TerminateThread` is a dangerous function that should only be used in the most extreme cases. For more information about problems associated with `TerminateThread` call, see [TerminateThread function](#).

Code analysis name: `USINGTERMINATETHREAD`

To properly terminate threads

1. Create an event object using the `CreateEvent` function.
2. Create the threads.
3. Each thread monitors the event state by calling the `WaitForSingleObject` function.
4. Each thread ends its own execution when the event is set to the signaled state (`WaitForSingleObject` returns `WAIT_OBJECT_0`).

See also

- [Terminating a Thread](#)
- [WaitForSingleObject](#)
- [SetEvent](#)

Warning C6259

Article • 10/07/2022

Labeled code is unreachable: ('*expression*' & '*constant*') in switch-expr cannot evaluate to '*case-label*'

Remarks

This warning indicates unreachable code caused by the result of a bitwise-AND (&) comparison in a switch expression. Only the case statement that matches the constant in the switch expression is reachable; all other case statements aren't reachable.

Code analysis name: DEADCODEINBITORLIMITEDSWITCH

Example

The following sample code generates this warning because the 'switch' expression `(rand() & 3)` can't evaluate to case label (`case 4`):

```
C++  
  
#include <stdlib.h>  
  
void f()  
{  
    switch (rand () & 3) {  
        case 3:  
            /* Reachable */  
            break;  
        case 4:  
            /* Not reachable */  
            break;  
        default:  
            break;  
    }  
}
```

To correct this warning, remove the unreachable code or verify that the constant used in the case statement is correct. The following code removes the unreachable case statement:

```
C++
```

```
#include <stdlib.h>

void f()
{
    switch (rand () & 3) {
        case 3:
            /* Reachable */
            break;
        default:
            break;
    }
}
```

See also

- [switch Statement \(C++\)](#)
- [switch Statement](#)

Warning C6260

Article • 10/07/2022

`sizeof * sizeof` is almost always wrong, did you intend to use a character count or a byte count?

This warning indicates that the results of two `sizeof` operations have been multiplied together.

Remarks

The C/C++ `sizeof` operator returns the number of bytes of storage an object uses. It's typically incorrect to multiply it by another `sizeof` operation. Usually, you're interested in the number of bytes in an object or the number of elements in an array (for example, the number of wide-characters in an array).

There's some unintuitive behavior associated with `sizeof` operator. For example, in C, `sizeof ('\\0') == 4`, because a character is of an integral type. In C++, the type of a character literal is `char`, so `sizeof ('\\0') == 1`. However, in both C and C++, the following relation is true:

C++

```
sizeof ("\\0") == 2
```

Code analysis name: USEOFBYTEAREA

Example

The following code generates this warning:

C++

```
#include <windows.h>

void f( )
{
    int i;
    i = sizeof (L"String") * sizeof (WCHAR);
    // code ...
}
```

To correct this warning, use the following code:

C++

```
#include <windows.h>

void f( )
{
    // use divide to calculate how many WCHAR characters are in the string
    int i = sizeof(L"String") / sizeof(WCHAR);

    // get the number of bytes in the character array
    int j = sizeof(L"String");

    // code ...
}
```

See also

- [sizeof Operator](#)
- [sizeof Operator \(C\)](#)

Warning C6262

Article • 10/07/2022

Function uses *constant_1* bytes of stack: exceeds /analyze:stacksize *constant_2*.

Consider moving some data to heap

Remarks

This warning indicates that stack usage that exceeds a preset threshold (*constant_2*) has been detected in a function. The default stack frame size for this warning is 16 KB for user mode, 1 KB for kernel mode. Stack—even in user mode—is limited, and failure to commit a page of stack causes a stack overflow exception. Kernel mode has a 12 KB stack size limit, which can't be increased. Try to aggressively limit stack use in kernel-mode code.

To correct the problem behind this warning, you can either move some data to the heap or to other dynamic memory. In user mode, one large stack frame may not be a problem—and this warning may be suppressed—but a large stack frame increases the risk of a stack overflow. (A large stack frame might occur if the function uses the stack heavily or is recursive.) The total stack size in user mode can be increased if stack overflow actually occurs, but only up to the system limit.

For kernel-mode code—for example, in driver projects—the value of *constant_2* is set to 1 KB. Well-written drivers should have few functions that approach this value, and changing the limit downward may be desirable. The same general techniques that are used for user-mode code to reduce the stack size can be adapted to kernel-mode code.

Code analysis name: EXCESSIVESTACKUSAGE

Adjust the stack size to suppress the warning

You can use the [/analyze:stacksize](#) command-line option to change the value for *constant_2*, but increasing it introduces a risk that an error may not be reported.

To suppress the warning on the command line

- Add the `/analyze:stacksize <new-size>` option to the compiler command line. Use a value for `<new-size>` that's larger than *constant_1*. For example, if *constant_1* is 27180, you might enter `/analyze:stacksize 32768`.

To suppress the warning in the IDE

1. In the Visual Studio IDE, select the project in the **Solution Explorer** window.
2. On the menu bar, choose **Project > Properties**.
3. In the **Property Pages** dialog box, select the **Configuration Properties > C/C++ > Command Line** property page.
4. In **Additional options**, add `/analyze:stacksize <new-size>`, where `<new-size>` is larger than `constant_1`. For example, if `constant_1` is 27180, you might enter `/analyze:stacksize 32768`. Choose **OK** to save your changes.

Example

The following code generates this warning because `char buffer` requires 16,382 bytes on the stack, and the local integer variable `i` requires another 4 bytes, which together exceed the default stack size limit of 16 KB.

```
C++

// cl.exe /c /analyze /EHsc /W4
#include <windows.h>
#define MAX_SIZE 16382

void f( )
{
    int i;
    char buffer[MAX_SIZE];

    i = 0;
    buffer[0]='\0';

    // code...
}
```

The following code corrects this warning by moving some data to heap.

```
C++

// cl.exe /c /analyze /EHsc /W4
#include <stdlib.h>
#include <malloc.h>
#define MAX_SIZE 16382

void f( )
{
```

```
int i;
char *buffer;

i = 0;
buffer = (char *) malloc( MAX_SIZE );
if (buffer != NULL)
{
    buffer[0] = '\0';
    // code...
    free(buffer);
}
}
```

The use of `malloc` and `free` has many pitfalls, such as memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

See also

[/STACK \(Stack allocations\)](#)

[_resetstkoflw](#)

[How to: Use native run-time checks](#)

Warning C6263

Article • 10/07/2022

Using `_alloca` in a loop; this can quickly overflow stack

Remarks

This warning indicates that calling `_alloca` inside a loop to allocate memory can cause stack overflow. `_alloca` allocates memory from the stack, but that memory is only freed when the calling function exits. Stack, even in user-mode, is limited, and failure to commit a page of stack causes a stack overflow exception. The `_resetstkoflw` function recovers from a stack overflow condition, allowing a program to continue instead of failing with a fatal exception error. If the `_resetstkoflw` function isn't called, there's no guard page after the previous exception. The next time that there's a stack overflow, there are no exceptions at all and the process terminates without warning.

You should avoid calling `_alloca` inside a loop if either the allocation size or the iteration count is unknown because it might cause stack overflow. In these cases, consider other options such as heap memory or [C++ Standard Library](#) classes.

Code analysis name: `USINGALLOCAINLOOP`

Example

The following code generates this warning:

C++

```
#include <windows.h>
#include <malloc.h>
#include <excpt.h>
#include <stdio.h>

#define MAX_SIZE 50

void f ( int size )
{
    char* cArray;
    __try
    {
        for(int i = 0; i < MAX_SIZE; i++)
        {
            cArray = (char *)_alloca(size);
```

```
// process cArray...
}
}
__except(GetExceptionCode() == STATUS_STACK_OVERFLOW ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH )
{
    // code...
    puts("Allocation Failed");
    _resetstkoflw();
}
}
```

The following code uses malloc() to correct this warning:

C++

```
#include <windows.h>
#define MAX_SIZE 50

void f ( int size )
{
    char* cArray;

    for(int i = 0; i < MAX_SIZE; i++)
    {
        cArray = (char *) malloc(size);
        if (cArray != NULL)
        {
            // process cArray...
            free(cArray);
        }
    }
}
```

See also

- [malloc](#)
- [_alloca](#)
- [_malloca](#)
- [C++ Standard Library](#)

Warning C6268

Article • 10/07/2022

Incorrect order of operations: ('TYPE1')('TYPE2')x + y. Possible missing parentheses in ('TYPE1')('TYPE2')x + y)

This warning indicates that a complex cast expression might involve a precedence problem when performing pointer arithmetic. Because casts group more closely than binary operators, the result might not be what the programmer intended. In some cases, this defect causes incorrect behavior or a program crash.

Remarks

In an expression such as:

C++

```
(char *)p + offset
```

the offset is interpreted as an offset in characters; however, an expression such as:

C++

```
(int *)(char *)p + offset
```

is equivalent to:

C++

```
((int *)(char *)p) + offset
```

and so the offset is interpreted as an offset in integers. In other words, it's equivalent to:

C++

```
(int *)((char *)p + (offset * sizeof(int)))
```

which isn't likely to be what the programmer intended.

Depending on the relative sizes of the two types, this offset can lead to a buffer overrun.

Code analysis name: MISPARENTHESIZED_CASTS

Example

The following code generates this warning:

C++

```
void f(int *p, int offset_in_bytes)
{
    int *ptr;
    ptr = (int *) (char *) p + offset_in_bytes;
    // code ...
}
```

To correct this warning, use the following code:

C++

```
void f(int *p, int offset_in_bytes)
{
    int *ptr;
    ptr = (int *) ((char *) p + offset_in_bytes);
    // code ...
}
```

Warning C6269

Article • 10/07/2022

Possible incorrect order of operations: dereference ignored

This warning indicates that the result of a pointer dereference is being ignored, which raises the question of why the pointer is being dereferenced in the first place.

Remarks

The compiler will correctly optimize away the gratuitous dereference. In some cases, however, this defect may reflect a precedence or logic error.

One common cause for this defect is an expression statement of the form:

C++

```
*p++;
```

If the intent of this statement is simply to increment the pointer `p`, then dereference is unnecessary; however, if the intent is to increment the location that `p` is pointing to, then the program won't behave as intended because `p++` construct is interpreted as `(p++)` instead of `(*p)++`.

Code analysis name: `POINTER_DEREF_DISCARDED`

Example

The following code generates this warning:

C++

```
#include <windows.h>

void f( int *p )
{
    // code ...
    if( p != NULL )
        *p++;
    // code ...
}
```

To correct this warning, use parentheses as shown in the following code:

C++

```
#include <windows.h>

void f( int *p )
{
    // code ...
    if( p != NULL )
        (*p)++;
    // code ...
}
```

Warning C6270

Article • 03/23/2023

Missing float argument to '*function-name*': add a float argument corresponding to conversion specifier '*number*'

Remarks

This warning indicates that not enough arguments are provided to match a format string. At least one of the missing arguments is a floating-point number. This defect can lead to crashes, in addition to potentially incorrect output.

Code analysis name: `MISSING_FLOAT_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates warning C6270. `sprintf_s` expects a second float argument as denoted by `%f` but none is provided:

```
C++

void f()
{
    char buff[25];
    sprintf_s(buff, sizeof(buff), "%s %f", "pi: ");
```

To correct this warning, pass the missing float argument as shown in the following code:

```
C++

void f()
{
    char buff[25];
    sprintf_s(buff, sizeof(buff), "%s %f", "pi: ", 3.14159);
```

See also

[Format specification syntax: printf and wprintf functions](#)
[sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l](#)

`sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l`

C4473

Warning C6271

Article • 03/23/2023

Extra argument passed to '*function*'

Remarks

This warning indicates that extra arguments are being provided beyond the ones specified by the format string. By itself, this defect doesn't have any visible effect although it indicates that the programmer's intent isn't reflected in the code.

Code analysis name: EXTRA_ARGUMENT_TO_FORMAT_FUNCTION

Example

The following sample code generates this warning:

C++

```
#include <stdio.h>

void f()
{
    char buff[5];

    sprintf(buff, "%d", 1, 2);
}
```

To correct this warning, remove the unused parameter or modify the format string to take it into account:

C++

```
#include <stdio.h>

void f()
{
    char buff[5];

    sprintf(buff, "%d, %d", 1, 2);
}
```

The following sample code calls the safe string manipulation function, `sprintf_s`, to correct this warning:

C++

```
#include <stdio.h>

void f()
{
    char buff[5];

    sprintf_s( buff, 5, "%d %d", 1, 2 ); //safe version
}
```

See also

[Format specification syntax: printf and wprintf functions](#)

[sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[C4474](#)

Warning C6272

Article • 03/23/2023

Non-float passed as argument '*number*' when float is required in call to '*function-name*'

Remarks

This warning indicates that the format string specifies that a float is required. For example, a `%f` or `%g` specification for `printf`, but a non-float such as an integer or string is being passed. This defect can lead to crashes, in addition to potentially incorrect output.

Code analysis name: `NON_FLOAT_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning. `%f` indicates a float is expected, but the integer `i` is passed instead:

```
C++  
  
void f()  
{  
    char buff[5];  
    int i=5;  
    sprintf_s(buff, sizeof(buff), "%s %f", "a", i);  
}
```

To correct this warning, change the format specifier or modify the parameters passed to the function. In this example, we correct this warning by using `%i` instead of `%f`.

```
C++  
  
void f()  
{  
    char buff[5];  
    int i=5;  
    sprintf_s(buff, sizeof(buff), "%s %i", "a", i);  
}
```

See also

[Format specification syntax: printf and wprintf functions](#)

[sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[C4477](#)

Warning C6273

Article • 03/23/2023

Non-integer passed as parameter '*number*' when an integer is required in call to '*function*'

Remarks

This warning indicates that the format string specifier in a `printf` like function is expecting an integer type, but a non-integer such as a `float`, string, or `struct` is being passed instead. This warning checks integer type specifiers like `%d`, and width/precision specifier that use integers like `.*f`. This defect is likely to result in incorrect output.

Code analysis name: `NON_INTEGER_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning because an integer is required instead of a `float` in the `sprintf` function:

C++

```
#include <stdio.h>

void f_defective()
{
    char buff[50];
    float f=1.5;

    sprintf(buff, "%d", f);
}
```

The following code uses an integer cast to correct this warning. Alternatively it could have corrected the warning by modifying the format specifier to match the type.

C++

```
#include <stdio.h>

void f_corrected()
{
    char buff[50];
    float f=1.5;
```

```
    sprintf(buff,"%d", (int)f);
}
```

The following code uses safe string manipulation function, `sprintf_s`, to correct this warning:

C++

```
#include <stdio.h>

void f_safe()
{
    char buff[50];
    float f=1.5;

    sprintf_s(buff,50,"%d", (int)f);
}
```

This warning isn't applicable on Windows 9x and Windows NT version 4 because %p isn't supported on these platforms.

See also

[Format specification syntax: printf and wprintf functions](#)

[sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l](#)

[C4477](#)

Warning C6274

Article • 03/23/2023

Non-character passed as parameter '*number*' when character is required in call to '*function*'

Remarks

This warning indicates that the format string specifies that a character is required (for example, a `%c` or `%C` specification) but a non-integer such as a float, string, or struct is being passed. This defect is likely to cause incorrect output.

Code analysis name: `NON_CHAR_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning:

```
C++

#include <stdio.h>

void f(char str[])
{
    char buff[5];
    sprintf(buff, "%c", str);
}
```

To correct this warning, use the following code:

```
C++

#include <stdio.h>

void f(char str[])
{
    char buff[5];
    sprintf(buff, "%c", str[0]);
}
```

The following code uses safe string manipulation function, `sprintf_s`, to correct this warning:

C++

```
#include <stdio.h>

void f(char str[])
{
    char buff[5];
    sprintf_s(buff,5,"%c", str[0]);
}
```

Format specification syntax: printf and wprintf functions

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

C4477

C4313

Warning C6276

Article • 10/07/2022

Cast between semantically different string types. Use of invalid string can lead to undefined behavior.

This warning indicates a potentially incorrect cast from a narrow character string (`char*`) to a wide character string (`wchar_t*`).

Remarks

Because the Microsoft compiler implements wide strings with a character size of 2 bytes, casting from a narrow string might produce strings that aren't correctly terminated. If you use such strings with the `wcs*` functions in the runtime library, they could cause buffer overruns and access violations.

Code analysis name: `CHAR_TO_WCHAR_CAST`

Example

The following code generates warning C6276. It's caused by an improper cast of the narrow string "a" (2 bytes, one for the 'a' and one for the null terminator) to a wide string (a 2-byte wide character 'a' with no null terminator):

```
C++  
  
#include <windows.h>  
  
void f()  
{  
    WCHAR szBuffer[8];  
    LPWSTR pSrc;  
    pSrc = (LPWSTR)"a";  
    wcscpy_s(szBuffer, pSrc);  
}
```

The following code corrects this warning. It removes the problem cast and adds an `L` prefix to the string to define it as a properly terminated wide character string:

```
C++  
  
#include <windows.h>  
  
void f()  
{  
    WCHAR szBuffer[8];  
    LPWSTR pSrc;  
    pSrc = L"a";  
    wcscpy_s(szBuffer, pSrc);  
}
```

```
#include <windows.h>

void f()
{
    WCHAR szBuffer[8];
    LPWSTR pSrc;
    pSrc = L"a";
    wcscpy_s(szBuffer, pSrc);
}
```

Warning C6277

Article • 10/07/2022

NULL application name with an unquoted path in call to '*function-name*': results in a security vulnerability if the path contains spaces

This warning indicates that the application name parameter is null and that there might be spaces in the executable path name.

Remarks

Unless the executable name is fully qualified, there's likely to be a security problem. A malicious user could insert a rogue executable with the same name earlier in the path. To correct this warning, you can specify the application name instead of passing null. Alternatively, if you do pass null for the application name, use quotation marks around the executable path.

Code analysis name: CREATEPROCESS_ESCAPE

Example

The following sample code generates warning C6277. The warning is caused by the NULL application name and from the executable path name having a space. Due to how the function parses spaces, there's a risk that a different executable could be run. For more information, see [CreateProcessA](#).

C++

```

        FALSE,
        0,
        NULL,
        NULL,
        &si,
        &pi ) )

{
    puts( "CreateProcess failed." );
    return;
}
// Wait until child process exits.
WaitForSingleObject( pi.hProcess, INFINITE );

// Close process and thread handles.
CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );
}

```

To correct this warning, use quotation marks around the executable path, as shown in the following example:

C++

```

#include <windows.h>
#include <stdio.h>

void f ()
{
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );

    if( !CreateProcessA( NULL,
                        "\"C:\\Program Files\\MyApp.exe\"",
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,
                        &pi ) )

    {
        puts( "CreateProcess failed." );
        return;
    }
    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.

```

```
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Warning C6278

Article • 10/07/2022

'variable' is allocated with array new [], but deleted with scalar delete. Destructors will not be called.

Remarks

This warning appears only in C++ code and indicates that the calling function has inconsistently allocated memory with the array `new []` operator, but freed it with the scalar `delete` operator. This usage is undefined behavior according to the C++ standard and the Microsoft C++ implementation.

There are at least three reasons that this mismatch is likely to cause problems:

- The constructors for the individual objects in the array are invoked, but the destructors aren't invoked.
- If global, or class-specific, `operator new` and `operator delete` aren't compatible with `operator new[]` and `operator delete[]`, unexpected results are likely to occur.
- It's always risky to rely on undefined behavior.

The exact ramifications of this defect are difficult to predict. It might result in leaks for classes with destructors that perform memory de-allocation. It could cause inconsistent behavior for classes with destructors that perform some semantically significant operation, or memory corruptions and crashes when operators have been overridden. In other cases the mismatch might be unimportant, depending on the implementation of the compiler and its libraries. Analysis tools can't always distinguish between these situations.

If memory is allocated with array `new []`, it should be freed with array `delete[]`.

Code analysis name: `ARRAY_NEW_DELETE_MISMATCH`

Example

The following sample code generates warning C6278:

C++

```
class A
{
    // members
};

void f( )
{
    A *pA = new A[5];
    // code ...
    delete pA;
}
```

To correct this warning, use the following sample code:

C++

```
void f( )
{
    A *pA = new A[5];
    // code ...
    delete[] pA;
}
```

If the underlying object in the array is a primitive type such as `int`, `float`, `enum`, or pointer, there are no destructors to call. In these cases, warning [C6283](#) is reported instead.

The use of `new` and `delete` has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Warning C6279

Article • 10/07/2022

'variable-name' is allocated with scalar new, deleted with array delete []

This warning appears only in C++ code and indicates that the calling function has inconsistently allocated memory with the scalar `new` operator, but freed it with the array `delete[]` operator. If memory is allocated with scalar `new`, it should typically be freed with scalar `delete`.

Remarks

The exact ramifications of this defect are difficult to predict. It might cause random behavior or crashes due to usage of uninitialized memory as constructors aren't invoked. Or, it might cause memory allocations and crashes in situations where operators have been overridden. The analysis tool doesn't currently distinguish between these situations.

To avoid these kinds of allocation problems altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers and C++ Standard Library](#).

Code analysis name: `NEW_ARRAY_DELETE_MISMATCH`

Example

The following code generates warning C6279. `A` is allocated using `new` but deleted using `delete[]`:

C++

```
class A
{
    // members
};

void f()
{
    A *pA = new A;
    //code ...
}
```

```
    delete[] pA;  
}
```

The following code avoids this warning by using `delete` instead:

C++

```
class A  
{  
    // members  
};  
  
void f()  
{  
    A *pA = new A;  
    //code ...  
    delete pA;  
}
```

See also

- [C6014](#)

Warning C6280

Article • 10/07/2022

'variable-name' is allocated with '*function-name-1*', but deleted with '*function-name-2*'

This warning indicates that the calling function has inconsistently allocated memory by using a function from one family and freed it by using a function from another.

Remarks

The analyzer checks for this condition only when the `_Analysis_mode_(_Analysis_local_leak_checks_)` SAL annotation is specified. By default, this annotation is specified for Windows kernel mode (driver) code. For more information about SAL annotations, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

For example, this warning would be produced if memory is allocated by using `malloc` but freed by using `GlobalFree` or `delete`. In the specific cases of mismatches between array `new[]` and scalar `delete`, more precise warnings are reported instead of this one.

Code analysis name: `MEMORY_ALLOCATION_MISMATCH`

Example

The following sample code generates this warning. `pInt` is allocated using `calloc` but is freed using the mismatched function `delete`:

C++

```
// C6280a_warning.cpp
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>

_ANALYSIS_MODE_(ANALYSIS_LOCAL_LEAK_CHECKS_)

void f(int arraySize)
{
    int *pInt = (int *)calloc(arraySize, sizeof (int));
    // code ...
```

```
    delete pInt;  
}
```

The following code avoids this warning by using the deallocation function `free`, the match to `calloc`:

C++

```
// C6280a_no_warning.cpp  
// cl.exe /analyze /c /EHsc /nologo /W4  
#include <sal.h>  
#include <stdlib.h>  
  
_Analysis_mode_( _Analysis_local_leak_checks_ )  
  
void f(int arraySize)  
{  
    int *pInt = (int *)calloc(arraySize, sizeof (int));  
    // code ...  
    free(pInt);  
}
```

Different API definitions can use different heaps. For example, `GlobalAlloc` uses the system heap, and `free` uses the process heap. This issue is likely to cause memory corruptions and crashes.

These inconsistencies apply to the `new/delete` and `malloc/free` memory allocation mechanisms. To avoid these kinds of potential inconsistencies altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

The following code avoids this problem entirely by using `unique_ptr`:

C++

```
// C6280b_no_warning.cpp  
// cl.exe /analyze /c /EHsc /nologo /W4  
#include <sal.h>  
#include <vector>  
#include <memory>  
  
using namespace std;  
  
_Analysis_mode_( _Analysis_local_leak_checks_ )  
  
void f(int arraySize)  
{
```

```
// use unique_ptr instead of calloc/malloc/new
unique_ptr<int[]> pInt(new int[arraySize]);

// code ...

// No need for free because unique_ptr
// cleans up when out of scope.
}
```

See also

[calloc](#)
[malloc](#)
[free](#)
[operator new](#)
[delete Operator](#)
[shared_ptr](#)
[unique_ptr](#)
[Smart pointers](#)

Warning C6281

Article • 10/07/2022

Incorrect order of operations: relational operators have higher precedence than bitwise operators

Remarks

This warning indicates a possible error in the operator precedence, which might produce incorrect results. You should check the precedence and use parentheses to clarify the intent. Relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) have higher precedence than bitwise operators (`&`, `|`, `^`).

Code analysis name: `BITWISERELATIONPRECEDENCEERROR`

Example

The following code generates this warning:

```
C++

#include <stdlib.h>
#define FORMAT 1
#define TYPE 2

void f(int input)
{
    if (FORMAT & TYPE != input)
    {
        // code...
    }
}
```

The following code uses parentheses to correct this warning:

```
C++

#include <stdlib.h>
#define FORMAT 1
#define TYPE 2

void f(int input)
{
    if ((FORMAT & TYPE) != input)
```

```
{  
    // code...  
}  
}
```

See also

[Compiler Warning \(level 3\) C4554](#)

Warning C6282

Article • 10/07/2022

Incorrect operator: assignment of constant in Boolean context. Consider using '==' instead

Remarks

This warning indicates that an assignment of a constant to a variable was detected in a test context. Assignment of a constant to a variable in a test context is almost always incorrect. Replace the `=` with `==`, or remove the assignment from the test context to resolve this warning.

Code analysis name: `ASSIGNMENTREPLACESTEST`

Example

The following code generates this warning:

```
C++  
  
void f( int i )  
{  
    while (i = 5)  
    {  
        // code  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f( int i )  
{  
    while (i == 5)  
    {  
        // code  
    }  
}
```

See also

Compiler Warning (level 4) C4706

Warning C6283

Article • 10/07/2022

'*variable-name*' is allocated with array `new []`, but deleted with scalar `delete`

This warning appears only in C++ code and indicates that the calling function has inconsistently allocated memory with the array `new []` operator, but freed it with the scalar `delete` operator.

Remarks

This defect might cause leaks, memory corruptions, and, in situations where operators have been overridden, crashes. If memory is allocated with array `new []`, it should typically be freed with array `delete[]`.

Warning C6283 only applies to arrays of primitive types such as integers or characters. If elements of the array are objects of class type then warning [C6278](#) is issued.

The use of `new` and `delete` has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Code analysis name: `PRIMITIVE_ARRAY_NEW_DELETE_MISMATCH`

Example

The following code generates warning C6283. `str` is allocated using `new ...[...]` but is freed using the mismatched function `delete`:

```
C++  
  
void f( )  
{  
    char *str = new char[50];  
    delete str;  
}
```

The following code remediates this warning by using the matching function `delete[]`:

```
C++  
  
void f( )  
{  
    char *str = new char[50];  
    delete[] str;  
}
```

```
void f( )
{
    char *str = new char[50];
    delete[] str;
}
```

Warning C6284

Article • 03/23/2023

Object passed as parameter when string is required in call to '*function*'

Remarks

This warning indicates that there's a mismatch between the format specifier and the type being used in a `printf`-style function. The format specifier is a C style String type such as `%s` or `%ws`, and the argument is a class/struct/union type. This defect can lead to crashes, in addition to potentially incorrect output.

This defect is frequently due to forgetting to convert an object string type such as `std::string`, `CComBSTR` or `bstr_t` into the C style string the `printf`-style function expects. If so, then the fix is to add the appropriate conversion to the type. The conversion is needed because the variadic parameters to `printf`-style functions are untyped, so no automatic conversion occurs.

Code analysis name: `OBJECT_AS_STRING_ARGUMENT_TO_FORMAT_FUNCTION`

Example

C++

```
#include <atlbase.h>
#include <string>

void f()
{
    char buff[50];
    CComBSTR bstrValue{"Hello"};
    std::string str{"World"};

    // Oops, %ws and %s require C-style strings but CComBSTR and std::strings
    // are being passed instead
    sprintf(buff, "%ws %s", bstrValue, str);
}
```

Fix the warning by adding the appropriate conversions:

C++

```
#include <atlbase.h>
#include <string>

void f()
{
    char buff[50];
    CComBSTR bstrValue{"Hello"};
    std::string str{"World"};

    // Fixed by adding a static_cast to the CComBSTR and calling c_str() on
    // the std::string
    sprintf(buff, "%ws %s", static_cast<wchar_t*>(bstrValue), str.c_str());
}
```

See also

[static_cast Operator](#)

[sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l](#)

[C4477](#)

[C4840](#)

Warning C6285

Article • 10/07/2022

('non-zero constant' || 'non-zero constant') is always a non-zero constant. Did you intend to use the bitwise-and operator?

This warning indicates that two constant values, both greater than one, were detected as arguments to a logical-or operation that occurs in a test context. This expression is always TRUE.

Remarks

Constant values greater than one suggest that the arguments to logical-or could be bit fields. Consider whether a bitwise operator might be a more appropriate operator in this case.

Code analysis name: LOGICALOROFCONSTANTS

Example

The following code generates this warning:

```
C++  
  
#include <stdio.h>  
#define TESTED_VALUE 0x37  
#define MASK 0xaa  
  
void f()  
{  
    if (TESTED_VALUE || MASK)  
    {  
        puts("(TESTED_VALUE || MASK) True");  
        // code ...  
    }  
    else  
    {  
        puts("(TESTED_VALUE || MASK) False");  
        // code ...  
    }  
}
```

To correct this warning, use the following code:

C++

```
#include <stdio.h>
#define TESTED_VALUE 0x37
#define MASK 0xaa

void f(int flag)
{
    if ((TESTED_VALUE & MASK)== flag)
    {
        puts("true");
        // code ...
    }
    else
    {
        puts("false");
        // code ...
    }
}
```

See also

[Expressions with Binary Operators](#)

Warning C6286

Article • 10/07/2022

('non-zero constant' || 'expression') is always a non-zero constant. 'expression' is never evaluated and may have side effects

Remarks

This warning indicates that a non-zero constant was detected on the left side of a logical-or operation that occurs in a test context. The resulting expression always evaluates to TRUE. In addition, the right side of the expression appears to have side effects, and they'll be lost.

You may want to examine the right side of the expression carefully to ensure that any side effects needed for proper functionality aren't lost.

The `(!0 || <expression>)` construction is commonly used to force execution of a controlled block.

Code analysis name: `NONZEROLOGICALORLOSINGSIDEFFECTS`

Example

The following code generates this warning:

C++

```
#include <stdlib.h>
#include <stdio.h>
#define INPUT_TYPE 1

int test();

void f()
{
    if (INPUT_TYPE || test())
    {
        puts("INPUT_TYPE == 1, expression not evaluated");
        // code...
    }
    else
    {
        puts("INPUT_TYPE == 0. Call to test() returned 0");
        // code...
    }
}
```

```
}
```

The following code shows one possible solution by breaking `if` statement into two separate parts:

C++

```
#include <stdlib.h>
#include <stdio.h>
#define INPUT_TYPE 1

int test();

void f()
{
    int i;
    if (INPUT_TYPE)
    {
        i = test();
        // code...
    }
    else
    {
        puts("INPUT_TYPE false");
        // code...
    }
}
```

See also

[Logical OR Operator: ||](#)

Warning C6287

Article • 10/07/2022

Redundant code: the left and right subexpressions are identical

Remarks

This warning is emitted when an expression contains redundant logic. The warning can indicate a logic error. For example, accidentally using the wrong variable. It might also be a redundant test that can be removed. Inspect the code to verify that there's no logic error.

Code analysis name: REDUNDANTTEST

Example

The following code generates this warning:

C++

```
void f(int x, int y)
{
    // comparing against x twice is suspicious, should the second comparison
    use y?
    if ((x == 1) && (x == 1))
    {
        //...
    }
}
```

The following code shows various ways to correct this warning:

C++

```
void f(int x, int y)
{
    // Fixed the second comparison to use y
    if ((x == 1) && (y == 1))
    {
        // ...
    }

    // If the second comparison was unnecessary it could be removed
    if (x == 1)
    {
```

```
// ...  
}  
}
```

Warning C6288

Article • 10/07/2022

Incorrect operator: mutual inclusion over `&&` is always zero. Did you intend to use `||` instead?

Remarks

This warning indicates that in a test expression, a variable is being tested against two different constants. The result depends on both conditions being true, which is impossible. The code in these cases indicates that the programmer's intent isn't captured correctly. It's important to examine the code and correct the problem. Otherwise, your code won't behave the way you expected it to.

This problem is often caused by using `&&` in place of `||`, but can also be caused by using `==` where `!=` was intended.

Code analysis name: `MUTUALINCLUSIONOVERANDISFALSE`

Example

The following code generates this warning:

```
C++  
  
void f(int x)  
{  
    if ((x == 1) && (x == 2)) // warning  
    {  
        // code ...  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f(int x)  
{  
    if ((x == 1) || (x == 2))  
    {  
        // logic  
    }  
}
```

```
/* or */
if ((x != 1) && (x != 2))
{
    // code ...
}
```

The analysis tool doesn't warn if the expression has side effects.

Warning C6289

Article • 10/07/2022

Incorrect operator: mutual exclusion over `||` is always a non-zero constant. Did you intend to use `&&` instead?

Remarks

This warning indicates that in a test expression a variable is being tested as unequal to two different constants. The result depends on either condition being true, but it always evaluates to true.

This problem is often caused by using `||` in place of `&&`, but can also be caused by using `!=` where `==` was intended.

Code analysis name: `MUTUALEXCLUSIONOVERORISTRUE`

Example

The following code generates this warning:

```
C++  
  
void f(int x)  
{  
    if ((x != 1) || (x != 3))  
    {  
        // code  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f(int x)  
{  
    if ((x != 1) && (x != 3))  
    {  
        // code  
    }  
}  
  
/* or */
```

```
void f(int x)
{
    if ((x == 1) || (x == 3))
    {
        // code
    }
}
```

Warning C6290

Article • 10/07/2022

Bitwise operation on logical result: ! has higher precedence than &. Use && or !(x & y)) instead

This warning indicates possible confusion in the use of an operator or an operator precedence.

Remarks

The ! operator yields a Boolean result, and it has higher precedence than the &. The bitwise-and (&) operator takes two arithmetic arguments. Therefore, one of the following errors has been detected:

- The expression is mis-parenthesized:

Because the result of ! is Boolean (zero or one), an attempt to test that two variables have bits in common will only end up testing that the lowest bit is present in the right side: ((!8) & 1) == 0.

- The ! operator is incorrect, and should be a ~ instead:

The ! operator has a Boolean result, while the ~ operator has an arithmetic result. These operators are never interchangeable, even when operating on a Boolean value (zero or one): (!0x01) & 0x10 == 0x0, while (~0x01) & 0x10 == 0x10.

- The binary operator & is incorrect, and should instead be &&:

While & can sometimes be interchanged with &&, it isn't equivalent because it forces evaluation of the right side of the expression. Certain side effects in this type of expression can be terminal.

It's difficult to judge the severity of this problem without examining the code. The code should be inspected to ensure that the intended test is occurring.

Code analysis name: LOGICALNOTBITWISEAND

Example

The following code generates this warning:

C++

```
void f(int x, int y)
{
    if (!x & y)
    {
        // code ...
    }
}
```

To correct this warning, use the following sample code:

C++

```
void f(int x, int y)
{
    /* When testing that x has no bits in common with y. */
    if (!(x & y))
    {
        // code ...
    }

    /* When testing for the complement of x in y. */
    if ((~x) & y)
    {
        // code ...
    }
}

#include <windows.h>
void fc(int x, BOOL y )
{
    /* When y is a Boolean or Boolean result. */
    if ((!x) && y)
    {
        // code ...
    }
}
```

Warning C6317 is reported if the ! operator is on the right side of the & operator.

See also

- [C6317](#)

Warning C6291

Article • 10/07/2022

Bitwise operation on logical result: `!` has higher precedence than `|`. Use `||` or `(!(x | y))` instead

The `!` operator yields a Boolean result, and the `|` (bitwise-or) operator takes two arithmetic arguments. The `!` operator also has higher precedence than `|`.

Therefore, one of the following errors has been detected:

- The expression is mis-parenthesized:

Because the result of `!` is Boolean (zero or one), an attempt to test that two variables have bits set will only end up testing that the lowest bit is present in the right side: `((!x) | y) != (!(x | y))` when `x == 0` and `y == 1`.

- The `!` operator is incorrect, and should be a `~` instead:

The `!` operator has a Boolean result, but the `~` operator has an arithmetic result. These operators are never interchangeable, even when operating on a Boolean value (zero or one): `((!x) | y) != ((~x) | y)` when `x == 1` and `y == 0`.

- The binary operator `|` is incorrect, and should instead be `||`:

Even though `|` can sometimes be interchanged with `||`, it isn't equivalent because it forces evaluation of the right side of the expression. Certain side-effects in this type of expression can be terminal: `(!p | (*p == '\0'))`, when `p == NULL`, we must dereference it to evaluate the other half of the expression.

This warning isn't reported if the `!` operator is on the right side of the `|` operator because this case is typically just the relatively harmless case of an incorrect operator.

It's difficult to judge the severity of this problem without examining the code. The code should be inspected to ensure that the intended test is occurring.

This warning always indicates possible confusion in the use of an operator or operator precedence.

Code analysis name: `LOGICALNOTBITWISEOR`

Example

The following code generates this warning:

C++

```
void f(int x, int y )
{
    if (!x | y)
    {
        //code
    }
}
```

To correct this warning, use one of the examples shown in the following code:

C++

```
void fC(int x, int y )
{
    /* When checking whether any bits are set in either x or y. */
    if (!(x | y))
    {
        // code
    }
    /* When checking whether bits are set in either */
    /* the complement of x or in y. */
    if ((~x) | y)
    {
        // code
    }
}

#include <windows.h>
void f(int x, BOOL y )
{
    /* When y is a Boolean or Boolean result. */
    if ((!x) || y)
    {
        // code
    }
}
```

Warning C6292

Article • 10/07/2022

Ill-defined for-loop: counts up from maximum

Remarks

This warning indicates that a for-loop might not function as intended.

It occurs when a loop counts up from a maximum, but has a lower termination condition. This loop will terminate only after integer overflow occurs.

Code analysis name: `LOOP_COUNTS_UP_FROM_MAX`

Example

The following code generates this warning:

```
C++  
  
void f( )  
{  
    int i;  
  
    for (i = 100; i >= 0; i++)  
    {  
        // code ...  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f( )  
{  
    int i;  
  
    for (i = 100; i >= 0; i--)  
    {  
        // code ...  
    }  
}
```

Warning C6293

Article • 10/07/2022

III-defined for-loop: counts down from minimum

Remarks

This warning indicates that a for-loop might not function as intended. It occurs when a loop counts down from a minimum, but has a higher termination condition.

A signed or unsigned index variable, together with a negative increment, will cause the loop to count negative until an overflow occurs, which will terminate the loop.

Code analysis name: `LOOP_INDEX_Goes_NEGATIVE`

Example

The following sample code generates this warning:

```
C++  
  
void f( )  
{  
    signed char i;  
  
    for (i = 0; i < 100; i--)  
    {  
        // code ...  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f( )  
{  
    signed char i;  
  
    for (i = 0; i < 100; i++)  
    {  
        // code ...  
    }  
}
```

Warning C6294

Article • 10/07/2022

III-defined for-loop: initial condition does not satisfy test. Loop body not executed

Remarks

This warning indicates that a for-loop can't be executed because the terminating condition is true. This warning suggests that the programmer's intent isn't correctly captured.

Code analysis name: `LOOP_BODY_NEVER_EXECUTED`

Example

The following sample code generates this warning because MAX_VALUE is 0:

```
C++  
  
#define MAX_VALUE 0  
void f()  
{  
    int i;  
    for (i = 0; i < MAX_VALUE; i++)  
    {  
        // code  
    }  
}
```

The following sample code corrects this warning by changing the value of MAX_VALUE to 25

```
C++  
  
#define MAX_VALUE 25  
void f()  
{  
    int i;  
    for (i = 0; i < MAX_VALUE; i++)  
    {  
        // code  
    }  
}
```

Warning C6295

Article • 10/07/2022

III-defined for-loop: '*variable*' values are of the range "min" to "max". Loop executed indefinitely

Remarks

This warning indicates that a for-loop might not function as intended. The for-loop tests an unsigned value against zero (0) with `>=`. The result is always true, therefore the loop is infinite.

Code analysis name: `INFINITE_LOOP`

Example

The following code generates this warning:

```
C++  
  
void f( )  
{  
    for (unsigned int i = 100; i >= 0; i--)  
    {  
        // code ...  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f( )  
{  
    for (unsigned int i = 100; i > 0; i--)  
    {  
        // code ...  
    }  
}
```

Warning C6296

Article • 10/07/2022

III-defined for-loop: Loop body only executed once

Remarks

This warning indicates that a for-loop might not function as intended. When the index is unsigned and a loop counts down from zero, its body is run only once.

Code analysis name: `LOOP_ONLY_EXECUTED_ONCE`

Example

The following code generates this warning:

C++

```
void f( )
{
    unsigned int i;

    for (i = 0; i < 100; i--)
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

C++

```
void f( )
{
    unsigned int i;

    for (i = 0; i < 100; i++)
    {
        // code ...
    }
}
```

Warning C6297

Article • 10/07/2022

Arithmetic overflow: 32-bit value is shifted, then cast to 64-bit value. Result may not be an expected value

Remarks

This warning indicates incorrect behavior that results from integral promotion rules and types larger than the ones in which arithmetic is typically performed.

In this case, a 32-bit value was shifted left, and the result of that shift was cast to a 64-bit value. If the shift overflowed the 32-bit value, bits are lost.

If you don't want to lose bits, cast the value to shift to a 64-bit quantity before it's shifted. If you want to lose bits, perform the appropriate cast to `unsigned long` or a `short` type. Or, mask the result of the shift. Either approach eliminates this warning and makes the intent of the code clearer.

Code analysis name: `RESULTOFSHIFTCASTTOLARGERSIZE`

Example

The following code generates this warning:

```
C++  
  
void f(int i)  
{  
    unsigned __int64 x;  
  
    x = i << 34;  
    // code  
}
```

To correct this warning, use the following code:

```
C++  
  
void f(int i)  
{  
    unsigned __int64 x;  
    // code
```

```
x = static_cast<unsigned __int64>(i) << 34;  
}
```

See also

[Compiler Warning \(level 1\) C4293](#)

Warning C6298

Article • 10/07/2022

Using a read-only string '*pointer*' as a writable string argument: this will attempt to write into static read-only memory and cause random crashes

Remarks

This warning indicates the use of a constant string as an argument to a function that might modify the contents of that string. Because the compiler allocates constant strings in a static read-only memory, any attempts to modify it cause access violations and random crashes.

This warning can be avoided by storing the constant string into a local array and then using the array as the argument to the function.

Code analysis name: CONST_STRING_TO_WRITABLE_STRING

Example

The following sample code generates this warning:

C++

```
#include <windows.h>
#include <stdio.h>

void f()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );
    if( !CreateProcess(NULL,
                      "\c:\\Windows\\system32\\calc.exe\",
                      NULL,
                      NULL,
                      FALSE,
                      0,
                      NULL,
                      NULL,
                      &si,
                      &pi ) )
}
```

```

        puts( "CreateProcess failed." );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}

```

To correct this warning, use the following sample code:

C++

```

#include <windows.h>
#include <stdio.h>

void f( )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    char szCmdLine[] = "\"c:\\Windows\\system32\\calc.exe\"";
    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );

    if( !CreateProcess( NULL,
                       szCmdLine,
                       NULL,
                       NULL,
                       FALSE,
                       0,
                       NULL,
                       NULL,
                       &si,
                       &pi ) )
    {
        puts( "CreateProcess failed." );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}

```

Warning C6299

Article • 10/07/2022

Explicitly comparing a bit field to a Boolean type will yield unexpected results

Remarks

This warning indicates an incorrect assumption that Booleans and bit fields are equivalent. Assigning 1 to bit fields will place 1 in its single bit; however, any comparison of this bit field to 1 includes an implicit cast of the bit field to a signed int. This cast will convert the stored 1 to a -1 and the comparison can yield unexpected results.

Code analysis name: `BITFIELD_TO_BOOL_COMPARISON`

Example

The following code generates this warning:

```
C++  
  
struct myBits  
{  
    short flag : 1;  
    short done : 1;  
    //other members  
} bitType;  
  
void f( )  
{  
    if (bitType.flag == 1)  
    {  
        // code...  
    }  
}
```

To correct this warning, use a bit field as shown in the following code:

```
C++  
  
void f ()  
{  
    if(bitType.flag==bitType.done)  
    {  
        // code...  
    }  
}
```


Warning C6302

Article • 03/23/2023

Format string mismatch.

Remarks

This warning indicates that a format string specifies a wide character string, but is being passed a narrow character string instead. One cause of the warning is because the meaning of `%s` and `%S` flip when used with `printf` or `wprintf`. This defect can lead to crashes, in addition to potentially incorrect output.

Code analysis name: `CHAR_WCHAR_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following code generates this warning. `buff` is a narrow character string, but `wprintf_s` is the wide string variant of `printf_s` and so expects `%s` to be wide:

```
C++  
  
void f()  
{  
    char buff[5] = "hi";  
    wprintf_s(L"%s", buff);  
}
```

The following sample code remediates this issue by using `%hs` to specify a single-byte character string. Alternatively it could have switched to `%S`, which is a narrow string when used with `wprintf` like functions. See [Format specification syntax: printf and wprintf functions](#) for more options.

```
C++  
  
void f()  
{  
    char buff[5] = "hi";  
    wprintf_s(L"%hs", buff);  
}
```

See also

[Format specification syntax: printf and wprintf functions](#)

[C4477](#)

[C6303](#)

Warning C6303

Article • 03/23/2023

Format string mismatch.

Remarks

This warning indicates that a format string specifies a narrow character string, but is being passed a wide character string instead. One cause of the warning is because the meaning of `%s` and `%'s` flip when used with `printf` or `wprintf`. This defect can lead to crashes, in addition to potentially incorrect output.

Code analysis name: `WCHAR_CHAR_ARGUMENT_TO_FORMAT_FUNCTION`

Example

The following sample code generates this warning. `buff` is a wide character string, but the `printf_s` call expects a short string as denoted by `%s`:

```
C++  
  
#include <stdio.h>  
  
void f()  
{  
    wchar_t buff[5] = L"hi";  
    printf_s("%s", buff);  
}
```

The following sample code remediates this issue by using `%ls` to specify a wide character string. Alternatively it could have switched to `%'s`, which is a wide string when used with `printf` like functions. See [Format specification syntax: printf and wprintf functions](#) for more options.

```
C++  
  
#include <stdio.h>  
  
void f()  
{  
    wchar_t buff[5] = L"hi";  
    printf_s("%ls", buff);  
}
```

See also

[Format specification syntax: printf and wprintf functions](#)

[C4477](#)

[C6302](#)

Warning C6305

Article • 10/07/2022

Potential mismatch between sizeof and countof quantities

Remarks

This warning indicates that a variable holding a `sizeof` result is being added to or subtracted from a pointer or `countof` expression. This operation will cause unexpected scaling in pointer arithmetic.

Code analysis name: `SIZEOF_COUNTOF_MISMATCH`

Example

The following code generates this warning:

```
C++  
  
void f(int *p)  
{  
    int cb=sizeof(int);  
    //code...  
    p +=cb; // warning C6305  
}
```

To correct this warning, use the following code:

```
C++  
  
void f(int *p)  
{  
    // code...  
    p += 1;  
}
```

Warning C6306

Article • 10/07/2022

Incorrect call to '*function*': consider using '*function*' which accepts a `va_list` as an argument

Remarks

This warning indicates an incorrect function call. The `printf` family includes several functions that take a variable list of arguments; however, these functions can't be called with a `va_list` argument. There's a corresponding `vprintf` family of functions that can be used for such calls. Calling the wrong print function will cause incorrect output.

Code analysis name: INCORRECT_VARARG_FUNCTIONCALL

Example

The following code generates this warning:

```
C++

#include <stdio.h>
#include <stdarg.h>

void f(int i, ...)
{
    va_list v;

    va_start(v, i);
    //code...
    printf("%s", v); // warning C6306
    va_end(v);
}
```

To correct this warning, use the following code:

```
C++

#include <stdio.h>
#include <stdarg.h>

void f(int i, ...)
{
    va_list v;
```

```
va_start(v, i);
//code...
vprintf_s("%d",v);
va_end(v);
}
```

See also

[C6273](#)

Warning C6308

Article • 10/07/2022

'realloc' may return null pointer: assigning a null pointer to '*parameter-name*', which is passed as an argument to 'realloc', will cause the original memory block to be leaked

Remarks

Heap reallocation functions don't free the passed buffer if reallocation is unsuccessful, potentially resulting in a memory leak if not handled properly. To correct the issue, assign the result of the reallocation function to a temporary variable, and then replace the original pointer after successful reallocation.

Code analysis name: REALLOCLEAK

Example

The following sample code generates warning C6308. This issue stems from the assignment of the return value from `realloc` to `x`. If `realloc` fails and returns a null pointer, then the original memory pointed to by `x` won't be freed:

C++

```
#include <malloc.h>
#include <windows.h>

void f( )
{
    char *x = (char *) malloc(10);
    if (x != NULL)
    {
        x = (char *) realloc(x, 512);
        // code...
        free(x);
    }
}
```

To resolve the issue, you can create a temporary variable to store the return value of `realloc`. This change allows you to free the previously allocated memory safely if `realloc` fails:

C++

```
#include <malloc.h>
#include <windows.h>

void f()
{
    char *x = (char *) malloc(10);
    if (x != NULL)
    {
        char *tmp = (char *) realloc(x,512);
        if (tmp != NULL)
        {
            x = tmp;
        }
        // code...
        free(x);
    }
}
```

This warning might generate noise if there's a live alias to the buffer-to-be-reallocated at the time of the assignment of the result of the reallocation function.

To avoid these kinds of issues altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include [shared_ptr](#), [unique_ptr](#), and containers such as [vector](#). For more information, see [Smart pointers and C++ Standard Library](#).

See also

[Warning C6014](#)

Warning C6310

Article • 10/07/2022

Illegal constant in exception filter can cause unexpected behavior

Remarks

This message indicates that an illegal constant was detected in the filter expression of a structured exception handler. The constants defined for use in the filter expression of a structured exception handler are:

- `EXCEPTION_CONTINUE_EXECUTION`
- `EXCEPTION_CONTINUE_SEARCH`
- `EXCEPTION_EXECUTE_HANDLER`

These values are defined in the run-time header file `excpt.h`.

Using a constant that isn't in the preceding list can cause unexpected behavior.

Code analysis name: `ILLEGALXCEPTEXPRCONST`

Example

The following code generates this warning:

C++

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

BOOL LimitExceeded();

void fd( )
{
    __try
    {
        if (LimitExceeded())
        {
            RaiseException(EXCEPTION_ACCESS_VIOLATION, 0, 0, 0);
        }
        else
        {
```

```
// code
}
}
__except (EXCEPTION_ACCESS_VIOLATION)
{
    puts("Exception Occurred");
}
}
```

To correct this warning, use the following code:

C++

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

BOOL LimitExceeded();

void fd( )
{
    __try
    {
        if (LimitExceeded())
        {
            RaiseException(EXCEPTION_ACCESS_VIOLATION,0,0,0);
        }
        else
        {
            // code
        }
    }
    __except (GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        puts("Exception Occurred");
    }
}
```

Warning C6312

Article • 10/07/2022

Possible infinite loop: use of the constant EXCEPTION_CONTINUE_EXECUTION in the exception-filter expression of a try-except

Remarks

This warning indicates the use of the constant `EXCEPTION_CONTINUE_EXECUTION` (or another constant that evaluates to -1) in the filter expression of a structured exception handler. Use of the constant value `EXCEPTION_CONTINUE_EXECUTION` could lead to an infinite loop. For example, if an exception was raised by hardware, the instruction that caused the exception will be restarted. If the address that caused the exception is still bad, another exception will occur and be handled in the same way. The result is an infinite loop.

An explicit call to `RaiseException` won't directly cause an infinite loop, but it will continue execution of the code in the protected block. This behavior can be unexpected, and could lead to an infinite loop if `RaiseException` was used to avoid dereferencing an invalid pointer.

Typically, `EXCEPTION_CONTINUE_EXECUTION` should be returned only by a function called in the filter expression, which has a chance to fix either the pointer that caused the exception or the underlying memory.

Code analysis name: `EXCEPTIONCONTINUEEXECUTION`

Example

The following code generates this warning:

C++

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

void f (char *ptr)
{
    __try
    {
        // exception occurs if the caller passes null ptr
```

```
// code...
*ptr = '\0';
}
__except (EXCEPTION_CONTINUE_EXECUTION)
// When EXCEPTION_CONTINUE_EXECUTION is used, the handler
// block of the structured exception handler is not executed.
{
    puts("This block is never executed");
}
}
```

To correct this warning, use the following code:

C++

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

void f (char *ptr)
{
    __try
    {
        // exception occurs if the caller passes null ptr
        // code...
        *ptr = '\0';
    }
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        puts("Error Occurred");
    }
}
```

Warning C6313

Article • 10/07/2022

Incorrect operator: Zero-valued flag cannot be tested with bitwise-and. Use an equality test to check for zero-valued flags

Remarks

This warning indicates that a constant value of zero was provided as an argument to the bitwise-and (&) operator in a test context. The resulting expression is constant and evaluates to false; the result is different than intended.

This warning is typically caused by using bitwise-and to test for a flag that has the value zero. To test zero-valued flags, a test for equality must be performed, for example, using `==` or `!=`.

Code analysis name: `BITANDVSZEROVALUEDFLAG`

Example

The following code generates this warning:

```
C++  
  
#define FLAG 0  
  
void f(int Flags )  
{  
    if (Flags & FLAG)  
    {  
        // code  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
#define FLAG 0  
  
void f(int Flags )  
{  
    if (Flags == FLAG)  
    {
```

```
// code  
}  
}
```

Warning C6314

Article • 10/07/2022

Incorrect order of operations: bitwise-or has higher precedence than the conditional-expression operator. Add parentheses to clarify intent

Remarks

This message indicates that an expression that contains a bitwise-or operator (`|`) was detected in the tested expression of a conditional operation (`? :`).

The conditional operator has lower precedence than bitwise operators. If the tested expression should contain the bitwise-or operator, then parentheses should be added around the conditional-expression.

Code analysis name: `BITORVSQUESTION`

Example

The following code generates this warning:

```
C++  
  
int SystemState();  
  
int f(int SignalValue)  
{  
    return SystemState() | (SignalValue != 0) ? 1 : 0;  
}
```

To correct this warning, use the following code:

```
C++  
  
int SystemState();  
  
int f(int SignalValue)  
{  
    return SystemState() | ((SignalValue != 0) ? 1 : 0);  
}
```

See also

- Bitwise Inclusive OR Operator: |
- Conditional Operator: ?:

Warning C6315

Article • 10/07/2022

Incorrect order of operations: bitwise-and has higher precedence than bitwise-or.

Add parentheses to clarify intent

Remarks

This warning indicates that an expression in a test context contains both bitwise-and (`&`) and bitwise-or (`|`) operations, but causes a constant because the bitwise-or operation happens last. Parentheses should be added to clarify intent.

Code analysis name: `BITORVSBITAND`

Example

The following code generates this warning:

C++

```
void f( int i )
{
    if ( i & 2 | 4 ) // warning
    {
        // code
    }
}
```

To correct this warning, add parenthesis as shown in the following code:

C++

```
void f( int i )
{
    if ( i & ( 2 | 4 ) )
    {
        // code
    }
}
```

Warning C6316

Article • 10/07/2022

Incorrect operator: tested expression is constant and non-zero. Use bitwise-and to determine whether bits are set

Remarks

This warning indicates the use of bitwise-or (`|`) when bitwise-and (`&`) should have been used. Bitwise-or adds bits to the resulting expression, whereas bitwise-and selects only those bits in common between its two operators. Tests for flags must be performed with bitwise-and or a test of equality.

Code analysis name: `INAPPROPRIATEUSEOFBITOR`

Example

The following code generates this warning:

```
C++  
  
#define INPUT_VALUE 2  
void f( int Flags)  
{  
    if (Flags | INPUT_VALUE) // warning  
    {  
        // code  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
#define ALLOWED 1  
#define INPUT_VALUE 2  
  
void f( int Flags)  
{  
    if ((Flags & INPUT_VALUE) == ALLOWED)  
    {  
        // code  
    }  
}
```


Warning C6317

Article • 10/07/2022

Incorrect operator: logical-not (!) is not interchangeable with ones-complement (~)

Remarks

This warning indicates that a logical-not (!) is being applied to a constant that is likely to be a bit-flag. The result of logical-not is Boolean; it's incorrect to apply the bitwise-and (&) operator to a Boolean value. Use the ones-complement (~) operator to flip flags.

Code analysis name: NOTNOTCOMPLEMENT

Example

The following code generates this warning:

```
C++  
  
#define FLAGS 0x4004  
  
void f(int i)  
{  
    if (i & !FLAGS) // warning  
    {  
        // code  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
#define FLAGS 0x4004  
  
void f(int i)  
{  
    if (i & ~FLAGS)  
    {  
        // code  
    }  
}
```

See also

- Bitwise AND Operator: &
- Logical Negation Operator: !
- One's Complement Operator: ~

Warning C6318

Article • 10/07/2022

III-defined `__try`/`__except`: use of the constant `EXCEPTION_CONTINUE_SEARCH` or another constant that evaluates to zero in the exception-filter expression. The code in the exception handler block is not executed

Remarks

This warning indicates that if an exception occurs in the protected block of this structured exception handler, the exception won't be handled because the constant `EXCEPTION_CONTINUE_SEARCH` is used in the exception filter expression.

This code is equivalent to the protected block without the exception handler block because the handler block isn't executed.

Code analysis name: `EXCEPTIONCONTINUESEARCH`

Example

The following code generates this warning:

C++

```
#include <excpt.h>
#include <stdio.h>

void f (char *pch)
{
    __try
    {
        // assignment might fail
        *pch = 0;
    }
    __except (EXCEPTION_CONTINUE_SEARCH) // warning C6318
    {
        puts("Exception Occurred");
    }
}
```

To correct this warning, use the following code:

C++

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

void f (char *pch)
{
    __try
    {
        // assignment might fail
        *pch = 0;
    }
    __except( (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH )
    {
        puts("Access violation");
    }
}
```

See also

[try-except Statement](#)

Warning C6319

Article • 10/07/2022

Use of the comma-operator in a tested expression causes the left argument to be ignored when it has no side-effects

Remarks

This warning indicates an ignored sub-expression in test context because of the comma-operator (,). The comma operator has left-to-right associativity. The result of the comma-operator is the last expression evaluated. If the left expression to comma-operator has no side effects, the compiler might omit code generation for the expression.

Code analysis name: IGNOREDBYCOMMA

Example

The following code generates this warning:

```
C++  
  
void f()  
{  
    int i;  
    int x[10];  
  
    for ( i = 0; x[i] != 0, x[i] < 42; i++) // warning  
    {  
        // code  
    }  
}
```

To correct this warning, use the logical AND operator as shown in the following code:

```
C++  
  
void f()  
{  
    int i;  
    int x[10];  
  
    for ( i = 0; (x[i] != 0) && (x[i] < 42); i++)  
    {
```

```
// code  
}  
}
```

See also

- [Logical AND Operator: &&](#)
- [Comma Operator: ,](#)

Warning C6320

Article • 10/07/2022

Exception-filter expression is the constant EXCEPTION_EXECUTE_HANDLER. This may mask exceptions that were not intended to be handled

Remarks

This warning indicates the side effect of using `EXCEPTION_EXECUTE_HANDLER` constant in an `__except` block. In this case, the statement in the `__except` block will always execute to handle the exception, including exceptions you didn't want to handle in a particular function. It's recommended that you check the exception before handling it.

Code analysis name: `EXCEPTIONEXECUTEHANDLER`

Example

The following code generates this warning because the `__except` block will try to handle exceptions of all types:

C++

```
#include <stdio.h>
#include <excpt.h>

void f(int *p)
{
    __try
    {
        puts("in try");
        *p = 13; // might cause access violation exception
        // code ...
    }
    __except(EXCEPTION_EXECUTE_HANDLER) // warning
    {
        puts("in except");
        // code ...
    }
}
```

To correct this warning, use `GetExceptionCode` to check for a particular exception before handling it as shown in the following code:

C++

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

void f(int *p)
{
    __try
    {
        puts("in try");
        *p = 13;      // might cause access violation exception
        // code ...
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        puts("in except");
        // code ...
    }
}
```

See also

[try-except Statement](#)

Warning C6322

Article • 10/07/2022

Empty `__except` block

Remarks

This message indicates that there's no code in the `__except` block. As a result, exceptions might go unhandled.

Code analysis name: EXCEPT_BLOCK_EMPTY

Example

The following code generates this warning:

C++

```
#include <stdio.h>
#include <excpt.h>
#include <windows.h>

void fd(char *pch)
{
    __try
    {
        // exception raised if pch is null
        *pch= 0 ;
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION)
    {
    }
}
```

To correct this warning, use the following code:

C++

```
#include <stdio.h>
#include <excpt.h>
#include <windows.h>

void f(char *pch)
{
    __try
```

```
{  
    // exception raised if pch is null  
    *pch= 0 ;  
}  
__except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?  
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)  
{  
    // code to handle exception  
    puts("Exception Occurred");  
}  
}
```

See also

[try-except Statement](#)

Warning C6323

Article • 10/07/2022

Use of arithmetic operator on Boolean type(s)

Remarks

This warning occurs if arithmetic operators are used on Boolean data types. Use of incorrect operator might yield incorrect results. It also indicates that the programmer's intent isn't reflected in the code.

Code analysis name: ARITH_OP_ON_BOOL

Example

The following code generates this warning:

```
C++  
  
int test(bool a, bool b)  
{  
    int c = a + b;      //C6323  
    return c;  
}
```

To correct this warning, use correct data type and operator.

```
C++  
  
int test(int a, int b)  
{  
    int c = a + b;  
    return c;  
}
```

Warning C6324

Article • 10/07/2022

Potential incorrect use of '*function1*': Did you intend to use '*function2*'?

Remarks

This warning indicates that a string copy function was used where a string comparison function should have been used. Incorrect use of the function can cause an unexpected logic error.

Code analysis name: `STRCPY_INSTEAD_OF_STRCMP`

Example

The following code generates this warning:

```
C++  
  
#include <string.h>  
  
void f(char *title )  
{  
    if (strcpy (title, "Manager") == 0) // warning C6324  
    {  
        // code  
    }  
}
```

To correct this warning, use `strcmp` as shown in the following code:

```
C++  
  
#include <string.h>  
  
void f(char *title )  
{  
    if (strcmp (title, "Manager") == 0)  
    {  
        // code  
    }  
}
```

See also

- [strcpy, wcscopy, _mbscopy](#)
- [strcpy_s, wcscopy_s, _mbscopy_s](#)
- [strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l](#)
- [_mbsnbcpy, _mbsnbcpy_l](#)
- [strcmp, wcscmp, _mbscmp](#)
- [strncmp, wcsncmp, _mbsncmp, _mbsncmp_l](#)
- [_mbsnbcmp, _mbsnbcmp_l](#)

Warning C6326

Article • 10/07/2022

Potential comparison of a constant with another constant

Remarks

This warning indicates a potential comparison of a constant with another constant, which is redundant code. You must check to make sure that your intent is properly captured in the code. In some cases, you can simplify the test condition to achieve the same result.

Code analysis name: CONST_CONST_COMP

Example

The following code generates this warning because two constants are compared:

C++

```
#define LEVEL
const int STD_LEVEL = 5;

const int value =
#ifdef LEVEL
    10;
#else
    5;
#endif

void f()
{
    if( value > STD_LEVEL)
    {
        // code...
    }
    else
    {
        // code...
    }
}
```

The following code shows one way to correct this warning by using C++17 `if constexpr`.

C++

```
#define LEVEL
const int STD_LEVEL = 5;

const int value =
#ifdef LEVEL
    10;
#else
    5;
#endif

void f()
{
    if constexpr( value > STD_LEVEL)
    {
        // code...
    }
    else
    {
        // code...
    }
}
```

The following code shows one way to correct this warning by using the #ifdef statements to determine which code should execute if C++17 is unavailable:

C++

```
#define LEVEL
const int STD_LEVEL = 5;

const int value =
#ifdef LEVEL
    10;
#else
    5;
#endif

void f ()
{
#ifdef LEVEL
    {
        // code...
    }
#else
    {
        // code...
    }
#endif
}
```

Warning C6328

Article • 03/07/2024

Size mismatch: 'type' passed as *Param(number)* when 'type' is required in call to '*function-name*'

Remarks

This warning indicates that type required by the format specifier and the type of the expression passed in don't match. Using the wrong format specifier is undefined behavior. To fix the warning, make sure that the format specifiers match the types of the expressions passed in.

Code analysis name: FORMAT_SIZE_MISMATCH

Example

```
C++  
  
#include <cstdio>  
  
void f(long long a)  
{  
    printf("%d\n", a); // C6328 emitted.  
}
```

There are multiple ways to fix the undefined behavior. We can change the format specifier:

```
C++  
  
#include <cstdio>  
  
void f(long long a)  
{  
    printf("%lld\n", a); // No C6328 emitted.  
}
```

We can change the type of the expression:

```
C++
```

```
#include <cstdio>

void f(int a)
{
    printf("%d\n", a); // No C6328 emitted.
}
```

As a last resort when overflow can't happen, we can introduce a cast:

C++

```
#include <cstdio>

void f(unsigned char a)
{
    printf("%hd\n", static_cast<signed char>(a)); // No C6328 emitted.
}
```

See also

[C6340](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Warning C6329

Article • 10/07/2022

Return value for a call to 'function' should not be checked against 'number'

Remarks

The program is comparing a number against the return value from a call to `CreateFile`.

If `CreateFile` succeeds, it returns an open handle to the object. If it fails, it returns `INVALID_HANDLE_VALUE`.

Code analysis name: `POTENTIAL_INCORRECT_RETVAL_CHECK`

Examples

This code could cause the warning:

```
C++  
  
if (CreateFile() == NULL)  
{  
    return;  
}
```

This code corrects the error:

```
C++  
  
if (CreateFile() == INVALID_HANDLE_VALUE)  
{  
    return;  
}
```

Warning C6330

Article • 10/07/2022

'*type1*' passed as Parameter('number') when '*type2*' is required in call to '*function*'

Code analysis name: POTENTIAL_ARGUMENT_TYPE_MISMATCH

Warning C6331

Article • 10/07/2022

Invalid parameter: passing `MEM_RELEASE` and `MEM_DECOMMIT` in conjunction to `*function*` is not allowed. This results in the failure of this call

This message indicates that an invalid parameter is passed to `VirtualFree` or `VirtualFreeEx`. `VirtualFree` and `VirtualFreeEx` both reject the flags (`MEM_RELEASE` | `MEM_DECOMMIT`) in combination. Therefore, the values `MEM_DECOMMIT` and `MEM_RELEASE` may not be used together in the same call.

Remarks

It's not required for decommit and release to occur as independent steps. Releasing committed memory will decommit the pages as well. Also, ensure the return value of this function isn't ignored.

Code analysis name: `VirtualFreeInvalidParam1`

Example

The following sample code generates warning C6331:

C++

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID fd( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc (
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE
    );
}
```

```

        MEM_RESERVE,
        PAGE_NOACCESS );
if (lpvBase)
{
    // code to access memory
}
else
{
    return;
}
bSuccess = VirtualFree(lpvBase,
    0,
    MEM_DECOMMIT | MEM_RELEASE); // warning
// code...
}

```

To correct this warning, don't pass `MEM_DECOMMIT` to the `VirtualFree` call, as shown in the following code:

C++

```

#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc (
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );
if (lpvBase)
{
    // code to access memory
}
else
{
    return;
}
bSuccess = VirtualFree(lpvBase, 0, MEM_RELEASE);
// code...
}

```

The use of `malloc` and `free` (and related dynamic memory allocation APIs) has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

See also

[VirtualAlloc Method](#)

[VirtualFree Method](#)

Warning C6332

Article • 10/07/2022

Invalid parameter: passing zero as the dwFreeType parameter to 'function' is not allowed. This results in the failure of this call

This warning indicates that an invalid parameter is being passed to `VirtualFree` or `VirtualFreeEx`.

Remarks

`VirtualFree` and `VirtualFreeEx` both reject a `dwFreeType` parameter of zero. The `dwFreeType` parameter can be either `MEM_DECOMMIT` or `MEM_RELEASE`. However, the values `MEM_DECOMMIT` and `MEM_RELEASE` may not be used together in the same call. Also, make sure that the return value of the `VirtualFree` function isn't ignored.

Code analysis name: `VirtualFreeInvalidParam2`

Example

The following code generates warning C6332 because an invalid parameter is passed to the `VirtualFree` function:

C++

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
```

```

        MEM_RESERVE,
        PAGE_NOACCESS );

if (lpvBase)
{
    // code to access memory
}
else
{
    return;
}

bSuccess = VirtualFree( lpvBase, 0, 0 );
// code ...
}

```

To correct this warning, modify the call to the `VirtualFree` function, as shown in the following code:

C++

```

#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0;    // count of pages
DWORD dwPageSize;    // page size

VOID f( VOID )
{
    LPVOID lpvBase;           // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo;     // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
                NULL,                  // system selects address
                PAGELIMIT*dwPageSize,   // size of allocation
                MEM_RESERVE,
                PAGE_NOACCESS );
    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }

    bSuccess = VirtualFree( lpvBase, 0, MEM_RELEASE );
    // code ...
}

```

The use of `VirtualAlloc` and `VirtualFree` has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

See also

[VirtualAlloc Method](#)

[VirtualFree Method](#)

Warning C6333

Article • 10/07/2022

Invalid parameter: passing `MEM_RELEASE` and a non-zero `dwSize` parameter to '*function_name*' is not allowed. This results in the failure of this call

Remarks

Both `VirtualFree` and `VirtualFreeEx` reject a `dwFreeType` of `MEM_RELEASE` with a non-zero value of `dwSize`. When `MEM_RELEASE` is passed, the `dwSize` parameter must be zero.

Code analysis name: `VIRTUALFREEINVALIDPARAM3`

Example

The following code sample generates this warning:

C++

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f(VOID)
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS);

    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }
}
```

```
}

bSuccess = VirtualFree(lpvBase, PAGELIMIT * dwPageSize, MEM_RELEASE);
//code...
}
```

To correct this warning, ensure that the value of `dwSize` is 0 in the call to `VirtualFree`:

C++

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f(VOID)
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS);

    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }
    bSuccess = VirtualFree(lpvBase, 0, MEM_RELEASE);

    // VirtualFree(lpvBase, PAGELIMIT * dwPageSize, MEM_DECOMMIT);
    // code...
}
```

You can also use `VirtualFree(lpvBase, PAGELIMIT * dwPageSize, MEM_DECOMMIT);` call to decommit pages, and later release them using `MEM_RELEASE` flag.

See also

- VirtualAlloc Method
- VirtualFree Method

Warning C6334

Article • 10/07/2022

`sizeof` operator applied to an expression with an operator may yield unexpected results

Remarks

The `sizeof` operator, when applied to an expression, yields the size of the type of the resulting expression.

Code analysis name: `SIZEOFEEXPR`

Example

The following code generates this warning. Since `a - 4` is an expression, `sizeof` will return the size of the resulting pointer, not the size of the structure found at that pointer:

C++

```
void f( )
{
    size_t x;
    char a[100];
    x = sizeof(a - 4);
    assert(x == 96); //assert fails since x == sizeof(char*)
}
```

To correct this warning, ensure that you're working with the return value of `sizeof`, not the argument to it:

C++

```
void f( )
{
    size_t x;
    char a[100];
    x = sizeof(a) - 4;
    assert(x == 96); //assert succeeds
}
```

See also

[sizeof Operator](#)

Warning C6335

Article • 08/26/2024

Leaking process information handle '*handlename*'

Remarks

This warning indicates that the process information handles returned by the CreateProcess family of functions need to be closed using CloseHandle. Failure to do so will cause handle leaks.

Code analysis name: LEAKING_PROCESS_HANDLE

Example

The following code generates this warning:

C++

```
#include <windows.h>
#include <stdio.h>

void f( )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( "C:\\WINDOWS\\system32\\calc.exe",
                        NULL,
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,      // Pointer to STARTUPINFO structure.
                        &pi ) ) // Pointer to PROCESS_INFORMATION

    {
        puts("Error");
        return;
    }
    // Wait until child process exits.
```

```
    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
}
```

To correct this warning, call `CloseHandle(pi.hThread)` to close thread handle as shown in the following code:

C++

```
#include <windows.h>
#include <stdio.h>

void f( )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( "C:\\WINDOWS\\system32\\calc.exe",
                        NULL,
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,      // Pointer to STARTUPINFO structure.
                        &pi ) ) // Pointer to PROCESS_INFORMATION

    {
        puts("Error");
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

For more information, see [CreateProcess Function](#) and [CloseHandle Function](#).

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Warning C6336

Article • 10/07/2022

Arithmetic operator has precedence over question operator, use parentheses to clarify intent

Remarks

This warning indicates a possible operator precedence problem. The '+', '-', '*' and '/' operators have precedence over the '?' operator. If the precedence in the expression isn't correct, use parentheses to change the operator precedence.

Code analysis name: QUESTIONPRECEDENCE

Example

The following code generates this warning:

```
C++  
  
int Count();  
  
void f(int flag)  
{  
    int result;  
    result = Count() + flag ? 1 : 2;  
    // code...  
}
```

To correct this warning, add parenthesis as shown in the following code:

```
C++  
  
int Count();  
  
void f(int flag)  
{  
    int result;  
    result = Count() + (flag ? 1 : 2);  
    // code...  
}
```

See also

[C++ Built-in Operators, Precedence and Associativity](#)

Warning C6340

Article • 03/07/2024

Mismatch on sign: 'type' passed as *Param(number)* when some (signed|unsigned) type is required in call to '*function-name*'

Remarks

This warning indicates that sign of the type required by the format specifier and sign of the type of the expression passed in don't match. Using the wrong format specifier is undefined behavior. To fix the warning, make sure that the format specifiers match the types of the expressions passed in.

Code analysis name: FORMAT_SIGN_MISMATCH

Example

```
C++  
  
#include <cstdio>  
  
void f(unsigned char a)  
{  
    printf("%hd\n", a); // C6340 emitted.  
}
```

There are multiple ways to fix the undefined behavior. We can change the format specifier:

```
C++  
  
#include <cstdio>  
  
void f(unsigned char a)  
{  
    printf("%hu\n", a); // No C6340 emitted.  
}
```

We can change the type of the expression:

```
C++
```

```
#include <cstdio>

void f(signed char a)
{
    printf("%hd\n", a); // No C6340 emitted.
}
```

As a last resort when overflow can't happen, we can introduce a cast:

C++

```
#include <cstdio>

void f(long long a)
{
    printf("%d\n", static_cast<int>(a)); // No C6328 emitted.
}
```

See also

[C6328](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Warning C6381

Article • 10/07/2022

Shutdown API '*function*' requires a valid dwReason or lpMessage

Remarks

This warning is issued if `InitiateSystemShutdownEx` is called:

- Without passing a valid shutdown reason (`dwReason`). If `dwReason` parameter is zero, the default is an undefined shutdown. By default, it's also an unplanned shutdown. You should use one of the System Shutdown Reason Codes for this parameter.
- Without passing a shutdown message (`lpMessage`).

We recommend that you use appropriate parameters when calling this API to help system administrators determine the cause of the shutdown.

Code analysis name: `SHUTDOWN_API`

Example

The following code generates this warning because `dwReason` is zero and `lpMessage` is null:

C++

```
void f()
{
    //...
    BOOL bRet;
    bRet = InitiateSystemShutdownEx( NULL,
                                    NULL, // message
                                    0,
                                    FALSE,
                                    TRUE,
                                    0); // shutdown reason
    // ...
}
```

To correct this warning, specify `dwReason` and `lpMessage` as shown in the following code:

C++

```
#include <windows.h>
void f()
{
    //...
    BOOL bRet;
    bRet = InitiateSystemShutdownEx( NULL,
                                    "Hardware Failure", // message
                                    0,
                                    FALSE,
                                    TRUE,
                                    SHTDN_REASON_MAJOR_HARDWARE ); // reason
    // ...
}
```

Warning C6383

Article • 10/07/2022

Buffer overrun due to conversion of an element count into a byte count: an element count is expected for parameter `*parameter_name*` in call to `*function_name*`

This warning indicates that a non-constant byte count is being passed when an element count is instead required.

Remarks

Typically, this warning occurs when a variable is multiplied by the `sizeof` a type. This issue will likely result in more bytes being copied to the buffer than it can hold.

Code analysis name: `ELEMENTS_TO_BYTES`

Example

The following code generates this warning. `wcsncpy` will allow `n * sizeof(wchar_t)` characters to be copied, but the buffer can only hold `n` characters. It should be noted that `wcsncpy` is an unsafe function, and shouldn't be used per [C28719](#). The unsafe variant is used here only for the purposes of demonstrating this warning:

C++

```
void f(wchar_t* t, wchar_t* s, int n)
{
    wcsncpy (t, s, n*sizeof(wchar_t));
}
```

The following code corrects this warning by sending element count instead of the byte count:

C++

```
void f( wchar_t* t, wchar_t* s, int n )
{
    wcsncpy (t, s, n);
}
```

See also

- [_strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l](#)
- [sizeof Operator](#)

Warning C6384

Article • 10/07/2022

Dividing sizeof a pointer by another value

This warning indicates that a size calculation might be incorrect. To calculate the number of elements in an array, you sometimes divide the size of the array by the size of the first element. However, when the array is actually a pointer, the result is typically different than intended.

Remarks

If the pointer is a function parameter and the size of the buffer wasn't passed, it isn't possible to calculate the maximum buffer available. When the pointer is allocated locally, the size used in the allocation should be used.

Code analysis name: DIVIDING_SIZEOF_POINTER

Example

The following code generates warning C6384:

```
C++  
  
#include <windows.h>  
#include <TCHAR.h>  
  
#define SIZE 15  
  
void f( )  
{  
    LPTSTR dest = new TCHAR[SIZE];  
    char src [SIZE] = "Hello, World!!";  
    if (dest)  
    {  
        _tcscpy(dest, src, sizeof dest / sizeof dest[0]);  
    }  
}
```

To correct this warning, pass the buffer size as shown in the following code:

```
C++  
  
#include <windows.h>  
#include <TCHAR.h>  
  
#define SIZE 15  
  
void f( )  
{  
    LPTSTR dest = new TCHAR[SIZE];  
    char src [SIZE] = "Hello, World!!";  
    if (dest)  
    {  
        _tcscpy(dest, src, SIZE);  
    }  
}
```

```
#include <windows.h>
#include <TCHAR.h>

#define SIZE 15

void f( )
{
    LPTSTR dest = new TCHAR[SIZE];
    char src [SIZE] = "Hello, World!!";
    if (dest)
    {
        _tcsncpy(dest, src, SIZE);
    }
}
```

To correct this warning using the safe string function `_tcsncpy_s`, use the following code:

C++

```
void f( )
{
    LPTSTR dest = new TCHAR[SIZE];
    char src [SIZE] = "Hello, World!!";
    if (dest)
    {
        _tcsncpy_s(dest, SIZE, src, SIZE);
    }
}
```

The use of `new` and `delete` has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

See also

[_mbsncpy_s](#), [_mbsncpy_s_l](#)
[sizeof Operator](#)

Warning C6385

Article • 10/07/2022

Invalid data: accessing *buffer-name*, the readable size is *size1* bytes, but *size2* bytes may be read: Lines: *x, y*

Remarks

The readable extent of the buffer might be smaller than the index used to read from it. Attempts to read data outside the valid range leads to buffer overrun.

Code analysis name: READ_OVERRUN

Example

The following code generates this warning:

```
C++  
  
void f(unsigned int i)  
{  
    char a[20];  
    char j;  
    if (i <= 20) // C6385  
    {  
        j = a[i];  
    }  
}
```

To correct this warning, use the following code:

```
C++  
  
void f(unsigned int i)  
{  
    char a[20];  
    char j;  
    if (i < 20) // Okay  
    {  
        j = a[i];  
    }  
}
```

See also

[Avoiding buffer overruns](#)

Warning C6386

Article • 10/07/2022

Buffer overrun: accessing '*buffer name*', the writable size is '*size1*' bytes, but '*size2*' bytes may be written: Lines: x, y

Remarks

This warning indicates that the writable extent of the specified buffer might be smaller than the index used to write to it. This defect can cause buffer overrun.

Code analysis name: `WRITE_OVERRUN`

Example

The following code generates both this warning and [C6201](#):

```
C++  
  
#define MAX 25  
  
void f ()  
{  
    char ar[MAX];  
    // code ...  
    ar[MAX] = '\0';  
}
```

To correct both warnings, use the following code:

```
C++  
  
#define MAX 25  
  
void f ()  
{  
    char a[MAX];  
    // code ...  
    a[MAX - 1] = '\0';  
}
```

See also

C6201

Warning C6387

Article • 11/07/2022

'argument' may be 'value': this does not adhere to the specification for the function
'function name': Lines: x, y

Remarks

This warning is raised if an annotated function parameter is being passed an unexpected value. For example, passing a potentially null value to a parameter that is marked with `_In_` annotation generates this warning.

Code analysis name: `INVALID_PARAM_VALUE_1`

Example

The following code generates this warning because a null parameter is passed to `f(char *)`:

```
C++  
  
#include <sal.h>  
  
_Post_ _Null_ char * g();  
  
void f(_In_ char *pch);  
  
void main()  
{  
    char *pCh = g();  
    f(pCh); // Warning C6387  
}
```

To correct this warning, use the following code:

```
C++  
  
#include <sal.h>  
  
_Post_ _Notnull_ char * g();  
  
void f(_In_ char *pch);  
  
void main()
```

```
{  
    char *pCh = g();  
    f(pCh);  
}
```

See also

[strlen](#), [wcslen](#), [_mbslen](#), [_mbslen_l](#), [_mbstrlen](#), [_mbstrlen_l](#)

Warning C6388

Article • 10/07/2022

'*argument*' may not be '*value*': this does not adhere to the specification for the function '*function-name*': Lines: *x, y*

Remarks

This warning indicates that an unexpected value is being used in the specified context. This warning is typically reported for values passed as arguments to a function that doesn't expect it.

Code analysis name: `INVALID_PARAM_VALUE_2`

Example

The following code generates warning C6388 because `DoSomething` expects a null value but a potentially non-null value might be passed:

```
C++  
  
// C6388_warning.cpp  
#include <string.h>  
#include <malloc.h>  
#include <sal.h>  
  
void DoSomething( _Pre_ _Null_ void* pReserved );  
  
void f()  
{  
    void* p = malloc( 10 );  
    DoSomething( p ); // Warning C6388  
    // code...  
    free(p);  
}
```

To correct this warning, use the following sample code:

```
C++  
  
// C6388_no_warning.cpp  
#include <string.h>  
#include <malloc.h>  
#include <sal.h>
```

```
void DoSomething( _Pre_ _Null_ void* pReserved );
void f()
{
    void* p = malloc( 10 );
    if (!p)
    {
        DoSomething( p );
    }
    else
    {
        // code...
        free(p);
    }
}
```

The use of `malloc` and `free` has many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Library (STL). These include `shared_ptr`, `unique_ptr`, and containers such as `vector`. For more information, see [Smart pointers](#) and [C++ Standard Library](#).

Warning C6389

Article • 10/07/2022

Move '*declaration*' to anonymous namespace or put a forward declaration in a common header included in this file.

Remarks

This check is intended to help reduce the visibility of certain symbols and to modularize the code. In multi-file C++ projects, each declaration should be either local to a C++ file (part of the anonymous namespace) or declared in a common header file that's included by multiple C++ files.

When this check flags a declaration, either it should be moved to an anonymous namespace or a forward declaration should be moved to a header file, depending on the scope of the symbol.

The rule is an experimental rule that must be explicitly enabled in a rule set file to work. For more information about rule sets, see [Use rule sets to group code analysis rules](#).

Code analysis name: `MARK_INTERNAL_OR_MISSING_COMMON_DECL`

Example

```
C++

// A.h
struct X;

// A.cpp
#include "A.h"

// Not flagged, declared in a header file.
struct X { int x; };

struct Y { double y; }; // warning: Move 'Y' to anonymous namespace or put a
                      // forward declaration in a common header included in this file.

void f(); // warning: Move 'f' to anonymous namespace or put a forward
          // declaration in a common header included in this file.
```

One way to resolve these issues is to move `struct Y` into an anonymous namespace, and move the declaration of `f` into a header:

C++

```
// A.h
struct X;
void f();

// A.cpp
#include "A.h"

// Not flagged, declared in a header file.
struct X { int x; };

namespace {
    struct Y { double y; };
} // anonymous namespace

// Not flagged, declared in a header file.
void f();
```

Warning C6390

Article • 10/07/2022

According to the C++ standard, the value of 'this' is never null; some compilers will optimize this check out

Remarks

While MSVC doesn't do such optimizations, some compilers optimize null checks for `this` out. Code that relies on null-valued `this` can trigger unexpected behavior when compiled with other compilers. This warning helps detect these portability problems.

Code analysis name: `NO_NONNULLCHECK_FOR_THIS`

Example

C++

```
struct X
{
    void m()
    {
        if(!this) // Warning: According to the C++ standard, the value of
        'this' is never null; some compilers will optimize this check out
        return;
    }
};
```

To solve this problem, rewrite the code to never call non-static member functions on null pointers.

Warning C6392

Article • 03/08/2024

This expression writes the value of the pointer to the stream. If this is intentional, add an explicit cast to 'void *'

This rule was added in Visual Studio 2022 17.8.

Remarks

C++ supports wide character streams such as `std::wstringstream`, and nonwide character streams such as `std::ostringstream`. Trying to print a wide string to a nonwide stream calls the `void*` overload of `operator<<`. This overload prints the address of the wide string instead of the value.

Code analysis name: `STREAM_OUTPUT_VOID_PTR`

Example

The following code snippet prints the value of the pointer to the standard output instead of the string "Pear":

```
C++

#include <iostream>

int main() {
    std::cout << L"Pear\n"; // Warning: C6392
}
```

There are multiple ways to fix this error. If printing the pointer value is unintended, use a nonwide string:

```
C++

#include <iostream>

int main() {
    std::cout << "Pear\n"; // No warning.
}
```

Alternatively, use a wide stream:

C++

```
#include <iostream>

int main() {
    std::wcout << L"Pear\n"; // No warning.
}
```

If the behavior is intentional, make the intention explicit and silence the warning by using an explicit cast:

C++

```
#include <iostream>

int main() {
    std::cout << static_cast<void*>(L"Pear\n"); // No warning.
}
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Warning C6393

Article • 11/30/2023

A lookup table of size 365 isn't sufficient to handle leap years

This rule was added in Visual Studio 2022 17.8.

Remarks

In the Gregorian calendar, every year exactly divisible by four is a leap year--except for years that are exactly divisible by 100. The centurial years are also leap years if they're exactly divisible by 400.

A leap year bug occurs when software doesn't account for this leap year logic, or uses flawed logic. This can affect reliability, availability, or even the security of the affected system.

Lookup tables of size 365 are often used to quickly find the month a given day corresponds to. However, it isn't correct because a leap year has 366 days.

Code analysis name: LEAP_YEAR_INVALID_DATE_KEYED_LOOKUP

Example

The following code creates a lookup table for the day of the year, assuming 365 days per year. However, this doesn't work if the year is a leap year:

C++

```
#include <vector>

void foo(int year)
{
    const std::vector<int> items(365); // C6393
    // Initialize items and use it...
}
```

To fix the problem, adjust the size of the lookup table as the table is created according to the result of appropriate leap year check:

C++

```
#include <vector>

void foo(int year)
{
    bool isLeapYear = year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    const std::vector<int> items(isLeapYear ? 366 : 365);
    // Initialize items and use it...
}
```

Heuristics

This rule is enforced by checking if a constant lookup table is sized for 365 elements. Violation of this rule causes a high confidence warning to be reported.

See also

[C6394](#)

[C26861](#)

[C26862](#)

[C26863](#)

[C26864](#)

Warning C6394

Article • 11/30/2023

A lookup table of size 365 isn't sufficient to handle leap years

This rule was added in Visual Studio 2022 17.8.

Remarks

In the Gregorian calendar, every year exactly divisible by four is a leap year--except for years that are exactly divisible by 100. The centurial years are also leap years if they're exactly divisible by 400.

A leap year bug occurs when software doesn't account for this leap year logic, or uses flawed logic. This can affect reliability, availability, or even the security of the affected system.

Lookup tables of size 365 are often used to quickly find the month a given day corresponds to, and so on. However, it isn't correct because a leap year has 366 days.

Code analysis name: LEAP_YEAR_INVALID_DATE_KEYED_LOOKUP_MUTABLE

Example

The following code creates a lookup table for the day of the year, but assumes there are 365 days per year. However, this produces the wrong result, or can cause an out-of-bounds access of the lookup table, if the year is a leap year:

C++

```
#include <vector>

void foo(int year)
{
    std::vector<int> items(365); // C6394
    // Initialize items and use it...
    // Another item may be added to the vector if year is a leap year, but
    this
    // rule doesn't check if that is the case.
}
```

To fix this problem, adjust the size of the lookup table as the table is created according to the result of a leap year check:

C++

```
#include <vector>

void foo(int year)
{
    bool isLeapYear = year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    const std::vector<int> items(isLeapYear ? 366 : 365);
    // Initialize items and use it...
}
```

Heuristics

This rule is enforced by checking if a lookup table has an initial size of 365 elements, but can be expanded to 366. However, it doesn't check if the table's size is adjusted through proper leap year check or not, and so is a low confidence warning.

See also

[C6393](#)

[C26861](#)

[C26862](#)

[C26863](#)

[C26864](#)

Warning C6395

Article • 10/24/2023

%variable% has unsequenced reads and/or writes before C++17; changing the language standard might change the behavior of the code.

Remarks

C++17 made the evaluation order of certain expressions stricter. MSVC complied, which changed the evaluation order for some expressions. Thus, changing the language version might change the observable behavior of the program. This check diagnoses some of the cases where the meaning of the code changes due to switching language versions.

Code analysis name: EVAL_ORDER_CHANGE

Example

C++

```
void foo(int* a, int i)
{
    a[++i] = i; // Warning: 'i' has unsequenced reads and/or writes before
    // C++17; changing the language standard might change the behavior of the code
}
```

To solve this problem, separate the side effects from the rest of the expression to make the evaluation order well defined:

C++

```
void foo(int* a, int i)
{
    ++i;
    a[i] = i; // No warning.
}
```

Warning C6396

Article • 02/15/2024

`sizeof('integerConstant')` always returns the size of the underlying integer type

Remarks

This warning indicates where an integral constant is used as a `sizeof` argument. Such expression always returns the size of the type of the constant. It's better to write `sizeof(type)` instead. This warning catches common typos in buffer offset calculations.

This check ignores character literals because `buffer_size += sizeof(UNICODE_NULL)` is a common idiom.

Example

C++

```
void f()
{
    int a = sizeof(5);           // C6396 reported here
}
```

To fix this issue, replace the integral constant with its type:

C++

```
void f()
{
    int a = sizeof(int);        // no C6396 reported here
}
```

Warning C6397

Article • 02/15/2024

The address-of operator cannot return `null` pointer in well-defined code

Remarks

The address-of operator returns the address of its operand. This value should never be compared to `nullptr`:

- The address-of a field can only be `nullptr` if the base pointer was `nullptr` and the field is at the zero offset (`&p->field == nullptr` implies `p == nullptr`). In this case, the expression should be simplified.
- In any other cases, the value of the unary `&` operator can't be `nullptr` unless there's undefined behavior in the code (`&v == nullptr` always evaluates to false).

Example

C++

```
bool isNull(int *a)
{
    return &a == nullptr; // C6397 reported here.
}
```

To fix this issue, double check if the use of unary `&` was intentional:

C++

```
bool isNull(int *a)
{
    return a == nullptr; // no C6397 reported here.
}
```

See also

[C6398](#)

Warning C6398

Article • 02/15/2024

The address-of a field cannot be `null` in well-defined code

Remarks

The address-of operator returns the address of its operand. This value should never be compared to `nullptr`:

- The address-of a field can only be `nullptr` if the base pointer was `nullptr` and the field is at the zero offset (`&p->field == nullptr` implies `p == nullptr`). In this case, the expression should be simplified.
- In any other cases, the value of the unary `&` operator can't be `nullptr` unless there's undefined behavior in the code (`&v == nullptr` always evaluates to false).

Example

C++

```
struct A { int* x; };

bool hasNullField(A *a)
{
    return &a->x == nullptr; // C6398 reported here.
}
```

To fix this issue, double check if the use of unary `&` was intentional:

C++

```
struct A { int* x; };

bool hasNullField(A *a)
{
    return a->x == nullptr; // no C6398 reported here.
}
```

See also

[C6397](#)

Warning C6400

Article • 10/07/2022

Using '*function name*' to perform a case-insensitive compare to constant string '*string name*'. Yields unexpected results in non-English locales

Remarks

This warning indicates that a case-insensitive comparison to a constant string is being done in a locale-dependent way. It appears that a locale-independent comparison was intended.

The typical consequence of this defect is incorrect behavior in non-English speaking locales. For example, in Turkish, ".gif" won't match ".GIF"; in Vietnamese, "LookUp" won't match "LOOKUP".

String comparisons should typically be performed with the `CompareString` function. To perform a locale-independent comparison on Windows XP, the first parameter should be the constant `LOCALE_INVARIANT`.

Code analysis name: `LOCALE_DEPENDENT_CONSTANT_STRING_COMPARISON`

Example

The following code generates this warning:

C++

```
#include <windows.h>
int f(char *ext)
{
    // code...
    return (lstrcmpi(ext, TEXT("gif")) == 0);
}
```

To correct this warning, perform a locale-independent test for whether `char *ext` matches "gif" ignoring upper/lower case differences, use the following code:

C++

```
#include <windows.h>
int f(char *ext)
{
```

```
// code...
return (CompareString(
    LOCALE_INVARIANT,
    NORM_IGNORECASE,
    ext,
    -1,
    TEXT ("gif"),
    -1) == CSTR_EQUAL);
}
```

See also

[CompareString](#)

Warning C6401

Article • 10/07/2022

Using '*function name*' in a default locale to perform a case-insensitive compare to constant string '*string name*'. Yields unexpected results in non-English locales

Remarks

This warning indicates that a case-insensitive comparison to a constant string is being done when specifying the default locale. Usually, a locale-independent comparison was intended.

The typical consequence of this defect is incorrect behavior in non-English speaking locales. For example, in Turkish, ".gif" won't match ".GIF"; in Vietnamese, "LookUp" won't match "LOOKUP".

The `CompareString` function takes a locale as an argument; however, passing in a default locale, for example, the constant `LOCALE_USER_DEFAULT`, will cause different behaviors in different locales, depending on the user's default. Usually, case-insensitive comparisons against a constant string should be performed in a locale-independent comparison.

To perform a locale-independent comparison using `CompareString` on Windows XP, the first parameter should be the constant `LOCALE_INVARIANT`; for example, to perform a locale-independent test for whether `pString` matches `file1.gif` ignoring upper/lower case differences, use a call such as:

C++

```
CompareString(LOCALE_INVARIANT,  
              NORM_IGNORECASE,  
              pString,  
              -1,  
              TEXT("file1.gif"),  
              -1) == CSTR_EQUAL
```

Code analysis name: `DEFAULT_LOCALE_CONSTANT_STRING_COMPARISON`

Example

The following code generates this warning:

C++

```
include <windows.h>

int fd(char *ext)
{
    return (CompareString(LOCALE_USER_DEFAULT,
                          NORM_IGNORECASE,
                          ext,
                          -1,
                          TEXT("gif"),
                          -1) == 2);
}
```

To correct this warning, use the following code:

C++

```
include <windows.h>
int f(char *ext)
{
    return (CompareString(LOCALE_INVARIANT,
                          NORM_IGNORECASE,
                          ext,
                          -1,
                          TEXT("gif"),
                          -1) == 2);
}
```

See also

[CompareString](#)

Warning C6411

Article • 10/07/2022

Potentially reading invalid data from '*buffer*'.

Remarks

This warning indicates that the value of the index that is used to read from the buffer can exceed the readable size of the buffer. The code analysis tool may report this warning in error. The error may occur when it can't reduce a complex expression that represents the buffer size, or the index used to access the buffer.

Code analysis name: POTENTIAL_READ_OVERRUN

Example

The following code generates this warning.

C++

```
char *a = new char[strlen(InputParam)];  
delete[] a;  
a[10];
```

The following code corrects this error.

C++

```
int i = strlen(InputParam);  
char *a = new char[i];  
if (i > 10) a[10];  
delete[] a;
```

Warning C6412

Article • 10/07/2022

Potential buffer overrun while writing to buffer. The writable size is *write_size* bytes, but *write_index* bytes may be written.

Remarks

This warning indicates that the value of the index that's used to write to the buffer can exceed the writeable size of the buffer.

The code analysis tool may report this warning in error. It reports this warning when it can't reduce a complex expression that represents the buffer size, or the index used to access the buffer.

Code analysis name: POTENTIAL_WRITE_OVERRUN

Example

The following code generates this warning.

C++

```
char *a = new char[strlen(InputParam)];  
a[10] = 1;  
delete[] a;
```

The following code corrects this error.

C++

```
int i = strlen(InputParam);  
char *a = new char[i];  
if (i > 10) a[10] = 1;  
delete[] a;
```

Warning C6500

Article • 10/07/2022

Invalid annotation: value for '*name*' property is invalid

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that a property value used in the annotation isn't valid. For example, it can occur if an incorrect level of dereference is used in the `Deref` property, or if you use a constant value that is larger than `size_t` for properties like `ElementSize`.

Code analysis name: `INVALID_ATTRIBUTE_PROPERTY`

Example

The following code generates this warning because an incorrect level of dereference is used in the `Pre` condition:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre( Deref=2, Access=SA_Read )] char buffer[] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;

void f( [Pre( Deref=2, Access=Read )] char buffer[] );
```

To correct this warning, specify the correct level of dereference, as shown in the following sample code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre( Deref=1, Access=SA_Read )] char buffer[] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;

void f( [Pre( Deref=1, Access=Read )] char buffer[] );
```

This warning is generated for both Pre and Post conditions.

Warning C6501

Article • 10/07/2022

Annotation conflict: '*name*' property conflicts with previously specified property

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates the presence of conflicting properties in the annotation. The warning typically occurs when multiple properties that serve similar purpose are used to annotate a parameter or return value. To correct the warning, you must choose the property that best addresses your need.

Code analysis name: `CONFLICTING_ATTRIBUTE_PROPERTY_VALUES`

Example

The following code generates this warning because both `ValidElementsConst` and `ValidBytesConst` provide a mechanism to allow valid data to be read:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void fd([SA_Pre(ValidElementsConst =4, ValidBytesConst =4)] char pch[]);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pre(ValidElementsConst=4, ValidBytesConst=4 )] char pch[ ] );
```

To correct this warning, use the most appropriate property, as shown in the following code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre(ValidElementsConst=4)] char pch[] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pre(ValidElementsConst=4)] char pch[] );
```

Warning C6503

Article • 10/07/2022

Invalid annotation: references and arrays may not be marked `Null=Yes` or `Null=Maybe`

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that `Null` property is incorrectly used on a reference or array type. A reference or array type holds the address of an object and must point to a valid object. Because reference and array types can't be null, you must correct the error by either removing the `Null` property or by setting the `Null` property value to `No`.

Code analysis name: `REFERENCES_CANT_BE_NULL`

Example

The following code generates this warning:

C++

```
// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
class Point
{
public:
    // members
};

void f([Pre(Null=Yes)] Point& pt);
```

To correct this warning, set the Null property to No as shown in the following code:

C++

```
// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;

class Point
{
public:
    // members
};

void f([Pre(Null=No)] Point& pt);
```

Warning C6504

Article • 10/07/2022

Invalid annotation: property may only be used on values of pointer, pointer-to-member, or array type

This warning indicates the use of a pointer-specific SAL annotation on a non-pointer data type.

Remarks

For more information about what data types are supported by properties, see [Annotation Properties](#).

Code analysis name: `NULL_ON_NON_POINTER`

Example

The following code generates warning C6504. This issue stems from the use of the pointer-specific `_Maybenull_` and `_Notnull_` on reference `pt`.

C++

```
class Point
{
    public:
        // members
};

void f(_Pre_ _Maybenull_ Point& pt)
{
    // code ...
}

void g(_Pre_ _Notnull_ Point& pt)
{
    // code ...
}
```

To correct this warning, remove the annotation if it doesn't make sense. You could also change to an annotation that's applicable to the type used, or change the type to match the annotation. The following code remediates this warning by changing the first

instance of `pt` to a pointer and by removing the second annotation to match the reference type.

C++

```
class Point
{
    public:
        // members
};

void f(_Pre_ _Maybenull_ Point* pt)
{
    // code ...
}

void g(Point& pt)
{
    // code ...
}
```

See Also

[Annotation Properties](#)

Warning C6505

Article • 10/07/2022

Invalid annotation: MustCheck property may not be used on values of void type

Remarks

This warning indicated that MustCheck property was used on a void data type. You can't use `MustCheck` property on `void` type. Either remove the `MustCheck` property or use another data type.

Code analysis name: `MUSTCHECK_ON_VOID`

Example

The following code generates this warning:

```
C++  
  
#include <sal.h>  
_Must_inspect_result_ void f()  
{  
    //Code ...  
}
```

To correct this warning, use the following code:

```
C++  
  
#include <sal.h>  
_Must_inspect_result_ char* f()  
{  
    char *str = "Hello World";  
    //Code ...  
    return str;  
}
```

See also

[C6516](#)

Warning C6506

Article • 10/07/2022

Invalid annotation: '*name*' property may only be used on values of pointer or array types

Remarks

This warning indicates that a property is used on a type other than pointer or array types. The Access, Tainted, and Valid properties can be used on all data types. Other properties, such as ValidBytesConst, ValidElementsConst, ElementSize, and NullTerminated support pointer, pointer to members, or array types. For a complete list of properties and the supported data types, see [Using SAL Annotations to reduce code defects](#).

Code analysis name: `BUFFER_SIZE_ON_NON_POINTER_OR_ARRAY`

Example

The following code generates this warning:

C++

```
#include<sal.h>
void f(_Out_ char c)
{
    c = 'd';
}
```

To correct this warning, use a pointer or an array type, as shown in the following sample code:

C++

```
#include<sal.h>
void f(_Out_ char *c)
{
    *c = 'd';
}
```

See also

C6516

Warning C6508

Article • 10/07/2022

Invalid annotation: write access is not allowed on const values

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that the Access property specified on a const parameter implies that it can be written to. For constant values, Access=Read is the only valid setting.

Code analysis name: `WRITE_ACCESS_ON_CONST`

Example

The following code generates this warning:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void fD ([SA_Pre(Deref=1,Access=SA_Write)]const char *pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(Deref=1,Access=Write)]const char *pc);
```

To correct this warning, use the following code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(Deref=1,Access=SA_Read)]const char *pc);
```

```
// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(Deref=1,Access=Read)]const char *pc);
```

Warning C6509

Article • 10/07/2022

Invalid annotation: 'return' cannot be referenced from a precondition

Remarks

This warning indicates that the `return` keyword can't be used in a precondition. The `return` keyword is used to terminate the execution of a function and return control to the calling function.

Code analysis name: `RETURN_USED_ON_PRECONDITION`

Example

The following code generates this warning because `return` is used in a precondition:

```
C++  
  
#include <sal.h>  
  
int f (_In_reads_(return) char *pc)  
{  
    // code ...  
    return 1;  
}
```

To correct this warning, use the following code:

```
C++  
  
#include <sal.h>  
  
int f (_In_reads_(i) char *pc, int i)  
{  
    // code ...  
    return 1;  
}
```

Warning C6510

Article • 10/07/2022

Invalid annotation: 'NullTerminated' property may only be used on buffers whose elements are of integral or pointer type: Function "function" 'parameter'.

Remarks

This warning indicates an incorrect use of the **NullTerminated** property (the ones ending in '`_z`'). You can only use this type of property on pointer or array types.

Code analysis name: `NULLTERMINATED_ON_NON_POINTER`

Example

The following code generates this warning:

```
C++  
  
#include <sal.h>  
  
void f(_In_z_ char x)  
{  
    // code ...  
}
```

To correct this warning, use the following code:

```
C++  
  
#include <sal.h>  
  
void f(_In_z_ char * x)  
{  
    // code ...  
}
```

See also

[C6516](#)

Warning C6511

Article • 10/07/2022

Invalid annotation: MustCheck property must be Yes or No

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates an invalid value for `MustCheck` property was specified. The only valid values for this property are: Yes and No.

Code analysis name: `MUSTCHECK_MAYBE`

Example

The following code generates this warning:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
[returnvalue:SA_Post(MustCheck=SA_Maybe)] int f();

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
[returnvalue:Post(MustCheck=Maybe)] int f();
```

To correct this warning, a valid value for MustCheck property is used in the following code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
```

```
[returnvalue:SA_Post(MustCheck=SA_Yes)] int f();  
  
// C++  
#include <CodeAnalysis\SourceAnnotations.h>  
using namespace vc_attributes;  
[returnvalue:Post(MustCheck=Yes)] int f();
```

See also

- [Using SAL Annotations to reduce code defects](#)
- [C6516](#)

Warning C6513

Article • 10/07/2022

Invalid annotation: ElementSizeConst requires additional size properties

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that `ElementSizeConst` requires other properties that are missing from the annotation. Specifying `ElementSizeConst` alone doesn't provide any benefit to the analysis process. In addition to specifying `ElementSize`, other properties such as `ValidElementsConst` or `WritableDatabaseConst` must also be specified.

Code analysis name: `ELEMENT_SIZE_WITHOUT_BUFFER_SIZE`

Example

The following code generates this warning:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ElementSizeConst=4)] void* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(ElementSizeConst=4)] void* pc);
```

To correct this warning, use the following code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ElementSizeConst=4, ValidElementsConst=2)] void* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(ElementSizeConst=4, ValidElementsConst=2)] void* pc);
```

Incorrect use of ElementSize property also generates this warning.

See also

[Using SAL Annotations to reduce code defects](#)

Warning C6514

Article • 10/07/2022

Invalid annotation: value of the '*name*' property exceeds the size of the array

Remarks

This warning indicates that a property value exceeds the size of the array specified in the parameter being annotated. This warning occurs when the value specified for the annotation property is greater than the actual length of the array being passed.

Code analysis name: `BUFFER_SIZE_EXCEEDS_ARRAY_SIZE`

Example

The following code generates this warning because the size of the array is 6 but the `ValidElementsConst` property value is 8:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre(Deref=1, ValidElementsConst=8)] char(*matrix) [6] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pre(Deref=1, ValidElementsConst=8)] char(*matrix) [6] );
```

To correct this warning, make sure the size of specified in `ValidElementsConst` is less than or equal to the size of the array, as shown in the following sample code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre(Deref=1, ValidElementsConst=6)] char(*matrix) [6] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pre(Deref=1, ValidElementsConst=6)] char(*matrix) [6] );
```

Warning C6515

Article • 10/07/2022

Invalid annotation: '*name*' property may only be used on values of pointer type

Remarks

This warning indicates that a property for use on pointers was applied to a non-pointer type. For a list of annotation properties, see [Using SAL Annotations to reduce code defects](#).

Code analysis name: `BUFFER_SIZE_ON_NON_POINTER`

Example

The following code generates this warning:

```
C++  
  
#include <sal.h>  
  
void f(_Readable_bytes_(c) char pc, size_t c)  
{  
    // code ...  
}
```

To correct this warning, use the following code:

```
C++  
  
#include <sal.h>  
  
void f(_Readable_bytes_(c) char * pc, size_t c)  
{  
    // code ...  
}
```

See also

[C6516](#)

Warning C6516

Article • 10/07/2022

Invalid annotation: no properties specified for 'name' attribute

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that either no property was specified in the attribute or the property that was specified is invalid; therefore, the attribute can't be considered complete.

Code analysis name: NO_PROPERTIES_ON_ATTRIBUTE

Example

The following code generates this warning because Deref=1 only specifies the level of indirection, but this information alone doesn't help the analysis tool:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pref(Deref=1)] char* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pref(Deref=1)] char* pc);
```

To correct this warning, another property, such as Access, is required to indicate to the analysis tool what must be enforced on the de-referenced items. The following code corrects this warning:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pref(Deref=1, Access=SA_Read)] char* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pref(Deref=1, Access=Read)] char* pc);
```

Warning C6517

Article • 10/07/2022

Invalid annotation: 'SAL_readableTo' property may not be specified on buffers that are not readable: '*Parameter*'.

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that `SAL_readableTo` property doesn't have the required read access. You can't use this property to annotate a parameter without providing read access.

Code analysis name: `VALID_SIZE_ON_NON_READABLE_BUFFER`

Example

The following code generates this warning because read access isn't granted on the buffer:

C++

```
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre( ValidBytesConst=10 )][Pre( Deref=1, Access=Write )] char*
buffer );
```

To correct this warning, grant read access as shown in the following code:

C++

```
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
```

```
void f([Pre( ValidBytesConst=10 )][Pre( Deref=1, Access=Read)] char* buffer );
```

See also

[Using SAL Annotations to reduce code defects](#)

Warning C6518

Article • 10/07/2022

Invalid annotation: 'SAL_writableTo' property may not be specified as a precondition on buffers that are not writable: '*Parameter*'.

Remarks

This warning indicates that a conflict exists between a `SAL_writableTo` property value and a writable property. The warning ordinarily indicates that a writable property doesn't have write access to the parameter being annotated.

Code analysis name: `WRITABLE_SIZE_ON_NON_WRITABLE_BUFFER`

Example

The following code generates this warning because the `_out_` annotation compiles to include a `SAL_writableTo` property, which doesn't allow write access:

C++

```
#include <sal.h>
void f(_Out_ const char* pc)
{
    //code that can't write to *pc ...
}
```

To correct this warning, use the following code:

C++

```
#include <sal.h>
void f(_Out_ char* pc)
{
    pc = "Hello World";
    //code ...
}
```

Warning C6522

Article • 10/07/2022

Invalid size specification: expression must be of integral type: annotation '*annotation*' on function '*function*' '*parameter*'

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that an integral type was expected, but an incorrect data type was used. You can use annotation properties that accept the size of a parameter in terms of another parameter, but you must use correct data type. For a list of annotation properties, see [Using SAL Annotations to reduce code defects](#).

Code analysis name: `INVALID_SIZE_STRING_TYPE`

Example

The following code generates this warning:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ValidBytes="c")]) char *pc, double c;

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(ValidBytes="c")]) char *pc, double c;
```

To correct this warning, use `size_t` for the `ValidBytesParam` parameter data type, as shown in the following sample code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ValidBytes="c")] char *pc, size_t c);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(ValidBytes="c")] char *pc, size_t c);
```

Warning C6525

Article • 10/07/2022

Invalid size specification: property value may not be valid

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that the property value used to specify the size isn't valid. The warning occurs if the size parameter is annotated using `Valid=No`.

Code analysis name: `INVALID_SIZE_STRING_UNREACHABLE_LOCATION`

Example

The following code generates this warning because the `ValidElements` property uses a size parameter that is marked not valid:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(ValidElements="*count")] char * px,
[SA_Pre(Valid=SA_No)]size_t *count);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(ValidElements="*count")] char * px, [Pre(Valid=No)]size_t
*count);
```

To correct this warning, specify a valid size parameter as shown in the following code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(ValidElements=*count)] char * px,
[SA_Pre(Valid=SA_Yes)]size_t *count);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(ValidElements=*count)] char * px, [Pre(Valid=Yes)]size_t
*count);
```

See also

[Using SAL Annotations to reduce code defects](#)

Warning C6527

Article • 10/07/2022

Invalid annotation: NeedsRelease property may not be used on values of void type

Warning C6530

Article • 10/07/2022

Unrecognized format string style 'name'

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Remarks

This warning indicates that the `FormatString` property is using a value other than `scanf` or `printf`. To correct this warning, review your code and use a valid value for the `Style` property.

Code analysis name: `UNRECOGNIZED_FORMAT_STRING_STYLE`

Example

The following code generates this warning because of a typo in the `Style` property:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_FormatString(Style="printfd")] char *px);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([FormatString(Style="printfd")] char *px);
```

To correct this warning, use a valid Style as shown in the following code:

C++

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
```

```
void f([SA_FormatString(Style="printf")] char *px);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([FormatString(Style="printf")] char *px);
```

Warning C6540

Article • 10/07/2022

The use of attribute annotations on this function will invalidate all of its existing
`__declspec` annotations

Warning C6551

Article • 10/07/2022

Invalid size specification: expression not parseable

Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Warning C6552

Article • 10/07/2022

Invalid `Deref=` or `Notref=`: expression not parseable

ⓘ Note

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

Warning C6701

Article • 10/07/2022

The value is not a valid Yes/No/Maybe value: '*string*'

This warning is reported when there's an error in the annotations.

Warning C6702

Article • 10/07/2022

The value is not a string value: '*string*'

This warning is reported when there's an error in the annotations.

Warning C6703

Article • 10/07/2022

The value is not a number: '*string*'

This warning is reported when there's an error in the annotations.

Warning C6704

Article • 10/07/2022

Unexpected Annotation Expression Error: '*annotation*' ['*why*']

This warning is reported when there's an error in the annotations.

Warning C6705

Article • 10/07/2022

Annotation error expected <expected_number> arguments for annotation '*parameter*' found <actual_number>.

This warning is reported when there's an error in the annotations.

Warning C6706

Article • 10/07/2022

Unexpected Annotation Error for annotation '*annotation*': 'why'

This warning is reported when there's an error in the annotations.

Warning C6707

Article • 10/07/2022

Unexpected Model Error: 'why'

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Warning C6993

Article • 10/07/2022

Code analysis ignores OpenMP constructs; analyzing single-threaded code

This warning indicates that the Code Analyzer has encountered Open MP pragmas that it can't analyze.

Warning C6995

Article • 10/07/2022

Failed to save XML Log file

This warning indicates that the Code Analysis tool can't create the defect log, which is the output of the code analysis.

This error might indicate a disk error or indicate that you don't have permission to create a file in the specified directory.

Warning C6997

Article • 10/07/2022

Annotations at this location are meaningless and will be ignored.

Remarks

Annotations can't be applied to `extern "C" {...}`. Apply the annotations to a specific object.

Code analysis name: `IGNORED_ANNOTATIONS`

See also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C26100

Article • 10/07/2022

Race condition. Variable 'var' should be protected by lock 'lock'.

Remarks

The `_Guarded_by_` annotation in the code specifies the lock to use to guard a shared variable. Warning C26100 is generated when the guard contract is violated.

Code analysis name: `RACE_CONDITION`

Examples

The following example generates warning C26100 because there's a violation of the `_Guarded_by_` contract.

C++

```
CRITICAL_SECTION gCS;

_Guarded_by_(gCS) int gData;

typedef struct _DATA {
    _Guarded_by_(cs) int data;
    CRITICAL_SECTION cs;
} DATA;

void Safe(DATA* p) {
    EnterCriticalSection(&p->cs);
    p->data = 1; // OK
    LeaveCriticalSection(&p->cs);
    EnterCriticalSection(&gCS);
    gData = 1; // OK
    LeaveCriticalSection(&gCS);
}

void Unsafe(DATA* p) {
    EnterCriticalSection(&p->cs);
    gData = 1; // Warning C26100 (wrong lock)
    LeaveCriticalSection(&p->cs);
}
```

The contract violation occurs because an incorrect lock is used in the function `Unsafe`. In this case, `gCS` is the correct lock to use.

Occasionally a shared variable only has to be guarded for write access but not for read access. In that case, use the `_Write_guarded_by_` annotation, as shown in the following example.

C++

```
CRITICAL_SECTION gCS;

_Guarded_by_(gCS) int gData;

typedef struct _DATA2 {
    _Write_guarded_by_(cs) int data;
    CRITICAL_SECTION cs;
} DATA2;

int Safe2(DATA2* p) {
    // OK: read does not have to be guarded
    int result = p->data;
    return result;
}

void Unsafe2(DATA2* p) {
    EnterCriticalSection(&gCS);
    // Warning C26100 (write has to be guarded by p->cs)
    p->data = 1;
    LeaveCriticalSection(&gCS);
}
```

This example also generates warning C26100 because it uses an incorrect lock in the function `Unsafe2`.

Warning C26101

Article • 10/07/2022

Failing to use interlocked operation properly for variable 'var'.

Windows APIs offer various interlocked operations. Annotation `_Interlocked_` specifies that a variable should only be accessed through an interlocked operation. Warning C26101 is issued when a variable access isn't consistent with the `_Interlocked_` annotation.

Example

The following example generates warning C26101 because there's a violation of the `_Interlocked_` contract.

C++

```
CRITICAL_SECTION cs;
typedef struct _DATA
{
    _Interlocked_ LONG data;
} DATA;

void Safe(DATA* p)
{
    InterlockedIncrement(&p->data); // OK
}

void Unsafe(DATA* p)
{
    p->data += 1; // Warning C26101
    EnterCriticalSection(&cs);
    LeaveCriticalSection(&cs);
}
```

Warning C26105

Article • 10/07/2022

Lock order violation. Acquiring lock '*lock*' with level '*level*' causes order inversion.

Concurrency SAL supports *lock levels*. To declare a lock level, which is denoted by a string literal without double quotes, use `_Create_lock_level_`. You can impose an order of acquisition between two lock levels by using the annotation

`_Set_lock_level_order_(A,B)`, which states that locks that have level `A` must be acquired before locks that have level `B`. To establish a lock order hierarchy (a partial order among lock levels), use multiple `_Set_lock_level_order_` annotations. To associate a lock with a lock level, use the `_Set_lock_level_` annotation when you declare the lock. Warning C26105 is issued when a lock ordering violation is detected.

Example

The following example generates warning C26105 because there's a lock order inversion in the function `OrderInversion`.

C++

```
_Create_lock_level_(MutexLockLevel);
_Create_lock_level_(TunnelLockLevel);
_Create_lock_level_(ChannelLockLevel);
_Lock_level_order_(MutexLockLevel, TunnelLockLevel);
_Lock_level_order_(TunnelLockLevel, ChannelLockLevel);
_Has_lock_level_(MutexLockLevel) HANDLE gMutex;

struct Tunnel
{
    _Has_lock_level_(TunnelLockLevel) CRITICAL_SECTION cs;
};

struct Channel
{
    _Has_lock_level_(ChannelLockLevel) CRITICAL_SECTION cs;
};

void OrderInversion(Channel* pChannel, Tunnel* pTunnel)
{
    EnterCriticalSection(&pChannel->cs);
    // Warning C26105
    WaitForSingleObject(gMutex, INFINITE);
    EnterCriticalSection(&pTunnel->cs);
    LeaveCriticalSection(&pTunnel->cs);
```

```
    LeaveCriticalSection(&pChannel->cs);  
}
```

Warning C26110

Article • 10/07/2022

Caller failing to hold lock '*lock*' before calling function '*func*'.

When a lock is required, make sure to clarify whether the function itself, or its caller, should acquire the lock. Warning C26110 is issued when there's a violation of the `_Requires_lock_held_` annotation, or other lock-related annotations. For more information, see [Annotating Locking Behavior](#)

Example

In the following example, warning C26110 is generated because the annotation `_Requires_lock_held_` on function `LockRequired` states that the caller of `LockRequired` must acquire the lock before it calls `LockRequired`. Without this annotation, `LockRequired` has to acquire the lock before it accesses any shared data protected by the lock.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    int d;
} DATA;

__Requires_lock_held_(p->cs)

void LockRequired(DATA* p)
{
    p->d = 0;
}

void LockNotHeld(DATA* p)
{
    LockRequired(p); // Warning C26110
}
```

Warning C26111

Article • 10/07/2022

Caller failing to release lock '*lock*' before calling function '*func*'.

The annotation `_Requires_lock_not_held_` imposes a precondition that the lock count for the specified lock can't be greater than zero when the function is called. Warning C26111 is issued when a function fails to release the lock before it calls another function.

Example

The following example generates warning C26111 because the `_Requires_lock_not_held_` precondition is violated by the call to `DoNotLock` within the locked section.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    int d;
} DATA;

__Requires_lock_not_held_(p->cs)

void DoNotLock(DATA* p)
{
    EnterCriticalSection(&p->cs);
    p->d = 0;
    LeaveCriticalSection(&p->cs);
}

void LockedFunction(DATA* p)
{
    EnterCriticalSection(&p->cs);
    DoNotLock(p); // Warning C26111
    LeaveCriticalSection(&p->cs);
}
```

Warning C26112

Article • 10/07/2022

Caller cannot hold any lock before calling '*func*'.

The annotation `_Requires_no_locks_held_` imposes a precondition that the caller must not hold any lock while it calls the function. Warning C26112 is issued when a function fails to release all locks before it calls another function.

Example

The following example generates warning C26112 because the `_Requires_no_locks_held_` precondition is violated by the call to `NoLocksAllowed` within the locked section.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

__Requires_no_locks_held__

void NoLocksAllowed(DATA* p)
{
    // Lock sensitive operations here
}

void LocksHeldFunction(DATA* p)
{
    EnterCriticalSection(&p->cs);
    NoLocksAllowed(p); // Warning C26112
    LeaveCriticalSection(&p->cs);
}
```

Warning C26115

Article • 10/07/2022

Failing to release lock '*lock*' in function '*func*'.

Enforcement of syntactically scoped lock *acquire* and lock *release* pairs in C/C++ programs isn't performed by the language. A function may introduce a locking side effect by making an observable modification to the concurrency state. For example, a lock wrapper function increments the number of lock acquisitions, or lock count, for a given lock.

You can annotate a function that has a side effect from a lock acquire or lock release by using `_Acquires_lock_` or `_Releases_lock_`, respectively. Without such annotations, a function is expected not to change any lock count after it returns. If acquires and releases aren't balanced, they're considered to be *orphaned*. Warning C26115 is issued when a function introduces an orphaned lock.

Example

The following example generates warning C26115 because there's an orphaned lock in a function that isn't annotated with `_Acquires_lock_`.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

void FailToReleaseLock(int flag, DATA* p)
{
    EnterCriticalSection(&p->cs);

    if (flag)
        return; // Warning C26115

    LeaveCriticalSection(&p->cs);
}
```

Warning C26116

Article • 10/07/2022

Failing to acquire or to hold lock '*lock*' in '*func*'.

Enforcement of syntactically scoped lock *acquire* and lock *release* pairs in C/C++ programs isn't performed by the language. A function may introduce a locking side effect by making an observable modification to the concurrency state. For example, a lock wrapper function increments the number of lock acquisitions, or lock count, for a given lock. You can annotate a function that has a side effect from a lock acquire or lock release by using `_Acquires_lock_` or `_Requires_lock_held`, respectively. Without such annotations, a function is expected not to change any lock count after it returns. If acquires and releases aren't balanced, they're considered to be *orphaned*. Warning C26116 is issued when a function has been annotated with `_Acquires_lock_`, but it doesn't acquire a lock, or when a function is annotated with `_Requires_lock_held` and releases the lock.

Example

The following example generates warning C26116 because the function `DoesNotLock` was annotated with `_Acquires_lock_` but doesn't acquire it. The function `DoesNotHoldLock` generates the warning because it's annotated with `_Requires_lock_held` and doesn't hold it.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

__Acquires_lock_(p->cs) void DoesLock(DATA* p)
{
    EnterCriticalSection(&p->cs); // OK
}

__Acquires_lock_(p->cs) void DoesNotLock(DATA* p)
{
    // Warning C26116
}

__Requires_lock_held_(p->cs) void DoesNotHoldLock(DATA* p)
{
```

```
    LeaveCriticalSection(&p->cs); // Warning C26116  
}
```

See also

- [C26115](#)

Warning C26117

Article • 11/07/2022

Releasing unheld lock '*lock*' in function '*func*'.

Enforcement of syntactically scoped lock *acquire* and lock *release* pairs in C/C++ programs isn't performed by the language. A function may introduce a locking side effect by making an observable modification to the concurrency state. For example, a lock wrapper function increments the number of lock acquisitions, or lock count, for a given lock. You can annotate a function that has a side effect from a lock acquire or lock release by using `_Acquires_lock_` or `_Releases_lock_`, respectively. Without such annotations, a function is expected not to change any lock count after it returns. If acquires and releases aren't balanced, they're considered to be *orphaned*. Warning C26117 is issued when a function that hasn't been annotated with `_Releases_lock_` releases a lock that it doesn't hold, because the function must own the lock before it releases it.

Examples

The following example generates warning C26117 because the function

`ReleaseUnheldLock` releases a lock that it doesn't necessarily hold—the state of `flag` is ambiguous—and there's no annotation that specifies that it should.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

int flag;

void ReleaseUnheldLock(DATA* p)
{
    if (flag)
        EnterCriticalSection(&p->cs);
    // code ...
    LeaveCriticalSection(&p->cs);
}
```

The following code fixes the problem by guaranteeing that the released lock is also acquired under the same conditions.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

int flag;

void ReleaseUnheldLock(DATA* p)
{
    if (flag)
    {
        EnterCriticalSection(&p->cs);
        // code ...
        LeaveCriticalSection(&p->cs);
    }
}
```

See also

- [C26115](#)

Warning C26130

Article • 10/07/2022

Missing annotation `_Requires_lock_held_('lock')` or `_No_competing_thread_` at function '*func*'. Otherwise it could be a race condition. Variable '*var*' should be protected by lock '*lock*'.

Warning C26130 is issued when the analyzer detects a potential race condition but infers that the function is likely to be run in a single threaded mode. For example, when the function is in the initialization stage, based on certain heuristics.

Examples

In the following example, warning C26130 is generated because a `_Guarded_by_` member is being modified without a lock.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} DATA;

void Init(DATA* p)
{
    p->data = 0; // Warning C26130
}
```

If the previous code is guaranteed to operate in single-threaded mode, annotate the function by using `_No_competing_thread_`, as shown in the following example.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} DATA;

_No_competing_thread_ void Init(DATA* p)
{
    p->data = 0; // Warning C26130 will be resolved
}
```

Alternatively, you can annotate a code fragment by using `_No_competing_thread_begin_` and `_No_competing_thread_end_`, as follows.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} DATA;

void Init(DATA* p)
{
    _No_competing_thread_begin_
    p->data = 0; // Warning C26130 will be resolved
    _No_competing_thread_end_
}
```

Warning C26135

Article • 10/07/2022

Missing annotation '*annotation*' at function '*func*'.

Warning C26135 is issued when the analyzer infers that a function is a lock wrapper function that has a "lock acquire" or "lock release" side effect. If the code isn't intended to be a wrapper function, then either the lock is leaking (if it's being acquired), or it's being released incorrectly (if the lock is being released).

Examples

The following example generates warning C26135 because an appropriate side effect annotation is missing.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

void MyEnter(DATA* p)
{
    // Warning C26135:
    // Missing side effect annotation _Acquires_lock_(&p->cs)
    EnterCriticalSection(&p->cs);
}

void MyLeave(DATA* p)
{
    // warning C26135:
    // Missing side effect annotation _Releases_lock_(&p->cs)
    LeaveCriticalSection(&p->cs);
}
```

Warning C26135 is also issued when a conditional locking side effect is detected. To annotate a conditional effect, use the `_When_(ConditionExpr, LockAnnotation)` annotation, where `LockAnnotation` is either `_Acquires_lock_` or `_Releases_lock_` and the predicate expression `ConditionExpr` is a Boolean conditional expression. The side effects of other annotations on the same function only occur when `ConditionExpr` evaluates to true. Because `ConditionExpr` is used to relay the condition back to the caller, it must involve variables that are recognized in the calling context. These include function

parameters, global or class member variables, or the return value. To see the return value, use a special keyword in the annotation, `return`, as shown in the following example.

C++

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    int state;
} DATA;

_When_(return != 0, _Acquires_lock_(p->cs))
int TryEnter(DATA* p)
{
    if (p->state != 0)
    {
        EnterCriticalSection(&p->cs);
        return p->state;
    }

    return 0;
}
```

For shared/exclusive locks, also known as reader/writer locks, you can express locking side effects by using the following annotations:

- `_Acquires_shared_lock_(LockExpr)`
- `_Releases_shared_lock_(LockExpr)`
- `_Acquires_exclusive_lock_(LockExpr)`
- `_Releases_exclusive_lock_(LockExpr)`

Warning C26138

Article • 11/07/2022

Suspending a coroutine while holding lock '*lock*'.

Remarks

Warning C26138 warns when a coroutine is suspended while holding a lock. In general, we can't know how long will a coroutine remain in the suspended state so this pattern may result in longer critical sections than expected.

Code analysis name: SUSPENDED_WITH_LOCK

Examples

The following code will generate C26138.

C++

```
#include <experimental/generator>
#include <future>
#include <mutex>

using namespace std::experimental;

std::mutex global_m;
_Guarded_by_(global_m) int var = 0;

generator<int> mutex_acquiring_generator() {
    global_m.lock();
    ++var;
    co_yield 1; // @expected(26138), global_m is
    hold while yielding.
    global_m.unlock();
}

generator<int> mutex_acquiring_generator_report_once() {
    global_m.lock();
    ++var;
    co_yield 1; // @expected(26138), global_m is
    hold while yielding.
    co_yield 1; // @expected(26138), global_m is
    hold while yielding.
    global_m.unlock();
}
```

The following code will correct these warnings.

C++

```
#include <experimental/generator>
#include <future>
#include <mutex>

using namespace std::experimental;

std::mutex global_m;
_Guarded_by_(global_m) int var = 0;

generator<int> mutex_acquiring_generator2() {
{
    global_m.lock();
    ++var;
    global_m.unlock();
}
co_yield 1;                                // no 26138, global_m is already
released above.
}
```

Warning C26140

Article • 10/07/2022

Undefined lock kind 'lock' in annotation 'annotation' on lock 'lock'.

Example

C++

```
_Has_lock_kind_(MUTEXa) HANDLE gMutex;

struct CorrectExample
{
    _Has_lock_kind_(_Lock_kind_mutex_) HANDLE mMutex;
    _Guarded_by_(mMutex) int mData;
};

_WHEN_(return == WAIT_OBJECT_0 || return == WAIT_ABANDONED,
_ACQUIRES_LOCK_(gMutex))
DWORD UndefinedLockKind() // Warning C26140
{
    DWORD result = WaitForSingleObject(gMutex, 1000);
    return result;
}
```

Warning C26160

Article • 11/07/2022

Caller possibly failing to hold lock '*lock*' before calling function '*func*'.

Warning C26160 resembles warning [C26110](#) except that the confidence level is lower. For example, the function may contain annotation errors.

Examples

The following code generates warning C26160.

C++

```
struct Account
{
    _Guarded_by_(cs) int balance;
    CRITICAL_SECTION cs;

    _No_competing_thread_ void Init()
    {
        balance = 0; // OK
    }

    _Requires_lock_held_(this->cs) void FuncNeedsLock();

    _No_competing_thread_ void FuncInitCallOk()
        // this annotation requires this function is called
        // single-threaded, therefore we don't need to worry
        // about the lock
    {
        FuncNeedsLock(); // OK, single threaded
    }

    void FuncInitCallBad() // No annotation provided, analyzer generates
warning
    {
        FuncNeedsLock(); // Warning C26160
    }
};
```

The following code shows a solution to the previous example.

C++

```
struct Account
{
    _Guarded_by_(cs) int balance;
    CRITICAL_SECTION cs;

    _No_competing_thread_ void Init()
    {
        balance = 0; // OK
    }

    _Requires_lock_held_(this->cs) void FuncNeedsLock();

    _No_competing_thread_ void FuncInitCallOk()
        // this annotation requires this function is called
        // single-threaded, therefore we don't need to worry
        // about the lock
    {
        FuncNeedsLock(); // OK, single threaded
    }

    void FuncInitCallBadFixed() // this function now properly acquires (and
releases) the lock
    {
        EnterCriticalSection(&this-
>cs); FuncNeedsLock(); LeaveCriticalSection(&this->cs);
    }
};
```

Warning C26165

Article • 11/07/2022

Possibly failing to release lock '*lock*' in function '*func*'.

Warning C26165 resembles warning [C26115](#) except that the confidence level is lower. For example, the function may contain annotation errors.

Examples

The following code generates warning C26165.

C++

```
_Create_lock_level_(LockLevelOne);
_Create_lock_level_(LockLevelTwo);

struct LockLevelledStruct
{
    _Has_lock_level_(LockLevelOne) CRITICAL_SECTION a;
    _Has_lock_level_(LockLevelTwo) CRITICAL_SECTION b;
};

_Lock_level_order_(LockLevelOne, LockLevelTwo);

_Acquires_lock_(s->b) void GetLockFunc(LockLevelledStruct* s)
{
    EnterCriticalSection(&s->b);
}

void testLockLevelledStruct(LockLevelledStruct* s) // Warning C26165
{
    EnterCriticalSection(&s->a);
    GetLockFunc(s);
    LeaveCriticalSection(&s->a);
}
```

To correct this warning, change the previous example to the following.

C++

```
_Create_lock_level_(LockLevelOne);
_Create_lock_level_(LockLevelTwo);

struct LockLevelledStruct
{
    _Has_lock_level_(LockLevelOne) CRITICAL_SECTION a;
```

```
_Has_lock_level_(LockLevelTwo) CRITICAL_SECTION b;  
};  
  
_Lock_level_order_(LockLevelOne, LockLevelTwo);  
  
_Acquires_lock_(s->b) void GetLockFunc(LockLevelledStruct* s)  
{  
    EnterCriticalSection(&s->b);  
}  
  
_Releases_lock_(s->b) void ReleaseLockFunc(LockLevelledStruct* s)  
{  
    LeaveCriticalSection(&s->b);  
}  
  
void testLockLevelledStruct(LockLevelledStruct* s) // OK  
{  
    EnterCriticalSection(&s->a);  
    GetLockFunc(s);  
    ReleaseLockFunc(s);  
    LeaveCriticalSection(&s->a);  
}
```

Warning C26166

Article • 11/07/2022

Possibly failing to acquire or to hold lock '*lock*' in function '*func*'.

Warning C26166 resembles warning [C26116](#) except that the confidence level is lower. For example, the function may contain annotation errors.

Example

The following code shows code that will generate warning C26166.

C++

```
typedef struct _DATA {
    CRITICAL_SECTION cs;
} DATA;

_Acquires_lock_(p->cs) void Enter(DATA* p) {
    EnterCriticalSection(&p->cs); // OK
}

_Acquires_lock_(p->cs) void BAD(DATA* p) {} // Warning C26166
```

Warning C26167

Article • 11/07/2022

Possibly releasing unheld lock '*lock*' in function '*func*'.

Warning C26167 resembles warning [C26117](#) except that the confidence level is lower. For example, the function may contain annotation errors.

Examples

The following code will generate C26167 and C26110.

C++

```
typedef struct _DATA {
    CRITICAL_SECTION cs;
} DATA;

_Releases_lock_(p->cs) void Leave(DATA* p) {
    LeaveCriticalSection(&p->cs); // OK
}
void ReleaseUnheldLock(DATA* p) { // Warning C26167
    int i = 0;
    Leave(p); // Warning C26110
}
```

The following code will correct these warnings.

C++

```
typedef struct _DATA {
    CRITICAL_SECTION cs;
} DATA;

_Releases_lock_(p->cs) void Leave(DATA* p) {
    LeaveCriticalSection( &p->cs );
}

void ReleaseUnheldLock( DATA* p ) {
    EnterCriticalSection( &p->cs );
    int i = 0;
    Leave(p);
}
```

Warning C26800

Article • 06/26/2023

Use of a moved from object: '*object*'.

Remarks

Warning C26800 is triggered when a variable is used after it has been moved from. A variable is considered moved from after it's passed to a function as rvalue reference. There are some exceptions for assignment, destruction, and some state resetting functions such as `std::vector::clear`. After using a state resetting function, we're free to use the variable. This check only reasons about the local variables.

The following methods are considered state resetting methods:

- Functions with the following case-insensitive substring in their name: `clear`,
`clean`, `reset`, `free`, `destroy`, `release`, `dealloc`, `assign`
- Overloaded assignment operators, destructor

This check respects the `std::swap` operation:

C++

```
void f() {
    Y y1, y2;
    consume(std::move(y1));
    std::swap(y1, y2);
    y1.method();           // OK, valid after swap.
    y2.method();           // warning C26800
}
```

The check also supports the `try_emplace` operations in STL that conditionally move its argument:

C++

```
int g() {
    std::map<int, Y> m;
    Y val;
    auto emplRes = m.try_emplace(1, std::move(val));
    if (!emplRes.second) {
        val.method(); // No C26800, val was not moved because the insertion did
                      // not happen.
    }
}
```

Code analysis name: USE_OF_A_MOVED_FROM_OBJECT

Examples

The following code generates C26800.

C++

```
#include <utility>

struct X {
    X();
    X(const X&);
    X(X&&);
    X &operator=(X&);
    X &operator=(X&&);
    ~X();
};

template<typename T>
void use_cref(const T&);

void test() {
    X x1;
    X x2 = std::move(x1);
    use_cref(x1);           // warning C26800
}
```

The following code doesn't generate C26800.

C++

```
#include <utility>

struct MoveOnly {
    MoveOnly();
    MoveOnly(MoveOnly&) = delete;
    MoveOnly(MoveOnly&&);
    MoveOnly &operator=(MoveOnly&) = delete;
    MoveOnly &operator=(MoveOnly&&);
    ~MoveOnly();
};

template<typename T>
void use(T);

void test() {
    MoveOnly x;
```

```
use(std::move(x)); // no 26800
}
```

Warning C26810

Article • 06/05/2023

Lifetime of captured variable 'var' might end by the time the coroutine is resumed.

Remarks

Warning C26810 is triggered when a variable might be used after its lifetime ended in a resumed coroutine.

Code analysis name: COROUTINES_USE_AFTER_FREE_CAPTURE

Example

The following code generates C26810.

C++

```
#include <experimental/generator>
#include <future>

using namespace std::experimental;

coroutine_handle<> g_suspended_coro;

// Simple awaiter to allows to resume a suspended coroutine
struct ManualControl
{
    coroutine_handle<>& save_here;

    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<> h) { save_here = h; }
    void await_resume() {}
};

void bad_lambda_example1()
{
    int x = 5;
    auto bad = [x]() -> std::future<void> {
        co_await ManualControl{g_suspended_coro}; // @expected(26810), Lifetime
        of capture 'x' might end by the time this coroutine is resumed.
        printf("%d\n", x);
    };
    bad();
}
```

To fix this warning, consider using by-value arguments instead of captures:

C++

```
void bad_lambda_example1()
{
    int x = 5;
    auto good = [] (int x) -> std::future<void> {
        co_await ManualControl{g_suspended_coro};
        printf("%d\n", x);
    };
    good(x);
}
```

Alternatively, if the coroutine is guaranteed to live shorter than the lambda object, use `gsl::suppress` to suppress the warning and document the lifetime contracts in a comment.

See also

[C26811](#)

Warning C26811

Article • 06/05/2023

Lifetime of the memory referenced by parameter 'var' might end by the time the coroutine is resumed.

Remarks

Warning C26811 is triggered when a variable might be used after its lifetime ended in a resumed coroutine.

Code analysis name: COROUTINES_USE_AFTER_FREE_PARAM

Example

The following code generates C26811.

C++

```
#include <experimental/generator>
#include <future>

using namespace std::experimental;

// Simple awaiter to allows to resume a suspended coroutine
struct ManualControl
{
    coroutine_handle<>& save_here;

    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<> h) { save_here = h; }
    void await_resume() {}
};

coroutine_handle<> g_suspended_coro;

std::future<void> async_coro(int &a)
{
    co_await ManualControl{g_suspended_coro}; // @expected(26811), Lifetime
    of 'a' might end by the time this coroutine is resumed.
    ++a;
}
```

To fix this warning, consider taking the argument by value:

C++

```
std::future<void> async_coro(int a)
{
    co_await ManualControl{g_suspended_coro};
    ++a;
}
```

Alternatively, when the lifetime of `a` is guaranteed to outlive the lifetime of the coroutine, suppress the warning using `gsl::suppress` and document the lifetime contracts of the code.

See also

[C26810](#)

Warning C26813

Article • 10/07/2022

Use 'bitwise and' to check if a flag is set

Remarks

Most `enum` types with power of two member values are intended to be used as bit flags. As a result, you rarely want to compare these flags for equality. Instead, extract the bits you're interested in by using bitwise operations.

Code analysis name: `USE_BITWISE_AND_TO_CHECK_ENUM_FLAGS`

Example

C++

```
enum BitWise
{
    A = 1,
    B = 2,
    C = 4
};

void useEqualsWithBitwiseEnum(BitWise a)
{
    if (a == B) // Warning C26813: Use 'bitwise and' to check if a flag is
    set
        return;
}
```

To fix the warning, use bitwise operations:

C++

```
void useEqualsWithBitwiseEnum(BitWise a)
{
    if (a & B) // Fixed.
        return;
}
```

See also

C26827

C26828

Warning C26822

Article • 10/07/2022

Dereferencing a null pointer '*variable*' (lifetime.1)

Remarks

Dereferencing a null pointer is frequent problem in C and C++. We have several checks to deal with such problems. See this [blog post](#) for a comparison. When the analysis engine deduces the value of a pointer to be null and sees that pointer get dereferenced, it will emit a `C26822` warning. You can also enable `C26823` for a stricter analysis. This check also supports [SAL annotations](#) and `gsl::not_null` to describe invariants of the code.

Example

```
C++  
  
void f(int *p) {  
    if (p == nullptr)  
        *p = 42; // warning: C26822  
}  
  
void assign_to_gsl_notnull() {  
    int* p = nullptr;  
    auto q = gsl::make_not_null(p); // C26822 warning  
}
```

To solve this warning, make sure there's no null pointer dereference in the code, potentially by adding null checks. In case the code was found to be correct, false positive findings can often be fixed by using `gsl::not_null` or SAL annotations. There are some examples how to use some of those annotations below:

```
C++  
  
_Notnull_ int *get_my_ptr();  
gsl::not_null<int *> get_my_ptr2();  
  
void local_analysis(int *p) {  
    _Analysis_assume_(p != nullptr);  
    *p = 42;  
}
```

```
void local_analysis2(_In_ int *p) {
    int a = *p;
}
```

Warning C26823

Article • 10/07/2022

Dereferencing a possibly null pointer '*variable*' (lifetime.1)

Remarks

Dereferencing a null pointer is frequent problem in C and C++. We have several checks to deal with such problems. See this [blog post](#) for a comparison. When the analysis engine deduces that the value of a pointer might be null and sees that pointer get dereferenced, it will emit a `C26823` warning. You can enable `C26822` only for a more permissive analysis. This check also supports [SAL annotations](#) and [gsl::not_null](#) to describe invariants of the code.

Example

C++

```
void invalidate(int **pp);
void condition_null_dereference_invalidated(int* p)
{
    if (p)
        return;

    invalidate(&p);
    // The call above could reset the value of `p`, thus the low confidence
    // warning.
    *p = 5; // warning: C26823
}
```

To solve this warning, make sure there's no null pointer dereference in the code, potentially by adding null checks. In case the code was found to be correct, false positive findings can often be fixed by using `gsl::not_null` or SAL annotations. There are some examples how to use some of those annotations below:

C++

```
_Notnull_ int *get_my_ptr();
gsl::not_null<int *> get_my_ptr2();

void local_analysis(int *p) {
    _Analysis_assume_(p != nullptr);
    *p = 42;
```

```
}
```

```
void local_analysis2(_In_ int *p) {
    int a = *p;
}
```

Warning C26824

Article • 10/07/2022

Postcondition for null pointer '*variable*' requires it to be non-null (lifetime.1)

Remarks

Dereferencing a null pointer is a frequent problem in C and C++. We have several checks to deal with such problems. See this [blog post](#) for a comparison. When the analysis engine sees a null pointer returned from a function that has a contract forbidding such an operation, it will emit a `C26824` warning. You can also enable [C26825](#) for a stricter analysis. This check only works on functions annotated using [SAL annotations](#).

Example

C++

```
void postcondition_conditional(bool b, _When_(b == true, _Outptr_) int** p)
{
    if (b == true)
        *p = nullptr; // C26824 warning
}
```

To solve this warning, make sure there's no null pointer returned from the annotated function. Or, change the annotations to reflect the behavior of the function.

Warning C26825

Article • 10/07/2022

Postcondition for possibly null pointer '*variable*' requires it to be non-null (lifetime.1)

Remarks

Dereferencing a null pointer is a frequent problem in C and C++. We have several checks to deal with such problems. See this [blog post](#) for a comparison. When the analysis engine sees a potentially null pointer returned from a function that has a contract forbidding such operation, it will emit a `C26825` warning. You can enable [C26824](#) only for a more permissive analysis. This check only works on functions annotated using [SAL annotations](#).

Example

C++

```
void postcondition_conditional(int *q, _Outptr_ int** p) {
    *p = q; // C26825 warning
}
```

To solve this warning, make sure there's no null pointer returned from the annotated function. Or, change the annotations to reflect the behavior of the function.

Warning C26826

Article • 10/07/2022

Don't use C-style variable arguments (f.55).

For more information, see [F.55: Don't use va_arg arguments ↗](#) in the C++ Core Guidelines.

Remarks

This check warns on all usages of `va_list`, `va_start`, `va_arg`, and `va_end`, discouraging the use of C-style variable arguments. C-style variable arguments are unsafe because they require the programmer to assume that the arguments are all passed and read with the correct types.

Warning C26826 is available starting in Visual Studio 2022 version 17.1.

Example

C++

```
int sum(int n, ...) {
    va_list l; // C26826 Don't use C-style variable arguments
    va_start(l, n); // C26826 Don't use C-style variable arguments

    int s = 0;
    for (int i = 0; i < n; ++i) {
        // BAD, assumes the variable arguments will be passed as ints
        s += va_arg(l, int); // C26826 Don't use C-style variable arguments
    }

    va_end(l); // C26826 Don't use C-style variable arguments
    return s;
}

int main() {
    sum(2, 1, 2, 3); // ok
    sum(2, 1.5, 3.14159, 2.71828); // BAD, undefined
}
```

Alternatives to C-style variable arguments include:

- function overloading
- variadic templates

- `std::variant` arguments
- `std::initializer_list`

Warning C26827

Article • 10/07/2022

Did you forget to initialize an enum, or intend to use another type?

Remarks

Most `enum` types used in bitwise operations are expected to have members with values of powers of two. This warning attempts to find cases where a value wasn't given explicitly to an enumeration constant. It also finds cases where the wrong enumeration type may have been used inadvertently.

Code analysis name: `ALMOST_BITWISE_ENUM`

Example

The following sample code causes warning C26827:

```
C++

enum class AlmostBitWise
{
    A = 1,
    B = 2,
    C = 4,
    D
};

int almostBitwiseEnums(AlmostBitWise a, bool cond)
{
    return (int)a|(int)AlmostBitWise::A; // Warning C26827: Did you forget
    to initialize an enum, or intend to use another type?
}
```

To fix the warning, initialize the enumeration constant to the correct value, or use the correct enumeration type in the operation.

```
C++

enum class AlmostBitWise
{
    A = 1,
    B = 2,
    C = 4,
```

```
D = 8
};

int almostBitwiseEnums(AlmostBitWise a, bool cond)
{
    return (int)a|(int)AlmostBitWise::A; // No warning.
}
```

See also

[C26813](#)

[C26828](#)

Warning C26828

Article • 11/07/2022

Different enum types have overlapping values. Did you want to use another enum constant here?

Remarks

Most of the time, a single enumeration type describes all the bit flags that you can use for an option. If you use two different enumeration types that have overlapping values in the same bitwise expression, the chances are good those enumeration types weren't designed for use together.

Code analysis name: `MIXING_OVERLAPPING_ENUMS`

Example

The following sample code causes warning C26828:

C++

```
enum BitWiseA
{
    A = 1,
    B = 2,
    C = 4
};

enum class BitWiseB
{
    AA = 1,
    BB = 2,
    CC = 4,
    All = 7
};

int overlappingBitwiseEnums(BitWiseA a)
{
    return (int)a|(int)BitWiseB::AA; // Warning C26828: Different enum types
                                    // have overlapping values. Did you want to use another enum constant here?
}
```

To fix the warning, make sure enumeration types designed for use together have no overlapping values. Or, make sure all the related options are in a single enumeration

type.

C++

```
enum BitWiseA
{
    A = 1,
    B = 2,
    C = 4
};

int overlappingBitwiseEnums(BitWiseA a)
{
    return (int)a|(int)BitWiseA::A; // No warning.
}
```

See also

[C26813](#)

[C26827](#)

Warning C26829

Article • 12/16/2022

Empty optional '*variable*' is unwrapped

Remarks

Unwrapping empty `std::optional` values is undefined behavior. Such operation is considered a security vulnerability as it can result in a crash, reading uninitialized memory, or other unexpected behavior. This check will attempt to find cases where a `std::optional` is known to be empty and unwrapped. You can also enable [C26830](#), [C26859](#), and [C26860](#) for a stricter analysis.

Example

C++

```
void f(std::optional<int> maybeEmpty)
{
    std::optional<int> empty;
    std::optional<int> nonEmpty{5};
    *nonEmpty = 42; // No warning
    *empty = 42; // warning: C26829
    if (!maybeEmpty)
        *maybeEmpty = 42; // warning: C26829
}
```

To solve this problem, make sure the code never unwraps an empty optional.

Alternatively, use the `value` method and make sure you handle the

`std::bad_optional_access` exception.

Warning C26830

Article • 12/16/2022

Potentially empty optional '*variable*' is unwrapped

Remarks

Unwrapping empty `std::optional` values is undefined behavior. Such operation is considered a security vulnerability as it can result in a crash, reading uninitialized memory, or other unexpected behavior. This check will attempt to find cases where a `std::optional` isn't checked for emptiness before unwrap operations. You can enable [C26829](#) only for a more permissive analysis.

Example

C++

```
std::optional<int> getOptional();

void f(std::optional<int> maybeEmpty)
{
    if (maybeEmpty)
        *maybeEmpty = 42; // No warning
    *maybeEmpty = 5; // warning: C26830
    std::optional<int> o = getOptional();
    *o = 42; // warning: C26830
}
```

To solve this problem, make sure the code never unwraps an empty optional. Alternatively, use the `value` method and make sure you handle the `std::bad_optional_access` exception.

Warning C26831

Article • 05/22/2023

Allocation size might be the result of a numerical overflow

Remarks

This warning reports that the size specified for an allocation may be the result of a numerical overflow. For example:

C++

```
void *SmallAlloc(int);

void foo(int i, int j)
{
    int* p = (int*)SmallAlloc(i + j); // Warning: C26831
    p[i] = 5;
}
```

If `i+j` overflows, `SmallAlloc` returns a buffer that is smaller than expected. That will likely lead to out of bounds attempts to access the buffer later on. This code pattern can result in remote code execution vulnerabilities.

The check applies to common allocation functions like `new`, `malloc`, and `VirtualAlloc`. The check also applies to custom allocator functions that have `alloc` (case insensitive) in the function name.

This check sometimes fails to recognize that certain checks can prevent overflows because the check is conservative.

This warning is available in Visual Studio 2022 version 17.7 and later versions.

Example

To fix the previous code example in which `i+j` might overflow, introduce a check to make sure it won't. For example:

C++

```
void *SmallAlloc(int);

void foo(int i, int j)
```

```
{  
    if (i < 0 || j < 0 )  
    {  
        return;  
    }  
  
    if (i > 100 || j > 100)  
    {  
        return;  
    }  
  
    int* p = (int*)SmallAlloc(i + j);  
    p[i] = 5;  
}
```

See also

[C26832](#)

[C26833](#)

Warning C26832

Article • 05/22/2023

Allocation size is the result of a narrowing conversion that could result in overflow

Remarks

This warning reports that the size specified for an allocation may be the result of a narrowing conversion that results in a numerical overflow. For example:

C++

```
void* SmallAlloc(int);

void foo(unsigned short i, unsigned short j)
{
    unsigned short size = i + j;

    int* p = (int*)SmallAlloc(size); // Warning: C26832
    p[i] = 5;
}
```

In the expression `i + j`, both `i` and `j` are promoted to integers, and the result of the addition is stored in a temporary integer. Then, the temporary integer is implicitly cast to an `unsigned short` before the value is stored in `size`. The cast to `unsigned short` might overflow, in which case `SmallAlloc` may return a smaller buffer than expected. That will likely lead to out of bounds attempts to access the buffer later on. This code pattern can result in remote code execution vulnerabilities.

This check applies to common allocation functions like `new`, `malloc`, and `VirtualAlloc`. The check also applies to custom allocator functions that have `alloc` (case insensitive) in the function name.

This check sometimes fails to recognize that certain checks can prevent overflows because the check is conservative.

This warning is available in Visual Studio 2022 version 17.7 and later versions.

Example

To fix the previous code example in which `i+j` might overflow, introduce a check to make sure it won't. For example:

C++

```
void *SmallAlloc(int);

void foo(unsigned short i, unsigned short j)
{
    if (i > 100 || j > 100)
        return;

    unsigned short size = i + j;

    int* p = (int*)SmallAlloc(size);
    p[i] = 5;
}
```

See also

[C26831](#)

[C26833](#)

Warning C26833

Article • 05/22/2023

Allocation size might be the result of a numerical overflow before the bound check

Remarks

This warning reports that the size specified for an allocation may be the result of a numerical overflow. For example:

C++

```
void* SmallAlloc(int);

void foo(unsigned i, unsigned j)
{
    unsigned size = i + j;

    if (size > 50)
    {
        return;
    }

    int* p = (int*)SmallAlloc(size + 5); // Warning: C26833
    p[j] = 5;
}
```

The check for `size > 50` is too late. If `i + j` overflows, it produces a small value that passes the check. Then, `SmallAlloc` allocates a buffer smaller than expected. That will likely lead to out of bounds attempts to access the buffer later on. This code pattern can result in remote code execution vulnerabilities.

This check applies to common allocation functions like `new`, `malloc`, and `VirtualAlloc`. The check also applies to custom allocator functions that have `alloc` (case insensitive) in the function name.

This check sometimes fails to recognize that certain checks can prevent overflows because the check is conservative.

This warning is available in Visual Studio 2022 version 17.7 and later versions.

Example

To fix the previous code example, make sure `i+j` can't overflow. For example:

C++

```
void* SmallAlloc(int);

void foo(unsigned i, unsigned j)
{
    if (i > 100 || j > 100)
    {
        return;
    }

    unsigned size = i + j;

    if (size > 50)
    {
        return;
    }

    int* p = (int*)SmallAlloc(size + 5);
    p[j] = 5;
}
```

See also

[C26831](#)

[C26832](#)

Warning C26835

Article • 05/22/2023

`RtlCompareMemory` returns the number of matching bytes. Consider replacing this call with `RtlEqualMemory`

Remarks

When `RtlCompareMemory`'s return value is treated as a boolean, it evaluates to true when there is at least 1 equal byte before finding a difference. Moreover, comparing the result of `RtlCompareMemory` to 0 evaluates to false if there is at least 1 matching byte. This behavior may be unexpected because it's different from other comparison functions such as `strcmp`, making the code harder to understand. To check for equality, consider using `RtlEqualMemory` instead.

This warning is available in Visual Studio 2022 version 17.7 and later versions.

Example

C++

```
int foo(const void* ptr)
{
    if (RtlCompareMemory("test", ptr, 5) == 0) // C26835
    {
        // ...
    }
}
```

To fix the issue, verify if the original intention was to check for equality and replace the function call with `RtlEqualMemory`:

C++

```
int foo(const void* ptr)
{
    if (RtlEqualMemory("test", ptr, 5)) // C26835
    {
        // ...
    }
}
```

See also

[RtlEqualMemory macro \(wdm.h\)](#)
[RtlCompareMemory function \(wdm.h\)](#)

Warning C26837

Article • 11/30/2023

Value for the comparand `comp` for function `func` has been loaded from the destination location `dest` through non-volatile read.

This rule was added in Visual Studio 2022 17.8.

Remarks

The [InterlockedCompareExchange](#) function, and its derivatives such as [InterlockedCompareExchangePointer](#), perform an atomic compare-and-exchange operation on the specified values. If the `Destination` value is equal to the `Comparand` value, the `exchange` value is stored in the address specified by `Destination`. Otherwise, no operation is performed. The `interlocked` functions provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. This function is atomic with respect to calls to other `interlocked` functions. Misuse of these functions can generate object code that behaves differently than you expect because optimization can change the behavior of the code in unexpected ways.

Consider the following code:

```
C++  
  
#include <Windows.h>  
  
bool TryLock(__int64* plock)  
{  
    __int64 lock = *plock;  
    return (lock & 1) &&  
        _InterlockedCompareExchange64(plock, lock & ~1, lock) == lock;  
}
```

The intent of this code is:

1. Read the current value from the `plock` pointer.
2. Check if this current value has the least significant bit set.
3. If it does have least significant bit set, clear the bit while preserving the other bits of the current value.

To accomplish this, a copy of the current value is read from the `plock` pointer and saved to a stack variable `lock`. `lock` is used three times:

1. First, to check if the least-significant bit is set.
2. Second, as the `Comparand` value to `InterlockedCompareExchange64`.
3. Finally, in the comparison of the return value from `InterlockedCompareExchange64`

This assumes that the current value saved to the stack variable is read once at the start of the function and doesn't change. This is necessary because the current value is first checked before attempting the operation, then explicitly used as the `Comparand` in `InterlockedCompareExchange64`, and finally used to compare the return value from `InterlockedCompareExchange64`.

Unfortunately, the previous code can be compiled into assembly that behaves differently than from what you expect from the source code. Compile the previous code with the Microsoft Visual C++ (MSVC) compiler and the `/O1` option and inspect the resultant assembly code to see how the value of the lock for each of the references to `lock` is obtained. The MSVC compiler version v19.37 produces assembly code that looks like:

```
x86asm

plock$ = 8
bool TryLock(__int64 *) PROC ; TryLock, COMDAT
    mov    r8b, 1
    test   BYTE PTR [rcx], r8b
    je     SHORT $LN3@TryLock
    mov    rdx, QWORD PTR [rcx]
    mov    rax, QWORD PTR [rcx]
    and    rdx, -2
    lock  cmpxchg QWORD PTR [rcx], rdx
    je     SHORT $LN4@TryLock
$LN3@TryLock:
    xor    r8b, r8b
$LN4@TryLock:
    mov    al, r8b
    ret    0
bool TryLock(__int64 *) ENDP ; TryLock
```

`rcx` holds the value of the parameter `plock`. Rather than make a copy of the current value on the stack, the assembly code is re-reading the value from `plock` every time. This means the value could be different each time it's read. This invalidates the sanitization that the developer is performing. The value is re-read from `plock` after it's verified that it has its least-significant bit set. Because it's re-read after this validation is performed, the new value might no longer have the least-significant bit set. Under a

race condition, this code might behave as if it successfully obtained the specified lock when it was already locked by another thread.

The compiler is allowed to remove or add memory reads or writes as long as the behavior of the code isn't altered. To prevent the compiler from making such changes, force reads to be `volatile` when you read the value from memory and cache it in a variable. Objects that are declared as `volatile` aren't used in certain optimizations because their values can change at any time. The generated code always reads the current value of a `volatile` object when it's requested, even if a previous instruction asked for a value from the same object. The reverse also applies for the same reason. The value of the `volatile` object isn't read again unless requested. For more information about `volatile`, see [volatile](#). For example:

C++

```
#include <Windows.h>

bool TryLock(__int64* plock)
{
    __int64 lock = *static_cast<volatile __int64*>(plock);
    return (lock & 1) &&
        _InterlockedCompareExchange64(plock, lock & ~1, lock) == lock;
}
```

Compile this code with same `/O1` option as before. The generated assembly no longer reads `plock` for use of the cached value in `lock`.

For more examples of how the code can be fixed, see [Example](#).

Code analysis name: [INTERLOCKED_COMPARE_EXCHANGE_MISUSE](#)

Example

The compiler might optimize the following code to read `plock` multiple times instead of using the cached value in `lock`:

C++

```
#include <Windows.h>

bool TryLock(__int64* plock)
{
    __int64 lock = *plock;
    return (lock & 1) &&
```

```
        _InterlockedCompareExchange64(plock, lock & ~1, lock) == lock;  
    }
```

To fix the problem, force reads to be `volatile` so that the compiler doesn't optimize code to read successively from the same memory unless explicitly instructed. This prevents the optimizer from introducing unexpected behavior.

The first method to treat memory as `volatile` is to take the destination address as `volatile` pointer:

C++

```
#include <Windows.h>  
  
bool TryLock(volatile __int64* plock)  
{  
    __int64 lock = *plock;  
    return (lock & 1) &&  
        _InterlockedCompareExchange64(plock, lock & ~1, lock) == lock;  
}
```

The second method is using `volatile` read from the destination address. There are a few different ways to do this:

- Casting the pointer to `volatile` pointer before dereferencing the pointer
- Creating a `volatile` pointer from the provided pointer
- Using `volatile` read helper functions.

For example:

C++

```
#include <Windows.h>  
  
bool TryLock(__int64* plock)  
{  
    __int64 lock = ReadNoFence64(plock);  
    return (lock & 1) &&  
        _InterlockedCompareExchange64(plock, lock & ~1, lock) == lock;  
}
```

Heuristics

This rule is enforced by detecting if the value in the `Destination` of the `InterlockedCompareExchange` function, or any of its derivatives, is loaded through a non-

`volatile` read, and then used as the `Comparand` value. However, it doesn't explicitly check if the loaded value is used to determine the *exchange* value. It assumes the *exchange* value is related to the `Comparand` value.

See also

[InterlockedCompareExchange function \(winnt.h\)](#)

[_InterlockedCompareExchange intrinsic functions](#)

Warning C26859

Article • 12/16/2022

Empty optional '*variable*' is unwrapped, will throw exception

Remarks

Unwrapping empty `std::optional` values via the `value` method will throw an exception. Such operation can result in a crash when the exception isn't handled. This check will attempt to find cases where a `std::optional` is known to be empty and unwrapped using the `value` method. You can also enable [C26829](#), [C26830](#), and [C26860](#) for a stricter analysis.

Example

C++

```
void f(std::optional<int> maybeEmpty)
{
    std::optional<int> empty;
    std::optional<int> nonEmpty{5};
    nonEmpty.value() = 42; // No warning
    empty.value() = 42; // warning: C26859
    if (!maybeEmpty)
        maybeEmpty.value() = 42; // warning: C26859
}
```

To solve this problem, make sure the code never unwraps an empty optional.

Warning C26860

Article • 12/16/2022

Potentially empty optional '*variable*' is unwrapped, may throw exception

Remarks

Unwrapping empty `std::optional` values via the `value` method will throw an exception. Such operation can result in a crash when the exception isn't handled. This check will attempt to find cases where a `std::optional` isn't checked for emptiness before unwrapping it via the `value` method. You can enable [C26829](#), and [C26859](#) only for a more permissive analysis.

Example

C++

```
std::optional<int> getOptional();

void f(std::optional<int> maybeEmpty)
{
    if (maybeEmpty)
        maybeEmpty.value() = 42; // No warning
    maybeEmpty.value() = 5; // warning: C26860
    std::optional<int> o = getOptional();
    o.value() = 42; // warning: C26860
}
```

To solve this problem, make sure the code never unwraps an empty optional.

Warning C26861

Article • 11/30/2023

Field of a date-time object `var` has been modified without proper leap year checking: `expr`

This rule was added in Visual Studio 2022 17.8.

Remarks

In the Gregorian calendar, every year exactly divisible by four is a leap year--except for years that are exactly divisible by 100. The centurial years are also leap years if they're exactly divisible by 400.

A leap year bug occurs when software doesn't account for this leap year logic, or uses flawed logic. This can affect reliability, availability, or even the security of the affected system.

It isn't safe to add or subtract some number to or from the year, month, or day field of a date-time object without taking leap years into account. This calculation is commonly performed to determine the expiration date for a certificate, for example. On many dates, a naive calculation may produce the desired result. However, when the result is February 29 (a leap day) and the year isn't a leap year, the result is invalid.

For example, adding a year to 2020-01-31 produces 2021-01-31. But adding a year to 2020-02-29 produces 2021-02-29, which isn't a valid date because 2021 isn't a leap year.

Be cautious when manipulating variables that represent date values. Handle leap years and leap days properly, or use an API or library that handles date arithmetic safely.

Code analysis name: `DATETIME_MANIPULATION_WITHOUT_LEAPYEAR_CHECK`

Example

The following code advances the system time by a year by incrementing the year field of the date-time object representing the system time. However, it can produce an invalid date-time object if the date was February 29 before the modification, because the next year isn't a leap year:

C++

```
SYSTEMTIME st;
GetSystemTime(&st);
st.wYear++; // warning C26861
```

To avoid creating an invalid date-time object due to a leap year, check if the resulting date is still valid and make the necessary adjustments to make it valid, as in this example:

C++

```
SYSTEMTIME st;
GetSystemTime(&st);
st.wYear++;
if (st.wMonth == 2 && st.wDay == 29)
{
    // move back a day when landing on Feb 29 in a non-leap year
    bool isLeapYear = st.wYear % 4 == 0 && (st.wYear % 100 != 0 || st.wYear
% 400 == 0);
    if (!isLeapYear)
    {
        st.wDay = 28;
    }
}
```

Heuristics

Currently, this rule only recognizes the Windows `SYSTEMTIME` struct and C `tm` struct.

This rule employs a simplified heuristic to find potentially risky changes and reports warnings unless there's appropriate leap year or leap day checking. It doesn't try to verify if the leap year or leap day checking is performed correctly for the modified date-time object.

This rule is an opt-in rule, which means that code analysis should use a ruleset file, and the rule should be explicitly included in the ruleset file, and enabled for it to be applied. For more information on creating a custom ruleset for code analysis, see: [Use Rule Sets to Specify the C++ Rules to Run](#).

See also

[C6393](#)

[C6394](#)

[C26862](#)

C26863

C26864

Warning C26862

Article • 11/30/2023

A date-time object `var` has been created from a different type of date-time object but conversion was incomplete: `expr`

This rule was added in Visual Studio 2022 17.8.

Remarks

Proper enforcement of leap year and leap day handling rules require tracking the proper conversion between date-time objects of different types such as the Windows `SYSTEMTIME` struct and the C `tm` struct. Different date-time types may have different bases for the year, month, and day fields. For example, `SYSTEMTIME` has a 0-based year, but 1-based month and day fields. On the other hand, `tm` has a 1900-based year, a 0-based month, and a 1-based day fields. To convert an object of one of these types to an object of another type, the year, month, and day fields must be adjusted appropriately.

Code analysis name: `INCOMPLETE_DATETIME_CONVERSION`

Example

The following code tries to convert an instance of `tm` into an instance of `SYSTEMTIME`. It makes the necessary adjustment to the year field, but doesn't properly adjust the month field:

C++

```
#include <Windows.h>
#include <ctime>

void ConvertTmToSystemTime1b(const tm& tm)
{
    SYSTEMTIME st;
    st.wYear = tm.tm_year + 1900;
    st.wMonth = tm.tm_mon; // C26862, Adjustment is missing
    st.wDay = tm.tm_mday;
}
```

To fix this problem, adjust the month and year fields:

C++

```
#include <Windows.h>
#include <ctime>

void ConvertTmToSystemTime(const tm& tm)
{
    SYSTEMTIME st;
    st.wYear = tm.tm_year + 1900;
    st.wMonth = tm.tm_mon + 1;
    st.wDay = tm.tm_mday;
}
```

Heuristics

This rule only recognizes the Windows `SYSTEMTIME` struct and the C `tm` struct.

This rule is an opt-in rule, meaning that code analysis should use a ruleset file, and the rule should be explicitly included in the ruleset file, and enabled for it to be applied. For more information on creating a custom ruleset for code analysis, see [Use Rule Sets to Specify the C++ Rules to Run](#).

See also

[C6393](#)

[C6394](#)

[C26861](#)

[C26863](#)

[C26864](#)

Warning C26863

Article • 11/30/2023

Return value from a date-time handling function `func` is ignored

This rule was added in Visual Studio 2022 17.8.

Remarks

It's important to verify the return value of a function that transforms a date structure when the year, month, or date input argument was manipulated without proper leap year handling. Otherwise, the function may have failed and execution continues with an output parameter containing invalid data.

The following is a list of the functions that this warning covers:

- [FileTimeToSystemTime](#)
- [SystemTimeToFileTime](#)
- [SystemTimeToTzSpecificLocalTime](#)
- [SystemTimeToTzSpecificLocalTimeEx](#)
- [TzSpecificLocalTimeToSystemTime](#)
- [TzSpecificLocalTimeToSystemTimeEx](#)
- [RtlLocalTimeToSystemTime](#)
- [RtlTimeToSecondsSince1970](#)

Code analysis name: `DATETIME_MANIPULATION_FUNCTION_RETURN_IGNORED`

Example

The following code tries to get current system time, advance the month field by one month, and get the file time that corresponds to the updated system time via [SystemTimeToFileTime](#). However, [SystemTimeToFileTime](#) might fail, as the updated system time may become invalid:

C++

```
#include <Windows.h>

void foo()
{
    FILETIME ft;
```

```
SYSTEMTIME st;
GetSystemTime(&st);
st.wMonth++; // Advance month by one
// Get the file time
SystemTimeToFileTime(&st, &ft); // C26863
}
```

To fix the problem, always check the return value from date-time manipulation functions and handle failures appropriately:

C++

```
#include <Windows.h>

void foo()
{
    FILETIME ft;
    SYSTEMTIME st;
    GetSystemTime(&st);

    st.wMonth++; // Advance month by one
    // Get file time
    if (SystemTimeToFileTime(&st, &ft))
    {
        // Use file time
    }
}
```

Heuristics

This rule only recognizes the Windows `SYSTEMTIME` struct and the C `tm` struct.

This rule is enforced regardless of whether the input arguments were validated before calling these functions. If all the input arguments are validated before calling the function, this rule can report false warning.

This rule is an opt-in rule, meaning that code analysis should use a ruleset file, and the rule should be explicitly included in the ruleset file, and enabled for it to be applied. For more information on creating a custom ruleset for code analysis, see [Use Rule Sets to Specify the C++ Rules to Run](#).

See also

[C6393](#)

[C6394](#)

C26861

C26862

C26864

Warning C26864

Article • 11/30/2023

Day field of a date-time object `var` has been modified assuming 365 days per year without proper leap year checking: `expr`

This rule was added in Visual Studio 2022 17.8.

Remarks

In the Gregorian calendar, every year exactly divisible by four is a leap year--except for years that are exactly divisible by 100. The centurial years are also leap years if they're exactly divisible by 400.

A leap year bug occurs when software doesn't account for this leap year logic, or uses flawed logic. This can affect reliability, availability, or even the security of the affected system.

You must take leap years into account when you perform arithmetic operations on a variable that represents a date. It's not safe to assume that a year is 365 days long. A leap year has 366 days because of the 'leap day' added as a 29th day in February.

To correctly advance a year, determine whether the time span contains a leap day, then perform the calculation using the correct number of days. It's better if the year is directly advanced, with an appropriate leap day check on the resulting date. Alternatively, use an established library routine that handles leap years correctly.

Code analysis name: `DATETIME_MANIPULATION_ASSUMING_365_DAYS_WITHOUT_LEAPYEAR_CHECK`

Example

The following code tries to get current system time, advance the day by one year by adding 365 days to the day field, and adjusting the date per leap year rule. However, the result may not fall on the same month/date of the next year:

C++

```
#include <Windows.h>

void foo()
{
```

```

SYSTEMTIME st;

GetSystemTime(&st);

// Advance a year by adding 365 days
st.wDay += 365;      // C26864
}

```

To fix this problem, advance the year field directly and adjust the date per the leap year rule:

C++

```

#include <Windows.h>

void foo()
{
    SYSTEMTIME st;
    GetSystemTime(&st);

    st.wYear++; // Advance a year

    // Adjust the date
    if (st.wMonth == 2 && st.wDay == 29)
    {
        // Move back a day when landing on Feb 29 in a non-leap year
        bool isLeapYear = st.wYear % 4 == 0 && (st.wYear % 100 != 0 ||
st.wYear % 400 == 0);
        if (!isLeapYear)
        {
            st.wDay = 28;
        }
    }
}

```

Heuristics

This rule only recognizes the Windows `SYSTEMTIME` struct and C `tm` struct.

This rule is enforced if the date field is directly modified by 365 days. It doesn't consider if the value of date field is assigned to another variable and then manipulated, and so can miss equivalent mistakes.

This rule is an opt-in rule, meaning that code analysis should use a ruleset file, and the rule should be explicitly included in the ruleset file, and enabled for it to be applied. For more information on creating a custom ruleset for code analysis, see [Use Rule Sets to Specify the C++ Rules to Run](#).

See also

[C6393](#)

[C6394](#)

[C26861](#)

[C26862](#)

[C26863](#)

Warning C28020

Article • 10/07/2022

The expression '*expr*' is not true at this call

This warning is reported when the `_Satisfies_` expression listed isn't true. Frequently, the warning indicates an incorrect parameter.

If this warning occurs on a function declaration, the annotations indicate an impossible condition.

Warning C28021

Article • 10/07/2022

The parameter '*param*' being annotated with '*annotation*' must be a pointer

This warning is reported when the object being annotated is not a pointer type. This annotation can't be used with `void` or integral types.

Warning C28022

Article • 10/07/2022

The function class(es) '*classlist1*' on this function do not match the function class(es) '*classlist2*' on the typedef used to define it.

This warning is reported when there's an error in the annotations. Both the typedef and the function itself have `_Function_class_` annotations, but they don't match. If both are used, they must match.

Warning C28023

Article • 10/07/2022

The function being assigned or passed should have a `_Function_class_` annotation for at least one of the class(es) in: '*classlist*'

This warning is often reported when only one function class is in use and a callback of the appropriate type isn't declared.

This warning is issued when the function on the left side of the assignment (or of the implied assignment, if it's a function call) is annotated to indicate that it's a driver-specific function type that uses the `_Function_class_` annotation, or a typedef that contains such an annotation. The function on the right side of the assignment doesn't have a `_Function_class_` annotation. The function on the right should be annotated to be of the same type as the function on the left. Often the best approach is to place the declaration `<class1> <funcname1>` before the current first declaration of `<funcname2>`.

Warning C28024

Article • 10/07/2022

The function pointer being assigned to is annotated with the function class '*class*', which is not contained in the function class(es) '*classlist*'.

This warning is reported when both functions were annotated with a function class, but the classes don't match.

This warning is issued when a function pointer has a `_Function_class_` annotation that specifies that only functions of a particular functional class should be assigned to it. In an assignment or implied assignment in a function call, the source and target must be of the same function class, but the function classes don't match.

Warning C28039

Article • 10/07/2022

The type of actual parameter '*operand*' should exactly match the type '*typename*'

This warning is reported when an `enum` formal wasn't passed a member of the `enum`, but may also be used for other types.

Because C permits `enum` types to be used interchangeably, and interchangeably with constants, it's easy to pass the wrong `enum` value to a function without an error.

For `enum` types, if the type of an `enum` parameter is annotated with `_Enum_is_bitflag_`, arithmetic is permitted on the parameter. Otherwise the parameter must be of exactly the correct type. If a constant is strictly required, warning C28137 may also apply.

This rule can be used for other parameter types as well; see the function documentation for why the types must match exactly.

Warning C28103

Article • 10/07/2022

Leaking resource

The specified object contains a resource that hasn't been freed. A function being called has been annotated with `__drv_acquiresResource` or `__drv_acquiresResourceGlobal` and this warning indicates that the resource named in the annotation wasn't freed.

Example

The following code example generates this warning:

C++

```
res = KeSaveFloatingPointState(buffer);
```

The following code example avoids this warning:

C++

```
res = KeSaveFloatingPointState(buffer);
if (NT_SUCCESS(res))
{
    res = KeRestoreFloatingPointState(buffer);
}
```

If this warning is reported as a false positive, the most likely cause is that the function that releases the resource isn't annotated with `__drv_releasesResource` or `__drv_releasesResourceGlobal`. If you're using wrapper functions for system functions, the wrapper functions should use the same annotations that the system functions do. Currently, many system functions are annotated in the model file, so the annotations aren't visible in the header files.

Warning C28104

Article • 10/07/2022

Resource that should have been acquired before function exit was not acquired

A function that is intended to acquire a resource before it exits has exited without acquiring the resource. This warning indicates that the function is annotated with `__drv_acquiresResource` but doesn't return having actually acquired the resource. If this function is a wrapper function, a path through the function didn't reach the wrapped function. If the failure to reach the wrapped function is because the function returned an error and didn't actually acquire the resource, you might need to use a conditional annotation (`__drv_when`).

If this function actually implements the acquisition of the resource, it might not be possible for PFD to detect that the resource is acquired. In that case, use a `#pragma` warning to suppress the error. You can probably place the `#pragma` on the line preceding the `{` that begins the function body. The calling functions still need the annotation, but the Code Analysis tool won't be able to detect that the resource was acquired.

Example

C++

```
__drv_acquireResourceGlobal(HWLock, lockid)
void GetHardwareLock(lockid)
#pragma warning (suppress: 28104)
{
    // code to implement a hardware lock (which the Code Analysis tool can't
    // recognize)
}
```

Warning C28105

Article • 10/07/2022

Leaking resource due to an exception

The specified resource isn't freed when an exception is raised. The statement specified by the path can raise an exception. This warning is similar to warning [C28103](#), except that in this case an exception is involved.

Example

The following code example generates this warning:

```
C++  
  
res = KeSaveFloatingPointState(buffer);  
  
res = AllocateResource(Resource);  
char *p2 = new char[10]; // could throw  
  
delete[] p2;  
FreeResource(Resource)
```

The following code example avoids this warning:

```
C++  
  
res = AllocateResource(Resource);  
char *p2;  
  
try {  
    p2 = new char[10];  
} catch (std::bad_alloc *e) {  
    // just handle the throw  
    ;  
}  
FreeResource(Resource)
```

Warning C28106

Article • 10/07/2022

Variable already holds resource possibly causing leak

A variable that contains a resource is used in a context in which a new value can be placed in the variable. If such placement occurs, the original resource can be lost and not properly freed, causing a resource leak.

Example

The following code example generates this warning:

C++

```
ExAcquireResourceLite(resource, true);
//...
ExAcquireResourceLite(resource, true);
```

The following code example avoids this warning:

C++

```
ExAcquireResourceLite(resource1, true);
//...
ExAcquireResourceLite(resource2, true);
```

Warning C28107

Article • 10/07/2022

Resource must be held when calling function

A resource that the program must acquire before calling the function wasn't acquired when the function was called. As a result, the function call will fail. This warning is reported only when resources are acquired and released in the same function.

Example

The following code example generates this warning:

C++

```
ExAcquireResourceLite(resource, true);
ExReleaseResourceLite(resource);
```

The following code example avoids this warning:

C++

```
KeEnterCriticalSection();
ExAcquireResourceLite(resource, true);
ExReleaseResourceLite(resource);
KeLeaveCriticalSection();
KeEnterCriticalSection();
ExAcquireResourceLite(resource, true);
ExReleaseResourceLite(resource);
KeLeaveCriticalSection();
```

Warning C28108

Article • 10/07/2022

Variable holds an unexpected resource

The resource that the driver is using is in the expected C language type, but has a different semantic type.

Example

The following code example generates this warning:

C++

```
KeAcquireInStackSpinLock(spinLock, lockHandle);
//...
KeReleaseSpinLock(spinLock, 0);
```

The following code example avoids this warning:

C++

```
KeAcquireInStackSpinLock(spinLock, lockHandle);
//...
KeReleaseInStackSpinLock(lockHandle);
```

Warning C28109

Article • 10/07/2022

Variable cannot be held at the time function is called

The program is holding a resource that shouldn't be held when it's calling this function. Typically, it indicates that the resource was unintentionally acquired twice. The Code Analysis tool reports this warning when resources are acquired and released in the same function.

Example

The following code example generates this warning:

C++

```
ExAcquireResourceLite(resource, true);
//...
ExAcquireResourceLite(resource, true);
```

The following code example avoids this warning:

C++

```
ExAcquireResourceLite(resource, true);
```

Warning C28112

Article • 10/07/2022

A variable (*parameter-name*) which is accessed via an Interlocked function must always be accessed via an Interlocked function. See line *line-number*: It is not always safe to access a variable which is accessed via the Interlocked* family of functions in any other way.

A variable that is accessed by using the Interlocked executive support routines, such as `InterlockedCompareExchangeAcquire`, is later accessed by using a different function.

Remarks

`InterlockedXXX` functions are intended to provide atomic operations, but are only atomic with respect to other `InterlockedXXX` functions. Although certain ordinary assignments, accesses, and comparisons to variables that are used by the Interlocked* routines can be safely accessed by using a different function, the risk is great enough to justify examining each instance.

Code analysis name: INTERLOCKED_ACCESS

Example

The following code generates this warning:

C++

```
inter_var--;  
//code  
InterlockedIncrement(&inter_var);
```

The following code corrects this warning by strictly accessing `inter_var` through `InterlockedXXX` functions:

C++

```
InterlockedDecrement(&inter_var);  
//code  
InterlockedIncrement(&inter_var);
```

See Also

[InterlockedIncrement function \(wdm.h\)](#) [InterlockedDecrement function \(wdm.h\)](#)

Warning C28113

Article • 10/07/2022

Accessing a local variable via an Interlocked function

The driver is using an Interlocked executive support routine, such as [InterlockedDecrement](#), to access a local variable.

Although drivers are permitted to pass the address of a local variable to another function, and then use an interlocked function to operate on that variable, it's important to verify that the stack won't be swapped out to disk unexpectedly and that the variable has the correct life time across all threads that might use it.

Example

Typically, the return value of an Interlocked executive support routine is used in subsequent computations, instead of the input arguments. Also, the Interlocked routines only protect the first (leftmost) argument. Using an Interlocked routine in the following way doesn't protect the value of global and often serves no purpose.

C++

```
InterlockedExchange(&local, global)
```

The following form has the same effect on the data and safely accesses the global variable.

C++

```
local = InterlockedExchange(&global, global)
```

Warning C28125

Article • 10/07/2022

The function must be called from within a try/except block

The driver is calling a function that must be called from within a try/except block, such as [ProbeForRead](#), [ProbeForWrite](#), or [MmProbeAndLockPages](#).

Example

The following code example generates this warning:

C++

```
ProbeForRead(addr, len, 4);
```

The following code example avoids this warning:

C++

```
__try
{
    ProbeForRead(addr, len, 4);
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    Status = GetExceptionCode();
    ... report error status
}
```

Warning C28137

Article • 10/07/2022

The variable argument should instead be a (literal) constant

This warning is reported when a function call is missing a required (literal) constant. Consult the documentation for the function.

Example

For example, the [ExAcquireResourceExclusiveLite](#) routine requires a value of TRUE or FALSE for the `Wait` parameter. The following example code generates this warning:

C++

```
ExAcquireResourceExclusiveLite(Resource, Wait);
```

The following code example avoids this warning:

C++

```
ExAcquireResourceExclusiveLite(Resource, TRUE);
```

Warning C28138

Article • 10/07/2022

The constant argument should instead be variable

This warning is reported in a function call that expects a variable or a non-constant expression, but the call includes a constant. For information about the function and its parameter, consult the WDK documentation of the function.

Example

For example, in the following code example, the parameter of the `READ_PORT_UCHAR` macro must be a pointer to the port address, not the address provided as a constant.

The following code example generates this warning message:

C++

```
READ_PORT_UCHAR(0x80001234);
```

To correct this warning, use a pointer to the port address.

C++

```
READ_PORT_UCHAR(PortAddress);
```

There are a few older devices for which a constant parameter is acceptable with the `READ_PORT` and `WRITE_PORT` family of functions. When those devices receive this warning, the warning can be suppressed or ignored. However, any new devices shouldn't assume a constant hardware address.

Warning C28159

Article • 10/07/2022

Consider using `*function_name_1*` instead of `*function_name_2*`. Reason: *reason*

This warning occurs when you use a function that is semantically equivalent to an alternative, preferred function call.

Remarks

C28159 is a general warning message; the annotation `__drv_preferredFunction` was used (possibly with a conditional `__drv_when()` annotation) to flag a bad coding practice.

Code analysis name: USE_OTHER_FUNCTION

Example

The following code example generates this warning. This issue is due to the use of `OemToChar`, which doesn't validate the buffer size:

C++

```
char buff[MAX_PATH];
OemToChar(buff, input); // If strlen(input) > MAX_PATH, this call leads to
buffer overrun
```

The following code example avoids this warning by using the recommended alternative `OemToCharBuff`, which takes in the destination buffer size and limits the copy appropriately:

C++

```
char buff[MAX_PATH];
OemToCharBuff(buff, input, MAX_PATH);
```

Warning C28160

Article • 10/07/2022

Error annotation: *reason*

This warning is reported when a `__drv_error` annotation has been encountered.

Remarks

This annotation is used to flag coding practices that should be fixed, and can be used with a `__drv_when` annotation to indicate specific combinations of parameters.

Code analysis name: ERROR

Warning C28163

Article • 10/07/2022

The function should never be called from within a try/except block

This warning is reported when a function is of a type that should never be enclosed in a `try/except` block is found in a `try/except` block. The code analysis tool found at least one path in which the function called was within a `try/except` block.

Warning C28164

Article • 10/07/2022

The argument is being passed to a function that expects a pointer to an object (not a pointer to a pointer)

This warning is reported when a pointer to a pointer is used in a call to a function that is expecting a pointer to an object.

The function takes a `PVOID` in this position. Usually, it indicates that `&pXXX` was used when `pXXX` is required.

Some *polymorphic functions* (functions that can evaluate to, and be applied to, values of different types) are implemented in C by using a `PVOID` argument that takes any pointer type. However, this allows the programmer to code a pointer to a pointer without causing a compiler error, even when this type isn't appropriate.

Example

The following code example generates this warning:

C++

```
PFAST_MUTEX pFm;  
//...  
KeWaitForSingleObject(&pFm, UserRequest, UserMode, false, NULL);
```

The following code example avoids the warning:

C++

```
PFAST_MUTEX pFm;  
//...  
KeWaitForSingleObject(pFm, UserRequest, UserMode, false, NULL);
```

Warning C28182

Article • 10/07/2022

Dereferencing NULL pointer.

Additional information: *<pointer1>* contains the same NULL value as *<pointer2>* did.

<note>

The code analysis tool reports this warning when it confirms that the pointer can be NULL. If there are unconfirmed instances where the error might occur earlier in the trace, the code analysis tool adds the line number of the first instance to the warning message so that you can change the code to address all instances.

<pointer2> is confirmed to be potentially NULL. *<pointer1>* contains the same value as *pointer2* and is being dereferenced. Because these pointers may be at different places in the code, both are reported so that you can determine why the code analysis tool is reporting this warning.

If an unconfirmed earlier instance of the condition exists, then *<note>* is replaced by this text: "See line *<number>* for an earlier location where this can occur."

Examples

The following example shows code that could cause the code analysis tool to generate this warning message. In this example, the code analysis tool determines that `pNodeFree` is NULL in the `if` statement, and the code path into the body of the `if` is taken. However, because `nBlockSize` is potentially zero, the body of the `for` statement isn't executed and `pNodeFree` is left unmodified. `pNodeFree` is then assigned to `pNode`, and `pNode` is used while a NULL dereference could occur.

C++

```
typedef struct xlist {
    struct xlist *pNext;
    struct xlist *pPrev;
} list;

list *pNodeFree;
list *masterList;
int nBlockSize;

void fun()
{
```

```

if (pNodeFree == 0)
{
    list *pNode = masterList;

    for (int i = nBlockSize-1; i >= 0; i--, pNode--)
    {
        pNode->pNext = pNodeFree;
        pNodeFree = pNode;
    }
}

list* pNode = pNodeFree;
pNode->pPrev = 0;
}

```

The code analysis tool reports the following warning:

Output

```

:\sample\testfile.cpp(24) : warning C28182: Dereferencing NULL pointer.
'pNode' contains the same NULL value as 'pNodeFree' did.: Lines: 12, 14, 16,
23, 24

```

One way to correct the earlier example is to check `pNode` for zero before dereferencing it so that a NULL dereference is averted. The following code shows this correction.

C++

```

typedef struct xlist {
    struct xlist *pNext;
    struct xlist *pPrev;
} list;

list *pNodeFree;
list *masterList;
int nBlockSize;

void fun()
{
    if (pNodeFree == 0)
    {
        list *pNode = masterList;

        for (int i = nBlockSize-1; i >= 0; i--, pNode--)
        {
            pNode->pNext = pNodeFree;
            pNodeFree = pNode;
        }
    }

    pNode = pNodeFree;
}

```

```
if(pNode != 0)
    pNode->pPrev = 0;
}
```

Warning C28183

Article • 10/07/2022

The argument could be one value, and is a copy of the value found in the pointer

This warning indicates that this value is unexpected in the current context. This warning usually appears when a `NULL` value is passed as an argument to a function that doesn't permit it. The value was actually found in the specified variable, and the argument is a copy of that variable.

The Code Analysis tool reports this warning at the first point where it can definitively determine that the pointer is `NULL` or that it contains an illegal value. However, it's often the case that the error could actually occur earlier in the trace. When it happens, the Code Analysis tool will also give the line number of the first possible instance, usually at a location where it couldn't definitively determine that the warning was appropriate. In those cases, the earlier location where it can occur is appended to the warning message. Typically, a code change should occur at or before that line number, rather than at the point of report.

Example

In the following example, the Code Analysis tool determines that `s` is `NULL` in the `if` statement, and the body of the `if` is taken. The pointer `s` is then assigned to `t` and then `t` is used in a way where a `NULL` dereference could occur.

C++

```
#include <windows.h>

int fun2(char *s)
{
    char *t;
    if (s == NULL) {
        //... but s is unchanged
    }

    t = s;

    return lstrlenA(t);
}
```

For this code example, the Code Analysis tool reports the following warning:

Output

```
d:\sample\testfile.cpp(38) : warning C28183: 't' could be '0', and is a copy  
of the value found in 's': this does not adhere to the specification for  
the function 'lstrlenA'.: Lines: 31, 32, 36, 38
```

Warning C28193

Article • 10/07/2022

The variable holds a value that must be examined

This warning indicates that the calling function isn't checking the value of the specified variable, which was supplied by a function. The returned value is annotated with the `_Check_return_` annotation, but the calling function is either not using the value or is overwriting the value without examining it.

This warning is similar to warning [C6031](#), but it's reported only when the code doesn't test or examine the value of the variable, such as by using it in a comparison. Simply assigning the value isn't considered to be a sufficient examination to avoid this warning. Aliasing the result out of the function is considered a sufficient examination, but the result itself should be annotated with `_Check_return_`.

Certain functions (such as `strlen`) exist almost exclusively for their return value, so it makes sense for them to have the `_Check_return_` annotation. For these functions, the Code Analysis tool might report this warning when the return value is unused. This warning usually indicates that the code is incorrect, for example, it might contain residual code that could be deleted. However, in some rare instances, the return value is intentionally not used. The most common of these instances is where a string length is returned but not actually used before some other test is made. That other test causes a path to be simulated where the string length ends up being unused. When this happens, the code can be correct, but it might be inefficient.

There are two primary strategies for dealing with these cases where the return value is unused:

Reorder the code so that the string length is only returned along the path where it's needed.

Use a `#pragma` warning to suppress the warning--if by reordering the code, you would make the code too complex or otherwise less useful.

Example

The following code example generates this warning:

C++

```
IoGetDmaAdapter(pPDO, &DevDesc, &nMapRegs);  
//...
```

The following code example avoids this warning:

C++

```
IoGetDmaAdapter(pPDO, &DevDesc, &nMapRegs);  
//...  
if (nMapRegs < MIN_REQUIRED_MAPS) {  
//...  
}
```

Warning C28194

Article • 10/07/2022

The function was declared as aliasing the value in variable and exited without doing so

This warning indicates that the function prototype for the function being analyzed has a `__drv_isAliased` annotation, which indicates that it will *alias* the specified argument (that is, assign the value in a way that it will survive returning from the function).

However, the function doesn't alias the argument along the path that is indicated by the annotation. Most functions that alias a variable save its value to a global data structure.

Warning C28195

Article • 10/07/2022

The function was declared as acquiring memory in a variable and exited without doing so

This warning indicates that the function prototype for the function being analyzed has a `__drv_allocatesMem` annotation. The `__drv_allocatesMem` annotation indicates that the function acquires memory in the designated result location, but in at least one path, the function didn't acquire the memory. The Code Analysis tool won't recognize the actual implementation of a memory allocator (involving address arithmetic) and won't recognize that memory is allocated (although many wrappers will be recognized). In this case, the Code Analysis tool doesn't recognize that the memory was allocated and issues this warning. To suppress the false positive, use a `#pragma` warning on the line that precedes the opening brace `{` of the function body

Warning C28196

Article • 06/12/2023

The requirement is not satisfied. (The expression does not evaluate to true.)

This warning indicates that the function being analyzed has a `_notnull`, `_null`, `_drv_valueIs` or similar annotation on an `_out_` parameter or the return value, but the value returned is inconsistent with that annotation.

Remarks

Annotations like `_notnull` describe invariants about `_out_` parameters and return values, which serves both as documentation and as a sanity check for the author of the function. Warning C28196 indicates a mismatch between the annotations and the actual behavior of the function. The warning can be useful for discovering cases where a function might behave unexpectedly for certain input values. It's then up to the author to decide what the intended behavior of the function is and either adapt the annotations or the implementation accordingly.

Examples

The following function causes warning C28196 because it's annotated with `_Ret_notnull_` even though some code paths return a null pointer.

C++

```
#include <sal.h>

_Ret_notnull_
Item *get_item(_In_reads_(len) Item *items, size_t len, size_t index) {
    if (index >= len) {
        return nullptr;
    }

    return items + index;
}
```

To resolve this issue, refine the annotation to accurately reflect the function's behavior.

C++

```
#include <sal.h>

_WHEN_(index < len, _Ret_notnull_)
Item *get_item(_In_reads_(len) Item *items, size_t len, size_t index) {
    if (index >= len) {
        return nullptr;
    }

    return items + index;
}
```

See also

[Annotating function parameters and return values](#)

[Specifying When and Where an Annotation Applies](#)\

Warning C28197

Article • 10/07/2022

Possibly leaking memory

This warning is reported for both memory and resource leaks when the resource is potentially aliased to another location.

The *pointer* points to allocated memory or to another allocated resource that wasn't explicitly freed. This warning is usually due to inadequate annotations on the called function, although inadequate annotations on the calling function can also make this warning more likely.

This warning can be reported on function exit if an input argument has a `_drv_freesMem` or `_drv_aliasesMem` annotation. This warning typically indicates either a valid leak or that a function called by the current function needs more annotation.

In particular, the absence of the basic `_In_` and `_Out_` annotations make this warning fairly likely, although the `_drv_aliasesMem` and `_drv_freesMem` annotations might be needed as well. A false positive is a likely result of a missing `_In_` annotation.

Functions that take a pointer and alias it (thus avoiding a leak) should be annotated with `_drv_aliasesMem`. If you create a function that inserts an object into a global structure, or passes it to a system function that does that, you should add the `_drv_aliasesMem` annotation.

Functions that free memory should be annotated with `_drv_freesMem`. The major functions that free memory already have this annotation.

Example

The following code example generates this warning:

C++

```
char *p = (char *)malloc(10);
test(p); // does not save a copy of p
```

The following code example avoids this warning:

C++

```
char *p = (char *)malloc(10);
test(p); // does not save a copy of p
free(p);
```

Warning C28198

Article • 10/07/2022

Possibly leaking memory due to an exception.

This warning indicates that allocated memory isn't being freed after an exception is raised. The statement at the end of the path can raise an exception. The memory was passed to a function that might have saved a copy to be freed later.

This warning is similar to warning [C28197](#). The annotations that are recommended for use with warning [C28197](#) can also be used here.

Example

The following code example generates this warning:

```
C++  
  
char *p1 = new char[10];  
char *p2 = new char[10];  
  
test(p1); // does not save a copy of p  
  
delete[] p2;  
delete[] p1;
```

The following code example avoids this warning:

```
C++  
  
char *p1 = new char[10];  
char *p2 = NULL;  
  
test(p1); // does not save a copy of p  
try {  
    p2 = new char[10];  
} catch (std::bad_alloc *e) {  
    // just handle the throw  
    ;  
}
```

Warning C28199

Article • 10/07/2022

Using possibly uninitialized memory

This message indicates that the variable has had its address taken but no assignment to it has been discovered.

The specified variable is used without being explicitly initialized, but at some point its address was taken, indicating that it might be initialized invisibly to the Code Analysis tool.

This warning can be mistaken if the variable is initialized outside of the function.

The Code Analysis tool reports this warning on function exit if a parameter has an `_Out_` or `_Inout_` annotation and the variable isn't initialized.

Warning C28202

Article • 10/07/2022

Illegal reference to non-static member

This warning is reported when there's an error in the annotations.

Warning C28203

Article • 10/07/2022

Ambiguous reference to class member. Could be '*name1*' or '*name2*'

This warning is reported when there's an error in the annotations.

Warning C28204

Article • 10/07/2022

<function> : Only one of this overload and the one at '*filename*('line') are annotated for '*paramname*': both or neither must be annotated.

This warning is reported when there's an error in the annotations.

Warning C28205

Article • 10/07/2022

'function': `_Success_` or `_On_failure_` used in an illegal context

The `_Success_` and `_On_failure_` annotations can only be used on function return values.

Examples

This sample shows how the warning finds a misplaced SAL annotation:

C++

```
#include <sal.h>

// Oops, _Success_ is not valid in parameter lists, should be moved to
// return value.
bool GetValue( _Success_(return != false) _Out_ int *pInt, bool flag)
{
    if(flag) {
        *pInt = 5;
        return true;
    } else {
        return false;
    }
}
```

To correct the issue, move the SAL annotation to the return value:

C++

```
#include <sal.h>

_Success_(return != false)
bool GetValue(_Out_ int *pInt, bool flag)
{
    if(flag) {
        *pInt = 5;
        return true;
    } else {
        return false;
    }
}
```

See also

[Understanding SAL](#)

Warning C28206

Article • 10/07/2022

| <expression> : left operand points to a struct, use `->`

This warning is reported when the struct pointer dereference notation `->` was expected.

Warning C28207

Article • 10/07/2022

| <expression>: left operand is a struct, use .

This warning is reported when a struct dereference dot (.) was expected.

Warning C28208

Article • 10/07/2022

Function *function_name* was previously defined with a different parameter list at *file_name*(*line_number*). Some analysis tools will yield incorrect results

Remarks

This warning almost always accompanies [Compiler Warning \(level 1\) C4028](#). Both warn of a mismatch between the parameters of a function's declaration and its definition. However, this specific error indicates a more niche case than C4028. C28208 indicates not only that a mismatch exists, but that it also can cause issues with analysis tools. This warning most notably occurs when the mismatch exists between a `typedef` function pointer and the definition of that function. This warning is demonstrated in the example below.

Code analysis name: `FUNCTION_TYPE_REDECLARATION`

Example

The following code generates C28208. `test_type` declares `my_test1` and `my_test2` to take a `void*` parameter, but the definition of `my_test1` takes an `int*` parameter instead. `my_test2` avoids this issue because the definition parameters match the declaration parameters.

C++

```
// c28208_example.h
typedef void test_type(void*);
```

Warning C28209

Article • 10/07/2022

The declaration for symbol has a conflicting declaration

This warning indicates an incorrectly constructed annotation declaration. This warning should never occur in normal use.

Warning C28210

Article • 10/07/2022

Annotations for the `_On_failure_` context must not be in explicit pre context

Annotations `_On_failure_` must be explicitly or implicitly indicated in `__post` context, that is, to be applied after the function returns. Use `_drv_out` to ensure this annotation is indicated.

Warning C28211

Article • 10/07/2022

Static context name expected for SAL_context

This warning indicates that the operand to the `_Static_context_` annotation must be the name of a tool-defined context. This warning should never occur in normal use.

Warning C28212

Article • 10/07/2022

Pointer expression expected for annotation

This warning indicates that the numbered parameter to an annotation (not the function being annotated) is expected to be a pointer or array type, but some other type was encountered. The annotation needs to be corrected.

Warning C28213

Article • 02/22/2023

The `_Use_decl_annotations_` annotation must be used to reference, without modification, a prior declaration.

Remarks

`_Use_decl_annotations_` tells the compiler to use the annotations from an earlier declaration of the function. If it can't find an earlier declaration, or if the current declaration makes changes to the annotations, it emits this warning.

`_Use_decl_annotations_` also lets you remove all other annotations from the definition, and uses the declaration annotations for analysis of the function.

This diagnostic is frequently a side effect of refactoring or fixing other warnings by adjusting the annotations on a function. To fix the issue, use the same annotations at the other locations. To determine the correct set of annotations, look at the behavior in the function definition. In most cases, this behavior is intentional and the annotations to the function should reflect it. For more information on SAL annotations, see [Using SAL Annotations to reduce code defects](#).

It's important for the annotations to match between the declarations and the definition of a function. When the analysis tools analyze the call site of the function, they use the declaration annotations. If the declaration and definition don't match, the static analysis tools may produce incorrect results. When you fix this warning, it's common for your changes to have cascading effects as the tool reanalyzes the source with updated information.

If this diagnostic occurs because the analyzer couldn't find a previous declaration in the translation unit, the most likely cause is a missing `#include` directive. To resolve this issue when you intentionally don't include the header file, verify that the annotations in the declaration and definition match, and remove the `_Use_decl_annotations_` annotation. Be careful when you don't include a header file, as the two sets of annotations may get out of sync in the future.

Code analysis name: `BAD_USEHEADER`

Examples

The following code generates C28160. The `buffer` parameter annotation doesn't match between the two files.

From example.h:

C++

```
void addNullTerminate(_Out_writes_(n) char* buffer, int n);
```

From example.cpp:

C++

```
_Use_decl_annotations_
void addNullTerminate(_Out_writes_z_(n) char* buffer, int n)
{
    buffer[n] = '\0';
}
```

Examine the function definition to determine what the correct annotations should be. In this case, `_Out_writes_z_(n)` appears to be correct, so we move that annotation to the function declaration in the header file. This change resolves the issue because the annotations in the declaration and definition now match.

From example.h:

C++

```
void addNullTerminate(_Out_writes_z_(n) char* buffer, int n);
```

Now we can remove the `buffer` annotation on the definition to simplify future maintenance (although this step is optional).

From example.cpp:

C++

```
_Use_decl_annotations_
void addNullTerminate(char* buffer, int n)
{
    buffer[n] = '\0';
}
```

In real world code, it's not usually as clear which annotation is correct. For more information and guidance, see [using SAL Annotations to reduce code defects](#).

See also

[Rule sets for C++ code](#)

[Using SAL Annotations to reduce code defects](#)

[C28252](#)

[C28253](#)

[C28301](#)

Warning C28214

Article • 10/07/2022

Attribute parameter names must be p1...p9

This warning indicates that when you construct an annotation declaration, the parameter names are limited to p1...p9. This warning should never occur in normal use.

Warning C28215

Article • 10/07/2022

The typefix cannot be applied to a parameter that already has a typefix

Applying a `_typefix` annotation to a parameter that already has that annotation is an error. The `_typefix` annotations are used only in a few special cases. We don't expect this warning to be seen in normal use.

Warning C28216

Article • 10/07/2022

The `_Check_return_` annotation only applies to post-conditions for the specific function parameter.

The `_Check_return_` annotation has been applied in a context other than `__post`; it may need a `__post` (or `__drv_out`) modifier, or it may be placed incorrectly.

Warning C28217

Article • 10/07/2022

For function, the number of parameters to annotation does not match that found at file

The annotations on the current line and on the line in the message are inconsistent. This inconsistency shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in normal use.

Warning C28218

Article • 10/07/2022

For function parameter, the annotation's parameter does not match that found at file

The annotations on the current line and on the line in the message are inconsistent. This inconsistency shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in normal use.

Warning C28219

Article • 10/07/2022

Member of enumeration expected for annotation the parameter in the annotation

A parameter to an annotation is expected to be a member of the named `enum` type, and some other symbol was encountered; use a member of that `enum` type. This warning usually indicates an incorrectly coded annotation macro.

Warning C28220

Article • 10/07/2022

Integer expression expected for annotation '*annotation*'

This warning indicates that an integer expression was expected as an annotation parameter, but an incompatible type was used instead. It can be caused by trying to use a SAL annotation macro that doesn't fit the current scenario.

Example

C++

```
#include <sal.h>

// Oops, the _In_reads_ annotation takes an integer type but is being passed
// a pointer
void f(_In_reads_(last) const int *buffer, const int *last)
{
    for(; buffer < last; ++buffer)
    {
        //...
    }
}
```

In this example, the intent of the developer was to indicate that `buffer` would be read up to the `last` element. The `_In_reads_` annotation doesn't fix the scenario because it's used to indicate a buffer size in number of elements. The correct annotation is `_In_reads_to_ptr_`, which indicates the end of the buffer with a pointer.

If there was no `_to_ptr_` equivalent to the annotation used, then the warning could have been addressed by converting the `last` pointer into a size value with `(buffer - size)` in the annotation.

C++

```
#include <sal.h>

void f(_In_reads_to_ptr_(last) const int *buffer, const int *last)
{
    for(; buffer < last; ++buffer)
    {
        //...
    }
}
```


Warning C28221

Article • 10/07/2022

| String expression expected for the parameter in the annotation

This warning indicates that a parameter to an annotation is expected to be a string, and some other type was encountered. This warning usually indicates an incorrectly coded annotation macro, and we don't expect it to be seen in normal use.

Warning C28222

Article • 10/07/2022

Yes, _No_, or _Maybe_ expected for annotation

This warning indicates that a parameter to an annotation is expected to be one of the symbols `_Yes_`, `_No_`, or `_Maybe_`, and some other symbol was encountered. This warning usually indicates an incorrectly coded annotation macro.

Warning C28223

Article • 10/07/2022

Did not find expected Token/identifier for annotation, parameter

This warning indicates that a parameter to an annotation is expected to be an identifier, and some other symbol was encountered. This warning usually indicates an incorrectly coded annotation macro. The use of C or C++ keywords in this position will cause this error.

Warning C28224

Article • 10/07/2022

Annotation requires parameters

This warning indicates that the named annotation is used without parameters and at least one parameter is required. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in normal use.

Warning C28225

Article • 10/07/2022

Did not find the correct number of required parameters in annotation

This warning indicates that the named annotation is used with the incorrect number of parameters. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28226

Article • 10/07/2022

Annotation cannot also be a PrimOp (in current declaration)

This warning indicates that the named annotation is being declared as a PrimOp, and also was previously declared as a normal annotation. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28227

Article • 10/07/2022

Annotation cannot also be a PrimOp (see prior declaration)

This warning indicates that the named annotation is being declared as an ordinary annotation, and also was previously declared as a PrimOp. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28228

Article • 10/07/2022

Annotation parameter: cannot use type in annotations

This warning indicates that a parameter is of type that isn't supported. Annotations can only use a limited set of types as parameters. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28229

Article • 10/07/2022

Annotation does not support parameters

This warning indicates that an annotation was used with a parameter when the annotation is declared without parameters. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28230

Article • 10/07/2022

The type of parameter has no member.

This warning indicates that an argument to an annotation attempts to access a member of a `struct`, `class`, or `union` that doesn't exist. This warning will also be emitted if a parameter tries to call a member function of the object.

Example

C++

```
#include <sal.h>

struct MyStruct
{
    //...
    int usefulMember;
};

// Oops, the name of the member is spelled wrong so it will not be found
void f(_Out_writes_(value.usefulmember) int *buffer, MyStruct value)
{
    for(int i = 0 ; i < value.usefulMember; i++)
    {
        buffer[i] = i;
        //...
    }
}
```

In this example, the spelling just needs to be corrected.

C++

```
void f(_Out_writes_(value.usefulMember) int *buffer, MyStruct value)
{
    for(int i = 0 ; i < value.usefulMember; i++)
    {
        buffer[i] = i;
        //...
    }
}
```

Warning C28231

Article • 10/07/2022

Annotation is only valid on array

This warning indicates that an argument to an annotation should be an array, and some other type was encountered.

Warning C28232

Article • 10/07/2022

Pre, _Post_, or _Deref_ not applied to any annotation

This warning indicates that a `_Pre_`, `_Post_`, or `_Deref_` operator appears in an annotation expression without a subsequent functional annotation; the modifier was ignored, but this warning indicates an incorrectly coded annotation.

Warning C28233

Article • 10/07/2022

Pre, post, or deref applied to a block

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Warning C28234

Article • 10/07/2022

At expression does not apply to current function

This warning indicates that the value of an `_At_` expression doesn't identify an accessible object.

Warning C28235

Article • 10/07/2022

The function cannot stand alone as an annotation

This warning indicates that an attempt was made to use a function that wasn't declared to be an annotation in an annotation context. This includes using a primitive operation (PrimOp) in a standalone context. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28236

Article • 10/07/2022

The annotation cannot be used in an expression

This warning indicates that an attempt was made to use a function declared to be an annotation in an expression context. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28237

Article • 10/07/2022

The annotation on parameter is no longer supported

An internal error has occurred in the PREfast model file. This warning shouldn't occur in typical use.

Warning C28238

Article • 10/07/2022

The annotation on parameter has more than one of value, stringValue, and longValue. Use paramn=xxx

An internal error has occurred in the PREFast model file. This warning shouldn't occur in typical use.

Warning C28239

Article • 10/07/2022

The annotation on parameter has both value, stringValue, or longValue; and paramn=xxx. Use only paramn=xxx

An internal error has occurred in the PREFast model file. This warning shouldn't occur in typical use.

Warning C28240

Article • 10/07/2022

The annotation on parameter has param2 but no param1

An internal error has occurred in the PREfast model file. This warning shouldn't occur in typical use.

Warning C28241

Article • 10/07/2022

The annotation for function on parameter is not recognized

An unrecognized annotation name was used. This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28243

Article • 10/07/2022

The annotation for function on parameter requires more dereferences than the actual type annotated allows

The number of `__deref` operators on an annotation is more than the number of levels of pointer defined by the parameter type. Correct this warning by changing either the number in the annotation or the pointer levels of the parameter referenced.

Warning C28244

Article • 10/07/2022

The annotation for function has an unparseable parameter/external annotation

This warning shouldn't be possible if the standard macros are being used for annotations. We don't expect this warning to be seen in typical use.

Warning C28245

Article • 10/07/2022

The annotation for function annotates 'this' on a non-member-function

An internal error has occurred in the PREfast model file. This warning shouldn't occur in typical use.

Warning C28246

Article • 10/07/2022

The annotation for function '*name*' - parameter '*parameter*' does not match the type of the parameter

A `__deref` operator was applied to a non-pointer type when creating an annotation.

Warning C28250

Article • 10/07/2022

Inconsistent annotation for function: the prior instance has an error.

Note: There are several prototypes for this function. This warning compares the first prototype with instance number <number>.

If a declaration is made using a `typedef`, the line where the `typedef` appears is more useful than the line of the declaration.

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match the ones on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield many unmatched low-level annotations. It's best to compare the declaration and definition source code to make sure that they're the same. (Trivial differences in the order of the annotations aren't reported.)

The comparison is always between the first declaration found and the current one. If there are more declarations, each declaration is checked pairwise. It's currently not possible to do a comparison other than in pairs, although it's possible to identify that there are more than two declarations/definitions. The *text* field above contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you don't need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

Warning C28251

Article • 10/07/2022

Inconsistent annotation for function: this instance has an error

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match the ones on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield many unmatched low-level annotations. It's best to compare the declaration and definition source code to make sure that they're the same. (Trivial differences in the order of the annotations aren't reported.)

The comparison is always between the first declaration found and the current one. If there are more declarations, then each declaration is checked in groups of two. It's currently not possible to do a comparison other than in pairs, although it's possible to identify that there are more than two declarations/definitions. The *text* field above contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you don't need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

Warning C28252

Article • 10/07/2022

Inconsistent annotation for function: parameter has another annotation on this instance

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match the ones on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield many unmatched low-level annotations. It's best to compare the declaration and definition source code to make sure that they're the same. (Trivial differences in the order of the annotations aren't reported.)

The comparison is always between the first declaration found and the current one. If there are more declarations, then each declaration is checked in groups of two. It's currently not possible to do a comparison other than in pairs, although it's possible to identify that there are more than two declarations/definitions. The error message contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you don't need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

Warning C28253

Article • 10/07/2022

Inconsistent annotation for function: parameter has another annotations on this instance

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match the ones on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield many unmatched low-level annotations. It's best to compare the declaration and definition source code to make sure that they're the same. (Trivial differences in the order of the annotations aren't reported.)

The comparison is always between the first declaration found and the current one. If there are more declarations, then each declaration is checked in groups of two. It's currently not possible to do a comparison other than in pairs, although it's possible to identify that there are more than two declarations/definitions. The error message contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you don't need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

Warning C28254

Article • 10/07/2022

`dynamic_cast<>()` is not supported in annotations

The C++ `dynamic_cast` operator can't be used in annotations.

Warning C28262

Article • 10/07/2022

A syntax error in the annotation was found in function '*function*' for annotation '*name*'

Warning C28263

Article • 10/07/2022

A syntax error in a conditional annotation was found for Intrinsic annotation

The Code Analysis tool reports this warning when the return value for the specified function has a conditional value. This warning indicates an error in the annotations, not in the code being analyzed. If the function declaration is in a header file, the annotation should be corrected so that it matches the header file.

The result list for the function and parameter specified has multiple unconditional values.

Typically, this warning indicates that more than one unconditional `_Null_` or `_drv_valueIs` annotations have been used to specify a result value.

Warning C28267

Article • 10/07/2022

A syntax error in the annotations was found for function '*function-name*', for annotation '*annotation-name*': *reason*.

Warning C28272

Article • 10/07/2022

The annotation for function '*function-name*', parameter '*parameter-name*' when examining '*clue*' is inconsistent with the function declaration

This warning indicates an error in the annotations, not in the code that is being analyzed. The annotations appearing on a function definition are inconsistent with the ones appearing on a declaration. The two annotations should be resolved to match.

Warning C28273

Article • 10/07/2022

For function '*function-name*', the clues are inconsistent with the function declaration

This warning indicates an error in the annotations, not in the code that is being analyzed. The annotations appearing on a function definition are inconsistent with the ones appearing on a declaration. The two annotations should be resolved to match.

Warning C28275

Article • 10/07/2022

The parameter to `_Macro_value_` is null

This warning indicates that there's an internal error in the model file, not in the code being analyzed. The `macroValue` function was called without a parameter.

Warning C28278

Article • 10/07/2022

Function name appears with no prototype in scope.

This warning typically indicates that a `_deref` is needed to apply the `_return` annotation to the value returned.

The Code Analysis tool reports this warning when a function without a declaration was called, and the analysis that can be performed is limited because the declaration contains important information.

The C language permits (but discourages) the use of a function for which no prototype has been declared. A function definition or a function declaration (prototype) should appear before the first use of the function. This warning indicates that a function without a declaration was called, and the analysis that can be performed is limited because declaration contains important information. If the function declaration contains annotations, the function declaration is even more useful to the Code Analysis tool.

Warning C28279

Article • 10/07/2022

| For symbol, a 'begin' was found without a matching 'end'

The annotation language supports a begin and end (`{` and `}` in C) construct, and the pairing has gotten unbalanced. This situation can be avoided if the macros are used.

Warning C28280

Article • 10/07/2022

| For symbol, an 'end' was found without a matching 'begin'

The annotation language supports a begin and an end (`{` and `}`) in C construct, and the pairing has gotten unbalanced. This situation can be avoided if the macros are used.

Warning C28282

Article • 10/07/2022

Format Strings must be in preconditions

This warning indicates that a `__drv_formatString` annotation is found, which isn't in a `_Pre_` (`__drv_in`) annotation (function parameters are by default `_Pre_`). Check whether the annotation used in an explicit block with a `_Post_` (`__drv_out`) annotation. If so, remove the annotation from any enclosing block that has put it in a `_Post_` context.

Warning C28283

Article • 10/07/2022

| For symbol, the specified size specification is not yet supported

The warning indicates that the size specification "sentinel" annotation received a value other than zero. Essentially, the caller tried to say that the string is terminated by a character other than binary zero.

Warning C28284

Article • 10/07/2022

| For symbol, Predicates are currently not supported for non-function symbols

This warning indicates that a conditional annotation (predicate, `__drv_when`) was found on something other than a function.

Warning C28285

Article • 10/07/2022

For function '*function-name*', syntax error in '*annotation*'

Remarks

The Code Analysis tool reports this warning for syntax errors in the SAL annotation. The SAL parser will recover by discarding the malformed annotation. Double check the documentation for the SAL annotations being used and try to simplify the annotation. You shouldn't use implementation layer annotations such as `_declspec("SAL_begin")` directly. If you're using that layer, then change them into higher layers such as `_In_/_Out_/_Ret_`. For more information, see [Annotating Function Parameters and Return Values](#).

Example

The following code generates this warning. The argument `(2,n)` is malformed and will cause a C28285 warning after the `_Out_writes_z_` macro is expanded.

C++

```
void example_func(_Out_writes_z_((2,n)) char* buffer, int n)
{
    buffer[n] = '\0';
}
```

The following code remediates this warning by correcting the malformed annotation

C++

```
void example_func(_Out_writes_z_(n) char* buffer, int n)
{
    buffer[n] = '\0';
}
```

Warning C28286

Article • 10/07/2022

For function, syntax error near the end

The Code Analysis tool reports this warning when a probable error is encountered in the model file. A few source file errors can also cause such errors.

Warning C28287

Article • 10/07/2022

For function, syntax Error in _At_() annotation (unrecognized parameter name)

The Code Analysis tool reports this warning when the `SAL_at (__drv_at)` annotation is used and the parameter expression can't be interpreted in the current context. Reasons might include using a misspelled parameter or member name, or a misspelling of "return" or "this" keywords.

Warning C28288

Article • 10/07/2022

For function, syntax Error in _At_() annotation (invalid parameter name)

The Code Analysis tool reports this warning when the `SAL_at (__drv_at)` annotation is used and the parameter expression can't be interpreted in the current context. Reasons might include using a misspelled parameter or member name, or a misspelling of "return" or "this" keywords.

Warning C28289

Article • 10/07/2022

| For function: ReadableTo or WritableTo did not have a limit-spec as a parameter

The Code Analysis tool reports this warning when the function/parameter annotation is miscoded as noted.

Warning C28290

Article • 10/07/2022

The annotation for function contains more Externals than the actual number of parameters

The Code Analysis tool reports this warning when the annotation for the function contains more Externals than the actual number of parameters.

Warning C28291

Article • 10/07/2022

Post null/notnull at deref level 0 is meaningless for function 'x' at param '*number*'

The Code Analysis tool reports this warning when the post condition of a dereference level-zero parameter is specified to have a null/non-null property. This error occurs because a value at dereference level zero can't change.

Warning C28300

Article • 10/07/2022

```
<parameter_name>: Expression operands of incompatible types for operator  
<operator_name>
```

This warning fires a SAL annotation contains an expression containing incompatible types.

Example

C++

```
union MyUnion  
{  
    int length;  
    //...  
};  
  
// Oops, int and MyUnion are not compatible with the + operator.  
void f(_In_reads_(10 + value) int *buffer, MyUnion value)  
{  
    for(int i = 0 ; i < (10 + value.length); i++)  
    {  
        //...  
    }  
}
```

In the previous example, the developer forgot to access the appropriate member variable. In other cases, you may need to fix the error with an explicit cast.

C++

```
void f(_In_reads_(10 + value.length) int *buffer, MyUnion value)  
{  
    for(int i = 0 ; i < (10 + value.length); i++)  
    {  
        //...  
    }  
}
```

Warning C28301

Article • 10/07/2022

No annotations for first declaration of '*function*'.*'note1'* See '*filename*('i*line*'). '*note2*'

This warning is reported when annotations weren't found at the first declaration of a given function.

Warning C28302

Article • 10/07/2022

For C++ reference-parameter '*parameter_name*', an extra `_Deref_` operator was found on '*annotation*'.

This warning is reported when an extra level of `_Deref_` is used on a parameter of a reference type such as `T &a`. A common mistake when using SAL1 annotations is to use `_deref` on a reference type. Reference types are understood by SAL, so all annotations are already applied to the underlying type. It's typically not an issue in SAL2 because the free-floating `_deref` annotation was removed. If you intend to apply an annotation to a subtype, then you should instead use the SAL2 `_AT_` or `_Outref_` annotations.

Example

C++

```
// Oops, trying to apply __elem_writableTo to the pointer being referenced
void f( __deref __elem_writableTo(size) int *& buffer, int size);

void func()
{
    int buffer[100] = {};
    int *pbuffer = buffer;
    f(pbuffer, 100);
}
```

To address the issue, update to SAL2 annotations:

C++

```
// Fix warning by switching to SAL2 syntax which has annotations that better
// describe what the function does.
void f( _Outref_result_buffer_(size) int *& buffer);
```

See also

[Understanding SAL](#)

Warning C28303

Article • 10/07/2022

For C++ reference-parameter <parameter_name>, an ambiguous `_Deref_` operator was found on '*annotation*'.

This warning similar to warning C28302 and is reported when an extra level of `_Deref_` is used on a parameter.

SAL2 doesn't require the use of an extra level of `_Deref_` when dealing with reference parameters. This particular annotation is ambiguous as to which level of dereference is intended to be annotated. It may be necessary to use `_At_` to reference the specific object to be annotated.

Example

The following code generates this warning because the use of `__deref_out_ecount(n)` is ambiguous:

C++

```
void ref(__deref_out_ecount(n) int **&buff, int &n)
```

The above annotation could be interpreted either as:

- a reference to an array (of n) pointers to integers (SAL1 interpretation)
- a reference to a pointer to an array (of n) integers (SAL2 interpretation)

Either of the following changes can correct this warning:

C++

```
void ref(_Out_writes_(n) int **&buff, int &n)
// or
_At_(*buff), _Out_writes_(n)) void ref(int **&buff, int &n)
```

Warning C28304

Article • 10/07/2022

For C++ reference-parameter <parameter_name>, an improperly placed `_Notref_` operator was found applied to '*token*'.

The `_Notref_` operator should only be used in special circumstances involving C++ reference parameters and only in system-provided macros. It must be immediately followed by a `_Deref_` operator or a functional annotation.

Warning C28305

Article • 10/07/2022

An error while parsing '*token*' was discovered.

This warning is reported when the expression containing the specified token is ill-formed.

Warning C28306

Article • 05/29/2023

The annotation on parameter is obsolescent

Use `_String_length_` with the appropriate SAL2 annotation instead.

See also

[Intrinsic Functions](#)

Warning C28307

Article • 05/29/2023

The annotation on parameter is obsolescent

Use `_String_length_` with the appropriate SAL2 annotation instead.

See also

[Intrinsic Functions](#)

Warning C28308

Article • 03/23/2023

The format list argument position specified by the annotation is incorrect.

Remarks

This warning indicates a `_*_format_strings_param(position)` SAL annotation is specifying an invalid position for the first parameter to the format string. The annotation helps the checker verify `printf` style formatting strings passed to the function. Other format string validity checks that rely on this annotation won't run as a result of this warning.

The `_*_format_strings_param(position)` SAL annotation is attached to the formatting string argument. `position` must be in one of these forms:

- An identifier, which is taken as the first argument to the format string. When the identifier isn't the name of a parameter to the function, a warning is emitted.
- A positive integer offset relative to the format-string parameter where `1` is the next parameter. When the offset is out of bounds for the parameters, a warning is emitted.
- The value `0`, which is interpreted as the `...` parameter. When the function isn't variadic, a warning is emitted.

One limitation of this check, is that it's run at the function call site and not at the declaration. This limitation is a side effect of the lazy evaluation of SAL annotations.

Examples

In this example, there's a specialized function for logging coordinates. The params annotation specifies the `...` parameter, which doesn't exist.

C++

```
void LogCoordinate(_Printf_format_string_params_(0) _In_ char *format, int
x, int y);

void func(int x, int y)
{
    LogCoordinate("(%d, %d)", x, y);
}
```

This issue is fixed by changing the annotated position to `x` or `1`. To determine the correct value for your code, check the behavior of the called function.

C++

```
void LogCoordinate(_Printf_format_string_params_(1) _In_ char *format, int
x, int y);

void func(int x, int y)
{
    LogCoordinate("(%d, %d)", x, y);
}
```

See also

[Annotating function parameters and return values](#)

Warning C28309

Article • 10/07/2022

<parameter_name>: Annotation operands must be integer/enum/pointer types.

Void operands and C++ overloaded operators are not supported. Floats are approximated as integers. Types: '*typelist*'.

You've tried to use a void or a function in an annotation expression, and Code Analysis can't handle it. This error typically occurs when an `operator==` that's implemented as a function is used, but other cases may also occur. Examine the types in <*typelist*> for clues about what's wrong.

Warning C28310

Article • 10/07/2022

The <annotation_name> annotation on '*function*' '*parameter*' has no SAL version.

All SAL annotations used in source code should have an annotation version applied by the use of SAL_name. This issue needs to be corrected in the macro definition.

This warning is reported only once per declaration. Inspect the rest of the declaration for more obsolete SAL.

See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C28311

Article • 10/07/2022

The <annotation_name> annotation on '*function*' '*parameter*' is an obsolescent version of SAL.

The annotation is an old version and should be updated to the equivalent [SAL2](#). This warning isn't emitted if a prior inconsistent annotation warning has been emitted, and is reported only once per declaration. Inspect the rest of the declaration for more obsolete SAL.

See also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C28312

Article • 10/07/2022

The <annotation_name> annotation on the repeated declaration of '*function*' '*parameter*' is an obsolescent version of SAL.

The annotation is an old version and should be updated to the equivalent [SAL2](#). This warning isn't emitted if a prior inconsistent annotation warning has been emitted, and is reported only once per declaration. Inspect the rest of the declaration for more obsolete SAL.

See also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

Warning C28350

Article • 10/07/2022

The annotation '*annotation*' describes a situation that is not conditionally applicable.

Usually this warning is generated when an annotation is applied where the C/C++ type is being inspected.

Warning C28351

Article • 10/07/2022

The annotation '*annotation*' describes where a dynamic value (a variable) cannot be used in the condition.

This warning is reported when an annotation is applied where the C/C++ type is being inspected.

The expression in the `_When_` should evaluate to a constant. The `_When_` is ignored.

Warning C33001

Article • 10/07/2022

VARIANT 'var' was cleared when it was uninitialized (expression 'expr')

Remarks

This warning is triggered when an uninitialized VARIANT is passed to an API such as VariantClear that expects an initialized VARIANT.

Code analysis name: VARIANTCLEAR_UNINITIALIZED

Example

The following sample code causes warning C33001:

C++

```
#include <Windows.h>

HRESULT foo(bool some_condition)
{
    VARIANT var;

    if (some_condition)
    {
        //...
        VariantInit(&var);
        //...
    }

    VariantClear(&var);      // C33001
}
```

These warnings are corrected by ensuring VariantClear is called only for a properly initialized VARIANT:

C++

```
#include <Windows.h>

HRESULT foo(bool some_condition)
{
    VARIANT var;
```

```
if (some_condition)
{
    //...
    VariantInit(&var);
    //...
    VariantClear(&var);      // C33001
}
}
```

See also

[C33004](#) [C33005](#)

Warning C33004

Article • 10/07/2022

VARIANT 'var', which is marked as `_Out_` was cleared before being initialized
(expression 'expr')

Remarks

This warning is triggered when a `VARIANT` parameter with `_Out_` SAL annotation may not have been initialized on input, and is then passed to an API such as `VariantClear` that expects an initialized `VARIANT`.

Code analysis name: `VARIANTCLEAR_UNINITOUTPARAM`

Example

The following sample code causes warning C33004:

```
C++  
  
#include <Windows.h>  
  
void t2(_Out_ VARIANT* pv)  
{  
    // .....  
    VariantClear(pv);    // C33004  
    // .....  
}
```

These warnings are corrected by ensuring `VariantClear` is called only for a properly initialized `VARIANT`:

```
C++  
  
#include <Windows.h>  
  
void t2(_Out_ VARIANT* pv)  
{  
    VariantInit(pv);  
    // .....  
    VariantClear(pv);    // OK  
    // .....  
}
```

See also

[C33001](#) [C33005](#)

Warning C33005

Article • 10/07/2022

VARIANT 'var' was provided as an `_In_` or `_InOut_` parameter but was not initialized
(expression 'expr')

Remarks

This warning is triggered when an uninitialized VARIANT is passed to a function as an input-only or input/output parameter. For example, a pass-by-reference parameter without an `_out_` SAL annotation.

Code analysis name: `VARIANTCLEAR_UNINITFUNCPARAM`

Example

The following sample code causes warning C33005:

C++

```
#include <Windows.h>

void bar(VARIANT* v); // v is assumed to be input/output

void foo()
{
    VARIANT v;
    bar(&v);           // C33005
    // .....
    VariantClear(&v); // OK, assumed to be initialized by bar
}
```

To correct these warnings, initialize the VARIANT before passing it to a function as input-only or input/output.

C++

```
#include <Windows.h>

void bar(VARIANT* v); // v is assumed to be input/output

void foo()
{
    VARIANT v;
```

```
VariantInit(&v);
bar(&v);           // OK
// .....
VariantClear(&v); // OK, assumed to be initialized by bar
}
```

See also

[C33001](#) [C33004](#)

Warning C33010

Article • 05/26/2023

Unchecked lower bound for enum 'enum' used as index.

This warning is triggered if an enum is both used as an index into an array and isn't checked on the lower bound.

Remarks

Code using enumerated types as indexes for arrays will often check for the upper bound in order to ensure the index isn't out of range. Because an enum variable is signed by default, it can have a negative value. A negative array index can allow arbitrary memory to be read, used, or even executed.

Code analysis name: UNCHECKED_LOWER_BOUND_FOR_ENUMINDEX

Example

The following code generates this warning. `idx` is used as an index to access `functions`, but the lower bound is never checked.

C++

```
typedef void (*PFN)();

enum class Index
{
    Zero,
    One,
    Two,
    Three,
    Max
};

void foo(Index idx, PFN(&functions)[5])
{
    if (idx > Index::Max)
        return;

    auto pfn = functions[static_cast<int>(idx)];
    if (pfn != nullptr)
        (*pfn)();
}
```

The following code remediates this warning by checking the lower bound as well to ensure `idx` isn't negative:

```
C++  
  
typedef void (*PFN)();  
  
enum class Index  
{  
    Zero,  
    One,  
    Two,  
    Three,  
    Max  
};  
  
void foo(Index idx, PFN(&functions)[5])  
{  
    if (idx < Index::Zero || idx > Index::Max)  
        return;  
  
    auto pfn = functions[static_cast<int>(idx)];  
    if (pfn != nullptr)  
        (*pfn)();  
}
```

Alternatively, the issue can be fixed by choosing an underlying type for `Index` that is unsigned. Because an unsigned value is always positive, it is sufficient to only check the upper bound.

```
C++  
  
typedef void (*PFN)();  
  
enum class Index : unsigned int  
{  
    Zero,  
    One,  
    Two,  
    Three,  
    Max  
};  
  
void foo(Index idx, PFN(&functions)[5])  
{  
    if (idx > Index::Max)  
        return;  
  
    auto pfn = functions[static_cast<unsigned int>(idx)];  
    if (pfn != nullptr)
```

```
(*pfn)();  
}
```

See also

[C33011](#)

Warning C33011

Article • 10/07/2022

Unchecked upper bound for enum 'enum' used as index.

Remarks

This warning is triggered for an enum that is used as an index into an array, if the lower bound is checked for its value, but not the upper bound.

Code analysis name: `UNCHECKED_UPPER_BOUND_FOR_ENUMINDEX`

Example

Code that uses enumerated types as indexes for arrays must check the enum value for both lower and upper bounds. If the enum value is checked only for the lower bound before used to index into an array of values (or an array of function pointers), then it can allow arbitrary memory to be read, used, or even executed.

C++

```
typedef void (*PFN)();

enum class Index
{
    Zero,
    One,
    Two,
    Three,
    Max
};

void foo(Index idx, PFN(&functions)[5])
{
    if (idx < Index::Zero)
        return;

    auto pfn = functions[static_cast<int>(idx)];      // C33011
    if (pfn != nullptr)
        (*pfn)();
    // .....
}
```

These warnings are corrected by checking the index value for upper bound as well:

C++

```
typedef void (*PFN)();

enum class Index
{
    Zero,
    One,
    Two,
    Three,
    Max
};

void foo(Index idx, PFN(&functions)[5])
{
    if (idx < Index::Zero || idx > Index::Max)
        return;

    auto pfn = functions[static_cast<int>(idx)];      // OK
    if (pfn != nullptr)
        (*pfn)();
    // .....
}
```

See also

[C33010](#)

Warning C33020

Article • 10/07/2022

Likely incorrect HRESULT usage detected.

Remarks

This warning is a high-confidence indication that an HRESULT-returning function returns FALSE.

Code analysis name: `HRESULT_LIKELY_INCORRECT_USAGE`

Example

The following sample code causes warning C33020:

C++

```
#include <Windows.h>

HRESULT foo()
{
    // .....
    return FALSE; // C33020
}
```

These warnings are corrected by using the proper HRESULT value:

C++

```
#include <Windows.h>

HRESULT foo()
{
    // .....
    return E_FAIL; // OK
}
```

See also

[C33022](#)

Warning C33022

Article • 10/07/2022

Potentially incorrect HRESULT usage detected (low confidence)

Remarks

This warning is a low-confidence indicator for a function that returns HRESULT, that there's a `FALSE` that is either eventually returned, or it's assigned to a variable that is returned.

Code analysis name: `HRESULT_USAGE_LOW_CONFIDENCE`

Example

The following sample code causes warning C33022:

```
C++  
  
#include <Windows.h>  
  
#define CHECK_RESULT(X) X ? S_OK : FALSE;  
#define RETURN_RESULT(X) return CHECK_RESULT(X)  
  
HRESULT foo()  
{  
    // .....  
    RETURN_RESULT(FALSE);    // C33022  
}
```

These warnings are corrected by using proper HRESULT value:

```
C++  
  
#include <Windows.h>  
  
#define CHECK_RESULT(X) X ? S_OK : E_FAIL;  
#define RETURN_RESULT(X) return CHECK_RESULT(X)  
  
HRESULT foo()  
{  
    // .....  
    RETURN_RESULT(FALSE);    // OK  
}
```

See also

[C33020](#)