# C/C++ Sanitizers | Instrument code for runtime bug detection

Use C and C++ sanitizers for defect reporting, analysis, and prevention.

## Find bugs using code sanitizers

OVERVIEW

Learn about AddressSanitizer

TUTORIAL

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

## Instrument your builds

REFERENCE

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer error examples

AddressSanitizer known issues

## Debug your results

REFERENCE

AddressSanitizer debugger integration

# AddressSanitizer

Article • 09/06/2024

## Overview

The C & C++ languages are powerful, but can suffer from a class of bugs that affect program correctness and program security. Starting in Visual Studio 2019 version 16.9, the Microsoft C/C++ compiler (MSVC) and IDE supports the *AddressSanitizer* sanitizer. AddressSanitizer (ASan) is a compiler and runtime technology that exposes many hard-to-find bugs with **zero** false positives:

- Alloc/dealloc mismatches and new/delete type mismatches
- Allocations too large for the heap
- calloc overflow and alloca overflow
- Double free and use after free
- Global variable overflow
- Heap buffer overflow
- Invalid alignment of aligned values
- memcpy and strncat parameter overlap
- Stack buffer overflow and underflow
- Stack use after return and use after scope
- Memory use after it's poisoned

Use AddressSanitizer to reduce your time spent on:

- Basic correctness
- Cross platform portability
- Security
- Stress testing
- Integrating new code

AddressSanitizer, originally introduced by Google, provides runtime bug-finding technologies that use your existing build systems and existing test assets directly.

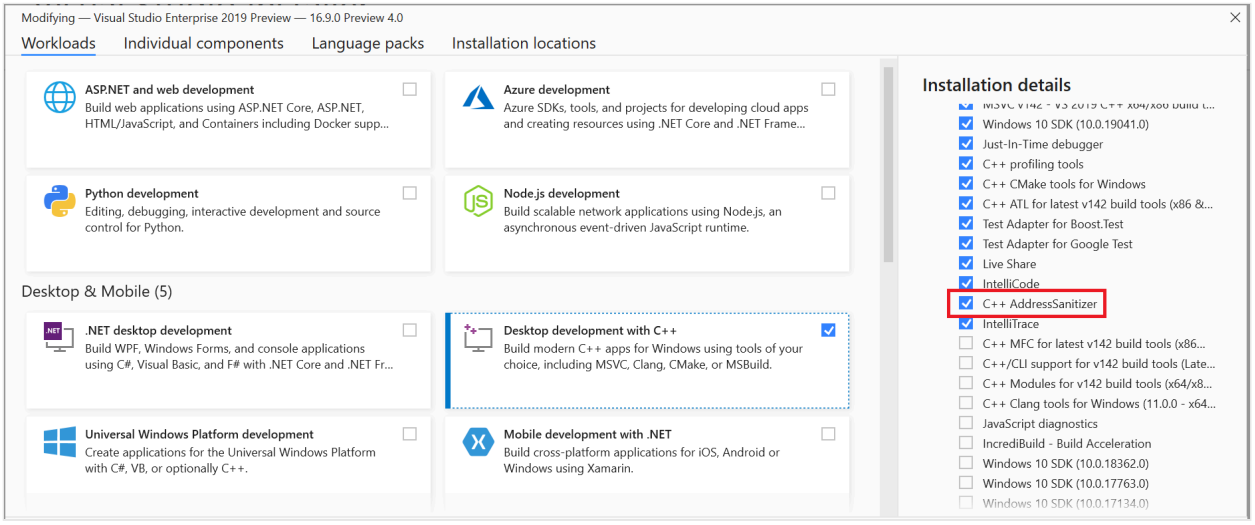
AddressSanitizer is integrated with the Visual Studio project system, the CMake build system, and the IDE. Projects can enable AddressSanitizer by setting a project property, or by using one extra compiler option: `/fsanitize=address`. The new option is compatible with all levels of optimization and configurations of x86 and x64. However, it isn't compatible with edit-and-continue, incremental linking, and /RTC.

Starting in Visual Studio 2019 version 16.9, Microsoft's AddressSanitizer technology enables integration with the Visual Studio IDE. The functionality can optionally create a crash dump file when the sanitizer finds a bug at runtime. If you set the `ASAN_SAVE_DUMPS=MyFileName.dmp` environment variable before you run your program, a crash dump file is created with extra metadata for efficient post-mortem debugging of precisely diagnosed bugs. These dump files make extended use of AddressSanitizer easier for:

- Local machine testing
- On-premises distributed testing
- Cloud-based workflows for testing

## Install AddressSanitizer

C++ workloads in the Visual Studio Installer install the AddressSanitizer libraries and IDE integration by default. However, if you're upgrading from an older version of Visual Studio 2019, use the Installer to enable ASan support after the upgrade. You can open the installer from the Visual Studio main menu via **Tools** > **Get Tools and Features...** Choose **Modify** on your existing Visual Studio installation from the Visual Studio Installer to get to the following screen.




> **Note**
>
> If you run Visual Studio on the new update but haven't installed ASan, you'll get an error when you run your code:
>
> LNK1356: cannot find library 'clang_rt.asan_dynamic-i386.lib'

## Use AddressSanitizer

Start building your executables with the `/fsanitize=address` compiler option using any of these common development methods:

- Command line builds
- Visual Studio project system
- Visual Studio CMake integration

Recompile, then run your program normally. This code generation exposes many types of precisely diagnosed bugs. These errors get reported in three ways: in the debugger IDE, on the command line, or stored in a new type of dump file for precise off-line processing.

Microsoft recommends you use AddressSanitizer in these three standard workflows:

- **Developer inner loop**
  - Visual Studio - Command line
  - Visual Studio - Project system
  - Visual Studio - CMake
- **CI/CD** - continuous integration / continuous development
  - Error reporting - New AddressSanitizer dump files
- **Fuzzing** - building with the libFuzzer wrapper
  - Azure OneFuzz
  - Local Machine

This article covers the information you require to enable the three workflows listed previously. The information is specific to the **platform-dependent** Windows 10 (and later) implementation of AddressSanitizer. This documentation supplements the excellent documentation from Google, Apple, and GCC already published.

> **Note**
>
> Support is limited to x86 and x64 on Windows 10 and later. Send us feedback on what you'd like to see in future releases. Your feedback helps us prioritize other sanitizers for the future, such as `/fsanitize=thread`, `/fsanitize=leak`, `/fsanitize=memory`, `/fsanitize=undefined`, or `/fsanitize=hwaddress`. You can report bugs here if you run into issues.

# Use AddressSanitizer from a developer command prompt

Use the `/fsanitize=address` compiler option in a developer command prompt to enable compiling for the AddressSanitizer runtime. The `/fsanitize=address` option is compatible with all existing C++ or C optimization levels (for example, `/Od`, `/O1`, `/O2`, `/O2 /GL`, and `PGO`). The option works with static and dynamic CRTs (for example, `/MD`, `/MDd`, `/MT`, and `/MTd`). It works whether you create an EXE or a DLL. Debug information is required for optimal formatting of call stacks. In the following example, `cl /fsanitize=address /Zi` is passed on the command line.

The AddressSanitizer libraries (.lib files) are linked for you automatically. For more information, see AddressSanitizer language, build, and debugging reference.

## Example - basic global buffer overflow

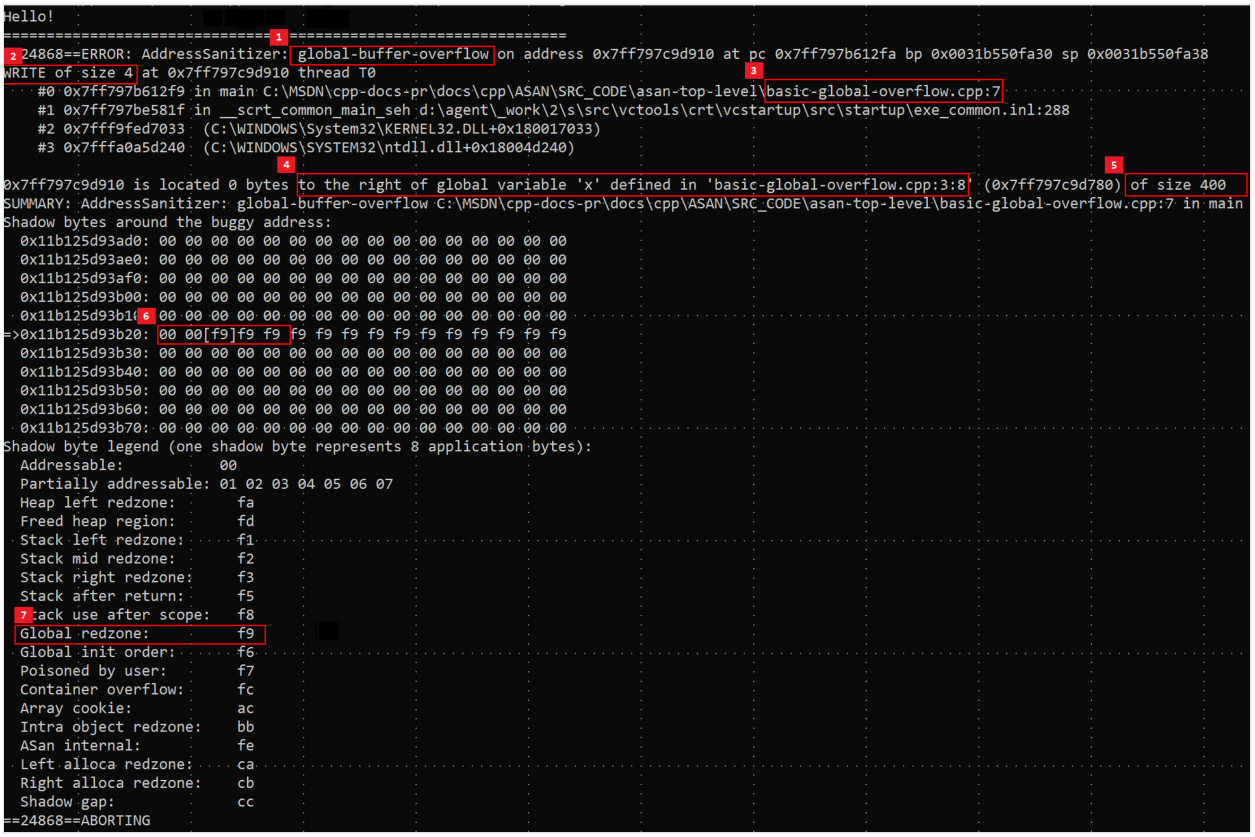
C++

```
// basic-global-overflow.cpp
#include <stdio.h>
int x[100];
int main() {
    printf("Hello!\n");
    x[100] = 5; // Boom!
    return 0;
}
```

Using a developer command prompt for Visual Studio 2019, compile *`main.cpp`* using `/fsanitize=address /Zi`


When you run the resulting *`main.exe`* at the command line, it creates the formatted error report that follows.

Consider the overlaid, red boxes that highlight seven key pieces of information:



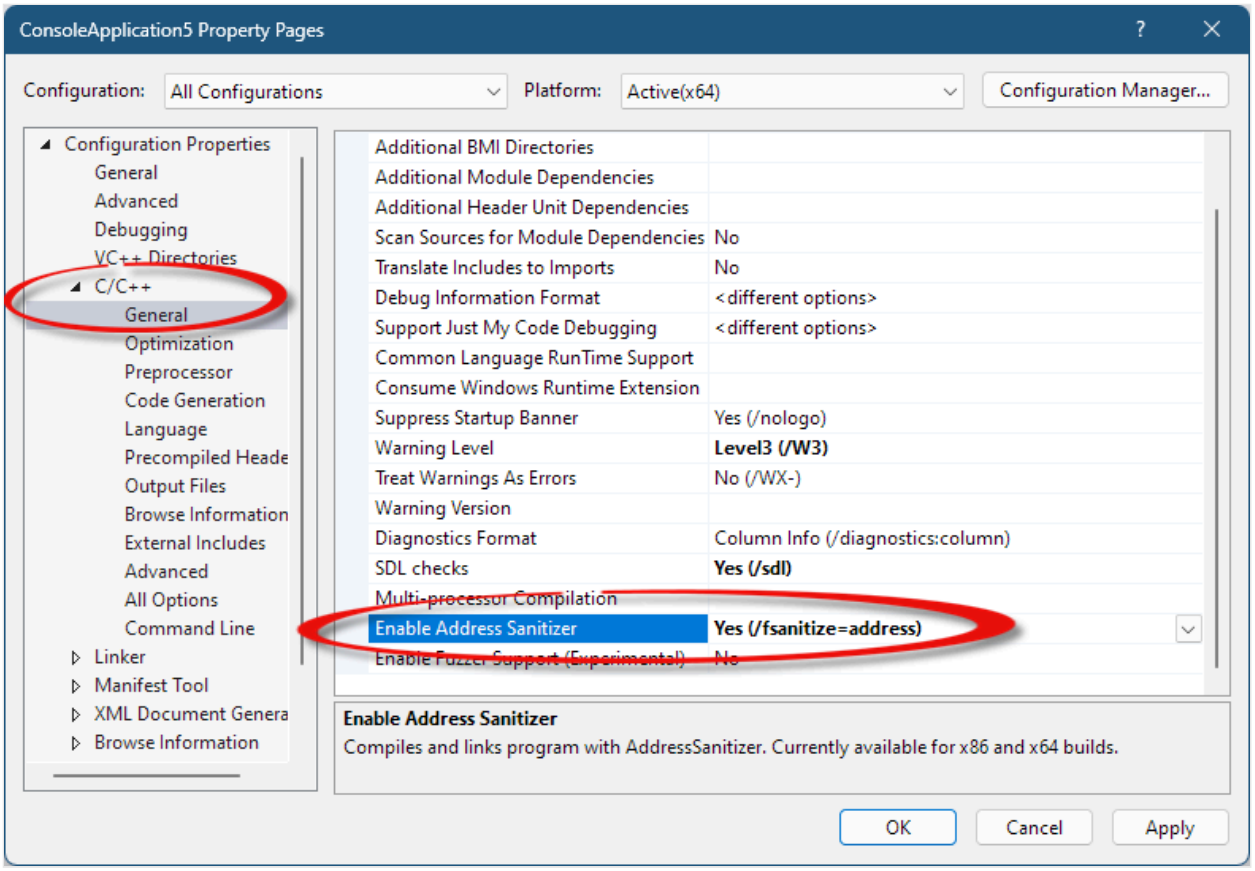

## Red highlights, from top to bottom

1. The memory safety bug is a global-buffer-overflow.
2. There were **4 bytes** (32 bits) **stored** outside any user-defined variable.
3. The store took place in function `main()` defined in file `basic-global-overflow.cpp` on line 7.
4. The variable named `x` gets defined in basic-global-overflow.cpp on line 3, starting at column 8
5. This global variable `x` is of size 400 bytes
6. The exact shadow byte describing the address targeted by the store had a value of `0xf9`
7. The shadow byte legend says `0xf9` is an area of padding to the right of `int x[100]`

> **Note**
>
> The function names in the call stack are produced through the LLVM symbolizer that's invoked by the runtime upon error.

# Use AddressSanitizer in Visual Studio

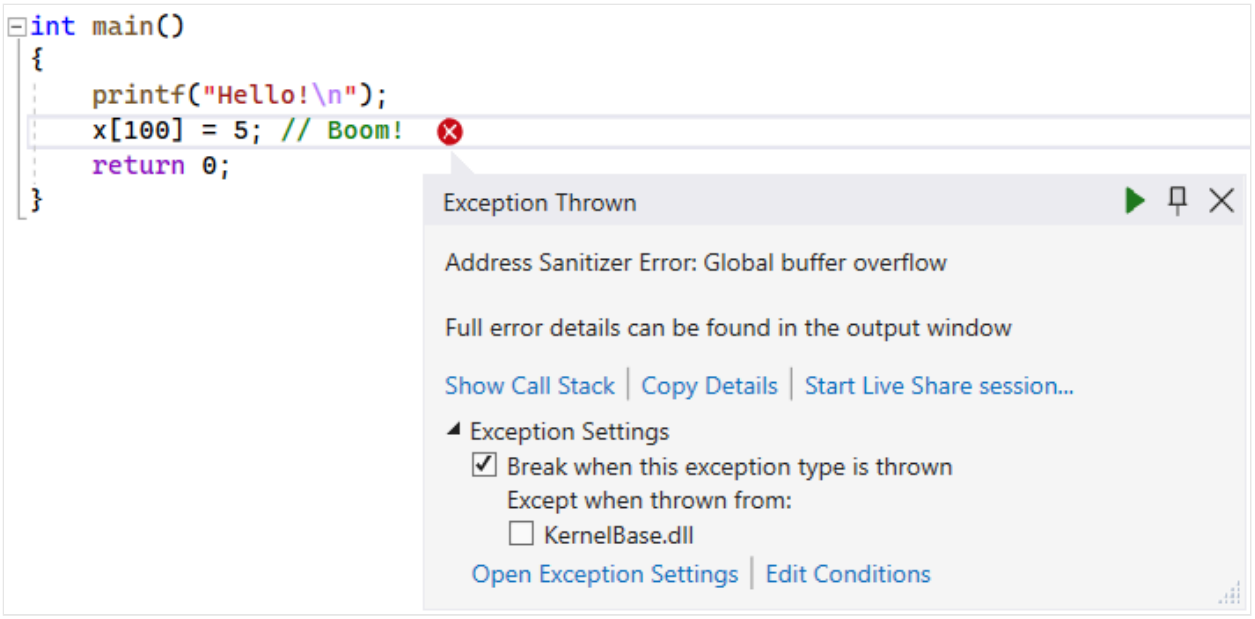
AddressSanitizer is integrated with the Visual Studio IDE. To turn on AddressSanitizer for an MSBuild project, right-click on the project in Solution Explorer and choose **Properties**. In the **Property Pages** dialog, select **Configuration Properties** > **C/C++** > **General**, then modify the **Enable AddressSanitizer** property. Choose **OK** to save your changes.




To build from the IDE, opt out of any incompatible options. For an existing project compiled by using `/Od` (or Debug mode), you may need to turn off these options:

- Turn off edit and continue
- Turn off /RTC1 (runtime checks)
- Turn off /INCREMENTAL (incremental linking)

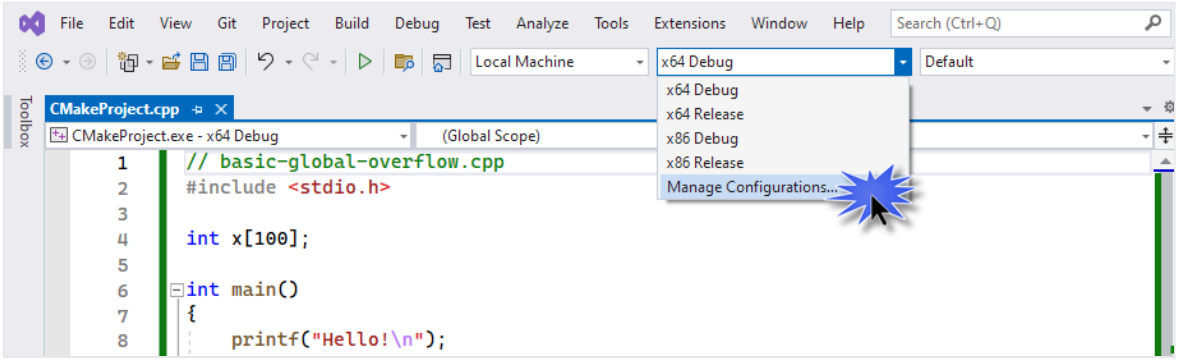
To build and run the debugger, press **F5**. An **Exception Thrown** window appears in Visual Studio:




# Use AddressSanitizer from Visual Studio: CMake

To enable AddressSanitizer for a CMake project created to target Windows, follow these steps:

1. Open the **Configurations** dropdown in the toolbar at the top of the IDE and select **Manage Configurations**.




   That opens the CMake Project Settings editor, which reflects the contents of your project's `CMakeSettings.json` file.

2. Choose the **Edit JSON** link in the editor. This selection switches the view to raw JSON.

3. Add the following snippet to the `"windows-base"` preset, inside `"configurePresets":` to turn on Address Sanitizer:

   JSON

```
"environment": {
  "CFLAGS": "/fsanitize=address",
  "CXXFLAGS": "/fsanitize=address"
}
```

`"configurePresets"` looks something like this, afterwards:

JSON

```
    "configurePresets": [
      {
        "name": "windows-base",
        "hidden": true,
        "generator": "Ninja",
        "binaryDir": "${sourceDir}/out/build/${presetName}",
        "installDir": "${sourceDir}/out/install/${presetName}",
        "cacheVariables": {
          "CMAKE_C_COMPILER": "cl.exe",
          "CMAKE_CXX_COMPILER": "cl.exe"
        },
        "condition": {
          "type": "equals",
          "lhs": "${hostSystemName}",
          "rhs": "Windows"
        },
        "environment": {
          "CFLAGS": "/fsanitize=address",
          "CXXFLAGS": "/fsanitize=address"
        }
      },
```

4. Address sanitizer doesn't work if edit-and-continue is specified (`/ZI`), which is enabled by default for new CMake projects. In `CMakeLists.txt`, comment out (prefix with `#`) the line that starts with `set(CMAKE_MSVC_DEBUG_INFORMATION_FORMAT"`. That line looks something like this, afterwards:

   JSON

   ```
   # set(CMAKE_MSVC_DEBUG_INFORMATION_FORMAT
   "$<IF:$<AND:$<C_COMPILER_ID:MSVC>,$<CXX_COMPILER_ID:MSVC>>,$<$<CONFIG:D
   ebug,RelWithDebInfo>:EditAndContinue>,$<$<CONFIG:Debug,RelWithDebInfo>:
   ProgramDatabase>>")
   ```

5. Enter **Ctrl+S** to save this JSON file

6. Clear your CMake cache directory and reconfigure by choosing from the Visual Studio menu: **Project** > **Delete cache and Reconfigure**. Choose **Yes** when the

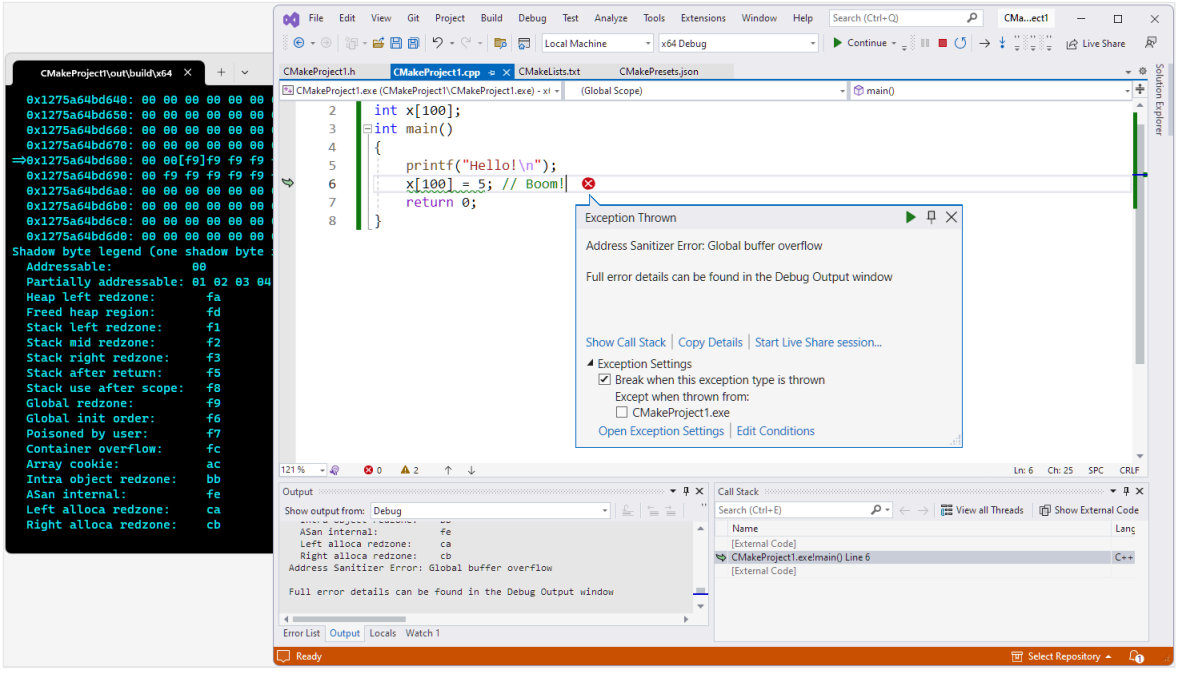
prompt appears to clear your cache directory and reconfigure.

7. Replace the contents of the source file (for example, `CMakeProject1.cpp`) with the following:

```
C++

// CMakeProject1.cpp : Defines the entry point for the application

#include <stdio.h>

int x[100];

int main()
{
    printf("Hello!\n");
    x[100] = 5; // Boom!
    return 0;
}
```

8. Choose **F5** to recompile and run under the debugger.

   This screenshot captures the error from the CMake build.




# AddressSanitizer crash dumps

We introduced new functionality in AddressSanitizer for use with cloud and distributed workflows. This functionality allows offline viewing of an AddressSanitizer error in the IDE. The error gets overlaid on top of your source, just as you would experience in a live debug session.

These new dump files can lead to efficiencies when analyzing a bug. You don't need to rerun, or find remote data or look for a machine that went off-line.

To produce a new type of dump file that can be viewed in Visual Studio on another machine at a later date:

Windows Command Prompt

```
set ASAN_SAVE_DUMPS=MyFileName.dmp
```

Starting with Visual Studio 16.9 you can display **a precisely diagnosed error**, stored in your `*.dmp` file, on top of your source code.

This new crash dump functionality enables cloud-based workflows, or distributed testing. It can also be used to file a detailed, actionable bug in any scenario.

# Example errors

AddressSanitizer can detect several kinds of memory misuse errors. Here are many of the runtime errors reported when you run your binaries compiled by using the AddressSanitizer (`/fsanitize=address`) compiler option:

- alloc-dealloc-mismatch
- allocation-size-too-big
- calloc-overflow
- double-free
- dynamic-stack-buffer-overflow
- global-buffer-overflow
- heap-buffer-overflow
- heap-use-after-free
- invalid-allocation-alignment
- memcpy-param-overlap
- new-delete-type-mismatch
- stack-buffer-overflow
- stack-buffer-underflow
- stack-use-after-return
- stack-use-after-scope
- strncat-param-overlap
- use-after-poison

For more information about the examples, see AddressSanitizer error examples.

# Differences with Clang 12.0

MSVC currently differs from Clang 12.0 in two functional areas:

- **stack-use-after-scope** - this setting is on by default and can't be turned off.
- **stack-use-after-return** - this functionality requires an extra compiler option, and isn't available by only setting `ASAN_OPTIONS`.

These decisions were made to reduce the test matrix required to deliver this first version.

Features that could lead to false positives in Visual Studio 2019 16.9 weren't included. That discipline enforced the effective testing integrity necessary when considering interop with decades of existing code. More capabilities may be considered in later releases:

- Initialization Order Fiasco
- Intra Object Overflow
- Container Overflow
- Pointer Subtraction/Comparison

For more information, see Building for AddressSanitizer with MSVC.

# Existing industry documentation

Extensive documentation already exists for these language and platform-dependent implementations of the AddressSanitizer technology.

- Google
- Apple
- GCC

This seminal paper on the AddressSanitizer (external) describes the implementation.

# See also

AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration
AddressSanitizer error examples

## Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

# Walkthrough: Use Address Sanitizer Continue On Error to find memory safety issues

Article • 08/01/2023

In this walkthrough, create checked builds that find and report memory safety errors.

Memory safety errors like out-of-bounds memory reads and writes, using memory after it has been freed, `NULL` pointer dereferences, and so on, are a top concern for C/C++ code. Address Sanitizer (ASAN) is a compiler and runtime technology that exposes these kinds of hard-to-find bugs, and does it with zero false positives. For an overview of ASAN, see AddressSanitizer.

Continue On Error (COE) is a new ASAN feature that automatically diagnoses and reports memory safety errors as your app runs. When your program exits, a summary of unique memory safety errors is output to `stdout`, `stderr`, or to a log file of your choice. When you create a standard C++ checked build with `-fsanitizer=address`, calls to allocators, deallocators such as `free`, `memcpy`, `memset`, and so on, are forwarded to the ASAN runtime. The ASAN runtime provides the same semantics for these functions, but monitors what happens with the memory. ASAN diagnoses and reports hidden memory safety errors, with zero false positives, as your app runs.

A significant advantage of COE is that, unlike the previous ASAN behavior, your program doesn't stop running when the first memory error is found. Instead, ASAN notes the error, and your app continues to run. After your app exits, a summary of all the memory issues is output.

It's a good practice to create a checked build of your C or C++ app with ASAN turned on, and then run your app in your test harness. As your tests exercise the code paths in your app looking for bugs, you'll also find out if those code paths harbor memory safety issues without interfering with the tests.

When your app finishes, you get a summary of the memory issues. With COE, you can compile and deploy an existing application into limited production to find memory safety issues. You can run the checked build for days to fully exercise the code, although the app will run slower due to the ASAN instrumentation.

You can use this feature to create a new shipping gate. If all your existing tests pass, but COE reports a memory safety error or a leak, don’t ship the new code or integrate it into a parent branch.

Don't deploy a build with COE enabled into production! COE is intended to be used in testing and development environments only. You shouldn't use an ASAN enabled build in production because of the performance impact of the instrumentation added to detect memory errors, the risk of exposing the internal implementation if errors are reported, and to avoid increasing the surface area of possible security exploits by shipping the library functions that ASAN substitutes for memory allocation, freeing, and so on.

In the following examples, you create checked builds and set an environment variable to output the address sanitizer information to `stdout` to see the memory safety errors that ASAN reports.

# Prerequisites

To complete this walkthrough, you need Visual Studio 2022 17.6 or later with the *Desktop development with C++ workload* installed.

# Double free example

In this example, you create a build with ASAN enabled to test what happens when memory is double freed. ASAN detects this error and reports it. In this example, the program continues to run after the error is detected, which leads to a second error--using memory that's been freed. A summary of the errors is output to `stdout` when the program exits.

Create the example:

1. Open a developer command prompt: Open the **Start** menu, type *Developer*, and select the latest command prompt such as **Developer Command Prompt for VS 2022** from the list of matches.

2. Create a directory on your machine to run this example. For example, `%USERPROFILE%\Desktop\COE`.

3. In that directory, create an empty source file. For example, `doublefree.cpp`

4. Paste the following code into the file:

```
C++

#include <stdio.h>
#include <stdlib.h>
```

```
void BadFunction(int *pointer)
{
    free(pointer);
    free(pointer); // double-free!
}

int main(int argc, const char *argv[])
{
    int *pointer = static_cast<int *>(malloc(4));
    BadFunction(pointer);

    // Normally we'd crash before this, but with COE we can see heap-
use-after-free error as well
    printf("\n\n******* Pointer value: %d\n", *pointer);

    return 1;
}
```

In the preceding code, `pointer` is freed twice. This is a contrived example, but double frees are an easy mistake to make in more complex C++ code.

Create a build of the preceding code with COE turned on with the following steps:

1. Compile the code in the developer command prompt you opened earlier: `cl -fsanitize=address -Zi doublefree.cpp`. The `-fsanitize=address` switch turns on ASAN, and `-Zi` creates a separate PDB file that address sanitizer uses to display memory error location information.
2. Send ASAN output to `stdout` by setting the `ASAN_OPTIONS` environment variable in the developer command prompt as follows: `set ASAN_OPTIONS=continue_on_error=1`
3. Run the test code with: `doublefree.exe`

The output shows that there was a double free error and the call stack where it happened. The report starts out with a call stack that shows the error happened in `BadFunction`:

Windows Command Prompt

```
==22976==ERROR: AddressSanitizer: attempting double-free on 0x01e03550 in
thread T0:
    #0  free
D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp(69)
    #1  BadFunction
C:\Users\xxx\Desktop\COE\doublefree.cpp(8)
    #2  main
C:\Users\xxx\Desktop\COE\doublefree.cpp(14)
    #3  __scrt_common_main_seh
```

```
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
    #4  BaseThreadInitThunk            Windows
    #5  RtlInitializeExceptionChain    Windows
```

Next, there's information about the freed memory and a call stack for where the memory was allocated:

Windows Command Prompt

```
0x01e03550 is located 0 bytes inside of 4-byte region
[0x01e03550,0x01e03554)
freed by thread T0 here:
    #0  free
D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp(69)
    #1  BadFunction
C:\Users\xxx\Desktop\COE\doublefree.cpp(7)
    #2  main
C:\Users\xxx\Desktop\COE\doublefree.cpp(14)
    #3  __scrt_common_main_seh
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
    #4  BaseThreadInitThunk            Windows
    #5  RtlInitializeExceptionChain    Windows

previously allocated by thread T0 here:
    #0  malloc
D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp(85)
    #1  main
C:\Users\xxx\Desktop\COE\doublefree.cpp(13)
    #2  __scrt_common_main_seh
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
    #3  BaseThreadInitThunk            Windows
    #4  RtlInitializeExceptionChain    Windows
```

Then there's information about the heap-use-after-free error. This refers to using `*pointer` in the `printf()` call because the memory `pointer` refers to was freed earlier. The call stack where the error occurs is listed, as are the call stacks where this memory was allocated and freed:

Windows Command Prompt

```
==35680==ERROR: AddressSanitizer: heap-use-after-free on address 0x02a03550
at pc 0x00e91097 bp 0x012ffc64 sp 0x012ffc58READ of size 4 at 0x02a03550
thread T0
        #0  main
C:\Users\xxx\Desktop\Projects\ASAN\doublefree.cpp(18)
        #1  __scrt_common_main_seh
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
        #2  BaseThreadInitThunk            Windows
```

```
          #3  RtlInitializeExceptionChain    Windows

0x02a03550 is located 0 bytes inside of 4-byte region
[0x02a03550,0x02a03554)
freed by thread T0 here:
          #0  free
D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp(69)
          #1  BadFunction
C:\Users\xxx\Desktop\Projects\ASAN\doublefree.cpp(7)
          #2  main
C:\Users\xxx\Desktop\Projects\ASAN\doublefree.cpp(14)
          #3  __scrt_common_main_seh
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
          #4  BaseThreadInitThunk            Windows
          #5  RtlInitializeExceptionChain    Windows

previously allocated by thread T0 here:
          #0  malloc
D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp(85)
          #1  main
C:\Users\xxx\Desktop\Projects\ASAN\doublefree.cpp(13)
          #2  __scrt_common_main_seh
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
          #3  BaseThreadInitThunk            Windows
          #4  RtlInitializeExceptionChain    Windows
```

Next, there's information about the shadow bytes in the vicinity of the buffer overflow. For more information about shadow bytes, see AddressSanitizer shadow bytes.

Following the shadow byte information, you'll see the output from the program, which indicates that it continued running after ASAN detected the error:

Windows Command Prompt

```
******* Pointer value: xxx
```

Then there's a summary of the source files where the memory error happened. It's sorted by the unique call stacks for the memory errors in that file. A unique call stack is determined by the type of error and the call stack where the error occurred.

This sorting prioritizes memory safety issues that may be the most concerning. For example, five unique call stacks leading to different memory safety errors in the same file is potentially more worrisome than one error that hits many times. The summary looks like this:

Windows Command Prompt

```
=== Files in priority order ===

File: D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp Unique call stacks: 1
File: C:\Users\xxx\Desktop\COE\doublefree.cpp Unique call stacks: 1
```

Finally, the report contains a summary of where the memory errors occurred:

Windows Command Prompt

```
=== Source Code Details: Unique errors caught at instruction offset from
source line number, in functions, in the same file. ===

File: D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp
        Func: free()
                Line: 69 Unique call stacks (paths) leading to error at line
69 : 1
                        Bug: double-free at instr 19 bytes from start of
line
File: C:\Users\xxx\Desktop\COE\doublefree.cpp
        Func: main()
                Line: 18 Unique call stacks (paths) leading to error at line
18 : 1
                        Bug: heap-use-after-free at instr 55 bytes from
start of line

>>>Total: 2 Unique Memory Safety Issues (based on call stacks not source
position) <<<

#0 C:\Users\xxx\Desktop\COE\doublefree.cpp Function: main(Line:18)
        Raw HitCnt: 1  On Reference: 4-byte-read-heap-use-after-free
#1 D:\a\_work\1\s\src\vctools\asan\llvm\compiler-
rt\lib\asan\asan_malloc_win_thunk.cpp Function: free(Line:69)
        Raw HitCnt: 1
```

# Out of bounds memory access example

In this example, you create a build with ASAN enabled to test what happens when an app access memory that is out-of-bounds. ASAN detects this error and reports a summary of the errors to `stdout` when the program exits.

Create the example:

1. Open a developer command prompt: open the **Start** menu, type *Developer*, and select the latest command prompt such as **Developer Command Prompt for VS 2022** from the list of matches.

2. Create a directory on your machine to run this example. For example, `%USERPROFILE%\Desktop\COE`.

3. In that directory, create a source file, for example, `coe.cpp`, and paste the following code:

```C++
#include <stdlib.h>

char* func(char* buf, size_t sz)
{
    char* local = (char*)malloc(sz);
    for (auto ii = 0; ii <= sz; ii++) // bad loop exit test
    {
        local[ii] = ~buf[ii]; // Two memory safety errors
    }

    return local;
}

char buffer[10] = {0,1,2,3,4,5,6,7,8,9};

void main()
{
    char* inverted_buf= func(buffer, 10);
}
```

In the preceding code, the parameter `sz` is 10 and the original buffer is 10 bytes. There are two memory safety errors:

- an out-of-bounds load from `buf` in the `for` loop
- an out-of-bounds store to `local` in the `for` loop

The buffer overflow is due to the loop exit test `<=sz`. When this example runs, it's *secure by coincidence*. That's because of the over-allocation and alignment done by most C++ runtime implementations. When `sz % 16 == 0`, the final write to `local[ii]` corrupts memory. Other cases only read/write to the "malloc slop," which is extra memory allocated due to the way the C Runtime (CRT) pads allocations to a 0 mod 16 boundary.

Errors are only observable if the page following the allocation is unmapped, or upon use of corrupted data. All other cases are silent in this example. With Continue On Error, the errors are made visible in the summary after the program runs to completion.

Create a build of the preceding code with COE turned on:

1. Compile the code with `cl -fsanitize=address -Zi coe.cpp`. The `-fsanitize=address` switch turns on ASAN, and `-Zi` creates a separate PDB file that address sanitizer uses to display memory error location information.
2. Send ASAN output to `stdout` by setting the `ASAN_OPTIONS` environment variable in the developer command prompt as follows: `set ASAN_OPTIONS=continue_on_error=1`
3. Run the test code with: `coe.exe`

The output shows that there were two memory buffer overflow errors and provides the call stack for where they happened. The report starts out like this:

Windows Command Prompt

```
==9776==ERROR: AddressSanitizer: global-buffer-overflow on address
0x0047b08a at pc 0x003c121b bp 0x012ffaec sp 0x012ffae0
READ of size 1 at 0x0047b08a thread T0
     #0  func                            C:\Users\xxx\Desktop\COE\coe.cpp(8)
     #1  main                            C:\Users\xxx\Desktop\COE\coe.cpp(18)
     #2  __scrt_common_main_seh
D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288)
     #3  BaseThreadInitThunk             Windows
     #4  RtlInitializeExceptionChain     Windows
```

Next, there's information about the shadow bytes in the vicinity of the buffer overflow. For more information about shadow bytes, see AddressSanitizer shadow bytes.

Following the shadow byte report, there's a summary of the source files where the memory errors happened. It's sorted by the unique call stacks for the memory errors in that file. A unique call stack is determined by the type of error and the call stack where the error occurred.

This sorting prioritizes memory safety issues that may be the most concerning. For example, five unique call stacks leading to different memory safety errors in the same file is potentially more worrisome than one error that hits many times.

The summary looks like this:

Windows Command Prompt

```
=== Files in priority order ===

File: C:\Users\xxx\Desktop\COE\coe.cpp Unique call stacks: 2
```

Finally, the report contains a summary of where the memory errors occurred. Continue On Error reports two distinct errors that occur on the same source line. The first error

reads memory at a global address in the `.data` section, and the other writes to memory allocated from the heap.

The report looks like this:

Windows Command Prompt

```
=== Source Code Details: Unique errors caught at instruction offset from
source line number, in functions, in the same file. ===

File: C:\Users\xxx\Desktop\COE\coe.cpp
    Func: func()
        Line: 8 Unique call stacks (paths) leading to error at line 8 : 2
            Bug: heap-buffer-overflow at instr 124 bytes from start of line

>>>Total: 2 Unique Memory Safety Issues (based on call stacks not source
position) <<<

#0 C:\Users\xxx\Desktop\COE\coe.cpp Function: func(Line:8)
    Raw HitCnt: 1  On Reference: 1-byte-read-global-buffer-overflow
#1 C:\Users\xxx\Desktop\COE\coe.cpp Function: func(Line:8)
    Raw HitCnt: 1  On Reference: 1-byte-write-heap-buffer-overflow
```

The default Address Sanitizer runtime behavior terminates the app after reporting the first error it finds. It doesn't allow the "bad" machine instruction to execute. The new Address Sanitizer runtime diagnoses and reports errors, but then executes subsequent instructions.

COE tries to automatically return control back to the application after reporting each memory safety error. There are situations when it can't, such as when there's a memory access violation (AV) or a failed memory allocation. COE doesn't continue after access violations that the program's structured exception handling doesn't catch. If COE can't return execution to the app, a `CONTINUE CANCELLED - Deadly Signal. Shutting down.` message is output.

# Select where to send ASAN output

Use the `ASAN_OPTIONS` environment variable to determine where to send ASAN output as follows:

- Output to stdout: `set ASAN_OPTIONS=continue_on_error=1`
- Output to stderr: `set ASAN_OPTIONS=continue_on_error=2`
- Output to a log file of your choice: `set COE_LOG_FILE=yourfile.log`

# Handling undefined behavior

The ASAN runtime doesn't mimic all of the undefined behaviors of the C and C++ allocation/deallocation functions. The following example demonstrates how the ASAN version of **_alloca** differs from the C runtime version:

C++

```
#include <cstdio>
#include <cstring>
#include <malloc.h>
#include <excpt.h>
#include <windows.h>

#define RET_FINISH 0
#define RET_STACK_EXCEPTION 1
#define RET_OTHER_EXCEPTION 2

int foo_redundant(unsigned long arg_var)
{
    char *a;
    int ret = -1;

    __try
    {
        if ((arg_var+3) > arg_var)
        {
            // Call to _alloca using parameter from main
            a = (char *) _alloca(arg_var);
            memset(a, 0, 10);
        }
        ret = RET_FINISH;
    }
    __except(1)
    {
        ret = RET_OTHER_EXCEPTION;
        int i = GetExceptionCode();
        if (i == EXCEPTION_STACK_OVERFLOW)
        {
            ret = RET_STACK_EXCEPTION;
        }
    }
    return ret;
}

void main()
{
    int cnt = 0;

    if (foo_redundant(0xfffffff0) == RET_STACK_EXCEPTION)
    {
        cnt++;
```

```
    }

    if (cnt == 1)
    {
        printf("pass\n");
    }
    else
    {
        printf("fail\n");
    }
}
```

In `main()` a large number is passed to `foo_redundant`, which is ultimately passed to `_alloca()`, which causes `_alloca()` to fail.

This example outputs `pass` when compiled without ASAN (that is, no `-fsanitize=address` switch) but outputs `fail` when compiled with ASAN turned on (that is, with the `-fsanitize=address` switch). That's because without ASAN, the exception code matches `RET_STACK_EXCEPTION` so `cnt` is set to 1. It behaves differently when compiled with ASAN on because the thrown exception is an Address Sanitizer error instead: dynamic-stack-buffer-overflow. That means the code returns `RET_OTHER_EXCEPTION` instead of `RET_STACK_EXCEPTION` so `cnt` isn't set to 1.

## Other benefits

With the new ASAN runtime, no extra binaries need to be deployed with your app. This makes it even easier to use ASAN with your normal test harness because you don't have to manage extra binaries.

## See also

AddressSanitizer Continue on Error blog post
Example memory safety errors
-Zi compiler flag
-fsanitize=address compiler flag
Top 25 most dangerous software weaknesses

# AddressSanitizer language, build, and debugging reference

Article • 02/06/2024

The sections in this article describe the AddressSanitizer language specification, compiler options, and linker options. They also describe the options that control Visual Studio debugger integration specific to the AddressSanitizer.

For more information on the AddressSanitizer runtime, see the runtime reference. It includes information on intercepted functions and how to hook custom allocators. For more information on saving crash dumps from AddressSanitizer failures, see the crash dump reference.

## Language specification

### `__SANITIZE_ADDRESS__`

The `__SANITIZE_ADDRESS__` preprocessor macro is defined as `1` when /fsanitize=address is set. This macro is useful for advanced users to conditionally specify source code for the presence of the AddressSanitizer runtime.

C++

```
#include <cstdio>

int main() {
    #ifdef __SANITIZE_ADDRESS__
        printf("Address sanitizer enabled");
    #else
        printf("Address sanitizer not enabled");
    #endif
    return 1;
}
```

### `__declspec(no_sanitize_address)`

The __declspec(no_sanitize_address) specifier can be used to selectively disable the sanitizer on functions, local variables, or global variables. This `__declspec` affects *compiler* behavior, not *runtime* behavior.

C++

```
__declspec(no_sanitize_address)
void test1() {
    int x[100];
    x[100] = 5; // ASan exception not caught
}

void test2() {
    __declspec(no_sanitize_address) int x[100];
    x[100] = 5; // ASan exception not caught
}

__declspec(no_sanitize_address) int g[100];
void test3() {
    g[100] = 5; // ASan exception not caught
}
```

# Compiler

## `/fsanitize=address` compiler option

The `/fsanitize=address` compiler option instruments memory references in your code to catch memory safety errors at runtime. The instrumentation hooks loads, stores, scopes, `alloca`, and CRT functions. It can detect hidden bugs such as out-of-bounds, use-after-free, use-after-scope, and so on. For a nonexhaustive list of errors detected at runtime, see AddressSanitizer error examples.

`/fsanitize=address` is compatible with all existing C++ or C optimization levels (for example, `/Od`, `/O1`, `/O2`, `/O2 /GL`, and profile guided optimization). The code produced with this option works with static and dynamic CRTs (for example, `/MD`, `/MDd`, `/MT`, and `/MTd`). This compiler option can be used to create an .EXE or .DLL targeting x86 or x64. Debug information is required for optimal formatting of call stacks.

For examples of code that demonstrates several kinds of error detection, see AddressSanitizer error examples.

## `/fsanitize=fuzzer` compiler option (experimental)

The /fsanitize=fuzzer compiler option adds LibFuzzer to the default library list. It also sets the following sanitizer coverage options:

- Edge instrumentation points (`/fsanitize-coverage=edge`),
- inline 8-bit counters (`/fsanitize-coverage=inline-8bit-counters`),

- comparisons (`/fsanitize-coverage=trace-cmp`), and
- integer divisions (`/fsanitize-coverage=trace-div`).

We recommend you use `/fsanitize=address` with `/fsanitize=fuzzer`.

These libraries are added to the default library list when you specify `/fsanitize=fuzzer`:

Expand table

| Runtime option | LibFuzzer library |
|---|---|
| `/MT` | *`clang_rt.fuzzer_MT-{arch}`* |
| `/MD` | *`clang_rt.fuzzer_MD-{arch}`* |
| `/MTd` | *`clang_rt.fuzzer_MTd-{arch}`* |
| `/MDd` | *`clang_rt.fuzzer_MDd-{arch}`* |

LibFuzzer libraries that omit the `main` function are also available. It's your responsibility to define `main` and to call `LLVMFuzzerInitialize` and `LLVMFuzzerTestOneInput` when you use these libraries. To use one of these libraries, specify /NODEFAULTLIB and explicitly link with the following library that corresponds to your runtime and architecture:

Expand table

| Runtime option | LibFuzzer no_main library |
|---|---|
| `/MT` | *`clang_rt.fuzzer_no_main_MT-{arch}`* |
| `/MD` | *`clang_rt.fuzzer_no_main_MD-{arch}`* |
| `/MTd` | *`clang_rt.fuzzer_no_main_MTd-{arch}`* |
| `/MDd` | *`clang_rt.fuzzer_no_main_MDd-{arch}`* |

If you specify `/NODEFAULTLIB` and you don't specify one of these libraries, you'll get an unresolved external symbol link error.

## `/fsanitize-address-use-after-return` compiler option (experimental)

By default, the MSVC compiler (unlike Clang) doesn't generate code to allocate frames in the heap to catch use-after-return errors. To catch these errors using AddressSanitizer, you must:

1. Compile using the /fsanitize-address-use-after-return option.
2. Before executing your program, run `set ASAN_OPTIONS=detect_stack_use_after_return=1` to set the runtime check option.

The **`/fsanitize-address-use-after-return`** option causes the compiler to generate code to use a dual stack frame in the heap when locals are considered "address taken." This code is *much slower* than just using **`/fsanitize=address`** alone. For more information and an example, see Error: stack-use-after-return.

The dual stack frame in the heap remains after the return from the function that created it. Consider an example where the address of a local, allocated to a slot in the heap, is used after the return. The shadow bytes associated with the fake heap frame contain the value 0xF9. That 0xF9 means a stack-use-after-return error when the runtime reports the error.

Stack frames are allocated in the heap and remain after functions return. The runtime uses garbage collection to asynchronously free these fake call-frame objects, after a certain time interval. Addresses of locals get transferred to persistent frames in the heap. It's how the system can detect when any locals get used after the defining function returns. For more information, see the algorithm for stack use after return as documented by Google.

# Linker

## `/INFERASANLIBS[:NO]` linker option

The **`/fsanitize=address`** compiler option marks objects to specify which AddressSanitizer library to link into your executable. The libraries have names that begin with *`clang_rt.asan*`*. The /INFERASANLIBS linker option (on by default) links these libraries from their default locations automatically. Here are the libraries chosen and automatically linked in.

> **Note**
>
> In the following table, *`{arch}`* is either *`i386`* or *`x86_64`*. These libraries use Clang conventions for architecture names. MSVC conventions are normally `x86` and `x64` rather than `i386` and `x86_64`. They refer to the same architectures.

Expand table

| CRT option | AddressSanitizer runtime library (.lib) | Address runtime binary (.dll) |
|---|---|---|
| `/MT` or `/MTd` | `clang_rt.asan_dynamic-{arch}`, `clang_rt.asan_static_runtime_thunk-{arch}` | `clang_rt.asan_dynamic-{arch}` |
| `/MD` or `/MDd` | `clang_rt.asan_dynamic-{arch}`, `clang_rt.asan_dynamic_runtime_thunk-{arch}` | `clang_rt.asan_dynamic-{arch}` |

The linker option /INFERASANLIBS:NO prevents the linker from linking a `clang_rt.asan*` library file from the default location. Add the library path in your build scripts if you use this option. Otherwise, the linker reports an unresolved external symbol error.

### Previous Versions

Prior to Visual Studio 17.7 Preview 3, statically linked (`/MT` or `/MTd`) builds didn't use a DLL dependency. Instead, the AddressSanitizer runtime was statically linked into the user's EXE. DLL projects would then load exports from the user's EXE to access ASan functionality. Also, dynamically linked projects (`/MD` or `/MTd`) used different libraries and DLLs depending on whether the project was configured for debug or release. For more information about these changes and their motivations, see MSVC Address Sanitizer – One DLL for all Runtime Configurations.

Expand table

| CRT runtime option | DLL or EXE | AddressSanitizer runtime libraries |
|---|---|---|
| `/MT` | EXE | `clang_rt.asan-{arch}`, `clang_rt.asan_cxx-{arch}` |
| `/MT` | DLL | `clang_rt.asan_dll_thunk-{arch}` |
| `/MD` | Either | `clang_rt.asan_dynamic-{arch}`, `clang_rt.asan_dynamic_runtime_thunk-{arch}` |
| `/MTd` | EXE | `clang_rt.asan_dbg-{arch}`, `clang_rt.asan_dbg_cxx-{arch}` |
| `/MTd` | DLL | `clang_rt.asan_dbg_dll_thunk-{arch}` |
| `/MDd` | Either | `clang_rt.asan_dbg_dynamic-{arch}`, `clang_rt.asan_dbg_dynamic_runtime_thunk-{arch}` |

# Visual Studio integration

## `/fno-sanitize-address-vcasan-lib` compiler option

The `/fsanitize=address` option links in extra libraries for an improved Visual Studio debugging experience when an AddressSanitizer exception is thrown. These libraries are called **VCAsan**. The libraries enable Visual Studio to display AddressSanitizer errors on your source code. They also enable the executable to generate crash dumps when an AddressSanitizer error report is created. For more information, see Visual Studio AddressSanitizer extended functionality library.

The library chosen depends on the compiler options, and is automatically linked in.

Expand table

| Runtime option | VCAsan version |
|---|---|
| `/MT` | *`libvcasan.lib`* |
| `/MD` | *`vcasan.lib`* |
| `/MTd` | *`libvcasand.lib`* |
| `/MDd` | *`vcasand.lib`* |

However, if you compile using `/Z1` (Omit default library name), you must manually specify the library. If you don't, you'll get an unresolved external symbol link error. Here are some typical examples:

Output

```
error LNK2001: unresolved external symbol __you_must_link_with_VCAsan_lib
error LNK2001: unresolved external symbol ___you_must_link_with_VCAsan_lib
```

The improved debugging can be disabled at compile time by using the /fno-sanitize-address-vcasan-lib option.

## `ASAN_VCASAN_DEBUGGING` environment variable

The `/fsanitize=address` compiler option produces a binary that exposes memory safety bugs at runtime. When the binary is started from the command line, and the runtime reports an error, it prints the error details. It then exits the process. The `ASAN_VCASAN_DEBUGGING` environment variable can be set to launch the Visual Studio IDE immediately when the runtime reports an error. This compiler option lets you view the error, superimposed over your source code, at the precise line and column that caused the error.

To enable this behavior, run the command `set ASAN_VCASAN_DEBUGGING=1` before you run your application. You can disable the enhanced debugging experience by running `set ASAN_VCASAN_DEBUGGING=0`.

## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# AddressSanitizer runtime

Article • 04/03/2024

The AddressSanitizer runtime library intercepts common memory allocation functions and operations to enable inspection of memory accesses. There are several different runtime libraries that support the various types of executables the compiler may generate. The compiler and linker automatically link the appropriate runtime libraries, as long as you pass the /fsanitize=address option at compile time. You can override the default behavior by using the `/NODEFAULTLIB` option at link time. For more information, see the section on linking in the AddressSanitizer language, build, and debugging reference.

When compiling with `cl /fsanitize=address`, the compiler generates instructions to manage and check shadow bytes. Your program uses this instrumentation to check memory accesses on the stack, in the heap, or in the global scope. The compiler also produces metadata describing stack and global variables. This metadata enables the runtime to generate precise error diagnostics: function names, lines, and columns in your source code. Combined, the compiler checks and runtime libraries can precisely diagnose many types of memory safety bugs if they're encountered at run-time.

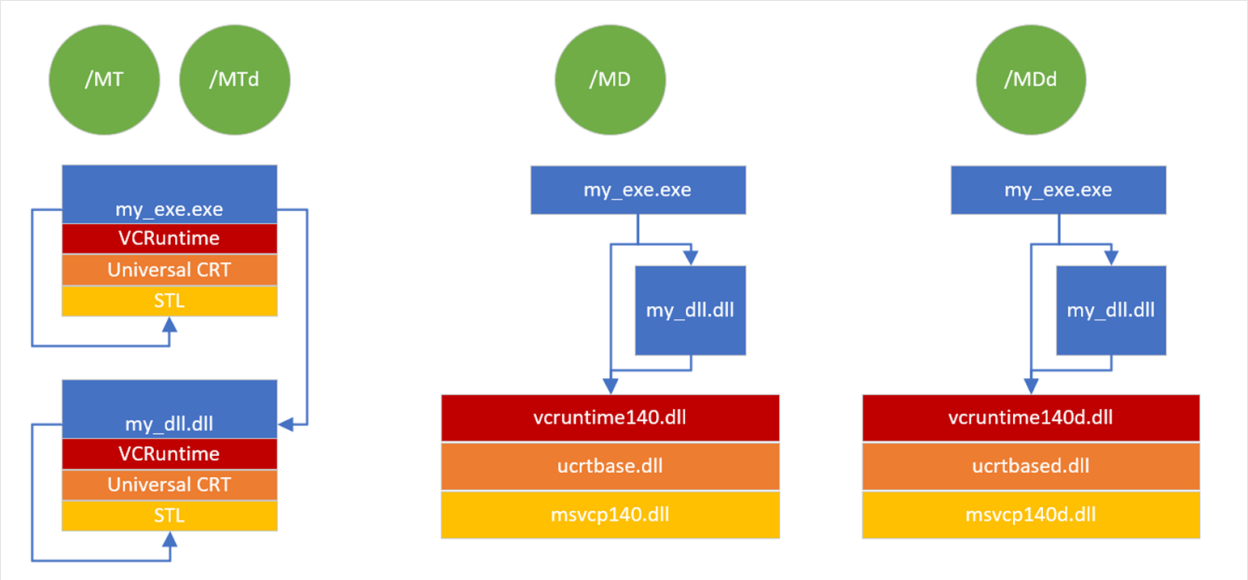
The list of runtime libraries for linking to the AddressSanitizer runtime, as of Visual Studio 17.7 Preview 3, follows. For more information about the `/MT` (statically link the runtime) and `/MD` (dynamically link the redist at runtime) options, see /MD, /MT, /LD (Use Run-Time Library).

> **Note**
>
> In the following table, *`{arch}`* is either *`i386`* or *`x86_64`*. These libraries use Clang conventions for architecture names. MSVC conventions are normally `x86` and `x64` rather than `i386` and `x86_64`, but they refer to the same architectures.

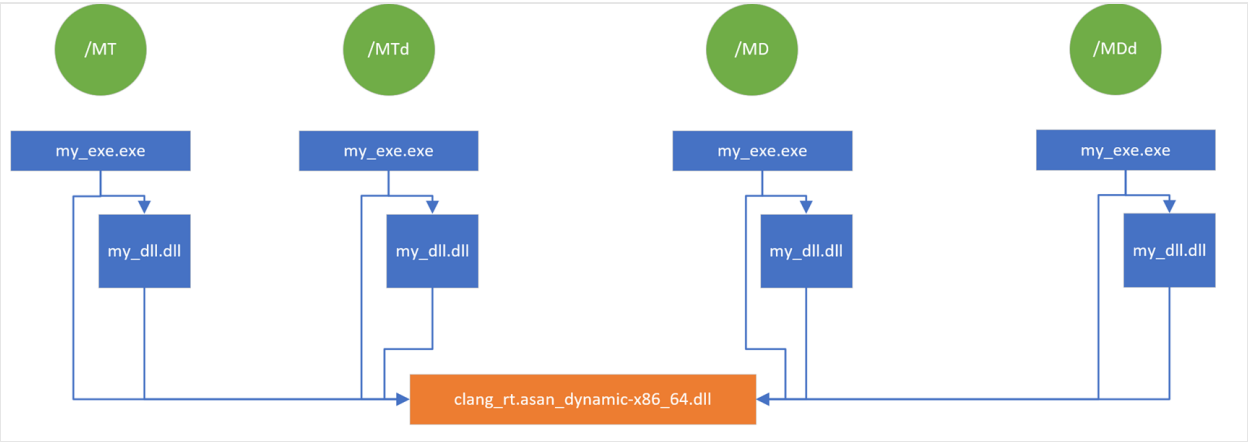
Expand table

| CRT option | AddressSanitizer runtime library (.lib) | Address runtime binary (.dll) |
|---|---|---|
| `/MT` or `/MTd` | *`clang_rt.asan_dynamic-{arch}`*, *`clang_rt.asan_static_runtime_thunk-{arch}`* | *`clang_rt.asan_dynamic-{arch}`* |
| `/MD` or `/MDd` | *`clang_rt.asan_dynamic-{arch}`*, *`clang_rt.asan_dynamic_runtime_thunk-{arch}`* | *`clang_rt.asan_dynamic-{arch}`* |

The following diagram shows how the language runtime libraries are linked for the `/MT`, `/MTd`, `/MD`, and `/MDd` compiler options:




The following diagram shows how the ASan library is linked for various compiler options:




Even when statically linking, the ASan runtime DLL must be present at runtime--unlike other C Runtime components.

## Previous versions

Before Visual Studio 17.7 Preview 3, statically linked (`/MT` or `/MTd`) builds didn't use a DLL dependency. Instead, the AddressSanitizer runtime was statically linked into the user's EXE. DLL projects would then load exports from the user's EXE to access ASan functionality.

Dynamically linked projects (`/MD` or `/MDd`) used different libraries and DLLs depending on whether the project was configured for debug or release. For more information

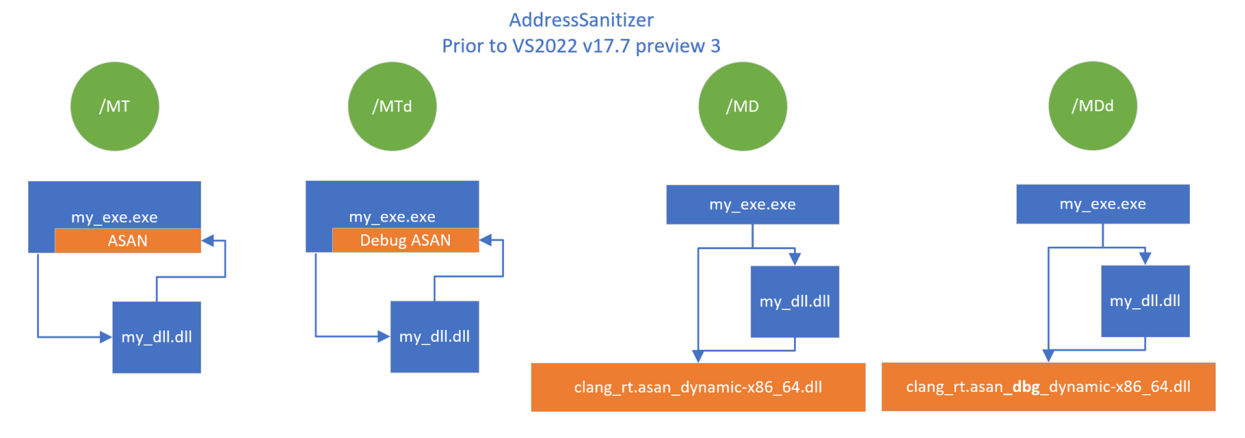
about these changes and their motivations, see MSVC Address Sanitizer – One DLL for all Runtime Configurations.

The following table describes the previous behavior of the AddressSanitizer runtime library linking, before Visual Studio 17.7 Preview 3:

Expand table

| CRT option | DLL or EXE | DEBUG? | ASan library (`.lib`) | ASan runtime binary (`.dll`) |
|---|---|---|---|---|
| `/MT` | EXE | No | *`clang_rt.asan-{arch}`*, *`clang_rt.asan_cxx-{arch}`* | None |
| `/MT` | DLL | No | *`clang_rt.asan_dll_thunk-{arch}`* | None |
| `/MD` | Either | No | *`clang_rt.asan_dynamic-{arch}`*, *`clang_rt.asan_dynamic_runtime_thunk-{arch}`* | *`clang_rt.asan_dynamic-{arch}`* |
| `/MT` | EXE | Yes | *`clang_rt.asan_dbg-{arch}`*, *`clang_rt.asan_dbg_cxx-{arch}`* | None |
| `/MT` | DLL | Yes | *`clang_rt.asan_dbg_dll_thunk-{arch}`* | None |
| `/MD` | Either | Yes | *`clang_rt.asan_dbg_dynamic-{arch}`*, *`clang_rt.asan_dbg_dynamic_runtime_thunk-{arch}`* | *`clang_rt.asan_dbg_dynamic-{arch}`* |

The following diagram shows how the ASan library was linked for various compiler options before Visual Studio 2022 17.7 Preview 3:




# Function interception

The AddressSanitizer achieves function interception through many hotpatching techniques. These techniques are best documented within the source code itself.

The runtime libraries intercept many common memory management and memory manipulation functions. For a list, see AddressSanitizer list of intercepted functions. The allocation interceptors manage metadata and shadow bytes related to each allocation call. Every time a CRT function such as `malloc` or `delete` is called, the interceptors set specific values in the AddressSanitizer shadow-memory region to indicate whether those heap locations are currently accessible and what the bounds of the allocation are. These shadow bytes allow the compiler-generated checks of the shadow bytes to determine whether a load or store is valid.

Interception isn't guaranteed to succeed. If a function prologue is too short for a `jmp` to be written, interception can fail. If an interception failure occurs, the program throws a `debugbreak` and halts. If you attach a debugger, it makes the cause of the interception issue clear. If you have this problem, report a bug.

> **Note**
>
> Users can optionally attempt to continue past a failed interception by setting the environment variable `ASAN_WIN_CONTINUE_ON_INTERCEPTION_FAILURE` to any value. Continuing past an interception failure can result in missed bug reports for that function.

# Custom allocators and the AddressSanitizer runtime

The AddressSanitizer runtime provides interceptors for common allocator interfaces, `malloc`/`free`, `new`/`delete`, `HeapAlloc`/`HeapFree` (via `RtlAllocateHeap`/`RtlFreeHeap`). Many programs make use of custom allocators for one reason or another, an example would be any program using `dlmalloc` or a solution using the `std::allocator` interface and `VirtualAlloc()`. The compiler is unable to automatically add shadow-memory management calls to a custom allocator. It's the user's responsibility to use the provided manual poisoning interface. This API enables these allocators to function properly with the existing AddressSanitizer runtime and shadow byte conventions.

# Manual AddressSanitizer poisoning interface

The interface for enlightening is simple, but it imposes alignment restrictions on the user. Users may import these prototypes by importing sanitizer/asan_interface.h. Here are the interface function prototypes:

```
C++

void __asan_poison_memory_region(void const volatile *addr, size_t size);
void __asan_unpoison_memory_region(void const volatile *addr, size_t size);
```

For convenience, the AddressSanitizer interface header file provides wrapper macros. These macros check whether the AddressSanitizer functionality is enabled during compilation. They allow your source code to omit the poisoning function calls when they're not needed. These macros should be preferred over calling the above functions directly:

```
C++

#define ASAN_POISON_MEMORY_REGION(addr, size)
#define ASAN_UNPOISON_MEMORY_REGION(addr, size)
```

# Alignment requirements for AddressSanitizer poisoning

Any manual poisoning of shadow bytes must consider the alignment requirements. The user must add padding if necessary so the shadow bytes end on a byte boundary in the shadow memory. Each bit in the AddressSanitizer shadow memory encodes the state of a single byte in the application's memory. This encoding means the total size of each allocation, including any padding, must align to an 8-byte boundary. If the alignment requirement isn't satisfied, it can lead to incorrect bug reporting. The incorrect reporting could manifest as missing reports (false negatives) or reports on non-errors (false-positives).

For an illustration of the alignment requirement and potential issues, see the provided ASan alignment examples. One is a small program to show what can go wrong with manual shadow memory poisoning. The second is an example implementation of manual poisoning using the `std::allocator` interface.

# Run-time options

Microsoft C/C++ (MSVC) uses a runtime based on the Clang AddressSanitizer runtime from the llvm-project repository. Because of this, most runtime options are shared

between the two versions. A complete list of the public Clang runtime options is available here. We document some differences in the sections that follow. If you discover options that don't function as expected, report a bug.

## Unsupported AddressSanitizer options

- detect_container_overflow
- unmap_shadow_on_exit

> **Note**
>
> The AddressSanitizer runtime option `halt_on_error` doesn't function the way you might expect. In both the Clang and the MSVC runtime libraries, many error types are considered **non-continuable**, including most memory corruption errors.

For more information, see the Differences with Clang 12.0 section.

## MSVC-specific AddressSanitizer runtime options

- `windows_hook_legacy_allocators` Boolean, set to `false` to disable interception of GlobalAlloc and LocalAlloc allocators.

  > **Note**
  >
  > The option `windows_hook_legacy_allocators` wasn't available in the public llvm-project runtime when this article was written. The option may eventually be contributed back to the public project; however, it's dependent on code review and community acceptance.
  >
  > The option `windows_hook_rtl_allocators`, previously an opt-in feature while AddressSanitizer was experimental, is now enabled by default. In versions before Visual Studio 2022 version 17.4.6, the default option value is `false`. In Visual Studio 2022 version 17.4.6 and later versions, the option `windows_hook_rtl_allocators` defaults to `true`.

- `iat_overwrite` String, set to `"error"` by default. Other possible values are `"protect"` and `"ignore"`. Some modules may overwrite the import address table of other modules to customize implementations of certain functions. For example, drivers commonly provide custom implementations for specific hardware. The `iat_overwrite` option manages the AddressSanitizer runtime's protection against

overwrites for specific memoryapi.h functions. The runtime currently tracks the VirtualAlloc, VirtualProtect, and VirtualQuery functions for protection. This option is available in Visual Studio 2022 version 17.5 Preview 1 and later versions. The following `iat_overwrite` values control how the runtime reacts when protected functions are overwritten:
    - If set to `"error"` (the default), the runtime reports an error whenever an overwrite is detected.
    - If set to `"protect"`, the runtime attempts to avoid using the overwritten definition and proceeds. Effectively, the original `memoryapi` definition of the function is used from inside the runtime to avoid infinite recursion. Other modules in the process still use the overwritten definition.
    - If set to `"ignore"`, the runtime doesn't attempt to correct any overwritten functions and proceeds with execution.

- `windows_fast_fail_on_error` Boolean (false by default), set to `true` to enable the process to terminate with a __fastfail(71) after printing the error report.

> **Note**
>
> When abort_on_error value is set to true, on Windows the program terminates with an exit(3). In order to not change current behavior we decided to introduce this new option instead. If both abort_on_error and windows_fast_fail_on_error are true, the program will exit with the __fastfail.

# AddressSanitizer list of intercepted functions (Windows)

The AddressSanitizer runtime hotpatches many functions to enable memory safety checks at runtime. Here's a non-exhaustive list of the functions that the AddressSanitizer runtime monitors.

## Default interceptors

- __C_specific_handler (x64 only)
- _aligned_free
- _aligned_malloc
- _aligned_msize
- _aligned_realloc
- _calloc_base

- _calloc_crt
- _calloc_dbg (debug runtime only)
- _except_handler3 (x86 only)
- _except_handler4 (x86 only) (undocumented)
- _expand
- `_expand_base` (undocumented)
- _expand_dbg (debug runtime only)
- _free_base (undocumented)
- _free_dbg (debug runtime only)
- _malloc_base (undocumented)
- `_malloc_crt` (undocumented)
- _malloc_dbg (debug runtime only)
- _msize
- `_msize_base` (undocumented)
- _msize_dbg (debug runtime only)
- _realloc_base (undocumented)
- `_realloc_crt` (undocumented)
- _realloc_dbg (debug runtime only)
- _recalloc
- `_recalloc_base` (undocumented)
- `_recalloc_crt` (undocumented)
- _recalloc_dbg (debug runtime only)
- _strdup
- atoi
- atol
- calloc
- CreateThread
- free
- frexp
- longjmp
- malloc
- memchr
- memcmp
- memcpy
- memmove
- memset
- RaiseException
- realloc
- RtlAllocateHeap
- RtlCreateHeap

- RtlDestroyHeap
- RtlFreeHeap
- RtlRaiseException
- `RtlReAllocateHeap` (undocumented)
- `RtlSizeHeap` (undocumented)
- SetUnhandledExceptionFilter
- strcat
- strchr
- strcmp
- strcpy
- strcspn
- strdup
- strlen
- strncat
- strncmp
- strncpy
- strnlen
- strpbrk
- strspn
- strstr
- strtok
- strtol
- wcslen
- wcsnlen

## Optional interceptors

The interceptors listed here are only installed when an AddressSanitizer runtime option is enabled. Set `windows_hook_legacy_allocators` to `false` to disable legacy allocator interception. `set ASAN_OPTIONS=windows_hook_legacy_allocators=false`

- GlobalAlloc
- GlobalFree
- GlobalHandle
- GlobalLock
- GlobalReAlloc
- GlobalSize
- GlobalUnlock
- LocalAlloc
- LocalFree

- LocalHandle
- LocalLock
- LocalReAlloc
- LocalSize
- LocalUnlock

## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

## Feedback

# Visual Studio AddressSanitizer extended functionality library (VCAsan)

Article • 07/25/2023

The `VCAsan*.lib` libraries implement extended debugger IDE features in Visual Studio. These features allow the IDE to show AddressSanitizer errors in live debug sessions, or offline by saving a crash dump file with metadata. The library is linked any time AddressSanitizer is enabled by the MSVC compiler.

## VCAsan library inventory

| Runtime option | VCAsan link library |
|---|---|
| `/MT` | `libvcasan.lib` |
| `/MD` | `vcasan.lib` |
| `/MTd` | `libvcasand.lib` |
| `/MDd` | `vcasand.lib` |

## VCAsan library features

### Rich AddressSanitizer error report window in Visual Studio IDE

The VCAsan library registers a callback within the AddressSanitizer runtime by using the interface function __asan_set_error_report_callback. If an AddressSanitizer report is generated, this callback gets used to throw an exception that's caught by Visual Studio. Visual Studio uses the exception data to generate the message that's displayed to the user in the IDE.

> **Note**
>
> The VCAsan library registers a callback function in the AddressSanitizer runtime. If your code calls this registration function a second time, it overwrites the VCAsan library callback registration. This results in the loss of all Visual Studio IDE integration. You'll revert back to the default IDE user experience. It's also possible

for a user's call that registers their callback to be lost. If you encounter either problem, file a **bug**.

## Save AddressSanitizer errors in a new type of crash dump file

When you link the VCAsan library to your executable, users can enable it to generate a crash dump at runtime. Then the AddressSanitizer runtime produces a dump file when a diagnosed error occurs. To enable this feature, the user sets the `ASAN_SAVE_DUMPS` environment variable by using a command such as this one:

```
set ASAN_SAVE_DUMPS=MyFileName.dmp
```

The file must have a `.dmp` extension to follow the Visual Studio IDE conventions. (Prior to 17.7)

Here's what happens when a dump file is specified for `ASAN_SAVE_DUMPS`: If an error gets caught by the AddressSanitizer runtime, it saves a crash dump file that has the metadata associated with the error. The debugger in Visual Studio version 16.9 and later can parse the metadata that's saved in the dump file. You can set `ASAN_SAVE_DUMPS` on a per-test basis, store these binary artifacts, and then view them in the IDE with proper source indexing.

Visual Studio version 17.7 and later supports the following:

- Quoted strings are now handled correctly. In previous versions, for environments inside of Visual Studio or when using PowerShell, setting the environment variable to contain quotes or spaces would fail to create the expected dump file.
- When the `.dmp` extension is explicitly specified (for example, `set ASAN_SAVE_DUMPS=MyDmp.dmp`), VCAsan uses it explicitly, and will not add an associated process ID to the dump file name.
- If a `.dmp` file already exists with the same name specified from the environment variable, VCAsan modifies the file name as follows:
  - Appends a number to the filename in parentheses. For example, `Myfile(1).dmp`.
  - If after several attempts appending a number in parentheses fails to generate a unique name, the file is saved to an `%APPLOCAL%` temporary path that VCAsan will print. For example, `C:\Users\~\AppData\Local\Temp\Dump.dmp`.
  - If saving to a temporary path fails, a diagnostic error is displayed.

# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer error examples

# AddressSanitizer shadow bytes

Article • 09/10/2024

We briefly summarize the concept of shadow bytes and how they can be used by the runtime implementation of /fsanitize=address. For further details, we refer you to the initial research AddressSanitizer - Serebryany, et al and the current AddressSanitizer algorithm documentation.

## Core concept

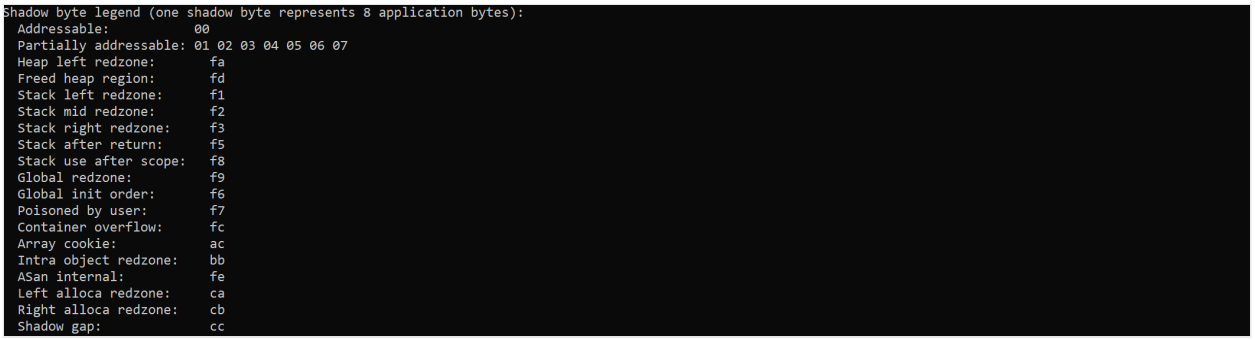
Every **8 bytes** in your application's virtual address space can be described using **one shadow byte**.

One shadow byte describes how many bytes are currently accessible as follows:

- 0 means all 8 bytes
- 1-7 means one to seven bytes
- Negative numbers encode context for the runtime to use for reporting diagnostics.

### Shadow byte legend

Consider this shadow byte legend where all negative numbers are defined:




## Mapping - describing your address space

Every 8 bytes in the application's virtual address space that's "0-mod-8" aligned can be mapped to the shadow byte that describes that slot in the virtual address space. This mapping can be accomplished with a **simple shift and add**.

On x86:

```
C++
```

```
char shadow_byte_value = *((Your_Address >> 3) + 0x30000000)
```

On x64:

```
C++

char shadow_byte_value = *((Your_Address >> 3) +
_asan_runtime_assigned_offset)
```

# Code generation - tests

Consider how specific shadow bytes might get written, either by the compiler-generated code, static data, or the runtime. This pseudo-code shows how it's possible to generate a check that precedes any load or store:

```
C++

ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0) {
    ReportAndCrash(Addr);
}
```

When instrumenting a memory reference that's less than 8 bytes wide, the instrumentation is slightly more complex. If the shadow value is positive (meaning only the first k bytes in the 8-byte word can be accessed), we need to compare the last 3 bits of the address with k.

```
C++

ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k)) {
    ReportAndCrash(Addr);
}
```

The runtime and the compiler-generated code both write shadow bytes. These shadow bytes either allow or revoke access when scopes end or storage is freed. The checks above read shadow bytes that describe 8-byte "slots" in your application's address space, at a certain time in the program's execution. Besides these explicitly generated checks, the runtime also checks shadow bytes after it intercepts (or "hooks") many functions in the CRT.

For more information, see the list of intercepted functions.

# Setting shadow bytes

Both the code the compiler generates and the AddressSanitizer runtime can write shadow bytes. For example, the compiler can set shadow bytes to allow fixed sized access to stack locals defined in an inner scope. The runtime can surround global variables in the data section with shadow bytes.

# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

---

# Feedback

# AddressSanitizer cloud or distributed testing

Article • 08/03/2021

You don't have to debug AddressSanitizer errors when and where they occur. Configure the runtime to create a crash dump that stores all the AddressSanitizer-specific context when an error happens. Then send that crash dump to another PC for debugging. Offline debugging can be a critical timesaver when running AddressSanitizer in the cloud or in distributed testing. You can create the dump on test or production infrastructure where the failure occurs, and debug it later on your developer PC.

The Visual Studio debugger provides precisely diagnosed AddressSanitizer errors. You can view these bugs without having to rerun tests, copy huge datasets, discover lost data, or find test machines that went offline. You only need to load the crash dump.

Crash dumps are created upon AddressSanitizer failures by setting the following environment variable:

```
set ASAN_SAVE_DUMPS=MyFileName.dmp
```

> **Note**
>
> The file name must have a suffix `.dmp` to follow Visual Studio naming conventions.

This dump file can be displayed by using Visual Studio at a later date on another machine.

Visual Studio can display the error information in the context of the original source code. To do so, Visual Studio requires debugging symbols and indexed source code. For the best debugging experience, the EXE, PDB, and source code used to produce those binaries must match.

For more information about storing sources and symbols, see the source and symbols section. For information about implementation details and fine grained control, see debugger integration.

## Example - build, test, and analyze

Consider three machines: A, B, and C. Builds are done on machine B, tests are run on machine C, and you analyze failures on machine A. The errors are reported against

source line and column numbers in your source code. You can see the call stack together with a set of symbols in the PDB file produced using that exact version of the source code.

The following steps are for local or distributed scenarios that lead to creation of a .dmp file, and for viewing that AddressSanitizer dump file offline.

## Produce a .dmp locally

- Build
- Test the executable
- Copy a generated .dmp file to the build directory
- Open the .dmp file with the paired .pdb, in the same directory

## Produce a .dmp on a distributed system

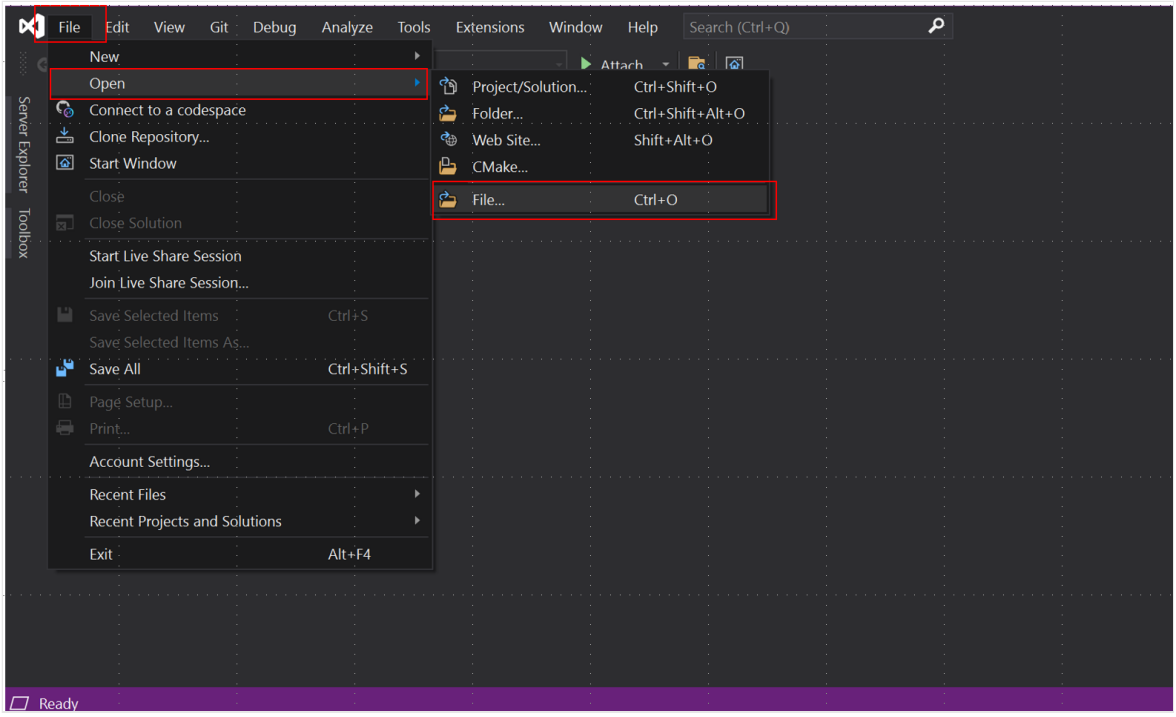
- Build and post-process the PDB for source indexing data blocks
- Copy the atomic pair of (.exe, .pdb) to the test machine and run tests
- Write the atomic pairs of (.pdb, .dmp) to the bug-reporting database
- Visual Studio opens a .dmp file with the paired .pdb, in the same directory

> **Note**
>
> The Visual Studio 2019 machine you use for analysis must have access to GitHub or the internal `\\Machine\share` where your indexed source is stored.

# View AddressSanitizer .dmp files

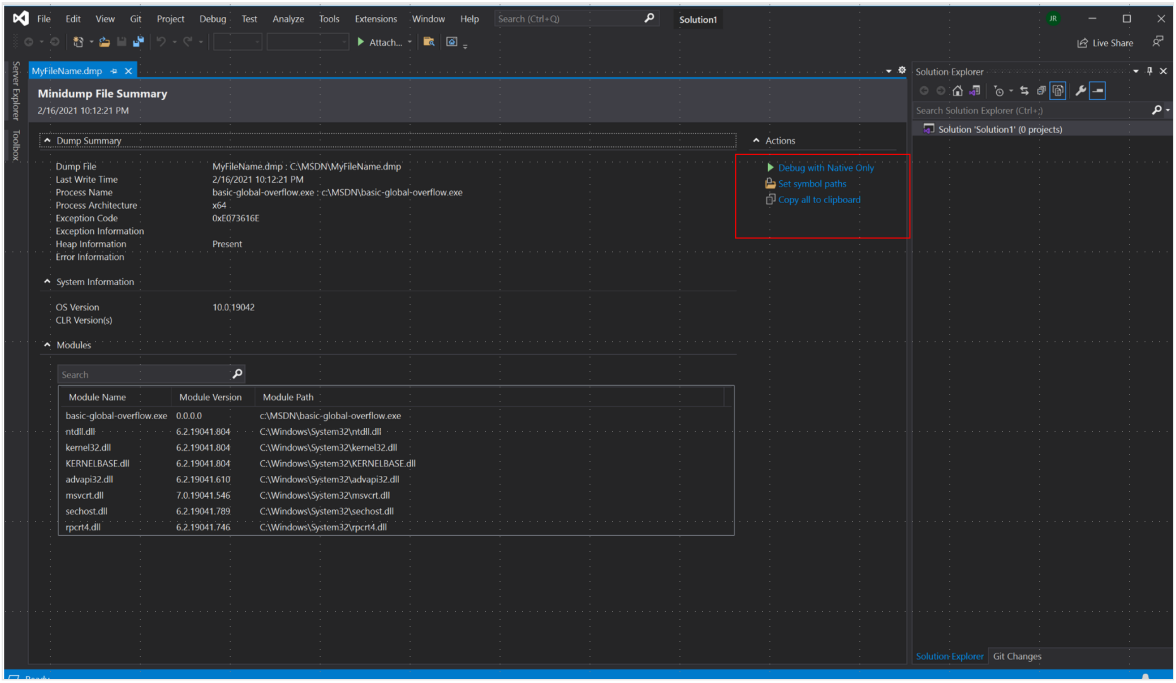
1. Make sure the debugger IDE can find your PDB and source files.

2. Open Visual Studio and select **Continue without code**. Then select **File** > **Open** > **File** to open the File Open dialog. Make sure the file name suffix is **.dmp**.

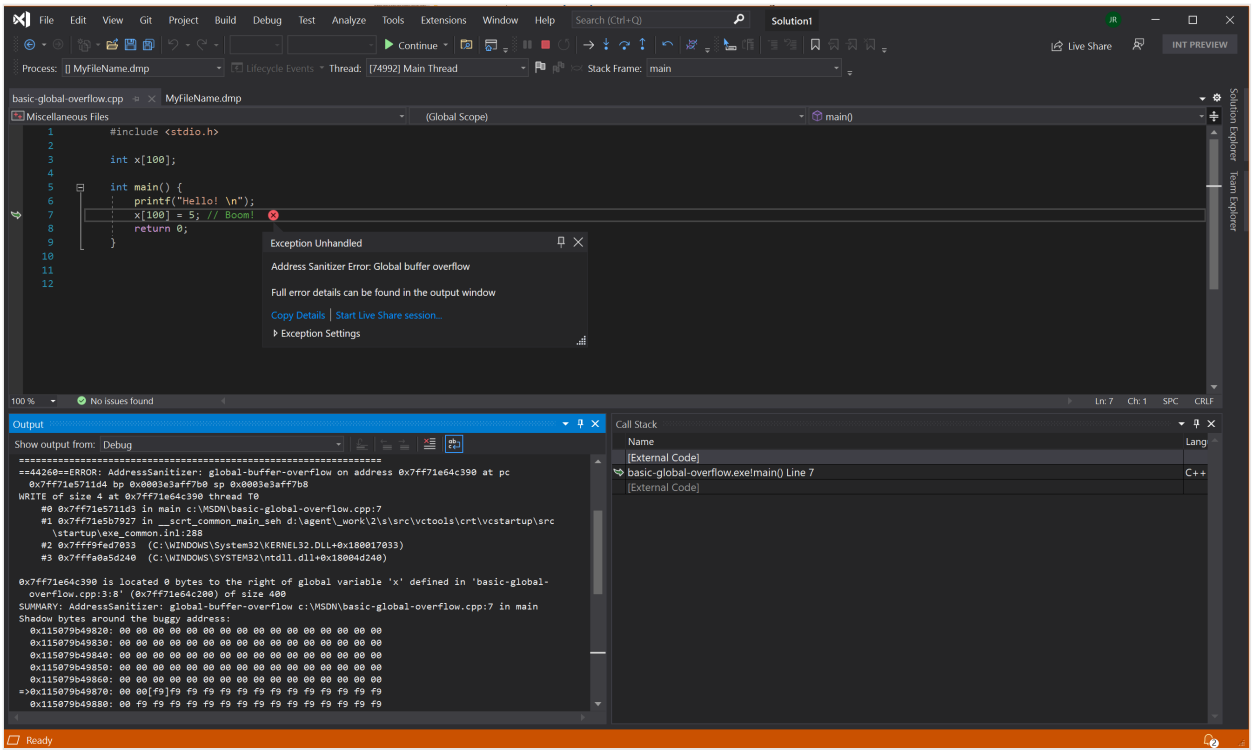



The screen shown here needs one more step to enable the IDE access to symbols and source.

3. Set the symbol paths, and then choose **Debug with Native Only**.




This screenshot shows the final loaded dump file, with sources and AddressSanitizer metadata loaded.




# Source and symbols

Source server lets a client retrieve the **exact version** of the source files used to build an application. The source code for an executable or DLL can change over time, and between versions. You can use it to look at the same source code that built a particular version of the application.

While debugging an EXE with its PDB file, the debugger can use the embedded source server data block to retrieve the appropriate files from source control. It loads the files that map to the fully qualified names put in the PDB automatically by the `/Zi` compiler option.

To use source server, the application must be "source indexed" by using `pdbstr.exe` to write a `srcsrv` data block into your PDB file. For more information, see the Data Block section of Source server and source indexing. You'll find the steps to index sources and publish symbols and how to specify symbols and source code for the debugger useful, too.

For external documentation, see:

- Source indexing with Git
- A guide to easier debugging
- Source Indexing is Underused Awesomeness

# See also

# AddressSanitizer error examples

Article • 05/02/2022

We list a subset of the errors supported by AddressSanitizer in Microsoft C/C++ (MSVC) in this section. This list is not an exhaustive error list. It's meant to show several kinds of errors you'll see in AddressSanitizer. In each article, we've included example code with build instructions and screenshots of the debugger in action. They'll help you learn to use the AddressSanitizer features supported by MSVC in your code. All screenshots were generated by using `devenv.exe /debugexe example.exe`. Some of these examples are based on sample code in the LLVM compiler-rt test suite.

## Build the error examples

Each error example provides source code and compilation instructions for a command-line build. To build each example, open a developer command prompt. Create a folder for your example project, then make it the current directory. Then copy the example code into a source file with the appropriate name, such as *`example1.cpp`*. Follow the build instructions to generate and run the instrumented code in the debugger.

## Errors with examples

- Error: alloc-dealloc-mismatch
- Error: allocation-size-too-big
- Error: calloc-overflow
- Error: container-overflow
- Error: double-free
- Error: dynamic-stack-buffer-overflow
- Error: global-buffer-overflow
- Error: heap-buffer-overflow
- Error: heap-use-after-free
- Error: invalid-allocation-alignment
- Error: memcpy-param-overlap

- Error: new-delete-type-mismatch
- Error: stack-buffer-overflow
- Error: stack-buffer-underflow
- Error: stack-use-after-return
- Error: stack-use-after-scope
- Error: strncat-param-overlap
- Error: use-after-poison

## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration

# Error: `alloc-dealloc-mismatch`

Article • 08/03/2021

> Address Sanitizer Error: Mismatch between allocation and deallocation APIs

The `alloc`/`dealloc` mismatch functionality in AddressSanitizer is off by default for Windows. To enable it, run `set ASAN_OPTIONS=alloc_dealloc_mismatch=1` before running the program. This environment variable is checked at runtime to report errors on `malloc`/`delete`, `new`/`free`, and `new`/`delete[]`.

## Example

C++

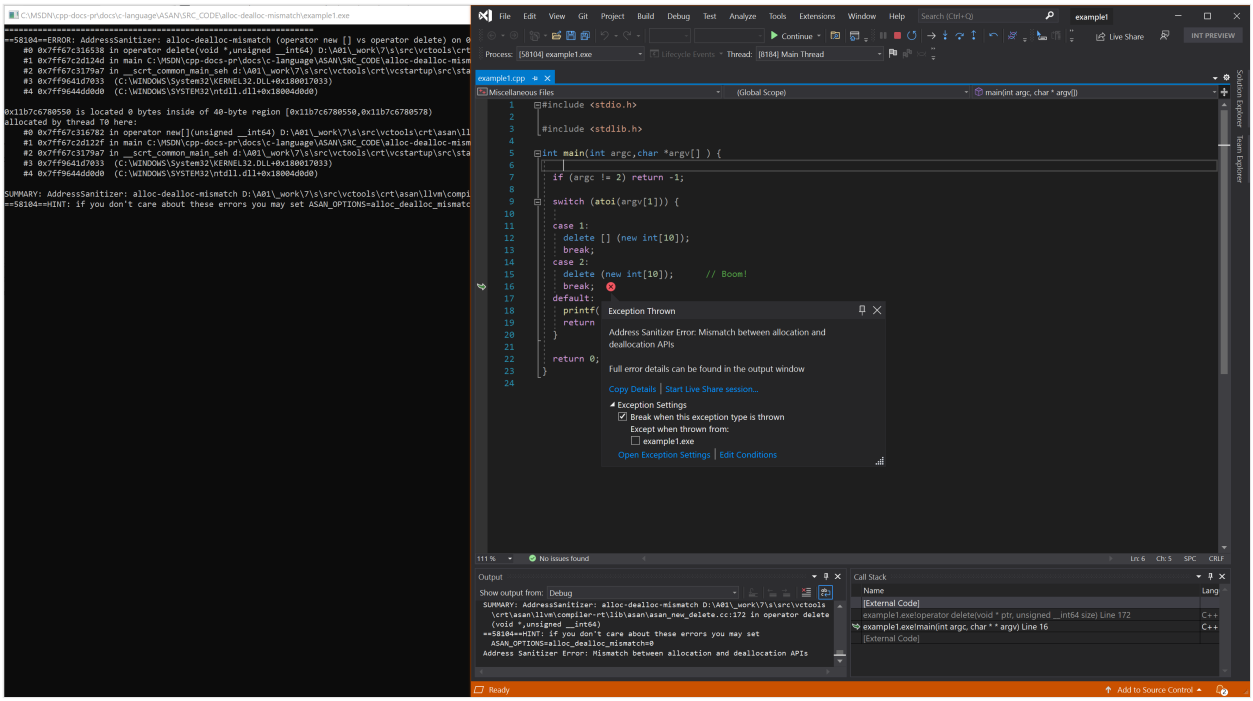
```
// example1.cpp
// alloc-dealloc-mismatch error
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {

    if (argc != 2) return -1;

    switch (atoi(argv[1])) {

    case 1:
        delete[](new int[10]);
        break;
    case 2:
        delete (new int[10]);      // Boom!
        break;
    default:
        printf("arguments: 1: no error 2: runtime error\n");
        return -1;
    }

    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
set ASAN_OPTIONS=alloc_dealloc_mismatch=1
devenv /debugexe example1.exe 2
```

## Resulting error




## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Error: `allocation-size-too-big`

Article • 08/03/2021

> Address Sanitizer Error: allocation-size-too-big

This example shows the error found when an allocation is too large for the heap. Example sourced from LLVM compiler-rt test suite.

## Example

C++

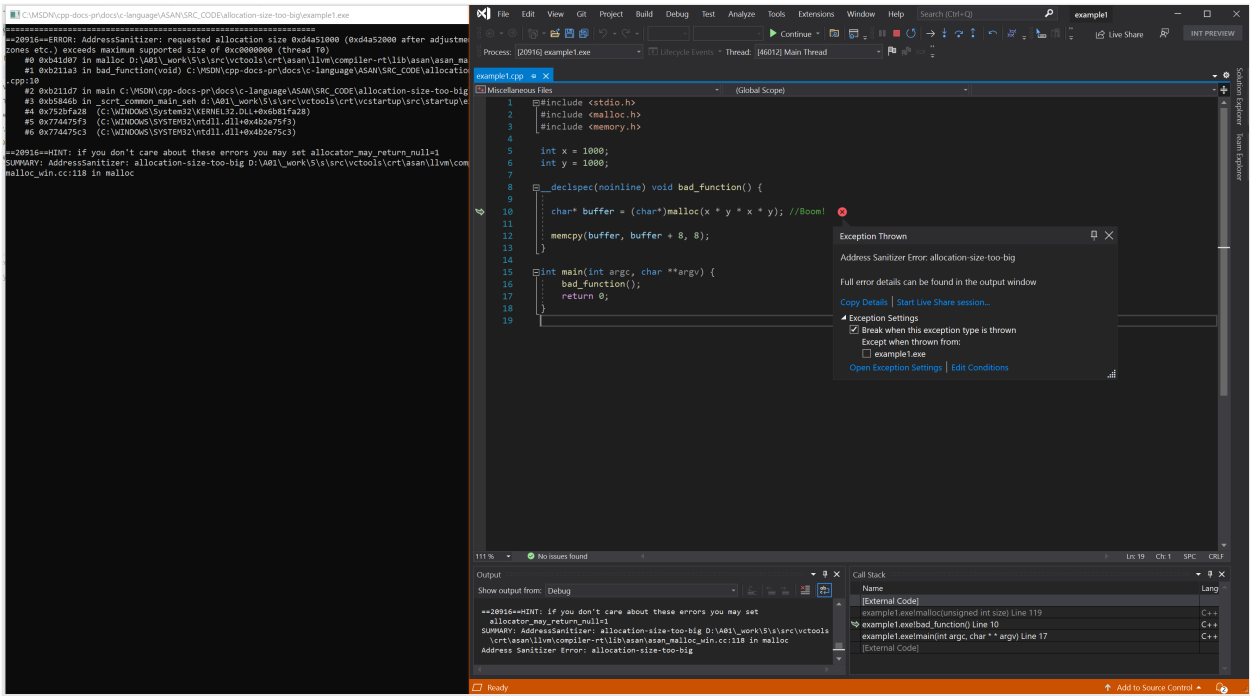
```
// example1.cpp
// allocation-size-too-big error
#include <stdio.h>
#include <malloc.h>
#include <memory.h>

int x = 1000;
int y = 1000;

__declspec(noinline) void bad_function() {

  char* buffer = (char*)malloc(x * y * x * y); //Boom!

  memcpy(buffer, buffer + 8, 8);
}

int main(int argc, char **argv) {
    bad_function();
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# Error: `calloc-overflow`

Article • 08/03/2021

> Address Sanitizer Error: calloc-overflow

The CRT function calloc creates an array in memory with elements initialized to 0. The arguments can create an internal error that leads to a NULL pointer as the return value.

## Example

C++

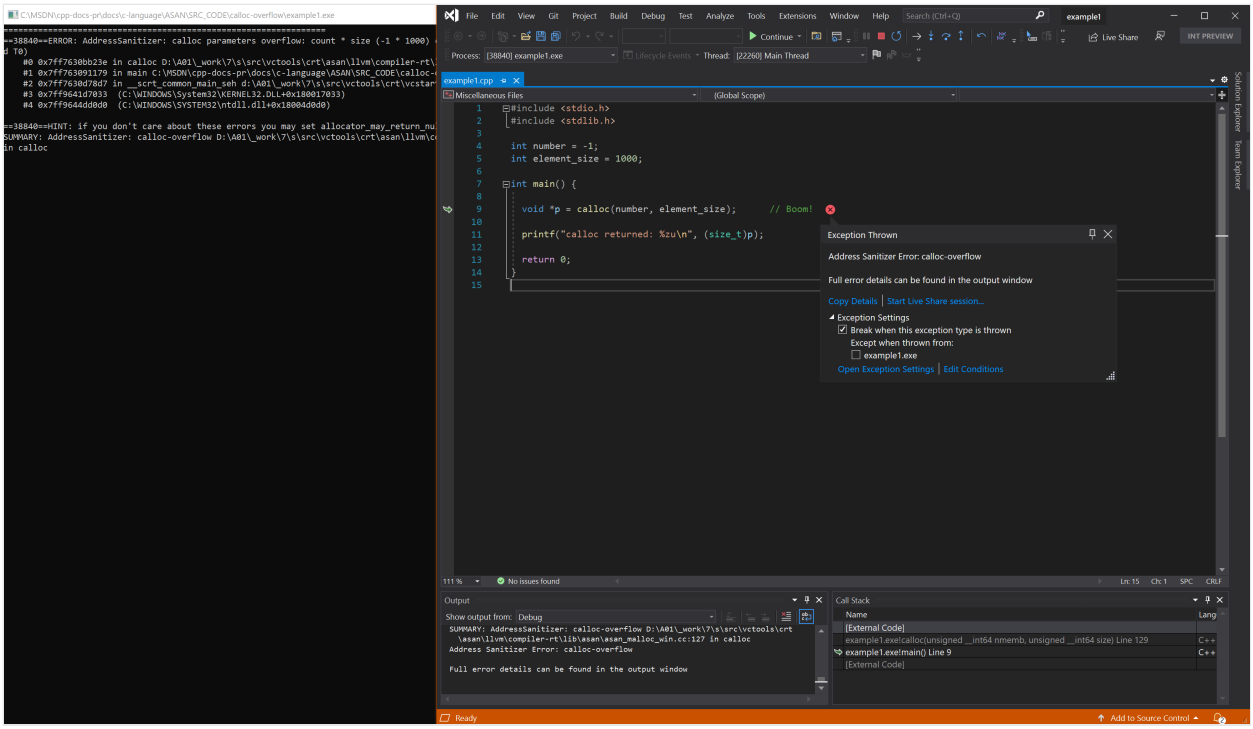
```
// example1.cpp
// calloc-overflow error
#include <stdio.h>
#include <stdlib.h>

int number = -1;
int element_size = 1000;

int main() {

    void *p = calloc(number, element_size);      // Boom!

    printf("calloc returned: %zu\n", (size_t)p);

    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# Error: `container-overflow`

Article • 07/03/2023

> Address Sanitizer Error: Container overflow

In Visual Studio 2022 version 17.2 and later, the Microsoft Visual C++ standard library (STL) is partially enlightened to work with the AddressSanitizer. The following container types have annotations to detect memory access issues:

| Standard container type | Disable annotations macro | Supported in version |
|---|---|---|
| `std::vector` | `_DISABLE_VECTOR_ANNOTATION` | Visual Studio 2022 17.2 |
| `std::string` | `_DISABLE_STRING_ANNOTATION` | Visual Studio 2022 17.6 |

There are checks to ensure that there are no one-definition-rule (ODR) violations. An ODR violation occurs when one translation unit annotates a standard type, such as `vector`, with ASan annotations, but another translation unit doesn't. In this example, the linker might simultaneously see one declaration of `vector<int>::push_back` that has address sanitizer annotations, and another declaration of `vector<int>::push_back` that doesn't. To avoid this problem, each static library and object used to link the binary must also enable ASan annotations. Effectively, you must build those static libraries and objects with AddressSanitizer enabled. Mixing code with different annotation settings causes an error:

Output

```
my_static.lib(my_code.obj) : error LNK2038: mismatch detected for
'annotate_vector': value '0' doesn't match value '1' in main.obj
```

To resolve this error, either disable annotations in all projects that use the corresponding macro, or build each project using `/fsanitize=address` and annotations enabled. (Annotations are enabled by default.)

# Example: Access reserved memory in a `std::vector`

C++

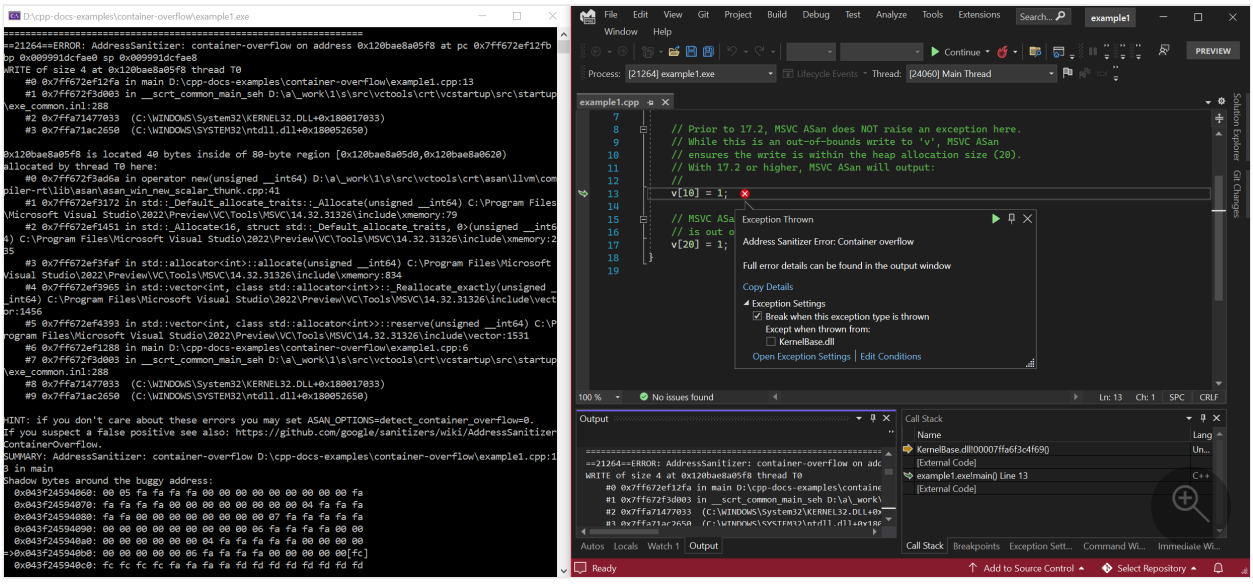
```
// Compile with: cl /EHsc /fsanitize=address /Zi
#include <vector>

int main() {
    // Create a vector of size 10, but with a capacity of 20.
    std::vector<int> v(10);
    v.reserve(20);

    // In versions prior to 17.2, MSVC ASan does NOT raise an exception
here.
    // While this is an out-of-bounds write to 'v', MSVC ASan
    // ensures the write is within the heap allocation size (20).
    // With 17.2 and later, MSVC ASan will raise a 'container-overflow'
exception:
    // ==18364==ERROR: AddressSanitizer: container-overflow on address
0x1263cb8a0048 at pc 0x7ff6466411ab bp 0x005cf81ef7b0 sp 0x005cf81ef7b8
    v[10] = 1;

    // Regardless of version, MSVC ASan DOES raise an exception here, as
this write
    // is out of bounds from the heap allocation.
    v[20] = 1;
}
```

To build and test this example, run the following commands in a Visual Studio 2022 version 17.2, or later Developer command prompt window:

Windows Command Prompt

```
cl /EHsc example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Error result of reserved memory access in a `std::vector`



# Custom allocators and container overflow

Address Sanitizer container overflow checks support non-`std::allocator` allocators. However, because AddressSanitizer doesn't know whether a custom allocator conforms to AddressSanitizer requirements such as aligning allocations on 8-byte boundaries, or not putting data between the end of the allocation and the next 8-byte boundary, it may not always be able to check that accesses on the latter end of an allocation are correct.

AddressSanitizer marks blocks of memory in 8-byte chunks. It can't place inaccessible bytes before accessible bytes in a single chunk. It's valid to have 8 accessible bytes in a chunk, or 4 accessible bytes followed by 4 inaccessible bytes. Four inaccessible bytes can't be followed by 4 accessible bytes.

If the end of an allocation from a custom allocator doesn't strictly align with the end of an 8-byte chunk, AddressSanitizer must assume that the allocator makes the bytes between the end of the allocation and the end of the chunk available to the allocator or the user to write to. Therefore, it can't mark the bytes in the final 8-byte chunk as inaccessible. In the following example of a `vector` that allocates memory using a custom allocator, '?' refers to uninitialized data and '-' refers to memory that is inaccessible.

C++

```
std::vector<uint8_t, MyCustomAlloc<uint8_t>> v;
v.reserve(20);
v.assign({0, 1, 2, 3});
// the buffer of `v` is as follows:
//    | v.data()
//    |       | v.data() + v.size()
//    |       |                                       | v.data() +
v.capacity()
//  [ 0 1 2 3 ? ? ? ? ][ ? ? ? ? ? ? ? ? ][ ? ? ? ? - - - - ]
//        chunk 1            chunk 2            chunk 3
```

In the previous example, chunk 3 has 4 bytes of memory that are assumed to be inaccessible because they fall between the end of the allocation of the 20 bytes that were reserved (`v.reserve(20)`) and the end of the third logical grouping of 8 bytes (remember that AddressSanitizer marks blocks of memory in 8-byte chunks).

Ideally, we'd mark the shadow memory, which Address Sanitizer sets aside for every 8-byte block of memory to track which bytes in that 8-byte block are valid and which are invalid (and why), such that `v.data() + [0, v.size())` are accessible, and `v.data() + [v.size(), v.capacity())` are inaccessible. Note the use of interval notation here: '[' means inclusive of, and ')' means exclusive of. If the user is using a custom allocator, we

don't know whether the memory after `v.data() + v.capacity()` is accessible or not. We must assume that it is. We'd prefer to mark those bytes as inaccessible in the shadow memory, but we must mark them as accessible so that it remains possible to access those bytes after the allocation.

`std::allocator` uses the `_Minimum_asan_allocation_alignment` static member variable to tell `vector` and `string` that they can trust the allocator not to put data right after the allocation. This ensures that the allocator won't use the memory between the end of the allocation and end of the chunk. Thus that part of the chunk can be marked inaccessible by the Address Sanitizer to catch overruns.

If you want the implementation to trust that your custom allocator is handling the memory between the end of the allocation and the end of the chunk so that it can mark that memory as inaccessible and catch overruns, set `_Minimum_asan_allocation_alignment` to your actual minimum alignment. For AddressSanitizer to work correctly, the alignment must be at least 8.

## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Error: `double-free`

Article • 08/03/2021

> Address Sanitizer Error: Deallocation of freed memory

In C, you can call `free` erroneously. In C++, you can call `delete` more than once. In these examples, we show errors with `delete`, `free`, and `HeapCreate`.

## Example C++ - double `operator delete`

C++

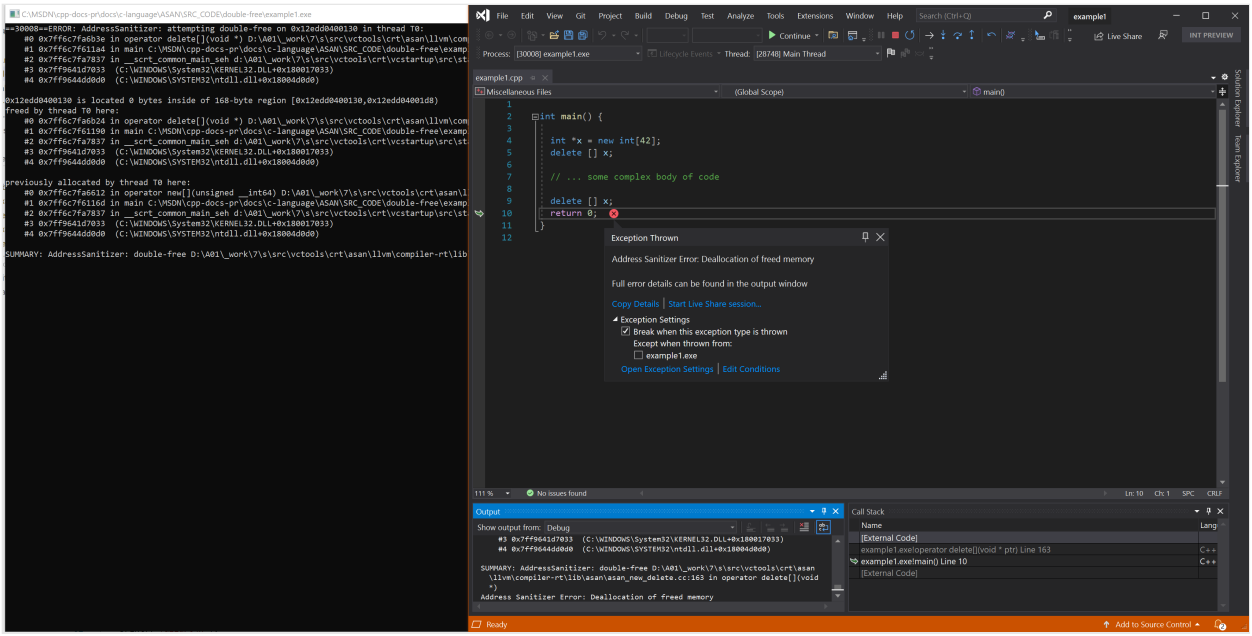
```
// example1.cpp
// double-free error
int main() {

    int *x = new int[42];
    delete [] x;

    // ... some complex body of code

    delete [] x;
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error - double `operator delete`



# Example 'C' - double `free`

C++

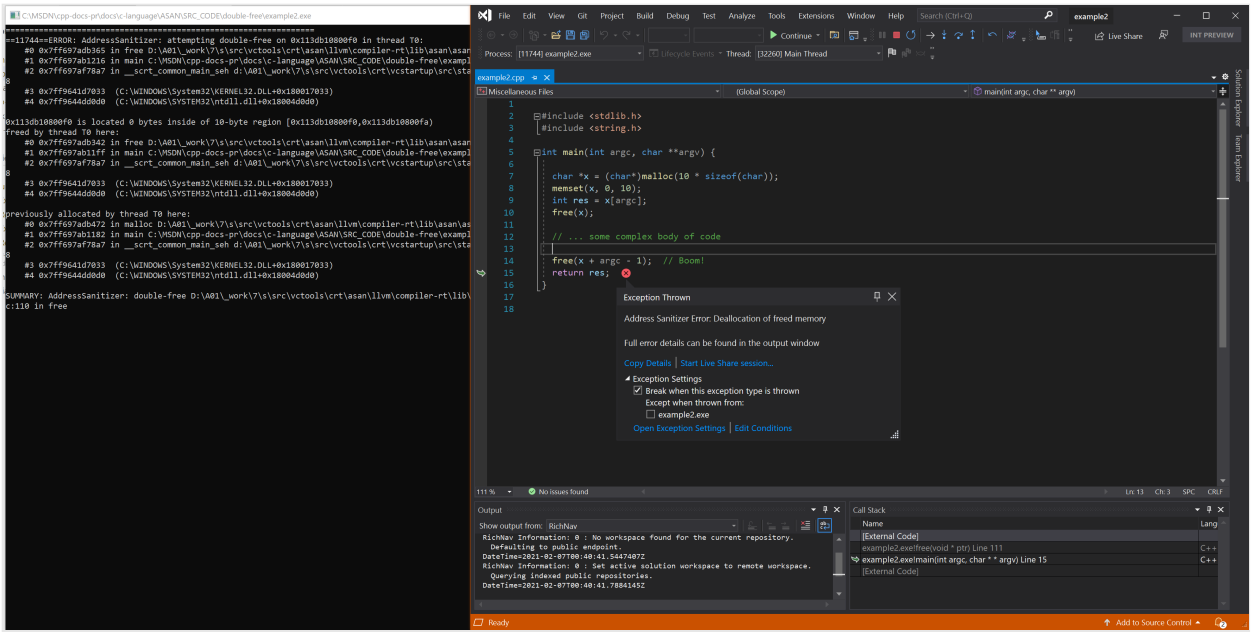
```
// example2.cpp
// double-free error
#include <stdlib.h>
#include <string.h>

int main(int argc, char** argv) {

    char* x = (char*)malloc(10 * sizeof(char));
    memset(x, 0, 10);
    int res = x[argc];
    free(x);

    // ... some complex body of code

    free(x + argc - 1);  // Boom!
    return res;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe
```

# Resulting error - double `free`




# Example - Windows `HeapCreate` double `HeapFree`

C++

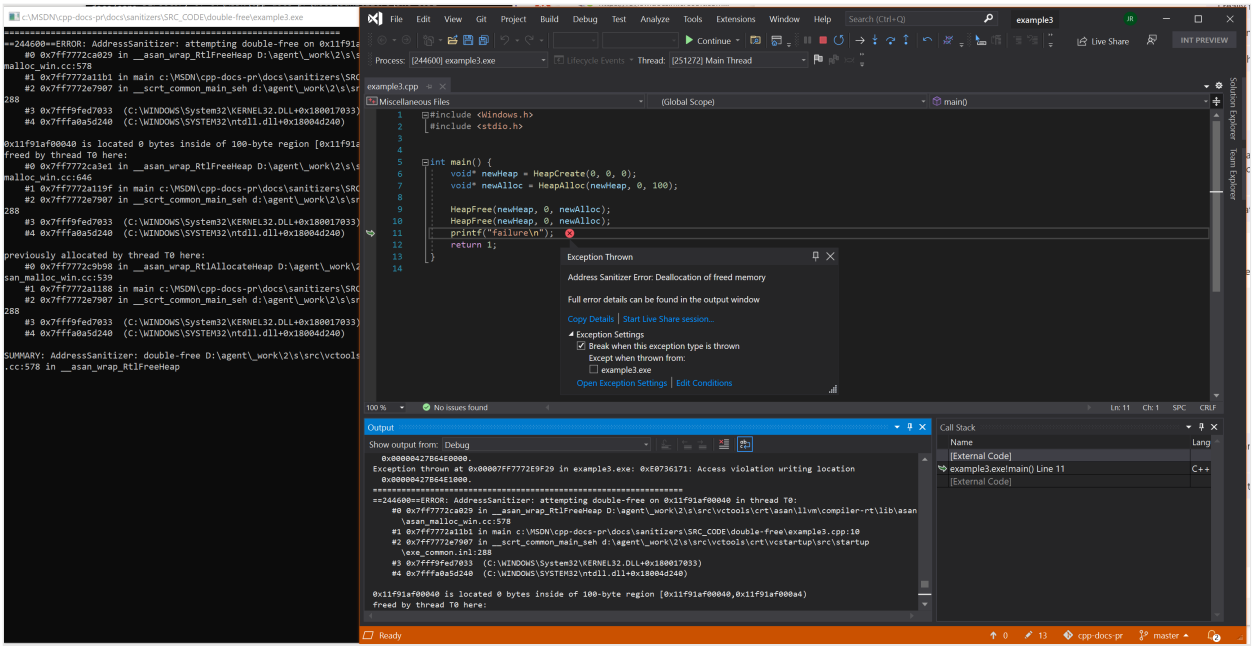
```
// example3.cpp
// double-free error
#include <Windows.h>
#include <stdio.h>

int main() {
    void* newHeap = HeapCreate(0, 0, 0);
    void* newAlloc = HeapAlloc(newHeap, 0, 100);

    HeapFree(newHeap, 0, newAlloc);
    HeapFree(newHeap, 0, newAlloc);
    printf("failure\n");
    return 1;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi
devenv /debugexe example3.exe
```

# Resulting error - Windows `HeapCreate` double `HeapFree`




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# Error: `dynamic-stack-buffer-overflow`

Article • 08/03/2021

> Address Sanitizer Error: dynamic-stack-buffer-overflow

This example shows the error that results from a buffer access outside the bounds of a stack-allocated object.

## Example - `alloca` overflow (right)

C++

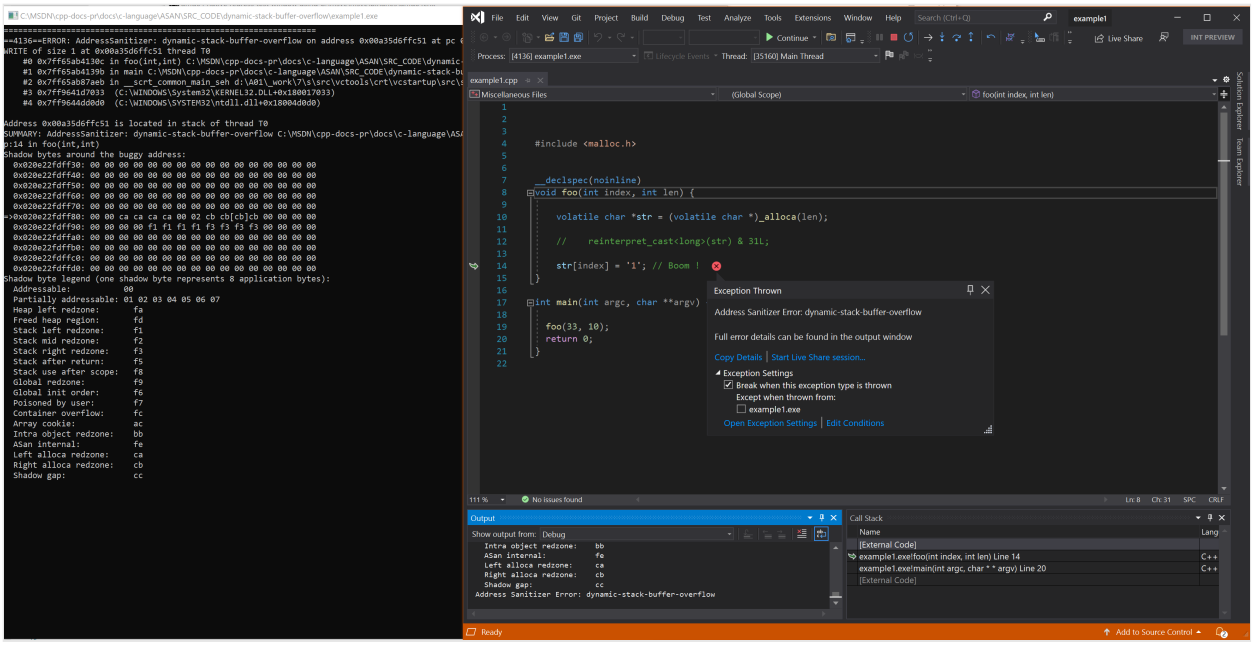
```
// example1.cpp
// dynamic-stack-buffer-overflow error
#include <malloc.h>

__declspec(noinline)
void foo(int index, int len) {

    volatile char *str = (volatile char *)_alloca(len);

    //    reinterpret_cast<long>(str) & 31L;

    str[index] = '1'; // Boom !
}

int main(int argc, char **argv) {

    foo(33, 10);
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error



# Example - `alloca` overflow (left)

C++

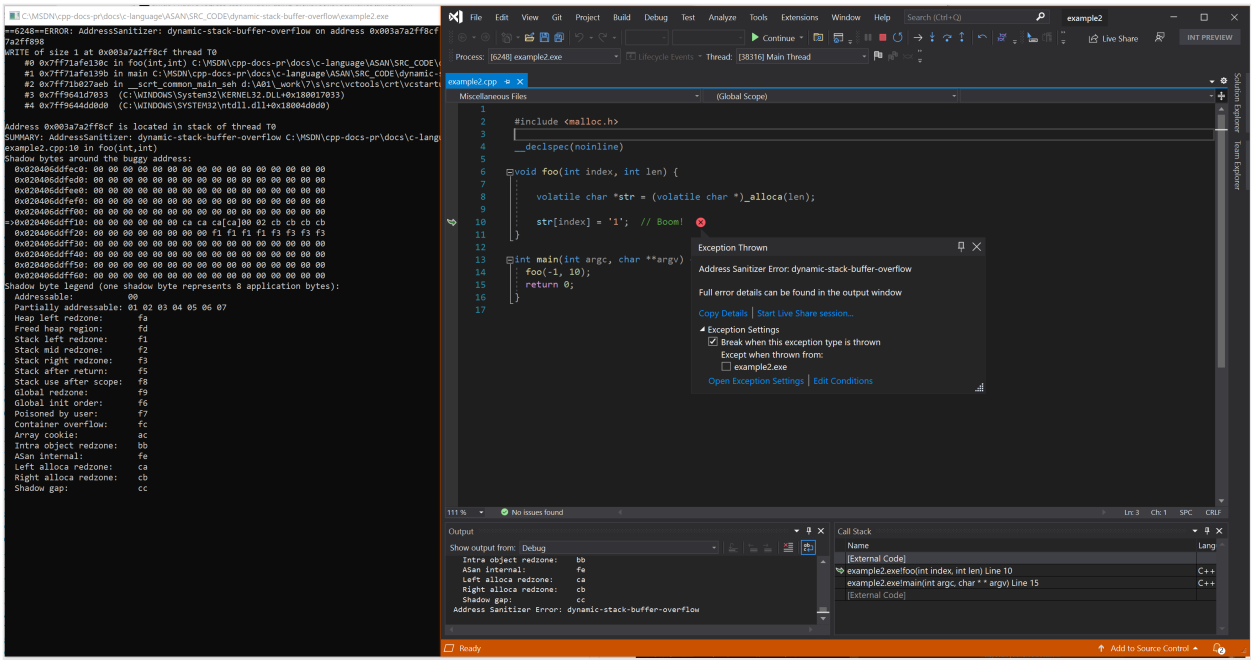
```
// example2.cpp
// dynamic-stack-buffer-overflow error
#include <malloc.h>

__declspec(noinline)
void foo(int index, int len) {

    volatile char *str = (volatile char *)_alloca(len);

    str[index] = '1';  // Boom!
}

int main(int argc, char **argv) {
    foo(-1, 10);
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe
```

# Resulting error - `alloca` overflow (left)




# Example - several calls to `alloca`

C++

```
// example3.cpp
// dynamic-stack-buffer-overflow error
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#define SIZE 7
extern void nothing();
int x=13,*aa,*bb,y=0;
int fail = 0;
int tmp;

int main()
{
    int* cc;
    int i;
    int k = 17;
    __try {
        tmp = k;
        aa = (int*)_alloca(SIZE * sizeof(int));
        if (((int)aa) & 0x3)
            fail = 1;
        for (i = 0; i < SIZE; i++) {
            aa[i] = x + 1 + i;
        }
        bb = (int*)_alloca(x * sizeof(int));
        if (((int)bb) & 0x3)
            fail = 1;

        for (i = 0; i < x; i++) {
```

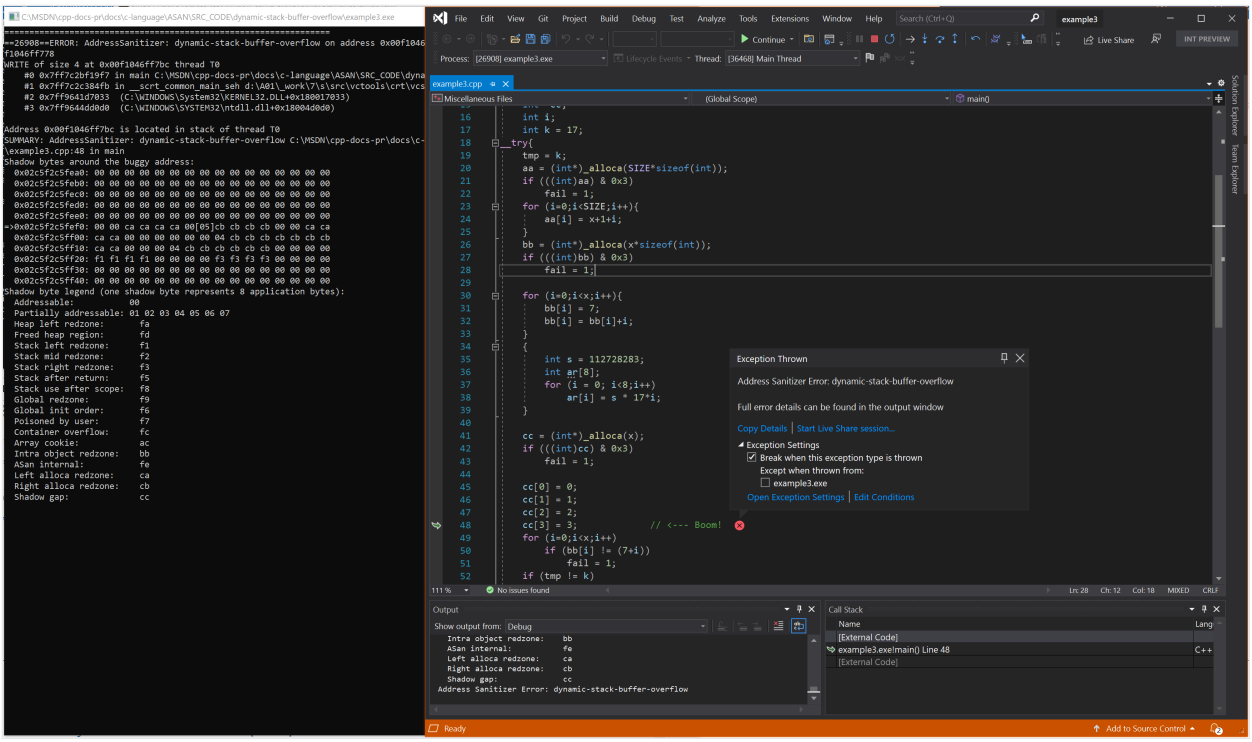
```
            bb[i] = 7;
            bb[i] = bb[i] + i;
        }
        {
            int s = 112728283;
            int ar[8];
            for (i = 0; i < 8; i++)
                ar[i] = s * 17 * i;
        }

        cc = (int*)_alloca(x);
        if (((int)cc) & 0x3)
            fail = 1;

        cc[0] = 0;
        cc[1] = 1;
        cc[2] = 2;
        cc[3] = 3;              // <--- Boom!
        for (i = 0; i < x; i++)
            if (bb[i] != (7 + i))
                fail = 1;
        if (tmp != k)
            fail = 1;
        if (fail) {
            printf("fail\n");
            exit(7);
        }
        printf("%d\n", (*cc) / y);
        printf("fail\n");
        exit(7);
    }
    __except (1)
    {
        for (i = 0; i < SIZE; i++)
            if (aa[i] != (x + i + 1))
                fail = 1;
        if (fail) {
            printf("fail\n");
            exit(7);
        }
        printf("pass\n");
        exit(0);
    }
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi
devenv /debugexe example3.exe
```

# Resulting error - several calls to alloca




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# Error: `global-buffer-overflow`

Article • 08/03/2021

> Address Sanitizer Error: Global buffer overflow

The compiler generates metadata for any variable in the `.data` or `.bss` sections. These variables have language scope of global or file static. They're allocated in memory before `main()` starts. Global variables in C are treated much differently than in C++. This difference is because of the complex rules for linking C.

In C, a global variable can be declared in several source files, and each definition can have different types. The compiler can't see all the possible definitions at once, but the linker can. For C, the linker defaults to selecting the largest-sized variable out of all the different declarations.

In C++, a global is allocated by the compiler. There can only be one definition, so the size of each definition is known at compile time.

## Example - globals in 'C' with multiple type definitions

C

```c
// file: a.c
int x;
```

C

```c
// file: b.c
char* x;
```

C

```c
// file: c.c
float* x[3];
```

C

```c
// file: example1-main.c
// global-buffer-overflow error
```

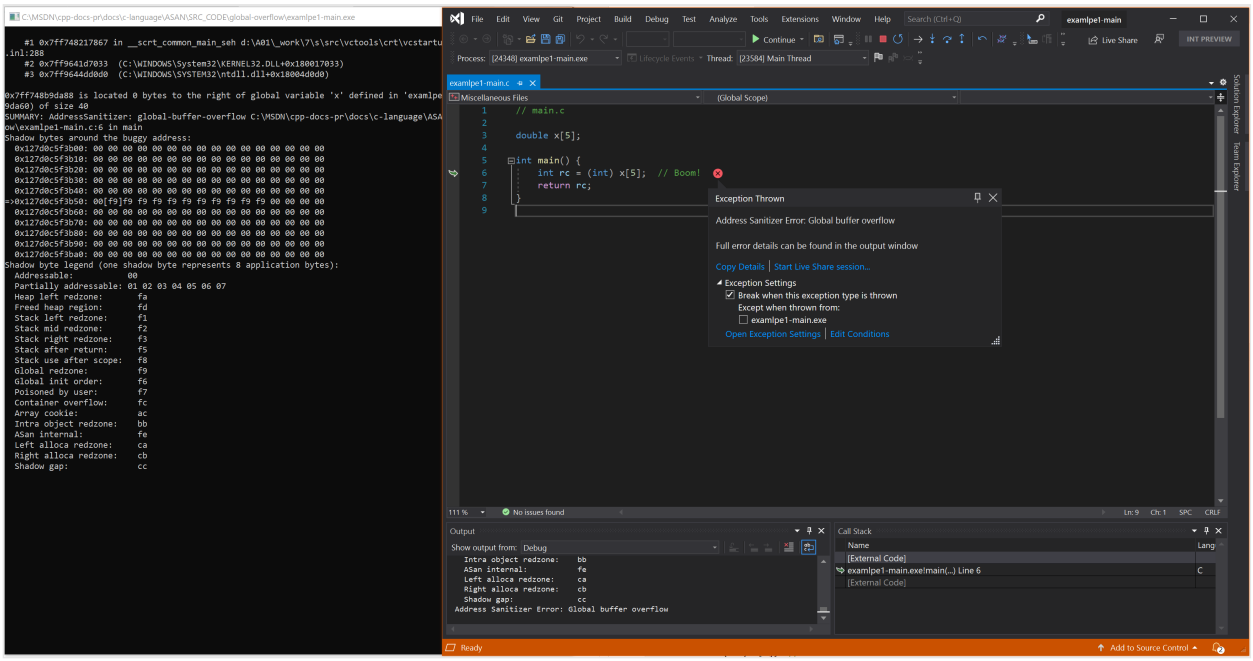
```
// AddressSanitizer reports a buffer overflow at the first line
// in function main() in all cases, REGARDLESS of the order in
// which the object files: a.obj, b.obj, and c.obj are linked.

double x[5];

int main() {
    int rc = (int) x[5];  // Boom!
    return rc;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl a.c b.c c.c example1-main.c /fsanitize=address /Zi
devenv /debugexe example1-main.exe
```

## Resulting error




## Example - simple function level static

C++

```
// example2.cpp
// global-buffer-overflow error
#include <string.h>

int
```

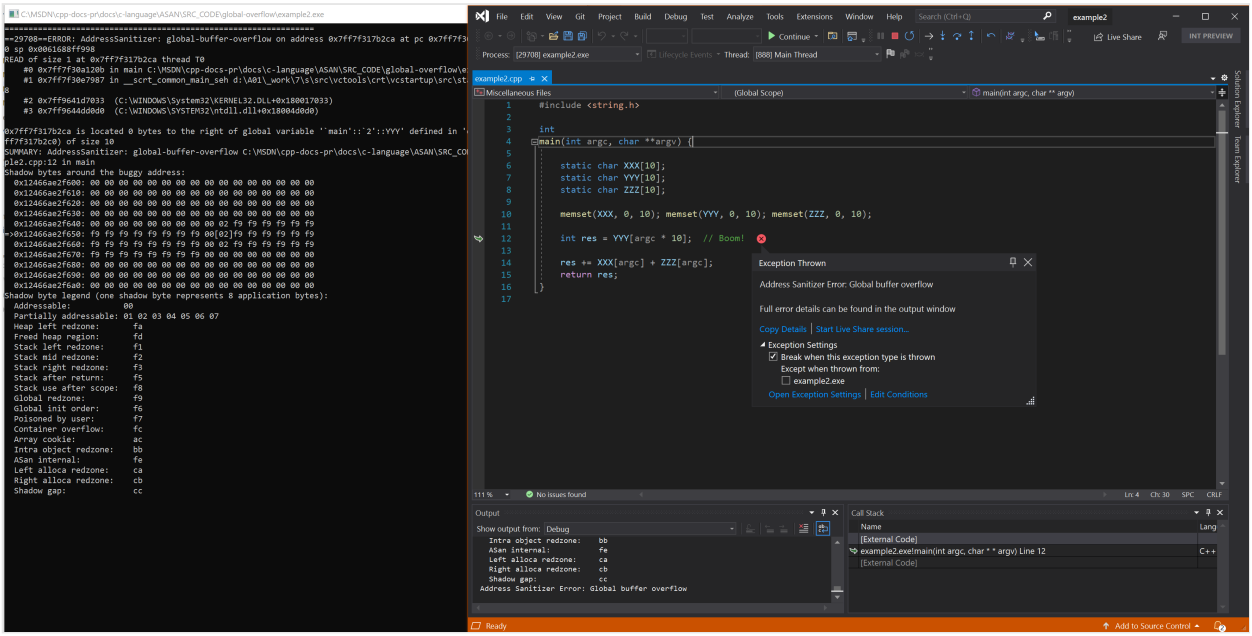
```
main(int argc, char **argv) {

    static char XXX[10];
    static char YYY[10];
    static char ZZZ[10];

    memset(XXX, 0, 10); memset(YYY, 0, 10); memset(ZZZ, 0, 10);

    int res = YYY[argc * 10];  // Boom!

    res += XXX[argc] + ZZZ[argc];
    return res;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe
```

## Resulting error - simple function level static




## Example - all global scopes in C++

C++

```
// example3.cpp
// global-buffer-overflow error
```

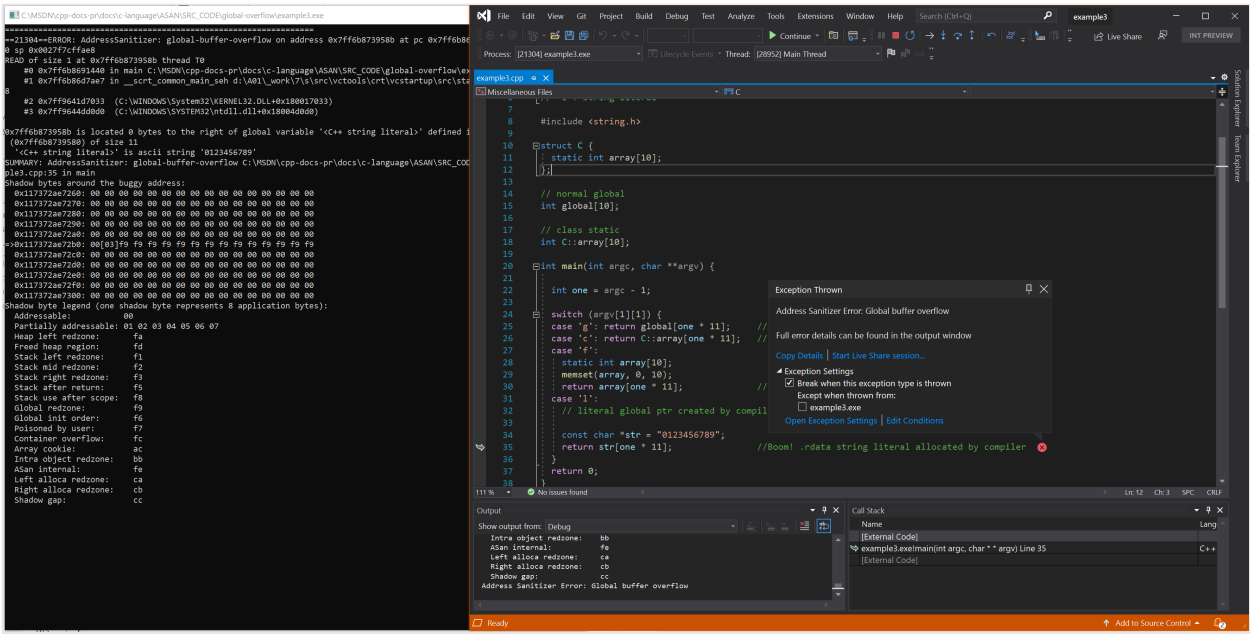
```
// Run 4 different ways with the choice of one of these options:
//
// -g : Global
// -c : File static
// -f : Function static
// -l : String literal

#include <string.h>

struct C {
    static int array[10];
};

// normal global
int global[10];

// class static
int C::array[10];

int main(int argc, char **argv) {

    int one = argc - 1;

    switch (argv[1][1]) {
    case 'g': return global[one * 11];     //Boom! simple global
    case 'c': return C::array[one * 11];   //Boom! class static
    case 'f':
        static int array[10];
        memset(array, 0, 10);
        return array[one * 11];            //Boom! function static
    case 'l':
        // literal global ptr created by compiler
        const char *str = "0123456789";
        return str[one * 11];              //Boom! .rdata string literal
allocated by compiler
    }
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi
devenv /debugexe example3.exe -l
```

## Resulting error - all global scopes in C++




# See also

- AddressSanitizer overview
- AddressSanitizer known issues
- AddressSanitizer build and language reference
- AddressSanitizer runtime reference
- AddressSanitizer shadow bytes
- AddressSanitizer cloud or distributed testing
- AddressSanitizer debugger integration
- AddressSanitizer error examples

# Error: `heap-buffer-overflow`

Article • 08/03/2021

> Address Sanitizer Error: Heap buffer overflow

This example demonstrates the error that results when a memory access occurs outside the bounds of a heap-allocated object.

## Example - classic heap buffer overflow

C++

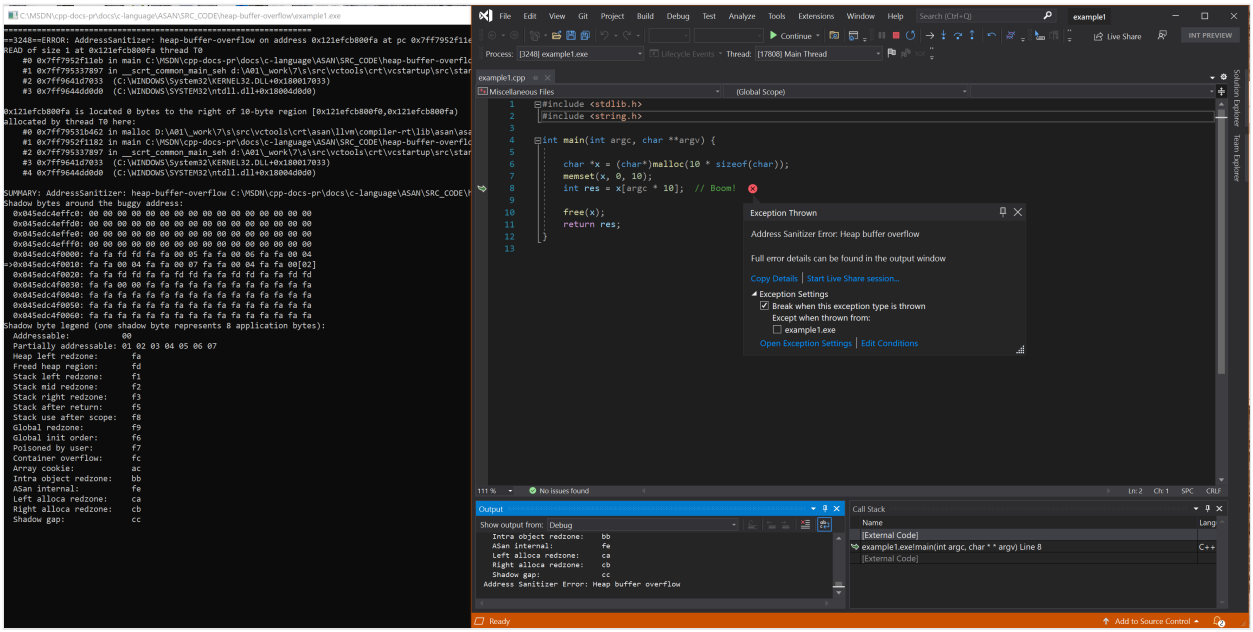
```
// example1.cpp
// heap-buffer-overflow error
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {

    char *x = (char*)malloc(10 * sizeof(char));
    memset(x, 0, 10);
    int res = x[argc * 10];  // Boom!

    free(x);
    return res;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




# Example - improper down cast

C++

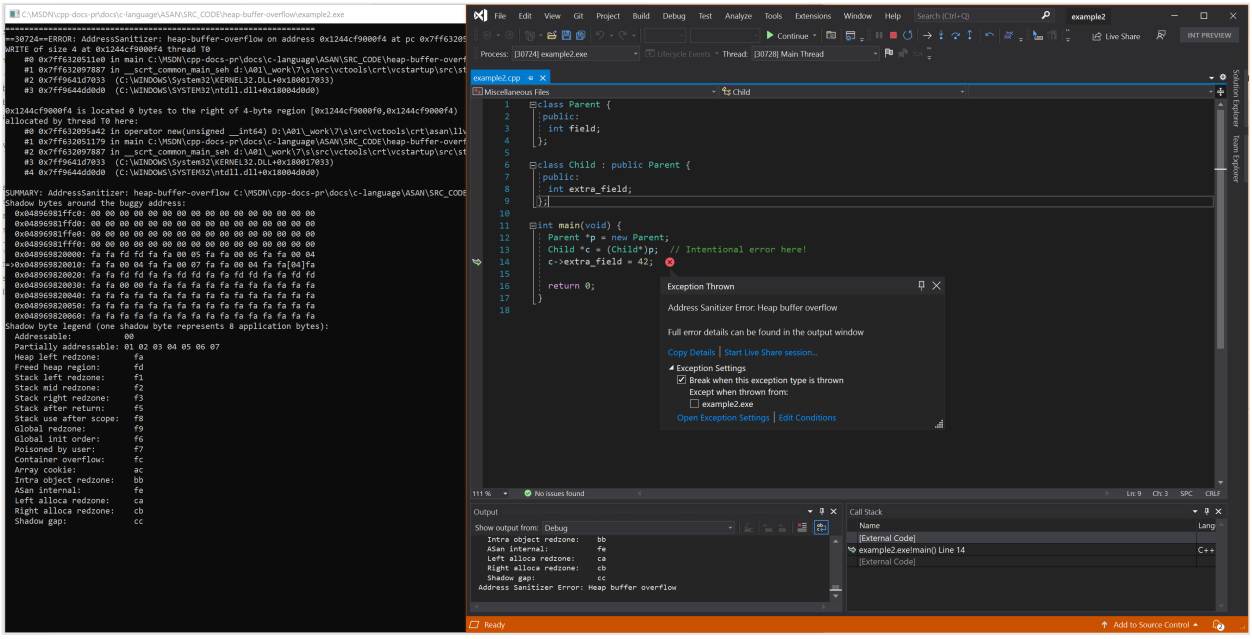
```
// example2.cpp
// heap-buffer-overflow error
class Parent {
public:
    int field;
};

class Child : public Parent {
public:
    int extra_field;
};

int main(void) {
    Parent *p = new Parent;
    Child *c = (Child*)p;  // Intentional error here!
    c->extra_field = 42;

    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe
```

## Resulting error - improper down cast




# Example - strncpy into heap

C++

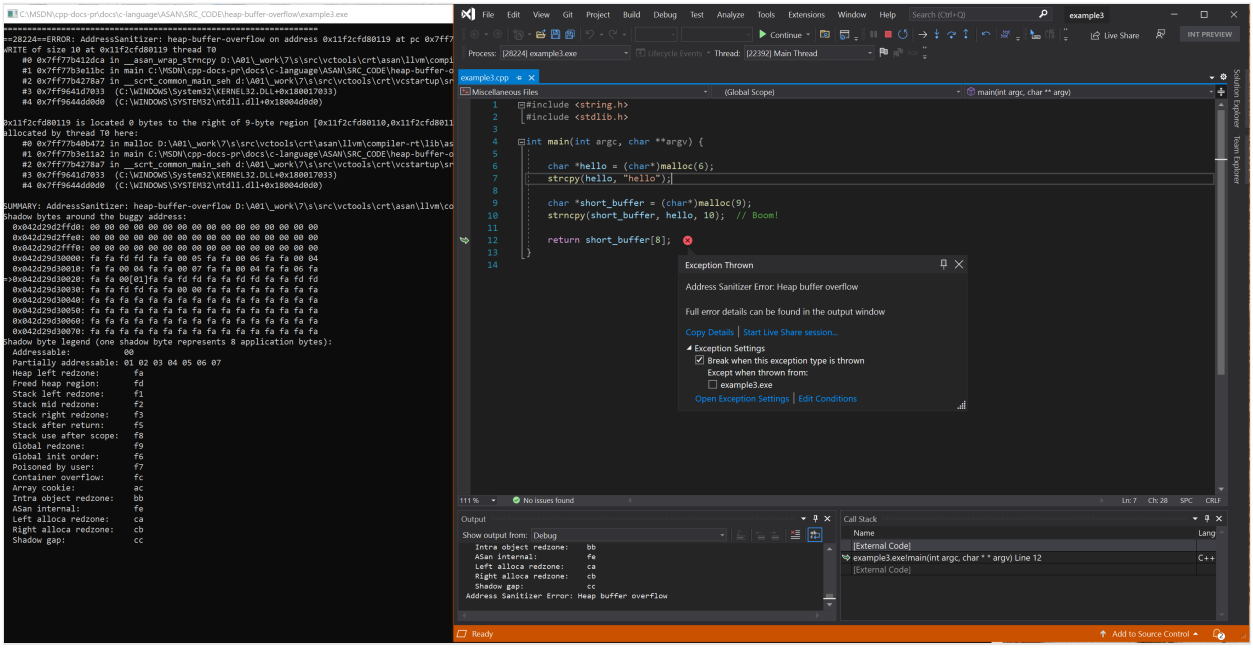
```
// example3.cpp
// heap-buffer-overflow error
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {

    char *hello = (char*)malloc(6);
    strcpy(hello, "hello");

    char *short_buffer = (char*)malloc(9);
    strncpy(short_buffer, hello, 10);  // Boom!

    return short_buffer[8];
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi
devenv /debugexe example3.exe
```

# Resulting error - strncpy into heap




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# Error: `heap-use-after-free`

Article • 04/10/2023

> Address Sanitizer Error: Use of deallocated memory

We show three examples where storage in the heap can be allocated via `malloc`, `realloc` (C), and `new` (C++), along with a mistaken use of `volatile`.

## Example - `malloc`

C++

```
// example1.cpp
// heap-use-after-free error
#include <stdlib.h>

int main() {
  char *x = (char*)malloc(10 * sizeof(char));
  free(x);

  // ...

  return x[5];   // Boom!
}
```

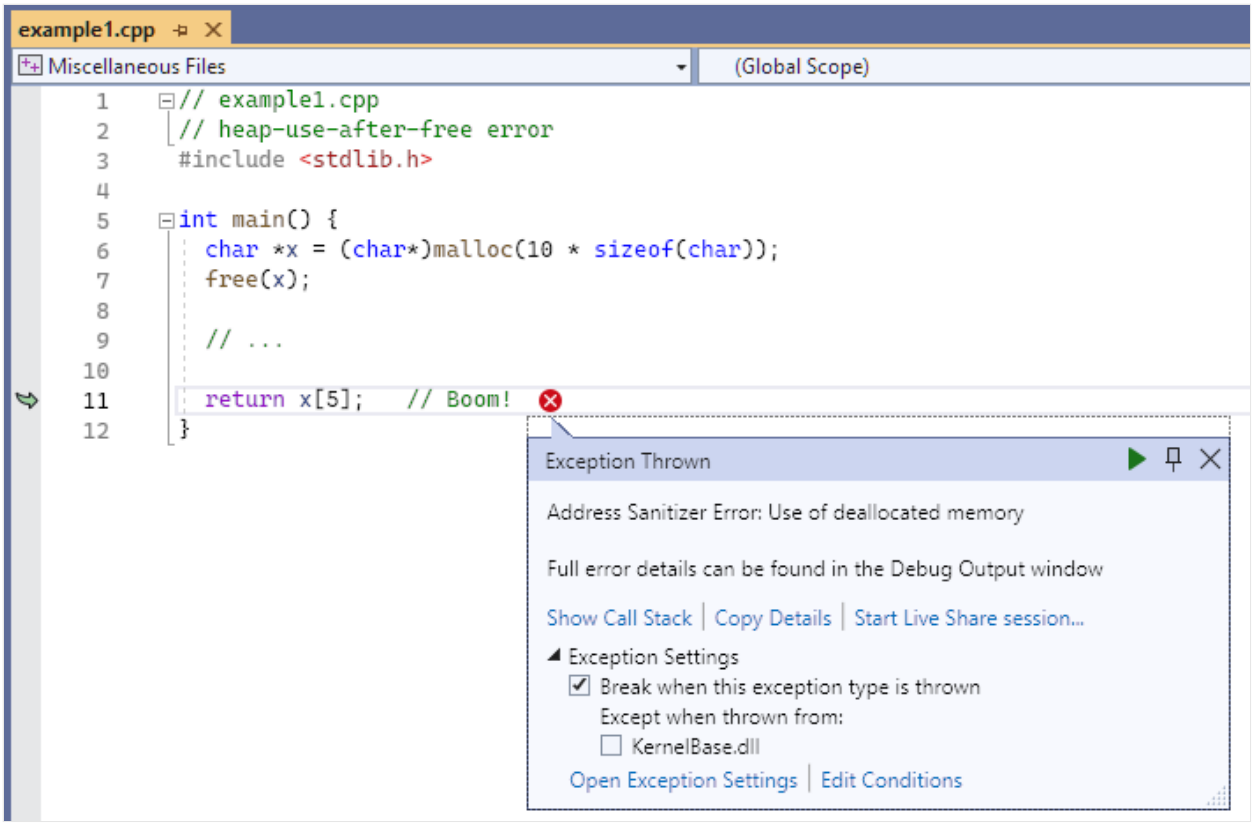
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

When Visual Studio appears, press `F5` to run example 1.

### Resulting error




# Example - `operator new`

C++

```
// example2.cpp
// heap-use-after-free error
#include <windows.h>

int main() {
  char *buffer = new char[42];
  delete [] buffer;

  // ...

  buffer[0] = 42;  // Boom!
  return 0;
}
```

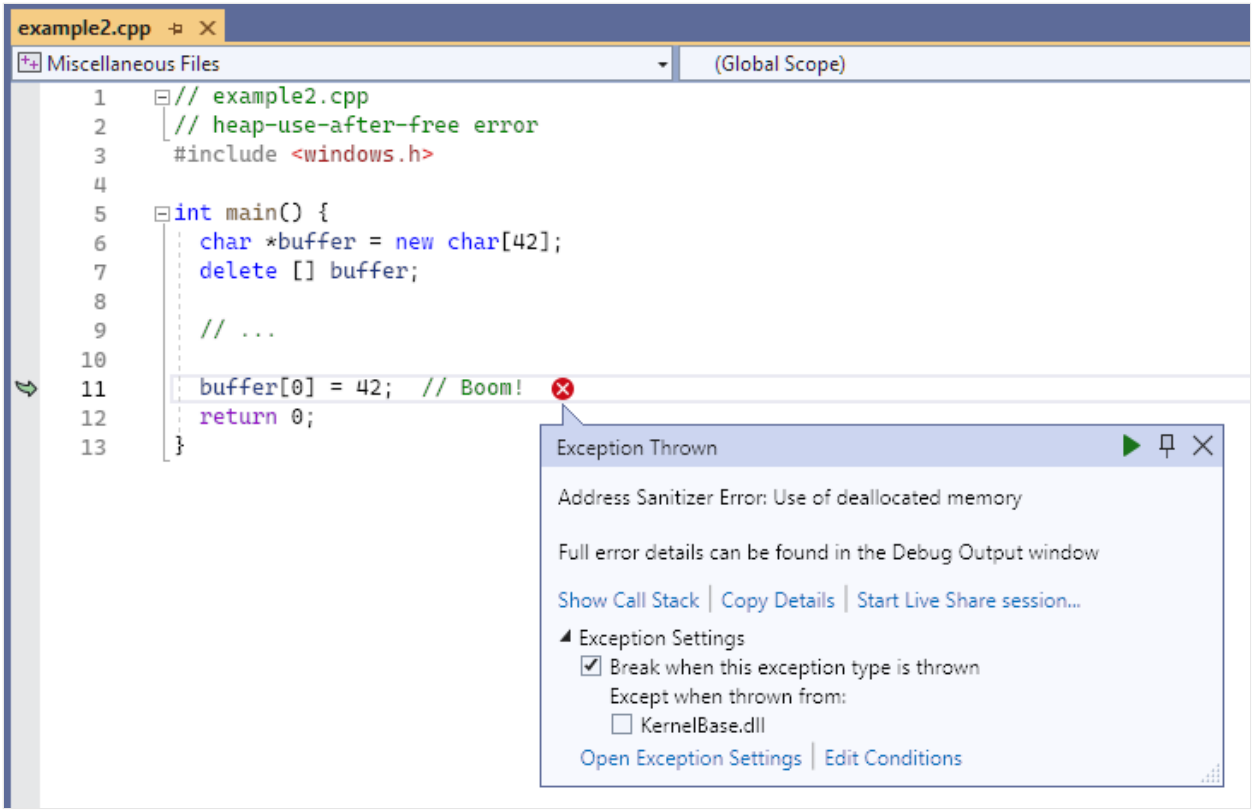
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe
```

When Visual Studio appears, press `F5` to run example 2.

## Resulting error - operator new




# Example - `realloc`

C++

```
// example3.cpp
// heap-use-after-free error
#include <malloc.h>

int main() {
  char *buffer = (char*)realloc(0, 42);
  free(buffer);

  // ...

  buffer[0] = 42;  // Boom!
  return 0;
}
```

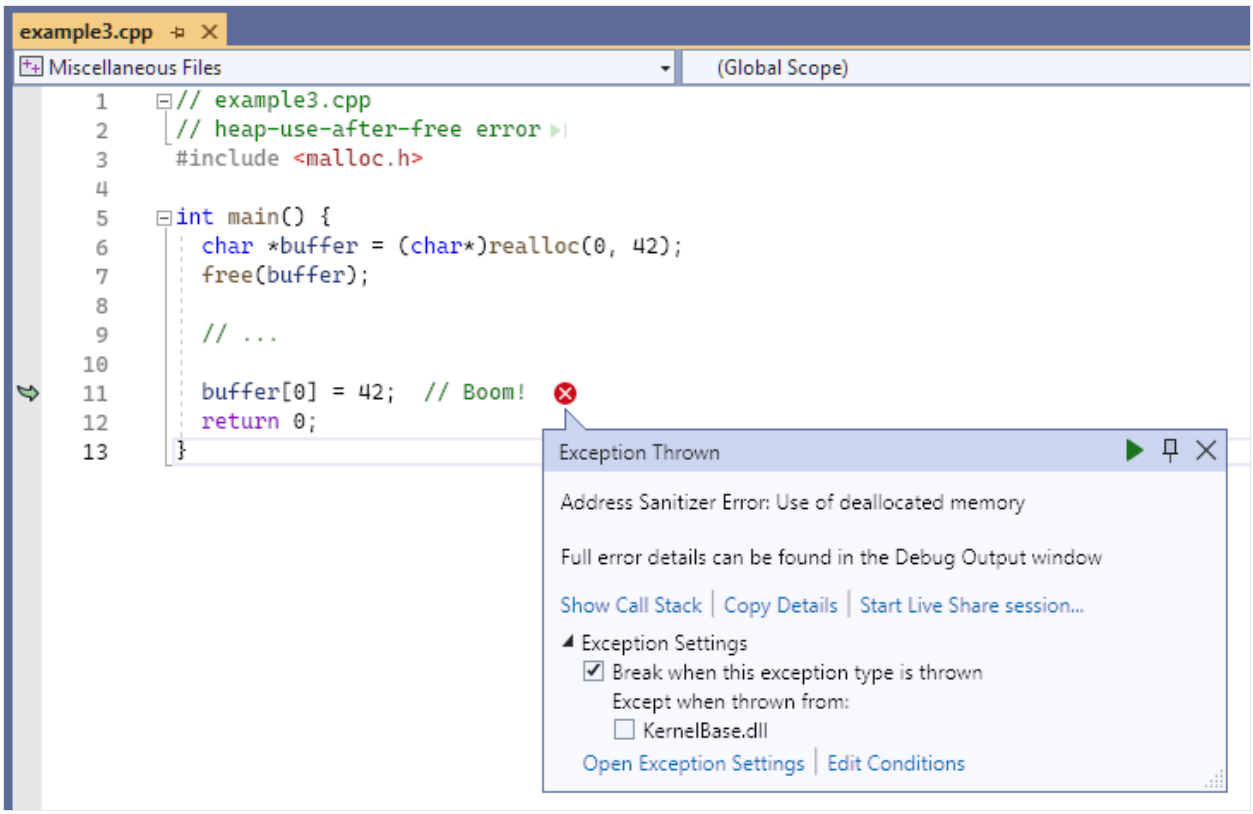
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi
devenv /debugexe example3.exe
```

When Visual Studio appears, press `F5` to run example 3.

## Resulting error - realloc




# Example - volatile

C++

```
// example4.cpp
// heap-use-after-free error
#include <stdlib.h>

int main() {

  volatile char *x = (char*)malloc(sizeof(char));
  free((void*)x);

      //...

  *x = 42;        // Boom!
}
```

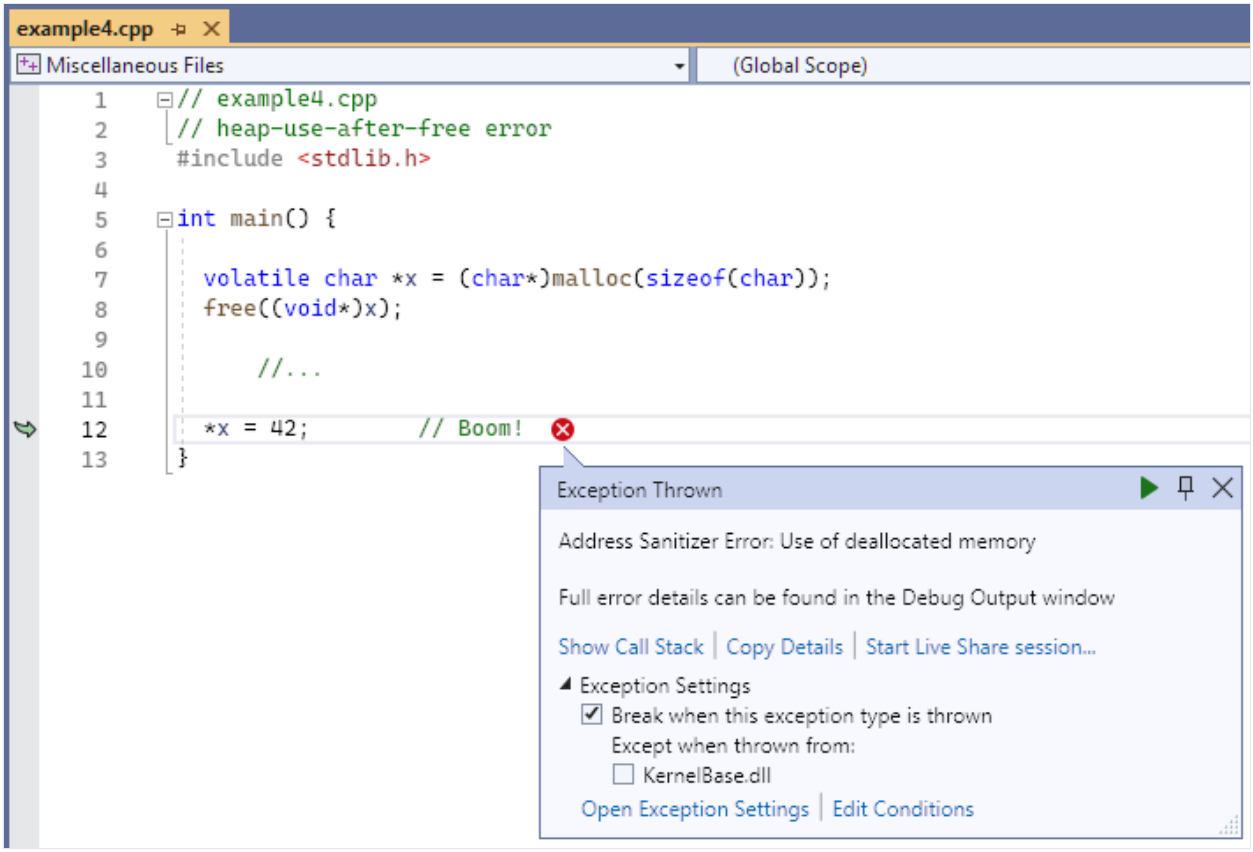
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example4.cpp /fsanitize=address /Zi
devenv /debugexe example4.exe
```

When Visual Studio appears, press `F5` to run example 4.

## Resulting error - volatile




# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Error: `invalid-allocation-alignment`

Article • 08/03/2021

> Address Sanitizer Error: invalid-allocation-alignment

The _aligned_malloc function requires a power of 2 for expressing the alignment. We simulate the "external" calculation of some alignment factor using an unoptimized global variable.

## Example

C++

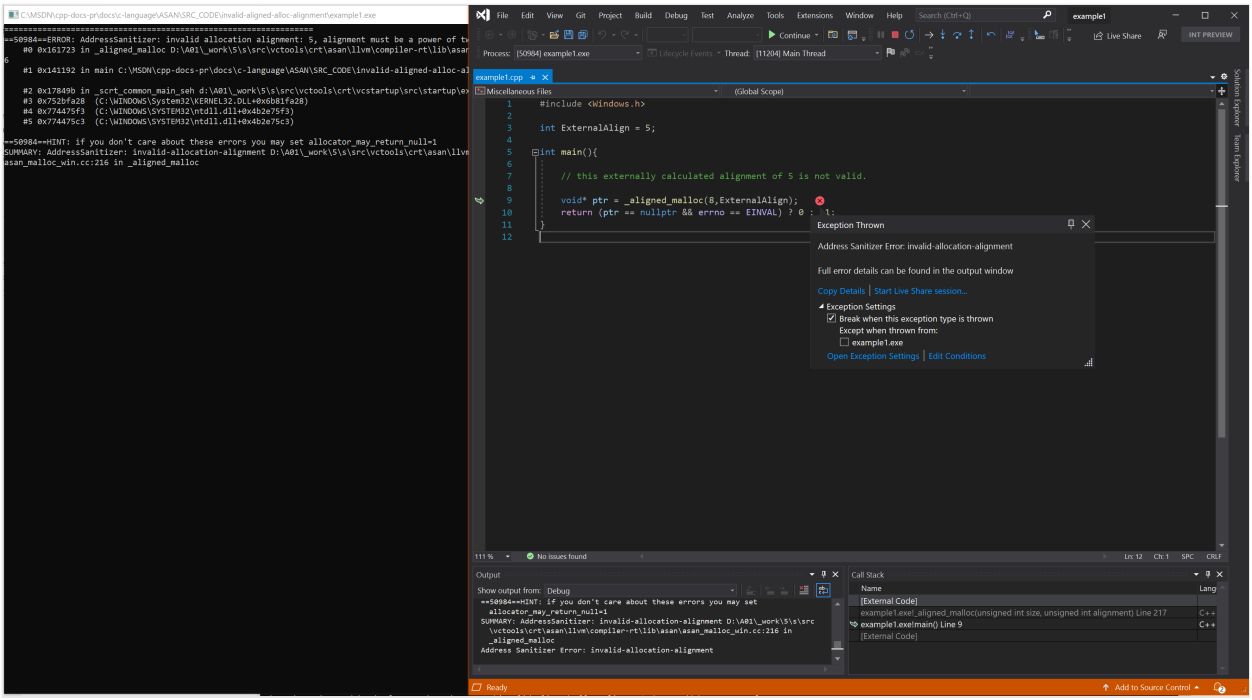
```
// example1.cpp
// invalid-allocation-alignment error
#include <Windows.h>

int ExternalAlign = 5;

int main(){

    // this externally calculated alignment of 5 isn't valid.

    void* ptr = _aligned_malloc(8,ExternalAlign);
    return (ptr == nullptr && errno == EINVAL) ? 0 : -1;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# Error: `memcpy-param-overlap`

Article • 03/21/2024

> Address Sanitizer Error: memcpy-param-overlap

The CRT function memcpy **doesn't support** overlapping memory. The CRT provides an alternative to `memcpy` that does support overlapping memory: memmove.

A common error is to treat `memmove` as being semantically equivalent to `memcpy`.

## Example

C++

```
// example1.cpp
// memcpy-param-overlap error
#include <string.h>

__declspec(noinline) void bad_function() {
    char buffer[] = "hello";

    memcpy(buffer, buffer + 1, 5); // BOOM!
}

int main(int argc, char **argv) {
    bad_function();
    return 0;
}
```

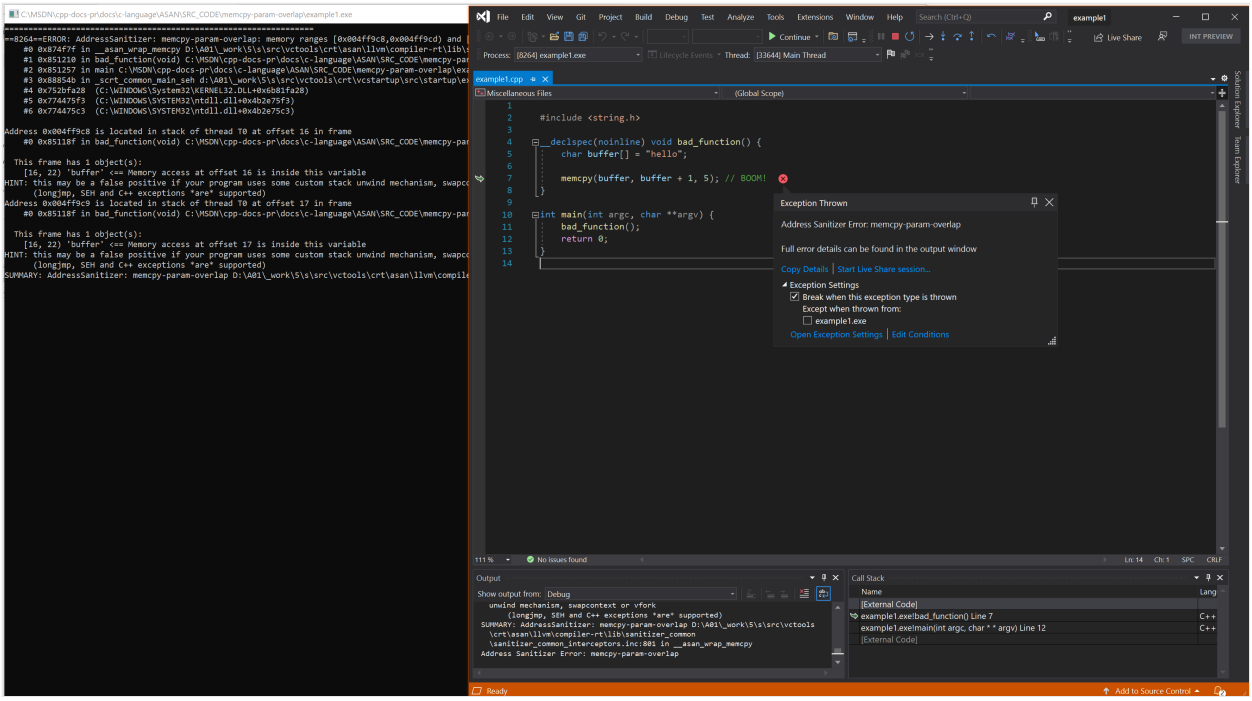
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi /Oi
devenv /debugexe example1.exe
```

The /Oi flag tells the compiler to treat `memcpy` and `memmove` as intrinsic functions. This is necessary because some versions of the standard library implement `memcpy` and `memmove` in the same way. Because ASAN is a dynamic analysis tool, it only detects errors with an observable runtime effect.

# Resulting error




# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

# Error: `new-delete-type-mismatch`

Article • 08/03/2021

> Address Sanitizer Error: Deallocation size different from allocation size

In this example, only `~Base`, and not `~Derived`, is called. The compiler generates a call to `~Base()` because the `Base` destructor isn't **`virtual`**. When we call `delete b`, the object's destructor is bound to the default definition. The code deletes an empty base class (or 1 byte on Windows). A missing **`virtual`** keyword on the destructor declaration is a common C++ error when using inheritance.

## Example - virtual destructor

C++

```
// example1.cpp
// new-delete-type-mismatch error
#include <memory>
#include <vector>

struct T {
    T() : v(100) {}
    std::vector<int> v;
};

struct Base {};

struct Derived : public Base {
    T t;
};

int main() {
    Base *b = new Derived;

    delete b;  // Boom!

    std::unique_ptr<Base> b1 = std::make_unique<Derived>();

    return 0;
}
```

Polymorphic base classes should declare **`virtual`** destructors. If a class has any virtual functions, it should have a virtual destructor.

To fix the example, add:

C++

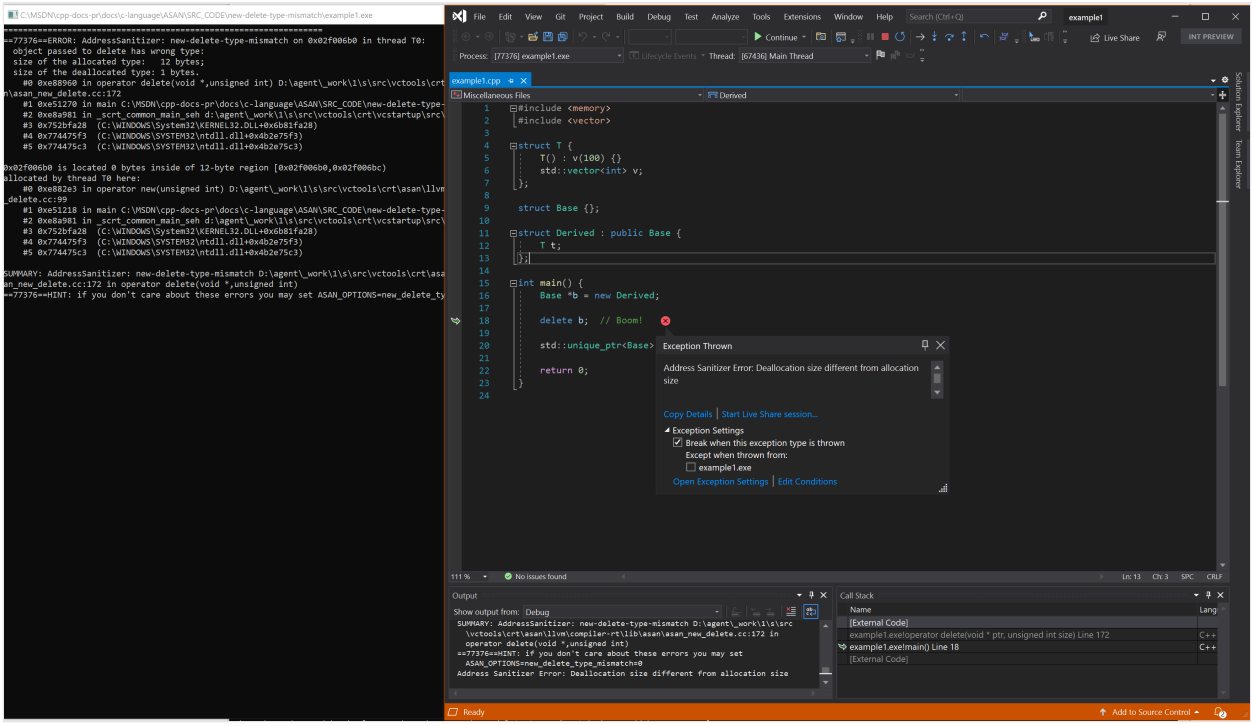
```
struct Base {
  virtual ~Base() = default;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Error: `stack-buffer-overflow`

Article • 09/30/2021

> Address Sanitizer Error: Stack buffer overflow

A stack buffer overflow can happen many ways in C or C++. We provide several examples for this category of error that you can catch by a simple recompile.

## Example - stack buffer overflow

C++

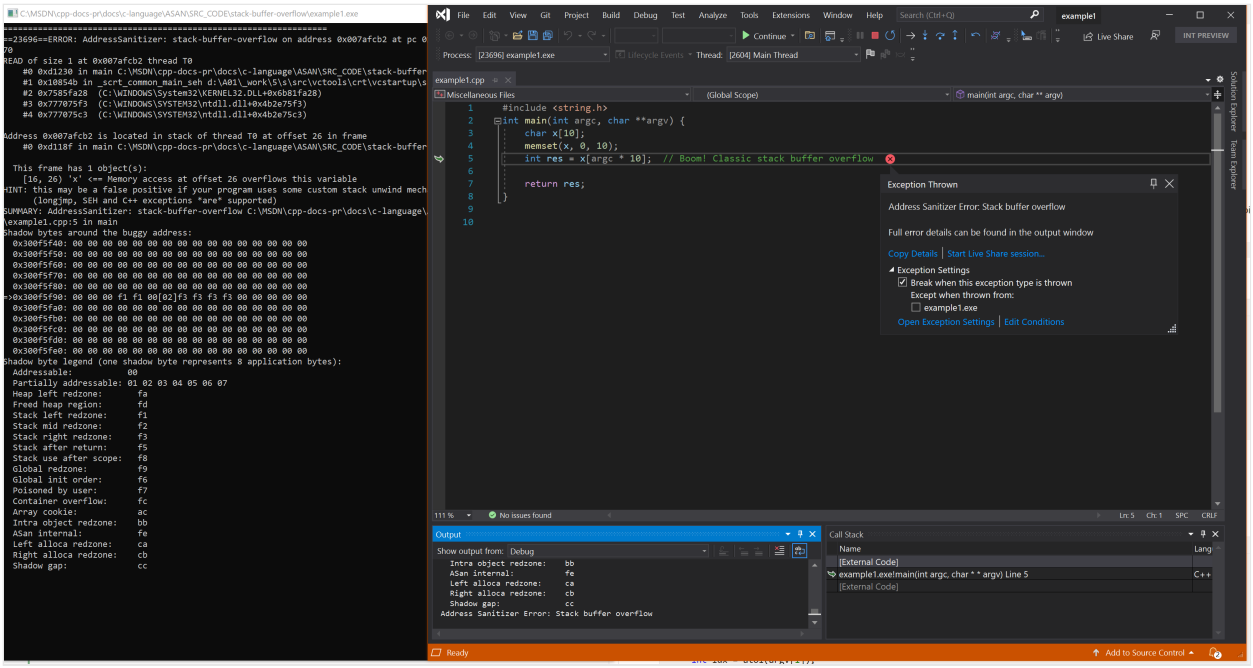
```
// example1.cpp
// stack-buffer-overflow error
#include <string.h>

int main(int argc, char **argv) {
    char x[10];
    memset(x, 0, 10);
    int res = x[argc * 10];  // Boom! Classic stack buffer overflow

    return res;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

### Resulting error




# Example - Stack buffer math

C++

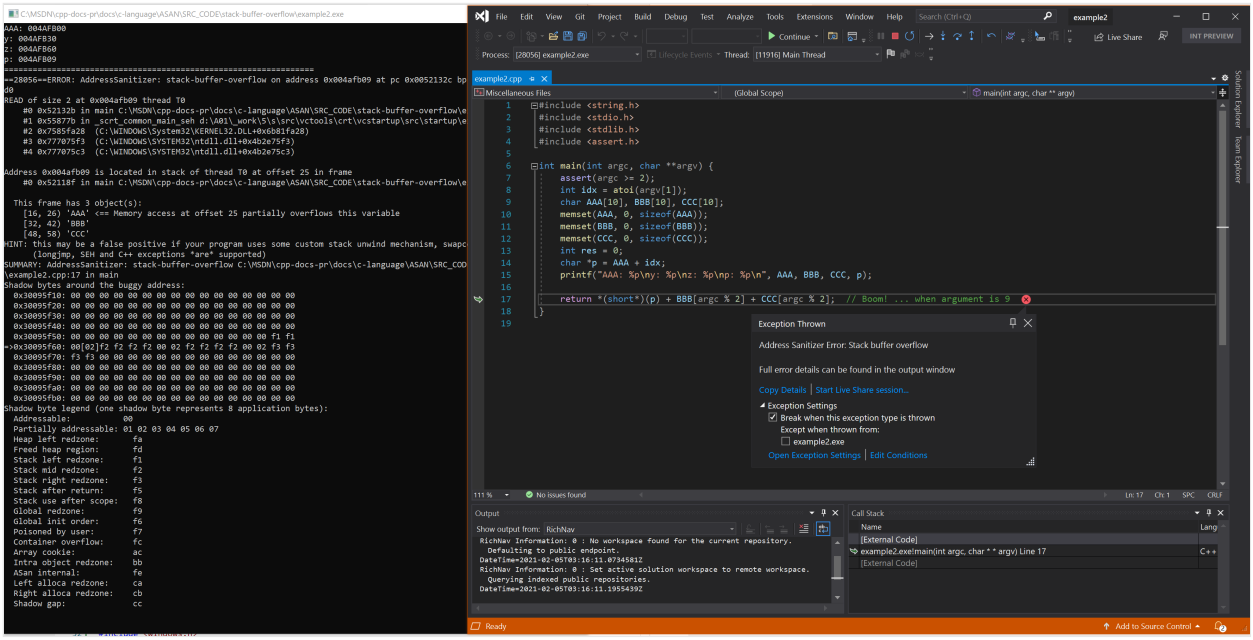
```
// example2.cpp
// stack-buffer-overflow error
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char **argv) {
    assert(argc >= 2);
    int idx = atoi(argv[1]);
    char AAA[10], BBB[10], CCC[10];
    memset(AAA, 0, sizeof(AAA));
    memset(BBB, 0, sizeof(BBB));
    memset(CCC, 0, sizeof(CCC));
    int res = 0;
    char *p = AAA + idx;
    printf("AAA: %p\ny: %p\nz: %p\np: %p\n", AAA, BBB, CCC, p);

    return *(short*)(p) + BBB[argc % 2] + CCC[argc % 2];  // Boom! ... when
argument is 9
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe 9
```

## Resulting error - Stack buffer math




# Example - improper down cast on stack

C++

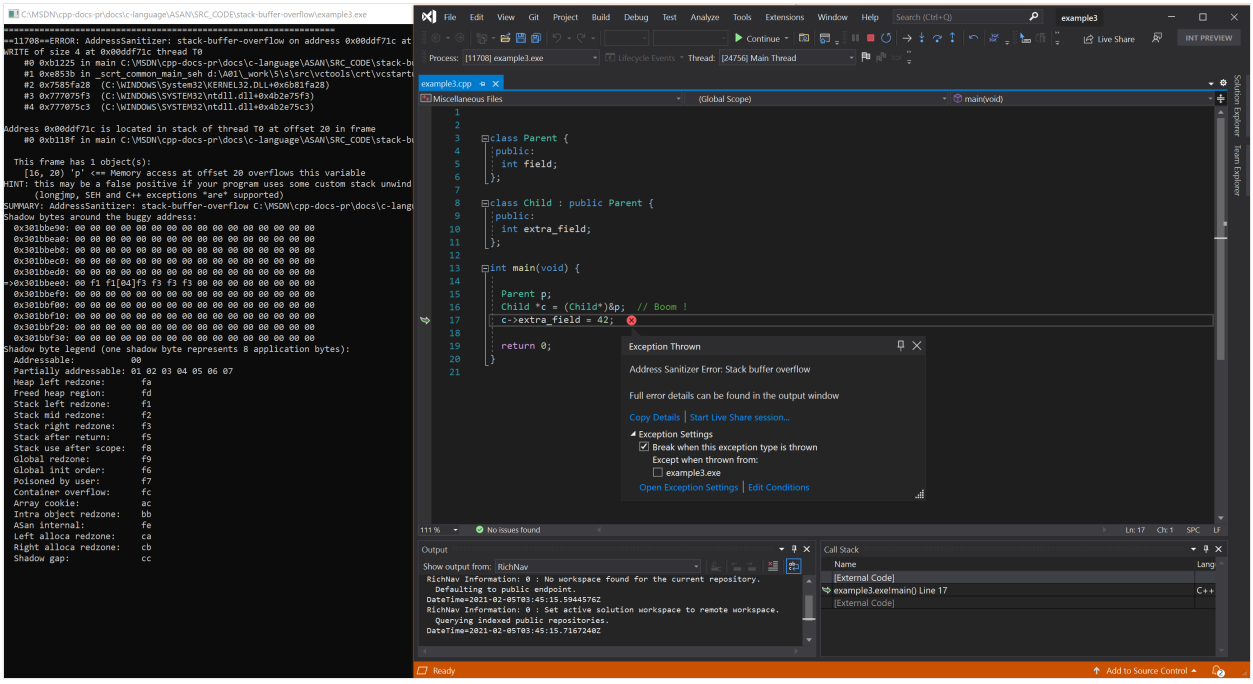
```
// example3.cpp
// stack-buffer-overflow error
class Parent {
public:
    int field;
};

class Child : public Parent {
public:
    int extra_field;
};

int main(void) {

    Parent p;
    Child *c = (Child*)&p;
    c->extra_field = 42;  // Boom !

    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi
devenv /debugexe example3.exe
```

## Resulting error - improper down cast on stack




## See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Error: `stack-buffer-underflow`

Article • 03/21/2024

> Address Sanitizer Error: Stack buffer underflow

These error messages indicate a memory access to somewhere before the beginning of a stack variable.

## Example - local array underflow

C++

```
// example1.cpp
// stack-buffer-underflow error
#include <stdio.h>

int main() {

    int subscript = -1;
    char buffer[42];
    buffer[subscript] = 42; // Boom!

    return 0;
}
```

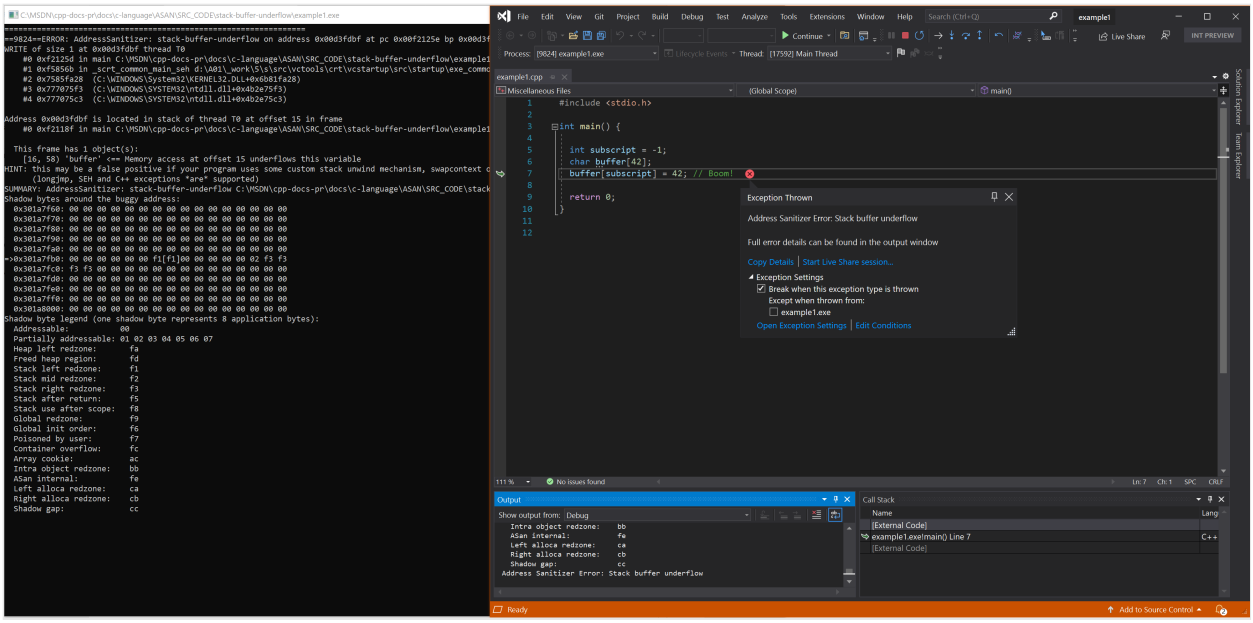
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi /Od
devenv /debugexe example1.exe
```

ASAN is a form of dynamic analysis, which means it can only detect bad code that is actually executed. An optimizer will remove the assignment to `buffer[subscript]` because `buffer[subscript]` is never read from. As a result, this example requires the `/Od` flag.

### Resulting error



# Example - stack underflow on thread

C++

```
// example2.cpp
// stack-buffer-underflow error
#include <windows.h>

DWORD WINAPI thread_proc(void *) {
    int subscript = -1;
    volatile char stack_buffer[42];
    stack_buffer[subscript] = 42;

    return 0;
}

int main() {
    HANDLE thr = CreateThread(NULL, 0, thread_proc, NULL, 0, NULL);

    if (thr == 0) return 0;

    WaitForSingleObject(thr, INFINITE);

    return 0;
}
```

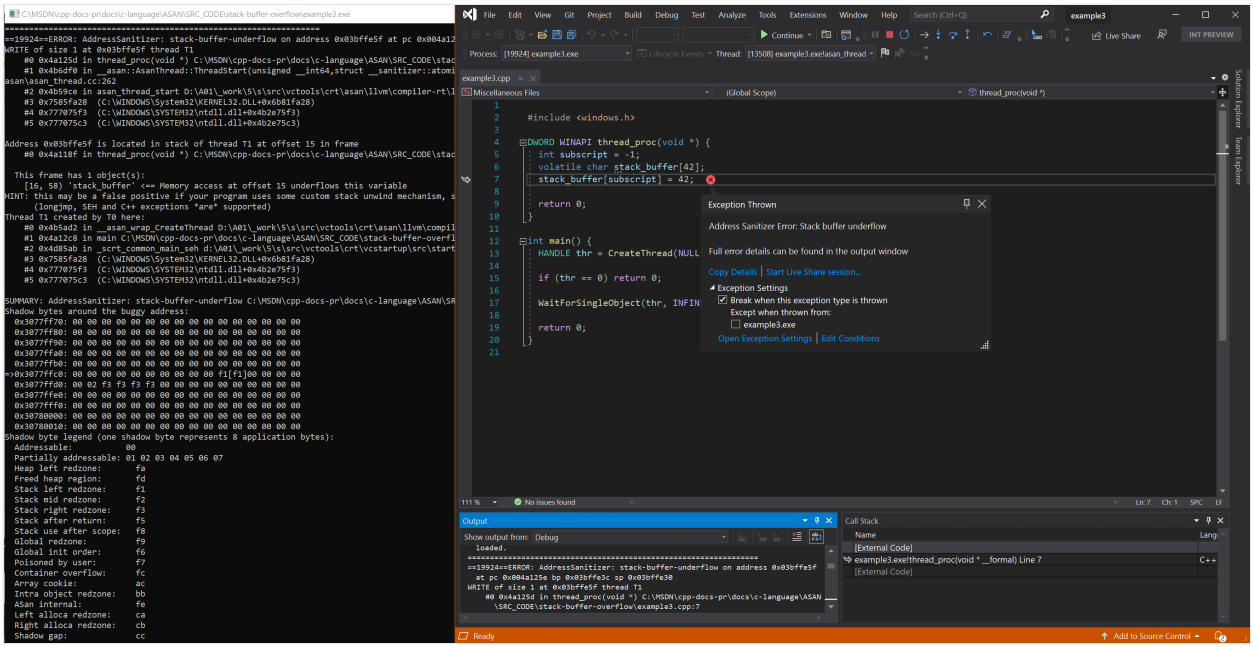
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
```

```
devenv /debugexe example2.exe
```

## Resulting error - stack underflow on thread



# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

# Error: `stack-use-after-return`

Article • 03/21/2024

> Address Sanitizer Error: Use of stack memory after return

This check requires code generation that's activated by an extra compiler option, /fsanitize-address-use-after-return, and by setting the environment variable `ASAN_OPTIONS=detect_stack_use_after_return=1`.

This check can slow your application down substantially. Consider the Clang summary of the algorithm that supports use after return, and the larger performance costs.

**ⓘ Important**

If you create an object file using the extra compiler option **`/fsanitize-address-use-after-return`**, the code generated by the compiler makes a runtime decision about how to allocate a stack frame. If the environment variable `ASAN_OPTIONS` isn't set to `detect_stack_use_after_return`, the code is still slower than using /fsanitize=address by itself. It's slower because there's still additional overhead from some stack frames that allocate space for parts of a frame by using `alloca()`. It's best to delete these object files when you're finished processing use-after-return errors.

## Example - Simple C

C++

```
// example1.cpp
// stack-use-after-return error
volatile char* x;

void foo() {
    char stack_buffer[42];
    x = &stack_buffer[13];
}

int main() {

    foo();
    *x = 42; // Boom!
```

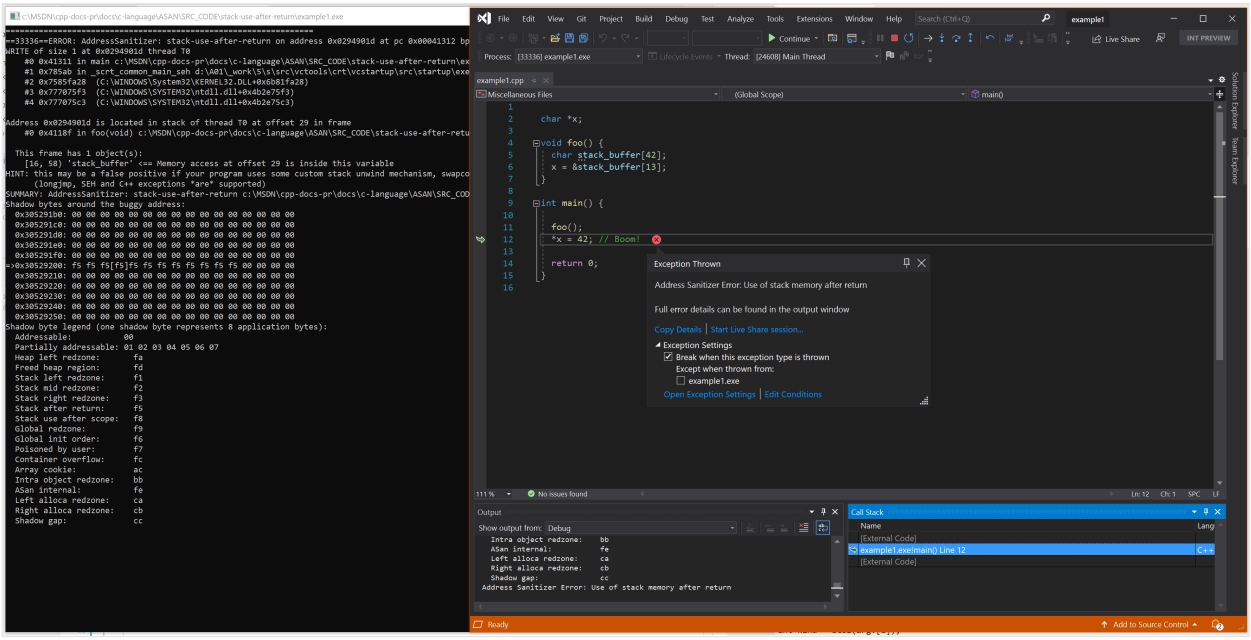
```
    return (*x == 42);
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /fsanitize-address-use-after-return /Zi
set ASAN_OPTIONS=detect_stack_use_after_return=1
devenv /debugexe example1.exe
```

## Resulting error - Simple C




# Example - C++ and templates

C++

```
// example2.cpp
// stack-use-after-return error
#include <stdlib.h>

enum ReadOrWrite { Read = 0, Write = 1 };

struct S32 {
    char x[32];
};

template<class T>
T* LeakStack() {
    T t[100];
```

```
    static volatile T* x;
    x = &t[0];
    return (T*)x;
}

template<class T>
void StackUseAfterReturn(int Idx, ReadOrWrite w) {
    static T sink;
    T* t = LeakStack<T>();
    if (w)
        t[100 + Idx] = T();
    else
        sink = t[100 + Idx];
}

int main(int argc, char* argv[]) {

    if (argc != 2) return 1;
    int kind = atoi(argv[1]);

    switch (kind) {
    case 1: StackUseAfterReturn<char>(0, Read); break;
    case 2: StackUseAfterReturn<S32>(0, Write); break;
    }
    return 0;
}
```

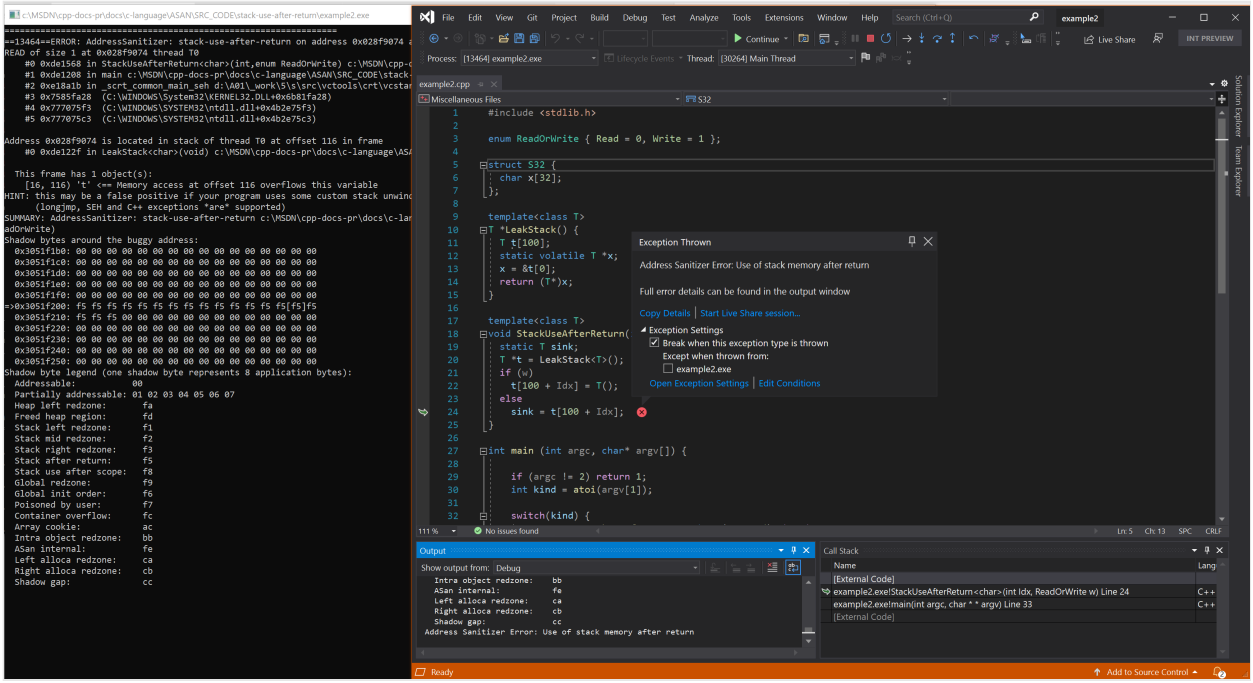
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /fsanitize-address-use-after-return /Zi
/Od
set ASAN_OPTIONS=detect_stack_use_after_return=1
devenv /debugexe example2.exe 1
```

ASAN is a form of dynamic analysis, which means it can only detect bad code that is actually executed. An optimizer may determine that the value of `t[100 + Idx]` or `sink` is never used and elide the assignment. As a result, this example requires the `/Od` flag.

## Resulting error - C++ and templates




# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Feedback

**Was this page helpful?** Yes No

Provide product feedback | Get help at Microsoft Q&A

# Error: `stack-use-after-scope`

Article • 03/21/2024

> Address Sanitizer Error: Use of out-of-scope stack memory

The use of a stack address outside the lexical scope of a variable's lifetime can happen many ways in C or C++.

## Example 1 - simple nested local

C++

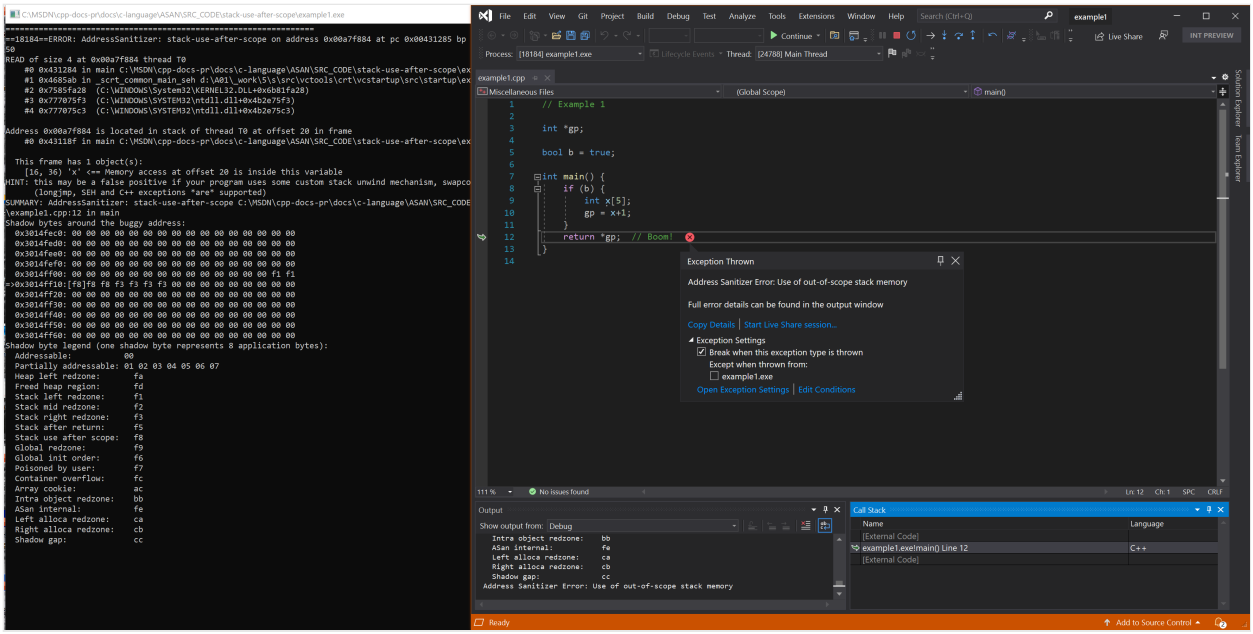
```
// example1.cpp
// stack-use-after-scope error
int *gp;

bool b = true;

int main() {
    if (b) {
        int x[5];
        gp = x+1;
    }
    return *gp;  // Boom!
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error - simple nested local



# Example 2 - lambda capture

C++

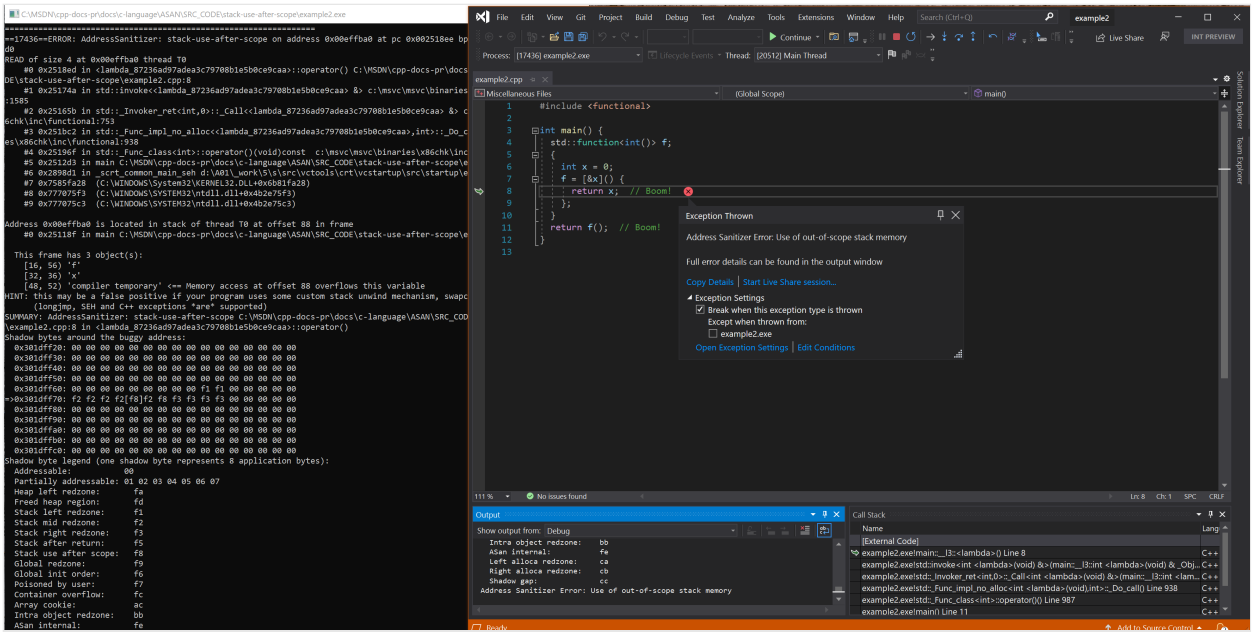
```
// example2.cpp
// stack-use-after-scope error
#include <functional>

int main() {
    std::function<int()> f;
    {
        int x = 0;
        f = [&x]() {
            return x;  // Boom!
        };
    }
    return f();  // Boom!
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example2.cpp /fsanitize=address /Zi
devenv /debugexe example2.exe
```

# Resulting error - lambda capture



# Example 3 - destructor ordering with locals

C++

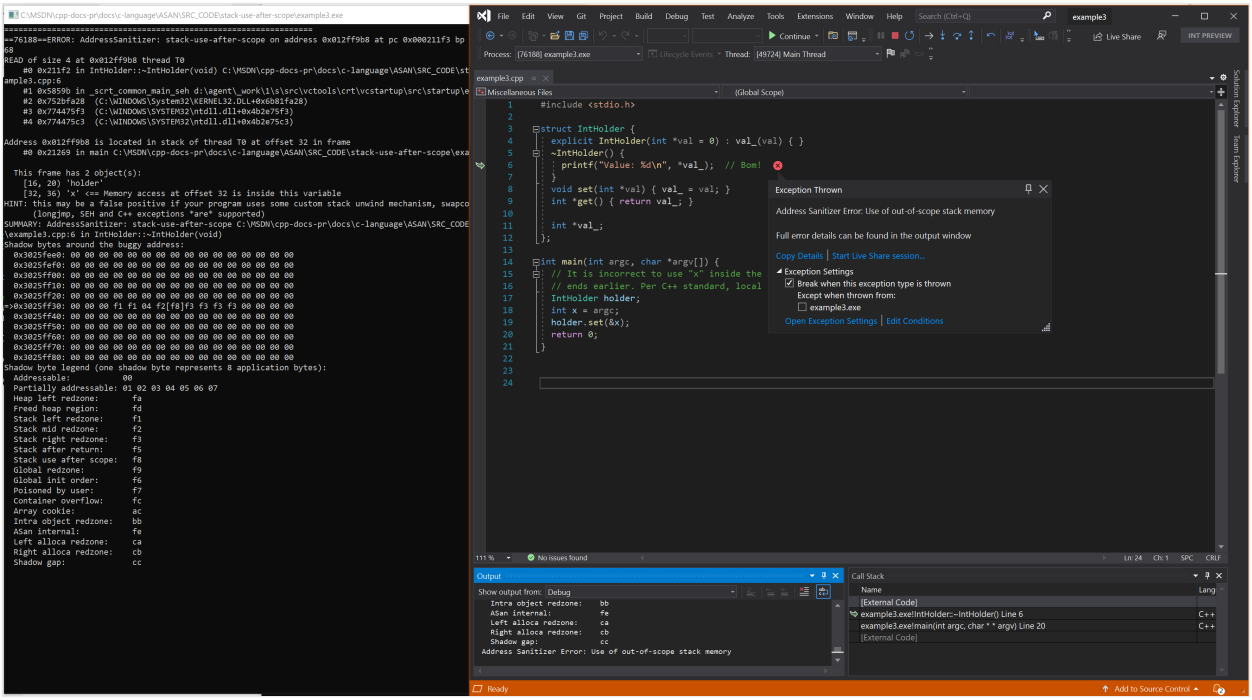
```
// example3.cpp
// stack-use-after-scope error
#include <stdio.h>

struct IntHolder {
    explicit IntHolder(int* val = 0) : val_(val) { }
    ~IntHolder() {
        printf("Value: %d\n", *val_);  // Bom!
    }
    void set(int* val) { val_ = val; }
    int* get() { return val_; }

    int* val_;
};

int main(int argc, char* argv[]) {
    // It's incorrect to use "x" inside the IntHolder destructor,
    // because the lifetime of "x" ends earlier. Per the C++ standard,
    // local lifetimes end in reverse order of declaration.
    IntHolder holder;
    int x = argc;
    holder.set(&x);
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example3.cpp /fsanitize=address /Zi /O1
devenv /debugexe example3.exe
```

## Resulting error - destructor ordering




# Example 4 - temporaries

C++

```
// example4.cpp
// stack-use-after-scope error
#include <iostream>

struct A {
    A(const int& v) {
        p = &v;
    }
    void print() {
        std::cout << *p;
    }
    const int* p;
};

void explicit_temp() {
    A a(5);     // the temp for 5 is no longer live;
    a.print();
}

void temp_from_conversion() {
    double v = 5;
```

```
    A a(v);     // local v is no longer live.
    a.print();
}

void main() {
    explicit_temp();
    temp_from_conversion();
}
```

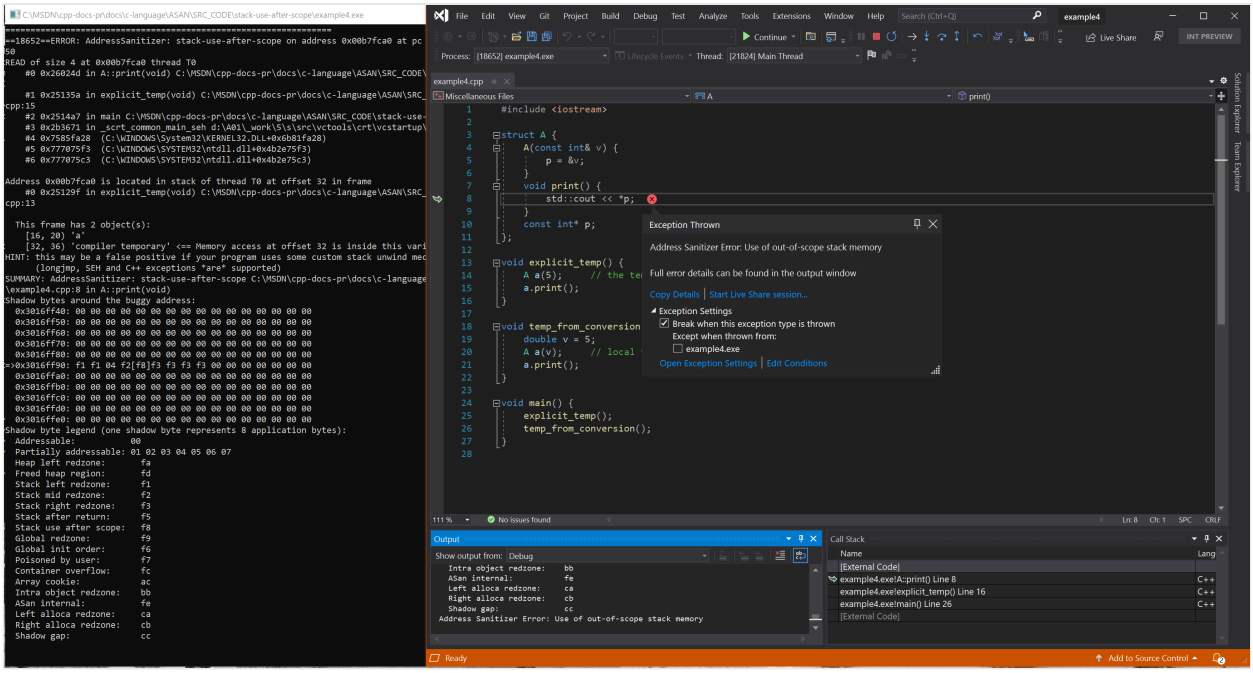
To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example4.cpp /EHsc /fsanitize=address /Zi /Od
devenv /debugexe example4.exe
```

ASAN is a form of dynamic analysis, which means it can only detect bad code that is actually executed. An optimizer may propagate the value of `v` in these cases instead of reading from the address stored in `p`. As a result, this example requires the `/Od` flag.

## Resulting error - temporaries




# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference

AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

## Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

# Error: `strncat-param-overlap`

Article • 08/03/2021

> Address Sanitizer Error: strncat-param-overlap

Code that moves memory in overlapping buffer can cause hard-to-diagnose errors.

## Example

This example shows how AddressSanitizer can catch errors caused by overlapped parameters to CRT functions.

(Based on llvm-project/compiler-rt/test/asan/TestCases/strncat-overlap.cpp .)

C++

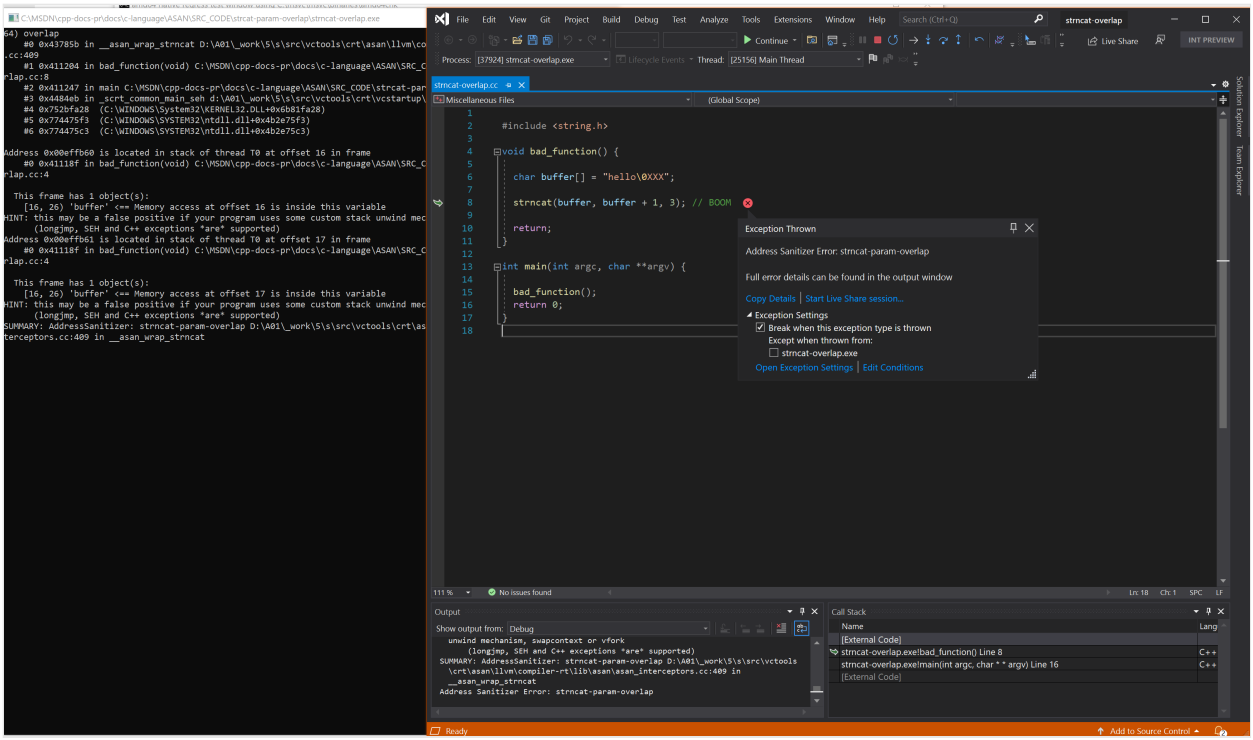
```
// example1.cpp
// strncat-param-overlap error
#include <string.h>

void bad_function() {
    char buffer[] = "hello\0XXX";
    strncat(buffer, buffer + 1, 3); // BOOM
    return;
}

int main(int argc, char **argv) {
    bad_function();
    return 0;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




# See also

AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples

# Error: `use-after-poison`

Article • 08/03/2021

> Address Sanitizer Error: Use of poisoned memory

A developer can manually poison memory to customize debugging.

## Example

C++

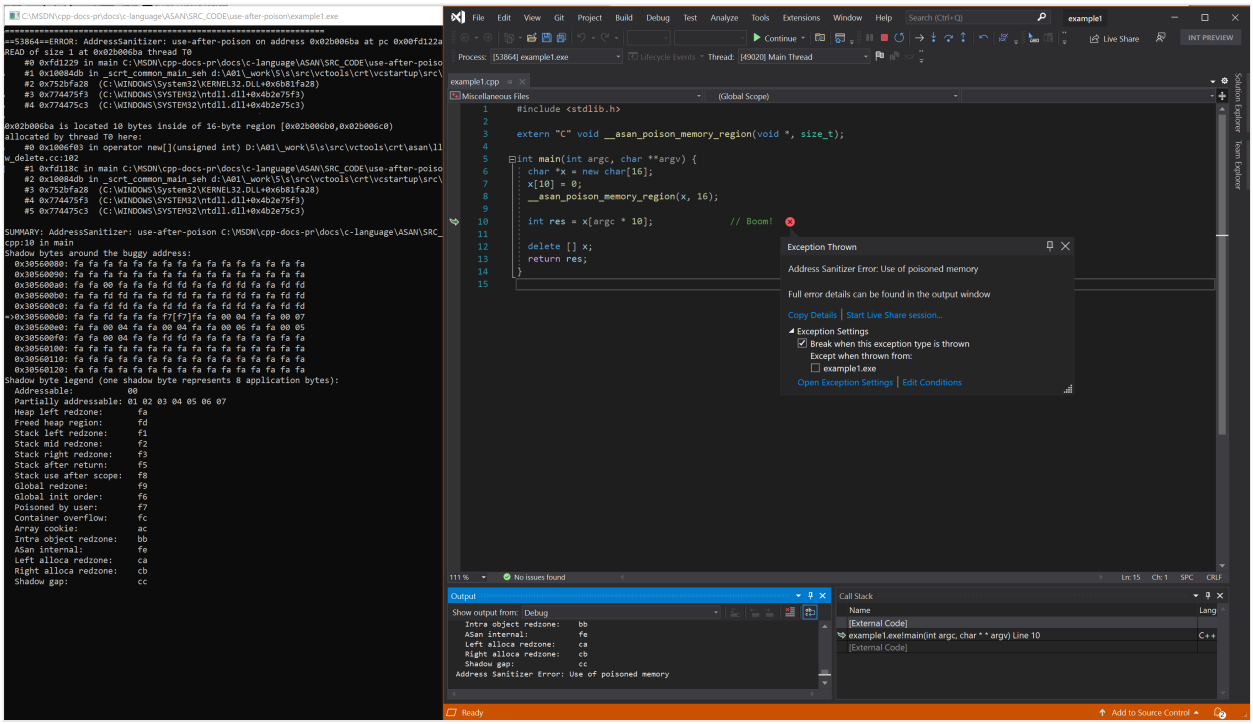
```
// example1.cpp
// use-after-poison error
#include <stdlib.h>

extern "C" void __asan_poison_memory_region(void *, size_t);

int main(int argc, char **argv) {
    char *x = new char[16];
    x[10] = 0;
    __asan_poison_memory_region(x, 16);

    int res = x[argc * 10];              // Boom!

    delete [] x;
    return res;
}
```

To build and test this example, run these commands in a Visual Studio 2019 version 16.9 or later developer command prompt:

Windows Command Prompt

```
cl example1.cpp /fsanitize=address /Zi
devenv /debugexe example1.exe
```

## Resulting error




# See also

AddressSanitizer overview

AddressSanitizer known issues

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

AddressSanitizer debugger integration

AddressSanitizer error examples

# AddressSanitizer known issues

Article • 12/04/2023

> **Note**
>
> Send us **feedback** on what you'd like to see in future releases, and **report bugs** if you run into issues.

## Incompatible options and functionality

These options and functionality are incompatible with /fsanitize=address and should be disabled or avoided.

- The /RTC options are incompatible with AddressSanitizer and should be disabled.
- Incremental linking is unsupported, and should be disabled.
- Edit-and-Continue is unsupported, and should be disabled.
- Coroutines are incompatible with AddressSanitizer, and resumable functions are exempt from instrumentation.
- OpenMP is unsupported, and should be disabled.
- Managed C++ is unsupported, and should be disabled.
- C++ AMP is unsupported, and should be disabled.
- Universal Windows Platform (UWP) applications are unsupported.
- Special case list files are unsupported.
- MFC using the optional alloc_dealloc_mismatch runtime option is unsupported.

## Standard library support

The MSVC standard library (STL) is partially enlightened to understand the AddressSanitizer and provide other checks. For more information, see container-overflow error.

When annotations are disabled or in versions without support, AddressSanitizer exceptions raised in STL code do still identify true bugs. However, they aren't as precise as they could be.

This example demonstrates the lack of precision and the benefits of enabling annotations:

```
C++
```

```
// Compile with: cl /fsanitize=address /Zi
#include <vector>

int main() {
    // Create a vector of size 10, but with a capacity of 20.
    std::vector<int> v(10);
    v.reserve(20);

    // In versions prior to 17.2, MSVC ASan does NOT raise an exception
here.
    // While this is an out-of-bounds write to 'v', MSVC ASan
    // ensures the write is within the heap allocation size (20).
    // With 17.2 and later, MSVC ASan will raise a 'container-overflow'
exception:
    // ==18364==ERROR: AddressSanitizer: container-overflow on address
0x1263cb8a0048 at pc 0x7ff6466411ab bp 0x005cf81ef7b0 sp 0x005cf81ef7b8
    v[10] = 1;

    // Regardless of version, MSVC ASan DOES raise an exception here, as
this write
    // is out of bounds from the heap allocation.
    v[20] = 1;
}
```

# Windows versions

As there are dependencies with specific Windows versions, support is focused on Windows 10. MSVC AddressSanitizer was tested on 10.0.14393 (RS1), and 10.0.21323 (prerelease insider build). Report a bug if you run into issues.

# Memory usage

The AddressSanitizer runtime doesn't release memory back to the OS during execution. From the OS's point of view, it may look like there's a memory leak. This design decision is intentional, so as not to allocate all the required memory up front.

# AddressSanitizer runtime DLL locations

The `clang_rt.asan*.dll` runtime files are installed next to the compilers in `%VSINSTALLDIR%\VC\Tools\MSVC\<version>\bin\<host-arch>\<target-arch>\`. These locations are on the path in debugging sessions, and in Visual Studio developer command prompts. These files are never placed in `C:\Windows\System32` or `C:\Windows\SysWOW64`.

## Custom property sheet support

The Property Manager window in the Visual Studio IDE allows you to add custom `.props` files to your projects. Even though the **Enable Address Sanitizer** property (`<EnableASAN>`) is shown, the build doesn't honor it. That's because the custom `.props` files get included after `Microsoft.cpp.props`, which uses the `<EnableASAN>` value to set other properties.

As a workaround, you can create a `Directory.Build.props` file in the root of your project to define the `<EnableASAN>` property. For more information, see Customize C++ builds.

## See also

AddressSanitizer overview
AddressSanitizer build and language reference
AddressSanitizer runtime reference
AddressSanitizer shadow bytes
AddressSanitizer cloud or distributed testing
AddressSanitizer debugger integration
AddressSanitizer error examples