C++ Build Insights | Automate build tool chain analysis

Use C++ Build Insights for real-time and automated build analysis and optimization.

Learn how your build performs

OVERVIEW

Get started with C++ Build Insights

TUTORIAL

Build Insights function view

Build Insights included files view

vcperf and Windows Performance Analyzer

Windows Performance Analyzer basics

Measure build performance

REFERENCE

vcperf commands

Windows Performance Analyzer views

Create your own build performance analysis

REFERENCE

SDK reference

Get started with C++ Build Insights

Article • 10/29/2021

C++ Build Insights is a collection of tools that provides increased visibility into the Microsoft Visual C++ (MSVC) tool chain. The tools collect data about your C++ builds, and present it in a format that can help you answer common questions, like:

- Are my builds sufficiently parallelized?
- What should I include in my pre-compiled header (PCH)?
- Is there a specific bottleneck I should focus on to increase my build speeds?

The main components of this technology are:

- *vcperf.exe*, a command-line utility that you can use to collect traces for your builds,
- a Windows Performance Analyzer (WPA) extension that allows you to view build traces in WPA, and
- the C++ Build Insights SDK, a software development kit for creating your own tools that consume C++ Build Insights data.

Documentation sections

Tutorial: vcperf and Windows Performance Analyzer
Learn how to collect build traces for your C++ projects and how to view them in WPA.

Tutorial: Windows Performance Basics
Discover useful WPA tips for analyzing your build traces.

C++ Build Insights SDK
An overview of the C++ Build Insights SDK.

Articles

Read these articles from the official C++ team blog for more information on C++ Build Insights:

Introducing C++ Build Insights

Analyze your builds programmatically with the C++ Build Insights SDK

Finding build bottlenecks with C++ Build Insights

Faster builds with PCH suggestions from C++ Build Insights

Profiling template metaprograms with C++ Build Insights

Improving code generation time with C++ Build Insights

Introducing vcperf /timetrace for C++ build time analysis

Faster C++ builds, simplified: a new metric for time

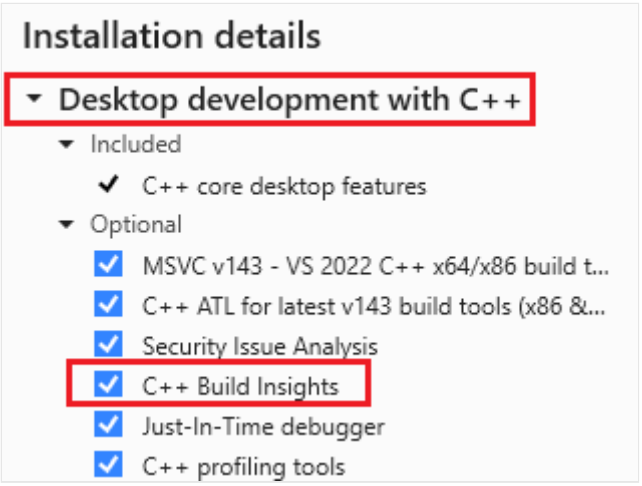
Tutorial: Troubleshoot function inlining on build time

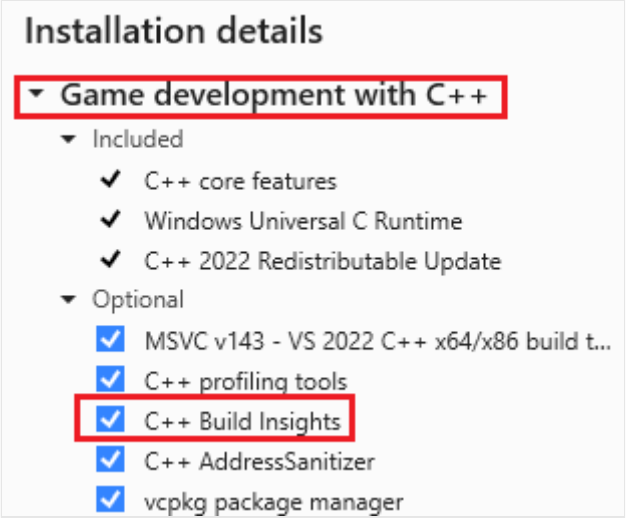
Article • 05/31/2024

Use Build Insights **Functions** view to troubleshoot the impact of function inlining on build time in your C++ projects.

Prerequisites

- Visual Studio 2022 17.8 or greater.
- C++ Build insights is enabled by default if you install either the Desktop development with C++ workload or the Game development with C++ workload.





Overview

Build Insights, now integrated into Visual Studio, helps you optimize your build times--especially for large projects like AAA games. Build Insights provides analytics such as

Functions view, which helps diagnose expensive code generation during build time. It displays the time it takes to generate code for each function, and shows the impact of __forceinline.

The `__forceinline` directive tells the compiler to inline a function regardless of its size or complexity. Inlining a function can improve runtime performance by reducing the overhead of calling the function. The tradeoff is that it can increase the size of the binary and impact your build times.

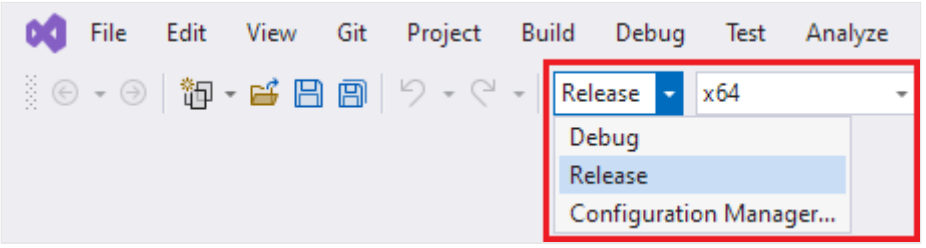
For optimized builds, the time spent generating code contributes significantly to the total build time. In general, C++ function optimization happens quickly. In exceptional cases, some functions can become large enough and complex enough to put pressure on the optimizer and noticeably slow down your builds.

In this article, learn how to use the Build Insights **Functions** view to find inlining bottlenecks in your build.

Set build options

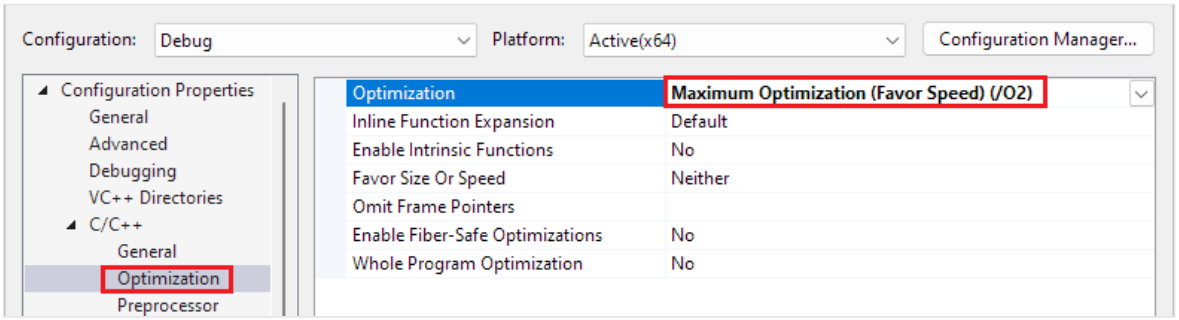
To measure the results of `__forceinline`, use a **Release** build because debug builds don't inline `__forceinline` since debug builds use the /Ob0 compiler switch, which disables that optimization. Set the build for **Release** and **x64**:

1. In the **Solution Configurations** dropdown, choose **Release**.
2. In the **Solution Platforms** dropdown, choose **x64**.



Set the optimization level to maximum optimizations:

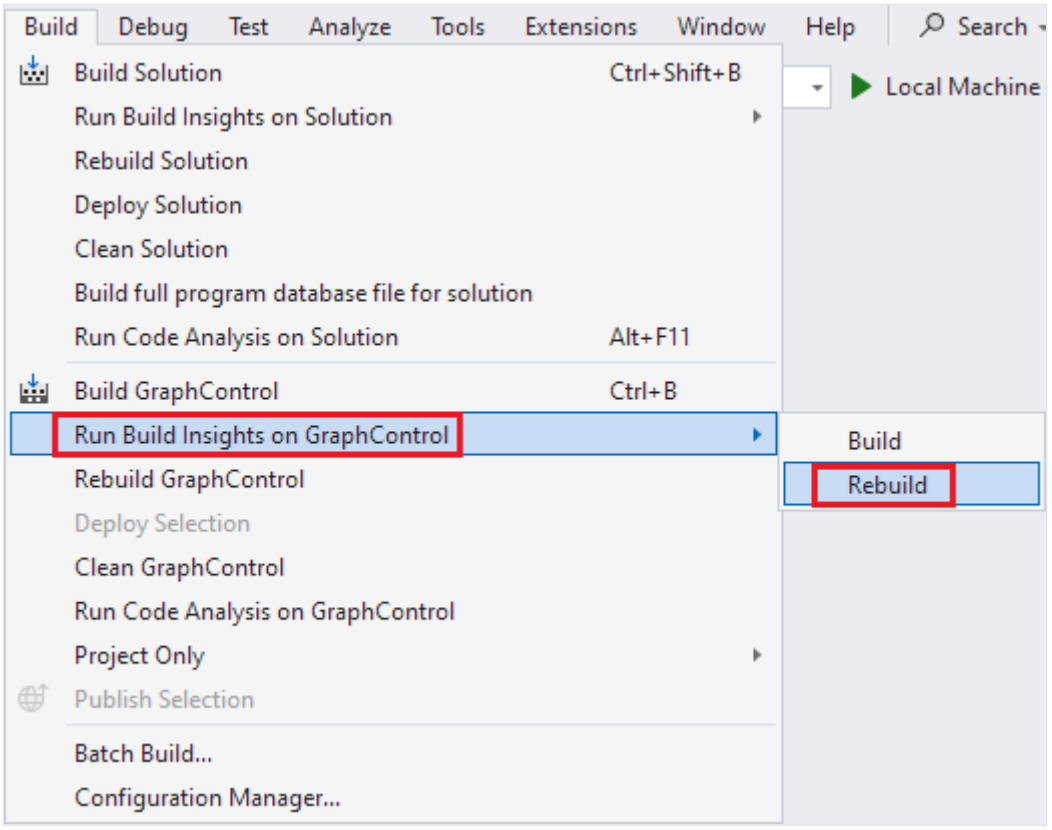
1. In the **Solution Explorer**, right-click the project name and select **Properties**.
2. In the project properties, navigate to **C/C++** > **Optimization**.
3. Set the **Optimization** dropdown to **Maximum Optimization (Favor Speed)** (/O2).



4. Click **OK** to close the dialog.

Run Build Insights

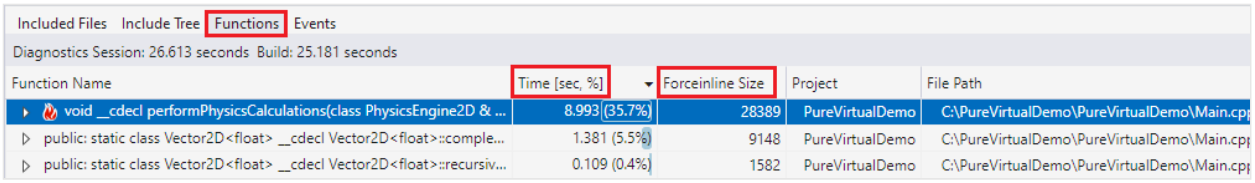
On a project of your choosing, and using the **Release** build options set in the previous section, run Build Insights by choosing from the main menu **Build** > **Run Build Insights on Selection** > **Rebuild**. You can also right-click a project in the solution explorer and choose **Run Build Insights** > **Rebuild**. Choose **Rebuild** instead of **Build** to measure the build time for the entire project and not for just the few files may be dirty right now.



When the build finishes, an Event Trace Log (ETL) file opens. It's saved in the folder pointed to by the Windows `TEMP` environment variable. The generated name is based on the collection time.

Function view

In the window for the ETL file, choose the **Functions** tab. It shows the functions that were compiled and the time it took to generate the code for each function. If the amount of code generated for a function is negligible, it won't appear in the list to avoid degrading build event collection performance.



The **Time [sec, %]** column shows how long it took to compile each function in wall clock responsibility time (WCTR). This metric distributes the wall clock time among functions based on their use of parallel compiler threads. For example, if two different threads are compiling two different functions simultaneously within a one-second period, each function's WCTR is recorded as 0.5 seconds. This reflects each function's proportional share of the total compilation time, taking into account the resources each consumed during parallel execution. Thus, WCTR provides a better measure of the impact each function has on the overall build time in environments where multiple compilation activities occur simultaneously.

The **Forceinline Size** column shows roughly how many instructions were generated for the function. Click the chevron before the function name to see the individual inlined functions that were expanded in that function how roughly how many instructions were generated for each.

You can sort the list by clicking on the **Time** column to see which functions are taking the most time to compile. A 'fire' icon indicates that cost of generating that function is high and is worth investigating. Excessive use of `__forceinline` functions can significantly slow compilation.

You can search for a specific function by using the **Filter Functions** box. If a function's code generation time is too small, it doesn't appear in the **Functions** View.

Improve build time by adjusting function inlining

In this example, the `performPhysicsCalculations` function is taking the most time to compile.

Included Files | Include Tree | **Functions** | Events

Diagnostics Session: 26.613 seconds Build: 25.181 seconds

Function Name	Time [sec, %]	Forceinline Size	Project	File Path
void __cdecl performPhysicsCalculations(class PhysicsEngine2D & ...	8.993 (35.7%)	28389	PureVirtualDemo	C:\PureVirtualDemo\PureVirtualDemo\Main.cpp
public: static class Vector2D<float> __cdecl Vector2D<float>::comple...	1.381 (5.5%)	9148	PureVirtualDemo	C:\PureVirtualDemo\PureVirtualDemo\Main.cpp
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiv...	0.109 (0.4%)	1582	PureVirtualDemo	C:\PureVirtualDemo\PureVirtualDemo\Main.cpp

Investigating further, by selecting the chevron before that function, and then sorting the **Forceinline Size** column from highest to lowest, we see the biggest contributors to the problem.

Included Files | Include Tree | Functions | Events

Diagnostics Session: 26.613 seconds Build: 25.181 seconds

Function Name	Time [sec, %]	Forceinline Size
void __cdecl performPhysicsCalculations(class PhysicsEngine2D & __ptr64,class PhysicsWorld2D & __ptr64)	8.993 (35.7%)	28389
public: static class Vector2D<float> __cdecl Vector2D<float>::complexOperation(class Vector2D<float> const & __ptr64,cla...	0.000 (0.0%)	315
public: static class Vector2D<float> __cdecl Vector2D<float>::complexOperation(class Vector2D<float> const & __ptr64,cla...	0.000 (0.0%)	315
public: static class Vector2D<float> __cdecl Vector2D<float>::complexOperation(class Vector2D<float> const & __ptr64,cla...	0.000 (0.0%)	315
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static class Vector2D<float> __cdecl Vector2D<float>::recursiveHelper(class Vector2D<float> const & __ptr64,int)	0.000 (0.0%)	119
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75
public: static float __cdecl Vector2D<float>::sin(float)	0.000 (0.0%)	75

There are some larger inlined functions, such as `Vector2D<float>::complexOperation()` and `Vector2D<float>::recursiveHelper()` that are contributing to the problem. But there are many more instances (not all shown here) of `Vector2d<float>::sin(float)`, `Vector2d<float>::cos(float)`, `Vector2D<float>::power(float,int)`, and `Vector2D<float>::factorial(int)`. When you add those up, the total number of generated instructions quickly exceeds the few larger generated functions.

Looking at those functions in the source code, we see that execution time is going to be spent inside loops. For example, here's the code for `factorial()`:

C++

```cpp
static __forceinline T factorial(int n)
{
    T result = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < i; ++j) {
```

```
            result *= (i - j) / (T)(j + 1);
        }
    }
    return result;
}
```

Perhaps the overall cost of calling this function is insignificant compared to the cost of the function itself. Making a function inline is most beneficial when the time it takes to call the function (pushing arguments on the stack, jumping to the function, popping return arguments, and returning from the function) is roughly similar to the time it takes to execute the function, and when the function is called a lot. When that's not the case, there may be diminishing returns on making it inline. We can try removing the `__forceinline` directive from it to see if it helps the build time. The code for `power`, `sin()` and `cos()` is similar in that the code consists of a loop that will execute many times. We can try removing the `__forceinline` directive from those functions as well.

We rerun Build Insights from the main menu by choosing **Build** > **Run Build Insights on Selection** > **Rebuild**. You can also right-click a project in the solution explorer and choose **Run Build Insights** > **Rebuild**. We choose **Rebuild** instead of **Build** to measure the build time for the entire project, as before, and not for just the few files may be dirty right now.

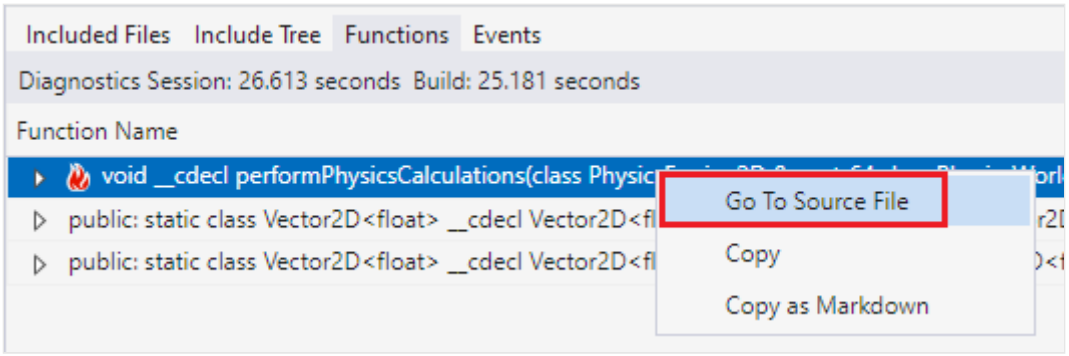
The build time goes from 25.181 seconds to 13.376 seconds and the `performPhysicsCalculations` function doesn't show up anymore in the **Functions** view because it doesn't contribute enough to the build time to be counted.

The Diagnostics Session time is the overall time it took do the build plus any overhead for gathering the Build Insights data.

The next step would be to profile the application to see if the performance of the application is negatively impacted by the change. If it is, we can selectively add `__forceinline` back as needed.

Navigate to the source code

Double-click, right-click, or press **Enter** while on a file in the **Functions** view to open the source code for that file.



Tips

- You can **File** > **Save As** the ETL file to a more permanent location to keep a record of the build time. You can then compare it to future builds to see if your changes are improving build time.
- If you inadvertently close the Build Insights window, reopen it by finding the `<dateandtime>.etl` file in your temporary folder. The `TEMP` Windows environment variable provides the path of your temporary files folder.
- To dig into the Build Insights data with Windows Performance Analyzer (WPA), click the **Open in WPA** button in the bottom right of the ETL window.
- Drag columns to change the order of the columns. For instance, you may prefer moving the **Time** column to be the first column. You can hide columns by right-clicking on the column header and deselecting the columns you don't want to see.
- The **Functions** view provides a filter box to find a function that you're interested in. It does partial matches on the name you provide.
- If you forget how to interpret what the **Functions** view is trying to show you, hover over the tab to see a tooltip that describes the view. If you hover over the **Functions** tab, the tooltip says: "View that shows statistics for functions where the children nodes are force-inlined functions."

Troubleshooting

- If the Build Insights window doesn't appear, do a rebuild instead of a build. The Build Insights window doesn't appear if nothing actually builds; which may be the case if no files changed since the last build.
- If the Functions view doesn't show any functions, you may not be building with the right optimization settings. Ensure that you're building Release with full optimizations, as described in Set build options. Also, if a function's code generation time is too small, it doesn't appear in the list.

See also

Inline functions (C++)
Faster C++ builds, simplified: a new metric for time
Build Insights in Visual Studio video - Pure Virtual C++ 2023
Troubleshoot header file impact on build time
Functions View for Build Insights in Visual Studio 2022 17.8
Tutorial: vcperf and Windows Performance Analyzer
Improving code generation time with C++ Build Insights

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

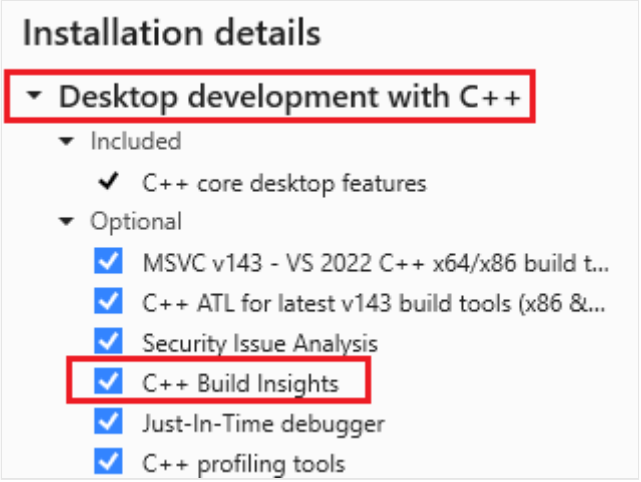
Tutorial: Troubleshoot header file impact on build time

Article • 05/31/2024

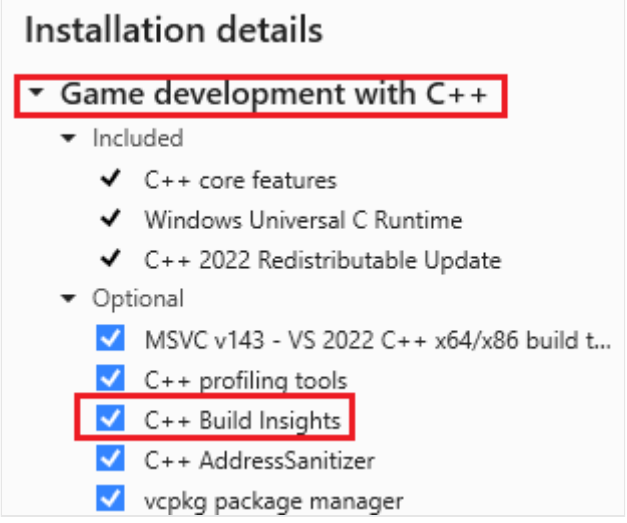
Use Build Insights **Included Files** and **Include Tree** views to troubleshoot the impact of `#include` files on C and C++ build times.

Prerequisites

- Visual Studio 2022 17.8 or greater.
- C++ Build Insights is enabled by default if you install either the Desktop development with C++ workload using the Visual Studio installer:



Or the Game development with C++ workload:



Overview

Build Insights, now integrated into Visual Studio, helps you optimize your build times--especially for large projects like triple-A games. When a large header file is parsed, and especially when it's repeatedly parsed, there's an impact on build time.

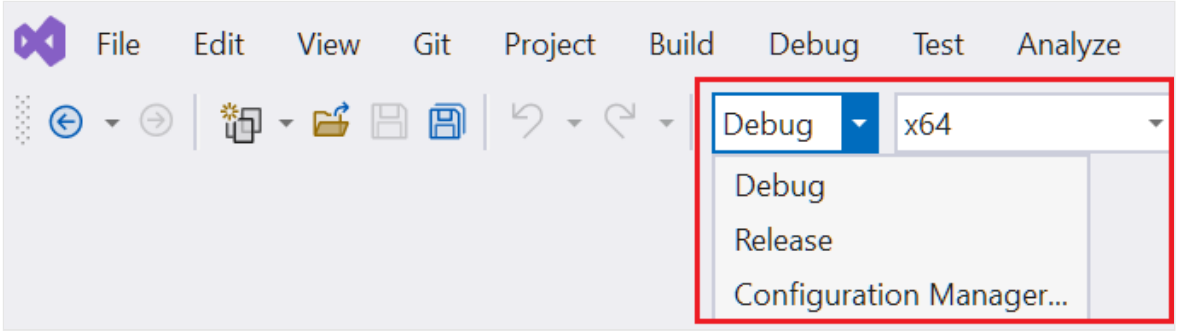
Build Insights provides analytics in the **Included Files** view, which helps diagnose the impact of parsing `#include` files in your project. It displays the time it takes to parse each header file and a view of the relationships between header files.

In this article, learn how to use the Build Insights **Included Files** and **Include Tree** views to identify the most expensive header files to parse and how to optimize build time by creating a precompiled header file.

Set build options

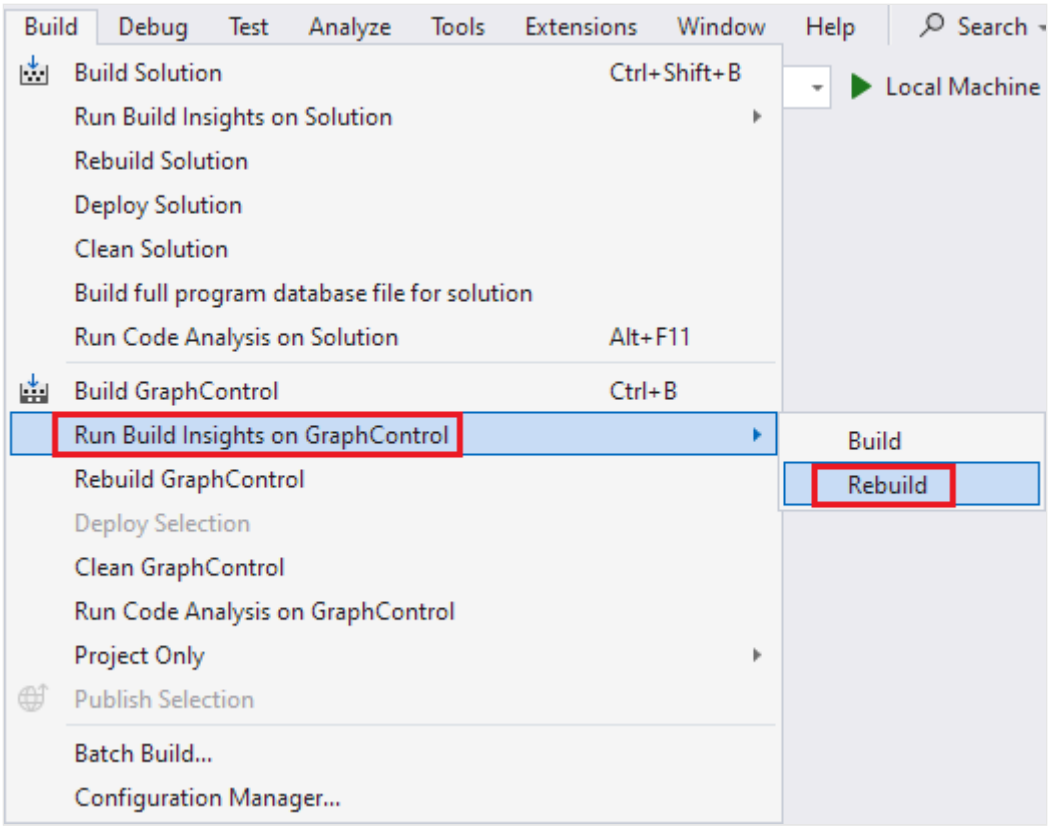
Before gathering Build Insights data, set the build options for the type of build you want to measure. For example, if you're concerned about your x64 debug build time, set the build for **Debug** and **x64**:

- In the **Solution Configurations** dropdown, choose **Debug**.
- In the **Solution Platforms** dropdown, choose **x64**.



Run Build Insights

On a project of your choosing, and using the **Debug** build options set in the previous section, run Build Insights by choosing from the main menu **Build** > **Run Build Insights on Selection** > **Rebuild**. You can also right-click a project in the solution explorer and choose **Run Build Insights** > **Rebuild**. Choose **Rebuild** instead of **Build** to measure the build time for the entire project and not for just the few files may be dirty right now.

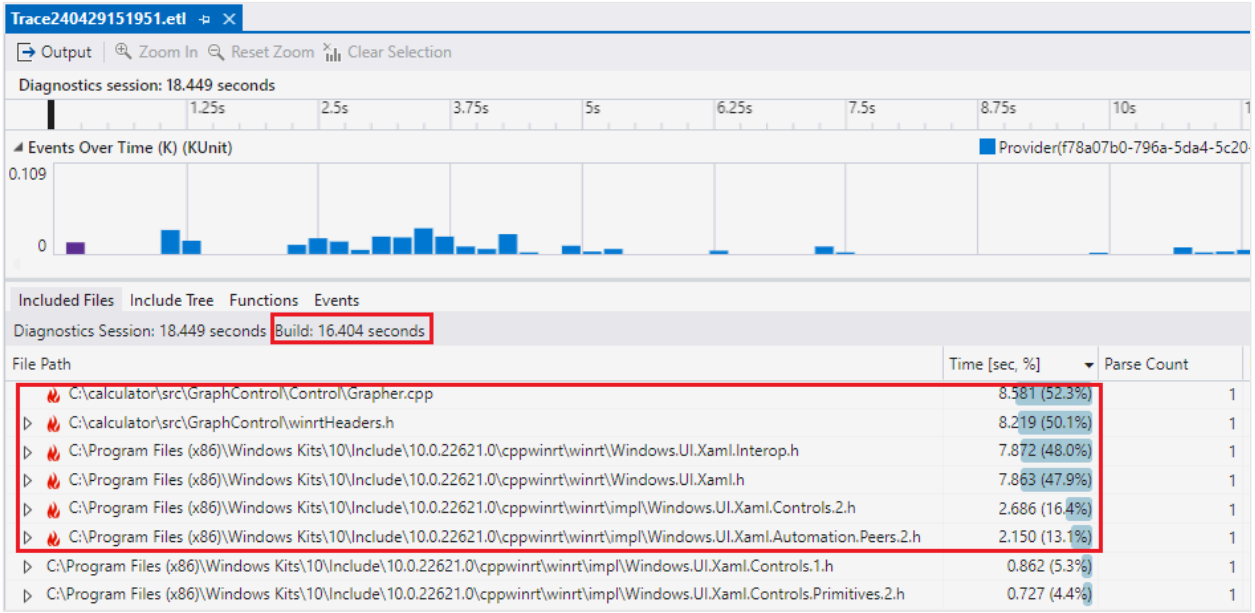


When the build finishes, an Event Trace Log (ETL) file opens. It's saved in the folder pointed to by the Windows `TEMP` environment variable. The generated name is based on the collection time.

Included Files view

The trace file shows the build time--which for this example was 16.404 seconds. The **Diagnostics Session** is the overall time taken to run the Build Insights session. Choose the **Included Files** tab.

This view shows the time spent processing `#include` files.



In the **File Path** column, some files have a fire icon next to them to indicate that they take up 10% or more of the build time.

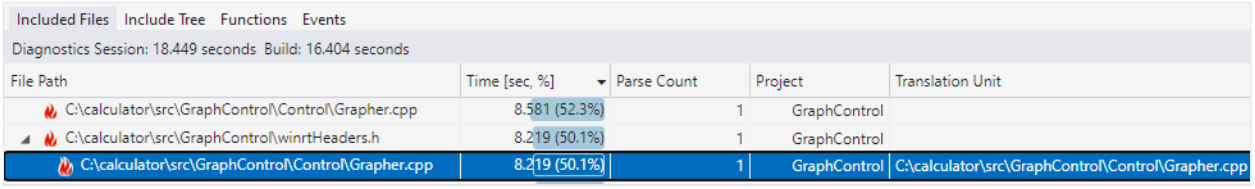
The **Time [sec, %]** column shows how long it took to compile each function in wall clock responsibility time (WCTR). This metric distributes the wall clock time it takes to parse files based on their use of parallel threads. For example, if two different threads are parsing two different files simultaneously within a one-second period, each file's WCTR is recorded as 0.5 seconds. This reflects each file's proportional share of the total compilation time, taking into account the resources each consumed during parallel execution. Thus, WCTR provides a better measure of the impact each file has on the overall build time in environments where multiple compilation activities occur simultaneously.

The **Parse Count** column shows how many time the header file was parsed.

The first header file highlighted in this list is `winrtHeaders.h` It takes 8.581 seconds of the overall 16.404-second build time, or 52.3% of the build time. The next most expensive is `Windows.UI.Xaml.Interop.h`, and then `Windows.Xaml.h`.

To see which file includes `winrtHeaders.h`, click the chevron next to it. The **Parse Count** column can be helpful by pointing out how many times a header file is included by other files. Perhaps a header file is included multiple times, which could be a sign that it's a good candidate for a precompiled header file or refactoring.

The **Translation Unit** column shows which file was being processed when the included file was processed. In this example, `winrtHeaders.h` was included while `Grapher.cpp` was compiled:



The translation unit column can help disambiguate which file was being compiled in cases where a header file is included many times and you want to find out where that happens the most.

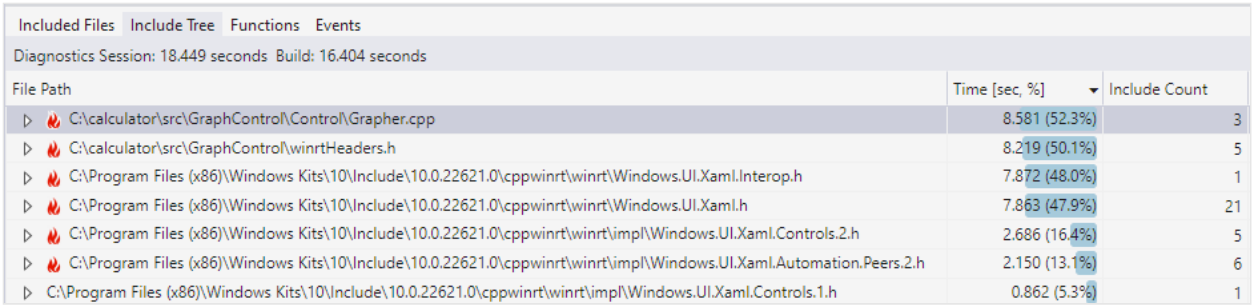
We know that `winrtHeaders.h` is expensive to parse, but we can learn more.

Include Tree view

In this view, the children nodes are the files included by the parent node. This can help you understand the relationships between header files and identify opportunities to

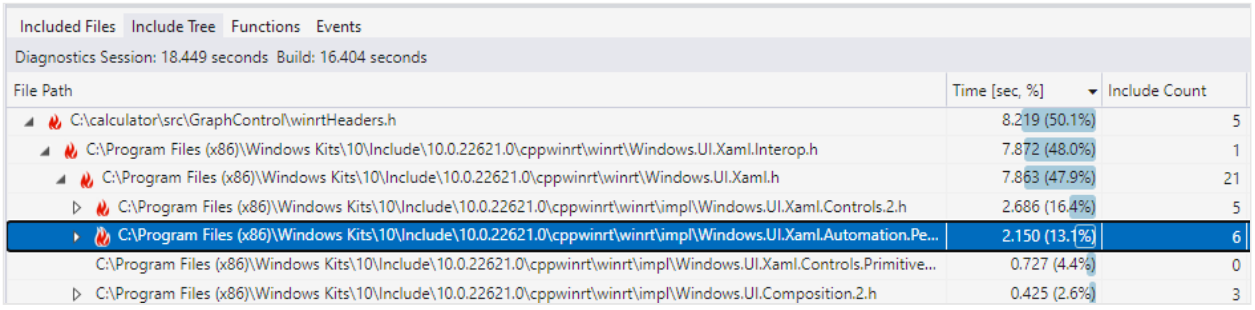
reduce the number of times a header file is parsed.

Select the **Include Tree** tab in the ETL file to see the Include Tree view:



In this view, the **File Path** column shows each file that includes other files. The **Include Count** lists how many files this header file includes. The time to parse this file is listed, and when expanded, lists the time to parse each individual header file that this header file includes.

Earlier, we saw that parsing `winrtHeaders.h` is time consuming. In the **Filter Files** text box, if we enter `winrtHeaders.h`, we can filter the view to only the entries that contain `winrtHeaders.h` in the name. Clicking the chevron next to `winrtHeaders.h` shows which files it includes:



We see that `winrtHeaders.h` includes `Windows.UI.Xaml.Interop.h`. Remember from the **Included Files** view that this was also time consuming to parse. Click the chevron next to `Windows.UI.Xaml.Interop.h` to see that it includes `Windows.UI.Xaml.h`, which includes 21 other header files, two of which are also on the hot list.

Having identified some of the most expensive header files to parse, and seeing that `winrtHeaders.h` is responsible for bringing them in, suggests that we can use a precompiled header to make including `winrtHeaders.h` faster.

Improve build time with precompiled headers

Because we know from the **Included Files** view that `winrtHeaders.h` is time consuming to parse, and because we know from the **Include Tree** view that `winrtHeaders.h` includes

several other header files that are time consuming to parse, we build a Precompiled header file (PCH) to speed that up by only parsing them once into a PCH.

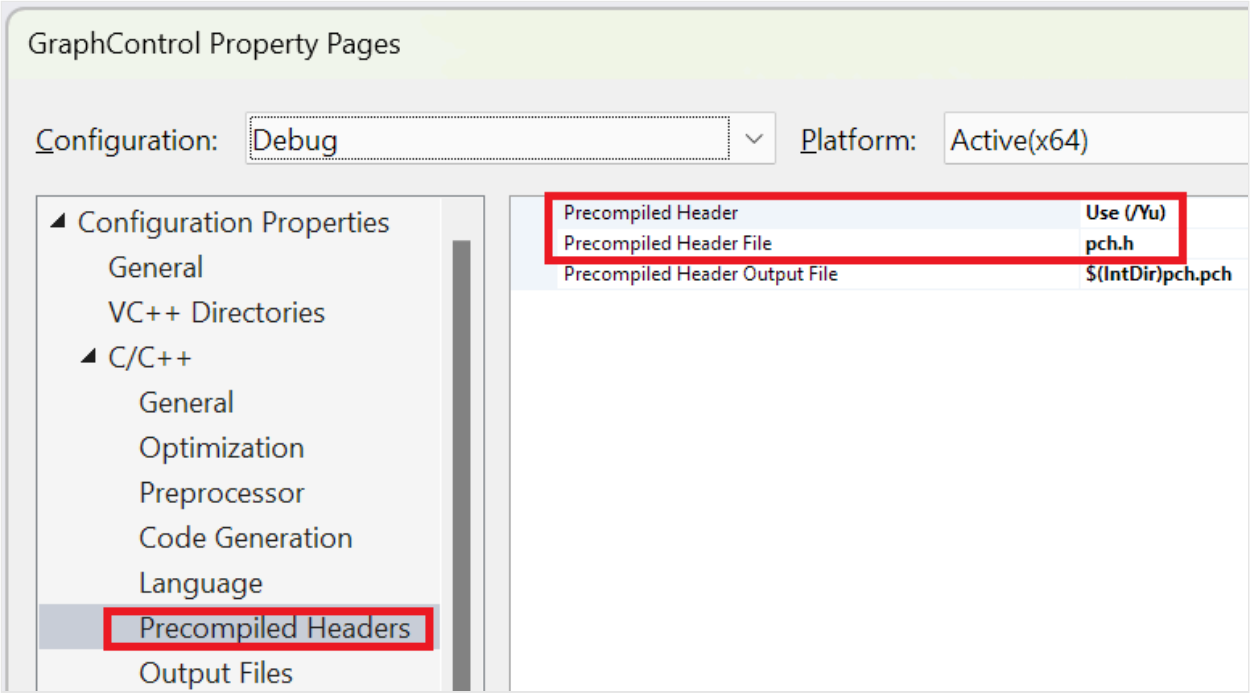
We add a `pch.h` to include `winrtHeaders.h`, which would look like this:

```
C++

#ifndef CALC_PCH
#define CALC_PCH

#include <winrtHeaders.h>

#endif // CALC_PCH
```

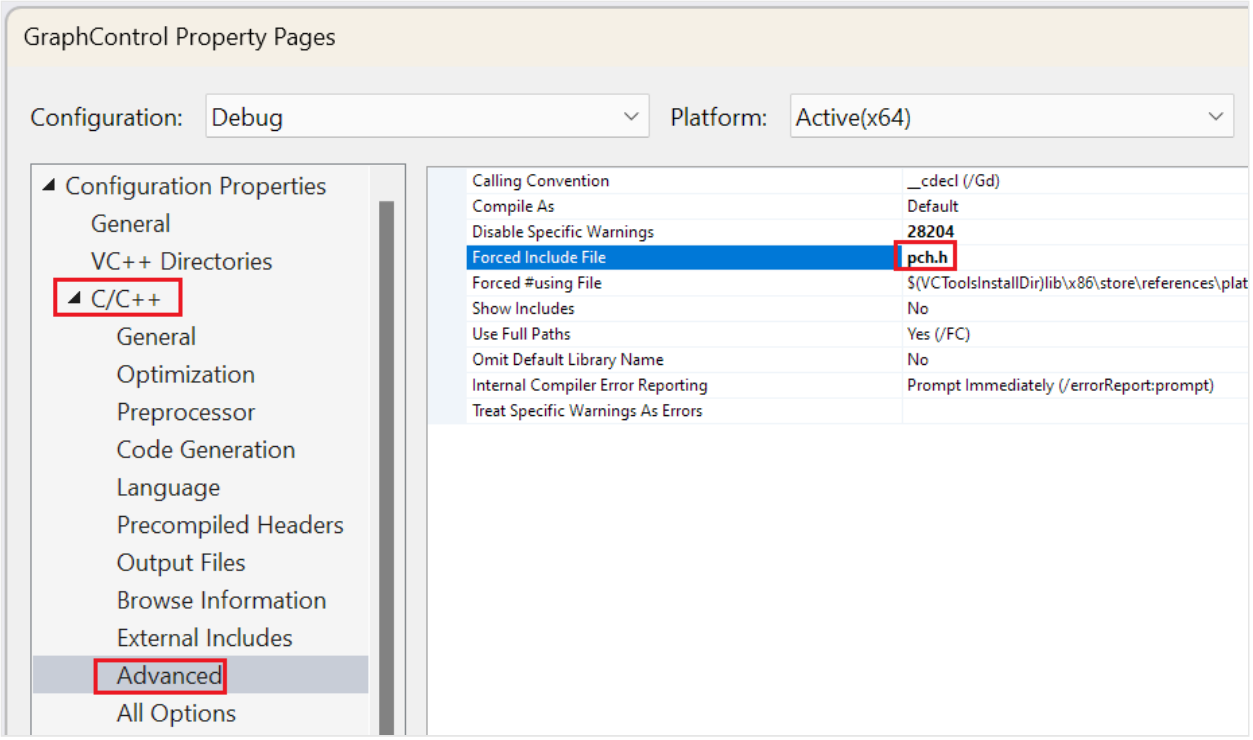
PCH files must be compiled before they can be used, so we add a file to the project, arbitrarily named `pch.cpp`, that includes `pch.h`. It contains one line:

```
C++

#include "pch.h"
```

Then we set our project to use the PCH. That's done in project properties via **C/C++** > **Precompiled Headers** and setting **Precompiled Header** to **Use (/Yu)** and **Precompiled Header File** to **pch.h**.



To use the PCH, we include it as the first line in the source files that use `winrtHeaders.h`. It must come before any other include files. Or, for simplicity, we could modify the project properties to include `pch.h` at the beginning of every file in the solution by setting the project property: **C/C++** > **Advanced** > **Forced Include File** to `pch.h`:



Since the PCH includes `winrtHeaders.h`, we could remove `winrtHeaders.h` from all the files that currently include it. It's not strictly necessary because the compiler realizes that `winrtHeaders.h` is already included and doesn't parse it again. Some developers prefer to keep the `#include` in the source file for clarity, or in case the PCH is likely to be refactored and may not include that header file anymore.

Test the changes

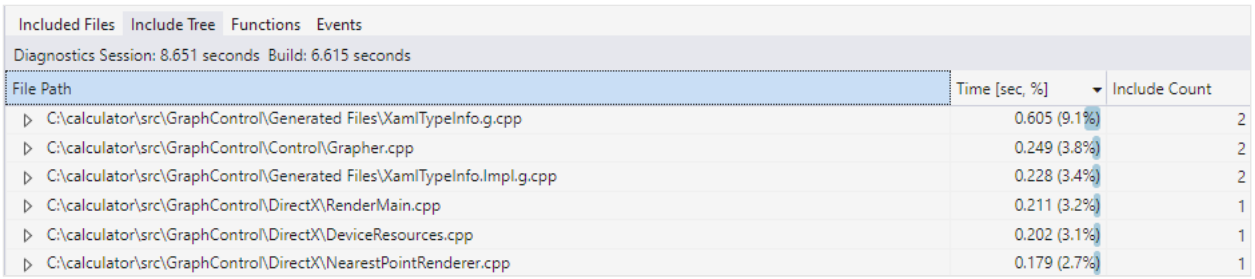
We first clean the project to make sure we're comparing building the same files as before. To clean just one project, right-click the project in the **Solution Explorer** and choose **Project only** > **Clean only <prj name>**.

Because this project now uses a precompiled header (PCH), we don't want to measure the time spent building the PCH because that only happens once. We do this by loading the `pch.cpp` file and choosing **Ctrl+F7** to build just that file. We could also compile this file by right-clicking `pch.cpp` in the Solution Explorer and choosing `Compile`.

Now we rerun Build Insights in the **Solution Explorer** by right-clicking the project and choosing **Project Only** > **Run Build Insights on Build**. You can also right-click a project in the solution explorer and choose **Run Build Insights** > **Build**. We don't want **Rebuild** this time because that will rebuild the PCH, which we don't want to measure. We cleaned the project earlier, which means that a normal build compiles all the project files we want to measure.

When the ETL files appear, we see that build time went from 16.404 seconds to 6.615 seconds. Put `winrtHeaders.h` into the filter box and nothing appears. This is because the

time spent parsing it is now negligible since it's being pulled in by the precompiled header.

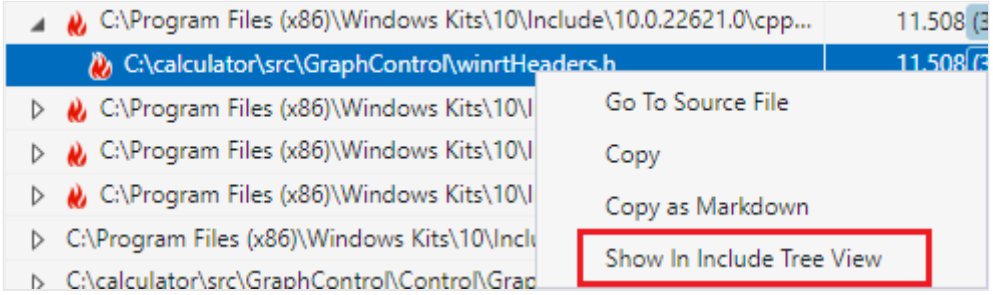


This example uses precompiled headers because they're a common solution before C++20. However, starting with C++20, there are other, faster, less brittle, ways to include header files--such as header units and modules. For more information, see Compare header units, modules, and precompiled headers.

Navigate between views

There are some navigation features for both the **Included Files** and **Include Tree** views:

- Double-click a file (or press **Enter**) in either the **Included Files** or **Include Tree** to open the source code for that file.
- Right-click on a header file to find that file in the other view. For example, in the **Included File**s view, right-click on `winrtHeaders.h` and choose **Find in Include Tree** to see it in the **Include Tree** view.



Or, you can right-click a file in the **Include Tree** view to jump to it in the **Included Files** view.

Tips

- You can **File** > **Save As** the ETL file to a more permanent location to keep a record of the build time. You can then compare it to future builds to see if your changes are improving build time.
- If you inadvertently close the Build Insights window, reopen it by finding the `<dateandtime>.etl` file in your temporary folder. The `TEMP` Windows environment

variable provides the path of your temporary files folder.

- To dig into the Build Insights data with Windows Performance Analyzer (WPA), click the **Open in WPA** button in the bottom right of the ETL window.
- Drag columns to change the order of the columns. For instance, you may prefer moving the **Time** column to be the first column. You can hide columns by right-clicking on the column header and deselecting the columns you don't want to see.
- The **Included Files** and **Include Tree** views provide a filter box to find a header file that you're interested in. It does partial matches on the name you provide.
- Sometimes the parse time reported for a header file is different depending on which file includes it. This can be due to the interplay of different `#define`s that affect which parts of the header are expanded, file caching, and other system factors.
- If you forget what the **Included Files** or **Include Tree** view is trying to show you, hover over the tab to see a tooltip that describes the view. For example, if you hover over the **Include Tree** tab, the tooltip says: "View that shows include statistics for every file where the children nodes are the files included by the parent node."
- You may see cases (like `Windows.h`) where the aggregated duration of all the times for a header file is longer than the duration of the entire build. What's happening is that headers are being parsed on multiple threads at the same time. If two threads simultaneously spend one second parsing a header file, that's 2 seconds of build time even though only one second of wall clock time has gone by. For more information, see wall clock responsibility time (WCTR).

Troubleshooting

- If the Build Insights window doesn't appear, do a rebuild instead of a build. The Build Insights window doesn't appear if nothing actually builds; which may be the case if no files changed since the last build.
- If a header file you're interested in doesn't appear in the **Included Files** or **Include Tree** views, it either didn't build or its build time isn't significant enough to be listed.

See also

Compare header units, modules, and precompiled headers
Build Insights in Visual Studio video - Pure Virtual C++ 2023
Faster C++ builds, simplified: a new metric for time
Tutorial: Troubleshoot function inlining on build time
Tutorial: vcperf and Windows Performance Analyzer

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Build Insights tips and tricks

Article • 11/12/2024

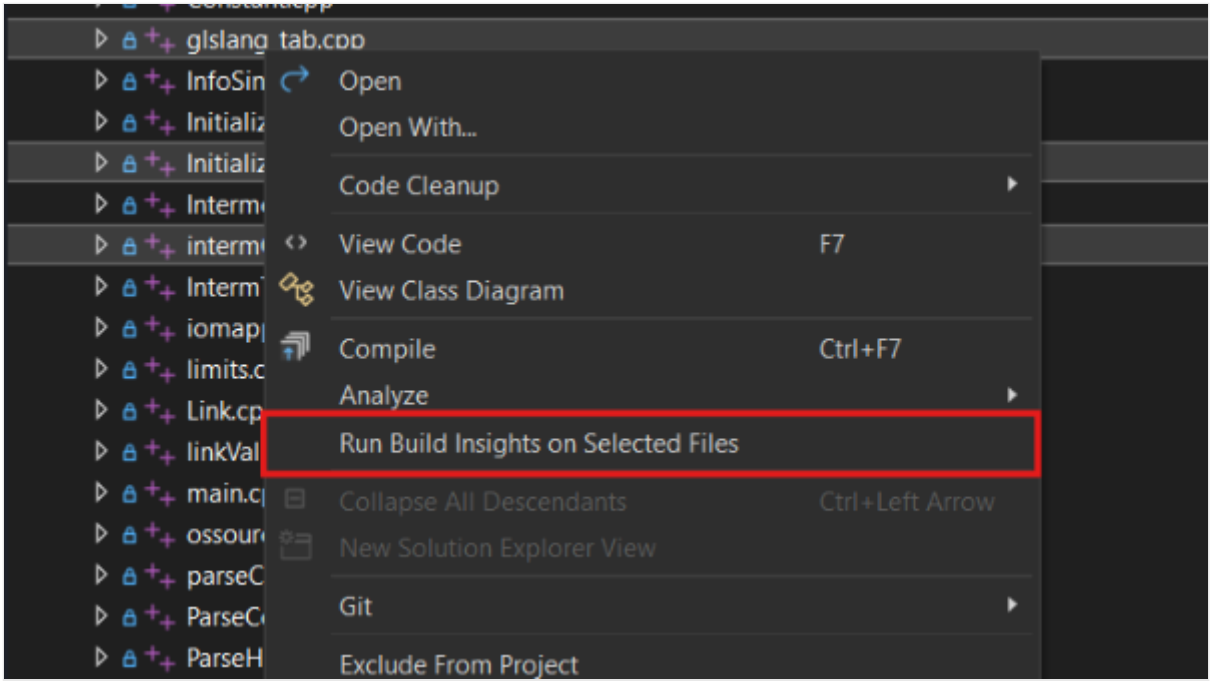
Learn time-saving tips for using Build Insights.

Run Build Insights on selected files

This feature requires Visual Studio 2022 17.12 or later.

If you're working on a specific file or files, and want to see how they impact your build time, you can run Build Insights on just those files. This feature is useful when you want to focus on a subset of files in your project.

To try it, in **Solution Explorer** select the files in your project you want to profile, right-click, and choose **Run Build Insights on Selected Files**:

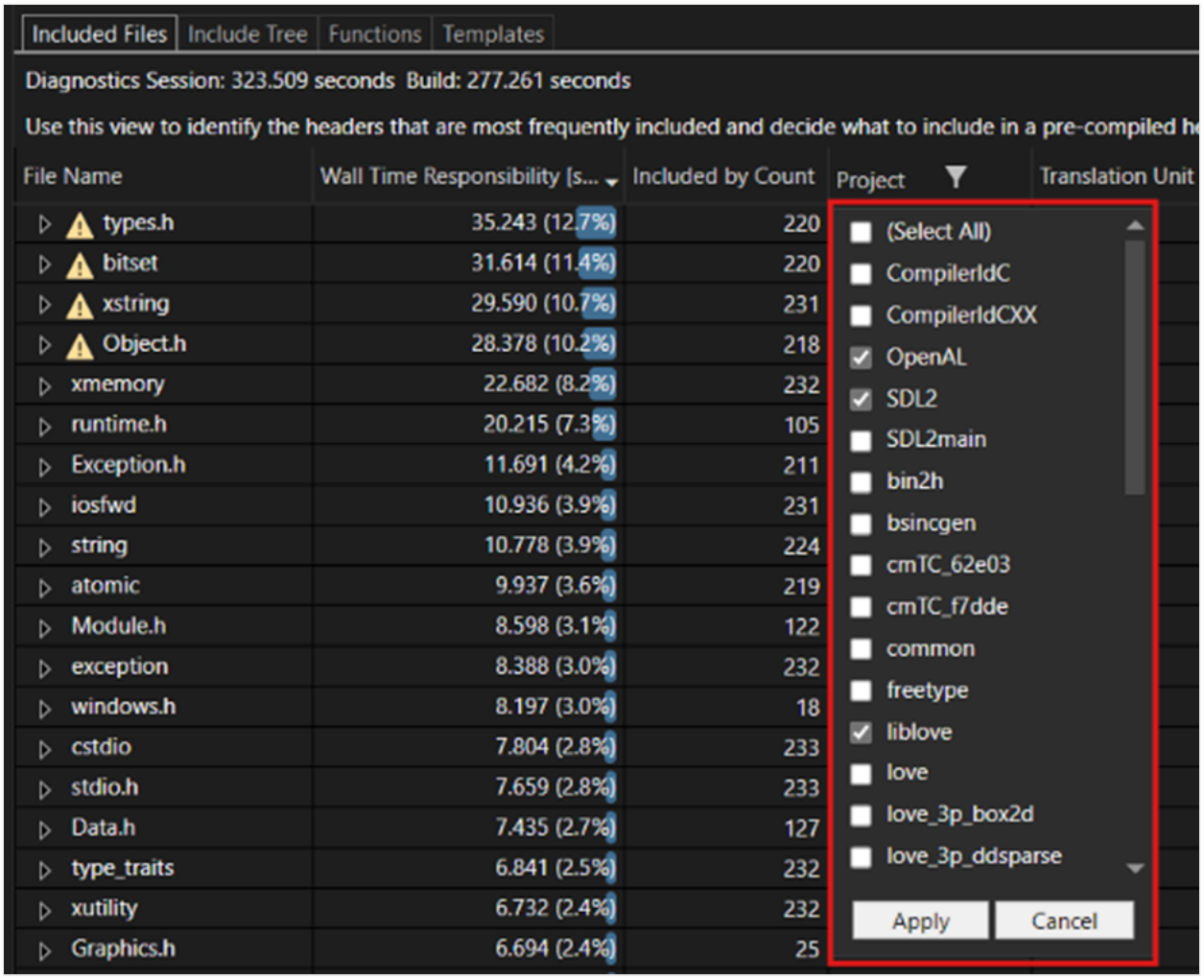


Filter Build Insights results

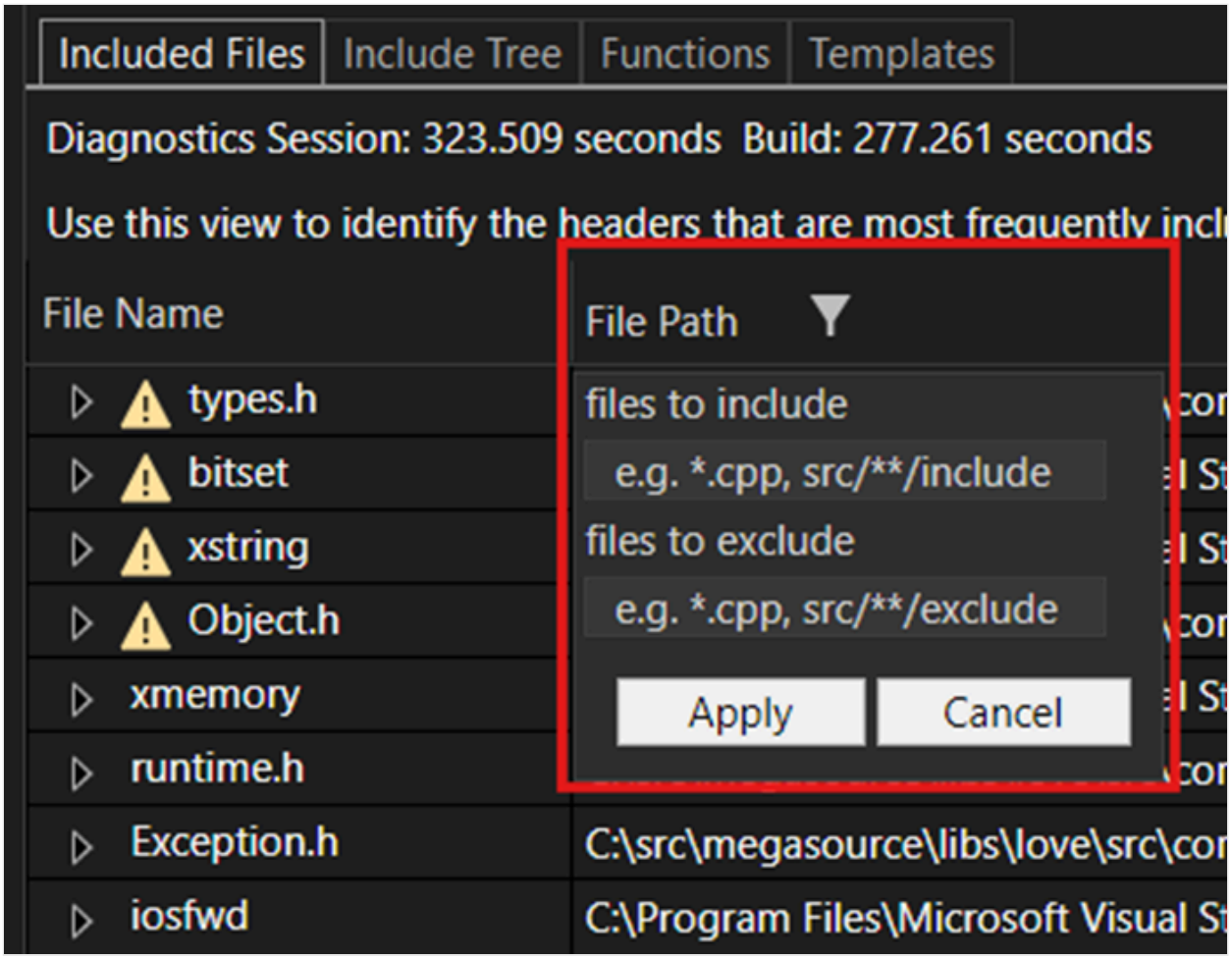
This feature requires Visual Studio 2022 17.12 or later.

If you have a large solution with many projects, you can filter the Build Insights results to see files only the projects you're interested in. This feature is useful when you want to focus on a subset of projects in your solution.

To try it, click the filter button on the filter column header and select the projects you want to see results for:



You can also use file wildcards to filter results. The search is case-insensitive:



This allows you to exclude files from a specific folder or only include files from a specific folder. For example, if your source is located at `C:\src\`, you could include files only from the renderer directory and its subdirectories by putting `C:/src/dev/renderer/**` into the **files to include** text box.

Here are some other examples:

- All files in the renderer directory: `C:/src/dev/renderer/*`
- All files in the `C:/src/dev/renderer/` directory *and all its subdirectories*: `C:/src/dev/renderer/**`
- All header files in the `C:/src/dev/renderer/` directory *and all its subdirectories*: `C:/src/dev/renderer/**/*.h`

For more examples, see the online glob pattern tester.

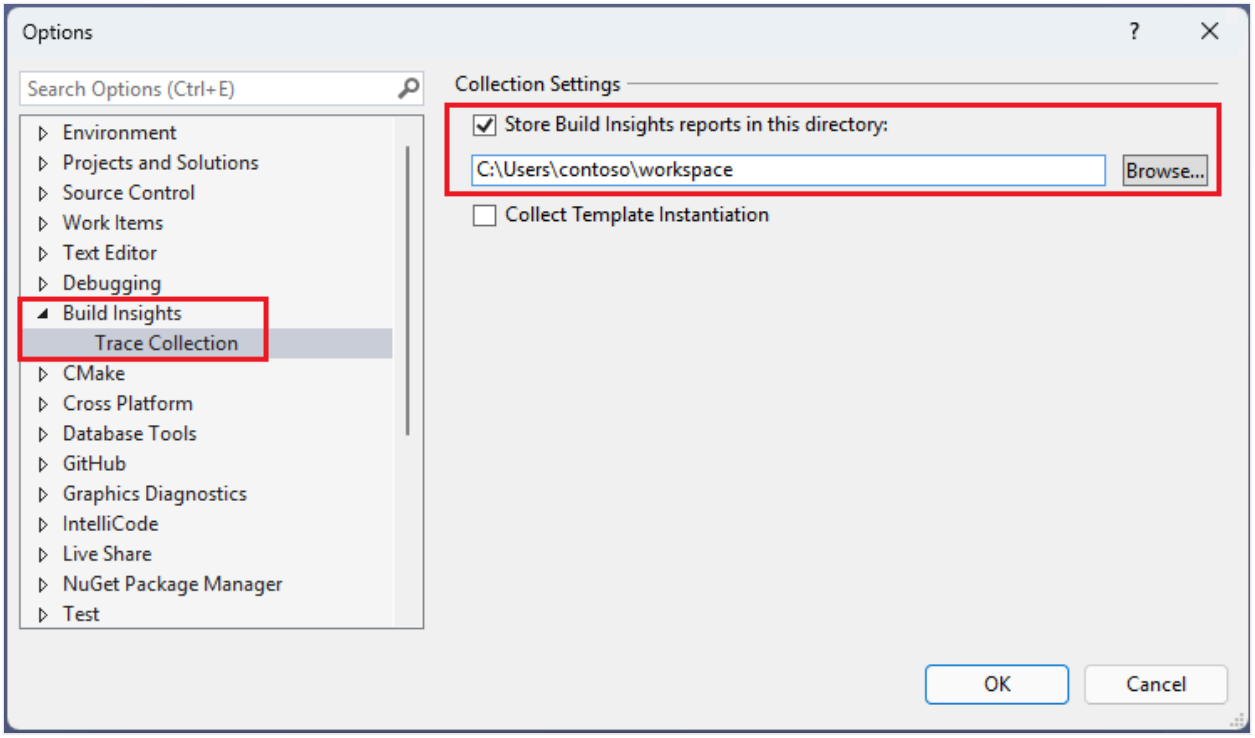
The filter you enter into either text box persists per solution. Filtering by wildcards isn't supported for CMAKE projects.

Save Build Insights reports to a designated folder

This feature requires Visual Studio 2022 17.12 or later.

Now you can designate a folder to automatically save Build Insight reports to so you can easily access them.

To set the designated folder, go to **Tools** > **Options** > **C++ Build Insights** > **Trace Collection**. Set a path in the **Store Build Insights reports in this directory** field:

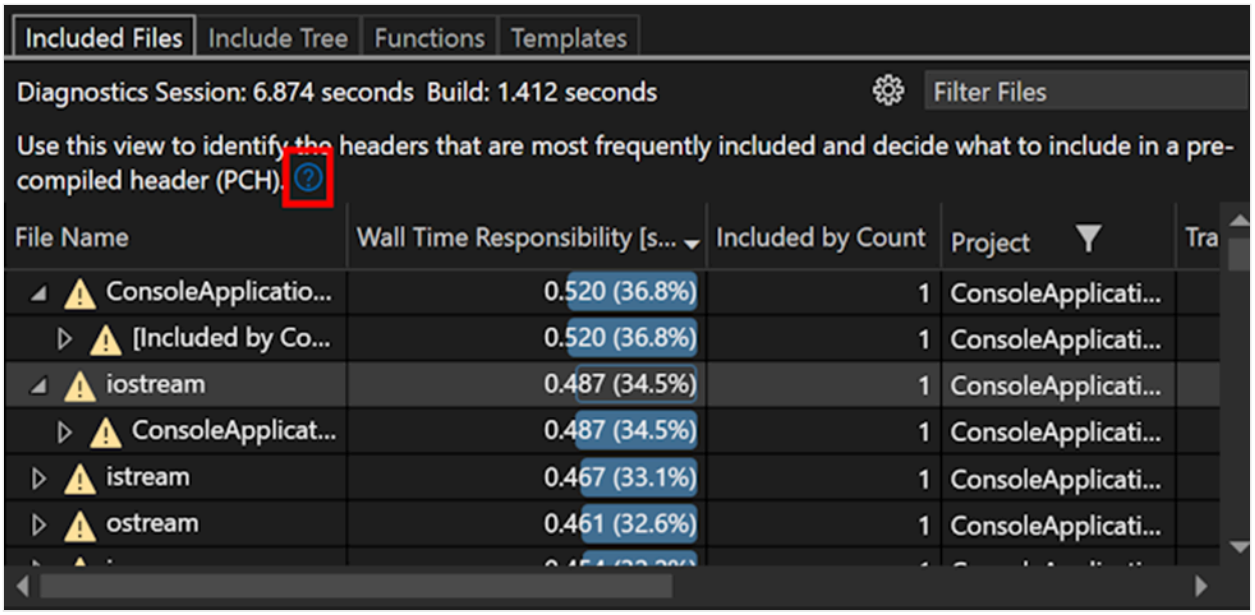


Reports are automatically saved to this folder when you run Build Insights. If a path isn't set, the `TEMP` folder is used.

Get help about the Build Insight window

This feature requires Visual Studio 2022 17.12 or later.

To see a short description for the tabs in the Build Insights window, along with a link to the documentation for a detailed explanation, click the question mark icon in the Build Insights window:



See also

Build Insights in Visual Studio video - Pure Virtual C++ 2023
Improving code generation time with C++ Build Insights
Troubleshoot header file impact on build time
Tutorial: Troubleshoot function inlining on build time

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Tutorial: vcperf and Windows Performance Analyzer

Article • 12/09/2021

In this tutorial, you'll learn how to use `vcperf.exe` to collect a trace of your C++ build. You'll also learn how to view this trace in Windows Performance Analyzer.

Step 1: Install and configure Windows Performance Analyzer

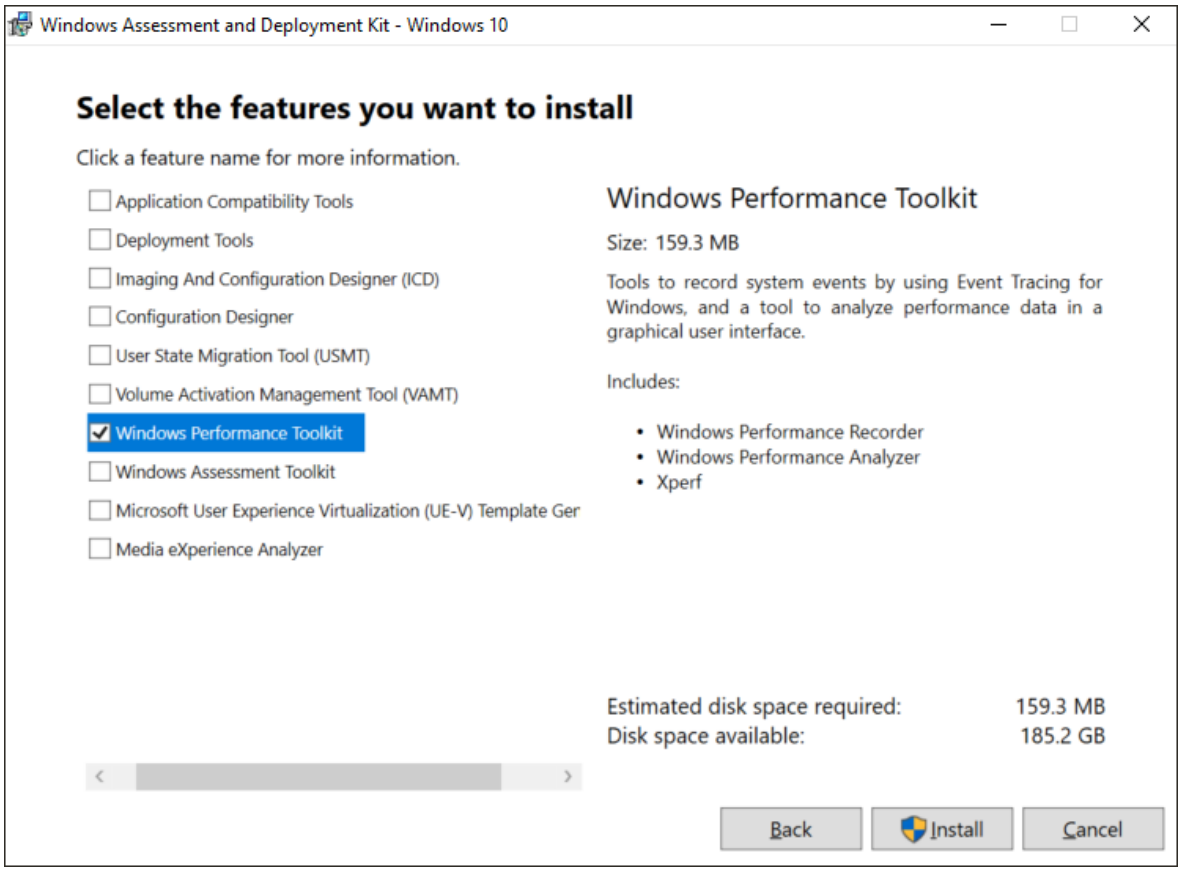
WPA is a trace viewer available in the Windows Assessment and Deployment Kit (ADK). It's a separate utility that's not part of the components you can install with the Visual Studio installer.

A version of WPA that supports C++ Build Insights is only available in versions of the Windows ADK numbered 10.1.19041.0 or later.

To download and install WPA

NOTE: Windows 8 or above is required for installing the Windows Performance Analyzer.

1. Browse to the Windows ADK download page.
2. Download and install the latest version of the Windows ADK.
3. When prompted for the features that you want to install, select the **Windows Performance Toolkit**. You may select other features if you wish, but they're not required to install WPA.



To configure WPA

Viewing C++ Build Insights traces in WPA requires a special add-in. Follow these steps to install it:

1. Obtain the add-in by downloading one of the components below. You don't need to get both. Choose the one that you find most convenient.

    - Visual Studio 2019 version 16.6 and above. For the latest version, see Visual Studio Downloads.
    - C++ Build Insights NuGet package.

2. Copy the `perf_msvcbuildinsights.dll` file into your WPA installation directory.
    a. In Visual Studio 2019 version 16.6 and above, this file is located here: `C:\Program Files (x86)\Microsoft Visual Studio\{Year}\{Edition}\VC\Tools\MSVC\{Version}\bin\Host{Architecture}\{Architecture}`.
    b. In the C++ Build Insights NuGet package, this file is located here: `wpa\{Architecture}`.
    c. In the paths above, replace the variables surrounded by curly brackets as follows:
        i. `{Year}` is your Visual Studio product year, such as 2019 or 2022.
        ii. `{Edition}` is your Visual Studio edition such as Community, Professional, or Enterprise.

iii. `{Version}` is your MSVC version. Choose the highest one available.
iv. `{Architecture}`: choose `x64` if you have a 64-bit version of Windows. Otherwise, choose `x86`.

d. The WPA installation directory is typically: `C:\Program Files (x86)\Windows Kits\10\Windows Performance Toolkit`.

3. In your WPA installation directory, open the `perfcore.ini` file and add an entry for `perf_msvcbuildinsights.dll`.

Step 2: Trace your build with vcperf.exe

To view C++ Build Insights data, first collect it into a trace file by following these steps:

1. Open an **x64** or **x86 Native Tools Command Prompt for VS** in administrator mode. (Right-click the Start menu item and choose **More** > **Run as administrator**.)
   a. Choose **x64** if you have a 64-bit version of Windows. Otherwise, choose **x86**.

2. In the command prompt window, enter this command:

   vcperf.exe /start *SessionName*

   Choose a session name you'll remember for *SessionName*.

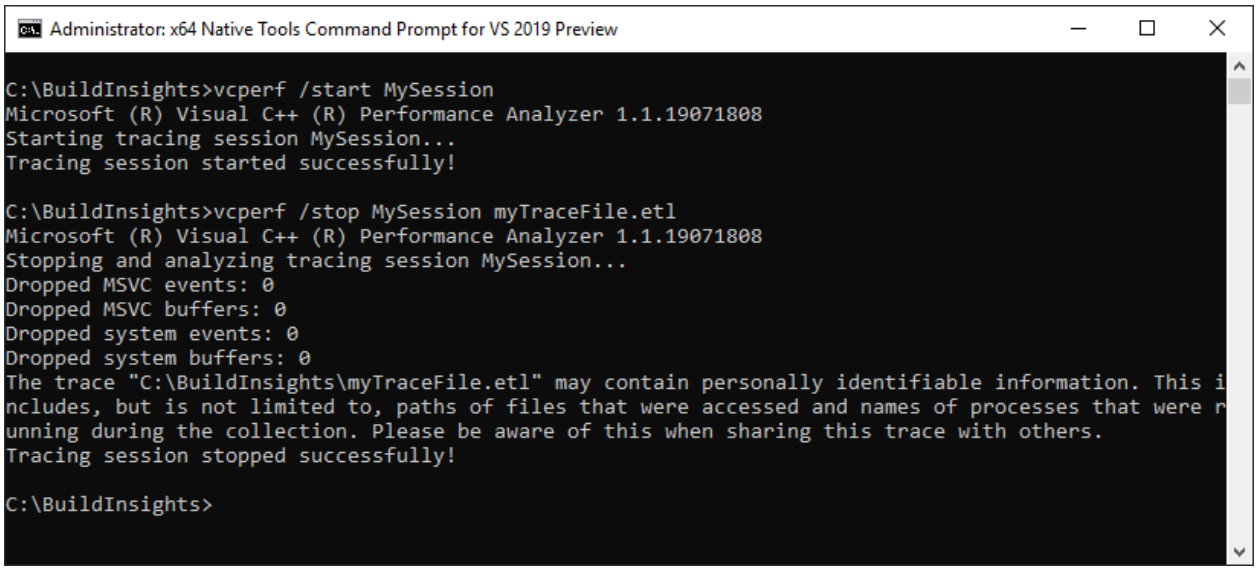
3. Build your project as you normally would. You don't need to use the same command prompt window to build.

4. In the command prompt window, enter this command:

   vcperf.exe /stop *SessionName traceFile.etl*

   Use the same session name you chose for *SessionName* before. Choose an appropriate name for the *traceFile.etl* trace file.

Here's what a typical *vcperf.exe* command sequence looks like in a developer command prompt window:



Important notes about vcperf.exe

- Administrator privileges are required to start or stop a *vcperf.exe* trace. Use a developer command prompt window that you open by using **Run as administrator**.
- Only one tracing session at a time may run on a machine.
- Make sure to remember the session name you used to start your trace. It can be troublesome to stop a running session without knowing its name.
- Just like *cl.exe* and *link.exe*, the command-line utility *vcperf.exe* is included in an MSVC installation. No additional steps are required to obtain this component.
- *vcperf.exe* collects information about all MSVC tools running on your system. As a result, you don't have to start your build from the same command prompt you used to collect the trace. You can build your project from either a different command prompt, or even in Visual Studio.

vcperf.exe is open-source

If you wish to build and run your own version of *vcperf.exe*, feel free to clone it from the vcperf GitHub repository.

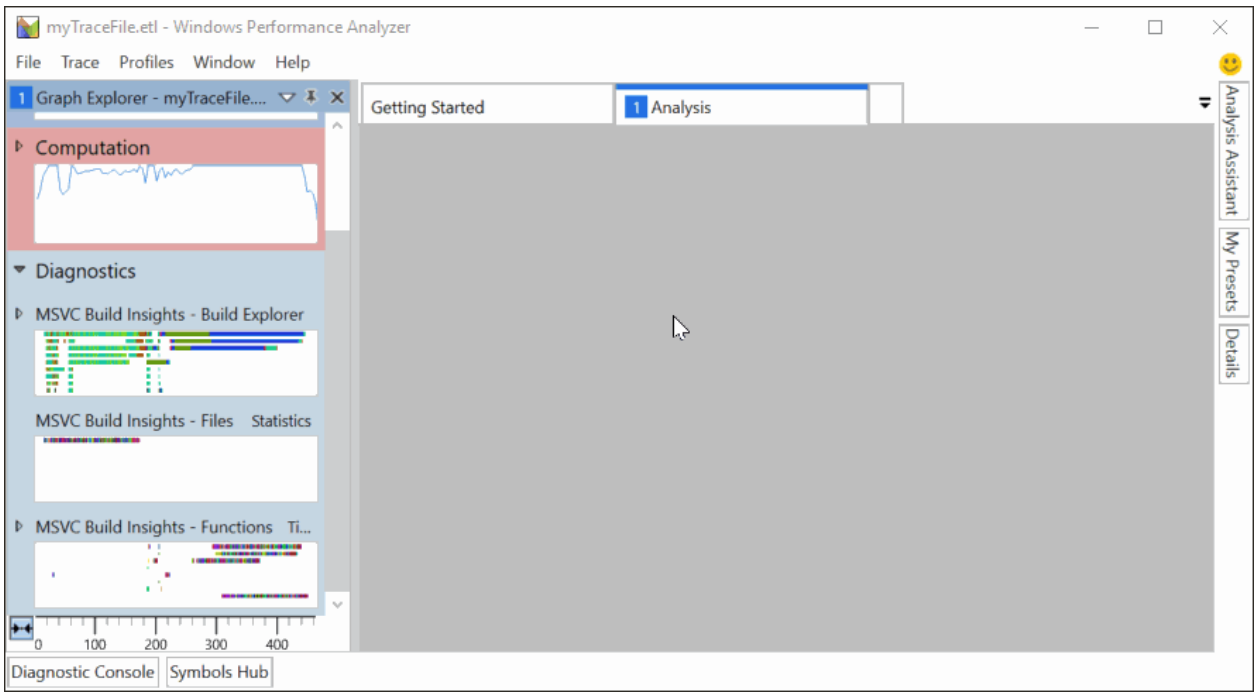
Step 3: View your trace in Windows Performance Analyzer

Launch WPA and open the trace you just collected. WPA should recognize it as a C++ Build Insights trace, and the following views should appear in the Graph Explorer panel

on the left:

- Build Explorer
- Files
- Functions
- Template Instantiations

If you can't see these views, double-check that WPA is configured correctly, as described in Step 1. You can view your build data by dragging the views into the empty Analysis window on the right, as shown here:



Other views are available in the Graph Explorer panel. Drag them into the Analysis window when you're interested in the information they contain. A useful one is the CPU (Sampled) view, which shows CPU utilization throughout your build.

More information

Tutorial: Windows Performance Analyzer basics
Learn about common WPA operations that can help you analyze your build traces.

Reference: vcperf commands
The *vcperf.exe* command reference lists all the available command options.

Reference: Windows Performance Analyzer views
Refer to this article for details about the C++ Build Insights views in WPA.

Windows Performance Analyzer
The official WPA documentation site.

Tutorial: Windows Performance Analyzer basics

Article • 10/29/2021

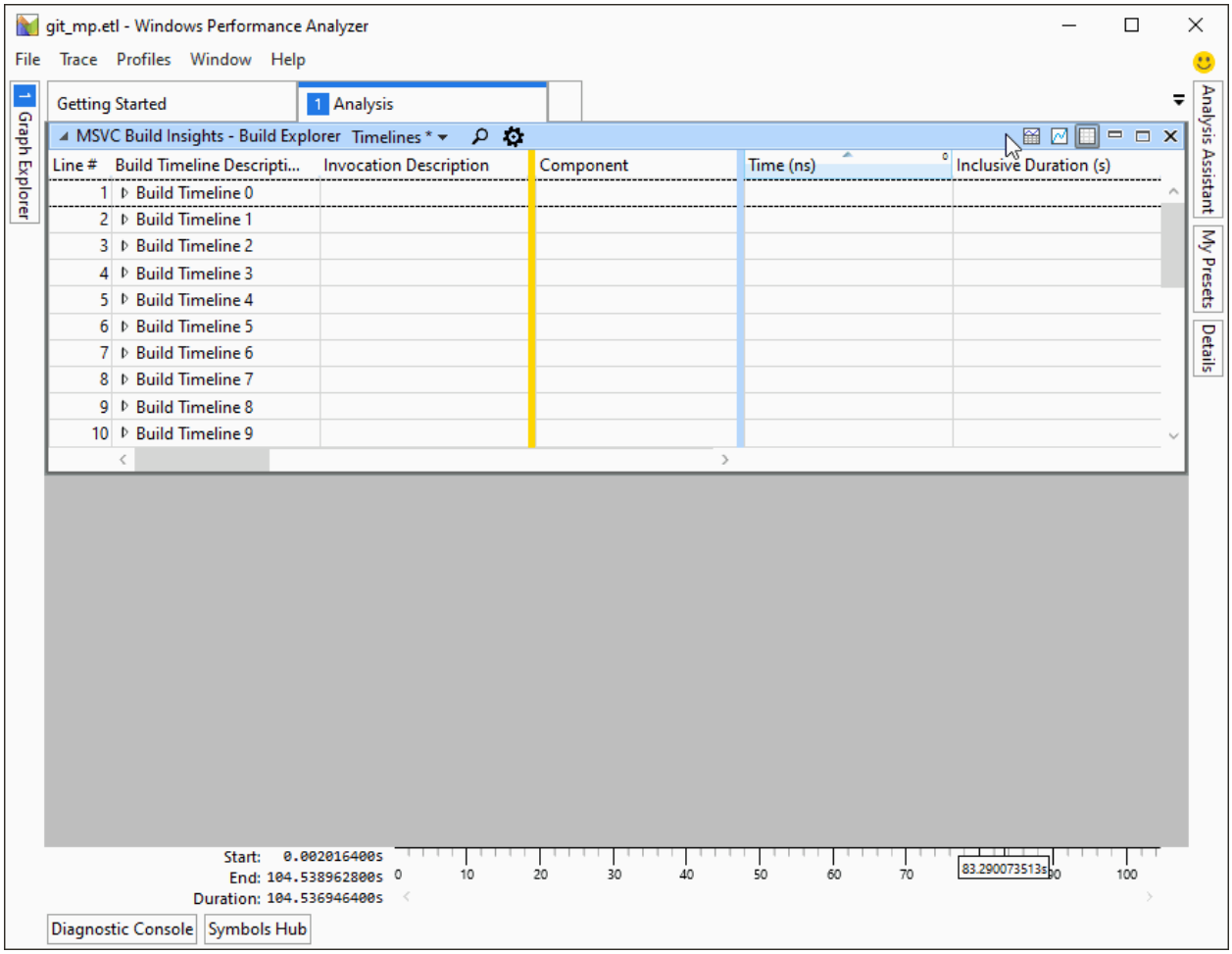
Using C++ Build Insights effectively requires some knowledge of Windows Performance Analyzer (WPA). This article helps you become familiar with common WPA operations. For more information on how to use WPA, see the Windows Performance Analyzer documentation.

Change the view mode

WPA offers two basic view modes for you to explore your traces:

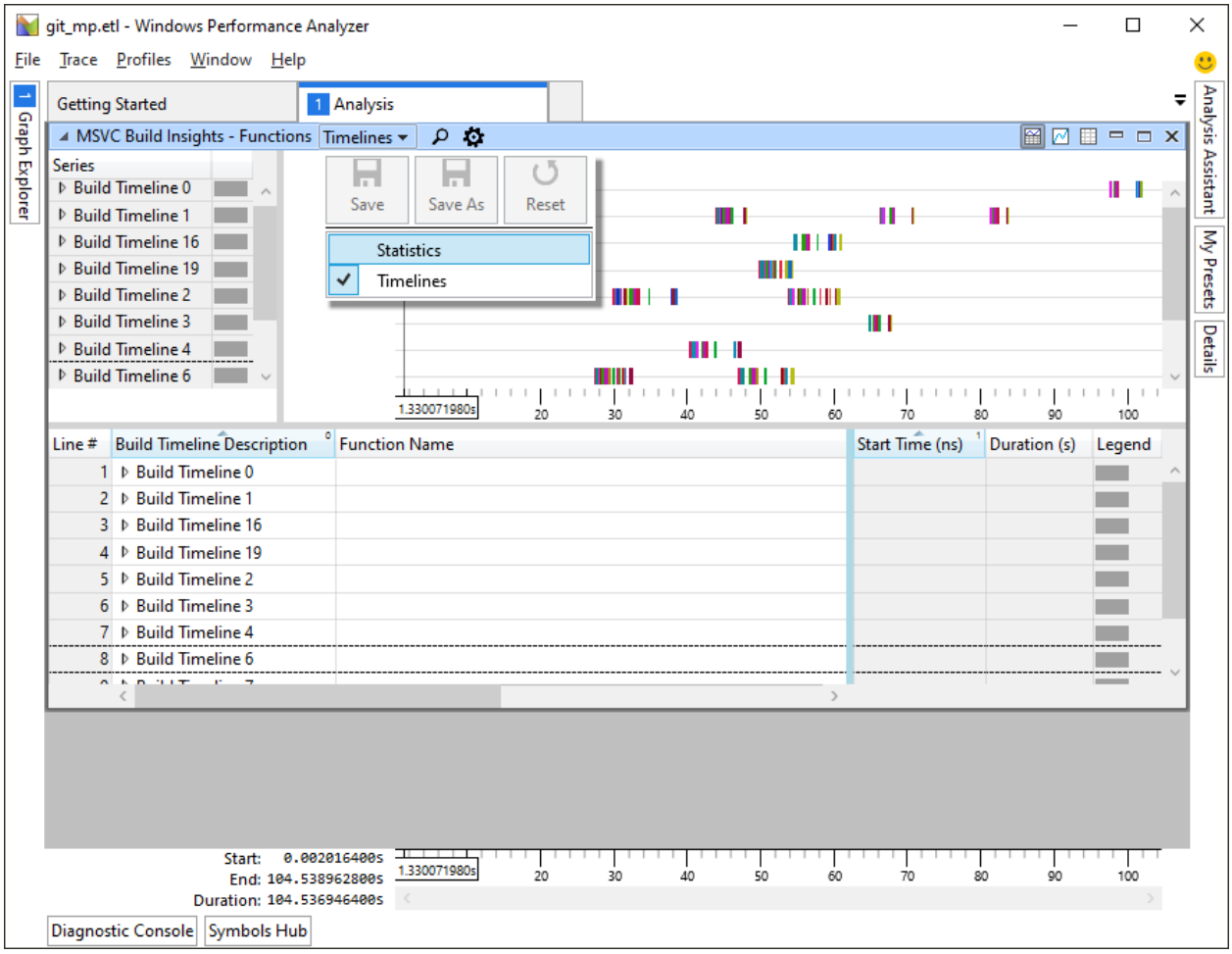
- graph mode, and
- table mode.

You can switch between them by using the view mode icons in the top of the view pane:



Select presets

Most C++ Build Insights WPA views have multiple presets for you to choose from. You can select the preset you want by using the drop-down menu in the top of the view pane:



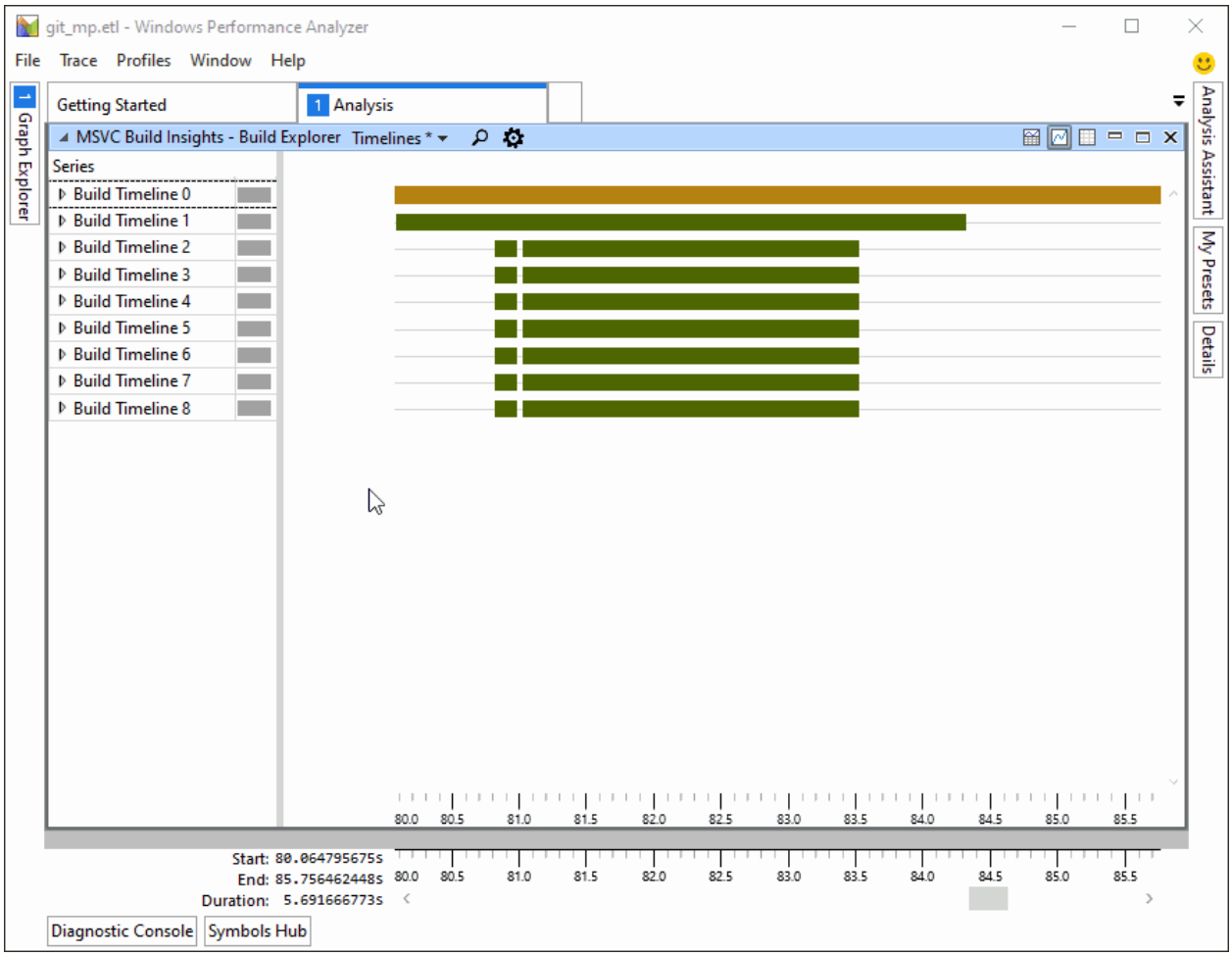
Zoom in and out

Some build traces are so large it's hard to make out the details. To zoom in on an area that interests you, right-click on the graph and select **Zoom**. You can always go back to the previous setting by choosing **Undo Zoom**. This image shows an example of using a selection and the **Zoom** command to zoom in on a section of the graph:

Group by different columns

You can customize the way your trace is displayed. Click on the gear icon at the top of a view pane and rearrange the columns in the Build Explorer View Editor. The columns found above the yellow line in this dialog are the ones your data rows are grouped by. The column right above the yellow line is special: in the graph view, it's displayed on the colored bars.

This image shows an example bar graph of a link invocation. We use the gear icon to open the Build Explorer View Editor dialog. Then we drag the Component and Name column entries above the yellow line. The configuration is changed to increase the level of detail, and to see what actually happened inside the linker:



See also

Tutorial: vcperf and Windows Performance Analyzer

Reference: vcperf commands

Reference: Windows Performance Analyzer views

Windows Performance Analyzer

Reference: vcperf commands

Article • 02/23/2023

This article lists and describes the commands available in `vcperf.exe`, and how to use them.

Commands to start and stop traces

> **ⓘ Important**
>
> Unless you specify `/noadmin`, the following commands require administrative privileges.

Option	Arguments and description
`/start`	[`/noadmin`] [`/nocpusampling`] [`/level1` | `/level2` | `/level3`] `<sessionName>`
	Starts a trace under the given session name. The `/noadmin` option runs *vcperf.exe* without admin privileges, and it ignores the `/nocpusampling` option. When you run vcperf without admin privileges, there can be more than one active session on a given machine. The `/nocpusampling` option specifies *vcperf.exe* doesn't collect CPU samples. It prevents the use of the CPU Usage (Sampled) view in Windows Performance Analyzer, but makes the collected traces smaller. The `/level1`, `/level2`, or `/level3` options specify which MSVC events to collect, in increasing level of information. Level 3 includes all events. Level 2 includes all events except template instantiation events. Level 1 includes all events except template instantiation, function, and file events. If unspecified, `/level2` is selected by default. Once *vcperf.exe* starts the trace, it returns immediately. The trace collects events system-wide for all processes running on the machine. That means that you don't need to build your project in the same command prompt window as the one you use to run *vcperf.exe*. For example, you can build your project in Visual Studio.
`/stop`	(1) [`/templates`] `<sessionName> <outputFile.etl>` (2) [`/templates`] `<sessionName> /timetrace <outputFile.json>`

Option	Arguments and description
	Stops the trace identified by the given session name. Runs a post-processing step on the trace to generate a file specified by the `<outputFile>` parameter. The `/templates` option includes template instantiation events in the file. (1) Generates a file viewable in Windows Performance Analyzer (WPA). The output file requires a `.etl` extension. (2) Generates a file viewable in the Microsoft Edge trace viewer (`edge://tracing`). The output file requires a `.json` extension.
`/stopnoanalyze`	`<sessionName> <rawOutputFile.etl>`
	Stops the trace identified by the given session name and writes the raw, unprocessed data in the specified output file. The resulting file isn't meant for viewing in WPA. The post-processing step involved in the `/stop` command can sometimes be lengthy. You can use the `/stopnoanalyze` command to delay this post-processing step. Use the `/analyze` command when you're ready to produce a file viewable in Windows Performance Analyzer or the Microsoft Edge trace viewer.

Miscellaneous commands

Option	Arguments and description
`/analyze`	(1) [`/templates`] `<rawInputFile.etl> <outputFile.etl>` (2) [`/templates`] `<rawInputFile.etl> /timetrace <outputFile.json>`
	Accepts a raw trace file produced by the `/stopnoanalyze` command. Runs a post-processing step on this trace to generate the file specified by the `<outputFile>` parameter. The `/templates` option includes template instantiation events in the file. (1) Generates a file viewable in Windows Performance Analyzer (WPA). The output file requires a `.etl` extension. (2) Generates a file viewable in the Microsoft Edge trace viewer (`edge://tracing`). The output file requires a `.json` extension.

See also

Get started with C++ Build Insights

Tutorial: Windows Performance Analyzer basics

Reference: Windows Performance Analyzer views

Windows Performance Analyzer

Reference: Windows Performance Analyzer views

Article • 10/29/2021

This article provides details on each of the C++ Build Insights views available in Windows Performance Analyzer (WPA). Use this page to find:

- data column descriptions; and
- available presets for each view, including their intended use and preferred viewing mode.

If you're new to WPA, we recommend that you first familiarize yourself with the basics of WPA for C++ Build Insights.

Build Explorer

The Build Explorer view is used to:

- diagnose parallelism issues,
- determine if your build time is dominated by parsing, code generation, or linking, and
- identify bottlenecks and unusually long build activities.

Build Explorer view data columns

Column Name	Description
BuildTimelineDescription	A textual description the timeline the current activity or property occurs on.
BuildTimelineId	A zero-based identifier for the timeline the current activity or property occurs on.
Component	The component being compiled or linked when the current event was emitted. The value of this column is *<Invocation X Info>* when no component is associated with this event. X is a unique numerical identifier for the invocation being executed at the time the event was emitted. This identifier is the same as the one in the InvocationId column for this event.
Count	The number of activities or properties represented by this data row. This value is always 1, and is only useful in aggregation scenarios when multiple rows are grouped.

Column Name	Description
ExclusiveCPUTime	The amount of CPU time in milliseconds used by this activity. The time spent in child activities isn't included in this amount.
ExclusiveDuration	The millisecond duration of the activity. The duration of child activities isn't included in this amount.
InclusiveCPUTime	The amount of CPU time in milliseconds used by this activity and all child activities.
InclusiveDuration	The millisecond duration of this activity, including all child activities.
InvocationDescription	A textual description of the invocation this event occurred in. The description includes whether it was *cl.exe* or *link.exe*, and a unique numerical invocation identifier. If applicable, it includes the full path to the component compiled or linked during the invocation. For invocations that don't build any component, or for the ones that build multiple components, the path is blank. The invocation identifier is the same as the one in the InvocationId column.
InvocationId	A unique numerical identifier for the invocation this event occurred in.
Name	The name of the activity or property represented by this event.
Time	A timestamp that identifies when the event occurred.
Tool	The tool executing when this event occurred. The value of this column is either CL or Link.
Type	The type of the current event. This value is either Activity or Property.
Value	If the current event is a property, this column contains its value. This column is left blank when the current event is an activity.

Build Explorer view presets

Preset Name	Preferred View Mode	How to Use
Activity Statistics	Graph / Table	Use this preset to view aggregated statistics for all Build Explorer activities. In table mode, tell at a glance if your build is dominated by parsing, code generation, or the linker. The aggregated durations for each activity are sorted in descending order. Drill in by expanding the top node to easily find which invocations take the most time for these top activities. If you like, you can adjust WPA settings to show averages or other types of aggregations. In graph mode, see when each activity is active during your build.

Preset Name	Preferred View Mode	How to Use
Invocations	Graph	Scroll down through a list of invocations in the graph view sorted by start time. You can use it together with the CPU (Sampled) view to find invocations that align with low CPU utilization zones. Detect parallelism issues.
Invocation Properties	Table	Quickly find key information about a given compiler or linker invocation. Determine its version, working directory, or the full command line used to invoke it.
Timelines	Graph	See a bar graph of how your build was parallelized. Identify parallelism issues and bottlenecks at a glance. Configure WPA to assign different meanings to the bars according to your needs. Choose invocation descriptions as the last grouped column to view a color coded bar graph of all your invocations. It helps you quickly identify time consuming culprits. Then, zoom in and choose the activity name as the last grouped column to see the longest parts.

Files

The Files view is used to:

- determine which headers get included the most often, and
- help you decide what to include in a pre-compiled header (PCH).

Files view data columns

Column Name	Description
ActivityName	The activity in progress when this file event was emitted. Currently, this value is always *Parsing*.
BuildTimelineDescription	*
BuildTimelineId	*
Component	*
Count	*
Depth	The zero-based position in the include tree in which this file is found. Counting starts at the root of the include tree. A value of 0 typically corresponds to a .c/.cpp file.

Column Name	Description
ExclusiveDuration	*
IncludedBy	The full path of the file that included the current file.
IncludedPath	The full path of the current file.
InclusiveDuration	*
InvocationId	*
StartTime	A timestamp that represents the time at which the current file event was emitted.
Tool	*

* The value of this column is the same as in the Build Explorer view.

Files view presets

Preset Name	Preferred View Mode	How to Use
Statistics	Table	See which files had the highest aggregated parsing time by looking at the list in descending order. Use this information to help you restructure your headers or decide what to include in your PCH.

Functions

The Functions view is used to identify functions with an excessively long code generation time.

Functions view data columns

Column Name	Description
ActivityName	The activity in progress when this function event was emitted. Currently, this value is always *CodeGeneration*.
BuildTimelineDescription	*
BuildTimelineId	*
Component	*

Column Name	Description
Count	*
Duration	The duration of the code generation activity for this function.
FunctionName	The name of the function undergoing code generation.
InvocationId	*
StartTime	A timestamp that represents when the current function event was emitted.
Tool	*

* The value of this column is the same as in the Build Explorer view.

Functions view presets

Preset Name	Preferred View Mode	How to Use
Statistics	Table	See which functions had the highest aggregated code generation time by looking at the list in descending order. They may hint where your code overuses the `__forceinline` keyword, or that some functions may be too large.
Timelines	Graph	Look at this bar graph to learn the location and duration of functions that take the most time to generate. See if they align with bottlenecks in the Build Explorer view. If they do, take appropriate action to reduce their code generation time and benefit your build times.

See also

Get started with C++ Build Insights
Reference: vcperf commands
Tutorial: Windows Performance Analyzer basics
Windows Performance Analyzer

C++ Build Insights SDK

Article • 10/29/2021

The C++ Build Insights SDK is a collection of APIs that allow you to create personalized tools on top of the C++ Build Insights platform. This page provides a high-level overview to help you get started.

Obtaining the SDK

You can download the C++ Build Insights SDK as a NuGet package by following these steps:

1. From Visual Studio 2017 and above, create a new C++ project.
2. In the **Solution Explorer** pane, right-click on your project.
3. Select **Manage NuGet Packages** from the context menu.
4. At the top right, select the **nuget.org** package source.
5. Search for the latest version of the Microsoft.Cpp.BuildInsights package.
6. Choose **Install**.
7. Accept the license.

Read on for information about the general concepts surrounding the SDK. You can also access the official C++ Build Insights samples GitHub repository to see examples of real C++ applications that use the SDK.

Collecting a trace

Using the C++ Build Insights SDK to analyze events coming out of the MSVC toolchain requires that you first collect a trace. The SDK makes use of Event Tracing for Windows (ETW) as the underlying tracing technology. Collecting a trace can be done in two ways:

Method 1: using vcperf in Visual Studio 2019 and above

1. Open an elevated x64 Native Tools Command Prompt for VS 2019.
2. Run the following command: `vcperf /start MySessionName`
3. Build your project.
4. Run the following command: `vcperf /stopnoanalyze MySessionName outputTraceFile.etl`

> **Important**
>
> Use the `/stopnoanalyze` command when stopping your trace with vcperf. You can't use the C++ Build Insights SDK to analyze traces stopped by the regular `/stop` command.

Method 2: programmatically

Use any of these C++ Build Insights SDK trace collection functions to start and stop traces programmatically. **The program that executes these function calls must have administrative privileges.** Only the start and stop tracing functions require administrative privileges. All other functions in the C++ Build Insights SDK can be executed without them.

SDK functions related to trace collection

Functionality	C++ API	C API
Starting a trace	StartTracingSession	StartTracingSessionA StartTracingSessionW
Stopping a trace	StopTracingSession	StopTracingSessionA StopTracingSessionW
Stopping a trace and immediately analyzing the result	StopAndAnalyzeTracingSession	StopAndAnalyzeTracingSessionA StopAndAnalyzeTracingSession
Stopping a trace and immediately relogging the result	StopAndRelogTracingSession	StopAndRelogTracingSessionA StopAndRelogTracingSessionW

The sections that follow show you how to configure an analysis or a relogging session. It's required for the combined functionality functions, such as StopAndAnalyzeTracingSession.

Consuming a trace

Once you have an ETW trace, use the C++ Build Insights SDK to unpack it. The SDK gives you the events in a format that allows you to develop your tools quickly. We don't recommend you consume the raw ETW trace without using the SDK. The event format used by MSVC is undocumented, optimized to scale to huge builds, and hard to make

sense of. Additionally, the C++ Build Insights SDK API is stable, while the raw ETW trace format is subject to change without notice.

SDK types and functions related to trace consumption

Functionality	C++ API	C API	Notes
Setting up event callbacks	IAnalyzer IRelogger	ANALYSIS_CALLBACKS RELOG_CALLBACKS	The C++ Build Insights SDK provides events through callback functions. In C++, implement the callback functions by creating an analyzer or relogger class that inherits the IAnalyzer or IRelogger interface. In C, implement the callbacks in global functions and provide pointers to them in the ANALYSIS_CALLBACKS or RELOG_CALLBACKS structure.
Building groups	MakeStaticAnalyzerGroup MakeStaticReloggerGroup MakeDynamicAnalyzerGroup MakeDynamicReloggerGroup		The C++ API provides helper functions and types to group multiple analyzer and relogger objects together. Groups are a neat way to divide a complex analysis into simpler steps. vcperf is organized in this way.
Analyzing or relogging	Analyze Relog	AnalyzeA AnalyzeW RelogA RelogW	

Analyzing and relogging

Consuming a trace is done through either an analysis session or a relogging session.

Using a regular analysis is appropriate for most scenarios. This method gives you the flexibility to choose your output format: `printf` text, xml, JSON, database, REST calls, and so on.

Relogging is for special-purpose analyses that need to produce an ETW output file. Using relogging, you can translate the C++ Build Insights events into your own ETW event format. An appropriate use of relogging would be to hook C++ Build Insights data to your existing ETW tools and infrastructure. For example, vcperf makes use of the relogging interfaces. That's because it must produce data the Windows Performance Analyzer, an ETW tool, can understand. Some prior knowledge of how ETW works is required if you plan on using the relogging interfaces.

Creating analyzer groups

It's important to know how to create groups. Here's an example that shows how to create an analyzer group that prints *Hello, world!* for every activity start event it receives.

C++

```
using namespace Microsoft::Cpp::BuildInsights;

class Hello : public IAnalyzer
{
public:
    AnalysisControl OnStartActivity(
        const EventStack& eventStack) override
    {
        std::cout << "Hello, " << std::endl;
        return AnalysisControl::CONTINUE;
    }
};

class World : public IAnalyzer
{
public:
    AnalysisControl OnStartActivity(
        const EventStack& eventStack) override
    {
        std::cout << "world!" << std::endl;
        return AnalysisControl::CONTINUE;
    }
};

int main()
{
    Hello hello;
    World world;

    // Let's make Hello the first analyzer in the group
    // so that it receives events and prints "Hello, "
    // first.
    auto group = MakeStaticAnalyzerGroup(&hello, &world);

    unsigned numberOfAnalysisPasses = 1;
```

```
    // Calling this function initiates the analysis and
    // forwards all events from "inputTrace.etl" to my analyzer
    // group.
    Analyze("inputTrace.etl", numberOfAnalysisPasses, group);

    return 0;
}
```

Using events

SDK types and functions related to events

Functionality	C++ API	C API	Notes
Matching and filtering events	MatchEventStackInMemberFunction MatchEventStack MatchEventInMemberFunction MatchEvent		The C++ API offers functions that make it easy to extract the events you care about from your traces. With the C API, this filtering must be done by hand.
Event data types	Activity BackEndPass BottomUp C1DLL C2DLL CodeGeneration CommandLine Compiler CompilerPass EnvironmentVariable Event	CL_PASS_DATA EVENT_COLLECTION_DATA EVENT_DATA EVENT_ID FILE_DATA FILE_TYPE_CODE FRONT_END_FILE_DATA FUNCTION_DATA FUNCTION_FORCE_INLINEE_DATA INVOCATION_DATA INVOCATION_VERSION_DATA	

Functionality	C++ API	C API	Notes
	EventGroup	MSVC_TOOL_CODE	
	EventStack	NAME_VALUE_PAIR_DATA	
	ExecutableImageOutput	SYMBOL_NAME_DATA	
	ExpOutput	TEMPLATE_INSTANTIATION_DATA	
	FileInput	TEMPLATE_INSTANTIATION_KIND_CODE	
	FileOutput	TRACE_INFO_DATA	
	ForceInlinee	TRANSLATION_UNIT_PASS_CODE	
	FrontEndFile	TRANSLATION_UNIT_TYPE	
	FrontEndFileGroup	TRANSLATION_UNIT_TYPE_DATA	
	FrontEndPass		
	Function		
	HeaderUnit		
	ImpLibOutput		
	Invocation		
	InvocationGroup		
	LibOutput		
	Linker		
	LinkerGroup		
	LinkerPass		
	LTCG		
	Module		
	ObjOutput		
	OptICF		
	OptLBR		
	OptRef		
	Pass1		
	Pass2		
	PrecompiledHeader		
	PreLTCGOptRef		
	SimpleEvent		
	SymbolName		
	TemplateInstantiation		
	TemplateInstantiationGroup		
	Thread		
	TopDown		
	TraceInfo		
	TranslationUnitType		
	WholeProgramAnalysis		

Activities and simple events

Events come in two categories: *activities* and *simple events*. Activities are ongoing processes in time that have a beginning and an end. Simple events are punctual occurrences and don't have a duration. When analyzing MSVC traces with the C++ Build Insights SDK, you'll receive separate events when an activity starts and stops. You'll receive only one event when a simple event occurs.

Parent-child relationships

Activities and simple events are related to each other via parent-child relationships. The parent of an activity or simple event is the encompassing activity in which they occur. For example, when compiling a source file the compiler has to parse the file, then generate the code. The parsing and code generation activities are both children of the compiler activity.

Simple events don't have a duration, so nothing else can happen inside them. As such, they never have any children.

The parent-child relationships of each activity and simple event are indicated in the event table. Knowing these relationships is important when consuming C++ Build Insights events. You'll often have to rely on them to understand the full context of an event.

Properties

All events have the following properties:

Property	Description
Type identifier	A number that uniquely identifies the event type.
Instance identifier	A number that uniquely identifies the event within the trace. If two events of the same type occur in a trace, both get a unique instance identifier.
Start time	The time when an activity started, or the time when a simple event occurred.
Process identifier	A number that identifies the process in which the event occurred.
Thread identifier	A number that identifies the thread in which the event occurred.
Processor index	A zero-based index indicating which logical processor the event was emitted by.
Event name	A string that describes the event type.

All activities other than simple events also have these properties:

Property	Description
Stop time	The time when the activity stopped.

Property	Description
Exclusive duration	The time spent in an activity, excluding the time spent in its child activities.
CPU time	The time that the CPU spent executing code in the thread attached to the activity. It doesn't include time when the thread attached to the activity was sleeping.
Exclusive CPU time	Same as CPU time, but excluding the CPU time spent by child activities.
Wall-clock time responsibility	The activity's contribution to overall wall-clock time. Wall-clock time responsibility takes into account parallelism between activities. For example, let's assume two unrelated activities run in parallel. Both have a duration of 10 seconds, and exactly the same start and stop time. In this case, Build Insights assigns both a wall-clock time responsibility of 5 seconds. In contrast, if these activities run one after the other with no overlap, they're both assigned a wall-clock time responsibility of 10 seconds.
Exclusive wall-clock time responsibility	Same as wall-clock time responsibility, but excludes the wall-clock time responsibility of child activities.

Some events have their own properties beyond the ones mentioned. In this case, these additional properties are listed in the event table.

Consuming events provided by the C++ Build Insights SDK

The event stack

Whenever the C++ Build Insights SDK gives you an event, it comes in the form of a stack. The last entry in the stack is the current event, and entries before it are its parent hierarchy. For example, LTCG start and stop events occur during pass 1 of the linker. In this case, the stack you'd receive contains: [LINKER, PASS1, LTCG]. The parent hierarchy is convenient because you can trace back an event to its root. If the LTCG activity mentioned above is slow, you can immediately learn which linker invocation was involved.

Matching events and event stacks

The C++ Build Insights SDK gives you every event in a trace, but most of the time you only care about a subset of them. In some cases, you may only care about a subset of *event stacks*. The SDK provides facilities to help you quickly extract the events or event

stack you need, and reject the ones you don't. It's done through these matching functions:

Function	Description
MatchEvent	Keep an event if it matches one of the specified types. Forward matched events to a lambda or other callable type. The event's parent hierarchy isn't considered by this function.
MatchEventInMemberFunction	Keep an event if it matches the type specified in a member function's parameter. Forward matched events to the member function. The event's parent hierarchy isn't considered by this function.
MatchEventStack	Keep an event if both the event and its parent hierarchy match the types specified. Forward the event and the matched parent hierarchy events to a lambda or other callable type.
MatchEventStackInMemberFunction	Keep an event if both the event and its parent hierarchy match the types specified in a member function's parameter list. Forward the event and the matched parent hierarchy events to the member function.

The event stack matching functions like `MatchEventStack` allow gaps when describing the parent hierarchy to match. For example, you can say you're interested in the [LINKER, LTCG] stack. It would also match the [LINKER, PASS1, LTCG] stack. The last type specified must be the event type to match, and isn't part of the parent hierarchy.

Capture classes

Using the `Match*` functions requires that you specify the types you want to match. These types are selected from a list of *capture classes*. Capture classes come in several categories, described below.

Category	Description
Exact	These capture classes are used to match a specific event type and none other. An example is the Compiler class, which matches the COMPILER event.
Wildcard	These capture classes can be used to match any event from the list of events they support. For example, the Activity wildcard matches any activity event. Another example is the CompilerPass wildcard, which can match either the FRONT_END_PASS or the BACK_END_PASS event.

Category	Description
Group	The names of group capture classes end in *Group*. They're used to match multiple events of the same type in a row, ignoring gaps. They only make sense when matching recursive events, because you don't know how many exist in the event stack. For example, the FRONT_END_FILE activity happens every time the compiler parses a file. This activity is recursive because the compiler may find an include directive while it's parsing the file. The FrontEndFile class matches only one FRONT_END_FILE event in the stack. Use the FrontEndFileGroup class to match the entire include hierarchy.
Wildcard group	A wildcard group combines the properties of wildcards and groups. The only class of this category is InvocationGroup, which match and capture all LINKER and COMPILER events in a single event stack.

Refer to the event table to learn which capture classes can be used to match each event.

After matching: using captured events

Once a match completes successfully, the `Match*` functions construct the capture class objects and forward them to the specified function. Use these capture class objects to access the events' properties.

Example

C++

```
AnalysisControl MyAnalyzer::OnStartActivity(const EventStack& eventStack)
{
    // The event types to match are specified in the PrintIncludes function
    // signature.
    MatchEventStackInMemberFunction(eventStack, this,
&MyAnalyzer::PrintIncludes);
}

// We want to capture event stacks where:
// 1. The current event is a FrontEndFile activity.
// 2. The current FrontEndFile activity has at least one parent FrontEndFile
activity
//    and possibly many.
void PrintIncludes(FrontEndFileGroup parentIncludes, FrontEndFile
currentFile)
{
    // Once we reach this point, the event stack we are interested in has
been matched.
    // The current FrontEndFile activity has been captured into
'currentFile', and
    // its entire inclusion hierarchy has been captured in 'parentIncludes'.

    cout << "The current file being parsed is: " << currentFile.Path() <<
```

```
endl;
    cout << "This file was reached through the following inclusions:" <<
endl;

    for (auto& f : parentIncludes)
    {
        cout << f.Path() << endl;
    }
}
```

C++ Build Insights SDK: event table

Article • 10/29/2021

Compiler events

COMPILER
COMMAND_LINE
ENVIRONMENT_VARIABLE
FILE_INPUT
OBJ_OUTPUT
FRONT_END_PASS
BACK_END_PASS

Compiler front-end events

C1_DLL
FRONT_END_FILE
TEMPLATE_INSTANTIATION
SYMBOL_NAME
MODULE
HEADER_UNIT
PRECOMPILED_HEADER

Compiler back-end events

C2_DLL
WHOLE_PROGRAM_ANALYSIS
TOP_DOWN
BOTTOM_UP
CODE_GENERATION
THREAD
FUNCTION
FORCE_INLINEE

Linker events

LINKER
COMMAND_LINE

ENVIRONMENT_VARIABLE
FILE_INPUT
EXECUTABLE_IMAGE_OUTPUT
EXP_OUTPUT
IMP_LIB_OUTPUT
LIB_OUTPUT
PASS1
PRE_LTCG_OPT_REF
LTCG
OPT_REF
OPT_ICF
OPT_LBR
PASS2

Event table

Event	Property	Description
BACK_END_PASS	Type	Activity
	Parents	COMPILER
	Children	C2_DLL
	Properties	- Absolute path to input source file - Absolute path to output object file
	Capture classes	Activity CompilerPass BackEndPass
	Description	Occurs at the start and stop of the compiler back-end pass. This pass is responsible for optimizing parsed C/C++ source code and converting it into machine code.
BOTTOM_UP	Type	Activity
	Parents	WHOLE_PROGRAM_ANALYSIS
	Children	None
	Properties	None
	Capture classes	Activity BottomUp

Event	Property	Description
	Description	Occurs at the start and stop of the whole program analysis' bottom-up pass.
C1_DLL	Type	Activity
	Parents	FRONT_END_PASS
	Children	FRONT_END_FILE SYMBOL_NAME TEMPLATE_INSTANTIATION
	Properties	None
	Capture classes	Activity C1DLL
	Description	Occurs at the start and stop of a *c1.dll* or *c1xx.dll* invocation. These DLLs are the C and C++ front end of the compiler. They're invoked solely by the compiler driver (*cl.exe*).
C2_DLL	Type	Activity
	Parents	BACK_END_PASS LTCG
	Children	CODE_GENERATION WHOLE_PROGRAM_ANALYSIS
	Properties	None
	Capture classes	Activity C2DLL
	Description	Occurs at the start and stop of a *c2.dll* invocation. This DLL is the back end of the compiler. It's called by the compiler driver (*cl.exe*). It's also invoked by the linker (*link.exe*) when link-time code generation is used.
CODE_GENERATION	Type	Activity
	Parents	C2_DLL
	Children	FUNCTION THREAD
	Properties	None

Event	Property	Description
	Capture classes	Activity CodeGeneration
	Description	Occurs at the start and stop of the back end's code generation phase.
COMMAND_LINE	Type	Simple Event
	Parents	COMPILER LINKER
	Children	None
	Properties	- The command line that was used to invoke *cl.exe* or *link.exe*
	Capture classes	SimpleEvent CommandLine
	Description	Occurs when the compiler and linker are done evaluating the command line. The evaluated command line includes all *cl.exe* and *link.exe* parameters passed via a response file. It also includes parameters to *cl.exe* and *link.exe* passed via environment variables such as CL, _CL_, LINK, and _LINK_.
COMPILER	Type	Activity
	Parents	None
	Children	BACK_END_PASS COMMAND_LINE ENVIRONMENT_VARIABLE FILE_INPUT OBJ_OUTPUT FRONT_END_PASS
	Properties	- Compiler version - Working directory - Absolute path to the invoked *cl.exe*
	Capture classes	Activity Invocation Compiler
	Description	Occurs at the start and stop of a *cl.exe* invocation.
ENVIRONMENT_VARIABLE	Type	Simple Event

Event	Property	Description
	Parents	COMPILER LINKER
	Children	None
	Properties	- The name of the environment variable - The value of the environment variable.
	Capture classes	SimpleEvent EnvironmentVariable
	Description	Occurs once for every existing environment variable at the time *cl.exe* or *link.exe* is invoked.
EXECUTABLE_IMAGE_OUTPUT	Type	Simple Event
	Parents	LINKER
	Children	None
	Properties	- The absolute path to a DLL or executable output file.
	Capture classes	SimpleEvent FileOutput ExecutableImageOutput
	Description	Occurs when one of the linker inputs is a DLL or an executable image file.
EXP_OUTPUT	Type	Simple Event
	Parents	LINKER
	Children	None
	Properties	- The absolute path to an *.exp* output file.
	Capture classes	SimpleEvent FileOutput ExpOutput
	Description	Occurs when one of the linker outputs is an *.exp* file.
FILE_INPUT	Type	Simple Event
	Parents	COMPILER LINKER

Event	Property	Description
	Children	None
	Properties	- The absolute path to the input file - The type of input file
	Capture classes	SimpleEvent FileInput
	Description	Occurs to announce a *cl.exe* or *link.exe* input.
FORCE_INLINEE	Type	Simple Event
	Parents	FUNCTION
	Children	None
	Properties	- The name of the force-inlined function. - The size of the force-inlined function, represented as an intermediate instruction count.
	Capture classes	Activity ForceInlinee
	Description	Occurs when a function is being force-inlined into another function through the use of the `__forceinline` keyword.
FRONT_END_FILE	Type	Activity
	Parents	C1_DLL FRONT_END_FILE
	Children	FRONT_END_FILE TEMPLATE_INSTANTIATION
	Properties	- Absolute path to the file.
	Capture classes	Activity FrontEndFile
	Description	Occurs when the compiler front end starts and stops processing a file. This event is recursive. Recursion happens when the front end is parsing included files.
FRONT_END_PASS	Type	Activity
	Parents	COMPILER

Event	Property	Description
	Children	C1_DLL MODULE HEADER_UNIT PRECOMPILED_HEADER
	Properties	- Absolute path to input source file - Absolute path to output object file
	Capture classes	Activity CompilerPass FrontEndPass
	Description	Occurs at the start and stop of the compiler front-end pass. This pass is responsible for parsing C/C++ source code and converting it into intermediate language.
FUNCTION	Type	Activity
	Parents	CODE_GENERATION THREAD TOP_DOWN
	Children	FORCE_INLINEE
	Properties	- Name of the function
	Capture classes	Activity Function
	Description	Occurs when starting and ending generating the code for a function.
HEADER_UNIT	Type	Activity
	Parents	FRONT_END_PASS
	Children	None
	Properties	None
	Capture classes	SimpleEvent TranslationUnitType HeaderUnit
	Description	Occurs at the start of the front-end pass and represents that a header unit translation unit is being processed.
IMP_LIB_OUTPUT	Type	Simple Event

Event	Property	Description
	Parents	LINKER
	Children	None
	Properties	- The absolute path to an import library output file.
	Capture classes	SimpleEvent FileOutput ImpLibOutput
	Description	Occurs when one of the linker's outputs is an import library.
LIB_OUTPUT	Type	Simple Event
	Parents	LINKER
	Children	None
	Properties	- The absolute path to a static library output file.
	Capture classes	SimpleEvent FileOutput LibOutput
	Description	Occurs when one of the linker's outputs is static library.
LINKER	Type	Activity
	Parents	None
	Children	COMMAND_LINE ENVIRONMENT_VARIABLE EXECUTABLE_IMAGE_OUTPUT EXP_OUTPUT FILE_INPUT IMP_LIB_OUTPUT LIB_OUTPUT PASS1 PASS2
	Properties	- Linker version - Working directory - Absolute path to the invoked *link.exe*
	Capture classes	Activity Invocation Linker

Event	Property	Description
	Description	Occurs at the start and stop of a *link.exe* invocation.
LTCG	Type	Activity
	Parents	PASS1
	Children	C2_DLL
	Properties	None
	Capture classes	Activity LTCG
	Description	Occurs at the start and stop of link-time code generation.
MODULE	Type	Activity
	Parents	FRONT_END_PASS
	Children	None
	Properties	None
	Capture classes	SimpleEvent TranslationUnitType Module
	Description	Occurs at the start of the front-end pass and represents that a module translation unit is being processed.
OBJ_OUTPUT	Type	Simple Event
	Parents	COMPILER
	Children	None
	Properties	- The absolute path to the *.obj* output file
	Capture classes	SimpleEvent FileOutput ObjOutput
	Description	Occurs once for every *.obj* output produced by *cl.exe*.
OPT_ICF	Type	Activity
	Parents	PASS1

Event	Property	Description
	Children	None
	Properties	None
	Capture classes	Activity OptICF
	Description	Occurs at the start and stop of the identical COMDAT folding (/OPT:ICF) linker optimization.
OPT_LBR	Type	Activity
	Parents	PASS1
	Children	None
	Properties	None
	Capture classes	Activity OptLBR
	Description	Occurs at the start and stop of the long branch (/OPT:LBR) linker optimization.
OPT_REF	Type	Activity
	Parents	PASS1
	Children	None
	Properties	None
	Capture classes	Activity OptRef
	Description	Occurs at the start and stop of the unreferenced functions and data elimination (/OPT:REF) linker optimization.
PASS1	Type	Activity
	Parents	LINKER
	Children	LTCG OPT_ICF OPT_LBR OPT_REF
	Properties	None

Event	Property	Description
	Capture classes	Activity Pass1
	Description	Occurs at the start and stop of the linker's pass 1.
PASS2	Type	Activity
	Parents	LINKER
	Children	None
	Properties	None
	Capture classes	Activity Pass2
	Description	Occurs at the start and stop of the linker's pass 2.
PRECOMPILED_HEADER	Type	Activity
	Parents	FRONT_END_PASS
	Children	None
	Properties	None
	Capture classes	SimpleEvent TranslationUnitType PrecompiledHeader
	Description	Occurs at the start of the front-end pass and represents that a precompiled header (PCH) translation unit is being processed.
PRE_LTCG_OPT_REF	Type	Activity
	Parents	PASS1
	Children	None
	Properties	None
	Capture classes	Activity PreLTCGOptRef
	Description	Occurs at the start and stop of the linker optimization pass that eliminates unreferenced functions and data (/OPT:REF). It's done before link-time code generation.
SYMBOL_NAME	Type	Simple Event

Event	Property	Description
	Parents	C1_DLL
	Children	None
	Properties	- A type key - The type's name
	Capture classes	SimpleEvent SymbolName
	Description	Occurs at the end of the front-end pass: once for every type involved in template instantiations. The key is a numerical identifier for the type, while the name is its text representation. Type keys are unique within the trace being analyzed. However, different keys coming from different compiler front-end passes may point to the same type. Comparing types between different compiler front-end passes requires using their names. SYMBOL_NAME events are emitted at the end of a compiler front-end pass, after all template instantiations have taken place.
TEMPLATE_INSTANTIATION	Type	Activity
	Parents	C1_DLL FRONT_END_FILE TEMPLATE_INSTANTIATION
	Children	TEMPLATE_INSTANTIATION
	Properties	- The key for the specialized type - The key for the primary template's type - The kind of template that was instantiated
	Capture classes	Activity TemplateInstantiation

Event	Property	Description
	Description	Occurs at the beginning and end of a template instantiation. A primary template type (such as `vector`) is instantiated, resulting in a specialized type (such as `std::vector<int>`). A key is given for both types. Use the SYMBOL_NAME event to convert a key into the type's name. Type keys are unique within the trace being analyzed. However, different keys coming from different compiler front-end passes may point to the same type. Comparing types between different compiler front-end passes requires using symbol names. This event is recursive. Recursion happens in some cases when the front end is instantiating nested templates.
THREAD	Type	Activity
	Parents	CODE_GENERATION TOP_DOWN
	Children	FUNCTION
	Properties	None
	Capture classes	Activity Thread
	Description	Occurs at the start and end of a compiler back-end thread execution. A thread being suspended is considered ended. A thread being woken up is considered started.
TOP_DOWN	Type	Activity
	Parents	WHOLE_PROGRAM_ANALYSIS
	Children	FUNCTION THREAD
	Properties	None
	Capture classes	Activity TopDown
	Description	Occurs at the start and stop of the whole program analysis' top-down pass.
TRANSLATION_UNIT_TYPE	Type	Activity

Event	Property	Description
	Parents	FRONT_END_PASS
	Children	MODULE HEADER_UNIT PRECOMPILED_HEADER
	Properties	- The type of translation unit.
	Capture classes	SimpleEvent TranslationUnitType
	Description	Occurs at the start of the front-end pass. The type identifies whether this pass is processing a module, header unit, or precompiled header.
WHOLE_PROGRAM_ANALYSIS	Type	Activity
	Parents	C2_DLL
	Children	BOTTOM_UP TOP_DOWN
	Properties	None
	Capture classes	Activity WholeProgramAnalysis
	Description	Occurs at the start and stop of the whole-program analysis phase of link-time code generation.

Analyze function

Article • 06/16/2022

The `Analyze` function is used to analyze an Event Tracing for Windows (ETW) trace obtained from MSVC while tracing a C++ build. The events in the ETW trace are forwarded sequentially to an analyzer group provided by the caller. This function supports multi-pass analyses that allow forwarding the event stream to the analyzer group multiple times in a row.

Syntax

C++

```
template <typename... TAnalyzerGroupMembers>
RESULT_CODE Analyze(
    const char*                                    inputLogFile,
    unsigned                                       numberOfPasses,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup);

template <typename... TAnalyzerGroupMembers>
RESULT_CODE Analyze(
    const wchar_t*                                 inputLogFile,
    unsigned                                       numberOfPasses,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup);
```

Parameters

TAnalyzerGroupMembers
This parameter is always deduced.

inputLogFile
The input ETW trace that you wish to read events from.

numberOfPasses
The number of analysis passes to run on the input trace. The trace gets passed through the provided analyzer group once per analysis pass.

analyzerGroup
The analyzer group used for the analysis. Call MakeStaticAnalyzerGroup to create an analyzer group. To use a dynamic analyzer group obtained from MakeDynamicAnalyzerGroup, first encapsulate it inside a static analyzer group by passing its address to `MakeStaticAnalyzerGroup`.

Return Value

A result code from the RESULT_CODE enum.

AnalyzeA

Article • 10/29/2021

The `AnalyzeA` function is used to analyze MSVC events read from an input Event Tracing for Windows (ETW) trace.

Syntax

C++

```
enum RESULT_CODE AnalyzeA(
    const char*                inputLogFile,
    const ANALYSIS_DESCRIPTOR* analysisDescriptor);
```

Parameters

inputLogFile
The input ETW trace that you wish to read events from.

analysisDescriptor
Pointer to an ANALYSIS_DESCRIPTOR object. Use this object to configure the analysis.

Return Value

A result code from the RESULT_CODE enum.

AnalyzeW function

Article • 06/16/2022

The `AnalyzeW` function is used to analyze MSVC events read from an input Event Tracing for Windows (ETW) trace.

Syntax

C++

```
enum RESULT_CODE AnalyzeW(
    const wchar_t*             inputLogFile,
    const ANALYSIS_DESCRIPTOR* analysisDescriptor);
```

Parameters

inputLogFile
The input ETW trace that you wish to read events from.

analysisDescriptor
Pointer to an ANALYSIS_DESCRIPTOR object. Use this object to configure the analysis.

Return Value

A result code from the RESULT_CODE enum.

InjectEvent

Article • 10/29/2021

The `InjectEvent` function is called within a relogger implementing the IRelogger interface. It's used to write an Event Tracing for Windows (ETW) event in the output trace file of a relogging session.

Syntax

C++

```
void InjectEvent(
    const void* relogSession,
    LPCGUID providerId,
    PCEVENT_DESCRIPTOR eventDescriptor,
    unsigned long processId,
    unsigned long threadId,
    unsigned short processorIndex,
    long long timestamp,
    unsigned char* data,
    unsigned long byteCount);
```

Parameters

relogSession
A pointer to the relogging session. A relogging session is provided to reloggers that implement the `IRelogger` interface. For more information, see IRelogger.

providerId
An Event Tracing for Windows (ETW) provider GUID under which the ETW event gets relogged.

eventDescriptor
The ETW event descriptor for the ETW event that's relogged.

processId
The process identifier (PID) for the ETW event that's relogged.

threadId
The thread identifier (TID) for the ETW event that's relogged.

processorIndex
The processor index for the ETW event that's relogged.

timestamp
The timestamp for the ETW event that's relogged.

data
A pointer to the data that should be included in the relogged ETW event.

byteCount
The size of the data, in bytes, pointed to by *data*.

Remarks

For more information on ETW concepts, such as *provider GUID* and *event descriptor*, see the ETW documentation. For details on how to start a relogging session with the C++ Build Insights SDK, see Relog.

MakeDynamicAnalyzerGroup

Article • 10/29/2021

The `MakeDynamicAnalyzerGroup` function is used to create a dynamic analyzer group. Members of an analyzer group receive events one by one from left to right, until all events in a trace get analyzed.

Syntax

C++

```
auto MakeDynamicAnalyzerGroup(std::vector<IAnalyzer*> analyzers);

auto MakeDynamicAnalyzerGroup(std::vector<std::shared_ptr<IAnalyzer>>
analyzers);

auto MakeDynamicAnalyzerGroup(std::vector<std::unique_ptr<IAnalyzer>>
analyzers);
```

Parameters

analyzers
A vector of IAnalyzer pointers included in the dynamic analyzer group. These pointers can be raw, `std::unique_ptr`, or `std::shared_ptr`.

Return Value

A dynamic analyzer group. Use the **`auto`** keyword to capture the return value.

Remarks

Unlike static analyzer groups, the members of a dynamic analyzer group don't need to be known at compile time. You can choose analyzer group members at runtime based on program input, or based on other values that are unknown at compile time. Unlike static analyzer groups, IAnalyzer pointers within a dynamic analyzer group have polymorphic behavior, and virtual function calls are dispatched correctly. This flexibility comes at the cost of a possibly slower event processing time. When all analyzer group members are known at compile time, and if you don't need polymorphic behavior, consider using a static analyzer group. To use a static analyzer group, call MakeStaticAnalyzerGroup instead.

A dynamic analyzer group can be encapsulated inside a static analyzer group. It's done by passing its address to MakeStaticAnalyzerGroup. Use this technique for passing dynamic analyzer groups to functions such as Analyze, which only accept static analyzer groups.

MakeDynamicReloggerGroup

Article • 10/29/2021

The `MakeDynamicReloggerGroup` function is used to create a dynamic relogger group. Members of a relogger group receive events one by one from left to right until all events in a trace have been processed.

Syntax

```
C++
```

```
auto MakeDynamicReloggerGroup(std::vector<IRelogger*> reloggers);

auto MakeDynamicReloggerGroup(std::vector<std::shared_ptr<IRelogger>>
reloggers);

auto MakeDynamicReloggerGroup(std::vector<std::unique_ptr<IRelogger>>
reloggers);
```

Parameters

reloggers
A vector of IRelogger pointers included in the dynamic relogger group. These pointers can be raw, `std::unique_ptr`, or `std::shared_ptr`. IAnalyzer pointers are also considered `IRelogger` pointers because of an inheritance relationship.

Return Value

A dynamic relogger group. Use the **`auto`** keyword to capture the return value.

Remarks

Unlike static relogger groups, the members of a dynamic relogger group don't need to be known at compile time. You can choose relogger group members at runtime based on program input, or based on other values that are unknown at compile time. Unlike static relogger groups, IRelogger pointers within a dynamic relogger group have polymorphic behavior, and virtual function calls are dispatched correctly. This flexibility comes at the cost of a possibly slower event processing time. When all relogger group members are known at compile time, and if you don't need polymorphic behavior,

consider using a static relogger group. To use a static relogger group, call MakeStaticReloggerGroup instead.

A dynamic relogger group can be encapsulated inside a static relogger group. You pass its address to MakeStaticReloggerGroup. Use this technique for passing dynamic relogger groups to functions such as Relog, which only accept static relogger groups.

MakeStaticAnalyzerGroup

Article • 04/27/2022

The `MakeStaticAnalyzerGroup` function creates a static analyzer group that you can pass to functions such as Analyze or Relog. Members of an analyzer group receive events one by one from left to right, until all events in a trace are analyzed.

Syntax

C++

```
template <typename... TAnalyzerPtrs>
auto MakeStaticAnalyzerGroup(TAnalyzerPtrs... analyzers);
```

Parameters

TAnalyzerPtrs
This parameter is always deduced.

analyzers
A parameter pack of IAnalyzer pointers included in the static analyzer group. These pointers can be raw, `std::unique_ptr`, or `std::shared_ptr`.

Return Value

A static analyzer group. Use the `auto` keyword to capture the return value.

Remarks

Unlike dynamic analyzer groups, the members of a static analyzer group must be known at compile time. Also, a static analyzer group contains IAnalyzer pointers that don't have polymorphic behavior. When a static analyzer group analyzes an Event Tracing for Windows (ETW) trace, calls to the `IAnalyzer` interface always resolve to the object directly pointed to by the analyzer group member. This loss of flexibility comes with a possibility of faster event processing.

If the members of an analyzer group can't be known at compile time, or if you require polymorphic behavior on your `IAnalyzer` pointers, consider using a dynamic analyzer group. To use a dynamic analyzer group, call MakeDynamicAnalyzerGroup instead.

MakeStaticReloggerGroup

Article • 10/29/2021

The `MakeStaticReloggerGroup` function is used to create a static relogger group that can be passed to functions such as Relog. Members of a relogger group receive events one by one from left to right until all events in a trace have been processed.

Syntax

```
C++
```

```
template <typename... TReloggerPtrs>
auto MakeStaticReloggerGroup(TReloggerPtrs... reloggers);
```

Parameters

TReloggerPtrs
This parameter is always deduced.

reloggers
A parameter pack of IRelogger pointers that's included in the static relogger group. These pointers can be raw, `std::unique_ptr`, or `std::shared_ptr`. IAnalyzer pointers are also considered `IRelogger` pointers because of an inheritance relationship.

Return Value

A static relogger group. Use the **`auto`** keyword to capture the return value.

Remarks

Unlike dynamic relogger groups, the members of a static relogger group must be known at compile time. Additionally, a static relogger group contains IRelogger pointers that don't have polymorphic behavior. When using a static relogger group to analyze an Event Tracing for Windows (ETW) trace, calls to the `IRelogger` interface always resolve to the object directly pointed to by the relogger group member. This loss of flexibility comes with a possibility of faster event processing times. If the members of a relogger group can't be known at compile time, or if you require polymorphic behavior on your

`IRelogger` pointers, consider using a dynamic relogger group. You can use a dynamic relogger group by calling MakeDynamicReloggerGroup instead.

MatchEvent

Article • 10/29/2021

The `MatchEvent` function is used to match an event against a list of event types. If the event matches a type in the list, it's forwarded to a handler for further processing.

Syntax

C++

```
template <
    typename        TEvent,
    typename...     TEvents,
    typename        TCallable,
    typename...     TExtraArgs>
bool MatchEvent(
    const RawEvent& event,
    TCallable&&     callable,
    TExtraArgs&&... extraArgs);
```

Parameters

TEvent
The first event type that you wish to match.

TEvents
The remaining event types that you wish to match.

TCallable
A type that supports `operator()`. For more information on which arguments get passed to this operator, see the *callable* parameter description.

TExtraArgs
The types of the extra arguments that were passed to `MatchEvent`.

event
The event to match against the event types described by *TEvent* and *TEvents*.

callable
`MatchEvent` invokes *callable* after successfully matching the event with any of the event types described by *TEvent* and *TEvents*. The first argument passed to *callable* is an r-

value of the matched event type. The *extraArgs* parameter pack is perfect-forwarded in the remaining parameters of *callable*.

extraArgs
The arguments that get perfect-forwarded to *callable* along with the matched event type.

Return Value

A `bool` value that's `true` if matching was successful, or `false` otherwise.

Remarks

Event types to use for the *TEvent* and *TEvents* parameters are selected from a list of *capture classes*. For a list of events and the capture classes you can use to match them, see event table.

Example

C++

```
void MyClass::OnStartActivity(const EventStack& eventStack)
{
    // Let's assume eventStack contains:
    // [Compiler, BackEndPass, C2DLL, CodeGeneration, Thread, Function]

    auto& functionEvent = eventStack.Back(); // The Function event

    bool b1 = MatchEvent<Function, Thread>(
        functionEvent, [](auto matchedEvent){ /* Do something... */});

    bool b2 = MatchEvent<CodeGeneration, Thread>(
        functionEvent, [](auto matchedEvent){ /* Do something... */});


    // b1: true because the list of types contains Function, which is
    //     the type of the event we are trying to match.
    // b2: false because the list of types doesn't contain Function.
}
```

MatchEventInMemberFunction

Article • 10/29/2021

The `MatchEventInMemberFunction` function is used to match an event against the type described by the first parameter of a member function. The matched event is forwarded to the member function for further processing.

Syntax

C++

```
template <
    typename     TInterface,
    typename     TReturn,
    typename     TEvent,
    typename...  TExtraParams,
    typename...  TExtraArgs>
bool MatchEventInMemberFunction(
    const RawEvent&          event,
    TInterface*              objectPtr,
    TReturn (TInterface::*   memberFunc)(TEvent, TExtraParams...),
    TExtraArgs&&...          extraArgs);
```

Parameters

TInterface
The type that contains the member function.

TReturn
The return type of the member function.

TEvent
The type of the event to match.

TExtraParams
The types of the extra parameters accepted by the member function along with the event type to match.

TExtraArgs
The types of the extra arguments that were passed to `MatchEventInMemberFunction`.

event
The event to match against the event type described by *TEvent*.

objectPtr
A pointer to an object on which *memberFunc* gets called.

memberFunc
The member function that describes the event type to match.

extraArgs
The arguments that get perfect-forwarded to *memberFunc* along with the event type parameter.

Return Value

A `bool` value that is `true` if matching was successful, or `false` otherwise.

Remarks

The event type to use for the *TEvent* parameter can be selected from a list of *capture classes*. For a list of events and the capture classes you can use to match them, see event table.

Example

C++

```
void MyClass::Foo1(Function f) {}

void MyClass::Foo2(Compiler cl) {}

void MyClass::OnStartActivity(const EventStack& eventStack)
{
    // Let's assume eventStack contains:
    // [Compiler, BackEndPass, C2DLL, CodeGeneration, Thread, Function]

    auto& functionEvent = eventStack.Back(); // The Function event

    bool b1 = MatchEventInMemberFunction(
        functionEvent, this, &MyClass::Foo1);

    bool b2 = MatchEventInMemberFunction(
        functionEvent, this, &MyClass::Foo2);

    // b1: true because the first parameter of Foo1 is Function.
    // b2: false because the first parameter of Foo2 isn't Function.
}
```

MatchEventStack

Article • 10/29/2021

The `MatchEventStack` function is used to match an event stack against a specific event hierarchy. Matched hierarchies are forwarded to a handler for further processing. To learn more about events, event stacks, and hierarchies, see event table.

Syntax

C++

```
template <
    typename          TEvent,
    typename...       TEvents,
    typename          TCallable,
    typename...       TExtraArgs>
bool MatchEventStack(
    const EventStack& eventStack,
    TCallable&&       callable,
    TExtraArgs&&...   extraArgs);
```

Parameters

TEvent
The type of the eldest parent to match in the event stack.

TEvents
The remaining types you wish to match in the event stack.

TCallable
A type that supports `operator()`. For more information on which arguments get passed to this operator, see the *callable* parameter description.

TExtraArgs
The types of the extra arguments passed to `MatchEventStack`.

eventStack
The event stack to match against the event type hierarchy described by *TEvent* and *TEvents*.

callable
Upon successfully matching the event stack with the event type hierarchy described by

TEvent and *TEvents*, `MatchEventStack` invokes *callable*. It passes to *callable* one r-value argument for each type in event hierarchy. The *extraArgs* parameter pack is perfect-forwarded in the remaining parameters of *callable*.

extraArgs
The arguments that get perfect-forwarded to *callable* along with the matched event type.

Return Value

A **`bool`** value that is **`true`** if matching was successful, or **`false`** otherwise.

Remarks

The last event in *eventStack* is always matched against the last entry in the concatenated [*TEvent*, *TEvents*...] type list. All other *TEvent* and *TEvents* entries can match any position in *eventStack* except the last, provided they're in the same order.

Event types to use for the *TEvent* and *TEvents* parameters are selected from a list of *capture classes*. For a list of events and the capture classes you can use to match them, see event table.

Example

C++

```
void MyClass::OnStartActivity(const EventStack& eventStack)
{
    // Let's assume eventStack contains:
    // [Compiler, BackEndPass, C2DLL, CodeGeneration, Thread, Function]

    bool b1 = MatchEventStack<Compiler, BackEndPass, C2DLL,
                CodeGeneration, Thread, Function>(
        eventStack, [](Compiler cl, BackEndPass bep, C2DLL c2,
            CodeGeneration cg, Thread t, Function f){ /* Do something ... */
});

    bool b2 = MatchEventStack<Compiler, Function>(
        eventStack, [](Compiler cl, Function f){ /* Do something... */ });

    bool b3 = MatchEventStack<Thread, Compiler, Function>(
        eventStack, [](Thread t, Compiler cl Function f){ /* Do something...
*/ });

    bool b4 = MatchEventStack<Compiler>(
        eventStack, [](Compiler cl){ /* Do something... */ });
```

```
    // b1: true because the list of types matches the eventStack exactly.
    // b2: true because Function is the last entry in both the type list
    //     and 'eventStack', and also because Compiler comes before
    //     Function in 'eventStack' and in the type list.
    // b3: false because, even though both Thread and Compiler come
    //     before Function in 'eventStack', they aren't listed in the
    //     right order in the type list.
    // b4: false because the last entry in the type list is Compiler,
    //     which doesn't match the last entry in 'eventStack' (Function).
}
```

MatchEventStackInMemberFunction

Article • 10/29/2021

The `MatchEventStackInMemberFunction` function is used to match an event stack against a specific event hierarchy, described by the parameter list of a member function. Matched hierarchies are forwarded to the member function for further processing. To learn more about events, event stacks, and hierarchies, see event table.

Syntax

C++

```
template <
    typename     TInterface,
    typename     TReturn,
    typename     T1,
    typename...  TExtraParams,
    typename...  TExtraArgs>
bool MatchEventStackInMemberFunction(
    const EventStack&         eventStack,
    TInterface*               objectPtr,
    TReturn (TInterface::*    memberFunc)(T1, TExtraParams...),
    TExtraArgs&&...           extraArgs);

template <
    typename     TInterface,
    typename     TReturn,
    typename     T1,
    typename     T2,
    typename...  TExtraParams,
    typename...  TExtraArgs>
bool MatchEventStackInMemberFunction(
    const EventStack&         eventStack,
    TInterface*               objectPtr,
    TReturn (TInterface::*    memberFunc)(T1, T2, TExtraParams...),
    TExtraArgs&&...           extraArgs);

// Etc...

template <
    typename     TInterface,
    typename     TReturn,
    typename     T1,
    typename     T2,
    typename     T3,
    typename     T4,
    typename     T5,
    typename     T6,
    typename     T7,
```

```
    typename     T8,
    typename     T9,
    typename     T10,
    typename...  TExtraParams,
    typename...  TExtraArgs>
bool MatchEventStackInMemberFunction(
    const EventStack&         eventStack,
    TInterface*               objectPtr,
    TReturn (TInterface::*    memberFunc)(T1, T2, T3, T4, T5, T6, T7, T8,
T9, T10, TExtraParams...),
    TExtraArgs&&...           extraArgs);
```

Parameters

TInterface
The type that contains the member function.

TReturn
The return type of the member function.

T1, ..., *T10*
The types describing the event hierarchy to match.

TExtraParams
The types of the extra parameters accepted by the member function, and the event hierarchy types.

TExtraArgs
The types of the extra arguments that were passed to `MatchEventStackInMemberFunction`.

eventStack
The event stack to match against the event type hierarchy described by *T1* through *T10*.

objectPtr
A pointer to an object on which *memberFunc* is called.

memberFunc
The member function that describes the event type hierarchy to match.

extraArgs
The arguments that get perfect-forwarded to *memberFunc* along with the event type hierarchy parameters.

Return Value

A `bool` value that is `true` if matching was successful, or `false` otherwise.

Remarks

The last event in *eventStack* is always matched against the last entry in the event type hierarchy to match. All other types in the event type hierarchy can match any position in *eventStack* except the last, provided they're in the same order.

Event types to use for the *T1* through *T10* parameters are selected from a list of *capture classes*. For a list of events and the capture classes you can use to match them, see event table.

Example

C++

```
void MyClass::Foo1(Compiler cl, BackEndPass bep, C2DLL c2,
    CodeGeneration cg, Thread t, Function f) { }

void MyClass::Foo2(Compiler cl, Function f) { }

void MyClass::Foo3(Thread t, Compiler cl, Function f) { }

void MyClass::Foo4(Compiler cl) { }

void MyClass::OnStartActivity(const EventStack& eventStack)
{
    // Let's assume eventStack contains:
    // [Compiler, BackEndPass, C2DLL, CodeGeneration, Thread, Function]

    bool b1 = MatchEventStackInMemberFunction(
        eventStack, this, &MyClass::Foo1);

    bool b2 = MatchEventStackInMemberFunction(
        eventStack, this, &MyClass::Foo2);

    bool b3 = MatchEventStackInMemberFunction(
        eventStack, this, &MyClass::Foo3);

    bool b4 = MatchEventStackInMemberFunction(
        eventStack, this, &MyClass::Foo4);

    // b1: true because the parameter types of Foo1 match the eventStack
    //     exactly.
    // b2: true because Function is the last entry in both the member
    //     function parameter list and 'eventStack', and also because
    //     Compiler comes before Function in 'eventStack' and in the
    //     parameter type list.
    // b3: false because, even though both Thread and Compiler come
```

```
    //     before Function in 'eventStack', the member function parameter
    //     list doesn't list them in the right order.
    // b4: false because the last entry in the member function parameter
    //     list is Compiler, which doesn't match the last entry in
'eventStack'
    //     (Function).
}
```

Relog

Article • 10/14/2022

The `Relog` function is used to read MSVC events from an Event Tracing for Windows (ETW) trace and write them into a new, modified ETW trace.

Syntax

C++

```
template <
    typename... TAnalyzerGroupMembers,
    typename... TReloggerGroupMembers>
RESULT_CODE Relog(
    const char*                                    inputLogFile,
    const char*                                    outputLogFile,
    unsigned                                       numberOfAnalysisPasses,
    unsigned long long
systemEventsRetentionFlags,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup,
    StaticReloggerGroup<TReloggerGroupMembers...> reloggerGroup);

template <
    typename... TAnalyzerGroupMembers,
    typename... TReloggerGroupMembers>
RESULT_CODE Relog(
    const wchar_t*                                 inputLogFile,
    const wchar_t*                                 outputLogFile,
    unsigned                                       numberOfAnalysisPasses,
    unsigned long long
systemEventsRetentionFlags,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup,
    StaticReloggerGroup<TReloggerGroupMembers...> reloggerGroup);
```

Parameters

TAnalyzerGroupMembers
This parameter is always deduced.

TReloggerGroupMembers
This parameter is always deduced.

inputLogFile
The input ETW trace that you wish to read events from.

outputLogFile
The file in which to write the new events.

numberOfAnalysisPasses
The number of analysis passes to run on the input trace. The trace gets passed through the provided analyzer group once per analysis pass.

systemEventsRetentionFlags
A bitmask that specifies which system ETW events to keep in the relogged trace. For more information, see RELOG_RETENTION_SYSTEM_EVENT_FLAGS.

analyzerGroup
The analyzer group used for the analysis phase of the relogging session. Call MakeStaticAnalyzerGroup to create an analyzer group. To use a dynamic analyzer group obtained from MakeDynamicAnalyzerGroup, first encapsulate it inside a static analyzer group by passing its address to `MakeStaticAnalyzerGroup`.

reloggerGroup
The relogger group that relogs events into the trace file specified in *outputLogFile*. Call MakeStaticReloggerGroup to create a relogger group. To use a dynamic relogger group obtained from MakeDynamicReloggerGroup, first encapsulate it inside a static relogger group by passing its address to `MakeStaticReloggerGroup`.

Return Value

A result code from the RESULT_CODE enum.

Remark

The input trace is passed through the analyzer group *numberOfAnalysisPasses* times. There's no similar option for relogging passes. The trace is passed trough the relogger group only once, after all analysis passes are complete.

The relogging of system events like CPU samples from within a relogger class isn't supported. Use the *systemEventsRetentionFlags* parameter to decide which system events to keep in the output trace.

The `relog` function depends on the COM API. You must call `CoInitialize` before you call `relog`. Call `CoUninitialize` once `relog` has finished. If you call `relog` without a call to `CoInitialize` first, you'll get error code 9 (`RESULT_CODE_FAILURE_START_RELOGGER`).

RelogA

Article • 10/29/2021

The `RelogA` function is used to read MSVC events from an Event Tracing for Windows (ETW) trace and write them into a new, modified ETW trace.

Syntax

C++

```
enum RESULT_CODE RelogA(
    const char*             inputLogFile,
    const char*             outputLogFile,
    const RELOG_DESCRIPTOR* relogDescriptor);
```

Parameters

inputLogFile
The input ETW trace that you wish to read events from.

outputLogFile
The file in which to write the new events.

relogDescriptor
Pointer to a RELOG_DESCRIPTOR object. Use this object to configure the relogging session.

Return Value

A result code from the RESULT_CODE enum.

RelogW

Article • 10/29/2021

The `RelogW` function is used to read MSVC events from an input Event Tracing for Windows (ETW) trace and write them into a new, modified ETW trace.

Syntax

C++

```
enum RESULT_CODE RelogW(
    const wchar_t*          inputLogFile,
    const wchar_t*          outputLogFile,
    const RELOG_DESCRIPTOR* relogDescriptor);
```

Parameters

inputLogFile
The input ETW trace that you wish to read events from.

outputLogFile
The file in which to write the new events.

relogDescriptor
Pointer to a RELOG_DESCRIPTOR object. Use this object to configure the relogging session.

Return Value

A result code from the RESULT_CODE enum.

StartTracingSession

Article • 10/29/2021

The `StartTracingSession` function starts a tracing session. Executables calling this function must have administrator privileges.

Syntax

C++

```
RESULT_CODE StartTracingSession(
    const char*                    sessionName,
    const TRACING_SESSION_OPTIONS& options);

RESULT_CODE StartTracingSession(
    const wchar_t*                 sessionName,
    const TRACING_SESSION_OPTIONS& options);
```

Parameters

sessionName
The name of the tracing session to start. Use the same name when calling StopTracingSession or any other stop trace function.

options
Pointer to a TRACING_SESSION_OPTIONS object. Use this object to select which events should be collected by the tracing session.

Return Value

A result code from the RESULT_CODE enum.

StartTracingSessionA

Article • 10/29/2021

The `StartTracingSessionA` function starts a tracing session. Executables calling this function must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StartTracingSessionA(
    const char*                    sessionName,
    const TRACING_SESSION_OPTIONS* options);
```

Parameters

sessionName
The name of the tracing session to start. Use the same name when calling StopTracingSessionA or any other stop trace function.

options
Pointer to a TRACING_SESSION_OPTIONS object. Use this object to select which events should be collected by the tracing session.

Return Value

A result code from the RESULT_CODE enum.

StartTracingSessionW

Article • 10/29/2021

The `StartTracingSessionW` function starts a tracing session. Executables calling this function must have administrator privileges.

Syntax

```
C++
```

```
enum RESULT_CODE StartTracingSessionW(
    const wchar_t*                 sessionName,
    const TRACING_SESSION_OPTIONS* options);
```

Parameters

sessionName
The name of the tracing session to start. Use the same name when calling StopTracingSessionW, or any other stop trace function.

options
Pointer to a TRACING_SESSION_OPTIONS object. Use this object to select which events should be collected by the tracing session.

Return Value

A result code from the RESULT_CODE enum.

StopAndAnalyzeTracingSession

Article • 10/29/2021

The `StopAndAnalyzeTracingSession` function stops an ongoing tracing session and saves the resulting trace in a temporary file. An analysis session is then immediately started using the temporary file as an input. Executables calling this function must have administrator privileges.

Syntax

C++

```
template <typename... TAnalyzerGroupMembers>
RESULT_CODE StopAndAnalyzeTracingSession(
    const char*                                  sessionName,
    unsigned                                     numberOfAnalysisPasses,
    TRACING_SESSION_STATISTICS*                  statistics,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup);

template <typename... TAnalyzerGroupMembers>
RESULT_CODE StopAndAnalyzeTracingSession(
    const wchar_t*                               sessionName,
    unsigned                                     numberOfAnalysisPasses,
    TRACING_SESSION_STATISTICS*                  statistics,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

numberOfAnalysisPasses
The number of analysis passes to run on the trace. The trace gets passed through the provided analyzer group once per analysis pass.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopAndAnalyzeTracingSession` writes trace collection statistics in this object before returning.

analyzerGroup
The analyzer group used for the analysis. Call MakeStaticAnalyzerGroup to create an analyzer group. If you wish to use a dynamic analyzer group obtained from

MakeDynamicAnalyzerGroup, first encapsulate it inside a static analyzer group by passing its address to `MakeStaticAnalyzerGroup`.

Return Value

A result code from the RESULT_CODE enum.

StopAndAnalyzeTracingSessionA

Article • 10/29/2021

The `StopAndAnalyzeTracingSessionA` function stops an ongoing tracing session and saves the resulting trace in a temporary file. An analysis session is then immediately started using the temporary file as an input. Executables calling this function must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StopAndAnalyzeTracingSessionA(
    const char*                 sessionName,
    TRACING_SESSION_STATISTICS* statistics,
    const ANALYSIS_DESCRIPTOR*  analysisDescriptor);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopAndAnalyzeTracingSessionA` writes trace collection statistics in this object before returning.

analysisDescriptor
Pointer to an ANALYSIS_DESCRIPTOR object. Use this object to configure the analysis session that's started by `StopAndAnalyzeTracingSessionA`.

Return Value

A result code from the RESULT_CODE enum.

StopAndAnalyzeTracingSessionW

Article • 10/29/2021

The `StopAndAnalyzeTracingSessionW` function stops an ongoing tracing session and saves the resulting trace in a temporary file. An analysis session is then immediately started using the temporary file as an input. Executables calling this function must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StopAndAnalyzeTracingSessionW(
    const wchar_t*              sessionName,
    TRACING_SESSION_STATISTICS* statistics,
    const ANALYSIS_DESCRIPTOR*  analysisDescriptor);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopAndAnalyzeTracingSessionW` writes trace collection statistics in this object before returning.

analysisDescriptor
Pointer to an ANALYSIS_DESCRIPTOR object. Use this object to configure the analysis session that's started by `StopAndAnalyzeTracingSessionW`.

Return Value

A result code from the RESULT_CODE enum.

StopAndRelogTracingSession

Article • 10/29/2021

The `StopAndRelogTracingSession` function stops an ongoing tracing session and saves the resulting trace in a temporary file. A relogging session is then immediately started using the temporary file as an input. The final relogged trace produced by the relogging session is saved in a file specified by the caller. Executables calling this function must have administrator privileges.

Syntax

C++

```
template <
    typename... TAnalyzerGroupMembers,
    typename... TReloggerGroupMembers>
RESULT_CODE StopAndRelogTracingSession(
    const char*                                    sessionName,
    const char*                                    outputLogFile,
    TRACING_SESSION_STATISTICS*                    statistics,
    unsigned                                       numberOfAnalysisPasses,
    unsigned long long
systemEventsRetentionFlags,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup,
    StaticReloggerGroup<TReloggerGroupMembers...> reloggerGroup);

template <
    typename... TAnalyzerGroupMembers,
    typename... TReloggerGroupMembers>
RESULT_CODE StopAndRelogTracingSession(
    const wchar_t*                                 sessionName,
    const wchar_t*                                 outputLogFile,
    TRACING_SESSION_STATISTICS*                    statistics,
    unsigned                                       numberOfAnalysisPasses,
    unsigned long long
systemEventsRetentionFlags,
    StaticAnalyzerGroup<TAnalyzerGroupMembers...> analyzerGroup,
    StaticReloggerGroup<TReloggerGroupMembers...> reloggerGroup);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

outputLogFile
The file in which to write the relogged trace produced by the relogging session.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopAndRelogTracingSession` writes trace collection statistics in this object before returning.

numberOfAnalysisPasses
The number of analysis passes to run on the trace. The trace gets passed through the provided analyzer group once per analysis pass.

systemEventsRetentionFlags
A RELOG_RETENTION_SYSTEM_EVENT_FLAGS bitmask that specifies which system ETW events to keep in the relogged trace.

analyzerGroup
The analyzer group used for the analysis phase of the relogging session. Call MakeStaticAnalyzerGroup to create an analyzer group. If you wish to use a dynamic analyzer group obtained from MakeDynamicAnalyzerGroup, first encapsulate it inside a static analyzer group by passing its address to `MakeStaticAnalyzerGroup`.

reloggerGroup
The relogger group that relogs events into the trace file specified in *outputLogFile*. Call MakeStaticReloggerGroup to create a relogger group. If you wish to use a dynamic relogger group obtained from MakeDynamicReloggerGroup, first encapsulate it inside a static relogger group by passing its address to `MakeStaticReloggerGroup`.

Return Value

A result code from the RESULT_CODE enum.

Remarks

The input trace is passed through the analyzer group *numberOfAnalysisPasses* times. There's no similar option for relogging passes. The trace is passed trough the relogger group only once, after all analysis passes are complete.

The relogging of system events like CPU samples from within a relogger class isn't supported. Use the *systemEventsRetentionFlags* parameter to decide which system events to keep in the output trace.

StopAndRelogTracingSessionA

Article • 10/29/2021

The `StopAndRelogTracingSessionA` function stops an ongoing tracing session and saves the resulting trace in a temporary file. A relogging session is then immediately started using the temporary file as an input. The final relogged trace produced by the relogging session is saved in a file specified by the caller. Executables calling this function must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StopAndRelogTracingSessionA(
    const char*                 sessionName,
    const char*                 outputLogFile,
    TRACING_SESSION_STATISTICS* statistics,
    const RELOG_DESCRIPTOR*     relogDescriptor);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

outputLogFile
The file in which to write the relogged trace produced by the relogging session.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopAndRelogTracingSessionA` writes trace collection statistics in this object before returning.

analysisDescriptor
Pointer to a RELOG_DESCRIPTOR object. Use this object to configure the relogging session that's started by `StopAndRelogTracingSessionA`.

Return Value

A result code from the RESULT_CODE enum.

StopAndRelogTracingSessionW

Article • 10/29/2021

The `StopAndRelogTracingSessionW` function stops an ongoing tracing session and saves the resulting trace in a temporary file. A relogging session is then immediately started using the temporary file as an input. The final relogged trace produced by the relogging session is saved in a file specified by the caller. Executables calling this function must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StopAndRelogTracingSessionW(
    const wchar_t*              sessionName,
    const wchar_t*              outputLogFile,
    TRACING_SESSION_STATISTICS* statistics,
    const RELOG_DESCRIPTOR*     relogDescriptor);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

outputLogFile
The file in which to write the relogged trace produced by the relogging session.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopAndRelogTracingSessionW` writes trace collection statistics in this object before returning.

analysisDescriptor
Pointer to a RELOG_DESCRIPTOR object. Use this object to configure the relogging session that's started by `StopAndRelogTracingSessionW`.

Return Value

A result code from the RESULT_CODE enum.

StopTracingSession

Article • 04/27/2022

The `StopTracingSession` function stops an ongoing tracing session and produces a raw trace file. You can pass raw trace files to the Analyze, AnalzeA, and AnalyzeW functions to start an analysis session. You can pass raw trace files to the Relog, RelogA, and RelogW functions to start a relogging session.

The caller must have administrator permissions to use `StopTracingSession`.

Syntax

```
C++
```

```
inline RESULT_CODE StopTracingSession(
    const char*                 sessionName,
    const char*                 outputLogFile,
    TRACING_SESSION_STATISTICS* statistics);

inline RESULT_CODE StopTracingSession(
    const wchar_t*              sessionName
    const wchar_t*              outputLogFile,
    TRACING_SESSION_STATISTICS* statistics);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as used for StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

outputLogFile
Full path of the final output log file to save the raw trace.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopTracingSession` writes trace collection statistics in this object before returning.

Return Value

A result code from the RESULT_CODE enum.

StopTracingSessionA

Article • 10/29/2021

The `StopTracingSessionA` function stops an ongoing tracing session and produces a raw trace file. Raw trace files can be passed to the Analyze, AnalzeA, and AnalyzeW functions to start an analysis session. Raw trace files can also be passed to the Relog, RelogA, and RelogW functions to start relogging session. Executables calling `StopTracingSessionA` must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StopTracingSessionA(
    const char*                 sessionName,
    const char*                 outputLogFile,
    TRACING_SESSION_STATISTICS* statistics);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

outputLogFile
Path to the final output log file where the raw trace should be saved.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopTracingSessionA` writes trace collection statistics in this object before returning.

Return Value

A result code from the RESULT_CODE enum.

StopTracingSessionW

Article • 10/29/2021

The `StopTracingSessionW` function stops an ongoing tracing session and produces a raw trace file. Raw trace files can be passed to the Analyze, AnalzeA, and AnalyzeW functions to start an analysis session. Raw trace files can also be passed to the Relog, RelogA, and RelogW functions to start relogging session. Executables calling `StopTracingSessionW` must have administrator privileges.

Syntax

C++

```
enum RESULT_CODE StopTracingSessionW(
    const wchar_t*              sessionName,
    const char*                 outputLogFile,
    TRACING_SESSION_STATISTICS* statistics);
```

Parameters

sessionName
The name of the tracing session to stop. Use the same session name as the one passed to StartTracingSession, StartTracingSessionA, or StartTracingSessionW.

outputLogFile
Path to the final output log file where the raw trace should be saved.

statistics
Pointer to a TRACING_SESSION_STATISTICS object. `StopTracingSessionW` writes trace collection statistics in this object before returning.

Return Value

A result code from the RESULT_CODE enum.

Activity class

Article • 06/16/2022

The `Activity` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match any activity event. Refer to event table to see all events that can be matched by the `Activity` class.

Syntax

C++

```
class Activity : public Event
{
public:
    Activity(const RawEvent& event);

    const long long& StartTimestamp() const;
    const long long& StopTimestamp() const;
    const long long& ExclusiveDurationTicks() const;
    const long long& CPUTicks() const;
    const long long& ExclusiveCPUTicks() const;
    const long long& WallClockTimeResponsibilityTicks() const;
    const long long& ExclusiveWallClockTimeResponsibilityTicks() const;

    std::chrono::nanoseconds Duration() const;
    std::chrono::nanoseconds ExclusiveDuration() const;
    std::chrono::nanoseconds CPUTime() const ;
    std::chrono::nanoseconds ExclusiveCPUTime() const;
    std::chrono::nanoseconds WallClockTimeResponsibility() const;
    std::chrono::nanoseconds ExclusiveWallClockTimeResponsibility() const;
};
```

Remarks

Several member functions in the `Activity` class return a tick count. C++ Build Insights uses Windows' performance counter as a source of ticks. A tick count must be used with a tick frequency to convert it into a time unit such as seconds. The `TickFrequency` member function, available in the Event base class, may be called to obtain the tick frequency. The EVENT_DATA page shows an example of converting ticks into a time unit.

If you don't want to convert ticks to time units yourself, the `Activity` class provides member functions that return time values in nanoseconds. Use the standard C++

`chrono` library to convert them into other time units.

Members

Along with its members inherited from the Event base class, the `Activity` class contains the following members:

Constructor

Activity

Functions

CPUTicks
CPUTime
Duration
ExclusiveCPUTicks
ExclusiveCPUTime
ExclusiveDuration
ExclusiveDurationTicks
ExclusiveWallClockTimeResponsibility
ExclusiveWallClockTimeResponsibilityTicks
StartTimestamp
StopTimestamp
WallClockTimeResponsibility
WallClockTimeResponsibilityTicks

Activity

C++

```
Activity(const RawEvent& event);
```

Parameters

event
Any activity event.

CPUTicks

```
C++
```

```
const long long& CPUTicks() const;
```

Return Value

The number of CPU ticks that occurred during this activity. A CPU tick is different from a regular tick. CPU ticks are only counted when the CPU is executing code in an activity. CPU ticks aren't counted when the thread associated with the activity is sleeping.

CPUTime

```
C++
```

```
std::chrono::nanoseconds CPUTime()() const;
```

Return Value

The amount of time the CPU was executing code inside this activity. This value may be higher than the duration of the activity if child activities are executed on separate threads. The value is returned in nanoseconds.

Duration

```
C++
```

```
std::chrono::nanoseconds Duration() const;
```

Return Value

The duration of the activity in nanoseconds.

ExclusiveCPUTicks

```
C++
```

```
const long long& ExclusiveCPUTicks() const;
```

Return Value

Same as CPUTicks, but not including the CPU ticks that occurred in child activities.

ExclusiveCPUTime

```
C++

std::chrono::nanoseconds ExclusiveCPUTime() const;
```

Return Value

Same as CPUTime, except that the CPU time of child activities isn't included.

ExclusiveDuration

```
C++

std::chrono::nanoseconds ExclusiveDuration() const;
```

Return Value

The duration of the activity in nanoseconds, not including the amount of time that was spent in child activities.

ExclusiveDurationTicks

```
C++

const long long& ExclusiveDurationTicks() const;
```

Return Value

The number of ticks that occurred in this activity, excluding the number of ticks that occurred in child activities.

ExclusiveWallClockTimeResponsibility

```
C++

std::chrono::nanoseconds ExclusiveWallClockTimeResponsibility() const;
```

Return Value

Same as WallClockTimeResponsibility, but not including the wall-clock time responsibility of child activities.

ExclusiveWallClockTimeResponsibilityTicks

```
C++

const long long& ExclusiveWallClockTimeResponsibilityTicks() const;
```

Return Value

Same as WallClockTimeResponsibilityTicks, but not including the wall-clock time responsibility ticks of child activities.

StartTimestamp

```
C++

const long long& StartTimestamp() const;
```

Return Value

A tick value captured at the time the activity started.

StopTimestamp

```
C++

const long long& StopTimestamp() const;
```

Return Value

A tick value captured at the time the activity stopped.

WallClockTimeResponsibility

```
C++
```

```
std::chrono::nanoseconds WallClockTimeResponsibility() const;
```

Return Value

The wall-clock time responsibility of this activity, in nanoseconds. For more information on what wall-clock time responsibility means, see WallClockTimeResponsibilityTicks.

WallClockTimeResponsibilityTicks

```
C++
```

```
const long long& WallClockTimeResponsibilityTicks() const;
```

Return Value

A tick count that represents this activity's contribution to overall wall-clock time. A wall-clock time responsibility tick is different from a regular tick. Wall-clock time responsibility ticks take into account parallelism between activities. Two parallel activities may have a duration of 50 ticks, and the same start and stop time. In this case, both get assigned a wall-clock time responsibility of 25 ticks.

BackEndPass class

Article • 06/16/2022

The `BackEndPass` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a BACK_END_PASS event.

Syntax

```
C++
```

```
class BackEndPass : public CompilerPass
{
public:
    BackEndPass(const RawEvent& event);
};
```

Members

Along with the inherited members from its CompilerPass base class, the `BackEndPass` class contains the following members:

Constructors

BackEndPass

BackEndPass

```
C++
```

```
BackEndPass(const RawEvent& event);
```

Parameters

event
A BACK_END_PASS event.

BottomUp class

Article • 10/29/2021

The `BottomUp` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a BOTTOM_UP event.

Syntax

C++

```
class BottomUp : public Activity
{
public:
    BottomUp(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `BottomUp` class contains the following members:

Constructors

BottomUp

BottomUp

C++

```
BottomUp(const RawEvent& event);
```

Parameters

event
A BOTTOM_UP event.

C1DLL class

Article • 06/16/2022

The `C1DLL` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a C1_DLL event.

Syntax

```
C++
```

```
class C1DLL : public Activity
{
public:
    C1DLL(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `C1DLL` class contains the following members:

Constructors

C1DLL

C1DLL

```
C++
```

```
C1DLL(const RawEvent& event);
```

Parameters

event
A C1_DLL event.

C2DLL class

Article • 10/29/2021

The `C2DLL` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a C2_DLL event.

Syntax

```
C++
```

```
class C2DLL : public Activity
{
public:
    C2DLL(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `C2DLL` class contains the following members:

Constructors

C2DLL

C2DLL

```
C++
```

```
C2DLL(const RawEvent& event);
```

Parameters

event
A C2_DLL event.

CodeGeneration class

Article • 10/29/2021

The `CodeGeneration` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a CODE_GENERATION event.

Syntax

```
C++
```

```
class CodeGeneration : public Activity
{
public:
    CodeGeneration(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `CodeGeneration` class contains the following members:

Constructors

CodeGeneration

CodeGeneration

```
C++
```

```
CodeGeneration(const RawEvent& event);
```

Parameters

event
A CODE_GENERATION event.

CommandLine class

Article • 10/29/2021

The `CommandLine` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a COMMAND_LINE event.

Syntax

```
C++
```

```
class CommandLine : public SimpleEvent
{
public:
    CommandLine(const RawEvent& event);

    const wchar_t* Value() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `CommandLine` class contains the following members:

Constructors

CommandLine

Functions

Value

CommandLine

```
C++
```

```
CommandLine(const RawEvent& event);
```

Parameters

event

A COMMAND_LINE event.

Value

```
C++
```

```
const wchar_t Value() const;
```

Return Value

A string containing a command line. The value includes arguments that were obtained from a response file and from environment variables such a CL, _CL_, Link, and _LINK_.

Compiler class

Article • 10/29/2021

The `Compiler` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a COMPILER event.

Syntax

```
C++
```

```
class Compiler : public Invocation
{
public:
    Compiler(const RawEvent& event);
};
```

Members

Along with the inherited members from its Invocation base class, the `Compiler` class contains the following members:

Constructors

Compiler

Compiler

```
C++
```

```
Compiler(const RawEvent& event);
```

Parameters

event
A COMPILER event.

CompilerPass class

Article • 03/28/2024

The `CompilerPass` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a BACK_END_PASS or FRONT_END_PASS event.

Syntax

C++

```
class CompilerPass : public Activity
{
public:
    enum class PassCode
    {
        FRONT_END,
        BACK_END
    };

    CompilerPass(const RawEvent& event);

    PassCode       PassCode() const;
    const wchar_t* InputSourcePath() const;
    const wchar_t* OutputObjectPath() const;
};
```

Members

Along with the inherited members from its Activity base class, the `CompilerPass` class contains the following members:

Constructors

CompilerPass

Enums

PassCode

Expand table

Value	Description
FRONT_END	The front-end pass.
BACK_END	The back-end pass.

Functions

InputSourcePath
OutputObjectPath
PassCode

CompilerPass

```
C++

CompilerPass(const RawEvent& event);
```

Parameters

event
A BACK_END_PASS or FRONT_END_PASS event.

InputSourcePath

```
C++

const wchar_t* InputSourcePath() const;
```

Return Value

The absolute path to the input source file processed by this compiler pass.

OutputObjectPath

```
C++

const wchar_t* OutputObjectPath() const;
```

Return Value

The absolute path to the output object file produced by this compiler pass.

PassCode

```
C++
```

```
PassCode PassCode() const;
```

Return Value

A code indicating which compiler pass is represented by this CompilerPass object.

Feedback

EnvironmentVariable class

Article • 10/29/2021

The `EnvironmentVariable` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an ENVIRONMENT_VARIABLE event.

Syntax

C++

```
class EnvironmentVariable : public SimpleEvent
{
public:
    EnvironmentVariable(const RawEvent& event);

    const wchar_t* Name() const;
    const wchar_t* Value() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `EnvironmentVariable` class contains the following members:

Constructors

EnvironmentVariable

Functions

Name Value

EnvironmentVariable

C++

```
EnvironmentVariable(const RawEvent& event);
```

Parameters

event
An ENVIRONMENT_VARIABLE event.

Name

```
C++
```

```
const wchar_t Name() const;
```

Return Value

The environment variable's name.

Value

```
C++
```

```
const wchar_t Value() const;
```

Return Value

The environment variable's value.

Event class

Article • 10/29/2021

The `Event` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match any event.

Syntax

C++

```
class Event
{
public:
    Event(const RawEvent& event);

    const unsigned short&        EventId() const;
    const unsigned long long&    EventInstanceId() const;
    const long long&             TickFrequency() const;
    const long long&             Timestamp() const;
    const unsigned long&         ProcessId() const;
    const unsigned long&         ThreadId() const;
    const unsigned short&        ProcessorIndex() const;
    const char*                  EventName() const;
    const wchar_t*               EventWideName() const;
};
```

Members

Constructors

Event

Functions

Data EventId
EventInstanceId
EventName
EventWideName
ProcessId
ProcessorIndex

ThreadId
TickFrequency
Timestamp

Event

```
C++
```

```
Event(const RawEvent& event);
```

Parameters

event
Any event.

Data

```
C++
```

```
const void* Data() const;
```

Return Value

A pointer to extra data contained in this event. For more information on how to interpret this field, see EVENT_DATA.

EventId

```
C++
```

```
const unsigned short& EventId() const;
```

Return Value

A number that identifies the event type. For a list of event identifiers, see EVENT_ID.

EventInstanceId

```
```

```
C++
```

```
const unsigned long long& EventInstanceId() const;
```

Return Value

A number that uniquely identifies the event inside a trace. This value doesn't change when analyzing or relogging the same trace multiple times. Use this value to identify the same event in multiple analysis or relogging passes over the same trace.

EventName

```
C++
```

```
const char* EventName() const;
```

Return Value

An ANSI string containing the name of the event type identified by EventId.

EventWideName

```
C++
```

```
const wchar_t* EventWideName() const;
```

Return Value

A wide string containing the name of the event identified by EventId.

ProcessId

```
C++
```

```
const unsigned long& ProcessId() const;
```

Return Value

The identifier for the process in which the event occurred.

ProcessorIndex

```
C++
```

```
const unsigned short& ProcessorIndex() const;
```

Return Value

The zero-based index for the logical processor on which the event occurred.

ThreadId

```
C++
```

```
const unsigned long& ThreadId() const;
```

Return Value

The identifier for the thread in which the event occurred.

TickFrequency

```
C++
```

```
const long long& TickFrequency() const;
```

Return Value

The number of ticks per second to use when evaluating a duration measured in ticks for this event.

Timestamp

```
C++
```

```
const long long& Timestamp() const;
```

Return Value

If the event is an activity, this function returns a tick value captured at the time the activity started. For a simple event, this function returns a tick value captured at the time the event occurred.

EventGroup class

Article • 10/29/2021

The `EventGroup` class template is the base class for all group capture classes.

Syntax

C++

```
template <typename TActivity>
class EventGroup;
{
public:
    size_t Size() const;

    const TActivity& operator[](size_t index) const;
    const TActivity& Front() const;
    const TActivity& Back() const;

    std::deque<TActivity>::const_iterator begin() const;
    std::deque<TActivity>::const_iterator end() const;
};
```

Parameters

TActivity The activity type contained in the group.

Members

Functions

Back begin end Front operator[] Size

Back

C++

```
const TActivity& Back() const;
```

Return Value

A reference to the last activity event in the group.

begin

```
C++
```

```
std::deque<TActivity>::const_iterator begin() const;
```

Return Value

An iterator pointing at the beginning of the activity event group.

end

```
C++
```

```
std::deque<TActivity>::const_iterator end() const;
```

Return Value

An iterator pointing one position past the end of the activity event group.

Front

```
C++
```

```
const TActivity& Front() const;
```

Return Value

A reference to the first activity event in the group.

operator[]

```
C++
```

```
const TActivity& operator[](size_t index) const;
```

Parameters

index
The index of the element to access in the activity event group.

Return Value

The event from the event stack stored at the position indicated by *index*.

Size

```
C++
```

```
size_t Size() const;
```

Return Value

The size of the event group.

EventStack class

Article • 10/29/2021

The `EventStack` class is a collection of Event objects. All events received from the C++ Build Insights SDK come in the form of an `EventStack` object. The last entry in this stack is the event currently being processed. The entries that precede the last entry are the parent hierarchy of the current event. For more information on the event model used in C++ Build Insights, see event table.

Syntax

C++

```
class EventStack
{
public:
    EventStack(const EVENT_COLLECTION_DATA& data);

    size_t      Size() const;
    RawEvent    Back() const;
    RawEvent    operator[] (size_t index) const;
};
```

Members

Constructors

EventStack

Functions

Back operator[] Size

Back

C++

```
RawEvent Back() const;
```

Return Value

A RawEvent object that represents the last entry in the stack. The last entry in an event stack is the event that was triggered.

EventStack

```
C++

EventStack(const EVENT_COLLECTION_DATA& data);
```

Parameters

data
The raw data from which the `EventStack` is built.

Remarks

You don't typically need to construct `EventStack` objects yourself. They're provided to you by the C++ Build Insights SDK when events are being processed during an analysis or relogging session.

operator[]

```
C++

RawEvent operator[] (size_t index) const;
```

Parameters

index
The index of the element to access in the event stack.

Return Value

A RawEvent object that represents the event stored at the position indicated by *index* in the event stack.

Size

```C++
size_t Size() const;
```

Return Value

The size of the event stack.

ExecutableImageOutput class

Article • 10/29/2021

The `ExecutableImageOutput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an EXECUTABLE_IMAGE_OUTPUT event.

Syntax

C++

```
class ExecutableImageOutput : public FileOutput
{
public:
    ExecutableImageOutput(const RawEvent& event);
};
```

Members

Along with the inherited members from its FileOutput base class, the `ExecutableImageOutput` class contains the following members:

Constructors

ExecutableImageOutput

ExecutableImageOutput

C++

```
ExecutableImageOutput(const RawEvent& event);
```

Parameters

event
An EXECUTABLE_IMAGE_OUTPUT event.

ExpOutput class

Article • 10/29/2021

The `ExpOutput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an EXP_OUTPUT event.

Syntax

C++

```
class ExpOutput : public FileOutput
{
public:
    ExpOutput(const RawEvent& event);
};
```

Members

Along with the inherited members from its FileOutput base class, the `ExpOutput` class contains the following members:

Constructors

ExpOutput

ExpOutput

C++

```
ExpOutput(const RawEvent& event);
```

Parameters

event
An EXP_OUTPUT event.

FileInput class

Article • 10/29/2021

The `FileInput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a FILE_INPUT event.

Syntax

C++

```
class FileInput : public SimpleEvent
{
public:
    enum class Type
    {
        OTHER               = FILE_TYPE_CODE_OTHER,
        OBJ                 = FILE_TYPE_CODE_OBJ,
        EXECUTABLE_IMAGE    = FILE_TYPE_CODE_EXECUTABLE_IMAGE,
        LIB                 = FILE_TYPE_CODE_LIB,
        IMP_LIB             = FILE_TYPE_CODE_IMP_LIB,
        EXP                 = FILE_TYPE_CODE_EXP
    };

    FileInput(const RawEvent& event);

    const wchar_t* Path() const;
    Type           Type() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `FileInput` class contains the following members:

Constructors

FileInput

Functions

Path Type

FileInput

```
C++
```

```
FileInput(const RawEvent& event);
```

Parameters

event
A FILE_INPUT event.

Path

```
C++
```

```
const wchar_t Path() const;
```

Return Value

The absolute path to the input file.

Type

```
C++
```

```
Type Type() const;
```

Return Value

A code describing the type of input file.

FileOutput class

Article • 10/29/2021

The `FileOutput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an EXECUTABLE_IMAGE_OUTPUT, EXP_OUTPUT, IMP_LIB_OUTPUT, LIB_OUTPUT, or OBJ_OUTPUT event.

Syntax

C++

```
class FileOutput : public SimpleEvent
{
public:
    enum class Type
    {
        OTHER               = FILE_TYPE_CODE_OTHER,
        OBJ                 = FILE_TYPE_CODE_OBJ,
        EXECUTABLE_IMAGE    = FILE_TYPE_CODE_EXECUTABLE_IMAGE,
        LIB                 = FILE_TYPE_CODE_LIB,
        IMP_LIB             = FILE_TYPE_CODE_IMP_LIB,
        EXP                 = FILE_TYPE_CODE_EXP
    };

    FileOutput(const RawEvent& event);

    const wchar_t* Path() const;
    Type           Type() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `FileOutput` class contains the following members:

Constructors

FileOutput

Functions

Path Type

FileOutput

```
C++

FileOutput(const RawEvent& event);
```

Parameters

event
An EXECUTABLE_IMAGE_OUTPUT, EXP_OUTPUT, IMP_LIB_OUTPUT, LIB_OUTPUT, or OBJ_OUTPUT event.

Path

```
C++

const wchar_t Path() const;
```

Return Value

The absolute path to the output file.

Type

```
C++

Type Type() const;
```

Return Value

A code describing the type of output file.

ForceInlinee class

Article • 10/29/2021

The `ForceInlinee` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a FORCE_INLINEE event.

Syntax

C++

```
class ForceInlinee : public SimpleEvent
{
public:
    ForceInlinee(const RawEvent& event);

    const char*             Name() const;
    const unsigned short&   Size() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `ForceInlinee` class contains the following members:

Constructors

ForceInlinee

Functions

Name Size

ForceInlinee

C++

```
ForceInlinee(const RawEvent& event);
```

Parameters

event
A FORCE_INLINEE event.

Name

```
C++
```

```
const char* Name() const;
```

Return Value

The name of the force-inlined function, encoded in UTF-8.

Size

```
C++
```

```
const unsigned short& Size() const;
```

Return Value

The size of the force-inlined function, as an intermediate instruction count.

FrontEndFile class

Article • 10/29/2021

The `FrontEndFile` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a FRONT_END_FILE event.

Syntax

C++

```
class FrontEndFile : public Activity
{
public:
    FrontEndFile(const RawEvent& event);

    const char* Path() const;
};
```

Members

Along with the inherited members from its Activity base class, the `FrontEndFile` class contains the following members:

Constructors

FrontEndFile

Functions

Path

FrontEndFile

C++

```
FrontEndFile(const RawEvent& event);
```

Parameters

event

A FRONT_END_FILE event.

Path

```
C++
```

```
const char* Path() const;
```

Return Value

The absolute path to the file, encoded in UTF-8.

FrontEndFileGroup class

Article • 10/29/2021

The `FrontEndFileGroup` class is used with the MatchEventStack and MatchEventStackInMemberFunction functions. Use it to match groups of FRONT_END_FILE events.

Syntax

C++

```
class FrontEndFileGroup : public EventGroup<FrontEndFile>
{
public:
    FrontEndFileGroup(std::deque<FrontEndFile>&& group);
};
```

Members

Along with the inherited members from its EventGroup<FrontEndFile> base class, the `FrontEndFileGroup` class contains the following members:

Constructors

FrontEndFileGroup

FrontEndFileGroup

C++

```
FrontEndFileGroup(std::deque<FrontEndFile>&& group);
```

Parameters

group
A group of FRONT_END_FILE events.

FrontEndPass class

Article • 10/29/2021

The `FrontEndPass` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a FRONT_END_PASS event.

Syntax

C++

```
class FrontEndPass : public CompilerPass
{
public:
    FrontEndPass(const RawEvent& event);
};
```

Members

Along with the inherited members from its CompilerPass base class, the `FrontEndPass` class contains the following members:

Constructors

FrontEndPass

FrontEndPass

C++

```
FrontEndPass(const RawEvent& event);
```

Parameters

event
A FRONT_END_PASS event.

Function class

Article • 04/27/2022

The `Function` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a FUNCTION event.

Syntax

C++

```
class Function : public Activity
{
public:
    Function(const RawEvent& event);

    const char* Name() const;
};
```

Members

Along with the inherited members from the Activity base class, the `Function` class contains the following members:

Constructors

Function

Functions

Name

Function

C++

```
Function(const RawEvent& event);
```

Parameters

event

A FUNCTION event.

Name

```
C++
```

```
const char* Name() const;
```

Return Value

The name of the function, encoded in UTF-8.

HeaderUnit class

Article • 10/29/2021

The `HeaderUnit` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a HEADER_UNIT event.

Syntax

C++

```
class HeaderUnit : public TranslationUnitType
{
public:
    HeaderUnit(const RawEvent& event);
};
```

Members

Along with the inherited members from its TranslationUnitType base class, the `HeaderUnit` class contains the following members:

Constructors

HeaderUnit

Functions

None

HeaderUnit

C++

```
HeaderUnit(const RawEvent& event);
```

Parameters

event

A HEADER_UNIT event.

ImpLibOutput class

Article • 10/29/2021

The `ImpLibOutput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an IMP_LIB_OUTPUT event.

Syntax

C++

```
class ImpLibOutput : public FileOutput
{
public:
    ImpLibOutput(const RawEvent& event);
};
```

Members

Along with the inherited members from its FileOutput base class, the `ImpLibOutput` class contains the following members:

Constructors

ImpLibOutput

ImpLibOutput

C++

```
ImpLibOutput(const RawEvent& event);
```

Parameters

event
An IMP_LIB_OUTPUT event.

Invocation class

Article • 10/29/2021

The `Invocation` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a COMPILER or LINKER event.

Syntax

C++

```
class Invocation : public Activity
{
    const INVOCATION_DATA* data_;

public:
    enum class Type
    {
        CL      = MSVC_TOOL_CODE_CL,
        LINK    = MSVC_TOOL_CODE_LINK
    };

    Invocation(const RawEvent& event);

    Type             Type() const;
    const char*      ToolVersionString() const;
    const wchar_t*   WorkingDirectory() const;
    const wchar_t*   ToolPath() const;

    const INVOCATION_VERSION_DATA& ToolVersion() const;
};
```

Members

Along with the inherited members from its Activity base class, the `Invocation` class contains the following members:

Constructors

Invocation

Functions

Invocation

```
C++

Invocation(const RawEvent& event);
```

Parameters

event
A COMPILER or LINKER event.

ToolPath

```
C++

const wchar_t* ToolPath() const;
```

Return Value

The absolute path to the tool that was invoked.

ToolVersion

```
C++

const INVOCATION_VERSION_DATA& ToolVersion() const;
```

Return Value

The version of the tool that was invoked, as an INVOCATION_VERSION_DATA reference.

ToolVersionString

```
C++

const char* ToolVersionString() const;
```

Return Value

The version of the tool that was invoked, as an ANSI string.

Type

```
C++

Type Type() const;
```

Return Value

A code indicating the tool that was invoked.

WorkingDirectory

```
C++

const wchar_t* WorkingDirectory() const;
```

Return Value

The absolute path to the directory in which the tool was invoked.

InvocationGroup class

Article • 10/29/2021

The `InvocationGroup` class is used with the MatchEventStack and MatchEventStackInMemberFunction functions. Use it to match groups containing a mix of COMPILER and LINKER events.

Syntax

C++

```
class InvocationGroup : public EventGroup<Invocation>
{
public:
    InvocationGroup(std::deque<Invocation>&& group);
};
```

Members

Along with the inherited members from its EventGroup<Invocation> base class, the `InvocationGroup` class contains the following members:

Constructors

InvocationGroup

InvocationGroup

C++

```
InvocationGroup(std::deque<Invocation>&& group);
```

Parameters

group
A group containing a mix of COMPILER and LINKER events.

LibOutput class

Article • 10/29/2021

The `LibOutput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a LIB_OUTPUT event.

Syntax

C++

```
class LibOutput : public FileOutput
{
public:
    LibOutput(const RawEvent& event);
};
```

Members

Along with the inherited members from its FileOutput base class, the `LibOutput` class contains the following members:

Constructors

LibOutput

LibOutput

C++

```
LibOutput(const RawEvent& event);
```

Parameters

event
A LIB_OUTPUT event.

Linker class

Article • 10/29/2021

The `Linker` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a LINKER event.

Syntax

```
C++
```

```
class Linker : public Invocation
{
public:
    Linker(const RawEvent& event);
};
```

Members

Along with the inherited members from its Invocation base class, the `Linker` class contains the following members:

Constructors

Linker

Linker

```
C++
```

```
Linker(const RawEvent& event);
```

Parameters

event
A LINKER event.

LinkerGroup class

Article • 10/29/2021

The `LinkerGroup` class is used with the MatchEventStack and MatchEventStackInMemberFunction functions. Use it to match groups of LINKER events.

Syntax

C++

```
class LinkerGroup : public EventGroup<Linker>
{
public:
    LinkerGroup(std::deque<Linker>&& group);
};
```

Members

Along with the inherited members from its EventGroup<Linker> base class, the `LinkerGroup` class contains the following members:

Constructors

LinkerGroup

LinkerGroup

C++

```
LinkerGroup(std::deque<Linker>&& group);
```

Parameters

group
A group of LINKER events.

LinkerPass class

Article • 10/29/2021

The `LinkerPass` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a PASS1 or PASS2 event.

Syntax

```
C++
```

```
class LinkerPass : public Activity
{
public:
    LinkerPass(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `LinkerPass` class contains the following members:

Constructors

LinkerPass

LinkerPass

```
C++
```

```
LinkerPass(const RawEvent& event);
```

Parameters

event
A PASS1 or PASS2 event.

LTCG class

Article • 10/29/2021

The `LTCG` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a LTCG event.

Syntax

```
C++
```

```
class LTCG : public Activity
{
public:
    LTCG(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `LTCG` class contains the following members:

Constructors

LTCG

LTCG

```
C++
```

```
LTCG(const RawEvent& event);
```

Parameters

event
A LTCG event.

Module class

Article • 10/29/2021

The `Module` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a MODULE event.

Syntax

```
C++

class Module : public TranslationUnitType
{
public:
    Module(const RawEvent& event);
};
```

Members

Along with the inherited members from its TranslationUnitType base class, the `Module` class contains the following members:

Constructors

Module

Functions

None

Module

```
C++

Module(const RawEvent& event);
```

Parameters

event

A MODULE event.

ObjOutput class

Article • 10/29/2021

The `ObjOutput` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an OBJ_OUTPUT event.

Syntax

```
C++

class ObjOutput : public FileOutput
{
public:
    ObjOutput(const RawEvent& event);
};
```

Members

Along with the inherited members from its FileOutput base class, the `ObjOutput` class contains the following members:

Constructors

ObjOutput

ObjOutput

```
C++

ObjOutput(const RawEvent& event);
```

Parameters

event
An OBJ_OUTPUT event.

OptICF class

Article • 10/29/2021

The `OptICF` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an OPT_ICF event.

Syntax

```
C++
```

```
class OptICF : public Activity
{
public:
    OptICF(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `OptICF` class contains the following members:

Constructors

OptICF

OptICF

```
C++
```

```
OptICF(const RawEvent& event);
```

Parameters

event
An OPT_ICF event.

OptLBR class

Article • 10/29/2021

The `OptLBR` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an OPT_LBR event.

Syntax

C++

```
class OptLBR : public Activity
{
public:
    OptLBR(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `OptLBR` class contains the following members:

Constructors

OptLBR

OptLBR

C++

```
OptLBR(const RawEvent& event);
```

Parameters

event
An OPT_LBR event.

OptRef class

Article • 10/29/2021

The `OptRef` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match an OPT_REF event.

Syntax

C++

```
class OptRef : public Activity
{
public:
    OptRef(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `OptRef` class contains the following members:

Constructors

OptRef

OptRef

C++

```
OptRef(const RawEvent& event);
```

Parameters

event
An OPT_REF event.

Pass1 class

Article • 04/27/2022

The `Pass1` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a PASS1 event.

Syntax

C++

```
class Pass1 : public LinkerPass
{
public:
    Pass1(const RawEvent& event);
};
```

Members

Along with the inherited members from the LinkerPass base class, the `Pass1` class contains the following members:

Constructors

Pass1

Pass1

C++

```
Pass1(const RawEvent& event);
```

Parameters

event
A PASS1 event.

Pass2 class

Article • 10/29/2021

The `Pass2` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a PASS2 event.

Syntax

```
C++
```

```
class Pass2 : public LinkerPass
{
public:
    Pass2(const RawEvent& event);
};
```

Members

Along with the inherited members from its LinkerPass base class, the `Pass2` class contains the following members:

Constructors

Pass2

Pass2

```
C++
```

```
Pass2(const RawEvent& event);
```

Parameters

event
A PASS2 event.

PrecompiledHeader class

Article • 10/29/2021

The `PrecompiledHeader` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a PRECOMPILED_HEADER event.

Syntax

C++

```
class PrecompiledHeader : public TranslationUnitType
{
public:
    PrecompiledHeader(const RawEvent& event);
};
```

Members

Along with the inherited members from its TranslationUnitType base class, the `PrecompiledHeader` class contains the following members:

Constructors

PrecompiledHeader

Functions

None

PrecompiledHeader

C++

```
PrecompiledHeader(const RawEvent& event);
```

Parameters

event

A PRECOMPILED_HEADER event.

PreLTCGOptRef class

Article • 10/29/2021

The `PreLTCGOptRef` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a PRE_LTCG_OPT_REF event.

Syntax

C++

```
class PreLTCGOptRef : public Activity
{
public:
    PreLTCGOptRef(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `PreLTCGOptRef` class contains the following members:

Constructors

PreLTCGOptRef

PreLTCGOptRef

C++

```
PreLTCGOptRef(const RawEvent& event);
```

Parameters

event
A PRE_LTCG_OPT_REF event.

RawEvent class

Article • 10/29/2021

The `RawEvent` class is used to represent a general event in an EventStack.

Syntax

C++

```
class RawEvent
{
public:
    RawEvent(const EVENT_DATA& event);

    const unsigned short&        EventId() const;
    const unsigned long long&    EventInstanceId() const;
    const long long&             TickFrequency() const;
    const long long&             StartTimestamp() const;
    const long long&             StopTimestamp() const;
    const long long&             ExclusiveDurationTicks() const;
    const long long&             CPUTicks() const;
    const long long&             ExclusiveCPUTicks() const;
    const long long&             WallClockTimeResponsibilityTicks() const;
    const long long&             ExclusiveWallClockTimeResponsibilityTicks()
const;
    const void*                  Data() const;
    const unsigned long&         ProcessId() const;
    const unsigned long&         ThreadId() const;
    const unsigned short&        ProcessorIndex() const;
    const char*                  EventName() const;
    const wchar_t*               EventWideName() const;

    std::chrono::nanoseconds Duration() const;
    std::chrono::nanoseconds ExclusiveDuration() const;
    std::chrono::nanoseconds CPUTime() const ;
    std::chrono::nanoseconds ExclusiveCPUTime() const;
    std::chrono::nanoseconds WallClockTimeResponsibility() const;
    std::chrono::nanoseconds ExclusiveWallClockTimeResponsibility() const;
};
```

Remarks

Several member functions in the `RawEvent` class return a tick count. C++ Build Insights uses Windows' performance counter as a source of ticks. A tick count must be used with a tick frequency to convert it into a time unit like seconds. The `TickFrequency` member

function may be called to obtain the tick frequency. See the EVENT_DATA page for an example on how to convert ticks into a time unit.

If you don't want to convert ticks yourself, the `RawEvent` class provides member functions that return time values in nanoseconds. Use the standard C++ `chrono` library to convert nanoseconds into other time units.

Members

Constructor

RawEvent

Functions

CPUTicks
CPUTime
Data
Duration
EventId EventInstanceId EventName
EventWideName
ExclusiveCPUTicks
ExclusiveCPUTime
ExclusiveDuration
ExclusiveDurationTicks
ExclusiveWallClockTimeResponsibility
ExclusiveWallClockTimeResponsibilityTicks
ProcessId
ProcessorIndex
StartTimestamp
StopTimestamp
ThreadId
TickFrequency
WallClockTimeResponsibility
WallClockTimeResponsibilityTicks

RawEvent

```
C++
```

```
RawEvent(const EVENT_DATA& data);
```

Parameters

event
The event data.

CPUTicks

C++

```
const long long& CPUTicks() const;
```

Return Value

The number of CPU ticks that occurred during this activity. A CPU tick is different from a regular tick. CPU ticks are only counted when the CPU is executing code in an activity. CPU ticks aren't counted when the thread associated with the activity is sleeping.

CPUTime

C++

```
std::chrono::nanoseconds CPUTime()() const;
```

Return Value

The amount of time the CPU was executing code inside this activity. This value may be higher than the duration of the activity if child activities are executed on separate threads. The value is returned in nanoseconds.

Data

C++

```
const void* Data() const;
```

Return Value

A pointer to extra data contained in this event. For more information on how to interpret this field, see EVENT_DATA.

Duration

```
C++
```

```
std::chrono::nanoseconds Duration() const;
```

Return Value

The duration of the activity in nanoseconds.

EventId

```
C++
```

```
const unsigned short& EventId() const;
```

Return Value

A number that identifies the event type. For a list of event identifiers, see EVENT_ID.

EventInstanceId

```
C++
```

```
const unsigned long long& EventInstanceId() const;
```

Return Value

A number that uniquely identifies the event inside a trace. This value doesn't change when analyzing or relogging the same trace multiple times. Use this value to identify the same event in multiple analysis or relogging passes over the same trace.

EventName

```
C++
```

```
const char* EventName() const;
```

Return Value

An ANSI string containing the name of the event type identified by EventId.

EventWideName

```
C++
```

```
const wchar_t* EventWideName() const;
```

Return Value

A wide string containing the name of the event type identified by EventId.

ExclusiveCPUTicks

```
C++
```

```
const long long& ExclusiveCPUTicks() const;
```

Return Value

Same as CPUTicks, but not including the CPU ticks that occurred in child activities.

ExclusiveCPUTime

```
C++
```

```
std::chrono::nanoseconds ExclusiveCPUTime() const;
```

Return Value

Same as CPUTime, except that the CPU time of child activities isn't included.

ExclusiveDuration

```
C++
```

```
std::chrono::nanoseconds ExclusiveDuration() const;
```

Return Value

The duration of the activity in nanoseconds, not including the amount of time that was spent in child activities.

ExclusiveDurationTicks

```
C++
```

```
const long long& ExclusiveDurationTicks() const;
```

Return Value

The number of ticks that occurred in this activity, excluding the number of ticks that occurred in child activities.

ExclusiveWallClockTimeResponsibility

```
C++
```

```
std::chrono::nanoseconds ExclusiveWallClockTimeResponsibility() const;
```

Return Value

Same as WallClockTimeResponsibility, but not including the wall-clock time responsibility of child activities.

ExclusiveWallClockTimeResponsibilityTicks

```
C++
```

```
const long long& ExclusiveWallClockTimeResponsibilityTicks() const;
```

Return Value

Same as WallClockTimeResponsibilityTicks, but not including the wall-clock time responsibility ticks of child activities.

ProcessId

```
C++
```

```
const unsigned long& ProcessId() const;
```

Return Value

The identifier for the process in which the event occurred.

ProcessorIndex

```
C++
```

```
const unsigned short& ProcessorIndex() const;
```

Return Value

The zero-based index for the logical processor on which the event occurred.

StartTimestamp

```
C++
```

```
const long long& StartTimestamp() const;
```

Return Value

A tick value captured at the time the activity started.

StopTimestamp

```
C++
```

```
const long long& StopTimestamp() const;
```

Return Value

A tick value captured at the time the activity stopped.

ThreadId

C++

```
const unsigned long& ThreadId() const;
```

Return Value

The identifier for the thread in which the event occurred.

TickFrequency

C++

```
const long long& TickFrequency() const;
```

Return Value

The number of ticks per second to use when evaluating a duration measured in ticks for this event.

WallClockTimeResponsibility

C++

```
std::chrono::nanoseconds WallClockTimeResponsibility() const;
```

Return Value

The wall-clock time responsibility of this activity, in nanoseconds. For more information on what wall-clock time responsibility means, see WallClockTimeResponsibilityTicks.

WallClockTimeResponsibilityTicks

```
C++
```

```
const long long& WallClockTimeResponsibilityTicks() const;
```

Return Value

A tick count that represents this activity's contribution to overall wall-clock time. A wall-clock time responsibility tick is different from a regular tick. Wall-clock time responsibility ticks take into account parallelism between activities. Two parallel activities may have a duration of 50 ticks and the same start and stop time. In this case, both get assigned a wall-clock time responsibility of 25 ticks.

SimpleEvent class

Article • 10/29/2021

The `SimpleEvent` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match any simple event. Refer to the event table to see all events that can be matched by the `SimpleEvent` class.

Syntax

C++

```
class SimpleEvent : public Event
{
public:
    SimpleEvent(const RawEvent& event);
};
```

Members

Along with the inherited members from its Event base class, the `SimpleEvent` class contains the following members:

Constructors

SimpleEvent

SimpleEvent

C++

```
SimpleEvent(const RawEvent& event);
```

Parameters

event
Any simple event.

SymbolName class

Article • 10/29/2021

The `SymbolName` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a SYMBOL_NAME event.

Syntax

C++

```
class SymbolName : public SimpleEvent
{
public:
    SymbolName(const RawEvent& event);

    const unsigned long long&  Key() const;
    const char*                Name() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `SymbolName` class contains the following members:

Constructors

SymbolName

Functions

Key Name

Key

C++

```
const unsigned long long& Key() const;
```

Return Value

A numerical identifier for the type represented by this symbol. This identifier is unique within a compiler front-end pass.

Name

```
C++
```

```
const char* Name() const;
```

Return Value

The name of the type represented by the symbol, encoded in UTF-8.

SymbolName

```
C++
```

```
SymbolName(const RawEvent& event);
```

Parameters

event
A SYMBOL_NAME event.

TemplateInstantiation class

Article • 10/29/2021

The `TemplateInstantiation` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a TEMPLATE_INSTANTIATION event.

Syntax

C++

```
class TemplateInstantiation : public Activity
{
public:
    enum class Kind
    {
        CLASS       = TEMPLATE_INSTANTIATION_KIND_CODE_CLASS,
        FUNCTION    = TEMPLATE_INSTANTIATION_KIND_CODE_FUNCTION,
        VARIABLE    = TEMPLATE_INSTANTIATION_KIND_CODE_VARIABLE,
        CONCEPT     = TEMPLATE_INSTANTIATION_KIND_CODE_CONCEPT
    };

    TemplateInstantiation(const RawEvent& event);

    const unsigned long long& SpecializationSymbolKey() const;
    const unsigned long long& PrimaryTemplateSymbolKey() const;

    Kind Kind() const;
};
```

Members

Along with the inherited members from its Activity base class, the `TemplateInstantiation` class contains the following members:

Constructors

TemplateInstantiation

Functions

Kind PrimaryTemplateSymbolKey SpecializationSymbolKey

Kind

```
C++

Kind Kind() const;
```

Return Value

A code describing the type of template instantiation that was done.

PrimaryTemplateSymbolKey

```
C++

const unsigned long long& PrimaryTemplateSymbolKey() const;
```

Return Value

A numerical identifier for the template type that was specialized. This identifier is unique within a compiler front-end pass.

SpecializationSymbolKey

```
C++

const unsigned long long& SpecializationSymbolKey() const;
```

Return Value

A numerical identifier for the specialization's type. This identifier is unique within a compiler front-end pass.

TemplateInstantiation

```
C++

TemplateInstantiation(const RawEvent& event);
```

Parameters

event

A TEMPLATE_INSTANTIATION event.

TemplateInstantiationGroup class

Article • 10/29/2021

The `TemplateInstantiationGroup` class is used with the MatchEventStack and MatchEventStackInMemberFunction functions. Use it to match groups of TEMPLATE_INSTANTIATION events.

Syntax

```
C++
```

```
class TemplateInstantiationGroup : public EventGroup<TemplateInstantiation>
{
public:
    TemplateInstantiationGroup(std::deque<TemplateInstantiation>&& group);
};
```

Members

Along with the inherited members from its EventGroup<TemplateInstantiation> base class, the `TemplateInstantiationGroup` class contains the following members:

Constructors

TemplateInstantiationGroup

TemplateInstantiationGroup

```
C++
```

```
TemplateInstantiationGroup(std::deque<TemplateInstantiation>&& group);
```

Parameters

group
A group of TEMPLATE_INSTANTIATION events.

Thread class

Article • 10/29/2021

The `Thread` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a THREAD event.

Syntax

```
C++
```

```
class Thread : public Activity
{
public:
    Thread(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `Thread` class contains the following members:

Constructors

Thread

Thread

```
C++
```

```
Thread(const RawEvent& event);
```

Parameters

event
A THREAD event.

TopDown class

Article • 10/29/2021

The `TopDown` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a TOP_DOWN event.

Syntax

```
C++
```

```
class TopDown : public Activity
{
public:
    TopDown(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `TopDown` class contains the following members:

Constructors

TopDown

TopDown

```
C++
```

```
TopDown(const RawEvent& event);
```

Parameters

event
A TOP_DOWN event.

TraceInfo class

Article • 10/29/2021

The `TraceInfo` class is used to access useful properties about a trace being analyzed or relogged.

Syntax

C++

```
class TraceInfo
{
public:
    TraceInfo(const TRACE_INFO_DATA& data);

    const unsigned long& LogicalProcessorCount() const;

    const long long& TickFrequency() const;
    const long long& StartTimestamp() const;
    const long long& StopTimestamp() const;

    std::chrono::nanoseconds Duration() const;
};
```

Remarks

Subtract the `StartTimestamp` from `StopTimestamp` to obtain the number of ticks elapsed during the whole trace. Use `TickFrequency` to convert the resulting value into a time unit. For an example of converting ticks to time, see EVENT_DATA.

If you don't want to convert ticks yourself, the `TraceInfo` class provides a member function that returns the trace duration in nanoseconds. Use the standard C++ `chrono` library to convert this value into other time units.

Members

Constructors

TraceInfo

Functions

Duration LogicalProcessorCount StartTimestamp StopTimestamp TickFrequency

Duration

C++

```
std::chrono::nanoseconds Duration() const;
```

Return Value

The duration of the activity in nanoseconds.

LogicalProcessorCount

C++

```
const unsigned long& LogicalProcessorCount() const;
```

Return Value

The number of logical processors on the machine where the trace was collected.

StartTimestamp

C++

```
const long long& StartTimestamp() const;
```

Return Value

A tick value captured at the time the trace was started.

StopTimestamp

C++

```
const long long& StopTimestamp() const;
```

Return Value

A tick value captured at the time the trace was stopped.

TickFrequency

C++

```
const long long& TickFrequency() const;
```

Return Value

The number of ticks per second to use when evaluating a duration measured in ticks.

TraceInfo

C++

```
TraceInfo(const TRACE_INFO_DATA& data);
```

Parameters

data
The data containing the information about the trace.

TranslationUnitType class

Article • 10/29/2021

The `TranslationUnitType` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a TRANSLATION_UNIT_TYPE event.

Syntax

C++

```
class TranslationUnitType : public SimpleEvent
{
public:
    enum class Type
    {
        MODULE              = TRANSLATION_UNIT_TYPE_MODULE,
        HEADER_UNIT         = TRANSLATION_UNIT_TYPE_HEADER_UNIT,
        PCH                 = TRANSLATION_UNIT_TYPE_PRECOMPILED_HEADER
    };

    TranslationUnitType(const RawEvent& event);

    Type Type() const;
};
```

Members

Along with the inherited members from its SimpleEvent base class, the `TranslationUnitType` class contains the following members:

Constructors

TranslationUnitType

Functions

Type

Type

C++

```
Type Type() const;
```

Return Value

The type of the translation unit: either MODULE, HEADER_UNIT, or PCH.

TranslationUnitType

C++

```
TranslationUnitType(const RawEvent& event);
```

Parameters

event
A TRANSLATION_UNIT_TYPE event.

WholeProgramAnalysis class

Article • 10/29/2021

The `WholeProgramAnalysis` class is used with the MatchEvent, MatchEventInMemberFunction, MatchEventStack, and MatchEventStackInMemberFunction functions. Use it to match a WHOLE_PROGRAM_ANALYSIS event.

Syntax

C++

```
class WholeProgramAnalysis : public Activity
{
public:
    WholeProgramAnalysis(const RawEvent& event);
};
```

Members

Along with the inherited members from its Activity base class, the `WholeProgramAnalysis` class contains the following members:

Constructors

WholeProgramAnalysis

WholeProgramAnalysis

C++

```
WholeProgramAnalysis(const RawEvent& event);
```

Parameters

event
A WHOLE_PROGRAM_ANALYSIS event.

CL_PASS_DATA structure

Article • 10/29/2021

The `CL_PASS_DATA` structure describes a compilation pass.

Syntax

C++

```
typedef struct CL_PASS_DATA_TAG
{
    int                         TranslationUnitPassCode;
    const wchar_t*              InputSourcePath;
    const wchar_t*              OutputObjectPath;

} CL_PASS_DATA;
```

Members

Name	Description
`TranslationUnitPassCode`	A code identifying the compilation pass being executed. For more information, see TRANSLATION_UNIT_PASS_CODE.
`InputSourcePath`	The C or C++ source file on which this compilation pass is being executed.
`OutputObjectPath`	The object file being produced by the compiler.

EVENT_COLLECTION_DATA structure

Article • 10/29/2021

The `EVENT_COLLECTION_DATA` structure describes an array of EVENT_DATA elements.

Syntax

C++

```
typedef struct EVENT_COLLECTION_DATA_TAG
{
    unsigned int            Count;
    const EVENT_DATA*       Elements;

} EVENT_COLLECTION_DATA;
```

Members

Name	Description
`Count`	The number of `EVENT_DATA` elements in the array.
`Elements`	Pointer to the first `EVENT_DATA` element in the array.

EVENT_DATA structure

Article • 04/27/2022

The `EVENT_DATA` structure describes an event from an analysis or relogging session. The Analyze, AnalyzeA, AnalyzeW, Relog, RelogA, or RelogW functions start these sessions.

Syntax

C++

```
typedef struct EVENT_DATA_TAG
{
    unsigned short          EventId;
    unsigned long long      EventInstanceId;

    long long               TickFrequency;
    long long               StartTimestamp;
    long long               StopTimestamp;
    long long               ExclusiveDurationTicks;
    long long               CPUTicks;
    long long               ExclusiveCPUTicks;
    long long               WallClockTimeResponsibilityTicks;
    long long               ExclusiveWallClockTimeResponsibilityTicks;

    const void*             Data;

    unsigned long           ProcessId;
    unsigned long           ThreadId;
    unsigned short          ProcessorIndex;

    const char*             EventName;
    const wchar_t*          EventWideName;

} EVENT_DATA;
```

Members

Name	Description
`EventId`	A number that identifies the event. For a list of event identifiers, see EVENT_ID.

Name	Description
`EventInstanceId`	A number that uniquely identifies the current event inside a trace. This value doesn't change when analyzing or relogging the same trace multiple times. Use this field to identify the same event in multiple analysis or relogging passes over the same trace.
`TickFrequency`	The number of ticks per second to use when evaluating a duration measured in ticks.
`StartTimestamp`	When the event is an *Activity*, this field is the tick value captured at the time the activity started. If this event is a *Simple Event*, this field is a tick value captured at the time the event occurred.
`StopTimestamp`	When the event is an *Activity*, this field is the tick value captured at the time the activity stopped. If the stop event hasn't yet been received for this activity, this field is zero. If this event is a *Simple Event*, this field is zero.
`ExclusiveDurationTicks`	If this event is an *Activity*, this field is the number of ticks that occurred directly in this activity. The number of ticks that occurred in a child activity are excluded. This field is zero for *Simple Events*.
`CPUTicks`	If this event is an *Activity*, this field is the number of CPU ticks that occurred during this activity. A CPU tick is different from a regular tick. CPU ticks are only counted when the CPU is executing code in an activity. CPU ticks aren't counted when the thread associated with the activity is sleeping. This field is zero for *Simple Events*.
`ExclusiveCPUTicks`	This field has the same meaning as `CPUTicks`, except that it doesn't include CPU ticks that occurred in children activities. This field is zero for *Simple Events*.

Name	Description
`WallClockTimeResponsibilityTicks`	If this event is an *Activity*, this field is a tick count that represents this activity's contribution to overall wall-clock time. A wall-clock time responsibility tick is different from a regular tick. Wall-clock time responsibility ticks take into account parallelism between activities. For example, two parallel activities may have a duration of 50 ticks and the same start and stop time. In this case, both will be assigned a wall-clock time responsibility of 25 ticks. This field is zero for *Simple Events*.
`ExclusiveWallClockTimeResponsibilityTicks`	This field has the same meaning as `WallClockTimeResponsibilityTicks`, except that it doesn't include the wall-clock time responsibility ticks of children activities. This field is zero for *Simple Events*.
`Data`	Points to other data stored in the event. The type of data pointed to is different, depending on the `EventId` field.
`ProcessId`	The identifier for the process in which the event occurred.
`ThreadId`	The identifier for the thread in which the event occurred.
`ProcessorIndex`	The zero-indexed CPU number on which the event occurred.
`EventName`	An ANSI string containing the name of the entity identified by `EventId`.
`EventWideName`	A wide string containing the name of the entity identified by `EventId`.

Remarks

Many fields in `EVENT_DATA` contain tick counts. C++ Build Insights uses the Windows performance counter as a source of ticks. A tick count must be used with the `TickFrequency` field to convert it into an appropriate time unit such as seconds. See the tick conversion example, below.

`EVENT_DATA` doesn't contain a field for the regular tick count of an activity. To obtain this value, subtract `StartTimestamp` from `StopTimestamp`.

`EVENT_DATA` is a structure that's meant to be used by C API users. For C++ API users, classes like Event do time conversions automatically.

The value of the `EVENT_DATA` `Data` field depends on the value of its `EventId` field. The value of `Data` is described in the table below. Some entity identifiers may be missing from the table below. In this case, the `Data` field is set to `nullptr` or zero.

`EventId` value	**Type pointed to by `Data`**
`EVENT_ID_BACK_END_PASS`	CL_PASS_DATA
`EVENT_ID_COMMAND_LINE`	`const wchar_t`
`EVENT_ID_COMPILER`	INVOCATION_DATA
`EVENT_ID_ENVIRONMENT_VARIABLE`	NAME_VALUE_PAIR_DATA
`EVENT_ID_EXECUTABLE_IMAGE_OUTPUT`	FILE_DATA
`EVENT_ID_EXP_OUTPUT`	FILE_DATA
`EVENT_ID_FILE_INPUT`	FILE_DATA
`EVENT_ID_FORCE_INLINE`	FUNCTION_FORCE_INLINEE_DATA
`EVENT_ID_FRONT_END_FILE`	FRONT_END_FILE_DATA
`EVENT_ID_FRONT_END_PASS`	CL_PASS_DATA
`EVENT_ID_FUNCTION`	FUNCTION_DATA
`EVENT_ID_IMP_LIB_OUTPUT`	FILE_DATA
`EVENT_ID_LIB_OUTPUT`	FILE_DATA
`EVENT_ID_LINKER`	INVOCATION_DATA
`EVENT_ID_OBJ_OUTPUT`	FILE_DATA
`EVENT_ID_SYMBOL_NAME`	SYMBOL_NAME_DATA
`EVENT_ID_TEMPLATE_INSTANTIATION`	TEMPLATE_INSTANTIATION_DATA

Tick conversion example

C++

```
//
// We have the elapsed number of ticks, along with the
```

```
// number of ticks per second. We use these values
// to convert to the number of elapsed microseconds.
// To guard against loss-of-precision, we convert
// to microseconds *before* dividing by ticks-per-second.
//

long long ConvertDurationToMicroseconds(const sruct EVENT_DATA& eventData)
{
    long long duration = eventData.StopTimestamp - eventData.StartTimestamp;
    long long duration *= 1000000;
    return duration / eventData.TickFrequency;
}
```

EVENT_ID enum

Article • 10/29/2021

The `EVENT_ID` enum.

Members

Name	Value	Event details URL
`EVENT_ID_BACK_END_PASS`	1 (0x00000001)	BACK_END_PASS
`EVENT_ID_BOTTOM_UP`	2 (0x00000002)	BOTTOM_UP
`EVENT_ID_C1_DLL`	3 (0x00000003)	C1_DLL
`EVENT_ID_C2_DLL`	4 (0x00000004)	C2_DLL
`EVENT_ID_CODE_GENERATION`	5 (0x00000005)	CODE_GENERATION
`EVENT_ID_COMMAND_LINE`	6 (0x00000006)	COMMAND_LINE
`EVENT_ID_COMPILER`	7 (0x00000007)	COMPILER
`EVENT_ID_ENVIRONMENT_VARIABLE`	8 (0x00000008)	ENVIRONMENT_VARIABLE
`EVENT_ID_EXECUTABLE_IMAGE_OUTPUT`	9 (0x00000009)	EXECUTABLE_IMAGE_OUTPUT
`EVENT_ID_EXP_OUTPUT`	10 (0x000000A)	EXP_OUTPUT
`EVENT_ID_FILE_INPUT`	11 (0x000000B)	FILE_INPUT
`EVENT_ID_FORCE_INLINEE`	12 (0x000000C)	FORCE_INLINEE
`EVENT_ID_FRONT_END_FILE`	13 (0x000000D)	FRONT_END_FILE
`EVENT_ID_FRONT_END_PASS`	14 (0x000000E)	FRONT_END_PASS
`EVENT_ID_FUNCTION`	15 (0x000000F)	FUNCTION
`EVENT_ID_IMP_LIB_OUTPUT`	16 (0x00000010)	IMP_LIB_OUTPUT
`EVENT_ID_LIB_OUTPUT`	17 (0x00000011)	LIB_OUTPUT
`EVENT_ID_LINKER`	18 (0x00000012)	LINKER
`EVENT_ID_LTCG`	19 (0x00000013)	LTCG
`EVENT_ID_OBJ_OUTPUT`	20 (0x00000014)	OBJ_OUTPUT

Name	Value	Event details URL
`EVENT_ID_OPT_ICF`	21 (0x00000015)	OPT_ICF
`EVENT_ID_OPT_LBR`	22 (0x00000016)	OPT_LBR
`EVENT_ID_OPT_REF`	23 (0x00000017)	OPT_REF
`EVENT_ID_PASS1`	24 (0x00000018)	PASS1
`EVENT_ID_PASS2`	25 (0x00000019)	PASS2
`EVENT_ID_PRE_LTCG_OPT_REF`	26 (0x0000001A)	PRE_LTCG_OPT_REF
`EVENT_ID_SYMBOL_NAME`	27 (0x0000001B)	SYMBOL_NAME
`EVENT_ID_TEMPLATE_INSTANTIATION`	28 (0x0000001C)	TEMPLATE_INSTANTIATION
`EVENT_ID_THREAD`	29 (0x0000001D)	THREAD
`EVENT_ID_TOP_DOWN`	30 (0x0000001E)	TOP_DOWN
`EVENT_ID_WHOLE_PROGRAM_ANALYSIS`	31 (0x0000001F)	WHOLE_PROGRAM_ANALYSIS

Remarks

These values are used with the C SDK functions.

FILE_DATA structure

Article • 10/29/2021

The `FILE_DATA` structure describes a file input or output.

Syntax

C++

```
typedef struct FILE_DATA_TAG
{
    const wchar_t*      Path;
    int                 TypeCode;

} FILE_DATA;
```

Members

Name	Description
`Path`	The file's absolute path
`TypeCode`	A code describing the type of the file. For more information, see FILE_TYPE_CODE.

FILE_TYPE_CODE enum

Article • 10/29/2021

The `FILE_TYPE_CODE` enum describes the type of a file.

Members

Name	Value	Description
`FILE_TYPE_CODE_OTHER`	0 (0x00000000)	A file type not listed in this enum.
`FILE_TYPE_CODE_OBJ`	1 (0x00000001)	An object (*.obj) file.
`FILE_TYPE_CODE_EXECUTABLE_IMAGE`	2 (0x00000002)	An executable (*.exe) or DLL (*.dll) file.
`FILE_TYPE_CODE_LIB`	3 (0x00000003)	A static library (*.lib) file.
`FILE_TYPE_CODE_IMP_LIB`	4 (0x00000004)	An import library (*.lib)
`FILE_TYPE_CODE_EXP`	5 (0x00000005)	An export (*.exp) file.

Remarks

FRONT_END_FILE_DATA structure

Article • 10/29/2021

The `FRONT_END_FILE_DATA` structure describes the processing of a file by the compiler front end.

Syntax

C++

```
typedef struct FRONT_END_FILE_DATA_TAG
{
    const char*         Path;

} FRONT_END_FILE_DATA;
```

Members

Name	Description
`Path`	The file's absolute path, encoded in UTF-8.

FUNCTION_DATA structure

Article • 10/29/2021

The `FUNCTION_DATA` structure describes a function.

Syntax

C++

```
typedef struct FUNCTION_DATA_TAG
{
    const char*         Name;

} FUNCTION_DATA;
```

Members

Name	Description
`Name`	The function's name, encoded in UTF-8.

FUNCTION_FORCE_INLINEE_DATA structure

Article • 10/29/2021

The `FUNCTION_FORCE_INLINEE_DATA` structure describes a force-inlined function.

Syntax

C++

```
typedef struct FUNCTION_FORCE_INLINEE_DATA_TAG
{
    const char*         Name;
    unsigned short      Size;

} FUNCTION_FORCE_INLINEE_DATA;
```

Members

Name	Description
`Name`	The function's name, encoded in UTF-8.
`Size`	The function's size, as a number of intermediate instructions.

INVOCATION_DATA structure

Article • 10/29/2021

The `INVOCATION_DATA` structure describes a compiler or linker invocation.

Syntax

C++

```
typedef struct INVOCATION_DATA_TAG
{
    int                         MSVCToolCode;

    INVOCATION_VERSION_DATA     ToolVersion;

    const char*                 ToolVersionString;
    const wchar_t*              WorkingDirectory;
    const wchar_t*              ToolPath;

} INVOCATION_DATA;
```

Members

Name	Description
`MSVCToolCode`	A code that identifies the invocation's type. For more information, see MSVC_TOOL_CODE.
`ToolVersion`	An object that stores the invoked tool's version as a group of integral values.
`ToolVersionString`	Describes the invoked tool's version in text form.
`WorkingDirectory`	The directory from which the invocation was made.
`ToolPath`	The invoked tool's absolute path.

INVOCATION_VERSION_DATA structure

Article • 10/29/2021

The `INVOCATION_VERSION_DATA` structure describes a version number as a group of integral values.

Syntax

C++

```
typedef struct INVOCATION_VERSION_DATA_TAG
{
    unsigned short VersionMajor;
    unsigned short VersionMinor;
    unsigned short BuildNumberMajor;
    unsigned short BuildNumberMinor;

} INVOCATION_VERSION_DATA;
```

Members

Name	Description
`VersionMajor`	The version's major number.
`VersionMinor`	The version's minor number.
`BuildNumberMajor`	The build's major number.
`BuildNumberMinor`	The build's minor number.

MSVC_TOOL_CODE enum

Article • 10/29/2021

The `MSVC_TOOL_CODE` enum.

Members

Name	Value	Description
`MSVC_TOOL_CODE_CL`	0 (0x00000000)	The compiler (cl.exe).
`MSVC_TOOL_CODE_LINK`	1 (0x00000001)	The linker (link.exe).

Remarks

Used by the C SDK functions.

NAME_VALUE_PAIR_DATA structure

Article • 10/29/2021

The `NAME_VALUE_PAIR_DATA` structure describes a name and value pair.

Syntax

C++

```
typedef struct NAME_VALUE_PAIR_DATA_TAG
{
    const wchar_t*      Name;
    const wchar_t*      Value;
} NAME_VALUE_PAIR_DATA;
```

Members

Name	Description
`Name`	The name.
`Value`	The value.

SYMBOL_NAME_DATA structure

Article • 10/29/2021

The `SYMBOL_NAME_DATA` structure describes a compiler front-end symbol.

Syntax

```
C++
```

```
typedef struct SYMBOL_NAME_DATA_TAG
{
    unsigned long long  Key;
    const char*         Name;

} SYMBOL_NAME_DATA;
```

Members

Name	Description
`Key`	The symbol's key. This value is unique within the trace being analyzed.
`Name`	The symbol's name.

Remarks

Symbols coming from two different compiler front-end passes may have the same name but a different key. In this case, use symbol names to determine if two types are the same.

TEMPLATE_INSTANTIATION_DATA structure

Article • 10/29/2021

The `TEMPLATE_INSTANTIATION_DATA` structure describes a template instantiation.

Syntax

```
C++
```

```
typedef struct TEMPLATE_INSTANTIATION_DATA_TAG
{
    unsigned long long  SpecializationSymbolKey;
    unsigned long long  PrimaryTemplateSymbolKey;
    int                 KindCode;

} TEMPLATE_INSTANTIATION_DATA;
```

Members

Name	Description
`SpecializationSymbolKey`	The key for the template specialization's type. This value is unique within the trace being analyzed.
`PrimaryTemplateSymbolKey`	The key for the primary template type that was specialized. This value is unique within the trace being analyzed.
`KindCode`	The template instantiation's type. For more information, see TEMPLATE_INSTANTIATION_KIND_CODE.

Remarks

The keys in the `TEMPLATE_INSTANTIATION_DATA` structure are unique within the trace being analyzed. However, two different keys coming from different compiler front-end passes may point to two identical types. When consuming `TEMPLATE_INSTANTIATION_DATA` information from multiple compiler front-end passes, use the SYMBOL_NAME events to determine if two types are the same. `SymbolName` events are emitted at the end of a compiler front-end pass, after all template instantiations have taken place.

TEMPLATE_INSTANTIATION_KIND_CODE enum

Article • 10/29/2021

The `TEMPLATE_INSTANTIATION_KIND_CODE` enum.

Members

Name	Value	Description
`TEMPLATE_INSTANTIATION_KIND_CODE_CLASS`	0 (0x00000000)	A class template instantiation.
`TEMPLATE_INSTANTIATION_KIND_CODE_FUNCTION`	1 (0x00000001)	A function template instantiation.
`TEMPLATE_INSTANTIATION_KIND_CODE_VARIABLE`	2 (0x00000002)	A constexpr variable instantiation.
`TEMPLATE_INSTANTIATION_KIND_CODE_CONCEPT`	3 (0x00000003)	A concept template instantiation.

Remarks

TRACE_INFO_DATA structure

Article • 10/29/2021

The `TRACE_INFO_DATA` structure describes a trace being analyzed or relogged.

Syntax

C++

```
typedef struct TRACE_INFO_DATA_TAG
{
    unsigned long           LogicalProcessorCount;
    long long               TickFrequency;
    long long               StartTimestamp;
    long long               StopTimestamp;

} TRACE_INFO_DATA;
```

Members

Name	Description
`LogicalProcessorCount`	The number of logical processors on the machine where the trace was collected.
`TickFrequency`	The number of ticks per second to use when evaluating a duration measured in ticks.
`StartTimestamp`	This field is set to a tick value captured at the time the trace was started.
`StopTimestamp`	This field is set to a tick value captured at the time the trace was stopped.

Remarks

Subtract `StartTimestamp` from `StopTimestamp` to obtain the number of ticks elapsed during the whole trace. Use `TickFrequency` to convert the resulting value into a time unit. For an example that converts ticks to time units, see EVENT_DATA.

TRANSLATION_UNIT_PASS_CODE enum

Article • 10/29/2021

The `TRANSLATION_UNIT_PASS_CODE` enum.

Members

Name	Value	Description
`TRANSLATION_UNIT_PASS_CODE_FRONT_END`	0 (0x00000000)	The front-end pass, responsible for parsing source code and converting it into intermediate language.
`TRANSLATION_UNIT_PASS_CODE_BACK_END`	1 (0x00000001)	The back-end pass, responsible for optimizing intermediate language and converting it into machine code.

Remarks

Used by the C SDK functions.

TRANSLATION_UNIT_TYPE enum

Article • 10/29/2021

The `TRANSLATION_UNIT_TYPE` enum.

Members

Name	Value	Description
`TRANSLATION_UNIT_TYPE_MODULE`	0 (0x00000000)	The type of this translation unit is a module interface.
`TRANSLATION_UNIT_TYPE_HEADER_UNIT`	1 (0x00000001)	The type of this translation unit is a header unit.
`TRANSLATION_UNIT_TYPE_PRECOMPILED_HEADER`	2 (0x00000002)	The type of this translation unit is a precompiled header (PCH).

Remarks

Used by the C SDK functions.

TRANSLATION_UNIT_TYPE_DATA enum

Article • 10/29/2021

The `TRANSLATION_UNIT_TYPE_DATA` structure describes the type of the translation unit in the compiler front end pass.

Syntax

```
C++
```

```
typedef struct TRANSLATION_UNIT_TYPE_DATA_TAG
{
    int  TranslationUnitType;

}  TRANSLATION_UNIT_TYPE_DATA;
```

Members

Name	Description
`TranslationUnitType`	The type of this translation unit (either modules, header unit, or precompiled header).

Remarks

Used by the C SDK functions.

ANALYSIS_CALLBACKS structure

Article • 10/29/2021

The `ANALYSIS_CALLBACKS` structure is used when initializing an ANALYSIS_DESCRIPTOR or RELOG_DESCRIPTOR object. It specifies which functions to call during the analysis or relogging of an Event Tracing for Windows (ETW) trace.

Syntax

C++

```
typedef struct ANALYSIS_CALLBACKS_TAG
{
    OnAnalysisEventFunc     OnStartActivity;
    OnAnalysisEventFunc     OnStopActivity;
    OnAnalysisEventFunc     OnSimpleEvent;
    OnTraceInfoFunc         OnTraceInfo;
    OnBeginEndPassFunc      OnBeginAnalysis;
    OnBeginEndPassFunc      OnEndAnalysis;
    OnBeginEndPassFunc      OnBeginAnalysisPass;
    OnBeginEndPassFunc      OnEndAnalysisPass;
} ANALYSIS_CALLBACKS;
```

Members

Name	Description
`OnStartActivity`	Called to process an activity start event.
`OnStopActivity`	Called to process an activity stop event.
`OnSimpleEvent`	Called to process a simple event.
`OnTraceInfo`	For analysis sessions, called at the beginning of each analysis pass. For relogging sessions, called at the beginning of each analysis pass, and again at the beginning of the relogging pass. This function is only called after OnBeginAnalysisPass has been called.
`OnBeginAnalysis`	For analysis sessions, called before any analysis pass has begun. For relogging sessions, called twice before the analysis phase has begun: once to announce the start of the relogging session, and once more to announce the beginning of the analysis phase.

Name	Description
`OnEndAnalysis`	For analysis sessions, this function is called after all analysis passes have ended. For relogging sessions, this function is called when all analysis passes of the analysis phase have ended. Then, it's called again after the relogging pass has ended.
`OnBeginAnalysisPass`	Called when beginning an analysis pass or the relogging pass, before processing any event.
`OnEndAnalysisPass`	Called when ending an analysis pass or the relogging pass, after processing all events.

Remarks

The analysis phase of a relogging session is considered part of the relogging session, and may contain multiple analysis passes. For this reason, `OnBeginAnalysis` is called twice in a row at the beginning of a relogging session. `OnEndAnalysis` is called at the end of the analysis phase, before starting the relogging phase, and once again at the end of the relogging phase. The relogging phase always contains a single relogging pass.

It's possible for analyzers to be part of both the analysis and the relogging phase of a relogging session. These analyzers can determine which phase is currently ongoing by keeping track of the OnBeginAnalysis and `OnEndAnalysis` call pairs. Two `OnBeginAnalysis` calls without any `OnEndAnalysis` call means the analysis phase is ongoing. Two `OnBeginAnalysis` calls and one `OnEndAnalysis` call means the relogging phase is ongoing. Two OnBeginAnalysis and two `OnEndAnalysis` calls means both phases have ended.

All members of the `ANALYSIS_CALLBACKS` structure must point to a valid function. For more information on the accepted function signatures, see OnAnalysisEventFunc, OnTraceInfoFunc, and OnBeginEndPassFunc.

AnalysisControl enum class

Article • 10/29/2021

The `AnalysisControl` enum class is used control the flow of an analysis or relogging session. Return an `AnalysisControl` code from an IAnalyzer or IRelogger member function to control what should happen next.

Members

Name	Description
`BLOCK`	Prevents the current event from propagating further in the analyzer or relogger group.
`CANCEL`	Cancel the current analysis or relogging session.
`CONTINUE`	Continue the current analysis or relogging session normally. Propagate the current event to the next analyzer or relogger group member.
`FAILURE`	Cancel the current analysis or relogging session and signal an error.

ANALYSIS_DESCRIPTOR structure

Article • 10/29/2021

The `ANALYSIS_DESCRIPTOR` structure is used with the AnalyzeA and AnalyzeW functions. It describes how an Event Tracing for Windows (ETW) trace should be analyzed.

Syntax

C++

```
typedef struct ANALYSIS_DESCRIPTOR_TAG
{
    unsigned                NumberOfPasses;
    ANALYSIS_CALLBACKS      Callbacks;
    void*                   Context;
} ANALYSIS_DESCRIPTOR;
```

Members

Name	Description
`NumberOfPasses`	The number of analysis passes that should be done over the ETW trace.
`Callbacks`	An ANALYSIS_CALLBACKS object that specifies which functions to call during the analysis session.
`Context`	A user-provided context that is passed as an argument to all callback functions specified in `Callbacks`

Remarks

The `Callbacks` structure only accepts pointers to non-member functions. You can get around this limitation by setting `Context` to an object pointer. This object pointer will be passed as an argument to all your non-member callback functions. Use this pointer to call member functions from within your non-member callback functions.

CALLBACK_CODE enum

Article • 10/29/2021

The `CALLBACK_CODE` enum is used control the flow of an analysis or relogging session. Return a CALLBACK_CODE value from the functions in ANALYSIS_CALLBACKS or RELOG_CALLBACKS to control what should happen next.

Members

Name	Value	Description
`CALLBACK_CODE_ANALYSIS_SUCCESS`	1 (0x00000001)	Continue the current analysis or relogging session normally.
`CALLBACK_CODE_ANALYSIS_FAILURE`	2 (0x00000002)	Cancel the current analysis or relogging session and signal an error.
`CALLBACK_CODE_ANALYSIS_CANCEL`	4 (0x00000004)	Cancel the current analysis or relogging session.

IAnalyzer class

Article • 10/29/2021

The `IAnalyzer` class provides an interface for analyzing an Event Tracing for Windows (ETW) trace. It's used with the MakeDynamicAnalyzerGroup, MakeDynamicReloggerGroup, MakeStaticAnalyzerGroup, and MakeStaticReloggerGroup functions. Use `IAnalyzer` as a base class to create your own analyzer that can be part of an analyzer or relogger group.

Syntax

C++

```
class IAnalyzer : public IRelogger
{
public:
    virtual AnalysisControl OnStartActivity(const EventStack& eventStack);
    virtual AnalysisControl OnStopActivity(const EventStack& eventStack)
    virtual AnalysisControl OnSimpleEvent(const EventStack& eventStack);
    virtual AnalysisControl OnBeginAnalysis();
    virtual AnalysisControl OnEndAnalysis();
    virtual AnalysisControl OnBeginAnalysisPass();
    virtual AnalysisControl OnEndAnalysisPass();

    AnalysisControl OnStartActivity(const EventStack& eventStack,
        const void* relogSession) final;

    AnalysisControl OnStopActivity(const EventStack& eventStack,
        const void* relogSession) final;

    AnalysisControl OnSimpleEvent(const EventStack& eventStack,
        const void* relogSession) final;

    AnalysisControl OnBeginRelogging() final;
    AnalysisControl OnEndRelogging() final;
    AnalysisControl OnBeginReloggingPass() final;
    AnalysisControl OnEndReloggingPass() final;

    virtual ~IAnalyzer();
};
```

Remarks

Classes that derive from `IAnalyzer` can be used as both analyzers and reloggers. When used as reloggers, the relogger-specific functions redirect to their analyzer equivalent.

The reverse isn't true: a class that derives from `IRelogger` can't be used as an analyzer. Using an analyzer in a relogger group is a common pattern. When placed in an early position of a relogger group, an analyzer can pre-compute information and make it available for reloggers in later positions.

The default return value for all functions that aren't overridden is `AnalysisControl::CONTINUE`. For more information, see AnalysisControl.

Members

In addition to the OnTraceInfo member from the `IRelogger` interface, the `IAnalyzer` class contains the following members:

Destructor

~IAnalyzer

Functions

OnBeginAnalysis
OnBeginAnalysisPass
OnBeginRelogging
OnBeginReloggingPass
OnEndAnalysis
OnEndAnalysisPass
OnEndRelogging
OnEndReloggingPass
OnSimpleEvent
OnStartActivity
OnStopActivity

~IAnalyzer

Destroys the IAnalyzer class.

C++

```
virtual ~IAnalyzer();
```

OnBeginAnalysis

For analyzers part of an analyzer group, this function is called before the first analysis pass begins. For analyzers part of a relogger group, this function is called before the relogging pass begins. For analyzers part of both the analyzer and relogger group of the same relogging session, this function is called twice before the first analysis pass begins.

```
C++

virtual AnalysisControl OnBeginAnalysis();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnBeginAnalysisPass

For analyzers part of an analyzer group, this function is called at the beginning of every analysis pass. For analyzers part of a relogger group, this function is called at the beginning of the relogger pass. For analyzers part of both the analyzer and relogger group of the same relogging session, this function is called at the beginning of every analysis pass, and at the beginning of the relogger pass.

```
C++

virtual AnalysisControl OnBeginAnalysisPass();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnBeginRelogging

```
C++

AnalysisControl OnBeginRelogging() final;
```

This function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. This function redirects the call to OnBeginAnalysis.

Return Value

The result of the OnBeginAnalysis call.

OnBeginReloggingPass

This function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. This function redirects the call to OnBeginAnalysisPass.

```
C++

AnalysisControl OnBeginReloggingPass() final;
```

Return Value

The result of the OnBeginAnalysisPass call.

OnEndAnalysis

For analyzers that are part of an analyzer group, this function is called after the last analysis pass has ended. For analyzers that are part of a relogger group, this function is called after the relogging pass has ended. For analyzers that are part of both the analyzer and relogger group of the same relogging session, this function is called twice:

1. after all analysis passes have ended and before the relogging pass begins, and
2. after the relogging pass has ended.

```
C++

virtual AnalysisControl OnEndAnalysis();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnEndAnalysisPass

For analyzers part of an analyzer group, this function is called at the end of every analysis pass. For analyzers part of a relogger group, this function is called at the end of

the relogger pass. For analyzers part of both the analyzer and relogger group of the same relogging session, this function is called at the end of every analysis pass, and at the end of the relogger pass.

```
C++

virtual AnalysisControl OnEndAnalysisPass();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnEndRelogging

This function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. This function redirects the call to OnEndAnalysis.

```
C++

AnalysisControl OnEndRelogging() final;
```

Return Value

The result of the OnEndAnalysis call.

OnEndReloggingPass

This function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. This function redirects the call to OnEndAnalysisPass.

```
C++

AnalysisControl OnEndReloggingPass() final;
```

Return Value

The result of the OnEndAnalysisPass call.

OnSimpleEvent

This function is called when a simple event is being processed. The second version of this function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. All calls to version 2 are redirected to version 1.

Version 1

```
C++

virtual AnalysisControl OnSimpleEvent(const EventStack& eventStack);
```

Version 2

```
C++

AnalysisControl OnSimpleEvent(const EventStack& eventStack,
        const void* relogSession) final;
```

Parameters

eventStack
The event stack for this simple event. For more information on event stacks, see Events.

relogSession
This parameter is unused.

Return Value

An AnalysisControl code that describes what should happen next.

OnStartActivity

This function is called when an activity start event is being processed. The second version of this function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. All calls to version 2 are redirected to version 1.

Version 1

C++

```
virtual AnalysisControl OnStartActivity(const EventStack& eventStack);
```

Version 2

C++

```
AnalysisControl OnStartActivity(const EventStack& eventStack,
        const void* relogSession) final;
```

Parameters

eventStack
The event stack for this activity start event. For more information on event stacks, see Events.

relogSession
This parameter is unused.

Return Value

An AnalysisControl code that describes what should happen next.

OnStopActivity

This function is called when an activity stop event is being processed. The second version of this function can't be overridden. It's called by the C++ Build Insights SDK when an analyzer is part of a relogger group. All calls to version 2 are redirected to version 1.

Version 1

C++

```
virtual AnalysisControl OnStopActivity(const EventStack& eventStack);
```

Version 2

C++

```
```

C++

```
AnalysisControl OnStopActivity(const EventStack& eventStack,
        const void* relogSession) final;
```

Parameters

eventStack

The event stack for this activity stop event. For more information on event stacks, see Events.

relogSession

This parameter is unused.

Return Value

An AnalysisControl code that describes what should happen next.

IRelogger class

Article • 10/29/2021

The `IRelogger` class provides an interface for relogging an Event Tracing for Windows (ETW) trace. It's used with the MakeDynamicReloggerGroup and MakeStaticReloggerGroup functions. Use `IRelogger` as a base class to create your own relogger that can be part of a relogger group.

Syntax

C++

```
class IRelogger
{
public:
    virtual AnalysisControl OnStartActivity(const EventStack& eventStack,
        const void* relogSession);

    virtual AnalysisControl OnStopActivity(const EventStack& eventStack,
        const void* relogSession);

    virtual AnalysisControl OnSimpleEvent(const EventStack& eventStack,
        const void* relogSession);

    virtual AnalysisControl OnTraceInfo(const TraceInfo& traceInfo);
    virtual AnalysisControl OnBeginRelogging();
    virtual AnalysisControl OnEndRelogging();
    virtual AnalysisControl OnBeginReloggingPass();
    virtual AnalysisControl OnEndReloggingPass() ;

    virtual ~IRelogger();
};
```

Remarks

The default return value for all functions that aren't overridden is `AnalysisControl::CONTINUE`. For more information, see AnalysisControl.

Members

Destructor

Functions

~IRelogger

Destroys the IRelogger class.

```
C++

virtual ~IRelogger();
```

OnBeginRelogging

This function is called before the relogging pass begins.

```
C++

virtual AnalysisControl OnBeginRelogging();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnBeginReloggingPass

This function is called at the beginning of the relogging pass.

```
C++

virtual AnalysisControl OnBeginReloggingPass();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnEndRelogging

This function is called after the relogging pass has ended.

```
C++

virtual AnalysisControl OnEndRelogging();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnEndReloggingPass

This function is called at the end of the relogging pass.

```
C++

virtual AnalysisControl OnEndReloggingPass();
```

Return Value

An AnalysisControl code that describes what should happen next.

OnSimpleEvent

```
C++

virtual AnalysisControl OnSimpleEvent(const EventStack& eventStack);
```

This function is called when a simple event is being processed.

Parameters

eventStack
The event stack for this simple event. For more information on event stacks, see Events.

Return Value

An AnalysisControl code that describes what should happen next.

OnStartActivity

```
C++
```

```
virtual AnalysisControl OnStartActivity(const EventStack& eventStack);
```

This function is called when an activity start event is being processed.

Parameters

eventStack
The event stack for this activity start event. For more information on event stacks, see Events.

Return Value

An AnalysisControl code that describes what should happen next.

OnStopActivity

This function is called when an activity stop event is being processed.

```
C++
```

```
virtual AnalysisControl OnStopActivity(const EventStack& eventStack);
```

Parameters

eventStack
The event stack for this activity stop event. For more information on event stacks, see Events.

Return Value

An AnalysisControl code that describes what should happen next.

OnTraceInfo

```
C++
```

```
virtual AnalysisControl OnTraceInfo(const TraceInfo& traceInfo);
```

This function is called once at the beginning of every analysis or relogging pass.

Parameters

traceInfo
A TraceInfo object that contains useful properties about the trace being consumed.

Return Value

An AnalysisControl code that describes what should happen next.

OnAnalysisEventFunc typedef

Article • 10/29/2021

The `OnAnalysisEventFunc` typedef is one of the function signatures used in the ANALYSIS_CALLBACKS structure.

Syntax

C++

```
typedef enum CALLBACK_CODE (BUILD_INSIGHTS_API *OnAnalysisEventFunc)(
    const EVENT_COLLECTION_DATA*    eventStack,
    void*                           callbackContext);
```

Parameters

eventStack

The event stack for the current event. For more information on event stacks, see Events.

callbackContext

The context value that was set for this callback in ANALYSIS_DESCRIPTOR or RELOG_DESCRIPTOR.

Return Value

A CALLBACK_CODE value that controls what should happen next.

OnBeginEndPassFunc typedef

Article • 10/29/2021

The `OnBeginEndPassFunc` typedef is one of the function signatures used in the ANALYSIS_CALLBACKS and RELOG_CALLBACKS structures.

Syntax

C++

```
typedef enum CALLBACK_CODE (BUILD_INSIGHTS_API *OnBeginEndPassFunc)(
    void*                           callbackContext);
```

Members

Name	Description
`callbackContext`	

OnRelogEventFunc typedef

Article • 10/29/2021

The `OnRelogEventFunc` typedef is one of the function signatures used in the RELOG_CALLBACKS structure.

Syntax

C++

```
typedef enum CALLBACK_CODE (BUILD_INSIGHTS_API *OnRelogEventFunc)(
    const EVENT_COLLECTION_DATA*    eventStack,
    const void*                     relogSession,
    void*                           callbackContext);
```

Parameters

eventStack

The event stack for the current event. For more information on event stacks, see Events.

relogSession

The relogging session pointer to use when calling the InjectEvent.

callbackContext

The context value that was set for this callback in RELOG_DESCRIPTOR.

Return Value

A CALLBACK_CODE value that controls what should happen next.

OnTraceInfoFunc typedef

Article • 10/29/2021

The `OnTraceInfoFunc` typedef is one of the function signatures used in the ANALYSIS_CALLBACKS and RELOG_CALLBACKS structures.

Syntax

C++

```
typedef enum CALLBACK_CODE (BUILD_INSIGHTS_API *OnTraceInfoFunc)(
    const TRACE_INFO_DATA*          traceInfo,
    void*                           callbackContext);
```

Parameters

traceInfo
A TRACE_INFO_DATA object that contains information about the trace currently being analyzed or relogged.

callbackContext
The context value that was set for this callback in ANALYSIS_DESCRIPTOR or RELOG_DESCRIPTOR.

Return Value

A CALLBACK_CODE value that controls what should happen next.

RELOG_CALLBACKS structure

Article • 10/29/2021

The `RELOG_CALLBACKS` structure is used when initializing a RELOG_DESCRIPTOR object. It specifies which functions to call during the relogging of an Event Tracing for Windows (ETW) trace.

Syntax

C++

```
typedef struct RELOG_CALLBACKS_TAG
{
    OnRelogEventFunc        OnStartActivity;
    OnRelogEventFunc        OnStopActivity;
    OnRelogEventFunc        OnSimpleEvent;
    OnTraceInfoFunc         OnTraceInfo;
    OnBeginEndPassFunc      OnBeginRelogging;
    OnBeginEndPassFunc      OnEndRelogging;
    OnBeginEndPassFunc      OnBeginReloggingPass;
    OnBeginEndPassFunc      OnEndReloggingPass;
} RELOG_CALLBACKS;
```

Members

Name	Description
`OnStartActivity`	Called to process an activity start event.
`OnStopActivity`	Called to process an activity stop event.
`OnSimpleEvent`	Called to process a simple event.
`OnTraceInfo`	Called once at the beginning of the relogging pass, after `OnBeginReloggingPass` has been called.
`OnBeginRelogging`	Called when beginning a relogging session, before the relogging pass has begun.
`OnEndRelogging`	Called when ending a relogging session, after the relogging pass has ended.
`OnBeginReloggingPass`	Called when beginning the relogging pass, before processing any event.
`OnEndReloggingPass`	Called when ending the relogging pass, after processing all events.

Remarks

All members of the `RELOG_CALLBACKS` structure must point to a valid function. For more information on the accepted function signatures, see OnRelogEventFunc, OnTraceInfoFunc, and OnBeginEndPassFunc.

RELOG_DESCRIPTOR structure

Article • 10/29/2021

The `RELOG_DESCRIPTOR` structure is used with the RelogA and RelogW functions. It describes how an Event Tracing for Windows (ETW) trace should be relogged.

Syntax

C++

```
typedef struct RELOG_DESCRIPTOR_TAG
{
    unsigned                NumberOfAnalysisPasses;
    ANALYSIS_CALLBACKS      AnalysisCallbacks;
    RELOG_CALLBACKS         RelogCallbacks;
    unsigned long long      SystemEventsRetentionFlags;
    void*                   AnalysisContext;
    void*                   RelogContext;
} RELOG_DESCRIPTOR;
```

Members

Name	Description
`NumberOfAnalysisPasses`	The number of analysis passes that should be done over the ETW trace during the relogging session's analysis phase.
`AnalysisCallbacks`	An ANALYSIS_CALLBACKS object that specifies which functions to call during the relogging session's analysis phase.
`RelogCallbacks`	A RELOG_CALLBACKS object that specifies which functions to call during the relogging session's relogging phase.
`SystemEventsRetentionFlags`	A RELOG_RETENTION_SYSTEM_EVENT_FLAGS bitmask that specifies which system ETW events to keep in the relogged trace.
`AnalysisContext`	A user-provided context that's passed as an argument to all callback functions specified in `AnalysisCallbacks`
`RelogContext`	A user-provided context that's passed as an argument to all callback functions specified in `RelogCallbacks`

Remarks

The relogging of ETW events during a relogging session is controlled by the user through the callback functions specified in `RelogCallbacks`. However, system ETW events such as CPU samples aren't forwarded to these callback functions. Use the `SystemEventsRetentionFlags` field to control the relogging of system ETW events.

The `AnalysisCallbacks` and `RelogCallbacks` structures only accept pointers to non-member functions. You can get around this limitation by setting them to an object pointer. This object pointer will be passed as an argument to all your non-member callback functions. Use this pointer to call member functions from within your non-member callback functions.

The analysis phase of a relogging session is always executed before the relogging phase.

RELOG_RETENTION_SYSTEM_EVENT_FLAGS constants

Article • 10/29/2021

The `RELOG_RETENTION_SYSTEM_EVENT_FLAGS` constants are used to describe which system events to keep in a relogged trace. Use them to initialize the RELOG_DESCRIPTOR structure's `SystemEventsRetentionFlags` field.

Syntax

```
C++
```

```
static const unsigned long long
    RELOG_RETENTION_SYSTEM_EVENT_FLAGS_CPU_SAMPLES = 0x1ULL;

static const unsigned long long
    RELOG_RETENTION_SYSTEM_EVENT_FLAGS_ALL         = 0xFFFFFFFFFFFFFFFFULL;
```

Members

Name	Description
`RELOG_RETENTION_SYSTEM_EVENT_FLAGS_CPU_SAMPLES`	Keep CPU sample system events in a relogged trace.
`RELOG_RETENTION_SYSTEM_EVENT_FLAGS_ALL`	Keep all system events in a relogged trace.

Remarks

RESULT_CODE enum

Article • 10/29/2021

The `RESULT_CODE` enum describes success and failure conditions.

Members

Name	Value	Description
`RESULT_CODE_SUCCESS`	0 (0x00000000)	The operation was successful.
`RESULT_CODE_FAILURE_ANALYSIS_ERROR`	1 (0x00000001)	One of your callback functions in ANALYSIS_DESCRIPTOR or RELOG_DESCRIPTOR returned the `CALLBACK_CODE_ANALYSIS_FAILURE` value. This value is a member of the CALLBACK_CODE enum.
`RESULT_CODE_FAILURE_CANCELLED`	2 (0x00000002)	One of your callback functions in ANALYSIS_DESCRIPTOR or RELOG_DESCRIPTOR returned the `CALLBACK_CODE_ANALYSIS_CANCEL` value. This value is a member of the CALLBACK_CODE enum.
`RESULT_CODE_FAILURE_INVALID_INPUT_LOG_FILE`	3 (0x00000003)	The input Event Tracing for Windows (ETW) trace specified is invalid.
`RESULT_CODE_FAILURE_INVALID_OUTPUT_LOG_FILE`	4 (0x00000004)	The output ETW trace specified is invalid.
`RESULT_CODE_FAILURE_MISSING_ANALYSIS_CALLBACK`	5 (0x00000005)	The ANALYSIS_CALLBACKS structure was not initialized correctly.
`RESULT_CODE_FAILURE_MISSING_RELOG_CALLBACK`	6 (0x00000006)	The RELOG_CALLBACKS structure was not initialized correctly.
`RESULT_CODE_FAILURE_OPEN_INPUT_TRACE`	7 (0x00000007)	Failed to open the input ETW trace.

Name	Value	Description
`RESULT_CODE_FAILURE_PROCESS_TRACE`	8 (0x00000008)	An error occurred while processing the input ETW trace.
`RESULT_CODE_FAILURE_START_RELOGGER`	9 (0x00000009)	An error occurred when trying to start the relogging session.
`RESULT_CODE_FAILURE_DROPPED_EVENTS`	10 (0x0000000A)	The input ETW trace is missing important events.
`RESULT_CODE_FAILURE_UNSUPPORTED_OS`	11 (0x0000000B)	You are using C++ Build Insights on an unsupported version of Windows.
`RESULT_CODE_FAILURE_INVALID_TRACING_SESSION_NAME`	12 (0x0000000C)	The provided session name is invalid.
`RESULT_CODE_FAILURE_INSUFFICIENT_PRIVILEGES`	13 (0x0000000D)	This operation requires administrator privileges.
`RESULT_CODE_FAILURE_GENERATE_GUID`	14 (0x0000000E)	An error occurred while generating a GUID.
`RESULT_CODE_FAILURE_OBTAINING_TEMP_DIRECTORY`	15 (0x0000000F)	An error occurred while trying to determine the temporary directory path.
`RESULT_CODE_FAILURE_CREATE_TEMPORARY_DIRECTORY`	16 (0x00000010)	An error occurred while trying to create a temporary directory for the tracing session being started.
`RESULT_CODE_FAILURE_START_SYSTEM_TRACE`	17 (0x00000011)	An error occurred when trying to start the system trace.
`RESULT_CODE_FAILURE_START_MSVC_TRACE`	18 (0x00000012)	An error occurred when trying to start the MSVC trace.
`RESULT_CODE_FAILURE_STOP_MSVC_TRACE`	19 (0x00000013)	An error occurred when trying to stop the MSVC trace.
`RESULT_CODE_FAILURE_STOP_SYSTEM_TRACE`	20 (0x00000014)	An error occurred when trying to start the system trace.
`RESULT_CODE_FAILURE_SESSION_DIRECTORY_RESOLUTION`	21 (0x00000015)	A trace was stopped but the tracing session's temporary directory cannot be found.
`RESULT_CODE_FAILURE_MSVC_TRACE_FILE_NOT_FOUND`	22 (0x00000016)	The trace file for the MSVC trace being stopped cannot be found.

Name	Value	Description
`RESULT_CODE_FAILURE_MERGE_TRACES`	23 (0x00000017)	An error occurred when merging traces using Kernel Trace Control.
`RESULT_CODE_FAILURE_UNKNOWN_ERROR`	24 (0x00000018)	An unknown error occurred.

TRACING_SESSION_MSVC_EVENT_FLAGS constants

Article • 10/29/2021

The `TRACING_SESSION_MSVC_EVENT_FLAGS` constants are used to describe which MSVC events to collect during a trace. Use them to initialize the TRACING_SESSION_OPTIONS structure's `MsvcEventFlags` field.

Syntax

C++

```
static const unsigned long long
    TRACING_SESSION_MSVC_EVENT_FLAGS_BASIC                                =
0x0001ULL;

static const unsigned long long
    TRACING_SESSION_MSVC_EVENT_FLAGS_FRONTEND_FILES                       =
0x0004ULL;

static const unsigned long long
    TRACING_SESSION_MSVC_EVENT_FLAGS_FRONTEND_TEMPLATE_INSTANTIATIONS     =
0x0008ULL;

static const unsigned long long
    TRACING_SESSION_MSVC_EVENT_FLAGS_BACKEND_FUNCTIONS                    =
0x1000ULL;

static const unsigned long long
    TRACING_SESSION_MSVC_EVENT_FLAGS_ALL                                  =
0xFFFFFFFFFFFFFFFFULL;
```

Members

Name	Events turned on by this flag

Name	Events turned on by this flag
`TRACING_SESSION_MSVC_EVENT_FLAGS_BASIC`	This flag is associated with the following events. It's activated by default by the C++ Build Insights SDK even if not specified explicitly. You can't disable these events. BACK_END_PASSBOTTOM_UP C1_DLL C2_DLL CODE_GENERATION COMMAND_LINE COMPILER ENVIRONMENT_VARIABLE EXECUTABLE_IMAGE_OUTPUT EXP_OUTPUT FILE_INPUT FRONT_END_PASS FRONT_END_PASS IMP_LIB_OUTPUT LIB_OUTPUT LINKER LTCG OBJ_OUTPUT OPT_ICF OPT_LBR OPT_REF PASS1 PASS2 PRE_LTCG_OPT_REF THREAD TOP_DOWN WHOLE_PROGRAM_ANALYSIS
`TRACING_SESSION_MSVC_EVENT_FLAGS_FRONTEND_FILES`	FRONT_END_FILE
`TRACING_SESSION_MSVC_EVENT_FLAGS_FRONTEND_TEMPLATE_INSTANTIATIONS`	SYMBOL_NAME TEMPLATE_INSTANTIATION
`TRACING_SESSION_MSVC_EVENT_FLAGS_BACKEND_FUNCTIONS`	FORCE_INLINEE FUNCTION
`TRACING_SESSION_MSVC_EVENT_FLAGS_ALL`	This flag turns on all events.

TRACING_SESSION_OPTIONS structure

Article • 10/29/2021

The `TRACING_SESSION_OPTIONS` structure is used when initializing an ANALYSIS_DESCRIPTOR or RELOG_DESCRIPTOR structure. It describes which events to capture during the collection of a trace.

Syntax

C++

```
typedef struct TRACING_SESSION_OPTIONS_TAG
{
    unsigned long long SystemEventFlags;
    unsigned long long MsvcEventFlags;

} TRACING_SESSION_OPTIONS;
```

Members

Name	Description
`SystemEventFlags`	A bitmask describing the system events to capture. For more information, see TRACING_SESSION_SYSTEM_EVENT_FLAGS.
`MsvcEventFlags`	A bitmask describing the MSVC events to capture. For more information, see TRACING_SESSION_MSVC_EVENT_FLAGS.

TRACING_SESSION_STATISTICS structure

Article • 10/29/2021

The `TRACING_SESSION_STATISTICS` structure describes statistics on a trace that was collected. Its fields are set when stopping a tracing session.

Syntax

C++

```
typedef struct TRACING_SESSION_STATISTICS_TAG
{
    unsigned long MSVCEventsLost;
    unsigned long MSVCBuffersLost;
    unsigned long SystemEventsLost;
    unsigned long SystemBuffersLost;

} TRACING_SESSION_STATISTICS;
```

Members

Name	Description
`MSVCEventsLost`	The number of MSVC events that were dropped.
`MSVCBuffersLost`	The number of MSVC event buffers that were dropped.
`SystemEventsLost`	The number of system events that were dropped.
`SystemBuffersLost`	The number of system event buffers that were dropped.

Remarks

This structure is populated when calling the following functions:

- StopTracingSession
- StopTracingSessionA
- StopTracingSessionW
- StopAndAnalyzeTracingSession
- StopAndAnalyzeTracingSessionA
- StopAndAnalyzeTracingSessionW
- StopAndRelogTracingSession

- StopAndRelogTracingSessionA
- StopAndRelogTracingSessionW

TRACING_SESSION_SYSTEM_EVENT_FLAGS constants

Article • 10/29/2021

The `TRACING_SESSION_SYSTEM_EVENT_FLAGS` constants are used to describe which system events to collect during a trace. Use them to initialize the TRACING_SESSION_OPTIONS structure's `SystemEventFlags` field.

Syntax

C++

```
static const unsigned long long
    TRACING_SESSION_SYSTEM_EVENT_FLAGS_CONTEXT      = 0x1ULL;

static const unsigned long long
    TRACING_SESSION_SYSTEM_EVENT_FLAGS_CPU_SAMPLES  = 0x2ULL;

static const unsigned long long
    TRACING_SESSION_SYSTEM_EVENT_FLAGS_ALL          = 0xFFFFFFFFFFFFFFFFULL;
```

Members

Name	Events turned on by this flag
`TRACING_SESSION_SYSTEM_EVENT_FLAGS_CONTEXT`	This flag is activated by default by the C++ Build Insights SDK even if not specified explicitly. It enables basic system events that are required by C++ Build Insights to function properly. The events enabled by this flag provide information about processes, threads, and image loading. You can't disable these events.
`TRACING_SESSION_SYSTEM_EVENT_FLAGS_CPU_SAMPLES`	CPU samples
`TRACING_SESSION_SYSTEM_EVENT_FLAGS_ALL`	This flag turns on all system events.