Flow Analysis

This topic covers how to analyze the bugs reported by Flow Analysis

In this section:

Using Flow Analysis

This topic explains how to run Flow Analysis to expose bugs such as use of uninitialized memory, null pointer dereferencing, division by zero, memory and resource leaks.

> **Important**
>
> An optional Flow Analysis license (with C++test Server Edition) is required to use Flow Analysis.

Running Flow Analysis

Flow Analysis is a new type of static analysis technology that uses several analysis techniques, including simulation of application execution paths, to identify paths that could trigger runtime defects. Defects detected include use of uninitialized memory, null pointer dereferencing, division by zero, memory and resource leaks.

Since this analysis involves identifying and tracing complex paths, it exposes bugs that typically evade syntax-based static code analysis and unit testing, and would be difficult to find through manual testing or inspection. Flow Analysis' ability to expose bugs without executing code is especially valuable for users with legacy code bases and embedded code (where runtime detection of such errors is not effective or possible).

> **Analyzing Headers**
>
> C++test does not directly analyze headers unless they are included by a source file under test. See How do I analyze header files/what files are analyzed? for details.

> **Analyzing Template Functions**
>
> C++test does perform static analysis of instantiated function templates and instantiated members of class templates.
> See Support for Template Functions for details.

The general procedure for running Flow Analysis is:

1. Identify or create a Test Configuration with your preferred Flow Analysis standard analysis settings.
   - For a description of preconfigured Test Configurations, see Built-in Test Configurations.
   - For general procedures related to configuring and sharing Test Configurations, see the Configuring Test Configurations and Rules for Policies. For details on C++test-specific Flow Analysis options, see Static Tab Settings: Defining How Static Analysis is Performed.

2. Start the test using the preferred Test Configuration.
    - For details on testing from the GUI, see Testing from the GUI.
    - For details on testing from the command line, see Testing from the Command Line Interface.
3. Review and respond to the results.
    - For details, see Reviewing Flow Analysis Static Analysis Results.
4. (Optional) Fine-tune Flow Analysis settings as needed.
    - For details, see Customizing Flow Analysis Static Analysis.

Configuring Batch-Mode Flow Analysis Analysis with cpptestcli

Regularly-schedule batch-mode Flow Analysis analysis should simply execute a built-in or custom Test Configuration that analyzes your project according to the Flow Analysis rules important to your team.

For example:

- `cpptestcli -data  /path/to/workspace  -resource "ProjectToTest" -config team://DataFlowAnalysis -publish`

See Testing from the Command Line Interface for more details on configuring batch-mode tests.

Running Flow Analysis in Incremental Mode

By default, Flow Analysis performs a complete analysis of the scope it is run on. This can take considerable time when running on large code bases.

The most common way of performing Flow Analysis analysis is to run nightly tests on a single code base that changes slightly from day to day. Its incremental analysis mode is designed to reduce the time required to run analysis in this typical scenario. With incremental analysis mode, Flow Analysis memorizes important analysis data during the initial run, then reuses it during the subsequent runs—rerunning analysis only for parts of the code that have been modified or are tightly connected to the modified code.

When using incremental analysis, remember that:

- The initial run of Flow Analysis may be slightly slower than running without incremental analysis. This is because Flow Analysis in addition to performing a complete analysis of the code base, Flow Analysis saves data to be reused in subsequent runs.
- Disk space is required to store the necessary data.

Incremental Mode can be enabled using the controls in the Test Configuration manager's Static tab. The available controls are detailed in Configuring Flow Analysis Analysis Options.

Running Flow Analysis with Swapping of Analysis Data Enabled

Swapping of analysis data mode is enabled by default. In this mode, analysis data is written to disk. Swapping of analysis data uses the same persistent storage and is done in a similar process as incremental analysis. If analysis is run on a large project, the analysis data that represents a semantical model of the analyzed source code may consume all the memory available for running Flow Analysis. If this occurs, Flow Analysis will remove from memory parts of the analysis data that are not currently necessary and reread it from disk later.

In general, we recommend running C++test in a large JVM heap configured with the Xmx JVM option. This is to minimize swapping, which results in greater performance.If sufficient memory is available, swapping of analysis data may be disabled, which may speed up code analysis. For information on how to disable swapping of analysis data please refer to Enable swapping of analysis data to disk bullet in the Performance Tab Options topic.

Introducing Built-in Flow Analysis Test Configurations

Flow Analysis Fast

The standard Flow Analysis Test Configuration includes all the Flow Analysis rules—except security rules and those that require customization. The fast configuration uses "Shallowest" depth of analysis and runs faster than the standard and aggressive configurations (see descriptions below). The fast configuration finds a moderate amount of problems and prevents violation number explosion. We recommend using this configuration to enforce Flow Analysis rules for any new project. If more rigorous analysis is required, consider switching to Flow Analysis Standard.

Flow Analysis Standard

The "Flow Analysis Standard" Test Configuration includes all the Flow Analysis rules--except those that require customization. This configuration performs deeper analysis and may find more bugs than "Flow Analysis Fast". Standard also requires more time to run. We recommend switching to this configuration when finding more bugs is a higher priority. If you need to redefine some analysis parameters or reconfigure any of the rules, you can use this Test Configuration as a starting point.

Flow Analysis Aggressive

We recommend the "Flow Analysis Aggressive" Test Configuration if you want C++test to report a violation for any suspect code. This configuration encourages Flow Analysis to report a violation any time that it suspects a problem—even in cases in which a false positive may be reported. Applying this configuration will result in more bugs being reported, but it can also increase the number of false alarms.

Understanding Flow Analysis' Analysis Scope

Flow Analysis performs interprocedural analysis that can cross boundaries of functions, classes, namespaces and compilation units within the scope of files selected for analysis. If there is a function call and the called function is defined in a different file, Flow Analysis will be able to trace the path inside the function—as long as the file containing it is included into analysis scope. Because of this, the precision of analysis highly depends on the scope.

Below are general guidelines for defining the scope of analysis (selecting which files are to be analyzed in a single session):

- Never put independent projects in the same analysis scope. Different projects can have different definitions of a type with the same fully-qualified name or a function with specific signature. If there is a call to a function that is defined multiple times, Flow Analysis may be unable to trace the call into the correct function.
- If you want to analyze compilation unit A and it depends on compilation unit B, then it is typically recommended to include both A and B in the analysis scope so that Flow Analysis can take into account any necessary information from unit B when analyzing A. The same pattern applies to dependent projects. Of course, the more files that are included in the analysis, the longer the analysis will take. Flow Analysis does allow analysis on just a single file. However, such limited analysis is likely to be less precise.

Reviewing Flow Analysis Static Analysis Results

This topic covers how to analyze the bugs reported by Flow Analysis.

Accessing Results

Flow Analysis findings are reported alongside static code analysis violations in Quality Tasks view. Flow Analysis content and format, however, is significantly different than that used for static code analysis violations. Results are organized into a prioritized task list that can be viewed by rule category or by severity. To view results by severity, open the Quality Tasks pull-down menu and choose **Show> Details**.

For tests run in the GUI, rule violations are reported in the **Fix Static Analysis Violations** category of the Quality Tasks view.

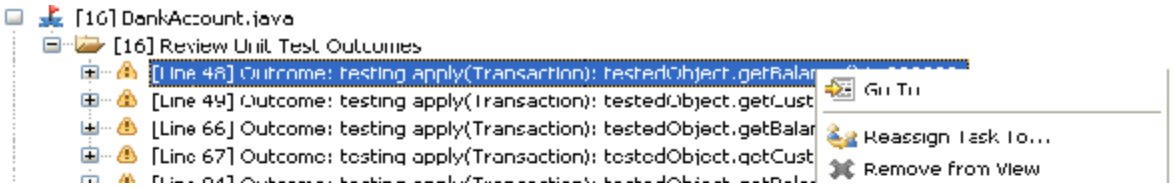
For tests run from the command line interface, rule violations are reported in the **Static Analysis** section of the report. If results were sent to Team Server, results can be imported into the GUI as described in Importing Results into the UI. They will then be available in the **Fix Static Analysis Violations** category of the Quality Tasks view.

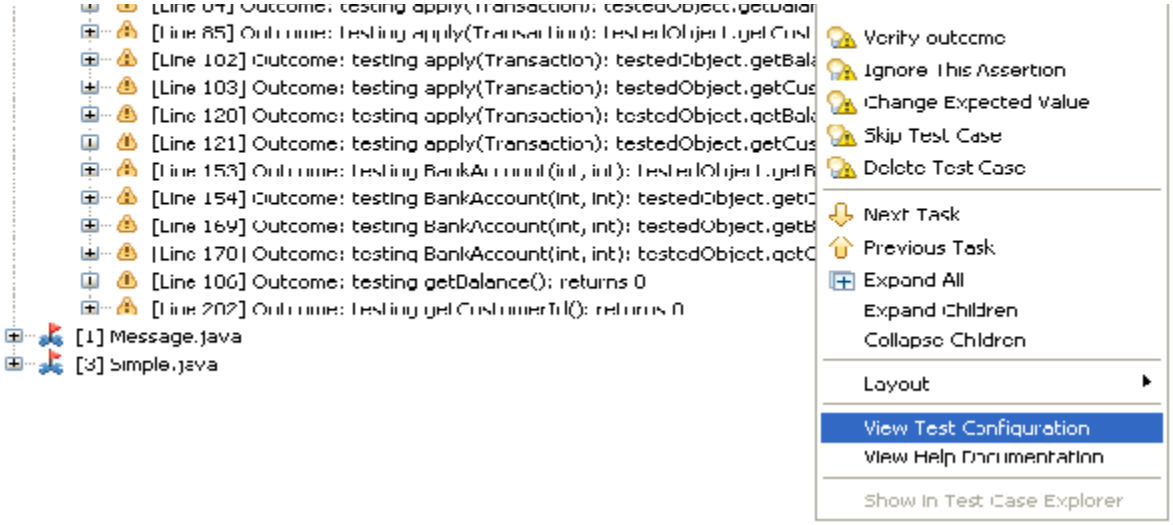
Learning More About Reported Bug Types

To view a detailed description of the reported bug type, right-click the bug message in the Quality Tasks view, then choose **View Rule Documentation** from the shortcut menu. A yellow "Yield" sign marks the node that you should right-click.

Opening Test Configurations that Trigger Violations

Test configurations that trigger violations can be opened from the Quality Tasks view: Right-click on a violation and choose **View Test Configuration**.

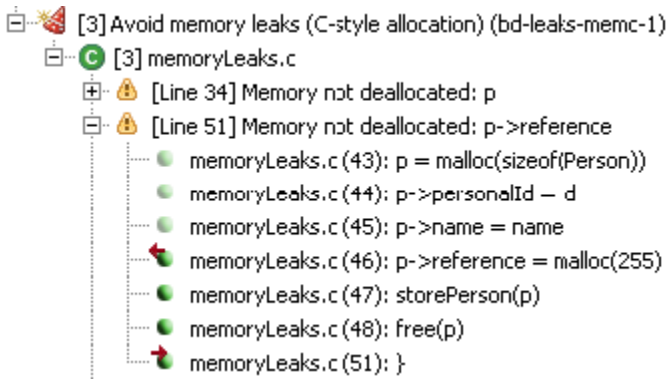




Quickly accessing test configuration from the violation is useful for group architects who are customizing tests and want to quickly disable rules that aren't applicable. Developers importing results from a server-based run may also need to open and review test configurations that trigger violations.

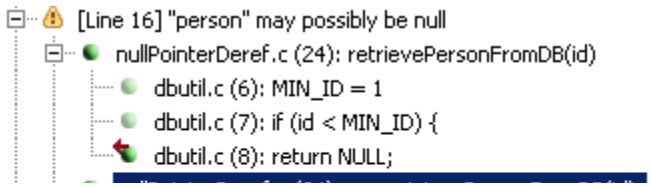
Understanding Flow Paths

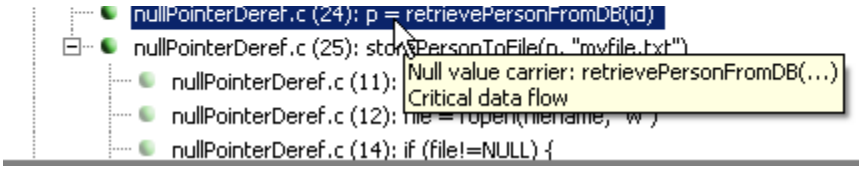
In the Quality Tasks view, each Flow Analysis violation is represented by a hierarchical flow path that precisely describes the code that leads to the identified problem. Each element in the path is a line of code that is executed during runtime. If a flow path has a call to a function, the element representing that function call is a node whose subnodes represent execution flow within the called function. The final element in the execution path is always the point where the bug manifests itself. The complete path is presented in order to explain why there is a bug at the final point.



Flow path elements are marked with icons that help explain exception handling behavior. If a path has a throw statement or a call to a function that happens to throw an exception on that path, the path element corresponding to the throw statement or the function call is marked by a red sphere. This red sphere indicates that the flow does not proceed as normal.

Each element in the flow path has a tool tip that describes the variables related to the violation. For example, an "Avoid null pointer dereferencing" violation description contains annotations describing which variables contain null values at which point in the flow path. To view a tool tip for a flow path element, place your cursor over it.





If you want to navigate through the code related to a reported execution path, use the **Next Task Element** and **Previous Task Element** buttons in the Quality Tasks view toolbar.

How do source code changes affect flow paths?

If the source code changes since Flow Analysis was run:

- Flow Analysis violations will continue to use the source code of the original flow path. This ensure that a valid violation path is shown.
- If an element in the flow path is outdated (i.e., the current source code does not match the analyzed code), the element—plus the violation node—will be marked with a special "out of date" icon and tool tip. Moreover, since the related source code is no longer available, the option to double-click an element to see the related code will be disabled.

The path may contain ellipses that indicate the differences between paths when there is more than one path leading from the violation cause to the violation point.

Compact vs. Detailed Path Trace View

By default, results are displayed using a compact path trace view, which shows only the essential executable statements in the defect path. This provides a quick overview of the various problems reported while still allowing you to drill down into the details in order to understand and repair each problem. Full path presentation is also available. See Compact vs Detailed Results Display for more details.

Identifying and Understanding Important Elements

Path elements that are important (with respect to the violation found) are marked with dark ball icons. Additionally, if you place your cursor over one of these specially-marked elements, a tooltip will explain why the element is considered important. For example, it might say "source of null value," "critical data flow," or "contains critical data flow" (output for function call nodes if they contain "important elements"). "Important elements" include violation cause, violation point, and all the elements which change the list of variables carrying data that leads to a violation. For example, assume the variable "a" carries a null pointer in the following excerpt:

```
1: b = a;
2: c = k;
```

Line 1 is important in case a violation is later found on variable "b", and is specially marked.

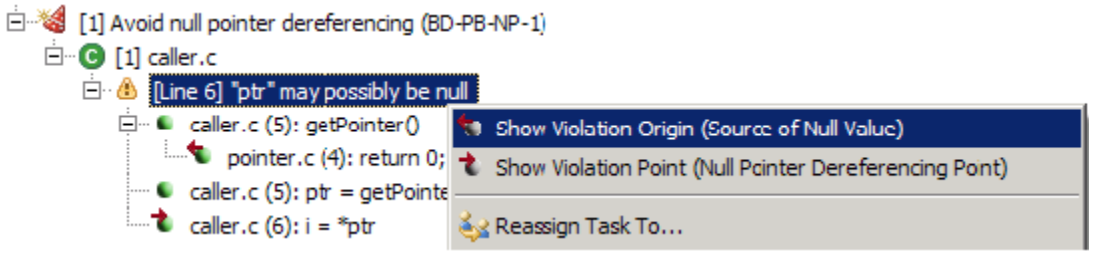
Line 2 is not related to the variables carrying bad data, so it's not important and isn't specially marked.

Understanding and Accessing the Violation Cause and Violation Point

The violation itself is represented by an execution path with two marked points:

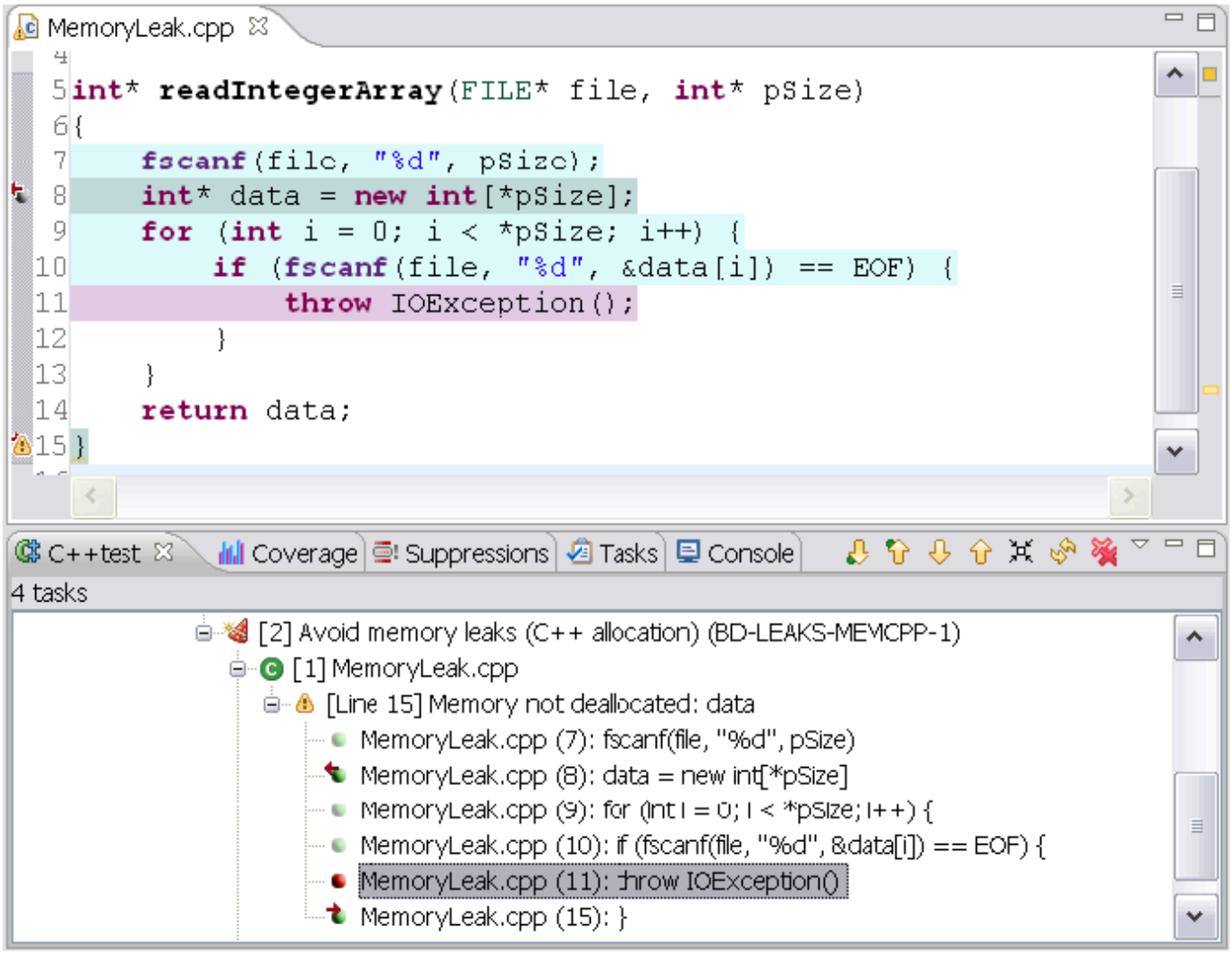
- **Violation cause.** This is the "source" of the violation. Normally this is the cause of the "bad data." For instance, in the "Avoid null pointer dereferencing" rule, the violation cause is the source of the null pointer.
- **Violation point.** This is the point of "bad data" usage which normally results in a bug in the program. For the "Avoid null pointer dereferencing" rule, this is the point where the variable holding the null pointer is dereferenced.

You can easily access the violation cause and violation point by right-clicking a reported violation (the node with the yellow caution icon), then choosing the appropriate command from the shortcut menu (either **Show Violation Cause** or **Show Violation Point**. For example, an "Avoid null pointer dereferencing" rule violation has the commands **Show Violation Cause (Source of Null Value)** and **Show Violation Point (Null Pointer Dereferencing Point)** in order to help you understand why the exception may occur in the code.



Highlighting the Source Code Referenced by a Violation Message

To highlight the source code referenced by a Flow Analysis violation message, simply double-click that message. An editor will open with the source code corresponding to the violation selected and the flow path highlighted in light blue. Any line at which an exception is thrown is marked with pink. All the lines which are important for the violation (violation cause, violation point, critical data flow) are highlighted in a dark color.



If you want to navigate through the code related to a reported execution path, use the **Next Violation Element** and **Previous Violation Element** buttons in the Quality Tasks view toolbar.

Tips

- You can disable or suppress rules that you do not want checked. For details, see Static Tab Settings: Defining How Static Analysis is Performed**.**
- To learn about the Flow Analysis rules that are included with C++test, choose **Parasoft> Help**, open the **C++test Static Analysis Rules** book, then browse the available Flow Analysis category rule description files.

Compact vs Detailed Results Display

Compact vs. Detailed Views

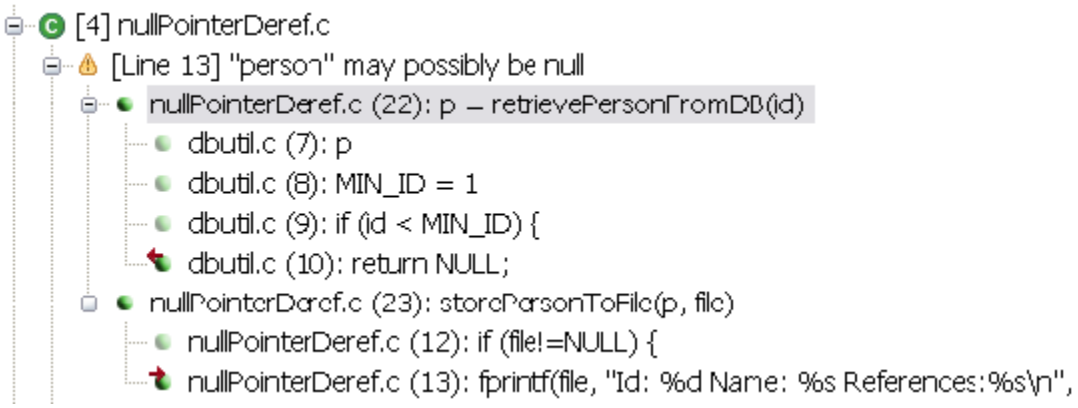
By default, results are displayed using a compact path trace view, which shows only the essential executable statements in the defect path. This provides a quick overview of the various problems reported while still allowing you to drill down into the details in order to understand and repair each problem.

Full path presentation is also available. When a violation is shown in the detailed form, the expandable task node contains all the elements of the execution path on which Flow Analysis thinks the detected bug manifests itself. In other words, Flow Analysis will show the exact sequence of lines that are executed to produce this bug. This gives you complete information about the problem found and the assumptions that Flow Analysis made. Sometimes this information is absolutely essential for understanding why Flow Analysis reports a possible problem.

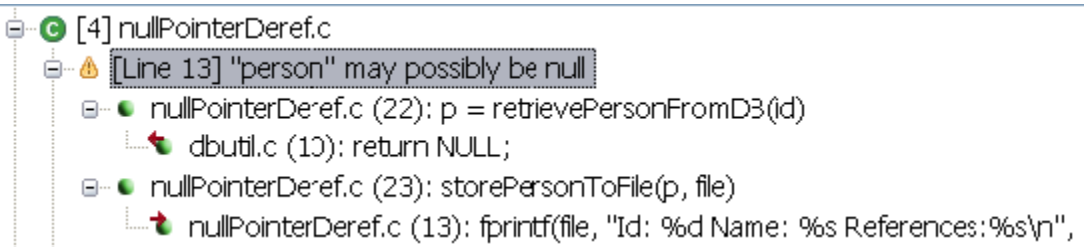
In many cases, such detailed information may not be necessary to understand the problem reported. In these cases, the compact form is convenient. When a violation is shown in the compact form, the expandable task node shows only the most important elements of the complete execution path on which Flow Analysis thinks the detected bug manifests itself. Flow Analysis treats the following execution path elements as important and shows in the compact view:

- Violation cause (normally the point from which the bad data originates).
- Violation point (the point where the bug manifests itself).
- Elements throwing exceptions (since they significantly influence control flow).
- Critical data flow elements (points where bad data flows from one variable to another [perhaps assignments of bad data to variables, passing bad data to methods as parameters or return of bad data from methods]; this information is key for understanding how a violation occurs).

Here is an example of the detailed view:



Here is an example of the compact view for the same violation:



Viewing Detailed Paths in the Editor

Even if the task view shows only the most important elements, all the path elements will be highlighted if a violation is selected and the corresponding code is shown in the IDE editor. Thus, you can use the compact view to get a quick assessment of the problem, then use the source editor to see the detailed execution path if you want more information about how the violation occurred.

Displaying the Detailed View

To show the detailed view for a particular violation:

- Right-click that violation's task node, then choose **Show complete violation path**.

To always show the detailed view:

1. Choose **Parasoft> Preferences**.
2. Open the **Quality Tasks** page.
3. Enable the **Show complete paths for BugDetective violations** option.

Customizing Flow Analysis Static Analysis

This topic explains how to customize Flow Analysis analysis—including what types of bugs detected, rule parameterization, analysis options, and the resources are checked by specific rules.

Specifying Which Bugs to Detect

The types of bugs that Flow Analysis detects is determined by which rules you have enabled in the Test Configuration—in much the same way as you configure static code analysis with coding standards. The basic static analysis settings (maximum tasks reported per rule, etc.—see Static Tab Settings: Defining How Static Analysis is Performed for details) apply to both static code analysis and Flow Analysis analysis.

For example, if you want to disable or enable Flow Analysis rules, you can do so by checking/clearing boxes in the Test Configuration's rule tree (in the **Static> Rules** Tree tab).

Configuring Rule Parameters

Some rules are parameterized, meaning that you can customize their settings to make the analysis process more flexible and tailored to your unique project needs. As a result, Flow Analysis can even be used to detect violations bound to usage of very specific APIs.

For details on how to parameterize rules, see Customizing Parameterized Rules.

Reporting Unvalidated Violations

One common rule parameter is **Report unvalidated violations**.

To explain how this parameter works, let's first look at the process of finding and reporting a violation. At first, Flow Analysis finds paths starting from the violation cause (the point where the bad or suspicious data originates) and ending at the violation point (the point where bad or suspicious data is used in a dangerous or suspicious operation). When Flow Analysis finds such a path violating a rule, it normally tries to validate the violation by verifying if the path is reachable from the beginning of the top-level function (the top-level function of the hierarchical representation of the violation path which may call other functions) of the violation path.

For example, here is a simple case where the violation path lies completely within a one function (no other functions are called from the violation path):

```
typedef struct SomeStruct {
    int hashCode;
} SomeStruct;
extern SomeStruct* getObject();
void check(int initialized)
{
    SomeStruct* obj;
    if (initialized) {
        obj = getObject();
```

```
    } else {
        obj = 0;
    }
    int hashCode = 0;
    // some other actions
    if (initialized) {
        hashCode = obj->hashCode;
    }
    // some other actions
}
```

When Flow Analysis analyzes this code, the following violation is found at first:

```
.C program.c (12): obj = 0; // Null value carrier: obj
.  program.c (14): int hashCode = 0; // Null value carrier: obj
.  program.c (16): if (initialized) { // Null value carrier: obj
.P program.c (17): hashCode = obj->hashCode; // Null value carrier: obj
```

The next stage is to validate the violation by checking whether the path of the violation can be reached from the beginning of the function. During this stage, Flow Analysis determines that "obj" can be assigned a null value only if "initialized" is false. However, in this case, the line where dereferencing occurs is not reachable. This is an example of how validating violation paths by verifying their reachability from the beginning of violation's top-level function can help to eliminate false positives. If the example was different and the path was reachable from the beginning of the function, the violation would pass validation—and in the detailed path mode, Flow Analysis would show the violation with the completed path, starting from the beginning of the top-level function, to explain how exactly it thinks the path may be reachable.

In some complicated cases, restricted analysis depth (used to keep analysis speed at a reasonable level) may prevent Flow Analysis from completing the violation validation procedure. If that happens, the violation remains unvalidated and Flow Analysis does not know whether it is reachable from the beginning of the corresponding function. In this case, the **Report unvalidated violations** parameter determines whether the violation is reported. Enabling this option increases the number of defects discovered by Flow Analysis, but it can also increase the number of false positives.

Configuring Flow Analysis Analysis Options

Additionally, you can control specific Flow Analysis options, such as analysis depth, support for multithreading APIs, and violation reporting verbosity, by modifying the parameters in the **Test Configuration's Static> Flow Analysis Advanced Settings** tab.

The available options in each subtab are described in the following sections:

- Performance Tab Options
- Verbosity Tab Options
- Terminators Tab Options
- Multithreading Tab Options
- Resources Tab Options
- Extended Scope of Analysis Tab Options
- Compiler-specific Settings Tab Options

Performance Tab Options

- **Incremental analysis** options control the incremental analysis feature, which is described in Running Flow Analysis in Incremental Mode. Available options are:
    - **Enable incremental analysis:** Determines whether incremental analysis is used.
    - **Compact incremental caches every [days]:** Determines how often compactization of incremental caches is run. Incremental analysis is optimized for speed; although Flow Analysis strives to always keep cache sizes small and remove unnecessary data, source code changes may result in these caches containing some data that will no longer be used. Compactization, which is run regularly as defined by this parameter, removes all outdated data. More precisely, if the time that has elapsed since the previous compactization is greater than the number of days specified for this option, compactization is performed immediately after the incremental run of Flow Analysis
    - **Clear cache**: Clears the incremental analysis cache for the current workspace. After the cache is cleaned the first run of incremental analysis on the workspace will require complete analysis of the code base.
- **Depth of analysis** options determine the depth of Flow Analysis analysis. Deeper analysis means more violations are found. However, the trade off of deeper analysis is slower performance and (slightly) increased memory consumption. Available options are:

    - **shallowest (fastest):** Finds only the most obvious problems in the source code. It is limited to cases where the cause of the problem is located close to the code where the problem occurs. The execution paths of violations found by this type of analysis normally span several lines of code in a single function. Only rarely will they span more than 3 function calls.
    - **shallow (fast):** Like the "shallowest" analysis type, finds only the most obvious problems in the source code. However, it produces a greater overall number of findings and allows for examination of somewhat longer execution paths.
    - **standard:** Finds many complicated problems with execution paths containing tens of elements. Standard analysis goes beyond shallow analysis and also looks for more complicated problems, which can occur because of bad flow in a single function or due to improper interaction between different functions in different parts of the analyzed project. Violations found by this type of analysis often reveal non-trivial bugs in the analyzed source code and often span tens of lines of code.
    - **deep (slow):** Allows for detection of a greater number of problems of the same complexity and nature as those defined for 'standard' depth. This type of analysis is slower than the standard one.
    - **thorough (slower):** Finds more complicated problems. This type of analysis will perform a thorough scan of the code base; this requires more time, but will uncover many very complicated problems whose violation paths can span more than a hundred lines of code in different parts of the scanned application. This option is recommended for nightly runs.
- **Strategy for timeout** options allow you to configure your timeout strategy ensure to the analysis completes within a reasonable time. Available options are:
    - **time**: Analysis of the given hotspot is stopped after spending the defined amount of time on it. Note: in some cases, using this option can result in a slightly unstable number of violations being reported.
    - **instructions**: Analysis of the given hotspot is stopped after executing the defined number of Flow Analysis instructions.Note: to determine the proper number of instructions to be set up for your environment, review information about timeouts in the Setup Problems section of the generated report.
    - **off**: No timeout. Note: using this option may require significantly more time to finish the analysis.

  The default timeout option is **time** set to 60 seconds. To get information about the Flow Analysis timeouts that occurred during the analysis, review the Setup Problems section of the report generated after the analysis.

- **Enable swapping of analysis data to disk:** Enabled by default. Flow Analysis writes to disk the data necessary for analysis. Swapping is done in a similar process as incremental analysis, as well as uses the same persistent storage.You can disable this option, which may result in faster analysis, if you are running Flow Analysis analysis on small to moderate size projects that do not require a lot of memory or when plenty of memory is available (for example, for 64-bit systems). For description of the swapping feature please refer to the Running Flow Analysis with Swapping of Analysis Data Enabled section.

Verbosity Tab Options

In the Verbosity subtab, you can configure the following options:

- **Do not report violations when cause cannot be shown:** Determines whether Flow Analysis reports violations where causes cannot be shown.
  Some Flow Analysis rules require that Flow Analysis checks all the possible paths leading to a certain point and verifies that a certain condition is met for all those paths. In such cases, a violation is associated with a set of paths (whereas in simple cases, a violation is represented by only one path). All of the paths in such a violation end with the violation point common to all the paths in the violation. However, different paths may start at different points in code. The beginning of each path is a violation cause (a point in code which stipulates a violation of a certain condition later in the code at the violation point). If a multipath violation's different paths have different causes, Flow Analysis will show only the violation point (and not the violation causes).

  Violations containing only the violation point may be difficult to understand (compared to regular cases where Flow Analysis shows complete paths starting from violation causes and leading to violation points). That's why we provide an option to hide violations where the cause cannot be shown.

  See the example below for additional details.

- **Do not report violations whose paths pass via inline assembly code:** Prevents the reporting of violations whose paths pass via inline assembly code instructions.
- **Level of reporting similar violations** (i.e those that share a violation point, or both the point and the cause): Allows you to determine whether Flow Analysis reports all the violations that it can find in the analyzed code or if it hides some of them. Available options are:
    - **Report all similar violations:** Reports all identified Flow Analysis violations.
    - **Do not show more than one violation with the same cause and violation point:** Prevents the reporting of duplicate Flow Analysis violations. Duplicate violations are those that share the same violation cause and the same violation point (even though their flow paths may be different).
    - **Do not show more than one violation per suspicious point:** Restricts reporting to one violation (for a single rule) per suspicious point. When verbosity is set to this level, Flow Analysis performance is somewhat faster.

Example - Impact of "Do not report violations when cause cannot be shown"

The following sample code causes a multi-cause violation of the BD.PB-SWITCH rule to be reported only when **Do not report violations when cause cannot be shown** is enabled:

```
#include "stdio.h"
enum Figures {
    SPHERE,
```

```
    CIRCLE,
    CUBE,
    SQUARE,
    HIMESPHERE
};
static void guessFigure(int round, int volumetric)
{
    int figure;
    if (round && volumetric) {
        figure = SPHERE; /* CAUSE 1 */
    } else if (round && !volumetric) {
        figure = CIRCLE; /* CAUSE 2 */
    } else if (!round && volumetric) {
        figure = CUBE; /* CAUSE 3 */
    } else {
        figure = SQUARE; /* CAUSE 4 */
    }
    switch (figure) {
        case SQUARE:
                printf("This is a sphere");
                break;
        case HIMESPHERE:
                printf("This is a hemispere");
                break;
        case CIRCLE:
                printf("This is a circle");
                break;
        case CUBE:
                printf("This is a cube");
                break;
        default:
                printf("This is a square");
                break;
    }
}
```

Terminators Tab Options

This tab allows you to define functions that terminate application execution. C/C++ developers sometimes use functions that terminate application execution in the event of a fatal error from which recovery is impossible. Examples of such functions are abort() and exit() from the standard library. Since Flow Analysis analyzes the application's execution flow, it's important for it to be aware of the terminating functions that break execution flow by immediately stopping the application. Without such knowledge, Flow Analysis might make incorrect assumptions about the application flow.

Flow Analysis is aware of the terminating functions that are defined in the standard library. However, this is often not sufficient because non-standard libraries define their own terminating functions. If your application uses one of these functions, you should communicate that to Flow Analysis by specifying the custom terminating function in this tab. Otherwise, Flow Analysis may produce false positives with execution paths passing by terminating functions.

Use the table listing supported APIs to enable/disable terminators from various APIs as well as to define your own APIs containing terminating functions. To add information about terminating functions from a certain library click Add, type the name of the library and press <Enter>. A new entry will be added to the terminators APIs table. Next, click the "Edit" button and complete the table that opens; the table has the following columns:

- **Enabled**: Specifies whether a built-in or custom terminator should be considered during the analysis.
- **Fully qualified type name or namespace:** Specifies the entity for a particular terminator. If this field is left empty, only the global function with the name specified in the 'Function name' column will be considered a terminator.

    - For example: The field value may be "myNameSpace::myClass" if the terminator is declared in 'myClass' coming from the 'myNameSpace' name space. Or, it may be "myNameSpace" if it is not declared in a type, but belongs only to the 'myNameSpace'.
- **Function name:** Specifies the name of the terminating function.
- **+ definitions in subclasses:** Indicates whether the terminating function definitions in subclasses should be considered terminating functions as well. This applies to both instance and non-instance functions, and makes sense only if its fully qualified type name is specified.

Multithreading Tab Options

This tab allows you to define functions for synchronization between threads as well as to activate/deactivate known multithreading functions. The information defined here affects the behavior of rules from the BD.TRS (Threads and Synchronization) category. These rules will check all the functions that are defined and activated on this tab.

Use the table that lists supported APIs to enable/disable synchronization functions from various APIs as well as to define your own APIs containing synchronization functions. To add information about synchronization functions from a certain library, click **Add**, type the name of the library and press **<Enter>**. A new entry will be added to the multithreading APIs table. Next, click **Edit** and complete the dialog to define particular synchronization functions. This dialog allows you to define the following types of functions:

- Functions for locking (for instance, obtaining a mutex)
- Functions for unlocking (for instance, releasing a mutex)
- Sleep functions

These functions are defined in much the same was as resources (described in Specifying Resources to Check for BD.RES Rules).

Resources Tab Options

Allows you to define which resources the BD.RES category (Resources) rules should check. These rules check for the correct usage of all resources that are defined and enabled on this tab.

For details on configuring these resource checking settings, see the following section.

Specifying Resources to Check for BD.RES Rules

The Test Configuration manager's **Static> Flow Analysis Advanced Options> Resources** tab allows you to define which resources the BD. RES category (Resources) rules should check. These rules check for the correct usage of all resources that are defined and enabled on this tab.

You can use the table to enable/disable checking for various types of resources. The **Add**, **Remove**, and **Edit** buttons can be used to manage custom resources.

To add information about resources from a certain library:

1. Click **Add**.
2. Type the name of the resource type.
3. Press **<Enter>.** A new entry will be added to the resource table.
4. If appropriate/desired, disable the **Do not report violations at application termination** option.
5. Complete the entry by clicking the Edit button, then completing the dialog that allows you to define how the given resource is allocated /deallocated. Details about completing this dialog are provided below.

Parameterization Common to Both Allocators and Deallocators

Any single row in either panel's table corresponds to exactly one allocating/closing function and contains enough information to unambiguously identify a function (or—if wildcards are used—a family of functions) and depict how it allocates/frees a resource.

The column labeled **Enabled** should be used to include/exclude a function from consideration during the analysis.

For any function, the corresponding field in the **Fully-qualified type name or namespace (wildcard)** column must be completed with the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type. Generally, '*' can be used as part of the name string (using its standard wildcard meaning: to denote any number of any symbols).

The fields in the **Function name (wildcard)** column should be completed with the names of the allocating/closing functions. '*' can be used to denote any number of any symbols.

Use the check box fields in the **+ definitions in subclasses** column to indicate whether the definitions (of functions with the given name) in subclasses should be considered allocators/closers as well. Note that this applies to both instance and static functions.

Allocators

The 'Resource allocators' panel can be completed with the descriptors of functions that can produce a resource. These can be represented by functions that are able to:

- Return an allocated resource. In this case, check the checkbox in the **Returns a resource object** column.

- Allocate a resource in the object on which the function is called. In this case, check the check-box in the **"this" object is a resource** column.
- Allocate a resource in one or more of its parameters. In this case, either specify a 1-based number of the parameter that is allocated by the function, or use '*' to denote that all of the parameters are allocated.

It is common that allocation functions return an error code to indicate allocation failure. When an allocation function returns a pointer to a resource, a NULL pointer normally indicates an allocation failure. When Flow Analysis is looking for resource leaks, it needs to understand if allocation succeeded or failed; this helps it report only missing calls to deallocation functions on paths where allocation actually occurred. In cases where a resource allocator function returns a pointer to a resource, Flow Analysis assumes that the resource is successfully allocated if the pointer is not NULL.

If a resource allocator returns an integral value, you can specify a return value constraint in case of allocation failure by entering the condition into the **Return value constraint on error** field. The condition must be specified according to the following format: `<comparison operator><integer value>`. For example, if the function returns non-zero value on failure, enter "!=0" (without quotes) into the field. If return code on error is -1, type "==-1" there. In addition to "!=" and "==", you can use the following operators for specifying error conditions: ">", ">=", "<", and "<=".

Deallocators

The second panel should be completed with functions that close resources. Two cases where a function closes a resource are possible:

- A resource in the object on which the function is called is closed. In this case, check the check-box in the **"this" object is a resource** column.
- A resource in one or more of its parameters is closed. In this case, either specify a 1-based number of the parameter that is closed by the function, or use '*' to denote that all of the parameters are allocated.

Resources with Built-In Support

- By default, the resource table contains entries for which there is built-in support. Here is a list of these supported resources:Directories (dirent.h)
- Dynamic libraries (dlfcn.h)
- Files (stdio.h)
- 'glob' results (glob.h)
  Corresponds to 'glob_t' structures allocated by calls to:

```
int glob(const char *pattern, int flags,
                int (*errfunc) (const char *epath, int eerrno),
                glob_t *pglob);
```

- 'hostent' structures (hostent.h)
  Corresponds to 'glob_t' structures allocated by calls to:

```
struct hostent *getipnodebyname(const char *name, int af, int flags,
int
error_num);
struct hostent *getipnodebyaddr(const void *addr, size_t len, int af,
int
error_num);
```

- Memory (standard C)
  Memory managed by standard C allocation functions.
- Memory (standard C++)
  Memory managed by C++ memory management operators new/delete.
- Memory (Windows API)
  Memory managed by Windows API.
- Message catalogs (nl_types.h)
- Pipes (stdio.h)
- PThreads (pthread.h)
- Regexps (regex.h)
- Sockets (socket.h, Winsock2.h)
- Socket address structures (netdb.h)
  Corresponds to 'addrinfo' structures allocated by calls to:

```
```

```
int getaddrinfo(const char *node, const char *service, const struct
addrinfo
hints, struct addrinfo **res);
```

- Windows API
  Includes the following resource types:
  - Kernel, User and GDI API (Windows.h)
  - Print spooler API (Winspool.h)
  - Backup Functions (Sisbkup.h)
  - Wait chain traversal (Wct.h)
  - Network Management (Lm.h)

Defining Custom Resources

For an example of how custom resources are defined, consider the following plain C code using a non-standard resource:

```
/* returns NULL on allocation failure */
void* myAlloc(void);
void myDealloc(void*);
static void createMyResource_Leak()
{
    void* resource = myAlloc();
} /* 'res' is not closed on the path where it is not NULL */
static void createMyResource_NoLeak()
{
    void* resource = myAlloc();
    if (!resource) {
        return; /* no leak here, if res is NULL, it means allocation
failed */
    }
    /* use the resource */
    myDealloc(resource);
}
```

We want to define methods allocating/deallocating the resource so that Flow Analysis can be used to find leaks of this non-standard resource. To achieve this, we need to perform the following steps:

1. In the Resource tab, click **Add** and define the name of the new resource. This name will be used to report violations associated with this resource.
2. Disable the **Do not report violations at application termination** option.
3. Click **Edit** to specify how this resource is manipulated.
4. Define `myAlloc` as an allocator as follows:
   - Fully qualified type name or namespace: empty
   - Method name: myAlloc
   - + definitions in subclasses: unchecked
   - '"this" object is a resource: unchecked
   - Returns a resource object: checked
   - Return value constraint on error: empty
   - Numbers of the parameters representing resources: empty
5. Specify the deallocator as follows:
   - Fully qualified type name or namespace: empty
   - Method name: myDealloc
   - + definitions in subclasses: unchecked
   - "this" object is a resource: unchecked
   - Numbers of the parameters representing resources: 1

Now let's consider a different example. Here, a resource-opening function receives a pointer to a resource handle as a parameter and initializes it with the handle of the newly allocated resource. An error code of -1 is returned if the allocation fails.

```
```

```
int openMyResource(int* pHandle);
void closeMyResource(int handle);

static void openMyResource_Leak()
{
    int handle;
    openMyResource(&handle);
 } // 'res' is not closed
static void openMyResource_NoLeak()
{
    int handle;
    int status = openMyResource(&handle);
    if (status == -1) {
        return; // no leak here, status == -1 indicates allocation failure
    }
    // use the resource
    closeMyResource(handle);
}
```

Now let's add support for this type of resource in Flow Analysis by performing the following actions:

1. In the Resource tab, click **Add** and define the name of the new resource. This name will be used to report violations associated with this resource.
2. Disable the **Do not report violations at application termination** option.
3. Click **Edit** to specify how this resource is manipulated.
4. Define `openMyResource` as an allocator as follows:
    - Fully qualified type name or namespace: empty
    - + definitions in subclasses: unchecked
    - '"this" object is a resource: unchecked
    - Returns a resource object: unchecked
    - Return value constraint on error: ==-1
    - Numbers of the parameters representing resources: 1
5. Specify the deallocator as follows:
    - Fully qualified type name or namespace: empty
    - Method name: closeMyResource
    - + definitions in subclasses: unchecked
    - "this" object is a resource: unchecked
    - Numbers of the parameters representing resources: 1

Extended Scope of Analysis Tab Options

When performing code analysis, Flow Analysis processes definitions of functions that are defined in source and header files under test. Functions that are defined in header files outside the testing scope are not analyzed, and Flow Analysis is not aware of their semantics. If Flow Analysis requires information about function definitions that are defined in header files outside the testing scope, you can configure the following options:

External files to analyze: Specifies absolute paths to additional header files to be analyzed by Flow Analysis. Use wildcards to specify the pattern.

External functions to analyze: Specifies additional functions to be analyzed by Flow Analysis. Complete the table with the following information:

- **Enabled**: specifies whether the function should be considered during analysis
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the function. '*' can be used to denote any number of any symbols.
- **Number of parameters**: specifies the number of function's parameters. '-1' can be used to denote any number of parameters.
- **+ definitions in subclasses**: a checkbox that indicates whether the definitions (of functions with the given name) in subclasses should be included as well. Note that this applies to both instance and static functions.

Compiler-specific Settings Tab Options

Internal representation of the "errno**" value**: The Standarddefineserno to be a modifiable lvalue of type int. It is unspecified whetherer rno is a macro or an identifier declared with an external linkage. Implementations may use the global variable "errno" or "__errno", or apply the

"(*errno_function())" pattern with different names of the called functions. This option allows you to specify the names of these variables and functions with regular expressions:

- **Function name pattern**: The name of the function that is called when the "errno" value is used. The name must be specified with regular expressions.
- **Variable name pattern**: The name of the variable that is called when the "errno" value is used. The name must be specified with regular expressions.

Internal representation of the call to a function from the header <ctype.h>: The Standard specifies several functions to be defined in the header <ctype.h>. Some implementations (e.g GNU GCC in the C mode) define these functions as macros that expand to the code which tests an element of the internal array against some flags. This can be either a global array or a pointer returned by a function. This option allows you to specify names of these variables and functions with regular expressions:

- **Function name pattern**: The name of the function that is invoked internally instead of one of the functions from the header <ctype.h> (define with regular expressions). The name must be specified with regular expressions.
- **Variable name pattern**: The name of the variable that is used internally after a call to one of the functions from the header <ctype.h>. The name must be specified with regular expressions.