code::dive

Status quo: `clang-tidy & AddressSanitizer` on Windows

November 20, 2019
Wrocław

CAPHYON

@ciura_victor

Victor Ciura
Principal Engineer
CAPHYON

Abstract

Clang-tidy is the go-to assistant for most C++ programmers looking to improve their code. If you set out to modernize your aging code base and find hidden bugs along the way, clang-tidy is your friend. My team brought all the clang-tidy magic to Visual Studio C++ developers with an open-source Visual Studio extension called “Clang Power Tools”. This helped tens of thousands of developers leverage its powers to improve their projects, regardless of their compiler of choice for building their applications.

Clang-tidy comes packed with hundreds of built-in checks: best practice fixits and static analysis for potential risks. Most of them are extremely valuable in real-world code, but there are several cases where you might need to run custom checks/transformations for your project. You will now get a crash course in writing your own tidy check/fix-it from scratch.

You think static analysis is great? Wait until you try dynamic/runtime analysis! After years of improvements and successes for Clang and GCC users, AddressSanitizer (ASan) is finally coming to Windows, in Visual Studio 2019. Let's take an overview of how this experience is going to be for MSVC projects.

Who Am I ?

Advanced Installer

Clang Power Tools

@ciura_victor

Vignette in 4 parts

The Tools
Legacy Code
Take Control
Status Quo

Part I

The Tools

Lunched 2 Years Ago: `September 2017`

Bringing Clang-tidy Magic to Visual Studio C++ Developers

Victor Ciura
Technical Lead, Advanced Installer
www.advancedinstaller.com

VICTOR CIURA

Bringing Clang-tidy Magic to Visual Studio C++ Developers

0:06 / 1:00:34

CppCon 2017: Victor Ciura “Bringing Clang-tidy Magic to Visual Studio C++ Developers”

https://www.youtube.com/watch?v=Wl-9ozmxXbo

=

->

Clang Power Tools

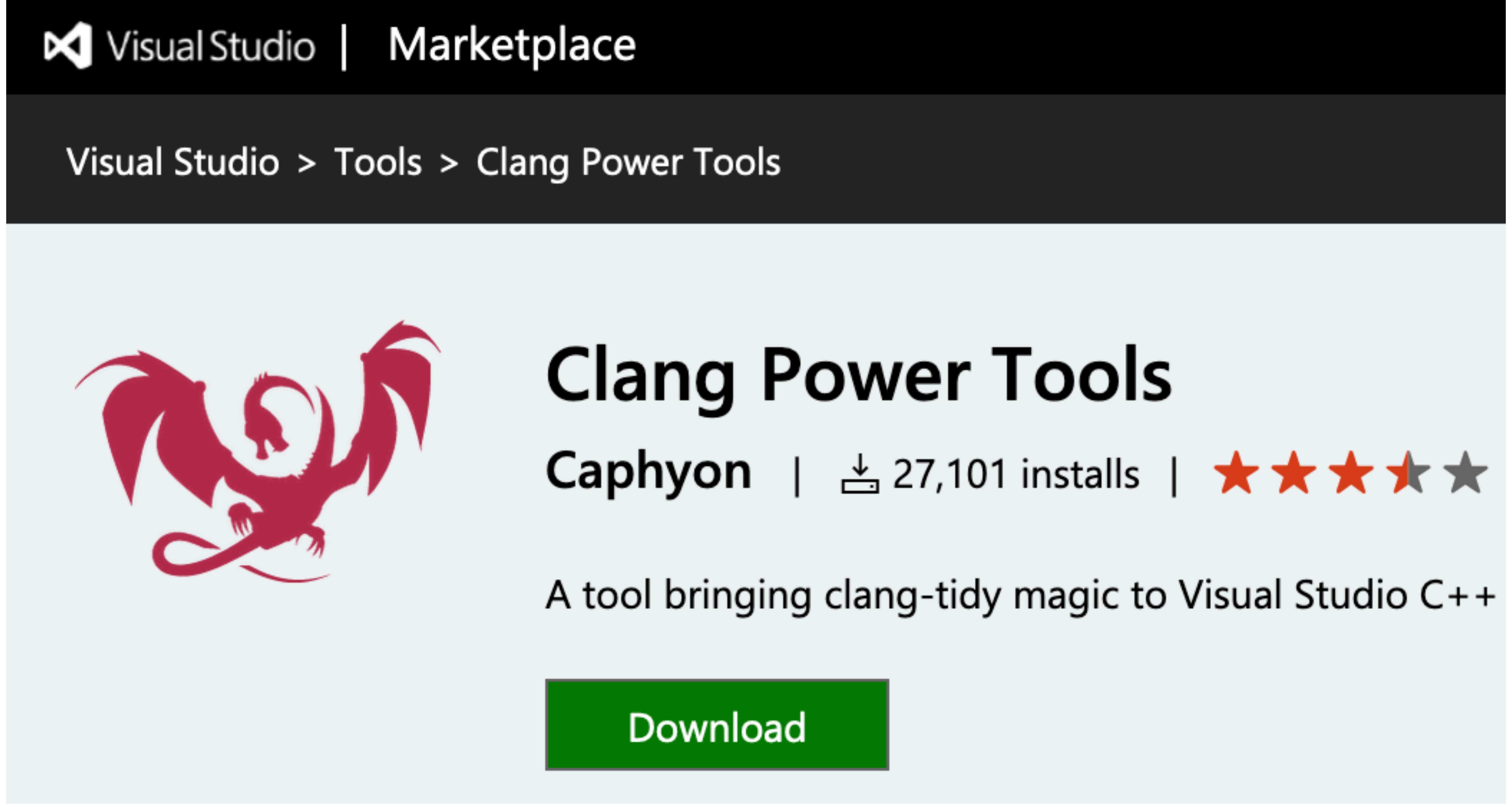
www.clangpowertools.com

LLVM

clang-tidy
clang++
clang-format

Visual Studio

2015/2017/2019



Visual Studio | Marketplace

Visual Studio > Tools > Clang Power Tools

Clang Power Tools

Caphyon | 27,101 installs |

A tool bringing clang-tidy magic to Visual Studio C++

- open-source Visual Studio extension:
  https://github.com/Caphyon/clang-power-tools
- helping developers leverage Clang/LLVM tools (`clang++`, `clang-tidy` and `clang-format`)
- perform various code transformations and fixes like **modernizing** code to C++ 11/14/17
- finding subtle latent **bugs** with its static analyzer and C++ Core Guidelines checks

www.clangpowertools.com

Clang PowerShell Script

- very configurable (many parameters)
- supports both clang compile and tidy workflows
- works directly on Visual Studio **.vcxproj** files (or MSBuild projects)

  ℹ️ **no** roundtrip transformation through Clang JSON compilation database

- supports parallel compilation
- constructs Clang PCH from VS project <stdafx.h>
- automatically extracts all necessary settings from VS projects:

  👉 preprocessor definitions, platform toolset, SDK version, include directories, PCH, etc.

`clang-build.ps1`

Using The PowerShell Script

-dir	Source directory to process for VS project files
-proj	List of projects to compile
-proj-ignore	List of projects to ignore
-file	What cpp(s) to compile from the found projects
-file-ignore	List of files to ignore
-parallel	Run clang++ in parallel mode, on all logical CPU cores
-continue	Continue project compilation even when errors occur
-clang-flags	Flags passed to clang++ driver
-tidy	Run specified clang-tidy checks
-tidy-fix	Run specified clang-tidy checks with auto-fix

...

clang-build.ps1

Using The PowerShell Script

You can run `clang-build.ps1` directly,

by specifying all required parameters (low-level control over details)

or

You can use a **configuration file** (`cpt.config`) 💡

that pre-loads some of the configurations specific for your team/project

=> store it in your *source control*

Using The PowerShell Script

```
PS>.\clang-build.ps1 -parallel
```

➡ Runs clang **compile** on all projects in current directory

```
PS>.\clang-build.ps1 -parallel -proj-ignore foo,bar
```

➡ Runs clang **compile** on all projects in current directory, except 'foo' and 'bar'

```
PS>.\clang-build.ps1 -proj foo,bar -file-ignore meow
-tidy-fix "-*,modernize-*"
```

➡ Runs **clang-tidy**, using all *modernize* checks, on all CPPs not containing 'meow' in their name, from the projects 'foo' and 'bar'.

cpt.config

```
<cpt-config>
  <clang-flags>  "-Werror"
               , "-Wall"
               , "-fms-compatibility-version=19.10"
               , "-Wmicrosoft"
               , "-Wno-invalid-token-paste"
               , "-Wno-unknown-pragmas"
               , "-Wno-unused-value"
  </clang-flags>
  <header-filter>'.*'</header-filter>
  <parallel/>
  <vs-sku>'Professional'</vs-sku>
  <file-ignore>  'htmlayoutsdk\\include\\behaviors'
               , 'vsphere\\vim25\\core'
  </file-ignore>
  <proj-ignore>  'SciLexer'
               , 'tools\\msix-psf'
  </proj-ignore>
</cpt-config>
```

Using The PowerShell Script

+

Jenkins

GitLab

Azure Pipelines
(Azure DevOps)

Using The PowerShell Script

+

Any CI/CD system with PowerShell support

Jenkins CI Configuration

Reference PowerShell script from the job working directory: `clang-build.ps1`

Build
Windows PowerShell
X
Add build step

Command

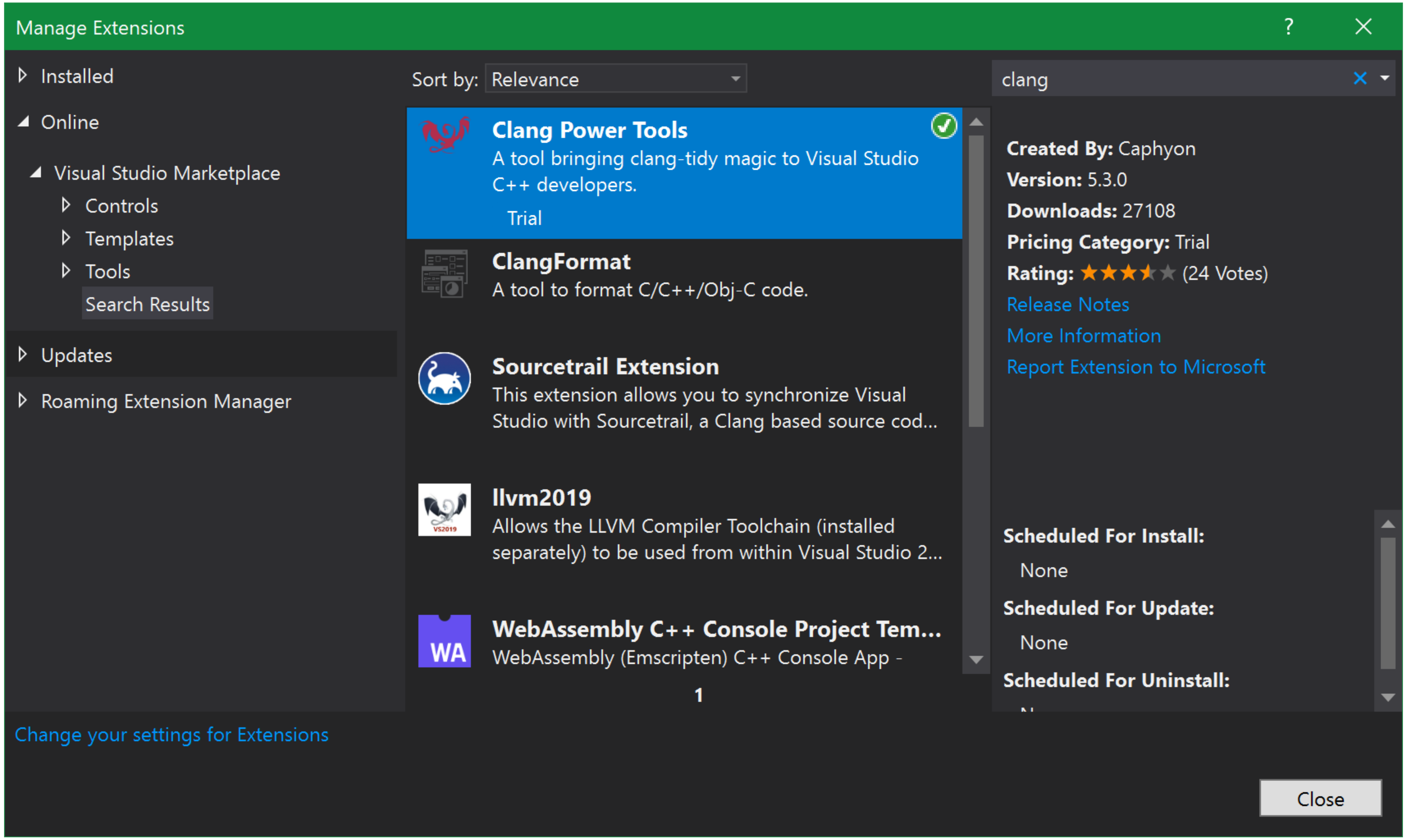
```
.\scripts\ai-clang-build.ps1 -parallel -proj-ignore LZMA.vcxproj
```

See the list of available environment variables

What About Developer Workflow?

+

Install Clang Power Tools
Visual Studio Extension



Manage Extensions

Sort by: Relevance

clang

Clang Power Tools
A tool bringing clang-tidy magic to Visual Studio C++ developers.
Trial

ClangFormat
A tool to format C/C++/Obj-C code.

Sourcetrail Extension
This extension allows you to synchronize Visual Studio with Sourcetrail, a Clang based source cod...

llvm2019
Allows the LLVM Compiler Toolchain (installed separately) to be used from within Visual Studio 2...

WebAssembly C++ Console Project Tem...
WebAssembly (Emscripten) C++ Console App -

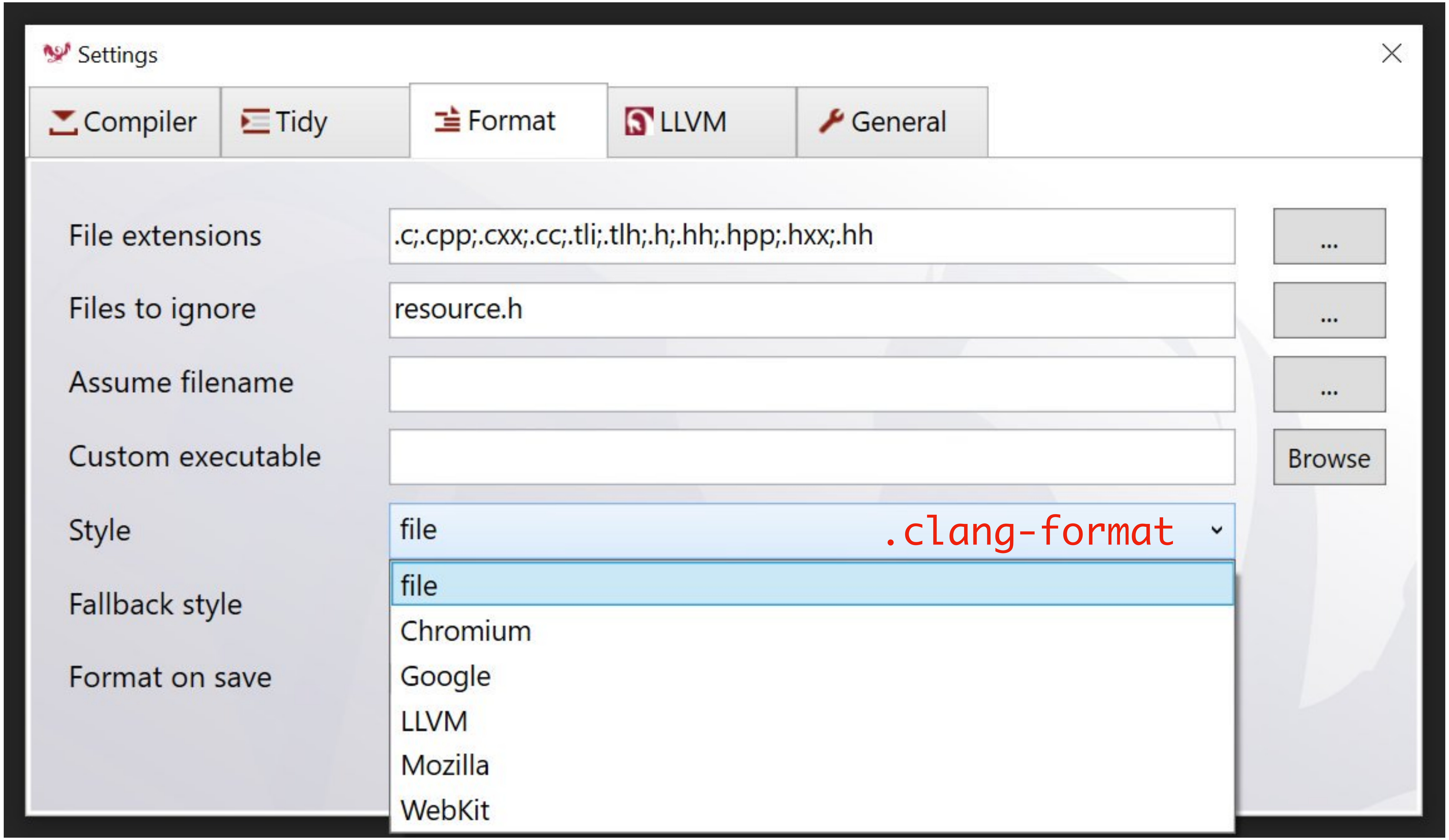
1

Created By: Caphyon
Version: 5.3.0
Downloads: 27108
Pricing Category: Trial
Rating: ★★★★☆ (24 Votes)
Release Notes
More Information
Report Extension to Microsoft

Scheduled For Install:
None
Scheduled For Update:
None
Scheduled For Uninstall:

Change your settings for Extensions

Clang Format



Settings

File extensions .c;.cpp;.cxx;.cc;.tli;.tlh;.h;.hh;.hpp;.hxx;.hh

Files to ignore resource.h

Assume filename

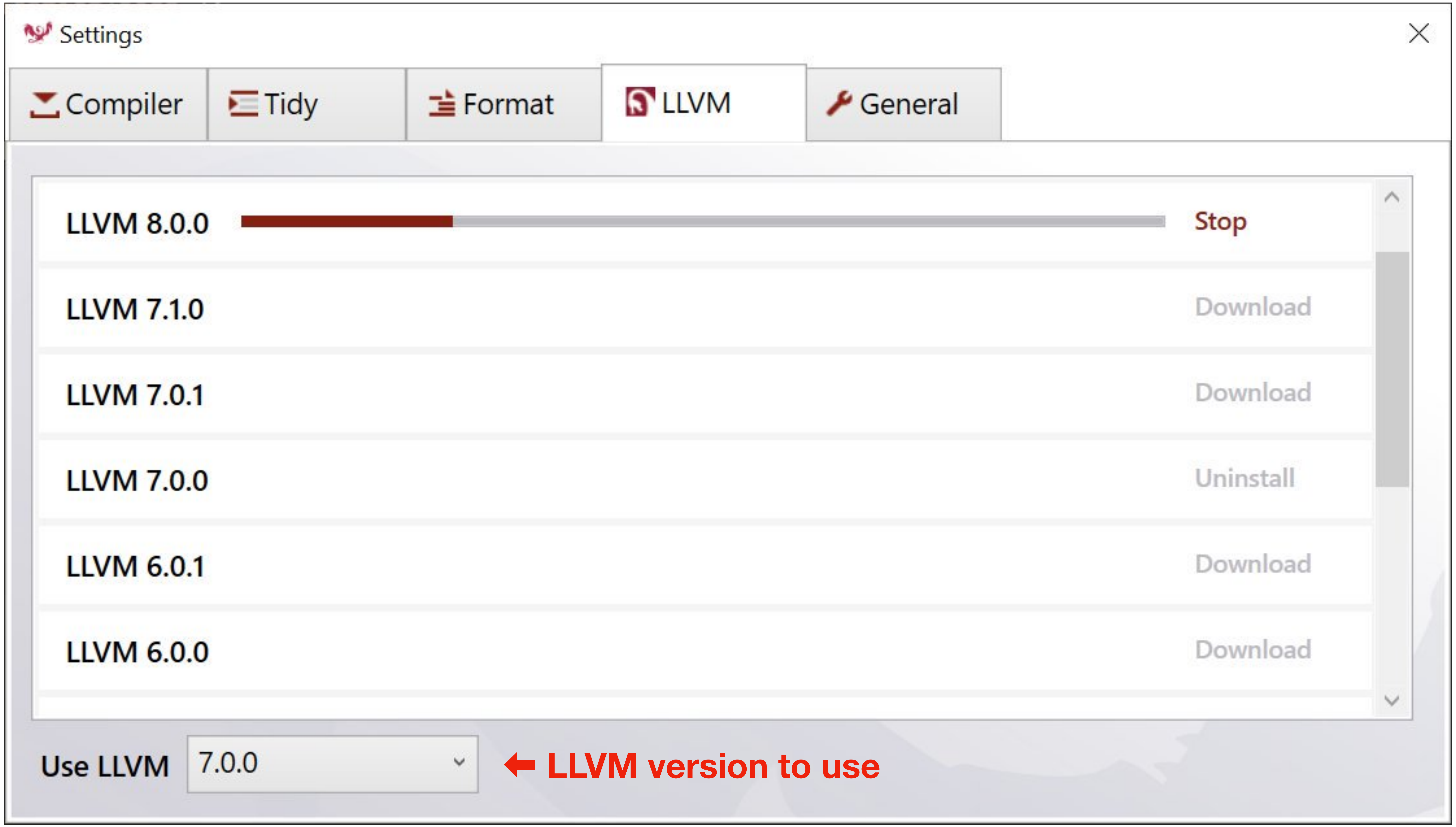
Custom executable

Style file

Fallback style

Format on save

Install side-by-side LLVM versions



Settings

Compiler | Tidy | Format | LLVM | General

LLVM 8.0.0 Stop

LLVM 7.1.0 Download

LLVM 7.0.1 Download

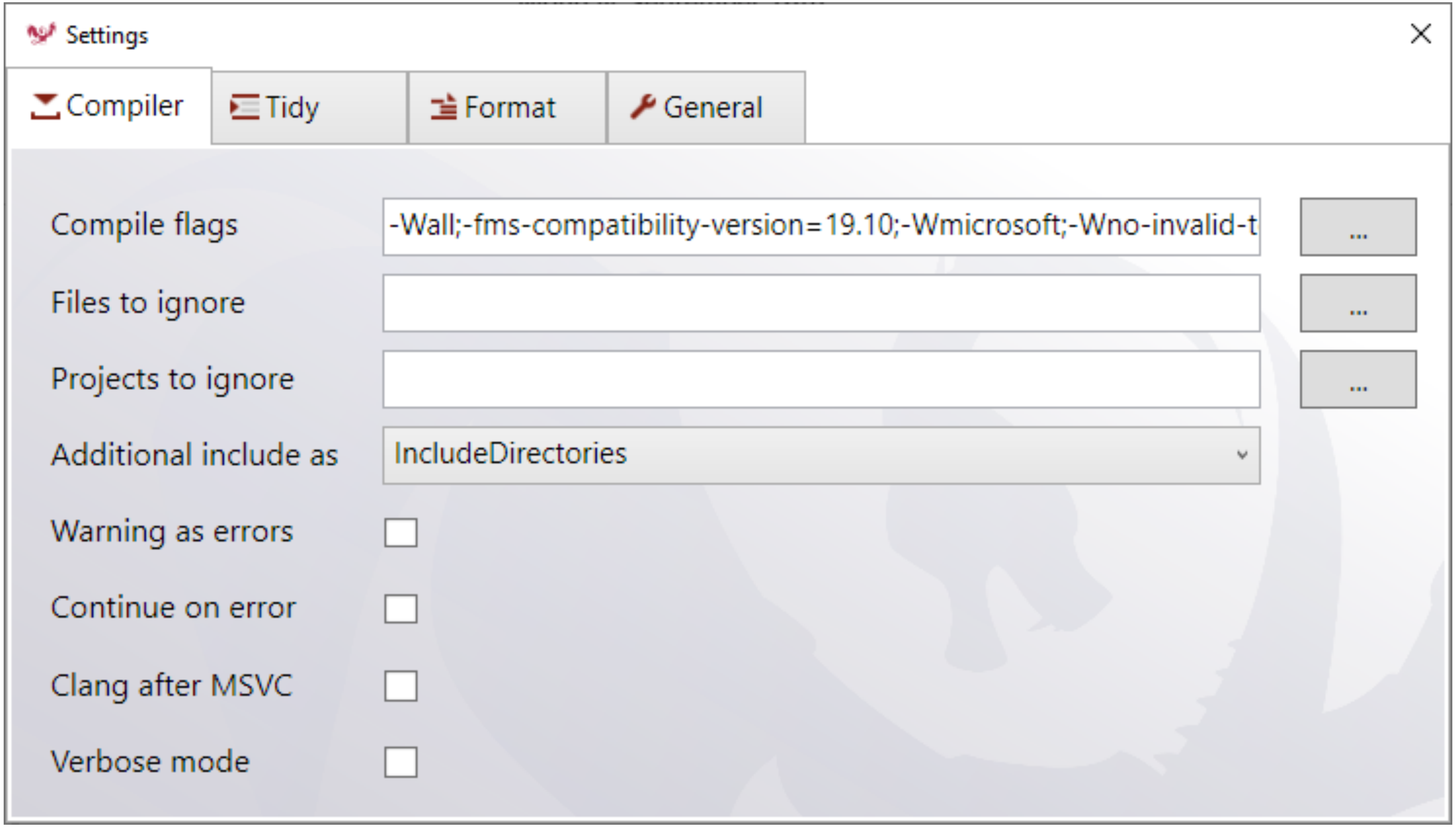
LLVM 7.0.0 Uninstall

LLVM 6.0.1 Download

LLVM 6.0.0 Download

Use LLVM 7.0.0

clang++ compilation flags

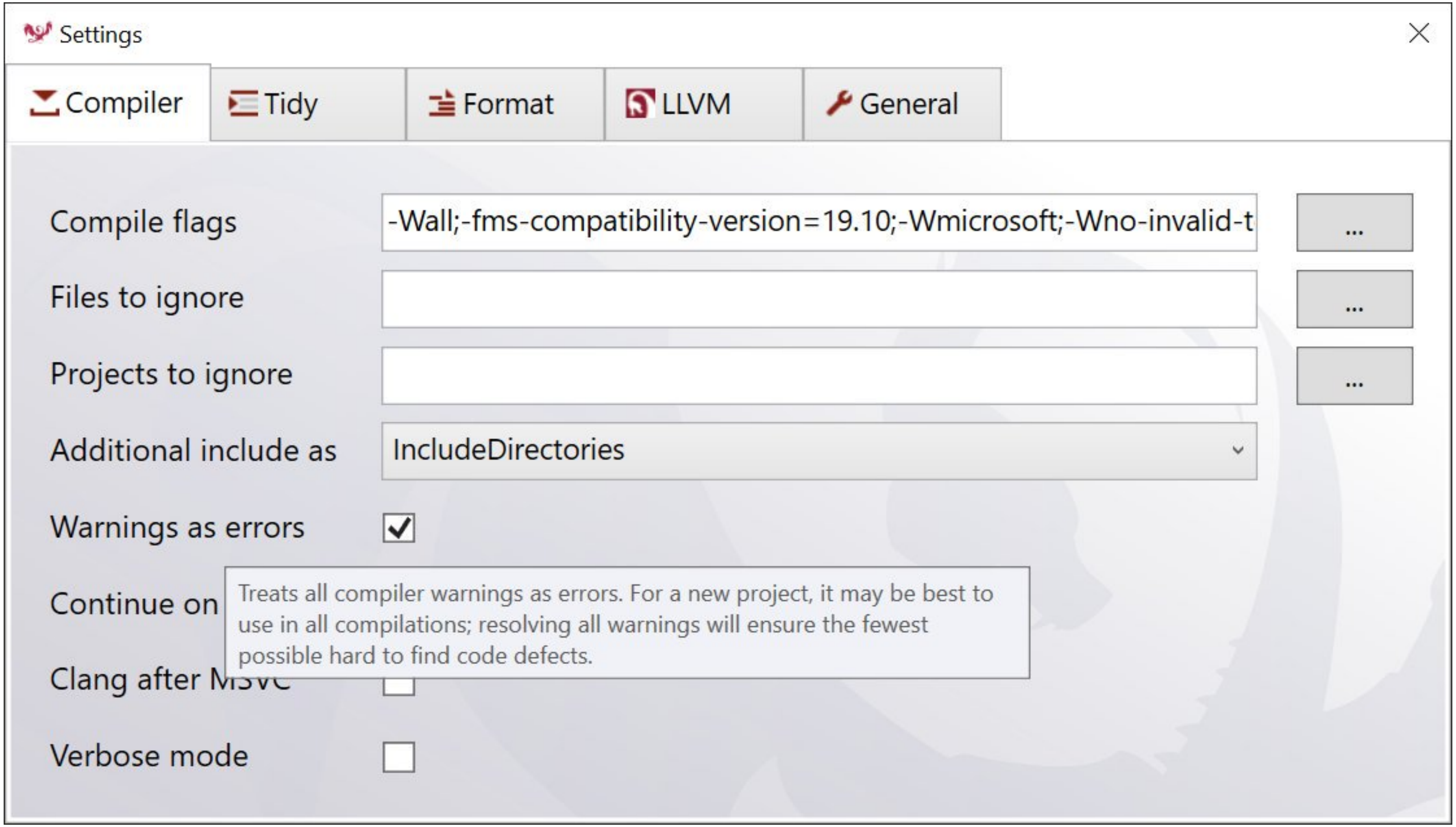


Settings

Compiler | Tidy | Format | General

Compile flags	-Wall;-fms-compatibility-version=19.10;-Wmicrosoft;-Wno-invalid-t
Files to ignore	
Projects to ignore	
Additional include as	IncludeDirectories
Warning as errors	☐
Continue on error	☐
Clang after MSVC	☐
Verbose mode	☐

-Werror /WX



Settings

Compile flags: -Wall;-fms-compatibility-version=19.10;-Wmicrosoft;-Wno-invalid-t

Files to ignore:

Projects to ignore:

Additional include as: IncludeDirectories

Warnings as errors: ✓

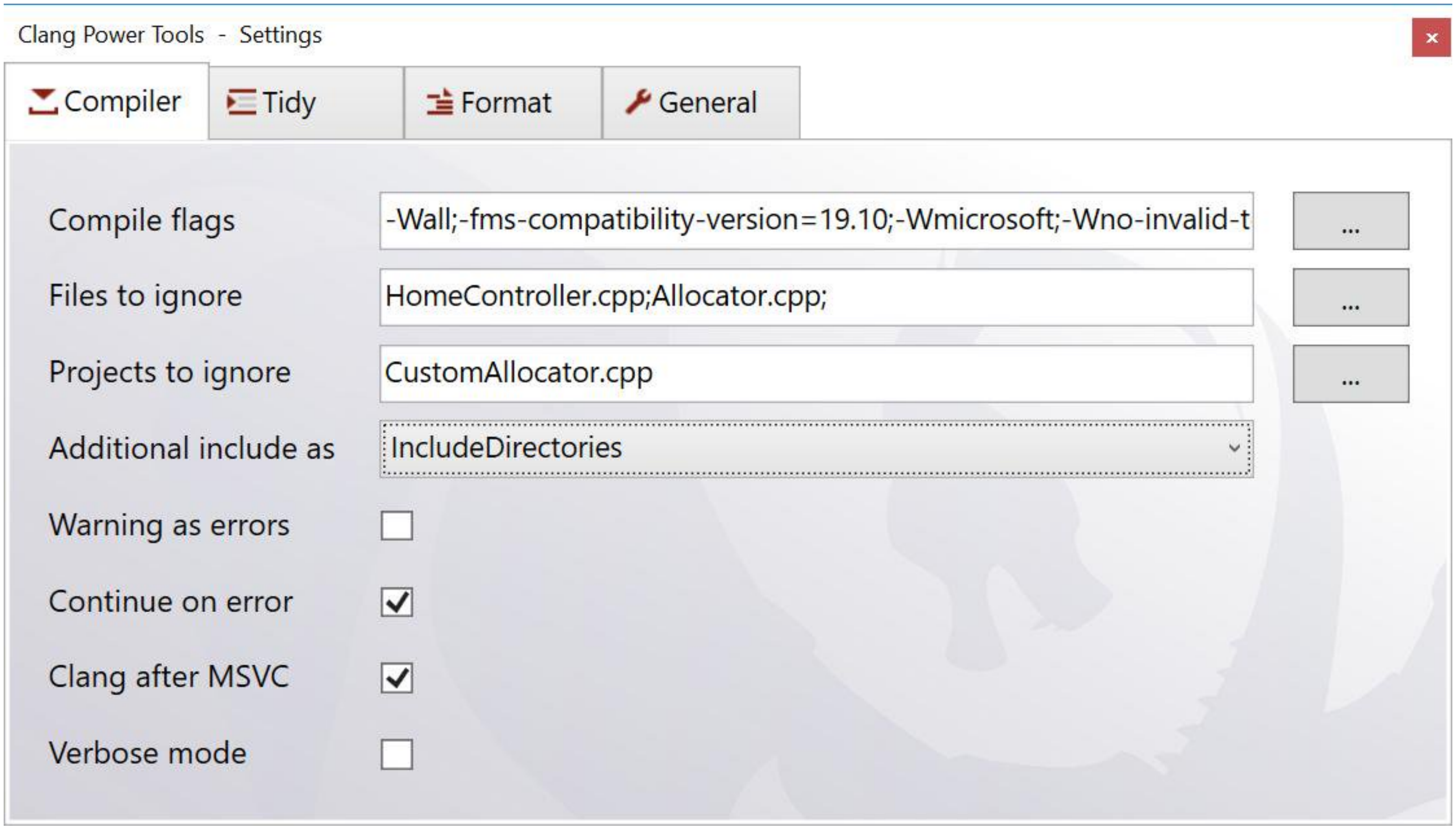
Continue on

Clang after MSVC

Verbose mode: ☐

Treats all compiler warnings as errors. For a new project, it may be best to use in all compilations; resolving all warnings will ensure the fewest possible hard to find code defects.

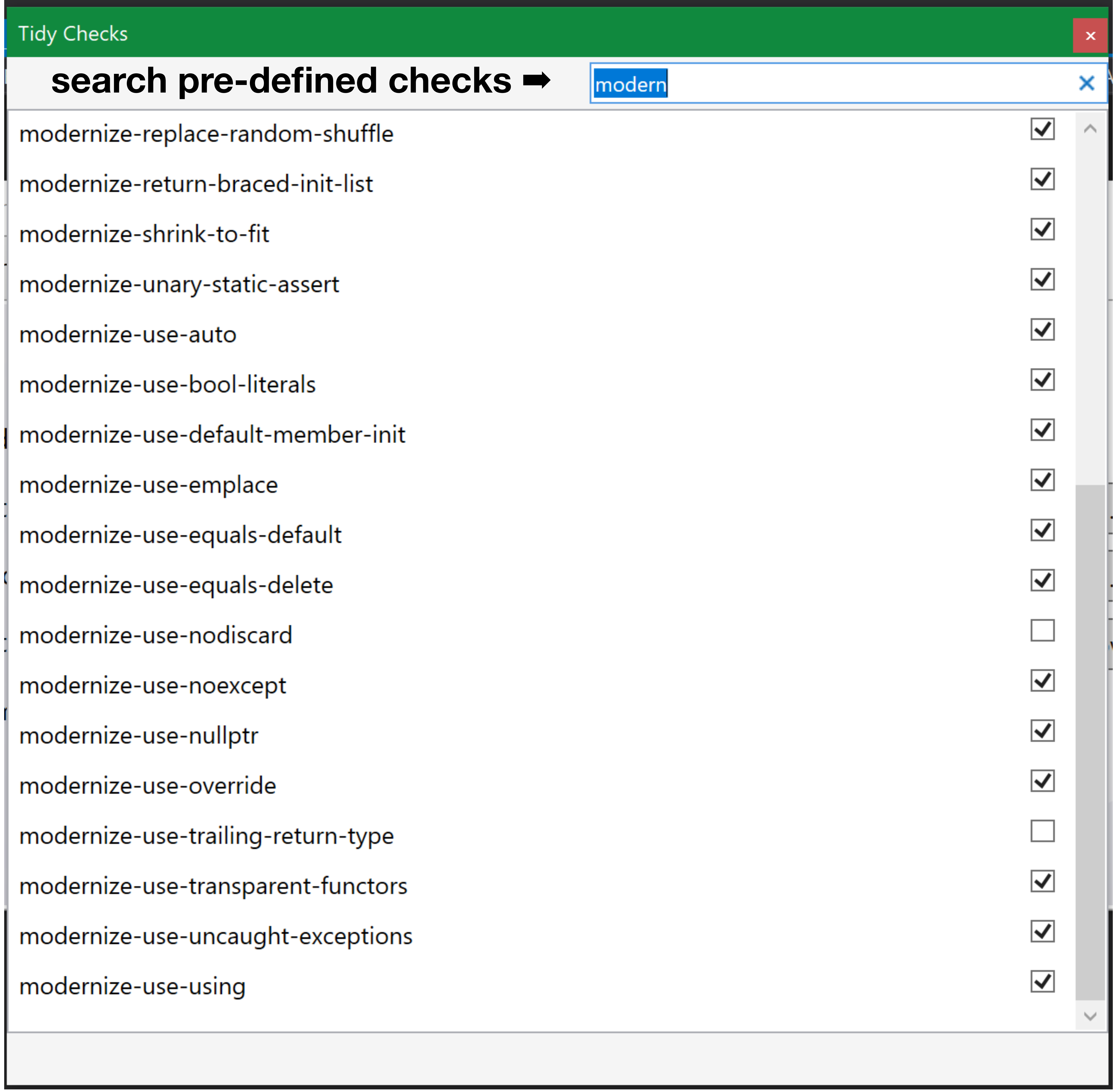
Auto Clang compile after MSVC compile



Clang Power Tools - Settings

Compile flags	-Wall;-fms-compatibility-version=19.10;-Wmicrosoft;-Wno-invalid-t
Files to ignore	HomeController.cpp;Allocator.cpp;
Projects to ignore	CustomAllocator.cpp
Additional include as	IncludeDirectories
Warning as errors	☐
Continue on error	✓
👉 Clang after MSVC	✓
Verbose mode	☐

clang-tidy built-in checks

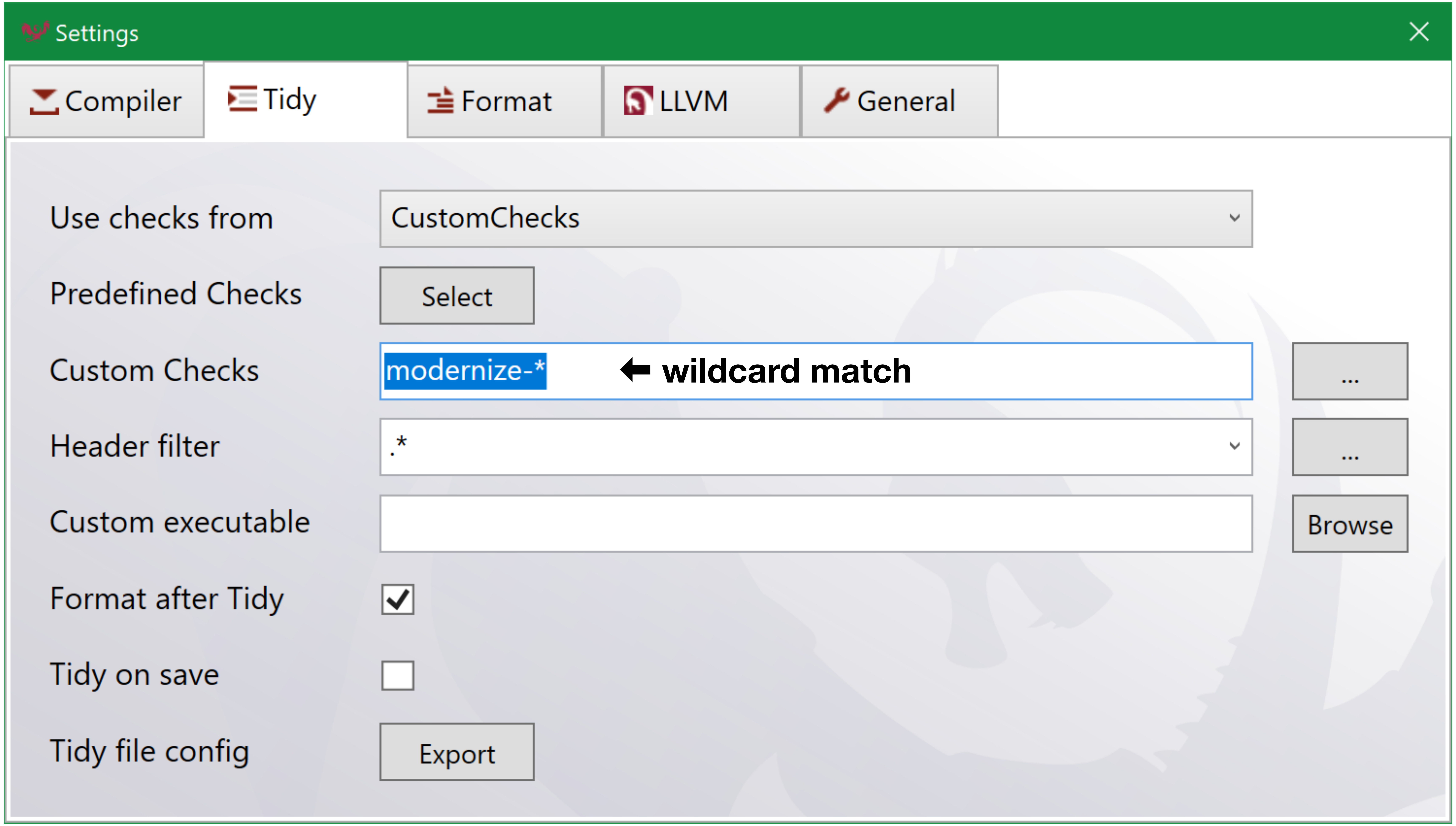


Tidy Checks

search pre-defined checks ➡ modern

modernize-replace-random-shuffle	✓
modernize-return-braced-init-list	✓
modernize-shrink-to-fit	✓
modernize-unary-static-assert	✓
modernize-use-auto	✓
modernize-use-bool-literals	✓
modernize-use-default-member-init	✓
modernize-use-emplace	✓
modernize-use-equals-default	✓
modernize-use-equals-delete	✓
modernize-use-nodiscard	☐
modernize-use-noexcept	✓
modernize-use-nullptr	✓
modernize-use-override	✓
modernize-use-trailing-return-type	☐
modernize-use-transparent-functors	✓
modernize-use-uncaught-exceptions	✓
modernize-use-using	✓

clang-tidy custom checks



Use checks from: CustomChecks

Predefined Checks: Select

Custom Checks: modernize-* ⬅ wildcard match

Header filter: .*

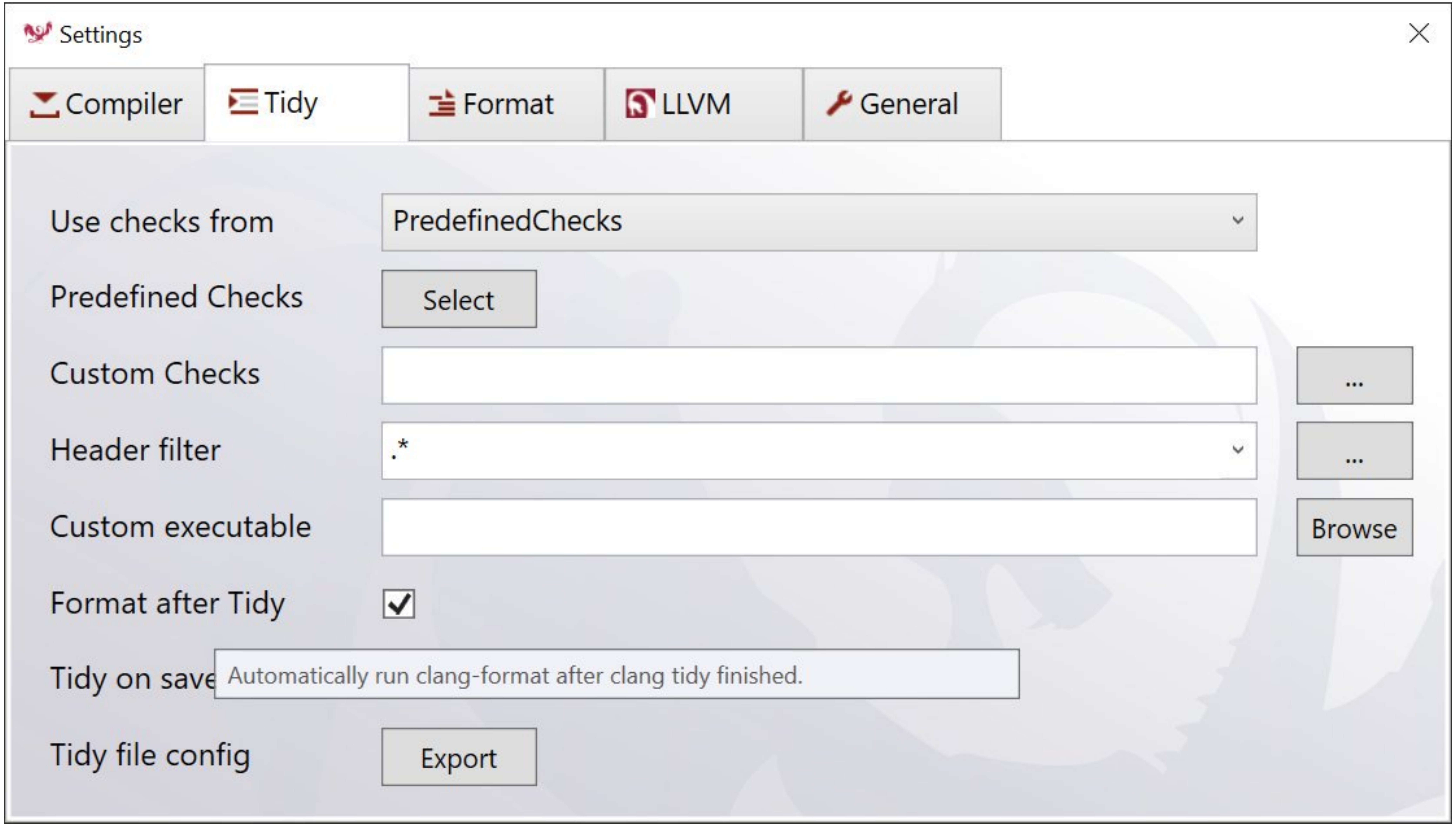
Custom executable:

Format after Tidy: ✓

Tidy on save: ☐

Tidy file config: Export

Run clang-format after tidy auto-fixes



Use checks from: PredefinedChecks

Predefined Checks: Select

Custom Checks:

Header filter: .*

Custom executable:

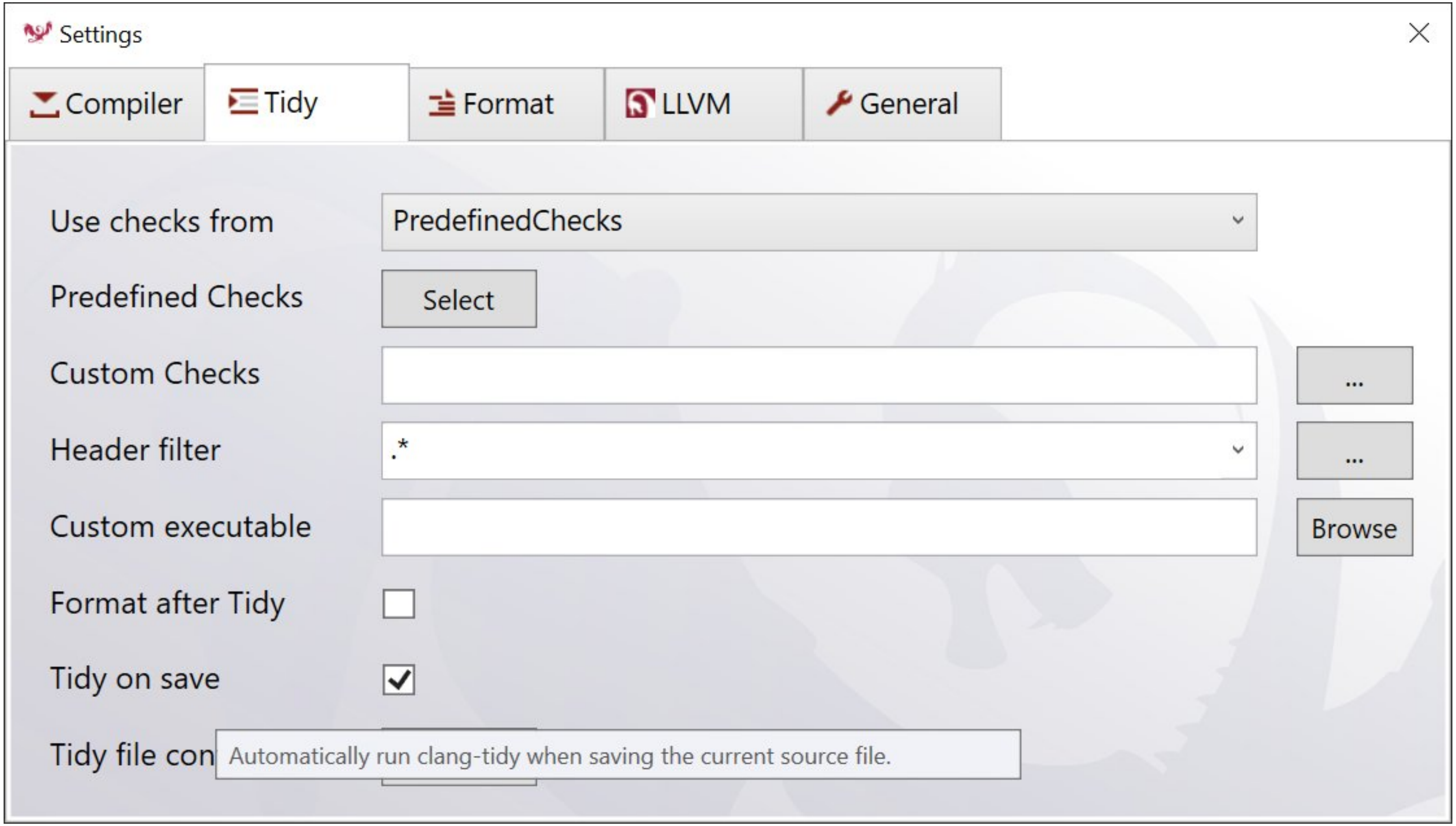
Format after Tidy: ✓

Tidy on save

Automatically run clang-format after clang tidy finished.

Tidy file config: Export

Auto apply tidy fixes as you edit/save

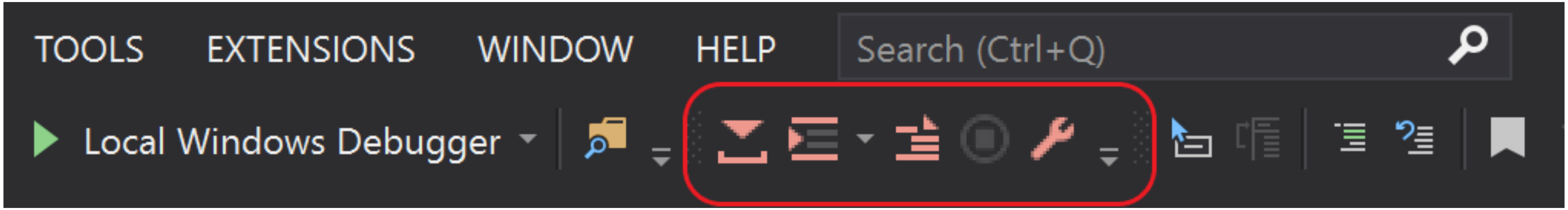

Settings

Compiler | Tidy | Format | LLVM | General

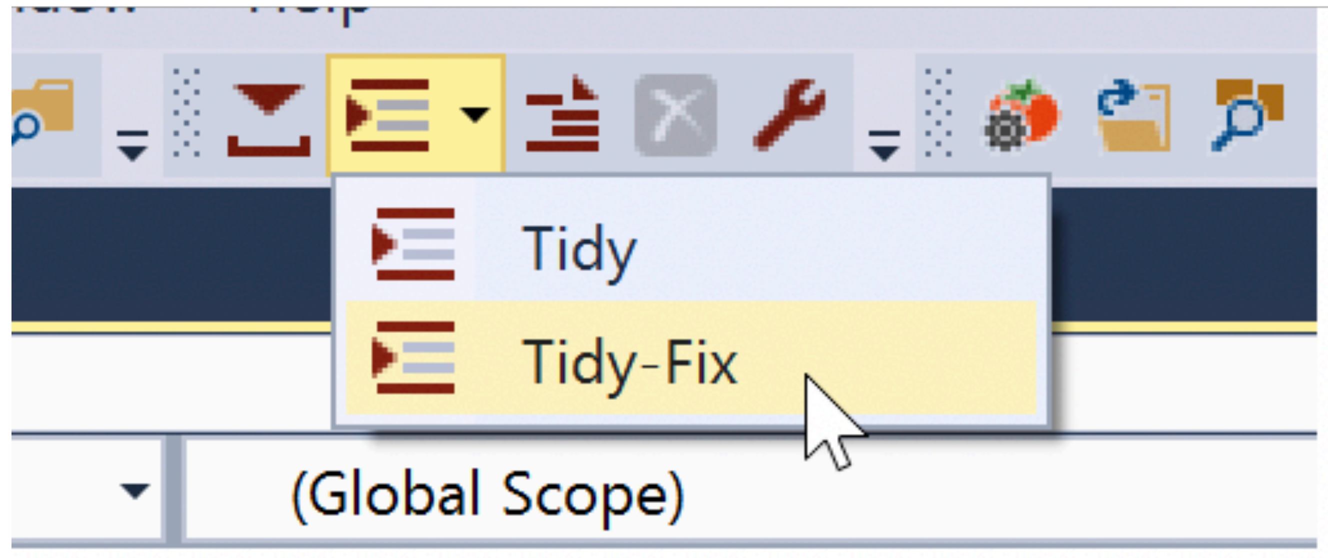
Use checks from	PredefinedChecks
Predefined Checks	Select
Custom Checks	
Header filter	.*
Custom executable	
Format after Tidy	☐
Tidy on save	☑
Tidy file con	

Automatically run clang-tidy when saving the current source file.

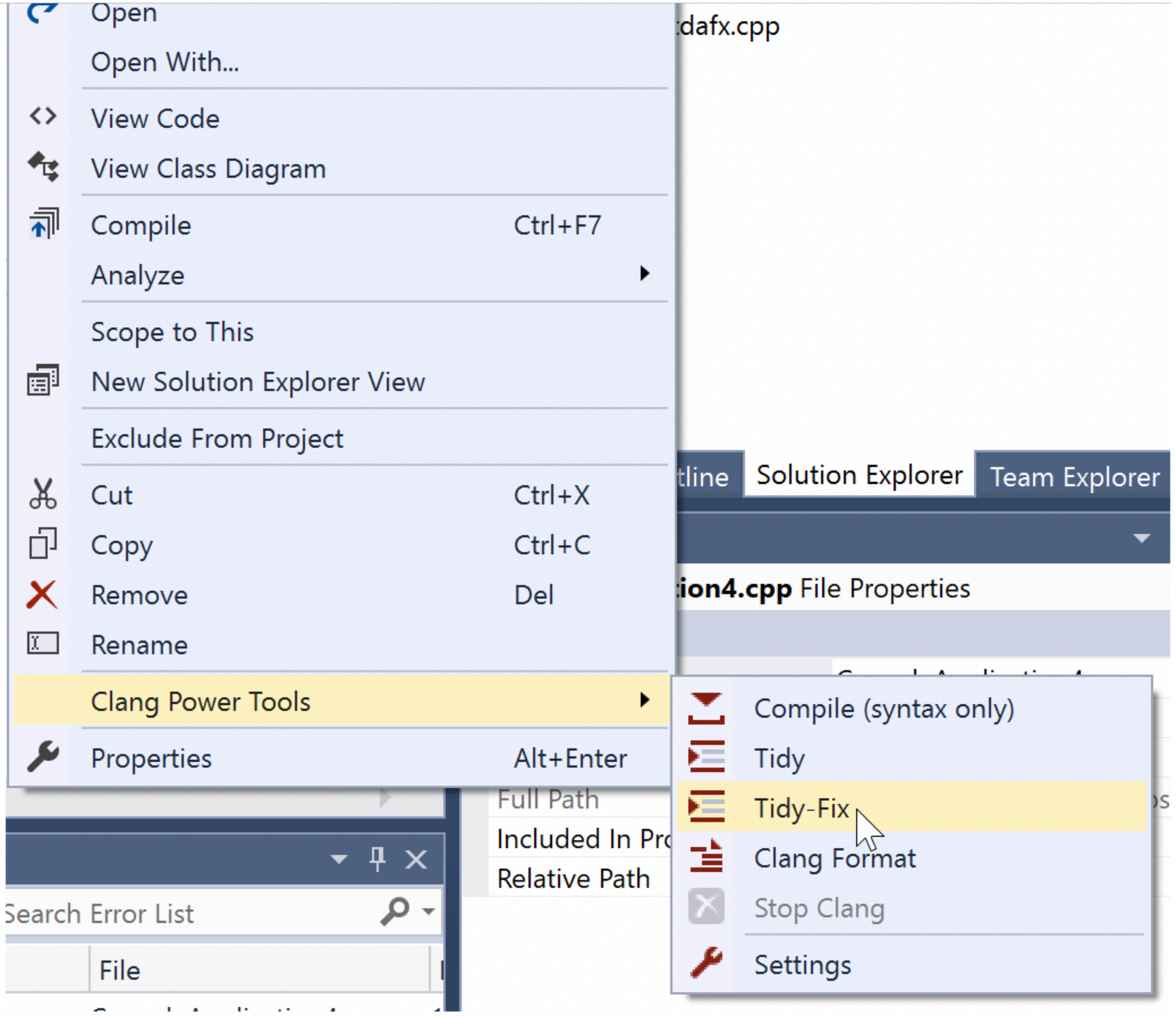
Clang Power Tools
toolbar



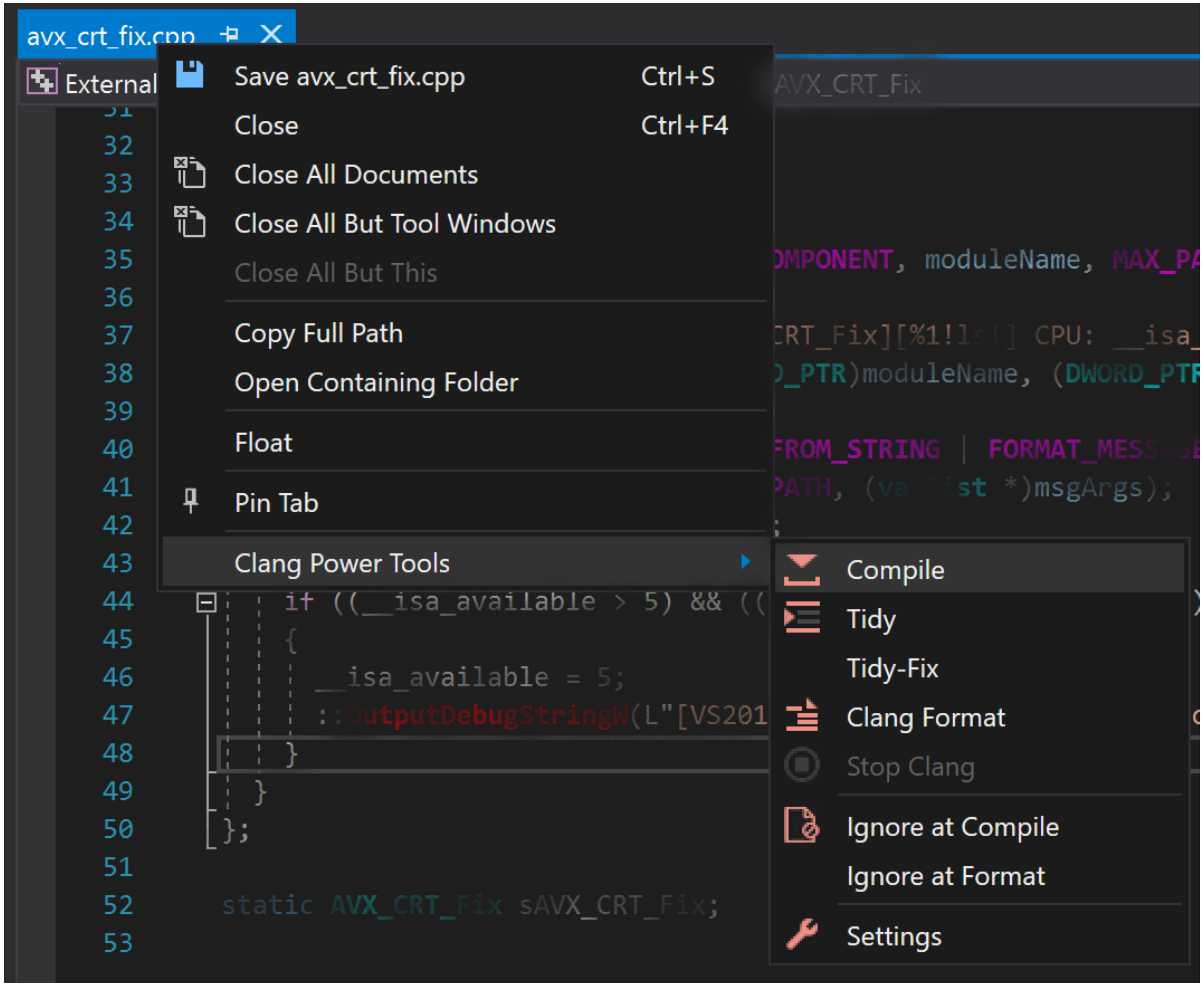
Run Clang Power Tools on a whole *project* or *solution*



Solution Explorer

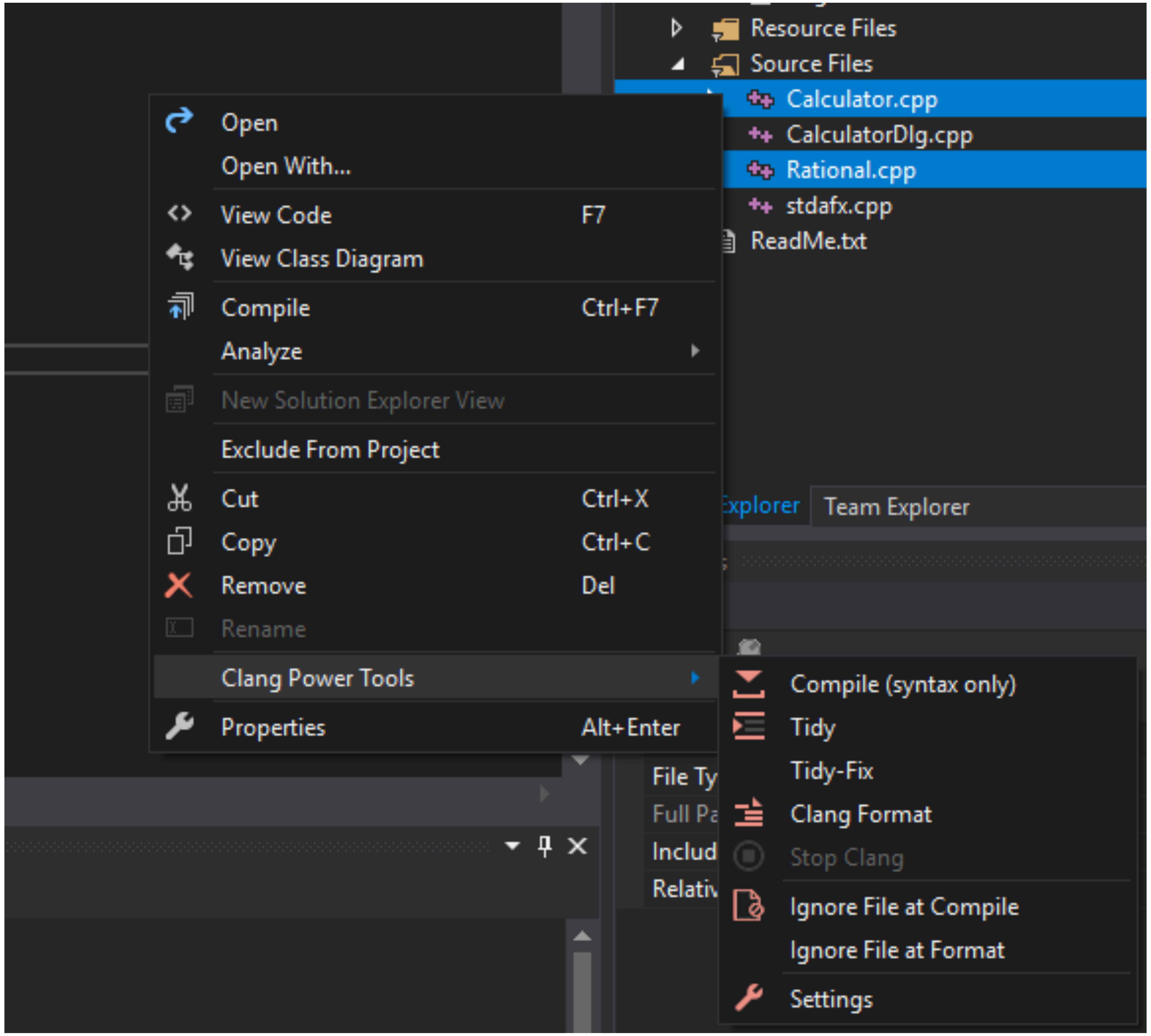


Run Clang Power Tools on a *source file* (tab-menu)

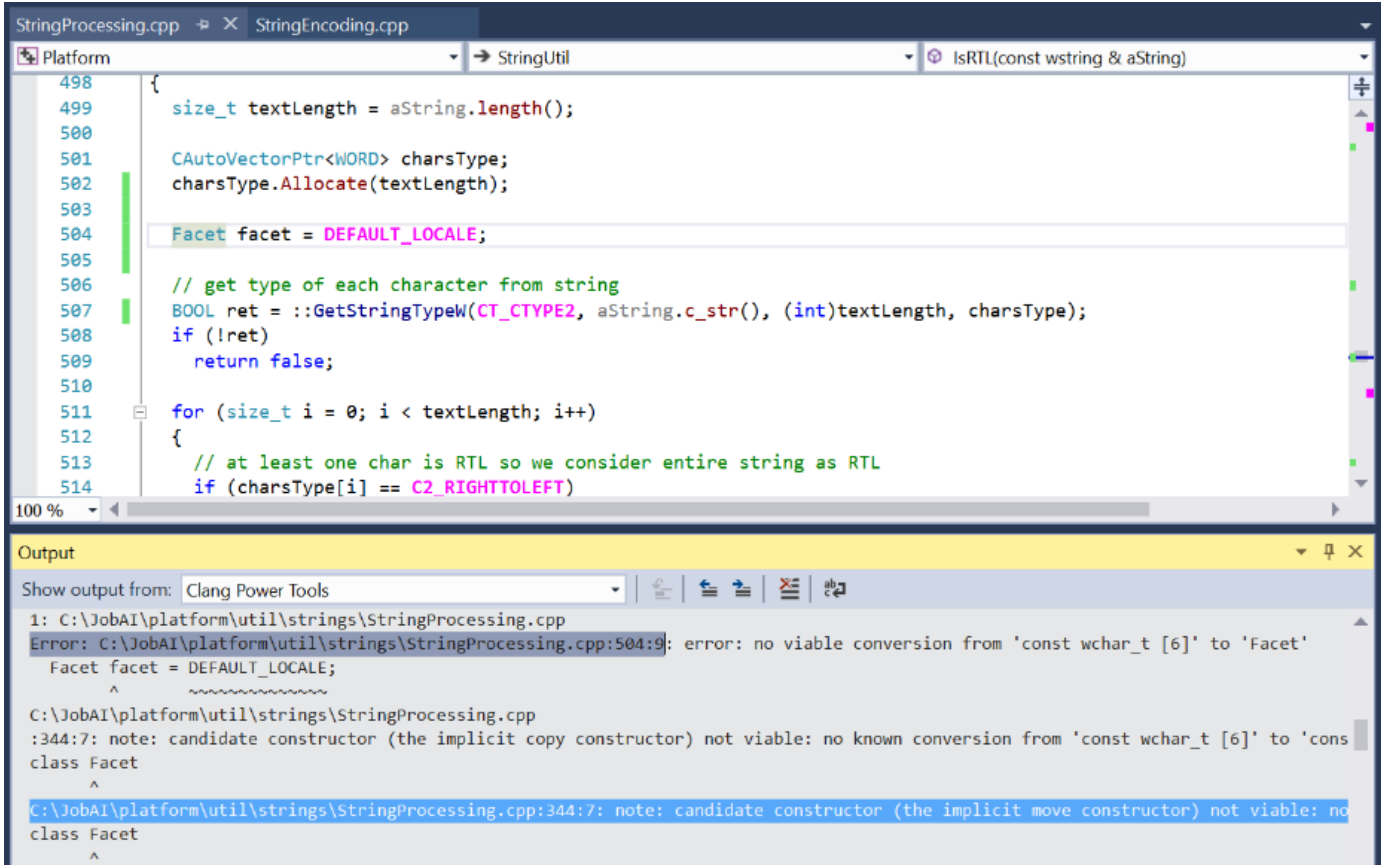


```
        if ((__isa_available > 5) && ((
        {
            __isa_available = 5;
            ::OutputDebugStringW(L"[VS201
        }
    }
};

static AVX_CRT_Fix sAVX_CRT_Fix;
```

Clang Power Tools on selected *source files* (Solution Explorer)



Clang Output



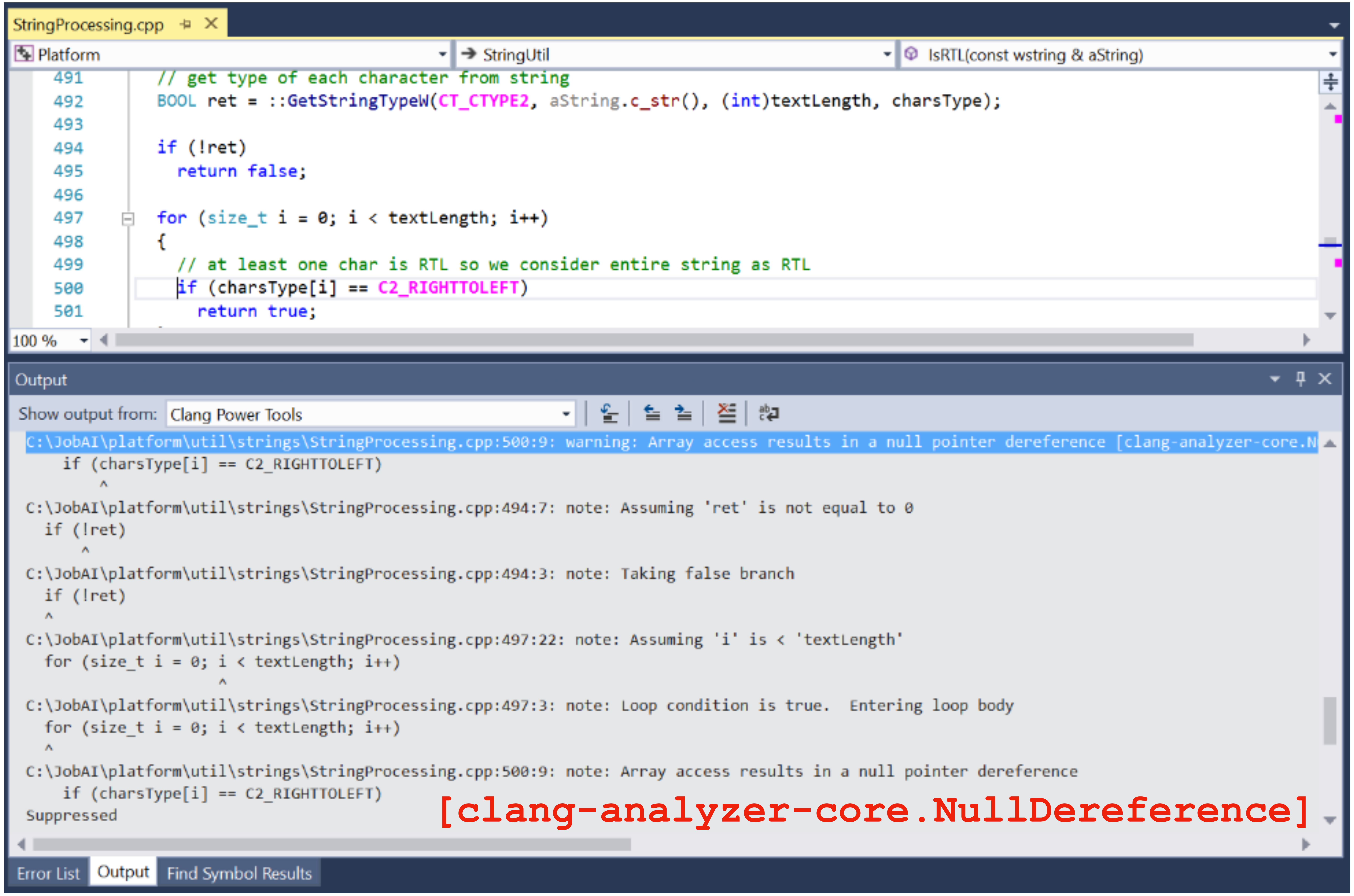
```
{
  size_t textLength = aString.length();

  CAutoVectorPtr<WORD> charsType;
  charsType.Allocate(textLength);

  Facet facet = DEFAULT_LOCALE;

  // get type of each character from string
  BOOL ret = ::GetStringTypeW(CT_CTYPE2, aString.c_str(), (int)textLength, charsType);
  if (!ret)
    return false;

  for (size_t i = 0; i < textLength; i++)
  {
    // at least one char is RTL so we consider entire string as RTL
    if (charsType[i] == C2_RIGHTTOLEFT)
```

```
1: C:\JobAI\platform\util\strings\StringProcessing.cpp
Error: C:\JobAI\platform\util\strings\StringProcessing.cpp:504:9: error: no viable conversion from 'const wchar_t [6]' to 'Facet'
  Facet facet = DEFAULT_LOCALE;
        ^       ~~~~~~~~~~~~~~
C:\JobAI\platform\util\strings\StringProcessing.cpp
:344:7: note: candidate constructor (the implicit copy constructor) not viable: no known conversion from 'const wchar_t [6]' to 'cons
class Facet
      ^
C:\JobAI\platform\util\strings\StringProcessing.cpp:344:7: note: candidate constructor (the implicit move constructor) not viable: no
class Facet
      ^
```

⬅ **Clang compile error**

🔥

Clang tidy static analysis report



StringProcessing.cpp

Platform | StringUtil | IsRTL(const wstring & aString)

```
    // get type of each character from string
    BOOL ret = ::GetStringTypeW(CT_CTYPE2, aString.c_str(), (int)textLength, charsType);

    if (!ret)
      return false;

    for (size_t i = 0; i < textLength; i++)
    {
      // at least one char is RTL so we consider entire string as RTL
      if (charsType[i] == C2_RIGHTTOLEFT)
        return true;
```

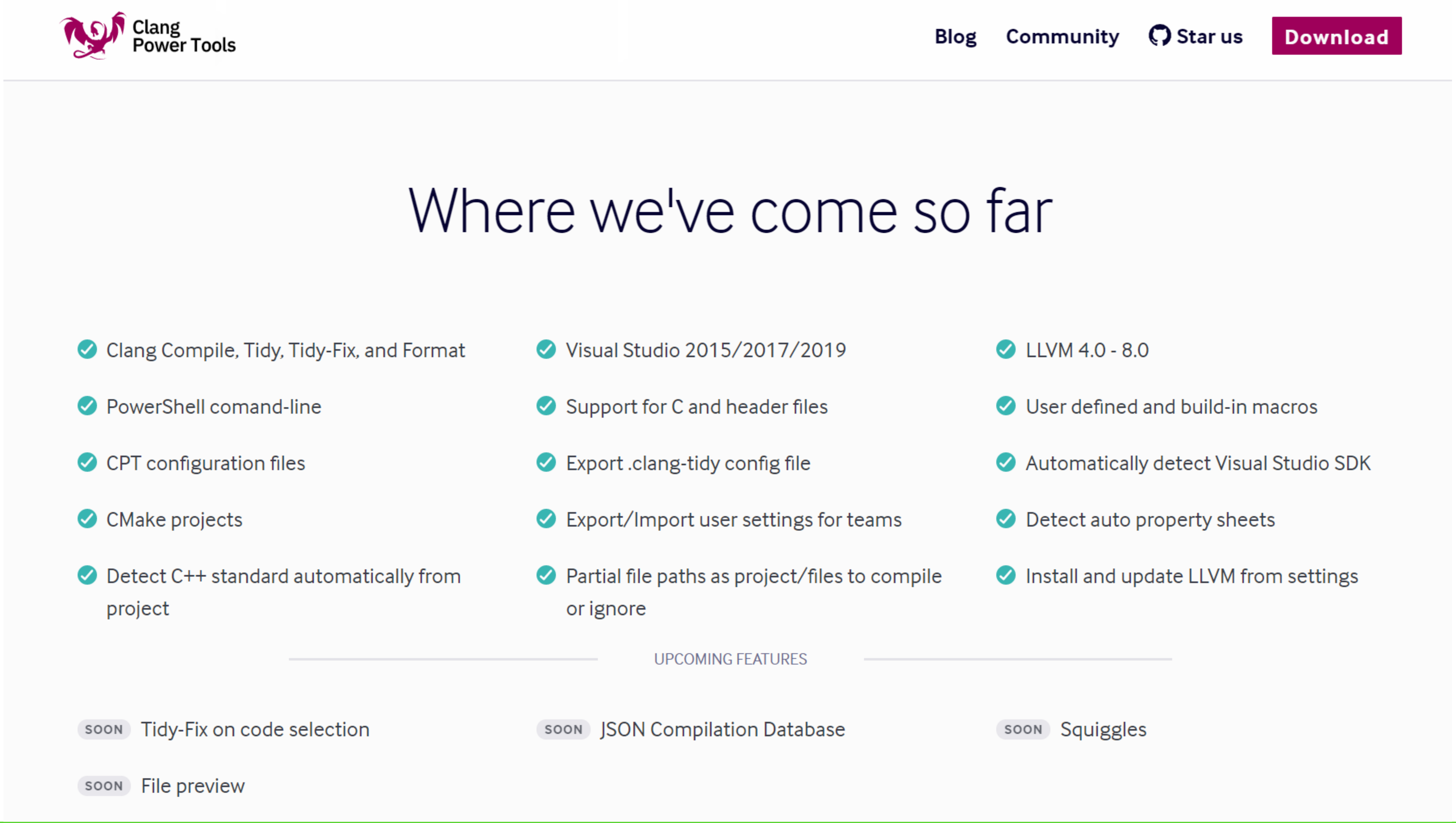
Output

Show output from: Clang Power Tools

```
C:\JobAI\platform\util\strings\StringProcessing.cpp:500:9: warning: Array access results in a null pointer dereference [clang-analyzer-core.N
    if (charsType[i] == C2_RIGHTTOLEFT)
        ^
C:\JobAI\platform\util\strings\StringProcessing.cpp:494:7: note: Assuming 'ret' is not equal to 0
  if (!ret)
      ^
C:\JobAI\platform\util\strings\StringProcessing.cpp:494:3: note: Taking false branch
  if (!ret)
  ^
C:\JobAI\platform\util\strings\StringProcessing.cpp:497:22: note: Assuming 'i' is < 'textLength'
  for (size_t i = 0; i < textLength; i++)
                     ^
C:\JobAI\platform\util\strings\StringProcessing.cpp:497:3: note: Loop condition is true.  Entering loop body
  for (size_t i = 0; i < textLength; i++)
  ^
C:\JobAI\platform\util\strings\StringProcessing.cpp:500:9: note: Array access results in a null pointer dereference
    if (charsType[i] == C2_RIGHTTOLEFT)
Suppressed
```

[clang-analyzer-core.NullDereference]

Error List | Output | Find Symbol Results



Where we've come so far

- ✓ Clang Compile, Tidy, Tidy-Fix, and Format
- ✓ Visual Studio 2015/2017/2019
- ✓ LLVM 4.0 - 8.0
- ✓ PowerShell comand-line
- ✓ Support for C and header files
- ✓ User defined and build-in macros
- ✓ CPT configuration files
- ✓ Export .clang-tidy config file
- ✓ Automatically detect Visual Studio SDK
- ✓ CMake projects
- ✓ Export/Import user settings for teams
- ✓ Detect auto property sheets
- ✓ Detect C++ standard automatically from project
- ✓ Partial file paths as project/files to compile or ignore
- ✓ Install and update LLVM from settings

UPCOMING FEATURES

- SOON Tidy-Fix on code selection
- SOON JSON Compilation Database
- SOON Squiggles
- SOON File preview

Why Do I Care ?

16 year old code base under active development
3.5 million lines of C++ code
a few brave nerds…

or

“How we managed to **clang-tidy** our whole code base,
while maintaining our monthly release cycle”

https://www.youtube.com/watch?v=Wl-9ozmxXbo

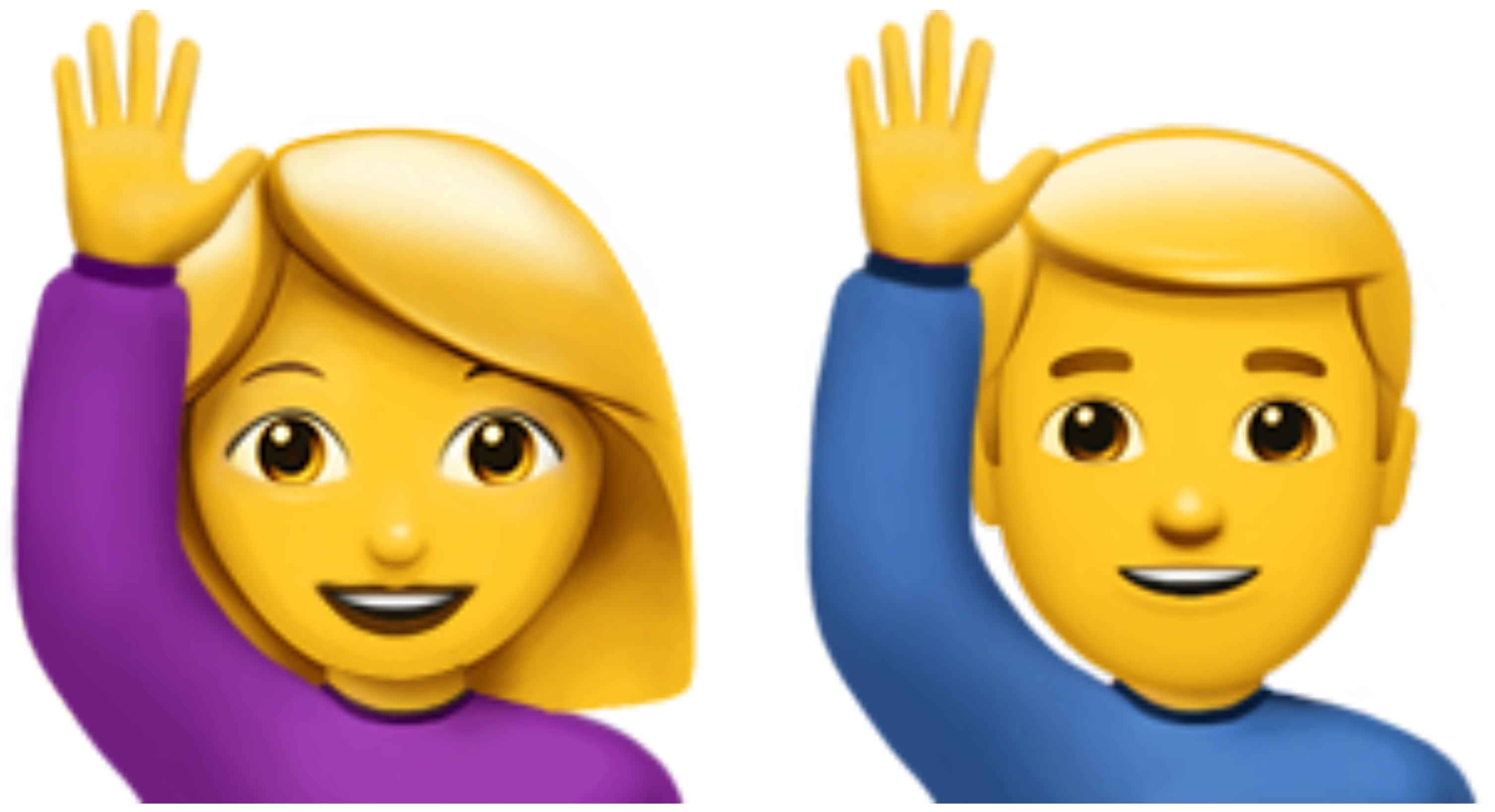
Part II

Legacy Code

Mandatory Slide

Gauging the audience...

C++98/03　　C++11　　C++14　　C++17



Why do we need this ?

ISO C++ standard conformance

Finding bugs

ISO C++ standard conformance

MSVC* `/permissive-`

Problem: older Windows SDKs

*** starting with Visual Studio 2017**

https://docs.microsoft.com/en-us/cpp/build/reference/permissive-standards-conformance?view=vs-2019

ISO C++ standard conformance

Latest **MSVC** STL

Compiles/requires Clang 8

https://docs.microsoft.com/en-us/cpp/build/reference/permissive-standards-conformance?view=vs-2019

Goals

- Experiment with **clang-tidy** checks / static analysis
- Getting all our code to fully ***compile*** with Clang, using the correct VS project settings
- We found several compatibility issues between MSVC compiler and Clang
- Note that we were already using MSVC **/W4** and **/WX** on all our projects

Goals

- Welcome to the land of **non-standard C++** language extensions and striving for C++ ISO conformance in our code
- We started **fixing** all non-conformant code... (some automation required)
- Perform large scale **refactorings** on our code with clang-tidy:
  `modernize-*, readability-*`
- Run **static analysis** on our code base to find subtle latent bugs
- Switch to the new MSVC compiler: `/permissive-`

Fixes, fixes, fixes...

🔥 **Just a few examples:**

```
Error: delete called on non-final 'AppPathVar' that has virtual functions
but non-virtual destructor [-Werror,-Wdelete-non-virtual-dtor]
```

```
Error: 'MsiComboBoxTable::PreRowChange' hides overloaded virtual function
[-Werror,-Woverloaded-virtual]
  void PreRowChange(const IMsiRow & aRow, BitField aModifiedContext);
```

```
Error: variable 'it' is incremented both in the loop header and in the
loop body [-Werror,-Wfor-loop-analysis]
```

Fixes, fixes, fixes...

🔥 **Just a few examples:**

```
Error: moving a temporary object prevents copy elision
[-Werror,-Wpessimizing-move]
  : GenericPath(move(UnboxHugePath(aPath)))
```

```
Error: moving a local object in a return statement prevents copy elision
[-Werror,-Wpessimizing-move]
  return move(replacedConnString);
```

Fixes, fixes, fixes...

Just a few examples:

```
Error: field 'mCommandContainer' will be initialized after field
'mRepackBuildType' [-Werror,-Wreorder]
```

```
Error: PipeServer.cpp:42:39: error: missing field 'InternalHigh' initializer
[-Werror,-Wmissing-field-initializers]
```

Fixes, fixes, fixes...

```
StringProcessing.cpp:504:9: error: no viable conversion from
'const wchar_t [6]' to 'Facet'
  Facet facet = DEFAULT_LOCALE;
        ^       ~~~~~~~~~~~~~~

StringProcessing.cpp:344:7: note: candidate constructor (the implicit copy
constructor) not viable: no known conversion from
'const wchar_t [6]' to 'const Facet &' for 1st argument
class Facet
      ^

StringProcessing.cpp:349:3: note: candidate constructor not viable: no known
conversion from 'const wchar_t [6]' to 'const std::wstring &' for 1st argument
  Facet(const wstring & facet)
  ^
```

🔥 **Frequent offender:** **Two user-defined conversions needed**

Fixes, fixes, fixes...

```
Error: destructor called on non-final 'InternalMessageGenerator' that has
virtual functions but non-virtual destructor
[-Werror,-Wdelete-non-virtual-dtor]
                _Getptr()->~_Ty();
                ^

MessageCenter.cpp:49:29: note: in instantiation of function template
specialization 'std::make_shared<InternalMessageGenerator>' requested here
    mInternalMsgGenerator = make_shared<InternalMessageGenerator>(...);
                            ^

...\VC\Tools\MSVC\include\memory:1783:15: note: qualify call to silence
this warning
                _Getptr()->~_Ty();
```

🔥 **Frequent offender**

Fixes, fixes, fixes...

```
Error: delete called on 'NetFirewall::INetFirewallMgr' that is abstract but has
non-virtual destructor [-Werror,-Wdelete-non-virtual-dtor]
                delete _Ptr;
                ^
...\VC\Tools\MSVC\include\memory:2267:4: note: in instantiation of member function
'std::default_delete<NetFirewall::INetFirewallMgr>::operator()' requested here
                        this->get_deleter()(get());
                        ^
NetFirewallMgrFactory.cpp:21:44: note: in instantiation of member function
'std::unique_ptr<NetFirewall::INetFirewallMgr,
std::default_delete<NetFirewall::INetFirewallMgr> >::~unique_ptr' requested here
  unique_ptr<NetFirewall::INetFirewallMgr> fwMgr;
```

🔥 **Frequent offender**

Fixes, fixes, fixes...

```
Error: comparison of two values with different enumeration types in switch
statement 'FormattedLexer::CharType' and 'FormattedLexer::TokenId'
  case REGULAR:
       ^~~~~~~
[-Werror,-Wenum-compare-switch]
```

🔥 **Frequent offender**

🔧 Fixes, fixes, fixes...

`[-Wunused-private-field]`

Remove unused class private fields:

- references
- pointers
- PODs

🔥 **Watch out for orphan method *declarations* in classes**

Iterative Conformance

```
-Wmicrosoft                              -Wno-xyz-warning
-Werror=microsoft                        -Wno-invalid-token-paste
-Werror=typename-missing                 -Wno-language-extension-token
-fms-compatibility-version=19.10         -Wno-unknown-pragmas
-fno-delayed-template-parsing            ...
```

Iterative Conformance

The long road to **MSVC** `/permissive-`

Problems:

fix issues in your code

deal with older Windows SDKs

(eg. targeting WinXP, Win7)

MSVC `/permissive-`

Fix issues in your code

Tips:

- lots of issues related to TPL **two-phase lookup**
- include headers required by your template inline code
- fix issues related to dependent types
- do not assume STL headers include each other => be explicit

MSVC `/permissive-`

Deal with older Windows SDKs

(eg. targeting Win7, WinXP)

Hello, COM !

Tips:

- **forward declare** `struct IUnknown` **before including Win SDK headers**

  (related to TPL two-phase lookup)

MSVC `/permissive-`

Deal with older Windows SDKs

(eg. targeting WinXP, Win7)

Tips:

- **use** `/Zc:strictStrings-` **for SDK headers (your PCH)**

Off by default; the `/permissive-` implicitly sets this option.

When set, the compiler requires strict const-qualification conformance for pointers initialized by using string literals.

https://docs.microsoft.com/en-us/cpp/build/reference/zc-strictstrings-disable-string-literal-type-conversion?view=vs-2019

cpt.config

```
<cpt-config>
  <clang-flags>   "-Werror"
                , "-Wall"
                ~~, "-fms-compatibility-version=19.10"~~
                ~~, "-Wmicrosoft"~~
                , "-Wno-invalid-token-paste"
                , "-Wno-unknown-pragmas"
                , "-Wno-unused-value"
  </clang-flags>
  <header-filter>'.*'</header-filter>
  <parallel/>
  <vs-sku>'Professional'</vs-sku>
  <file-ignore>   'htmlayoutsdk\\include\\behaviors'
                , 'vsphere\\vim25\\core'
  </file-ignore>
  <proj-ignore>   'SciLexer'
                , 'tools\\msix-psf'
  </proj-ignore>
</cpt-config>
```

`clang-tidy`

over 250 checks

https://clang.llvm.org/extra/clang-tidy/checks/list.html

clang-tidy

Large scale refactorings we performed:

- `modernize-use-nullptr`
- `modernize-loop-convert`
- `modernize-use-override`
- `readability-redundant-string-cstr`
- `modernize-use-emplace`
- `modernize-use-auto`
- `modernize-make-shared & modernize-make-unique`
- `modernize-use-equals-default & modernize-use-equals-delete`

clang-tidy

Large scale refactorings we performed:

- `modernize-use-default-member-init`
- `readability-redundant-member-init`
- `modernize-pass-by-value`
- `modernize-return-braced-init-list`
- `modernize-use-using`
- `cppcoreguidelines-pro-type-member-init`
- `readability-redundant-string-init & misc-string-constructor`
- `misc-suspicious-string-compare & misc-string-compare`
- `misc-inefficient-algorithm`
- `cppcoreguidelines-*`

clang-tidy

🔥 **Issues we found:**

```
        [readability-redundant-string-cstr]

        // mChRequest is a 1KB buffer, we don't want to send it whole
        // So copy it as a C string, until we reach a null char
        ret += mChRequest.c_str();
         ^
std::string
```

clang-tidy

🔥 **Issues we found:**

```
[modernize-make-shared, modernize-make-unique]

-   requestData.reset(new BYTE[reqLength]);

+   requestData = std::make_unique<BYTE>();
```

clang-tidy

🔥 **Issues we found:**

```
[modernize-make-shared, modernize-make-unique]

-   requestData.reset(new BYTE[reqLength]);

+   requestData = std::make_unique<BYTE[]>();
```

clang-tidy

🔥 **Issues we found:**

`[modernize-use-auto]` **Works very well, but leaves garbage typedefs:**

```
=> error: unused typedef 'BrowseIterator' [-Werror,-Wunused-local-typedef]

   typedef vector<BrowseSQLServerInfo>::iterator BrowseIterator;
```

clang-tidy

🔥 **Issues we found:**

[modernize-loop-convert]

=> **unused values (orphan)** [-Werror,-Wunused-value]

```
vector<ModuleInfo>::iterator first = Modules_.begin();
vector<ModuleInfo>::iterator last  = Modules_.end();

for (auto & module : Modules_)
{
  ...
}
```

clang-tidy

🔥 **Issues we found:**

```
[modernize-use-using]  => errors & incomplete

- typedef int (WINAPI * InitExtractionFcn)(ExtractInfo *);

+ using InitExtractionFcn =
          int (*)(ExtractInfo *) __attribute__((stdcall))) (ExtractInfo *);


=> using InitExtractionFcn = int (WINAPI *)(ExtractInfo *);
```

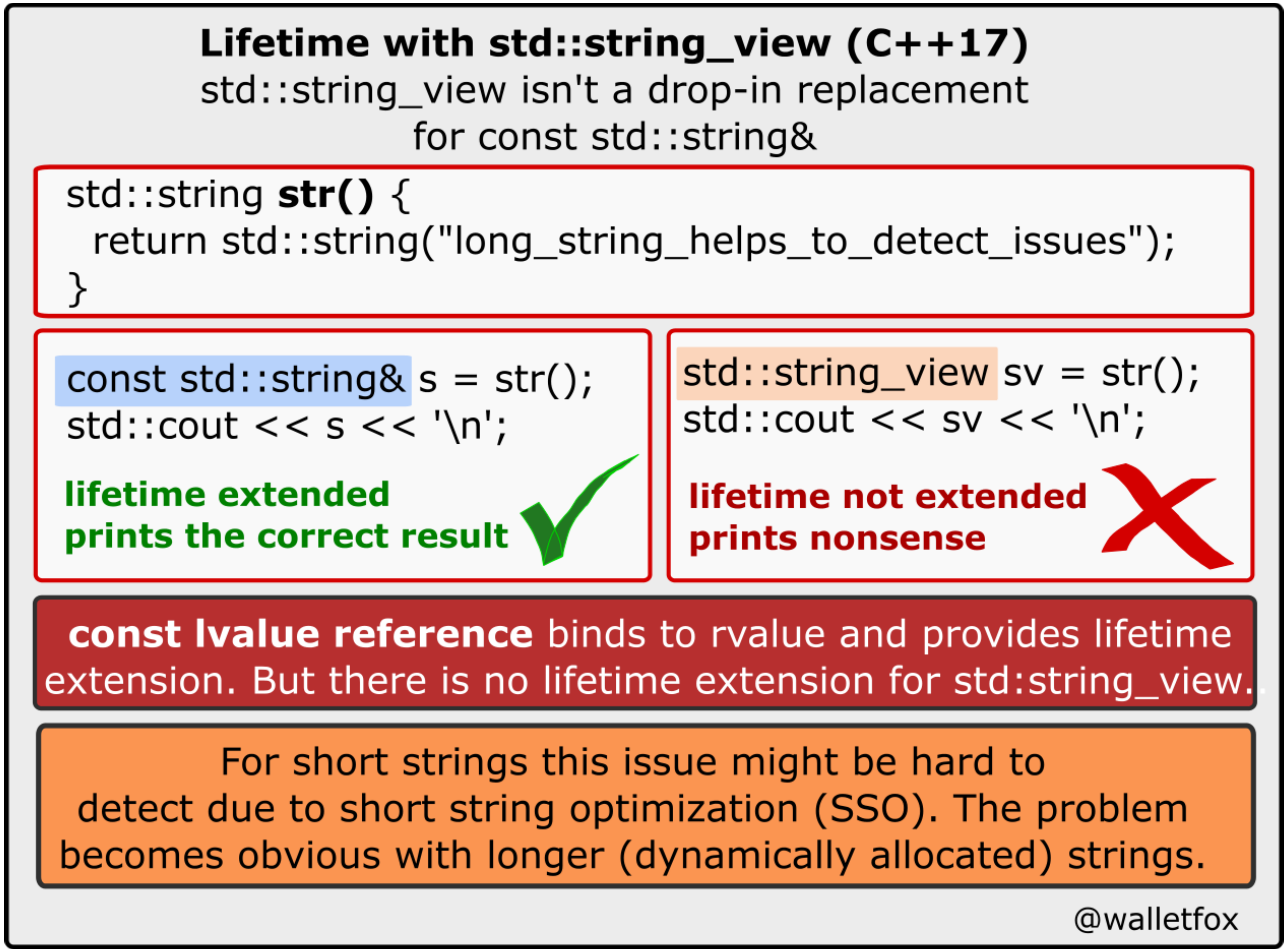
`String` **related checks**

clang-tidy

- `abseil-string-find-startswith`
- `boost-use-to-string`
- `bugprone-string-constructor`
- `bugprone-string-integer-assignment`
- `bugprone-string-literal-with-embedded-nul`
- `bugprone-suspicious-string-compare`
- `modernize-raw-string-literal`
- `performance-faster-string-find`
- `performance-inefficient-string-concatenation`
- `readability-redundant-string-cstr`
- `readability-redundant-string-init`
- `readability-string-compare`

https://clang.llvm.org/extra/clang-tidy/checks/list.html

std::string_view cheatsheet



Lifetime with std::string_view (C++17)
std::string_view isn't a drop-in replacement
for const std::string&

```
std::string str() {
  return std::string("long_string_helps_to_detect_issues");
}
```

```
const std::string& s = str();
std::cout << s << '\n';
```

lifetime extended
prints the correct result

```
std::string_view sv = str();
std::cout << sv << '\n';
```

lifetime not extended
prints nonsense

const lvalue reference binds to rvalue and provides lifetime extension. But there is no lifetime extension for std:string_view.

For short strings this issue might be hard to detect due to short string optimization (SSO). The problem becomes obvious with longer (dynamically allocated) strings.

@walletfox

clang-tidy bugprone-dangling-handle

" Detect dangling references in value handles like `std::string_view`

These dangling references can be a result of constructing handles from ***temporary*** values, where the temporary is destroyed **soon** after the handle is created.

Options:

`HandleClasses`
A semicolon-separated list of class names that should be treated as handles. By default only `std::string_view` is considered.

https://clang.llvm.org/extra/clang-tidy/checks/bugprone-dangling-handle.html

`Lifetime profile v1.0`

Lifetime safety: Preventing common dangling

This is important because it turns out to be ***easy*** to convert **[by design]**
a `std::string` to a `std::string_view`,
or a `std::vector/array` to a `std::span`,
so that dangling is almost the default behavior.

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Lifetime profile v1.0

Lifetime safety: Preventing common dangling

```
void example()
{
  std::string_view sv = std::string("dangling"); // A
  std::cout << sv;          // ERROR (lifetime.3): ‘sv’ was invalidated when
}                           // temporary was destroyed (line A)
```

`clang -Wlifetime` Experimental

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

NEW

Lifetime safety: Preventing common dangling

[-Wdangling-gsl] **diagnosed by default in Clang 10**

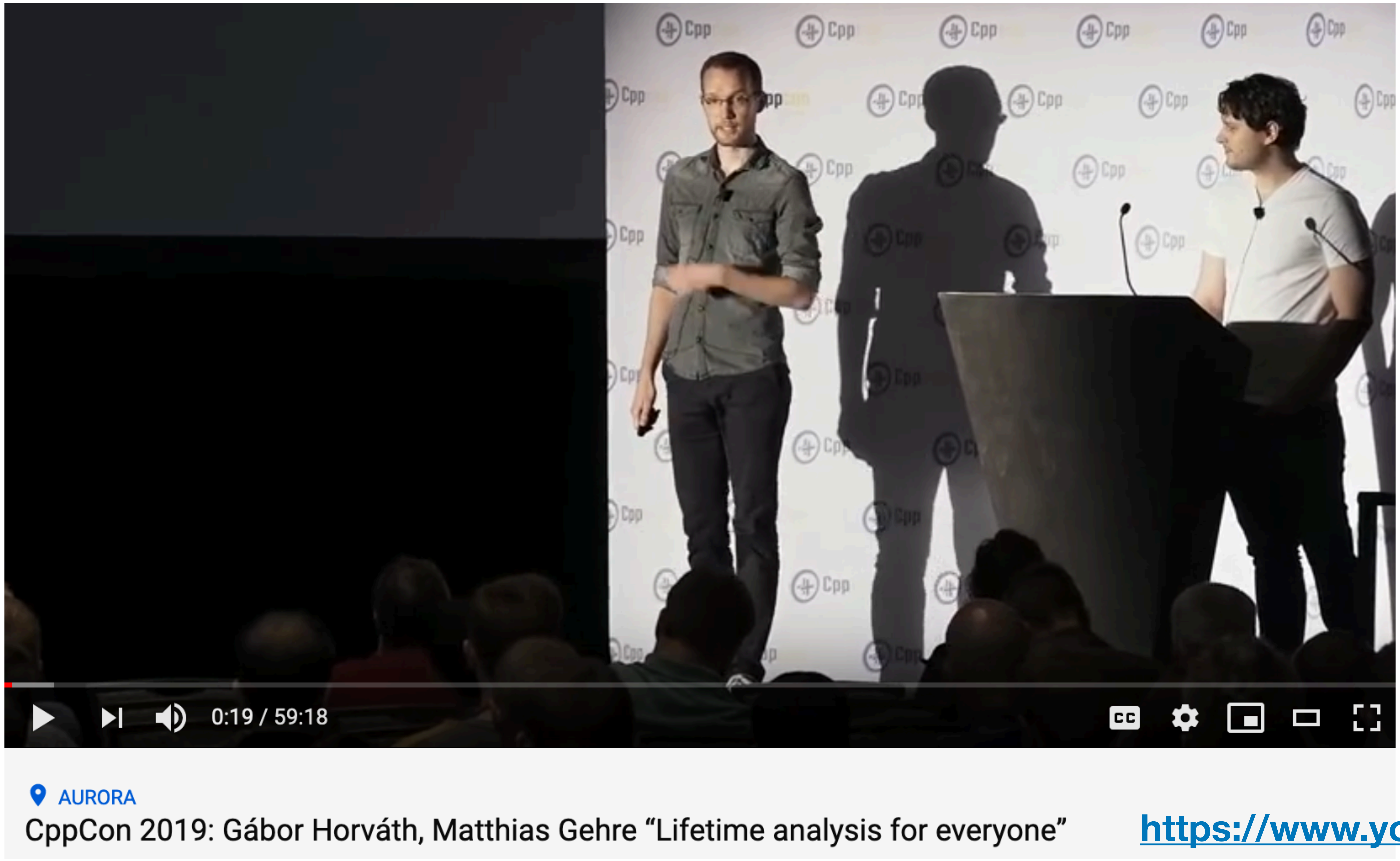
warning: initializing pointer member to point to a temporary object whose lifetime is shorter than the lifetime of the constructed object

```
void example()
{
  std::string_view sv = std::string("dangling");
                     // warning: object backing the pointer will be destroyed
                     // at the end of the full-expression [-Wdangling-gsl]
  std::cout << sv;
}
```

https://clang.llvm.org/docs/DiagnosticsReference.html#wdangling-gsl

Lifetime profile

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf



CppCon 2019: Gábor Horváth, Matthias Gehre "Lifetime analysis for everyone"

https://www.youtube.com/watch?v=d67kfSnhbpA

Part III

Take Control

More `clang-tidy` checks

https://github.com/llvm/llvm-project

clang-tidy

Checks are organized in **modules**, which can be linked into clang-tidy
with minimal or no code changes in clang-tidy

Checks can plug into the analysis on the **preprocessor** level using **PPCallbacks**
or on the AST level using **AST Matchers**

Checks can **report** issues in a similar way to how Clang diagnostics work.
A **fix-it** hint can be attached to a diagnostic message

Tools

- `add_new_check.py` - automate the process of adding a new check

  (creates check, update the CMake file and creates test)

- `rename_check.py` - renames an existing check
- `clang-query` - interactive prototyping of AST matchers and exploration of the Clang AST
- `clang-check -ast-dump` - provides a convenient way to dump the AST

```
clang-tidy/                         # Clang-tidy core.
|-- ClangTidy.h                     # Interfaces for users and checks.
|-- ClangTidyModule.h               # Interface for clang-tidy modules.
|-- ClangTidyModuleRegistry.h       # Interface for registering of modules.
   ...
|-- mymod/                          # My Own clang-tidy module.
|-+
  |-- MyModTidyModule.cpp
  |-- MyModTidyModule.h
        ...

|-- tool/                           # Sources of the clang-tidy binary.
        ...
test/clang-tidy/                    # Integration tests.
    ...
unittests/clang-tidy/               # Unit tests.
|-- ClangTidyTest.h
|-- MyModModuleTest.cpp
```

Setup

```
# download the sources
git clone git@github.com:llvm/llvm-project
cd clang-tools-extra

# build everything
mkdir build && cd build/
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
make check-clang-tools
```

Hello World

We will add our check to the [**readability**] category/module

```
add_new_check.py readability pretty-func
```

This will create:

```
/readability/PrettyFuncCheck.h
/readability/PrettyFuncCheck.cpp
```

=> include it in:

```
/readability/ReadabilityTidyModule.cpp
```

```cpp
#include "../ClangTidy.h"

namespace clang {
namespace tidy {
namespace readability {

class PrettyFuncCheck : public ClangTidyCheck
{
public:
  PrettyFuncCheck(StringRef Name, ClangTidyContext * Context)
    : ClangTidyCheck(Name, Context) {}

  void registerMatchers(ast_matchers::MatchFinder * Finder) override;
  void check(const ast_matchers::MatchFinder::MatchResult & Result) override;
};

} // namespace readability
} // namespace tidy
} // namespace clang
```

`ClangTidyCheck`

Our check needs to operate on the AST level:

- `registerMatchers()` - register clang AST matchers to filter out interesting source locations
- `check()` - provide a function which is called by the Clang whenever a match was found;
  we can perform further actions here (eg. emit diagnostics)

If we wanted to analyze code on the **preprocessor** level

=> override `registerPPCallbacks()` method

ClangTidyCheck

```
using namespace ast_matchers;


void PrettyFuncCheck::registerMatchers(MatchFinder * Finder)
{
  Finder->addMatcher(functionDecl().bind("needle"), this);
}
```

```
using namespace ast_matchers;


void PrettyFuncCheck::check(const MatchFinder::MatchResult & Result)
{
  const auto * MatchedDecl = Result.Nodes.getNodeAs<FunctionDecl>("needle");

  if (MatchedDecl->getName().startswith_lower("get_"))
  {
    diag(MatchedDecl->getLocation(), "function %0 needs your attention")
           << MatchedDecl
           << FixItHint::CreateInsertion(MatchedDecl->getLocation(), "Get");
  }
}
```

Test it...

```
clang-tidy -checks='-*,readability-pretty-func' some/file.cpp
```

Check Options

If a check needs configuration **options**, it can access check-specific options using:

```
Options.get<Type>("SomeOption", DefaultValue)
```

Check Options

```
class PrettyFuncCheck : public ClangTidyCheck
{
  const unsigned    Tolerance;  // option 1
  const std::string TargetFunc; // option 2
public:

  PrettyFuncCheck(StringRef Name, ClangTidyContext * Context)
    : ClangTidyCheck(Name, Context),
      Tolerance (Options.get("Tolerance", 0)),
      TargetFunc(Options.get("TargetFunc", "get_")) {}

  void storeOptions(ClangTidyOptions::OptionMap & Opts) override
  {
    Options.store(Opts, "Tolerance",  Tolerance);
    Options.store(Opts, "TargetFunc", TargetFunc);
  }
```

.clang-tidy

```
CheckOptions:
  - key: readability-pretty-func.Tolerance           a1
    value: 123                                       b1
  - key: readability-pretty-func.TargetFunc          a2
    value: 'get_'                                    b2
```

```
clang-tidy
  -config="{CheckOptions: [{key: a1, value: b1}, {key: a2, value: b2}]}" ...
```

Testing Our Check

Write some test units...

```
% ninja check-clang-tools
```

or

```
% make check-clang-tools
```

```
check_clang_tidy.py
```

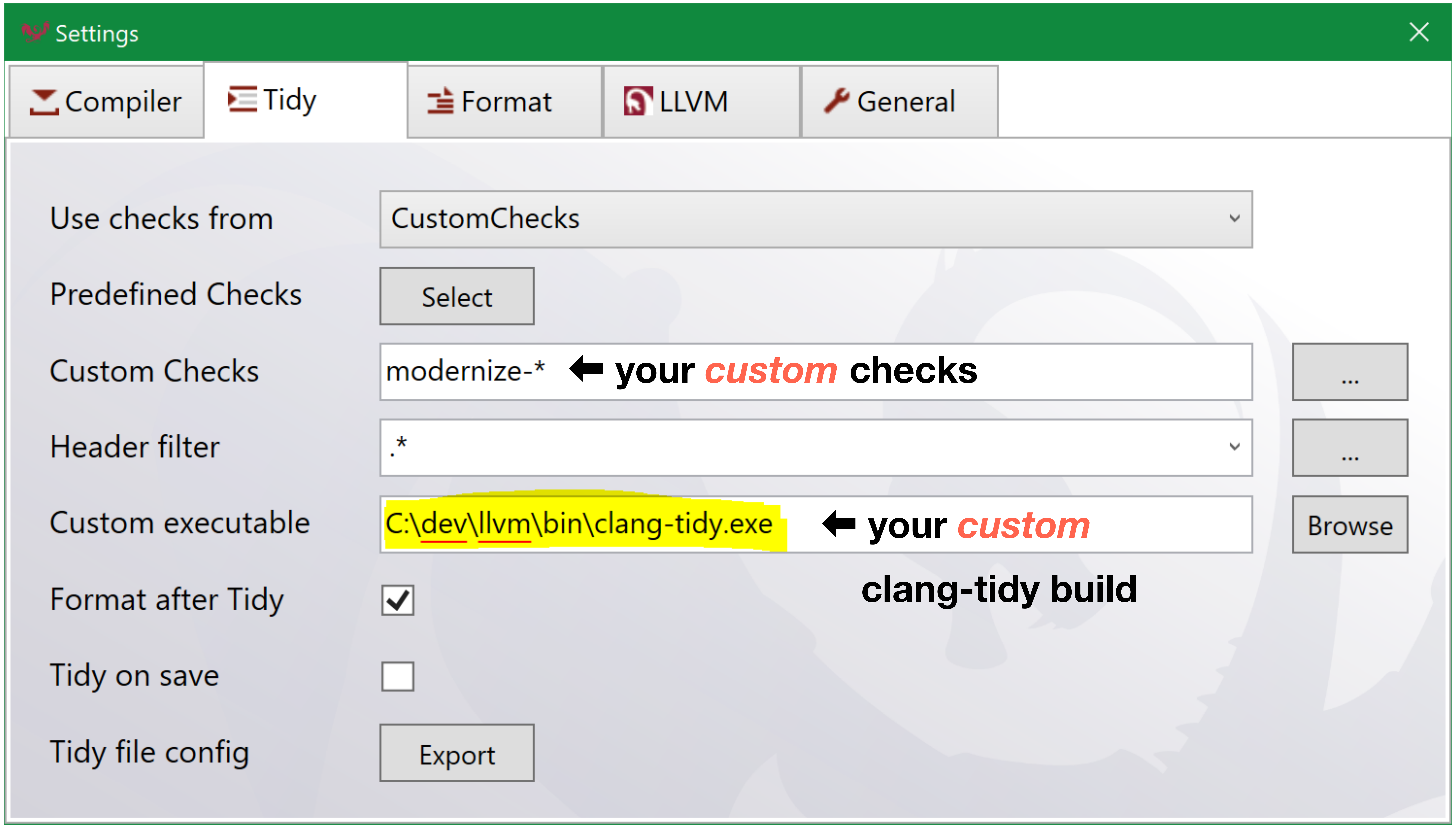
Debug AST Matcher

```
% clang-check -ast-dump my_source.cpp --
```

```
TranslationUnitDecl 0x2b3cd20 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x2b3d258 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
|-TypedefDecl 0x2b3d2b8 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
|-TypedefDecl 0x2b3d698 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag [1]'
|-CXXRecordDecl 0x2b3d6e8 </test.cpp:1:1, line:3:1> line:1:8 referenced struct A definition
| |-CXXRecordDecl 0x2b3d800 <col:1, col:8> col:8 implicit struct A
| `-CXXMethodDecl 0x2b3d8e0 <line:2:9, col:19> col:14 f 'void (void)'
|   `-CompoundStmt 0x2b3d9b8 <col:18, col:19>
`-CXXRecordDecl 0x2b3d9d0 <line:5:1, line:7:1> line:5:8 struct B definition
  |-public 'struct A'
  |-CXXRecordDecl 0x2b85050 <col:1, col:8> col:8 implicit struct B
  |-CXXMethodDecl 0x2b85100 <line:6:3, col:21> col:16 f 'void (void)' virtual
  | `-CompoundStmt 0x2b854f8 <col:20, col:21>
```

https://clang.llvm.org/docs/LibASTMatchersReference.html

Custom clang-tidy checks



Settings

Compiler | Tidy | Format | LLVM | General

Use checks from: CustomChecks

Predefined Checks: Select

Custom Checks: modernize-* ← your *custom* checks

Header filter: .*

Custom executable: C:\dev\llvm\bin\clang-tidy.exe ← your *custom* clang-tidy build

Format after Tidy: ☑

Tidy on save: ☐

Tidy file config: Export

Write *custom* checks for your needs (project specific)

Run them regularly !

Explore Further

code::dive 2018

Refactor with Clang Tooling

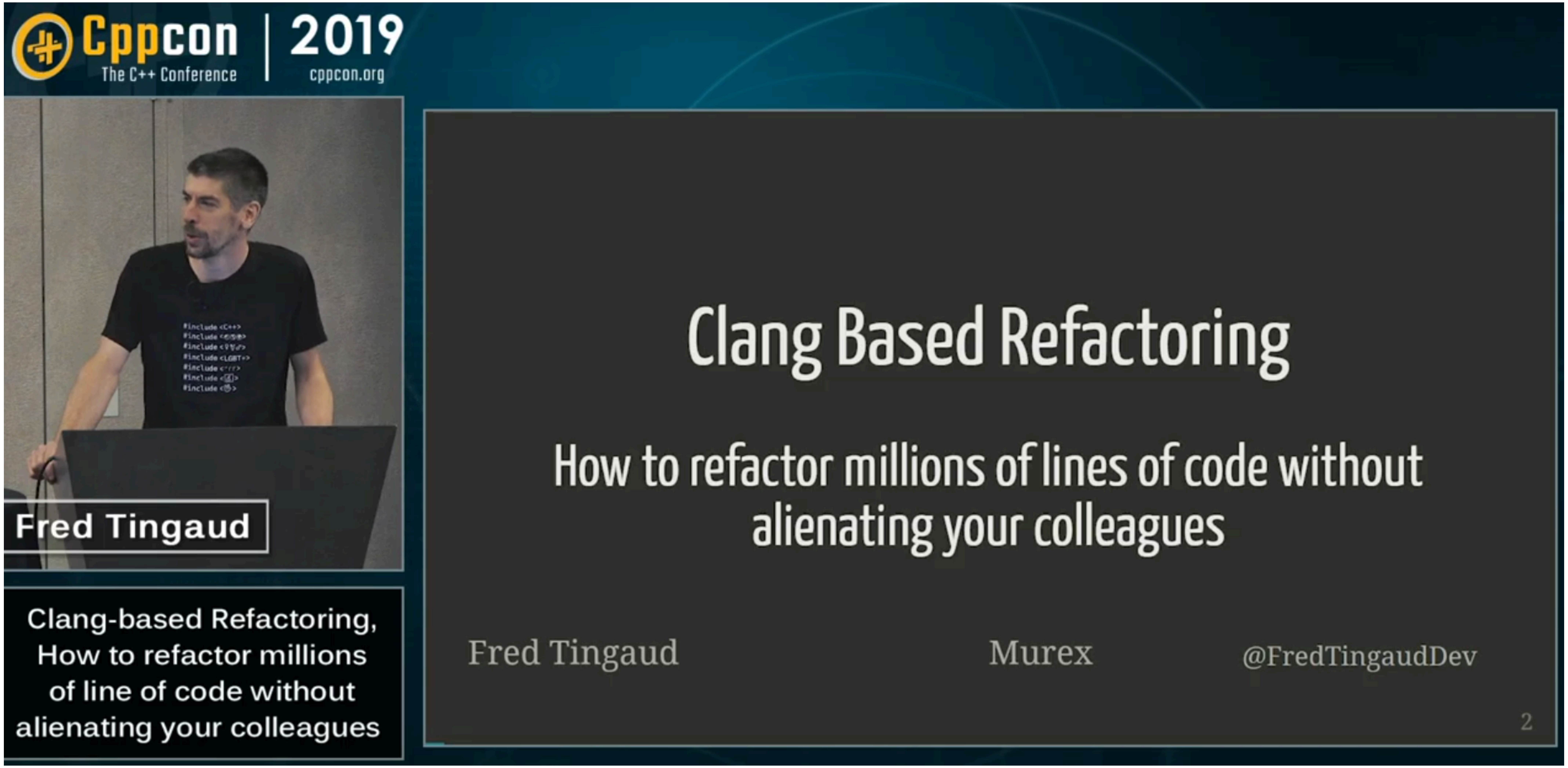
Tools, Tips, Tricks and Traps

Stephen Kelly
steveire.wordpress.com
@steveire

Stephen Kelly

https://steveire.wordpress.com/2019/01/02/refactor-with-clang-tooling-at-codedive-2018/

Explore Further



Clang-based Refactoring, How to refactor millions of line of code without alienating your colleagues

Clang Based Refactoring

How to refactor millions of lines of code without alienating your colleagues

Fred Tingaud Murex @FredTingaudDev

2

https://www.youtube.com/watch?v=JPnN2c2odNY

Explore Further

A new series of blog articles on **Visual C++ Team blog** by **Stephen Kelly**

Exploring Clang Tooling, Part 0: Building Your Code with Clang

https://blogs.msdn.microsoft.com/vcblog/2018/09/18/exploring-clang-tooling-part-0-building-your-code-with-clang/

Exploring Clang Tooling, Part 1: Extending Clang-Tidy

https://blogs.msdn.microsoft.com/vcblog/2018/10/19/exploring-clang-tooling-part-1-extending-clang-tidy/

Exploring Clang Tooling, Part 2: Examining the Clang AST with clang-query

https://blogs.msdn.microsoft.com/vcblog/2018/10/23/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/

Explore Further

A new series of blog articles on **Visual C++ Team blog** by **Stephen Kelly**

Exploring Clang Tooling, Part 3: Rewriting Code with clang-tidy

https://blogs.msdn.microsoft.com/vcblog/2018/11/06/exploring-clang-tooling-part-3-rewriting-code-with-clang-tidy

Exploring Clang Tooling: Using Build Tools with clang-tidy

https://blogs.msdn.microsoft.com/vcblog/2018/11/27/exploring-clang-tooling-using-build-tools-with-clang-tidy/

Explore Further

More blog articles by **Stephen Kelly**

Future Developments in clang-query

https://steveire.wordpress.com/2018/11/11/future-developments-in-clang-query/

Composing AST Matchers in clang-tidy

https://steveire.wordpress.com/2018/11/20/composing-ast-matchers-in-clang-tidy/

Part IV

Status Quo

Visual Studio 2019
v16.2

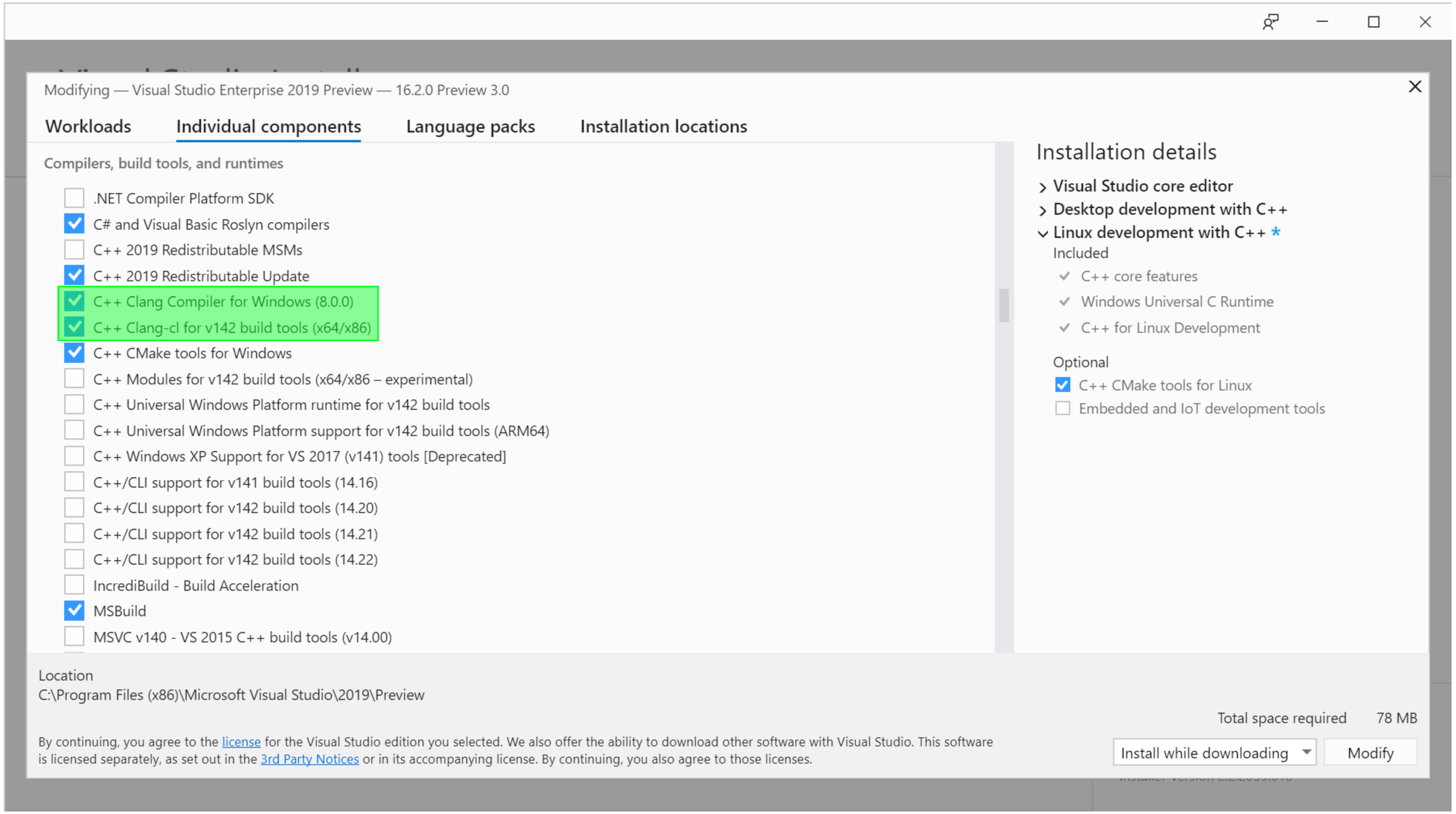
`Clang/LLVM` **support for** `MSBuild` **Projects**

Ships with `Clang 8` **(as optional component)**

`clang-cl.exe`

📖 https://devblogs.microsoft.com/cppblog/clang-llvm-support-for-msbuild-projects/

Visual Studio 2019
v16.2



Modifying — Visual Studio Enterprise 2019 Preview — 16.2.0 Preview 3.0

Workloads | Individual components | Language packs | Installation locations

Compilers, build tools, and runtimes

- [ ] .NET Compiler Platform SDK
- [✓] C# and Visual Basic Roslyn compilers
- [ ] C++ 2019 Redistributable MSMs
- [✓] C++ 2019 Redistributable Update
- [✓] C++ Clang Compiler for Windows (8.0.0)
- [✓] C++ Clang-cl for v142 build tools (x64/x86)
- [✓] C++ CMake tools for Windows
- [ ] C++ Modules for v142 build tools (x64/x86 – experimental)
- [ ] C++ Universal Windows Platform runtime for v142 build tools
- [ ] C++ Universal Windows Platform support for v142 build tools (ARM64)
- [ ] C++ Windows XP Support for VS 2017 (v141) tools [Deprecated]
- [ ] C++/CLI support for v141 build tools (14.16)
- [ ] C++/CLI support for v142 build tools (14.20)
- [ ] C++/CLI support for v142 build tools (14.21)
- [ ] C++/CLI support for v142 build tools (14.22)
- [ ] IncrediBuild - Build Acceleration
- [✓] MSBuild
- [ ] MSVC v140 - VS 2015 C++ build tools (v14.00)

Installation details

> Visual Studio core editor
> Desktop development with C++
∨ Linux development with C++ *

Included

✓ C++ core features
✓ Windows Universal C Runtime
✓ C++ for Linux Development

Optional

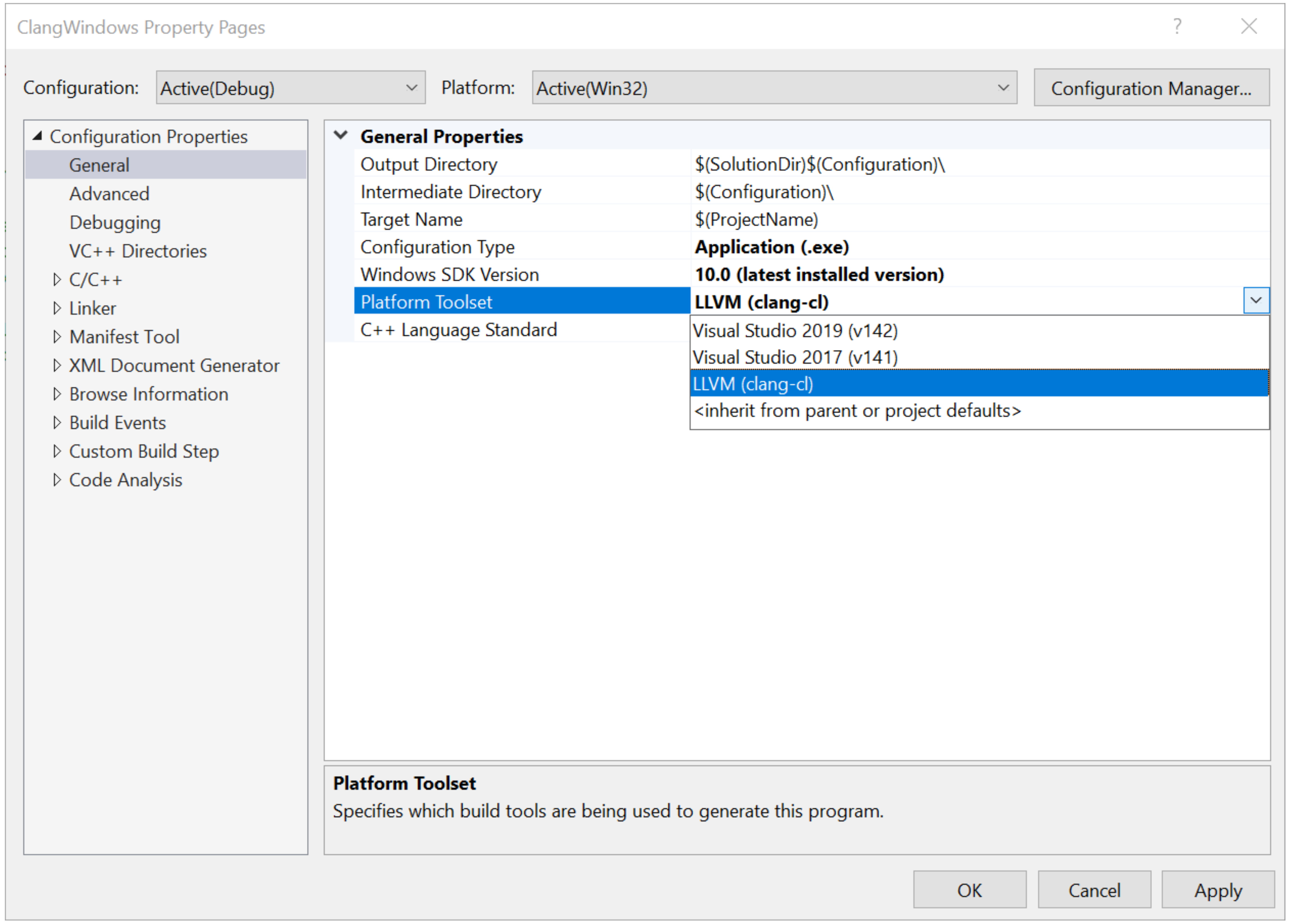
- [✓] C++ CMake tools for Linux
- [ ] Embedded and IoT development tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Preview

Total space required 78 MB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Visual Studio 2019
v16.2


ClangWindows Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)   Configuration Manager...

- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis

General Properties

Output Directory	$(SolutionDir)$(Configuration)\
Intermediate Directory	$(Configuration)\
Target Name	$(ProjectName)
Configuration Type	**Application (.exe)**
Windows SDK Version	**10.0 (latest installed version)**
Platform Toolset	**LLVM (clang-cl)**
C++ Language Standard	

Visual Studio 2019 (v142)
Visual Studio 2017 (v141)
LLVM (clang-cl)
<inherit from parent or project defaults>

Platform Toolset
Specifies which build tools are being used to generate this program.

`clang-cl.exe`

Visual Studio 2019
v16.4

The biggest VS release since VS 2019

🎉

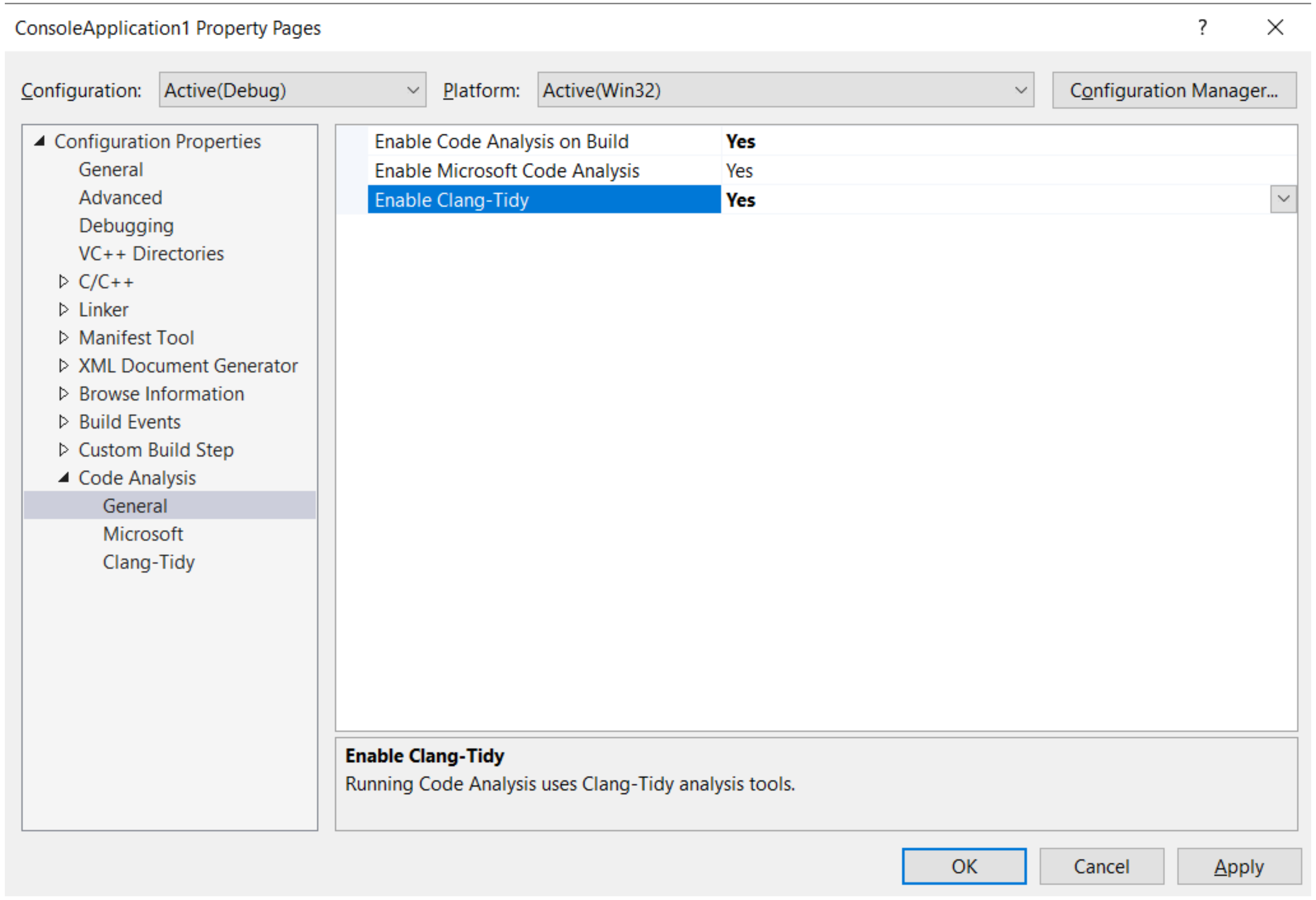
Preview

Visual Studio 2019
v16.4

`clang-tidy`
code analysis

📖 https://devblogs.microsoft.com/cppblog/code-analysis-with-clang-tidy-in-visual-studio/

Visual Studio 2019
v16.4



ConsoleApplication1 Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

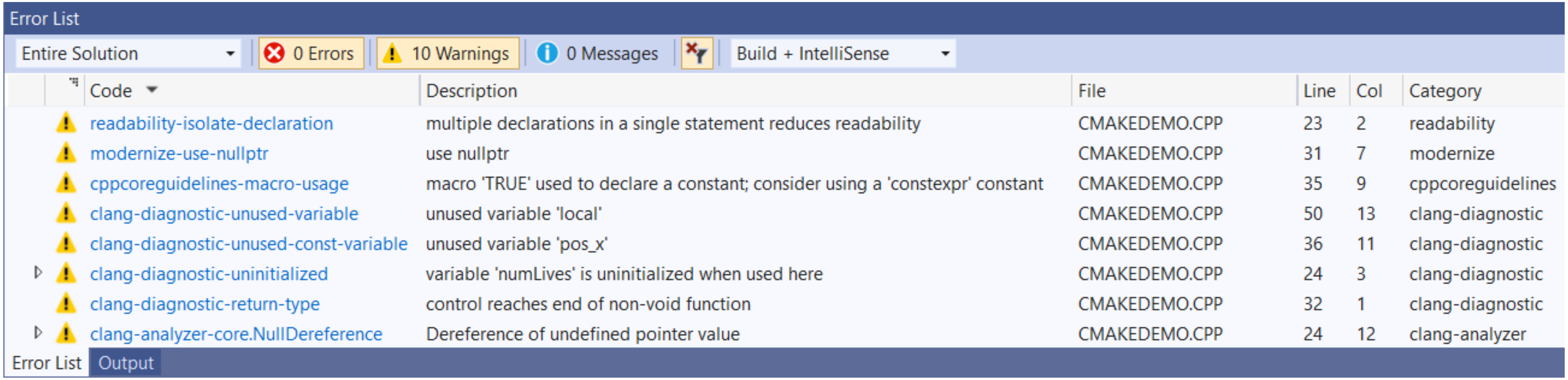
- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis
    - General
    - Microsoft
    - Clang-Tidy

Enable Code Analysis on Build	**Yes**
Enable Microsoft Code Analysis	Yes
Enable Clang-Tidy	**Yes**

Enable Clang-Tidy
Running Code Analysis uses Clang-Tidy analysis tools.

Visual Studio 2019
v16.4

`clang-tidy` **warnings**



Error List

Entire Solution | 0 Errors | 10 Warnings | 0 Messages | Build + IntelliSense

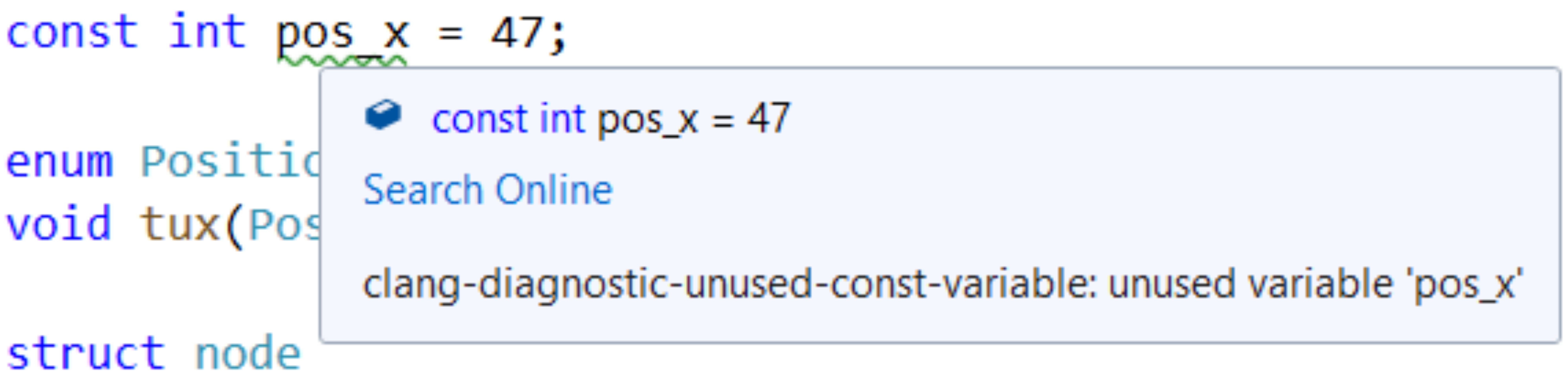
Code	Description	File	Line	Col	Category
readability-isolate-declaration	multiple declarations in a single statement reduces readability	CMAKEDEMO.CPP	23	2	readability
modernize-use-nullptr	use nullptr	CMAKEDEMO.CPP	31	7	modernize
cppcoreguidelines-macro-usage	macro 'TRUE' used to declare a constant; consider using a 'constexpr' constant	CMAKEDEMO.CPP	35	9	cppcoreguidelines
clang-diagnostic-unused-variable	unused variable 'local'	CMAKEDEMO.CPP	50	13	clang-diagnostic
clang-diagnostic-unused-const-variable	unused variable 'pos_x'	CMAKEDEMO.CPP	36	11	clang-diagnostic
clang-diagnostic-uninitialized	variable 'numLives' is uninitialized when used here	CMAKEDEMO.CPP	24	3	clang-diagnostic
clang-diagnostic-return-type	control reaches end of non-void function	CMAKEDEMO.CPP	32	1	clang-diagnostic
clang-analyzer-core.NullDereference	Dereference of undefined pointer value	CMAKEDEMO.CPP	24	12	clang-analyzer

Error List Output

📖 https://devblogs.microsoft.com/cppblog/code-analysis-with-clang-tidy-in-visual-studio/

Visual Studio 2019
v16.4

`clang-tidy` **warnings also display as in-editor squiggles**



```
const int pos_x = 47;

enum Positic
void tux(Pos

struct node
```

```
const int pos_x = 47
Search Online

clang-diagnostic-unused-const-variable: unused variable 'pos_x'
```

Code Analysis runs automatically in the background

Preview

NOT on
Visual Studio 2019 v16.4
yet ?

No problem

=

->

Clang Power Tools

www.clangpowertools.com

LLVM

clang-tidy
clang++
clang-format

Visual Studio

2015 / 2017 / 2019

Sanitizers

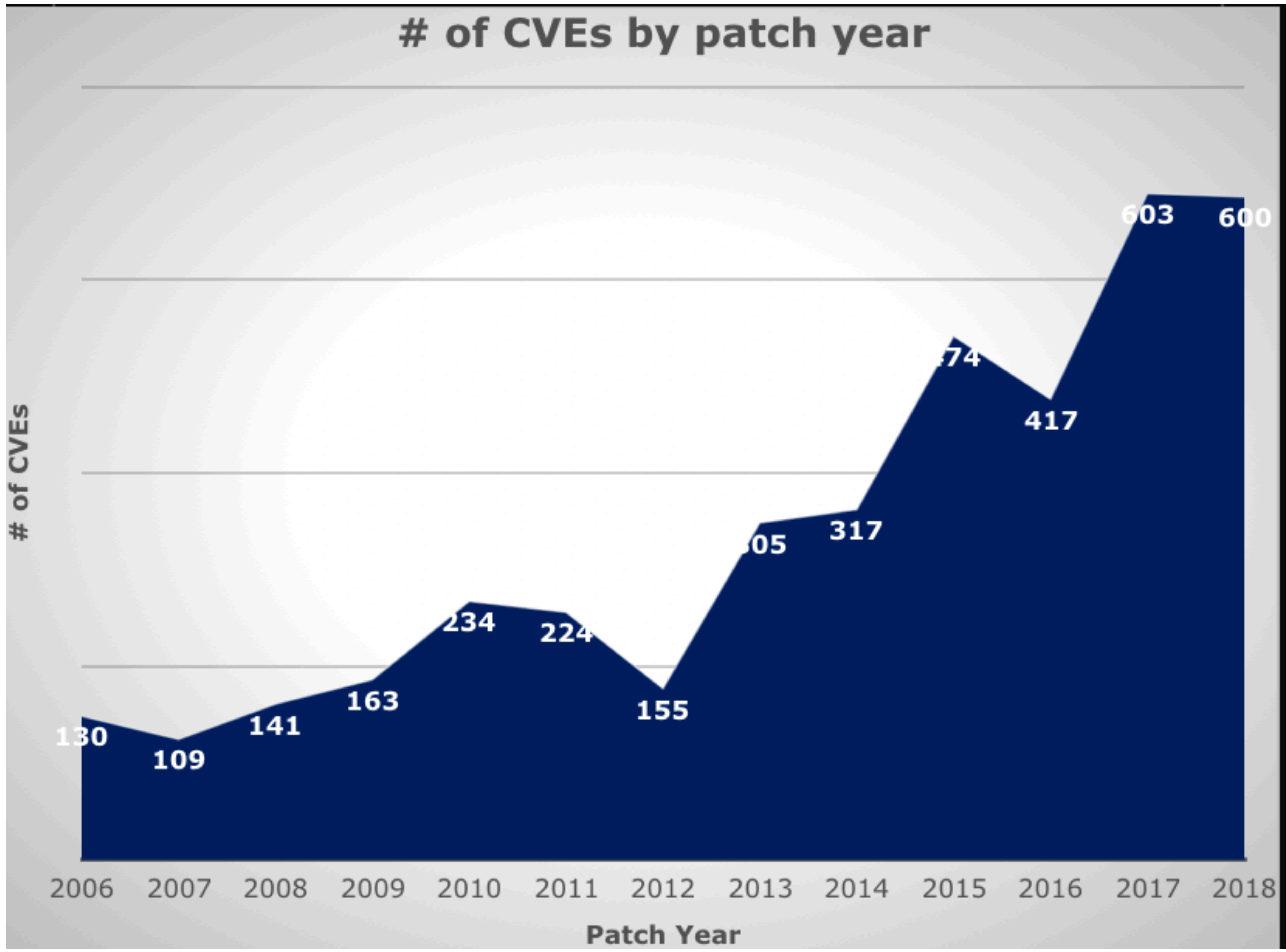
Sanitizers

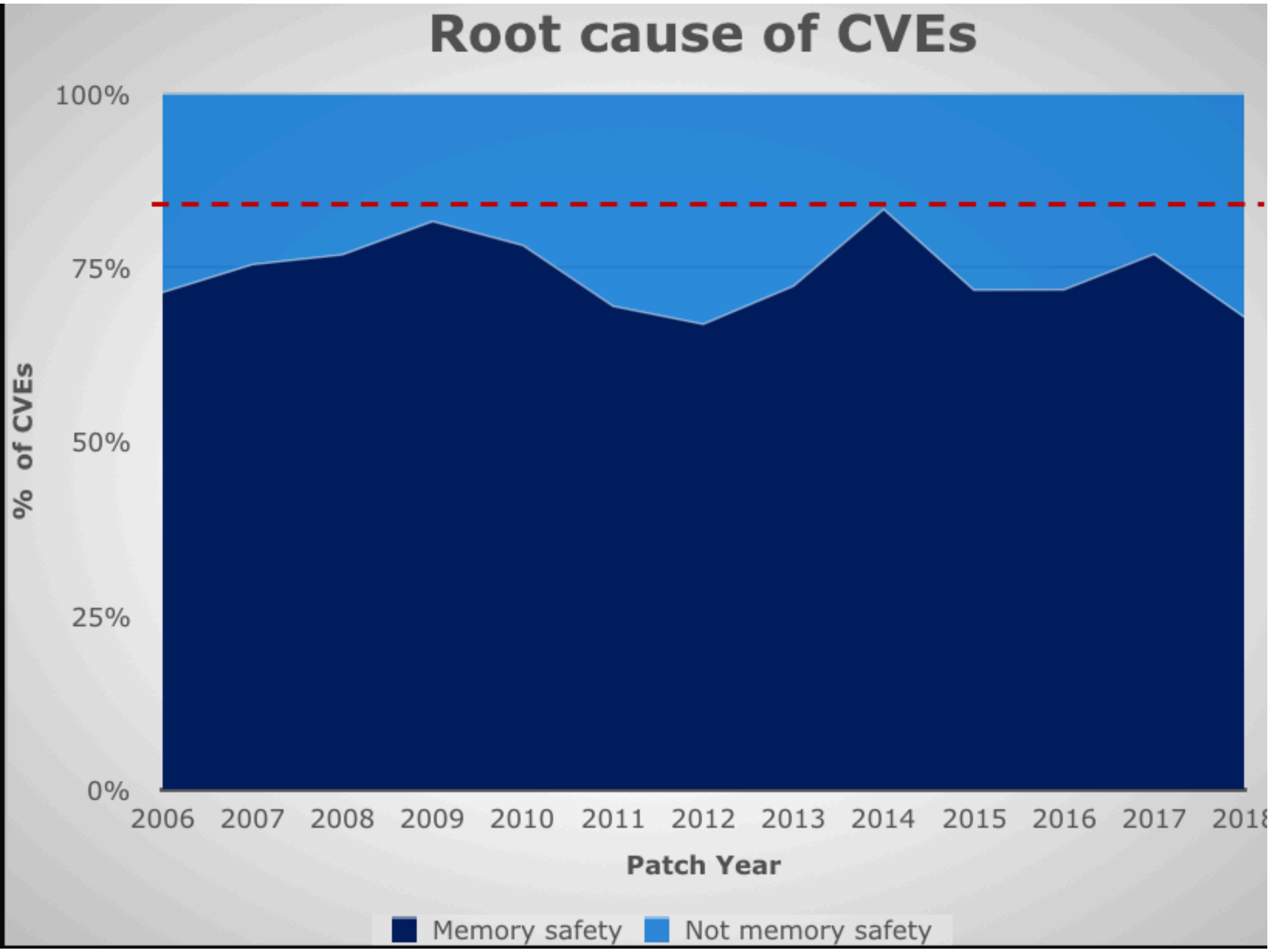
- `AddressSanitizer` - detects addressability issues
- `LeakSanitizer` - detects memory leaks
- `ThreadSanitizer` - detects data races and deadlocks
- `MemorySanitizer` - detects use of uninitialized memory
- `HWASAN` - hardware-assisted AddressSanitizer (consumes less memory)
- `UBSan` - detects Undefined Behavior

https://github.com/google/sanitizers

Common Vulnerabilities and Exposures

Memory safety continues to dominate





https://www.youtube.com/watch?v=0EsqxGgYOQU

Address Sanitizer (ASan)

de facto standard for detecting memory safety issues

Detects:

- **Use after free** (dangling pointer dereference)
- **Heap buffer overflow**
- **Stack buffer overflow**
- **Global buffer overflow**
- **Use after return**
- **Use after scope**
- **Initialization order bugs**
- **Memory leaks**

Very fast instrumentation
(average slowdown is ~2x)

https://github.com/google/sanitizers/wiki/AddressSanitizer

Address Sanitizer `(ASan)`

Compiler

- instrumentation code, stack layout, and calls into runtime
- meta-data in OBJ for the runtime

Sanitizer Runtime

- hooking `malloc(), free(), memset()`, etc.
- error analysis and reporting
- does not require complete recompile
- zero false positives

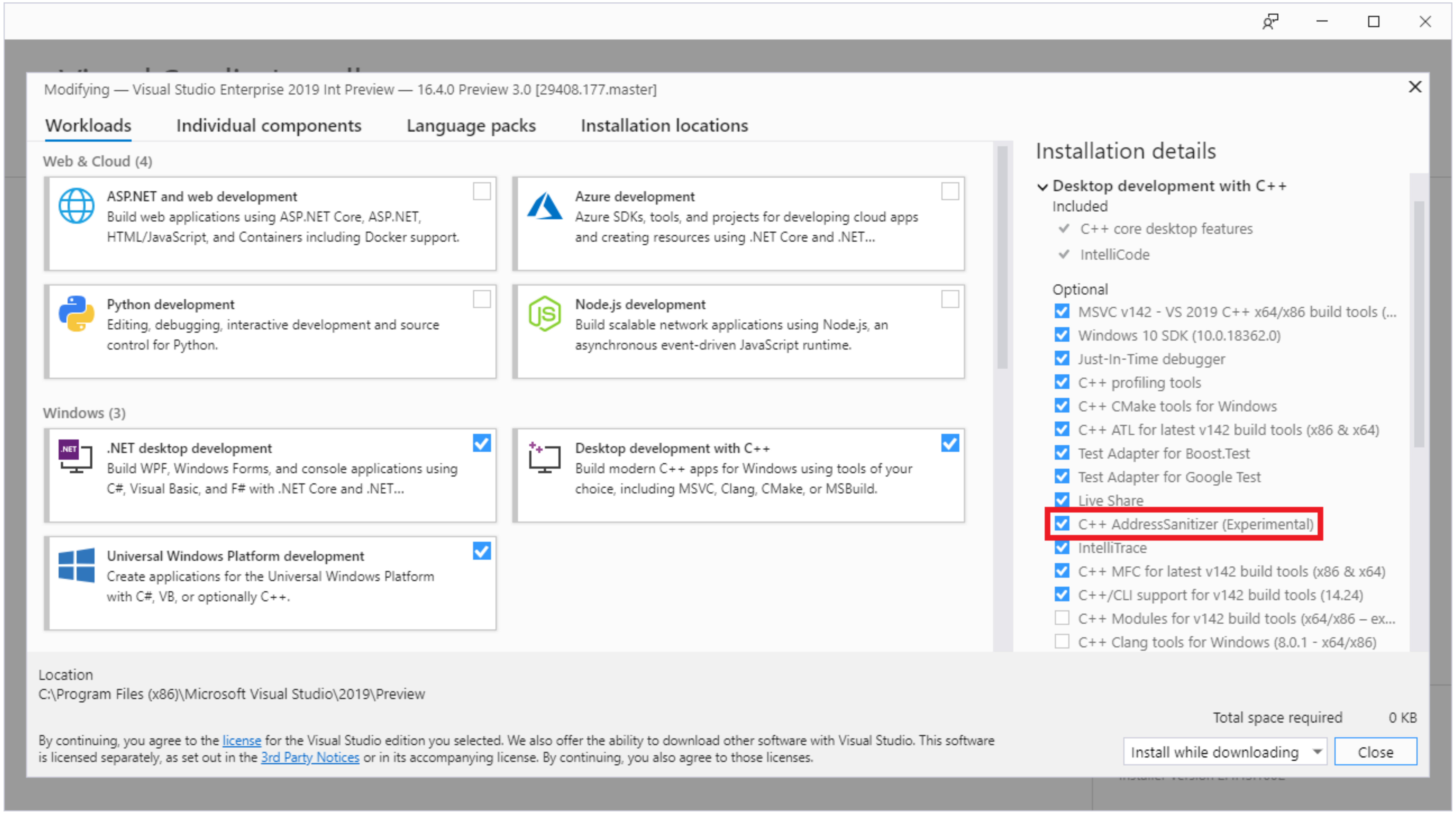
Visual Studio 2019
v16.4

Address Sanitizer
(ASan)

🎉

📖 https://devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/

Visual Studio 2019
v16.4



Modifying — Visual Studio Enterprise 2019 Int Preview — 16.4.0 Preview 3.0 [29408.177.master]

Web & Cloud (4)

- ☐ **ASP.NET and web development** — Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker support.
- ☐ **Azure development** — Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET Core and .NET...
- ☐ **Python development** — Editing, debugging, interactive development and source control for Python.
- ☐ **Node.js development** — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Windows (3)

- ✓ **.NET desktop development** — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET Core and .NET...
- ✓ **Desktop development with C++** — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
- ✓ **Universal Windows Platform development** — Create applications for the Universal Windows Platform with C#, VB, or optionally C++.

Installation details

Desktop development with C++

Included
- ✓ C++ core desktop features
- ✓ IntelliCode

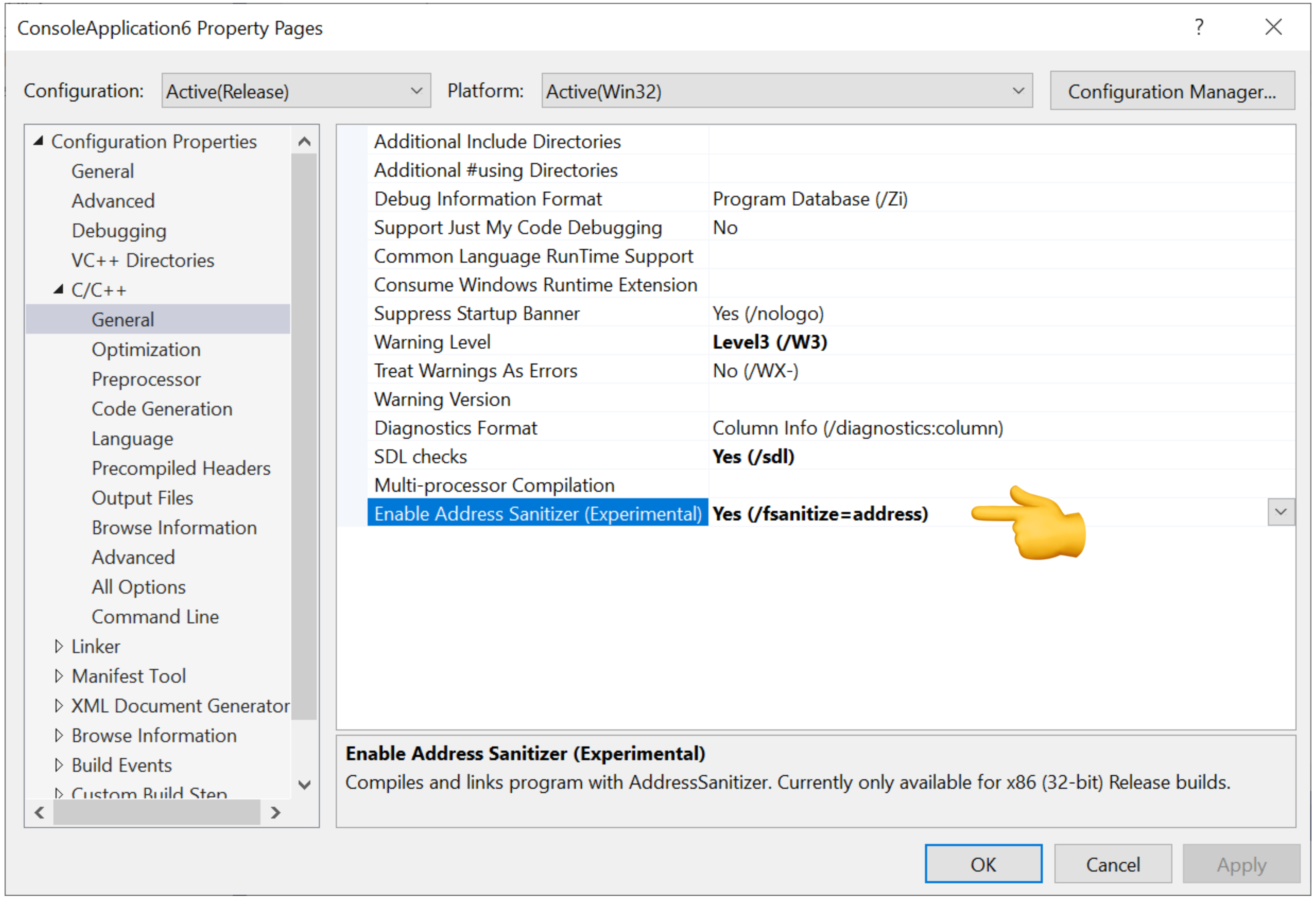
Optional
- ✓ MSVC v142 - VS 2019 C++ x64/x86 build tools (...
- ✓ Windows 10 SDK (10.0.18362.0)
- ✓ Just-In-Time debugger
- ✓ C++ profiling tools
- ✓ C++ CMake tools for Windows
- ✓ C++ ATL for latest v142 build tools (x86 & x64)
- ✓ Test Adapter for Boost.Test
- ✓ Test Adapter for Google Test
- ✓ Live Share
- ✓ C++ AddressSanitizer (Experimental)
- ✓ IntelliTrace
- ✓ C++ MFC for latest v142 build tools (x86 & x64)
- ✓ C++/CLI support for v142 build tools (14.24)
- ☐ C++ Modules for v142 build tools (x64/x86 – ex...
- ☐ C++ Clang tools for Windows (8.0.1 - x64/x86)

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Preview

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Total space required 0 KB

Visual Studio 2019
v16.4



ConsoleApplication6 Property Pages

Configuration: Active(Release) Platform: Active(Win32) Configuration Manager...

- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step

Additional Include Directories	
Additional #using Directories	
Debug Information Format	Program Database (/Zi)
Support Just My Code Debugging	No
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Column Info (/diagnostics:column)
SDL checks	**Yes (/sdl)**
Multi-processor Compilation	
Enable Address Sanitizer (Experimental)	**Yes (/fsanitize=address)**

Enable Address Sanitizer (Experimental)
Compiles and links program with AddressSanitizer. Currently only available for x86 (32-bit) Release builds.

Visual Studio 2019
v16.4

Just x86 at the moment :(

x64 support coming soon...

📖 https://devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/

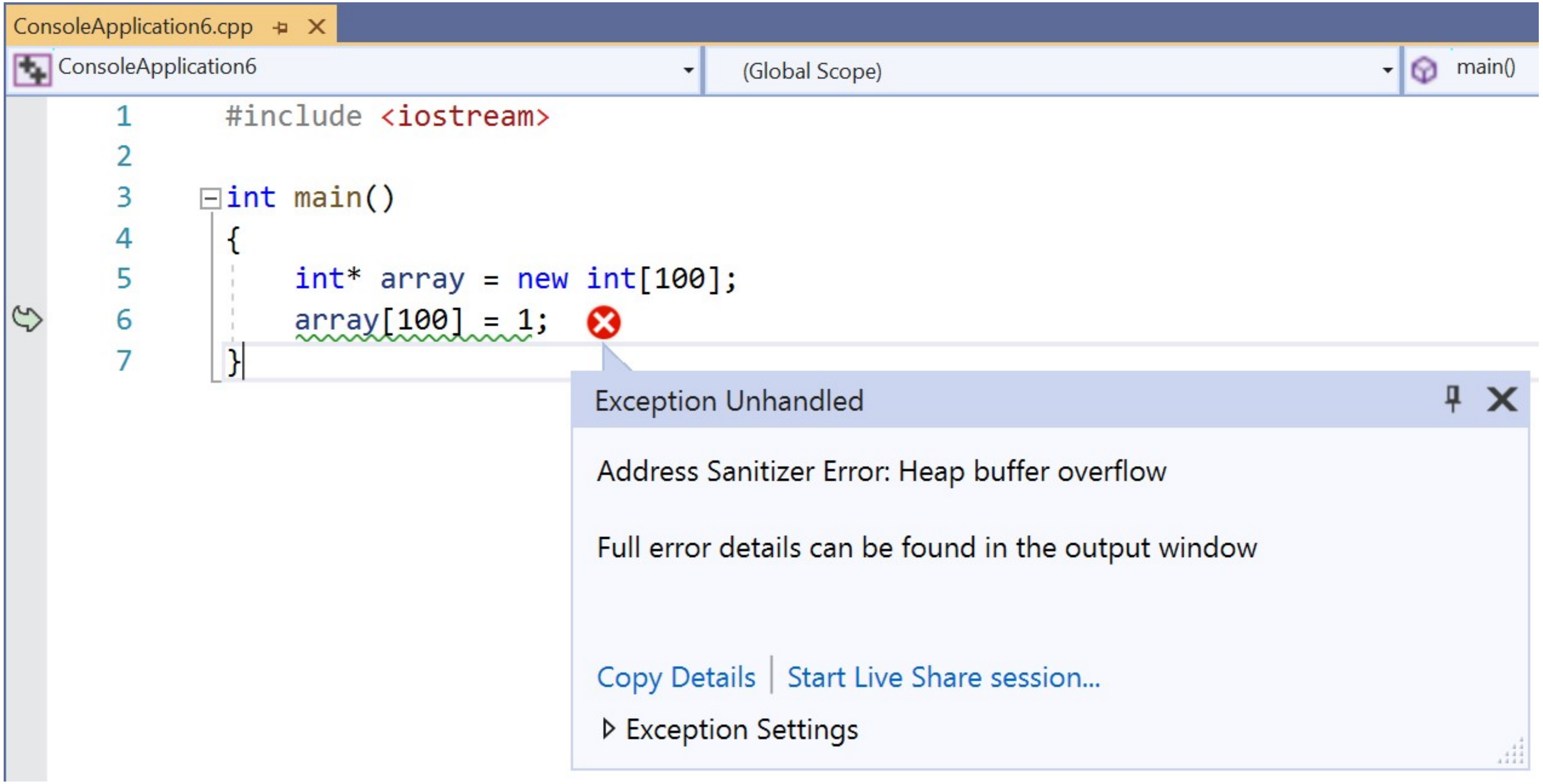
Visual Studio 2019
v16.4

- **Compiling a single static EXE**
  link the static runtime `asan-i386.lib` and the cxx library

- **Compiling an EXE with /MT runtime which will use ASan-instrumented DLLs**
  the EXE needs to have `asan-i386.lib` linked and
  the DLLs need the `clang_rt.asan_dll_thunk-i386.lib`

- **When compiling with the /MD dynamic runtime**
  all EXE and DLLs with instrumentation should be linked with
  `asan_dynamic-i386.lib` and `clang_rt.asan_dynamic_runtime_thunk-i386.lib`
  At runtime, these libraries will refer to the
  `clang_rt.asan_dynamic-i386.dll` shared ASan runtime.

Address Sanitizer (ASan)



```cpp
#include <iostream>

int main()
{
    int* array = new int[100];
    array[100] = 1;
}
```

Exception Unhandled

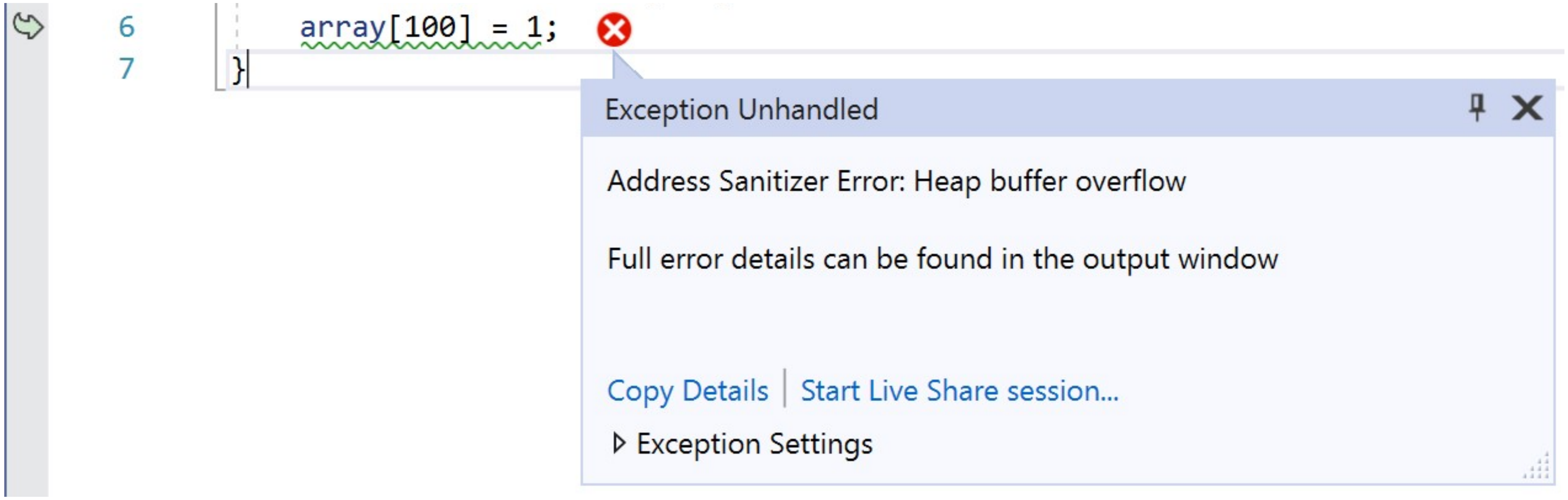
Address Sanitizer Error: Heap buffer overflow

Full error details can be found in the output window

Address Sanitizer (ASan)

IDE Exception Helper will be displayed when an issue is encountered
=> program execution will stop

ASan logging information => Output window



```
array[100] = 1;
}
```

Exception Unhandled

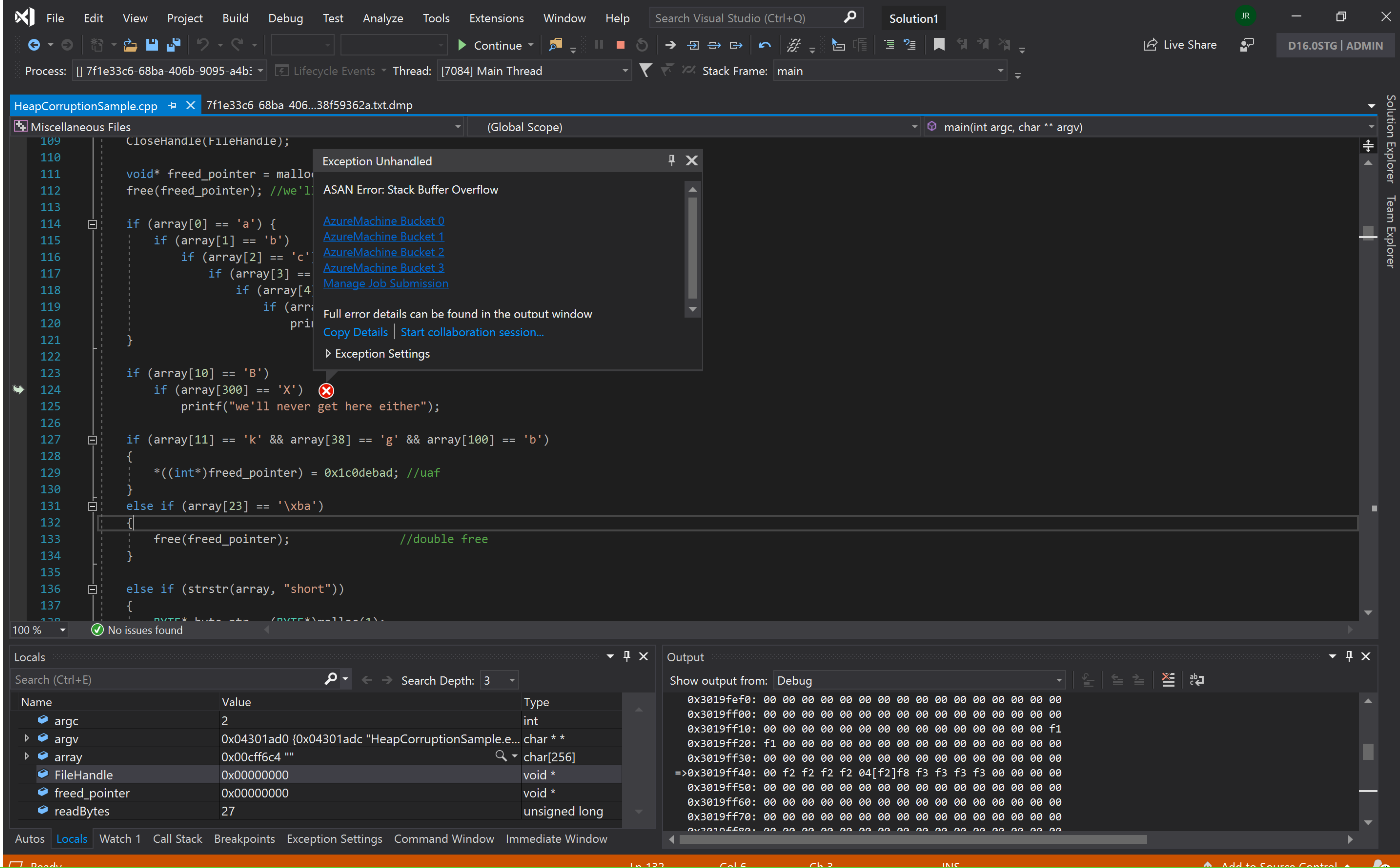
Address Sanitizer Error: Heap buffer overflow

Full error details can be found in the output window

Clang/LLVM

```
==27748==ERROR: AddressSanitizer: stack-use-after-scope on address 0x0055fc68 at pc 0x793d62de bp 0x0055fbf4 sp 0x0055fbe8
WRITE of size 80 at 0x0055fc68 thread T0
    #0 0x793d62f6 in __asan_wrap_memset d:\_work\5\s\llvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:764
    #1 0x77dd46e7  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2c46e7)
    #2 0x77dd4ce1  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2c4ce1)
    #3 0x75d408fe  (C:\WINDOWS\System32\KERNELBASE.dll+0x100f08fe)
    #4 0xa5ada0 in try_get_first_available_module minkernel\crts\ucrt\src\appcrt\internal\winapi_thunks.cpp:271
    #5 0xa5ae99 in try_get_function minkernel\crts\ucrt\src\appcrt\internal\winapi_thunks.cpp:326
    #6 0xa5b028 in __acrt_AppPolicyGetProcessTerminationMethodInternal minkernel\crts\ucrt\src\appcrt\internal\winapi_thunks.cpp:737
    #7 0xa606ad in __acrt_get_process_end_policy minkernel\crts\ucrt\src\appcrt\internal\win_policies.cpp:84
    #8 0xa52dcb in exit_or_terminate_process minkernel\crts\ucrt\src\appcrt\startup\exit.cpp:134
    #9 0xa52da7 in common_exit minkernel\crts\ucrt\src\appcrt\startup\exit.cpp:280
    #10 0xa52fb6 in exit minkernel\crts\ucrt\src\appcrt\startup\exit.cpp:293
    #11 0xa2deb3 in _scrt_common_main_seh d:\agent\_work\2\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:295
    #12 0x75ef6358  (C:\WINDOWS\System32\KERNEL32.DLL+0x6b816358)
    #13 0x77df7a93  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2e7a93)

Address 0x0055fc68 is located in stack of thread T0
SUMMARY: AddressSanitizer: stack-use-after-scope d:\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:764 in __asan_wrap_memset
Shadow bytes around the buggy address:
  0x300abf30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x300abf70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x300abf80: 00 00 00 00 00 00 00 00 00 00 00 00 00[f8]00 00
  0x300abf90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x300abfd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==27748==ABORTING
```

Snapshot



```
    CloseHandle(FileHandle);

    void* freed_pointer = mallo
    free(freed_pointer); //we'l

    if (array[0] == 'a') {
        if (array[1] == 'b')
            if (array[2] == 'c'
                if (array[3] ==
                    if (array[4
                        if (arr
                            pri
    }

    if (array[10] == 'B')
        if (array[300] == 'X')
            printf("we'll never get here either");

    if (array[11] == 'k' && array[38] == 'g' && array[100] == 'b')
    {
        *((int*)freed_pointer) = 0x1c0debad; //uaf
    }
    else if (array[23] == '\xba')
    {
        free(freed_pointer);              //double free
    }

    else if (strstr(array, "short"))
    {
```

Exception Unhandled

ASAN Error: Stack Buffer Overflow

AzureMachine Bucket 0
AzureMachine Bucket 1
AzureMachine Bucket 2
AzureMachine Bucket 3
Manage Job Submission

Full error details can be found in the output window

Copy Details | Start collaboration session...

Exception Settings

Locals

Search (Ctrl+E) Search Depth: 3

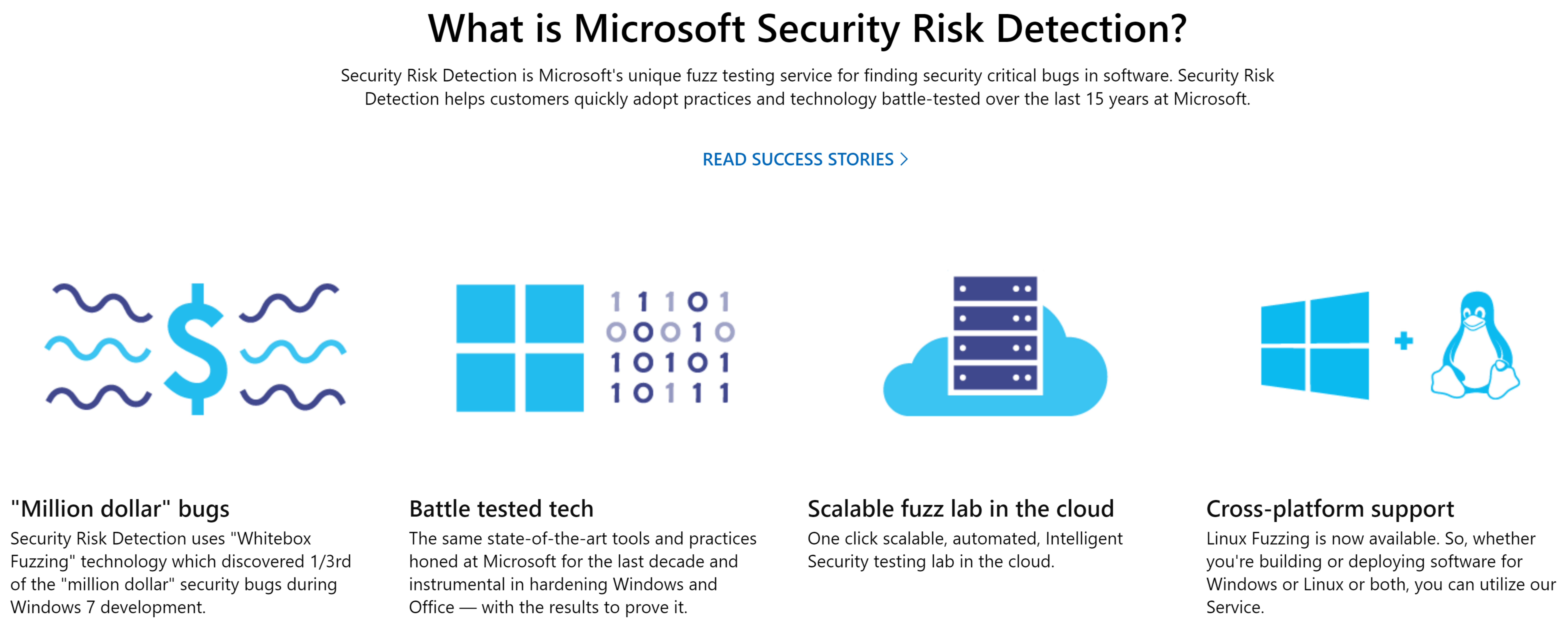
Name	Value	Type
argc	2	int
argv	0x04301ad0 {0x04301adc "HeapCorruptionSample.e...	char * *
array	0x00cff6c4 ""	char[256]
FileHandle	0x00000000	void *
freed_pointer	0x00000000	void *
readBytes	27	unsigned long

Output

Show output from: Debug

```
  0x3019fef0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3019ff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3019ff10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f1
  0x3019ff20: f1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3019ff30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x3019ff40: 00 f2 f2 f2 f2 04[f2]f8 f3 f3 f3 f3 00 00 00 00
  0x3019ff50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3019ff60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3019ff70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

{ ASan + Fuzzing } => Azure



What is Microsoft Security Risk Detection?

Security Risk Detection is Microsoft's unique fuzz testing service for finding security critical bugs in software. Security Risk Detection helps customers quickly adopt practices and technology battle-tested over the last 15 years at Microsoft.

READ SUCCESS STORIES >

"Million dollar" bugs

Security Risk Detection uses "Whitebox Fuzzing" technology which discovered 1/3rd of the "million dollar" security bugs during Windows 7 development.

Battle tested tech

The same state-of-the-art tools and practices honed at Microsoft for the last decade and instrumental in hardening Windows and Office — with the results to prove it.

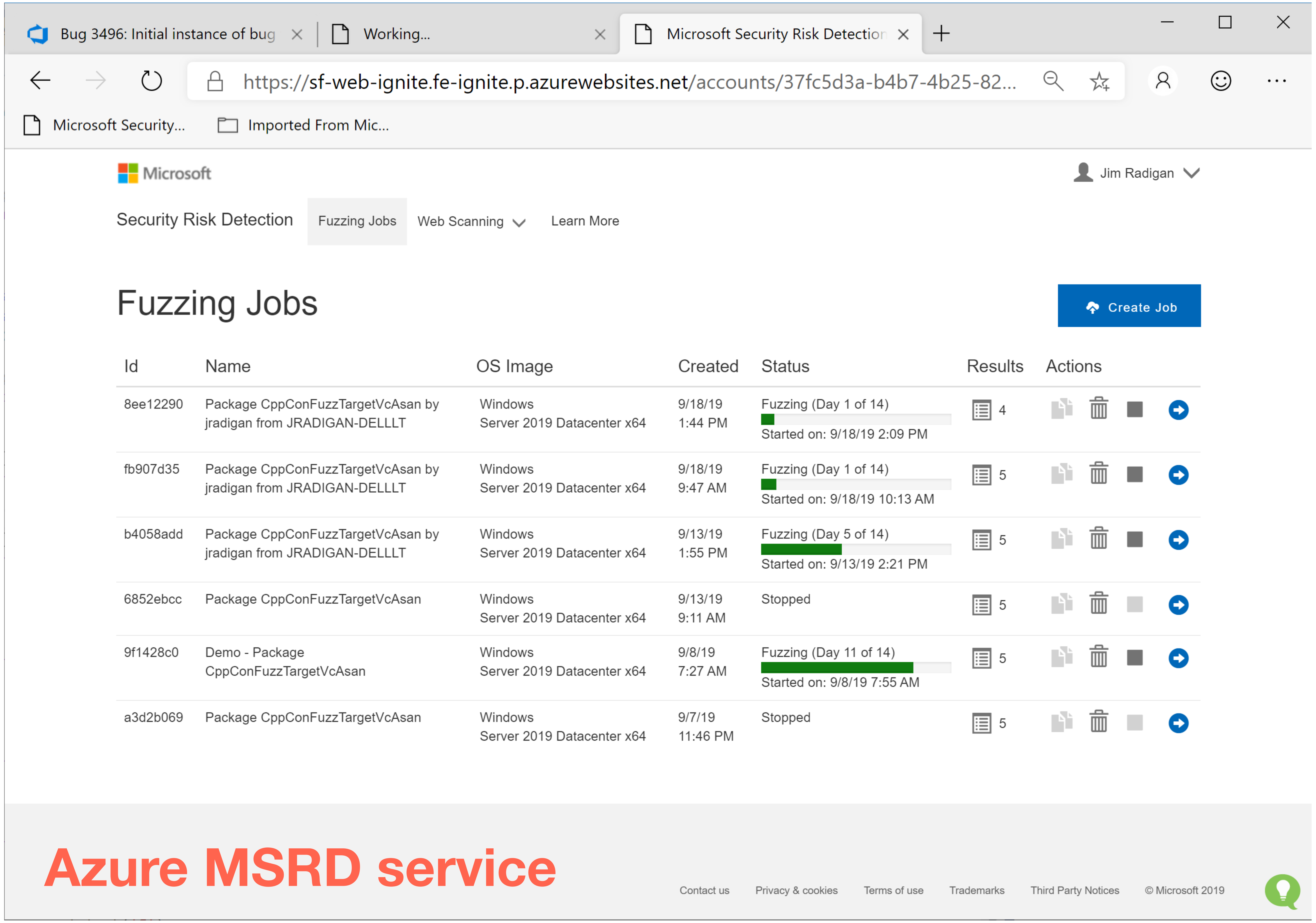
Scalable fuzz lab in the cloud

One click scalable, automated, Intelligent Security testing lab in the cloud.

Cross-platform support

Linux Fuzzing is now available. So, whether you're building or deploying software for Windows or Linux or both, you can utilize our Service.

{ ASan + Fuzzing } => Azure



Bug 3496: Initial instance of bug | Working... | Microsoft Security Risk Detection

https://sf-web-ignite.fe-ignite.p.azurewebsites.net/accounts/37fc5d3a-b4b7-4b25-82...

Microsoft Security... | Imported From Mic...

Microsoft — Jim Radigan

Security Risk Detection | Fuzzing Jobs | Web Scanning | Learn More

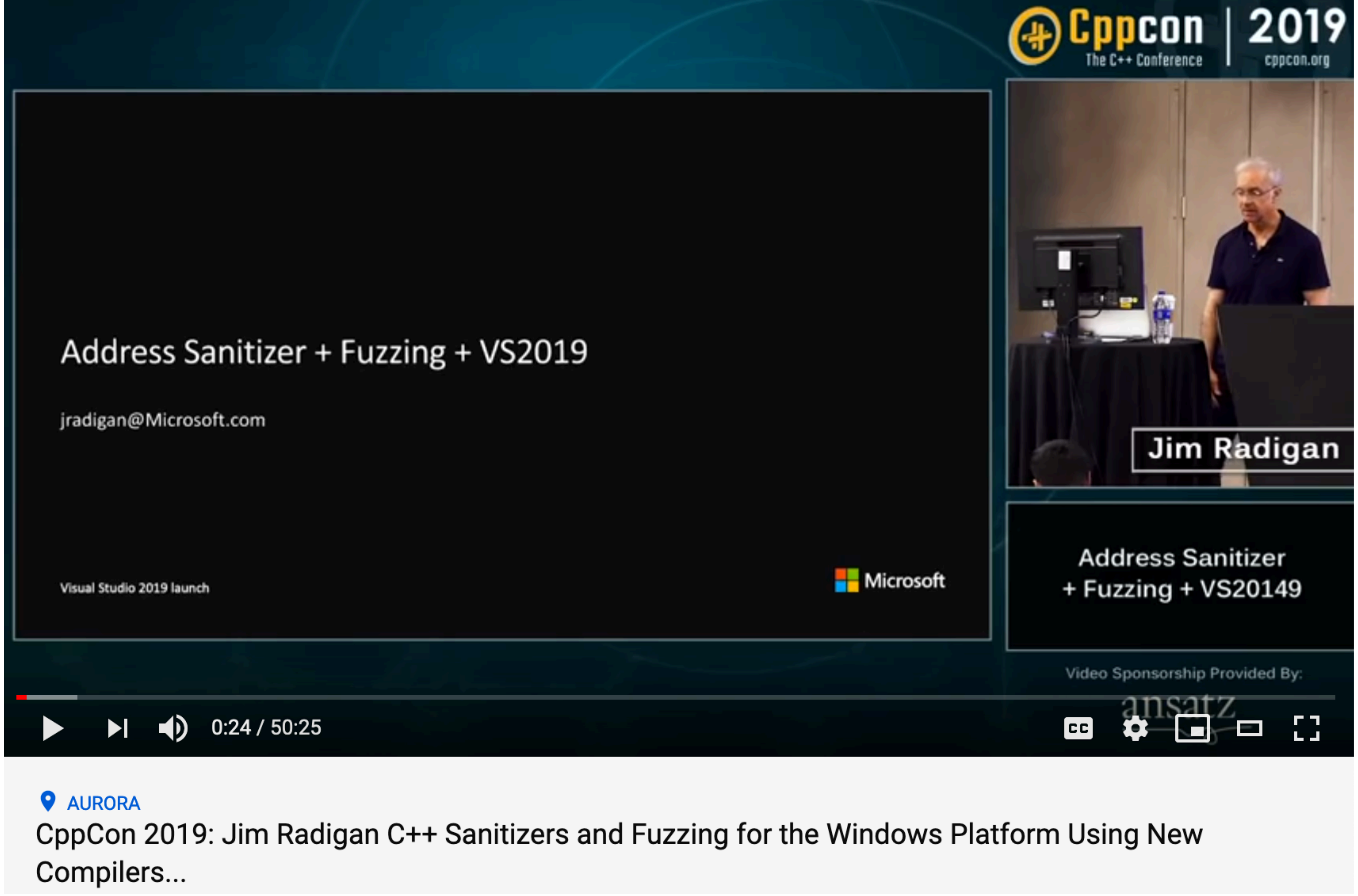
Fuzzing Jobs

Create Job

Id	Name	OS Image	Created	Status	Results	Actions
8ee12290	Package CppConFuzzTargetVcAsan by jradigan from JRADIGAN-DELLLT	Windows Server 2019 Datacenter x64	9/18/19 1:44 PM	Fuzzing (Day 1 of 14) Started on: 9/18/19 2:09 PM	4	
fb907d35	Package CppConFuzzTargetVcAsan by jradigan from JRADIGAN-DELLLT	Windows Server 2019 Datacenter x64	9/18/19 9:47 AM	Fuzzing (Day 1 of 14) Started on: 9/18/19 10:13 AM	5	
b4058add	Package CppConFuzzTargetVcAsan by jradigan from JRADIGAN-DELLLT	Windows Server 2019 Datacenter x64	9/13/19 1:55 PM	Fuzzing (Day 5 of 14) Started on: 9/13/19 2:21 PM	5	
6852ebcc	Package CppConFuzzTargetVcAsan	Windows Server 2019 Datacenter x64	9/13/19 9:11 AM	Stopped	5	
9f1428c0	Demo - Package CppConFuzzTargetVcAsan	Windows Server 2019 Datacenter x64	9/8/19 7:27 AM	Fuzzing (Day 11 of 14) Started on: 9/8/19 7:55 AM	5	
a3d2b069	Package CppConFuzzTargetVcAsan	Windows Server 2019 Datacenter x64	9/7/19 11:46 PM	Stopped	5	

Contact us | Privacy & cookies | Terms of use | Trademarks | Third Party Notices | © Microsoft 2019

Azure MSRD service


Cppcon The C++ Conference 2019 cppcon.org
Address Sanitizer + Fuzzing + VS2019
jradigan@Microsoft.com
Visual Studio 2019 launch
Microsoft
Jim Radigan
Address Sanitizer
+ Fuzzing + VS20149
Video Sponsorship Provided By:
ansatz
0:24 / 50:25
AURORA

CppCon 2019: Jim Radigan C++ Sanitizers and Fuzzing for the Windows Platform Using New Compilers...

https://www.youtube.com/watch?v=0EsqxGgYOQU

There's never too many sanitizers 🤓

- `ParmeSan`

  Find cheesy comments in code.

- `BipartiSan`

  Find code that uses two different containers in a complimentary way.

- `ArtiSan`

  Find code that took the writer a very long time to do and can be replaced with a common well tested library.

https://twitter.com/olafurw/status/1085544102870044674?s=21

Ólafur Waage
@olafurw

Fin

Status quo: `clang-tidy & ASan` on Windows

code::dive

November 20, 2019
Wrocław

clangpowertools.com
@ClangPowerTools

@ciura_victor

Victor Ciura
Principal Engineer
CAPHYON