

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

September 27, 2017



**Victor Ciura**  
Technical Lead, Advanced Installer  
[www.advancedinstaller.com](http://www.advancedinstaller.com)

# Intro

## Who Am I ?

## Context:

# Advanced Installer



[www.advancedinstaller.com](http://www.advancedinstaller.com)

- Powerful Windows Installer authoring tool (**IDE**)
- Helps developers and IT Pros create MSI/EXE, App-V and UWP AppX packages
- **14** year old code base, under active development (since 2003)
- **2.5** million lines of C++ code
- **134** Visual Studio projects (EXEs, DLLs, LIBs)
- Microsoft **Visual Studio 2017**
- **Monthly** release cycle (~3 week sprints)
- **Windows-only** deployment
- Strong Windows **SDK** dependencies: our code has a fairly wide Windows API surface area (because of the application domain)

# Intro

## Why Am I Here ?

# Intro

## Why Am I Here ?

“A 14 year old code base under active development, 2.5 million lines of C++ code, a few brave nerds, two powerful tools and one hot summer...”

or

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

# This talk is not about



VS

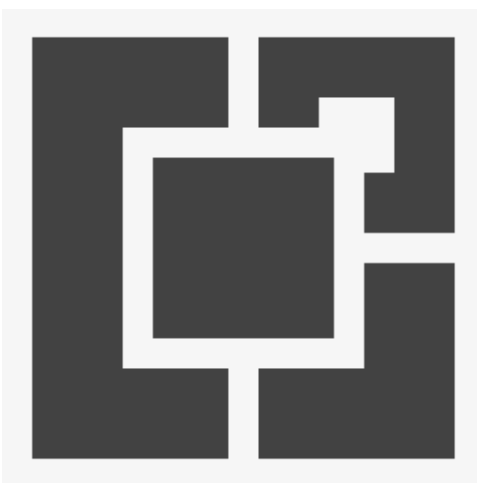


- We're a **Windows**-only dev team using Visual C++
- We're going to continue using **both** **Visual Studio** (2017) and **Clang** tools on the side, to modernize/refactor and improve our code quality

# Timeline

It all started a year ago, here, at CppCon...

- September, 2016 - started thinking about adopting **clang-format** (experimenting with various configs)
- September 21, 2016 - **CppCon**: “*clang-format Birds of a Feather*”
- October-November 2016 - preparing for clang-format adoption (rules, configs, exceptions, debates, strategy)
- December 16, 2016 - the BIG reformat (formatted *all* code with clang-format, using our custom style)
- December 2016-present - team workflow: use **CLangFormat VS extension** (auto-format on **save**)



<https://marketplace.visualstudio.com/items?itemName=LLVMExtensions.ClangFormat>

# Goals

- Building on the success of **clang-format** adoption within the team, we gained courage to experiment with **clang-tidy**
- New problem: getting all our code to fully **compile** with Clang, using the correct project settings (synced with Visual Studio) and Windows SDK dependencies
- We found several compatibility issues between MSVC compiler (VS2017) and Clang (4.0)
- Note that we were already using MSVC **/W4** and **/WX** on all our projects
- 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, batteries not included)
- Perform large scale **refactorings** on our code with clang-tidy (**modernize-\***, **readability-\***)
- Run **static analysis** on our code base to find subtle latent bugs





# 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]
```

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

```
Error: FilePath.cpp:36:17: 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);
```

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



# Fixes, fixes, fixes...



## Frequent offender:

```
Error: 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:344:7: note: candidate constructor (the implicit move constructor) not viable:  
no known conversion from 'const wchar_t [6]' to '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 &' (aka 'const basic_string<wchar_t, char_traits<wchar_t>,  
allocator<wchar_t> > &') for 1st argument
```

```
    Facet(const wstring & facet)  
      ^
```

# Timeline

- January 12, 2017 - started playing with Clang for Windows (LLVM 3.9.1)
- January 24, 2017 - first commit, started fixing the Clang errors/warnings (Note: we were already on **MSVC /W4 /WX**)
- February 3, 2017 - created a clang++ compilation **.bat** file (crude automation attempt)
- March 7, 2017 - upgraded the clang++ batch file to a **PowerShell** script (**clang-build.ps1**)
- March 13, 2017 - our PS script also gains the ability to run clang-tidy checks
- March 2017 - first experiments with **clang-tidy** on our source code (just some core libraries)
- April 11, 2017 🎉 - able to compile our **whole** codebase with Clang 3.9.1 (*some default warnings disabled*) ~ **3 months**
- April 12, 2017 - created a **Jenkins** job for Clang build (every SCM change is compiled with Clang)
- May 2017 - great improvements to our PowerShell script (PCH, parallel compilation, project filters, SDK versions)
- June 2017 - more experiments with **clang-tidy** on our source code (better coverage)
- June 16, 2017 - upgraded from VS2015 to **VS2017** (we also needed to update our Clang PS script)

# Timeline

- July 3, 2017 - started work on a custom clang-based refactoring tool (`libTooling`)
- July 10, 2017 - fixed new Clang 4 issues and upgraded to **4.0.1**
- July 2017 - started to tackle Clang `-Wall` warnings in our code
- August 2017 - made extensive code transformations with our custom `libTooling` helpers
- August 24, 2017 🎉 - our whole codebase compiles with Clang `-Wall`
- August 2017 - started work on our “**Clang Power Tools**” extension for Visual Studio
- August 25, 2017 - first refactorings with `clang-tidy: modernize-use-nullptr, modernize-loop-convert`
- Aug-Sep 2017 - multiple code transformations with `clang-tidy: modernize-*, readability-*, misc-*, ...`
- September 2017 - started to fix `-Wextra` warnings (*in progress...*)
- September 11, 2017 - upgraded to LLVM **5.0** (fixed new warnings) [`-Wunused-lambda-capture`]
- September 11, 2017 - **open-sourced** “Clang Power Tools” extension for VS and PowerShell script “`clang-build.ps1`”
- September 26, 2017 - published our “Clang Power Tools” extension to **Visual Studio Marketplace**
- **September 27, 2017** - here we are 😊



# 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`



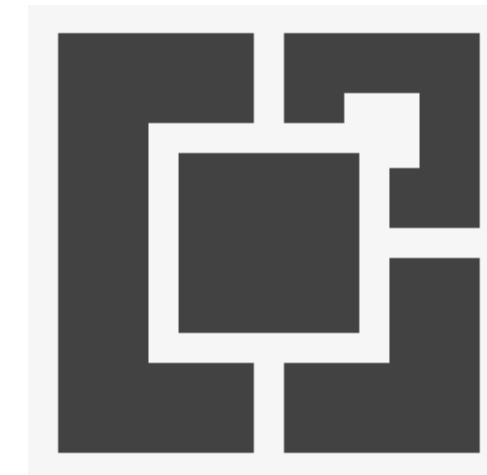
## Large scale refactorings we performed:

- `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-*`

# How Did We Achieve All That ?



**TOOLS**



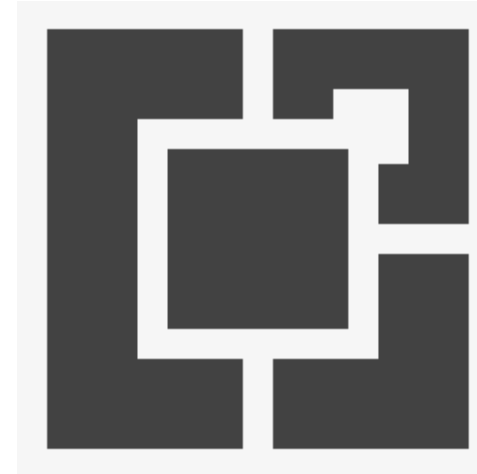
 **Power Team** 

**PowerShell scripts**



**Gabriel Diaconița**

**Clang Power Tools  
VS Extension**



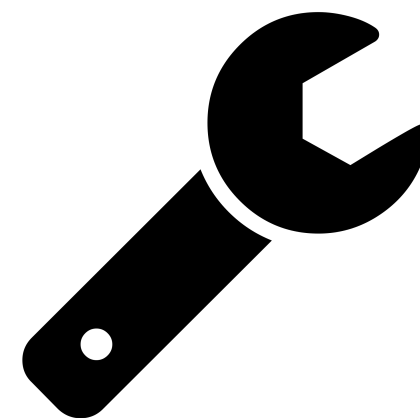
**Ionuț Enache  
Alexandru Dragomir**

**LibTooling**



**Mihai Udrea**

**Fixing Clang  
errors/warnings in our code**



**Myself & many others...**



## We started simple...



### compile.bat

```
SET INCLUDE="..\..;C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\include;C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\atlmfc\include;C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt;C:\Program Files (x86)\Windows Kits\8.1\Include\um;C:\Program Files (x86)\Windows Kits\8.1\Include\shared;"

setlocal EnableDelayedExpansion

For /R . %%G IN (*.cpp) do (
clang++ "%%G" -std=c++14 -fsyntax-only -Werror -Wmicrosoft -Wno-invalid-token-paste -Wno-unused-variable -Wno-unused-value -fms-extensions -fdelayed-template-parsing -fms-compatibility -D_ATL_NO_HOSTING -DUNICODE -D_UNICODE -DWIN32 -D_DEBUG -DDEBUG

IF !errorlevel! NEQ 0 goto exit
)
```



### tidy.bat

```
SET INCLUDE="..\..;C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\include;C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\atlmfc\include;C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt;C:\Program Files (x86)\Windows Kits\8.1\Include\um;C:\Program Files (x86)\Windows Kits\8.1\Include\shared;"

clang-tidy %1 -checks=-*,modernize-* -fix -- -std=c++14 -Werror -Wno-invalid-token-paste -Wmicrosoft -fms-extensions -fdelayed-template-parsing -fms-compatibility -D_ATL_NO_HOSTING -DUNICODE -D_UNICODE -DWIN32 -D_DEBUG -DDEBUG

clang-format -style=file -i %1
```

But soon came...



# Clang PowerShell Script

- over 1,000 lines
- very configurable (many parameters)
- supports both clang compile and tidy workflows
- works directly on Visual Studio .vcxproj files (or MSBuild projects)  
(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
- include-dirs** Directories to be used for includes
- parallel** Run clang++ in parallel mode, on all logical CPU cores (incompatible with **-tidy-fix**)
- continue** Continue project compilation even when errors occur
- clang-flags** Flags passed to clang++
- literal** Disable name regex matching for projects and source files
- tidy** Run specified clang-tidy checks
- tidy-fix** Run specified clang-tidy checks with auto-fix
- vs-ver** Visual Studio Edition (eg. "2017")
- vs-sku** Visual Studio SKU (eg. "Community", "Professional")
- win-sdk-ver** Default Windows SDK Version (if projects are missing Windows Target Platform)

**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 bootstrapper PS script (eg. `sample-clang-build.ps1`),  
that pre-loads some of the constant configurations for your team.

`sample-clang-build.ps1 ==> clang-build.ps1`

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

➔ Runs clang-tidy, using all *modernize* checks, on all CPPs containing 'meow' in their name, from the projects containing 'foo' or 'bar' in their names.

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

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

# Bootstrapper PS script



## sample-clang-build.ps1

```

param( [alias("proj")] [Parameter(Mandatory=$false)] [string[]] $aVcxprojToCompile
, [alias("proj-ignore")] [Parameter(Mandatory=$false)] [string[]] $aVcxprojToIgnore
, [alias("file")] [Parameter(Mandatory=$false)] [string] $aCppToCompile
, [alias("parallel")] [Parameter(Mandatory=$false)] [switch] $aUseParallelCompile
, [alias("continue")] [Parameter(Mandatory=$false)] [switch] $aContinueOnError
, [alias("literal")] [Parameter(Mandatory=$false)] [switch] $aDisableNameRegexMatching
, [alias("tidy")] [Parameter(Mandatory=$false)] [string] $aTidyFlags
, [alias("tidy-fix")] [Parameter(Mandatory=$false)] [string] $aTidyFixFlags
)

# -----

Set-Variable -name kClangCompileFlags -value @(
    "-std=c++14"
    , "-Wall"
    , "-fms-compatibility-version=19.10"
    , "-Wmicrosoft"
    , "-Wno-invalid-token-paste"
    , "-Wno-unknown-pragmas"
    , "-Wno-unused-value"
) -Option Constant

Set-Variable -name kIncludeDirectories -value @(
    "third-party"
    , "third-party\WTL\Include"
) -Option Constant

Set-Variable -name kVisualStudioVersion -value "2017" -Option Constant
Set-Variable -name kVisualStudioSku -value "Professional" -Option Constant

```



**Using The PowerShell Script**



**Jenkins CI Configuration**



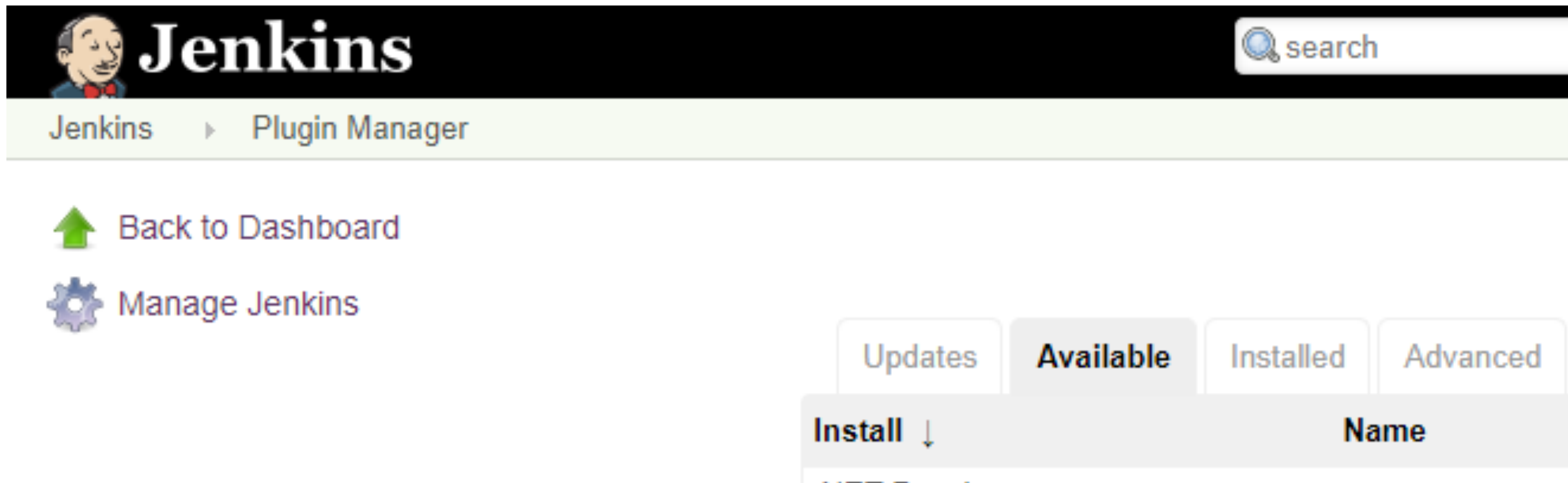
# Jenkins CI Configuration

Install PowerShell plugin (available from Jenkins gallery)



## Manage Plugins

Add, remove, disable or enable plugins that can extend the functionality of Jenkins.



The screenshot shows the Jenkins web interface. At the top left is the Jenkins logo and name. To the right is a search bar. Below the header is a breadcrumb trail: Jenkins > Plugin Manager. On the left side, there are two navigation links: 'Back to Dashboard' with an upward arrow icon and 'Manage Jenkins' with a gear icon. On the right side, there are four tabs: 'Updates', 'Available' (which is selected), 'Installed', and 'Advanced'. Below the tabs is a table with a header row containing 'Install ↓' and 'Name'.

<https://wiki.jenkins.io/display/JENKINS/PowerShell+Plugin>



# Jenkins CI Configuration

## Install PowerShell plugin

Jenkins		Plugin Manager	
<input type="checkbox"/>	<a href="#">Plain Credentials Plugin</a>	Allows use of plain strings and files as credentials.	<a href="#">1.4</a>
<input checked="" type="checkbox"/>	<a href="#">PowerShell plugin</a>	This plugin allows Jenkins to invoke <a href="#">Windows PowerShell</a> as build scripts.	<a href="#">1.3</a>
<input type="checkbox"/>	<a href="#">SCM API Plugin</a>	This plugin provides a new enhanced API for interacting with SCM systems.	<a href="#">2.2.2</a>

<https://wiki.jenkins.io/display/JENKINS/PowerShell+Plugin>





# Jenkins CI Configuration

- Create a **new job** just for clang builds

or

- Attach a **new build step** on an existing job

The screenshot shows the Jenkins 'Build' configuration page. At the top, there is a section titled 'Build'. Below it is a button labeled 'Add build step' with a downward arrow. A dropdown menu is open, listing various build steps. The 'Windows PowerShell' option is highlighted with a blue background. Other options include 'Advanced Installer', 'Build a Visual Studio project or solution using MSBuild', 'Execute Windows batch command', 'Execute shell', 'Execute shell script on remote host using ssh', 'Inject environment variables', 'Invoke Ant', 'Invoke top-level Maven targets', 'Set build status to "pending" on GitHub commit', and '[ArtifactDeployer] - Deploy the artifacts from build workspace to remote locations'.



# Jenkins CI Configuration



Reference PowerShell script from the job working directory.

Both the bootstrapper PS script (eg. `ai-clang-build.ps1`) and the main PS script (`clang-build.ps1`) should be in the same directory.

## Build

Windows PowerShell

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

See [the list of available environment variables](#)

Add build step



# Jenkins CI Configuration



If you configured Clang build as a new Jenkins job, a good workflow is to track and build any SCM changes:

## Build Triggers

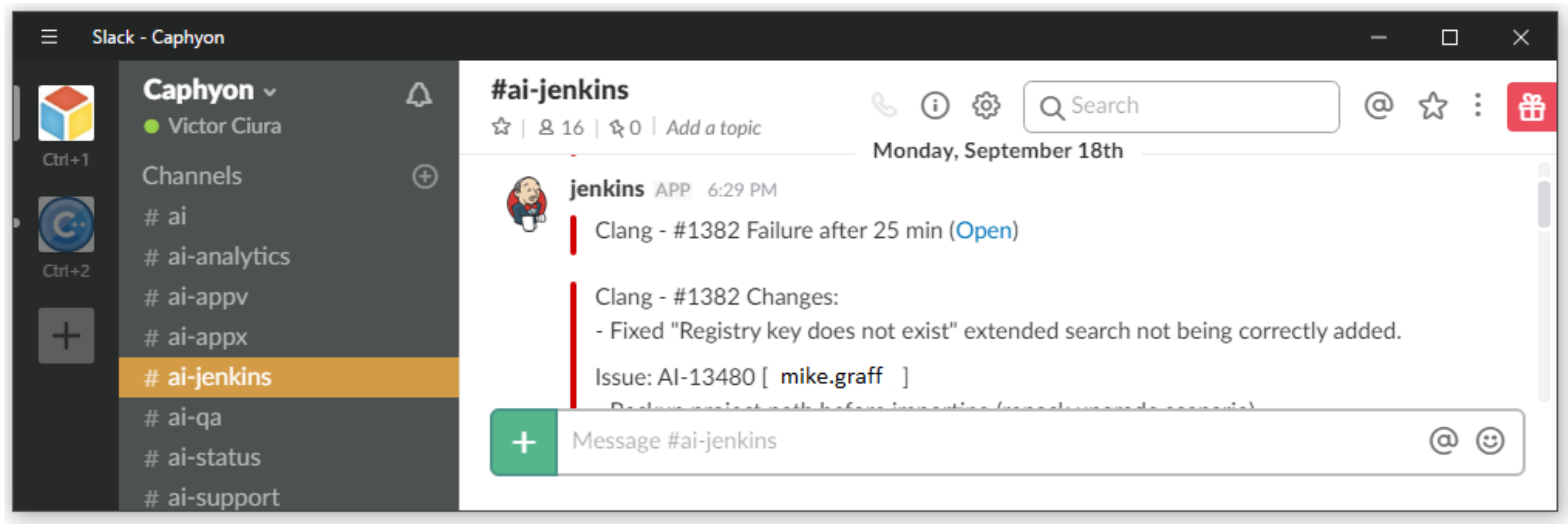
- Trigger builds remotely (e.g., from scripts)
- Build after other projects are built
- Build periodically
- GitHub hook trigger for GITScm polling
- Poll SCM



# Jenkins CI Workflow



When Clang build is broken...



Slack bot alert → #ai-jenkins



# Jenkins CI Workflow



When Clang build is broken...

Team devs email alert →

The screenshot shows an email client interface with a message from Jenkins. The message subject is "[AIROBOT] Build Still Failing Clang - Revision: 81423". The email body contains the following information:

- BUILD FAILURE**
- Build URL: <http://airobot/job/Clang/1385/>
- Project: Clang
- Date of build: Tue, 19 Sep 2017 18:05:05 +0300
- Build duration: 49 min

A blue highlighted section titled "CHANGES" contains the following text:

Revision 81423 by [redacted] (Added support for using formatted references for Service failure operations. Issue: AI-11790)

Below this, a list of files is shown with "edit" icons:

- advinst\msicomp\appxcfg\AppXNtServiceSync.cpp
- advinst\msicomp\servconfigfailactions\IMsiServConfigFailActionsTable.h
- advinst\msicomp\servconfigfailactions\MsiServConfigFailActionsRow.cpp
- advinst\msicomp\servconfigfailactions\MsiServConfigFailActionsRow.h
- advinst\msicomp\servconfigfailactions\MsiServConfigFailActionsTable.cpp
- advinst\msicomp\servconfigfailactions\MsiServConfigFailActionsTable.h
- advinst\msicomp\servinst\MsiServInstView.cpp
- advinst\msicomp\servinst\ServConfigFailActionsView.cpp

At the bottom, it indicates "1 attachment: build.log 431 KB".



# Jenkins CI Workflow



When Clang build is broken...

Team devs email alert →

The screenshot shows an Outlook window titled 'Jenkins'. The left sidebar shows a folder structure with 'Jenkins' selected. The main pane displays an email from 'Jenkins' with the subject '[AIROBOT] Build Still Failing Clang - Revision: 81423'. The email body contains a list of file paths and error messages:

```
6: C:\Jenkins\Clang\workspace\platform\os\UrlUtil.cpp
5: C:\Jenkins\Clang\workspace\platform\os\Version.cpp
4: C:\Jenkins\Clang\workspace\platform\os\WindowUtil.cpp
3: C:\Jenkins\Clang\workspace\platform\os\WinVersion.cpp
2: C:\Jenkins\Clang\workspace\platform\os\WorkerThread.cpp
1: C:\Jenkins\Clang\workspace\platform\os\WowRedirection.cpp
Error: C:\Jenkins\Clang\workspace\stubs\setup\LaunchManager.cpp:264:13: error: no viable conversion from
'ATL::CStringT > >' to 'std::wstring' (aka 'basic_string, allocator >')
wstring cmdWithExe = exeName + L" ";
^
C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\VC\Tools\MSVC\14.11.25503\include
\xstring:1922:2: note: candidate constructor not viable: no known conversion from 'ATL::CStringT > >' to
'const std::basic_string, std::allocator > &' for 1st argument
basic_string(const basic_string& _Right)
^
C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\VC\Tools\MSVC\14.11.25503\include
\xstring:1975:2: note: candidate constructor not viable: no known conversion from 'ATL::CStringT > >' to
'const wchar_t *const' for 1st argument
basic_string(In_z_const_Elem * const_Ptr)
^
C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\VC\Tools\MSVC\14.11.25503\include
\xstring:2052:2: note: candidate constructor not viable: no known conversion from 'ATL::CStringT > >' to
'std::basic_string, std::allocator > &&' for 1st argument
basic_string(basic_string&& _Right) NOEXCEPT
^
```

The email also indicates there is 1 attachment: build.log (431 KB).

# What About Developer Workflow?



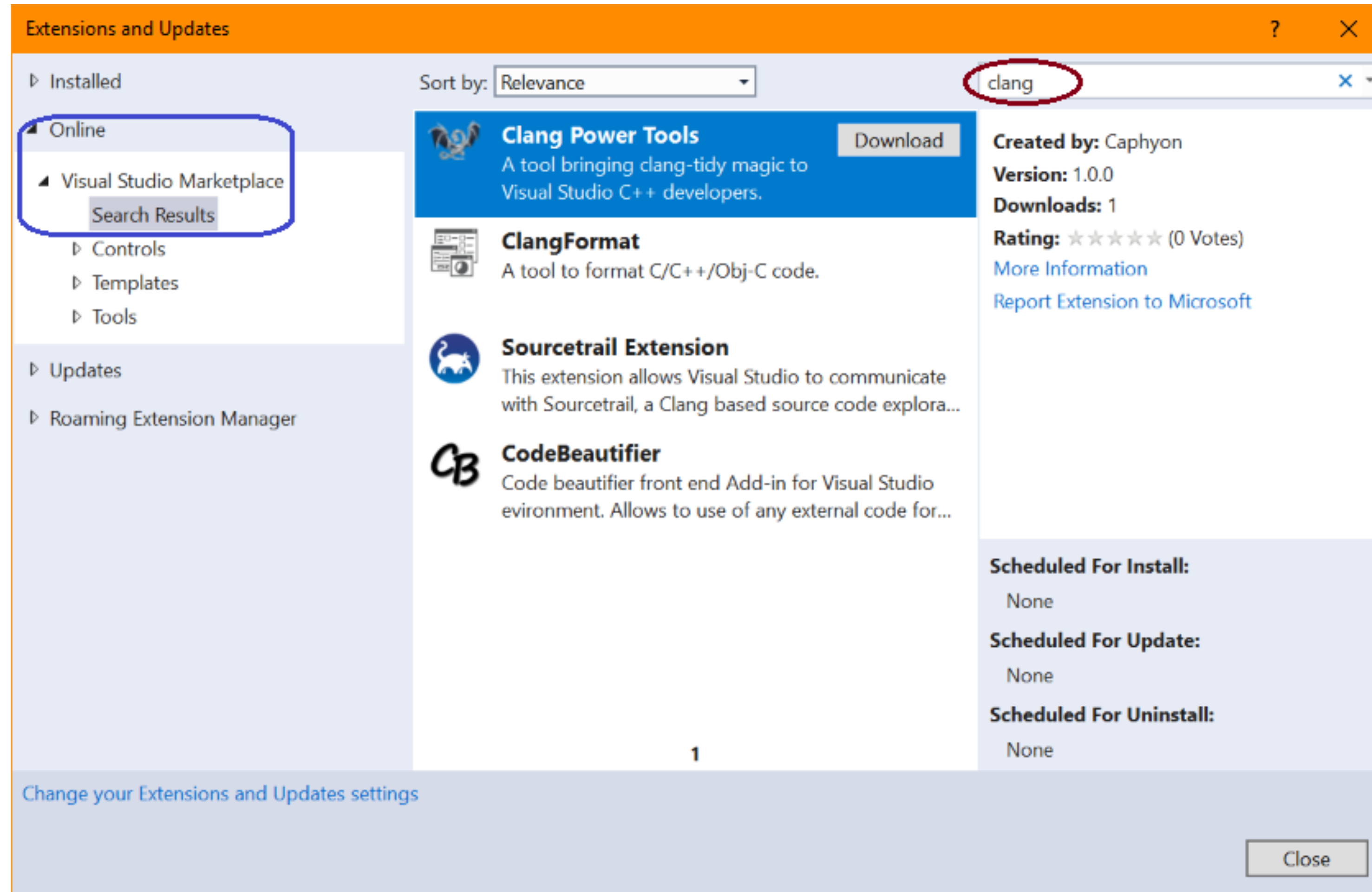
+





# Install The "Clang Power Tools" Visual Studio Extension

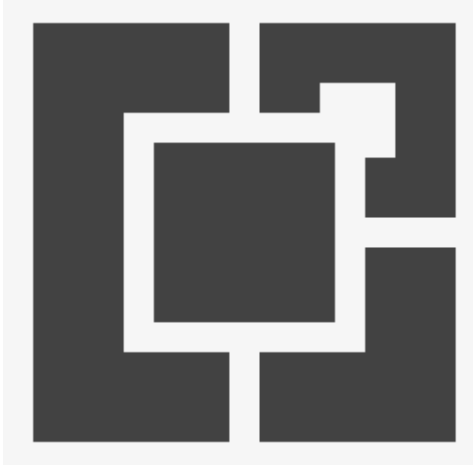
[Tools]  
↓  
Extensions and updates



Requires "Clang for Windows" (LLVM pre-built binary) to be installed.

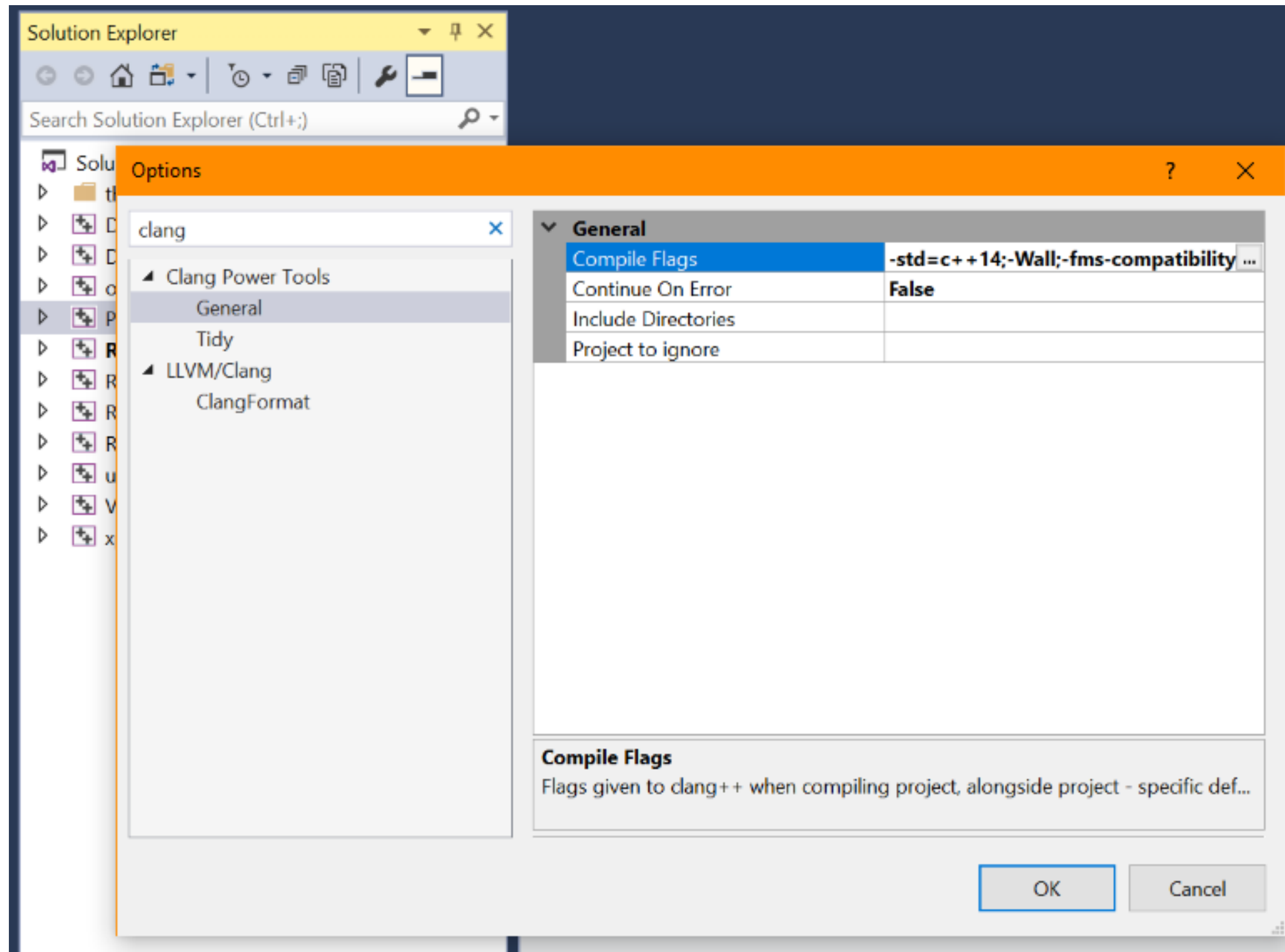
<http://releases.lvm.org/5.0.0/LLVM-5.0.0-win64.exe>



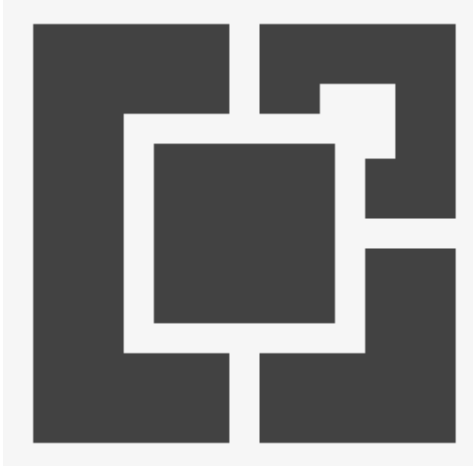


# Configure The "Clang Power Tools" Visual Studio Extension

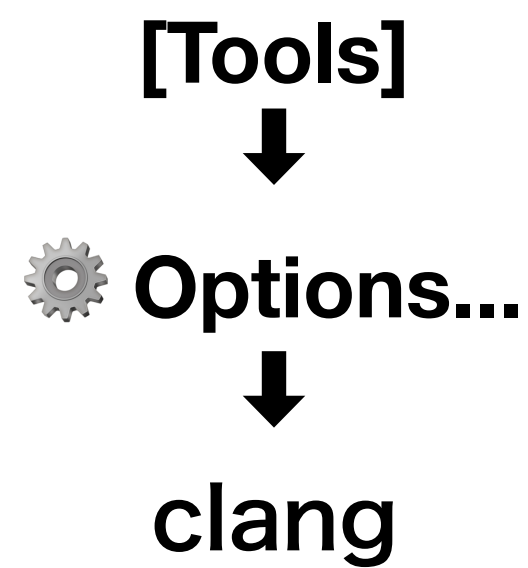
[Tools]  
 ↓  
 Options...  
 ↓  
 clang



← Compilation settings



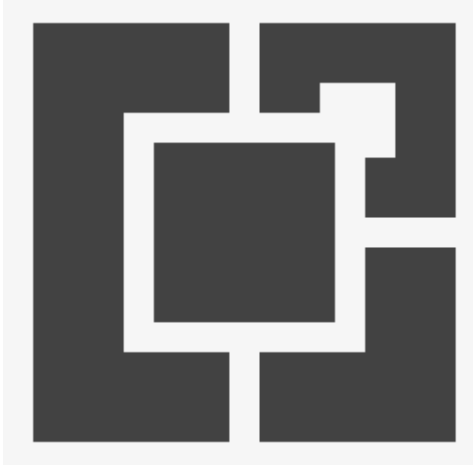
# Configure The "Clang Power Tools" Visual Studio Extension



The screenshot shows the Visual Studio Options dialog for the 'clang' extension. The 'String Collection Editor' dialog is open, displaying a list of clang++ flags. The flags listed are:

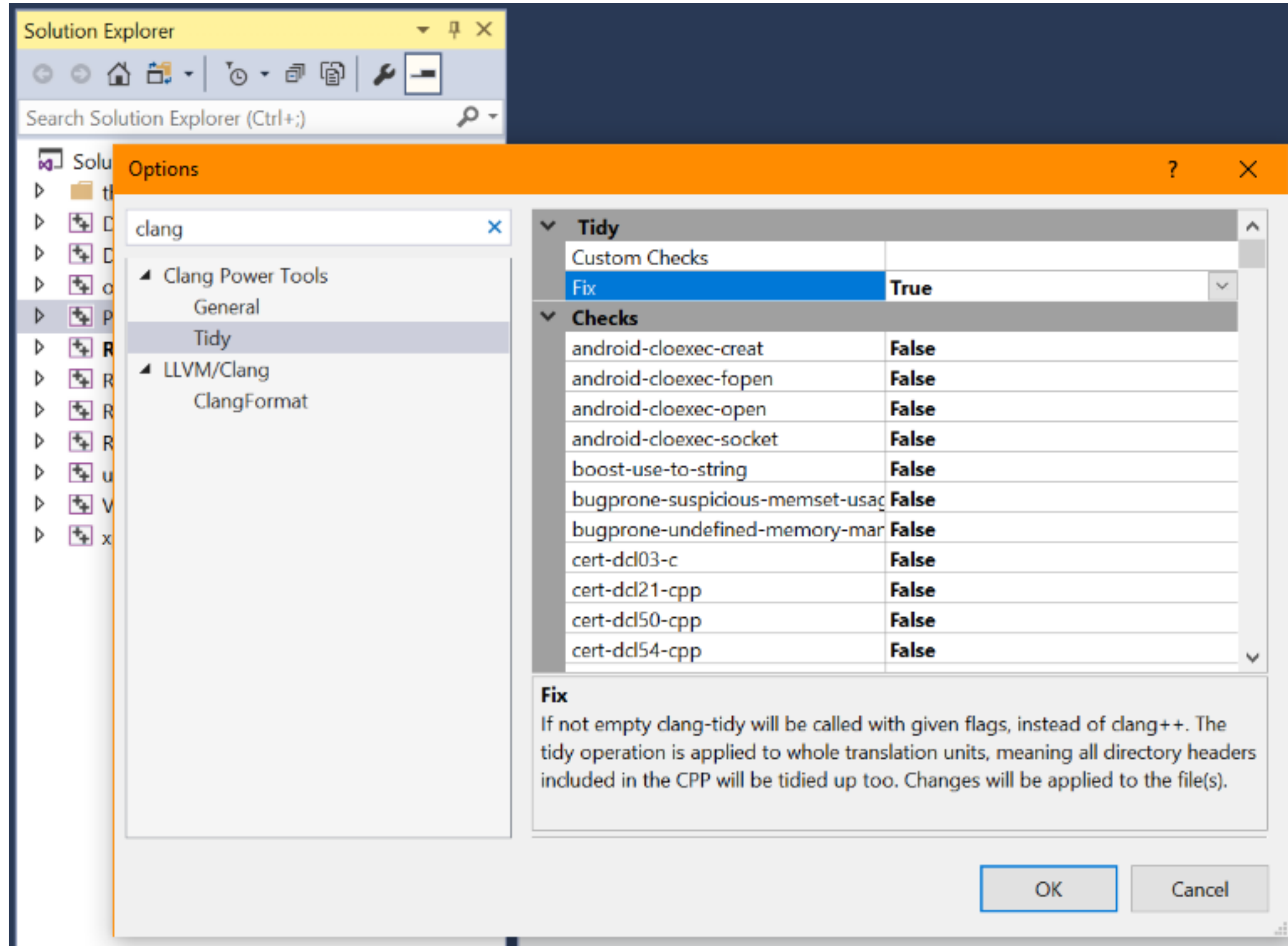
- std=c++14
- Wall
- fms-compatibility-version=19.10
- fms-compatibility
- Wmicrosoft
- Wno-invalid-token-paste
- Wno-unknown-pragmas
- Wno-unused-variable
- Wno-unused-value

← clang++ flags

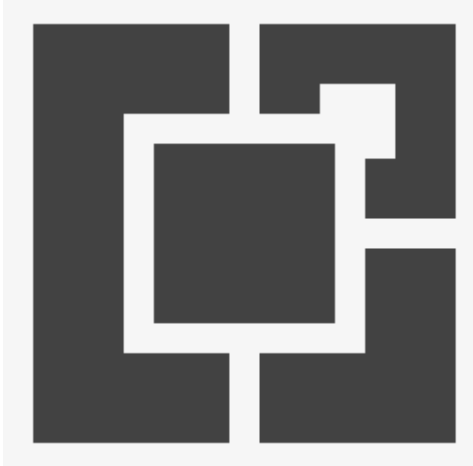


# Configure The "Clang Power Tools" Visual Studio Extension

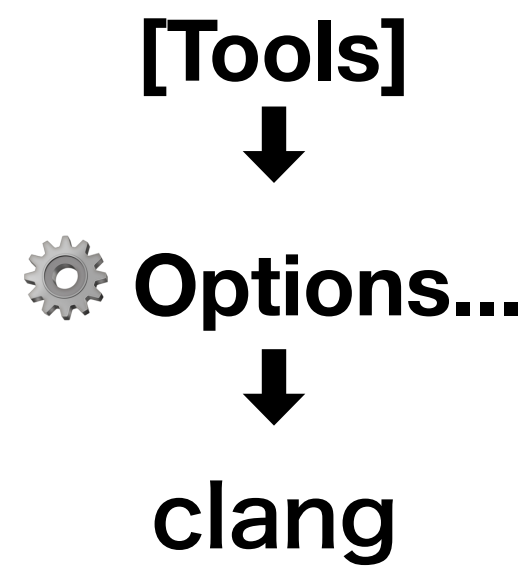
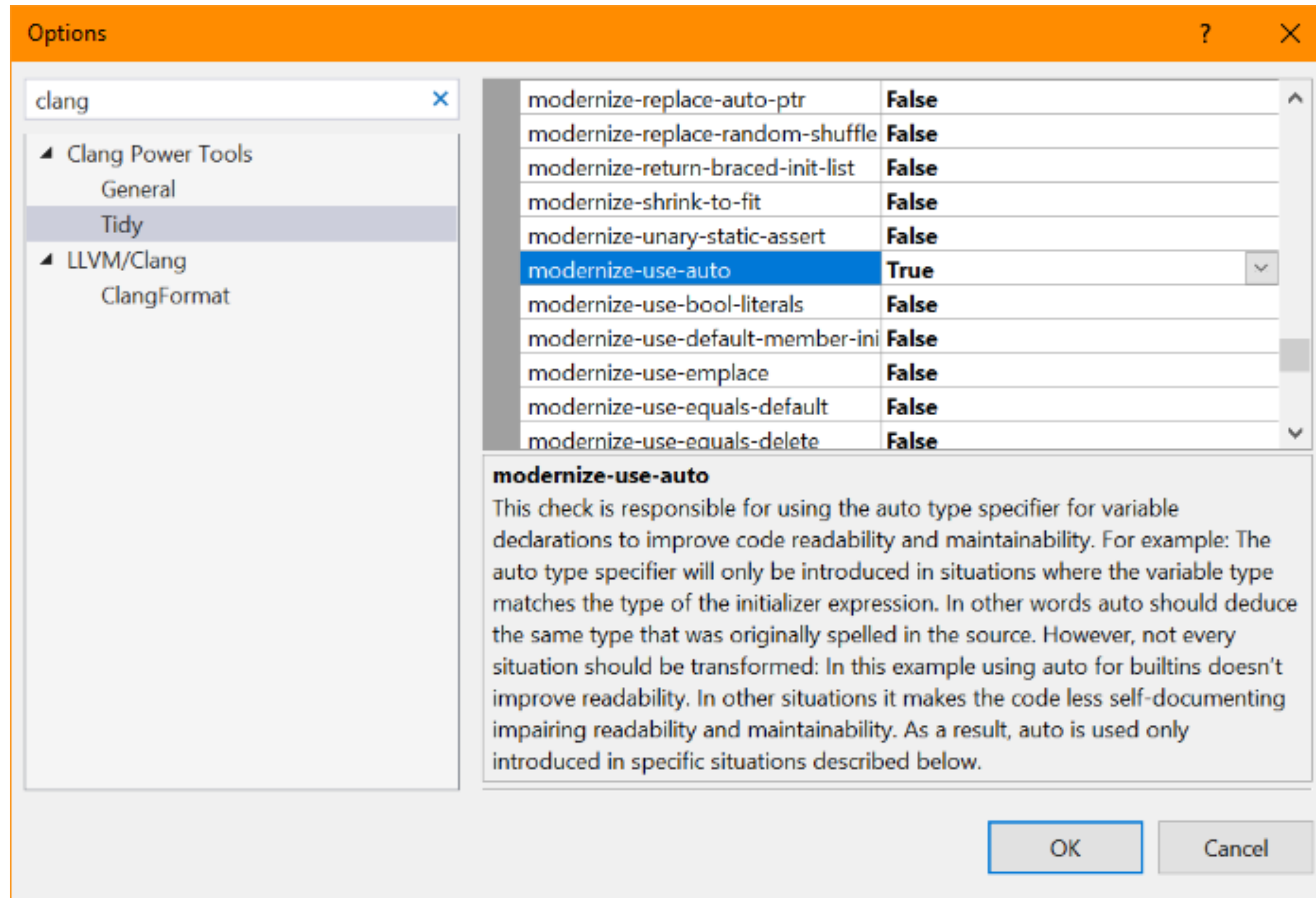
[Tools]  
↓  
Options...  
↓  
clang



← clang-tidy settings



# Configure The "Clang Power Tools" Visual Studio Extension

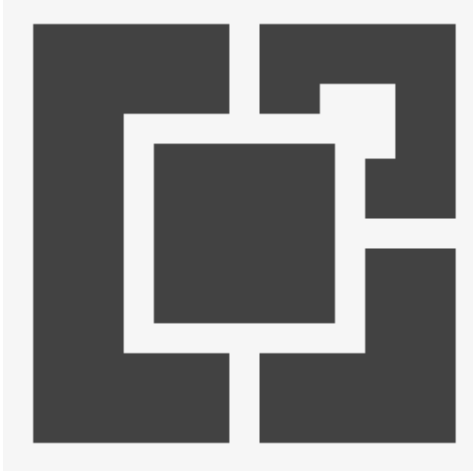
The screenshot shows the Visual Studio Options dialog for the clang extension. The left pane shows the tree structure: clang > Clang Power Tools > Tidy. The right pane shows a list of clang-tidy checks with their status. The 'modernize-use-auto' check is selected and set to 'True'. Below the list is a detailed description for the selected check.

modernize-replace-auto-ptr	False
modernize-replace-random-shuffle	False
modernize-return-braced-init-list	False
modernize-shrink-to-fit	False
modernize-unary-static-assert	False
<b>modernize-use-auto</b>	<b>True</b>
modernize-use-bool-literals	False
modernize-use-default-member-init	False
modernize-use-emplace	False
modernize-use-equals-default	False
modernize-use-equals-delete	False

**modernize-use-auto**  
This check is responsible for using the auto type specifier for variable declarations to improve code readability and maintainability. For example: The auto type specifier will only be introduced in situations where the variable type matches the type of the initializer expression. In other words auto should deduce the same type that was originally spelled in the source. However, not every situation should be transformed: In this example using auto for builtins doesn't improve readability. In other situations it makes the code less self-documenting impairing readability and maintainability. As a result, auto is used only introduced in specific situations described below.

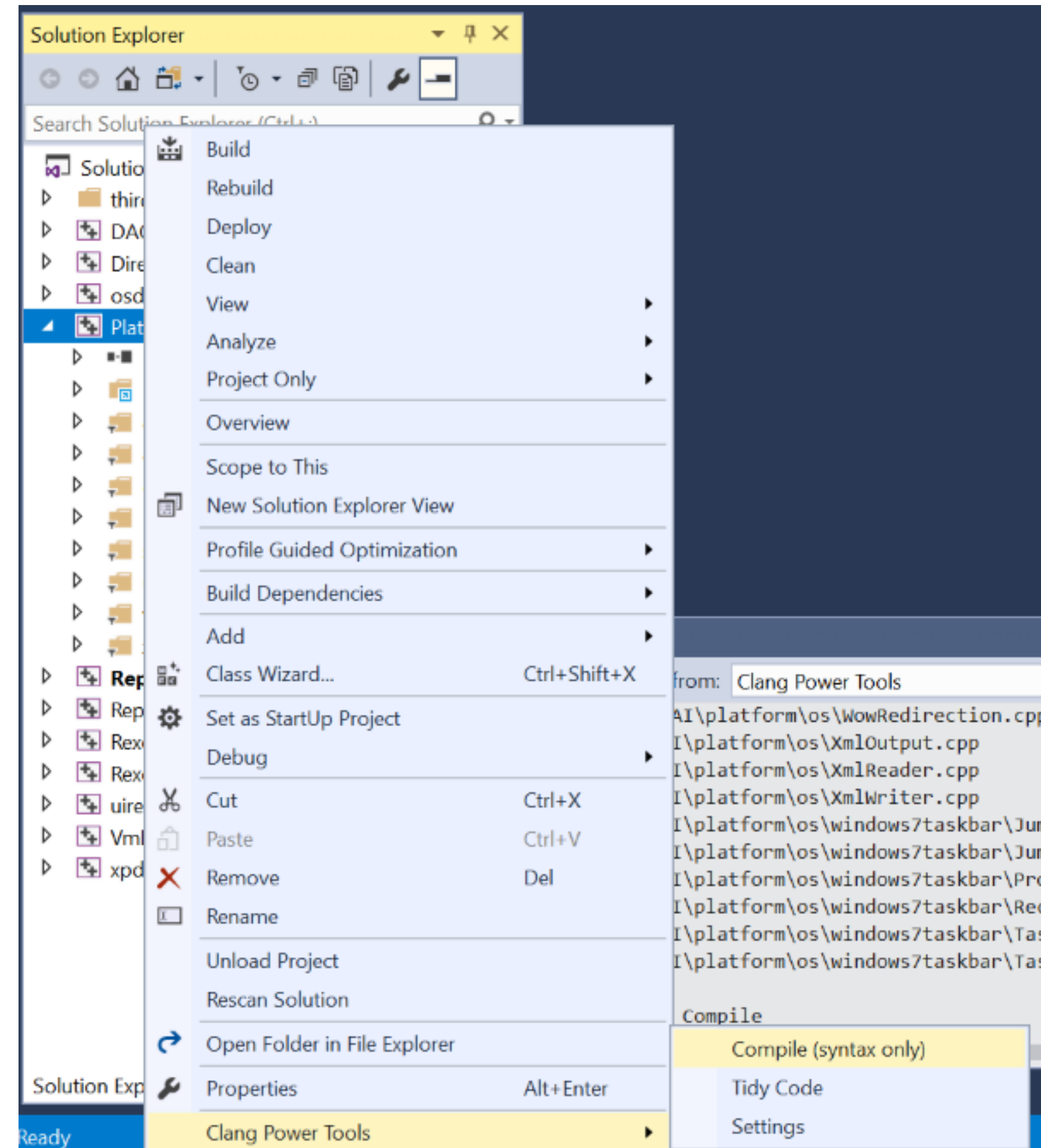
← clang-tidy checks

← inline documentation



# Using The "Clang Power Tools" Visual Studio Extension

Run Clang Power Tools on a whole project or solution →

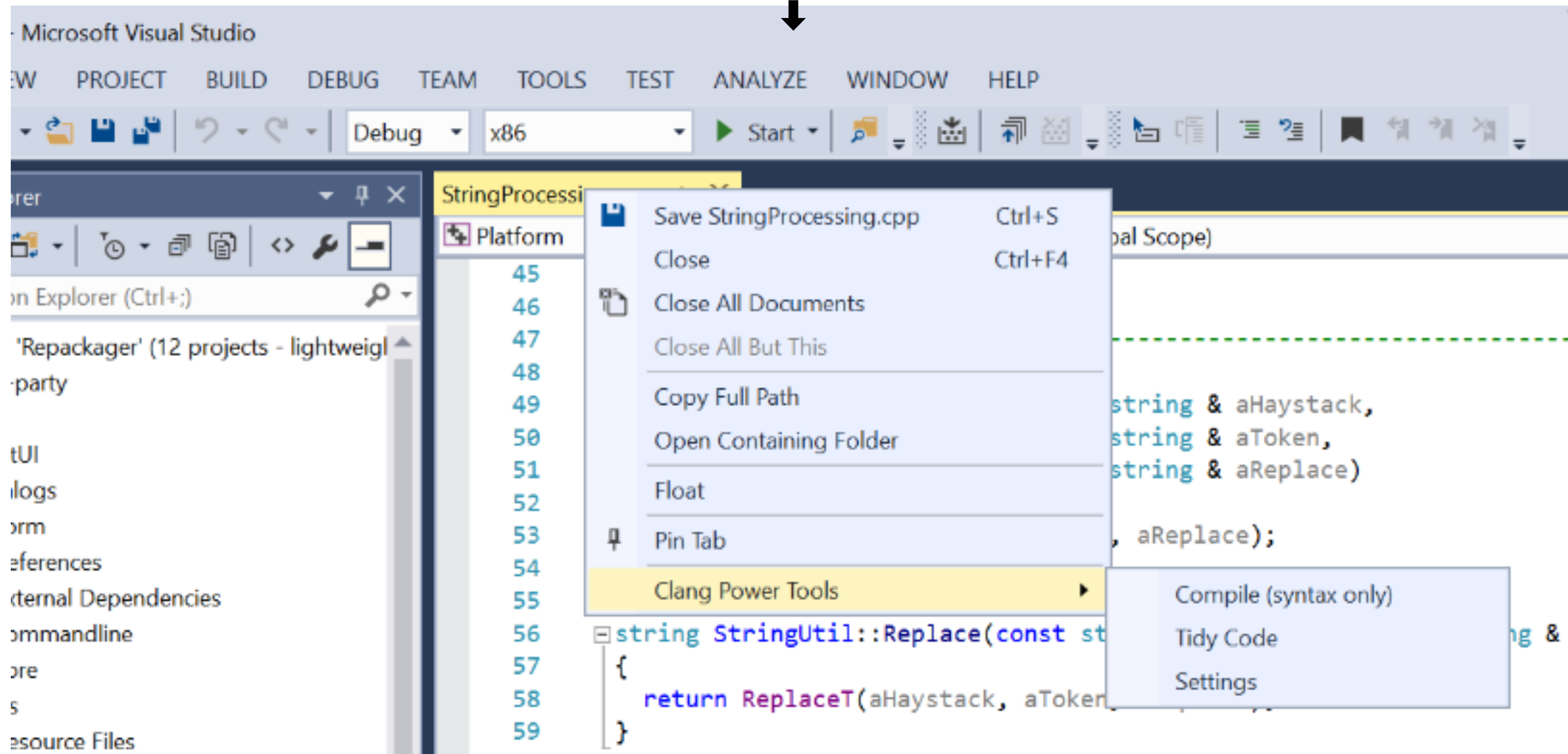


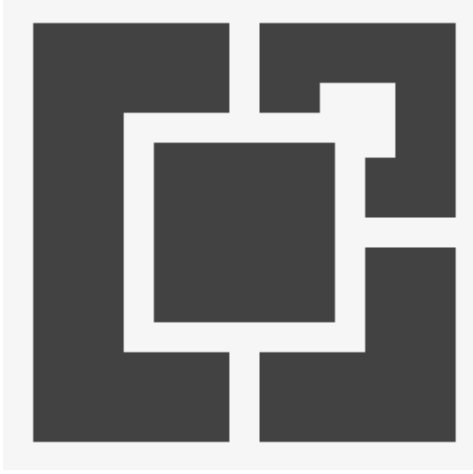
← Compile or Tidy code



# Using The "Clang Power Tools" Visual Studio Extension

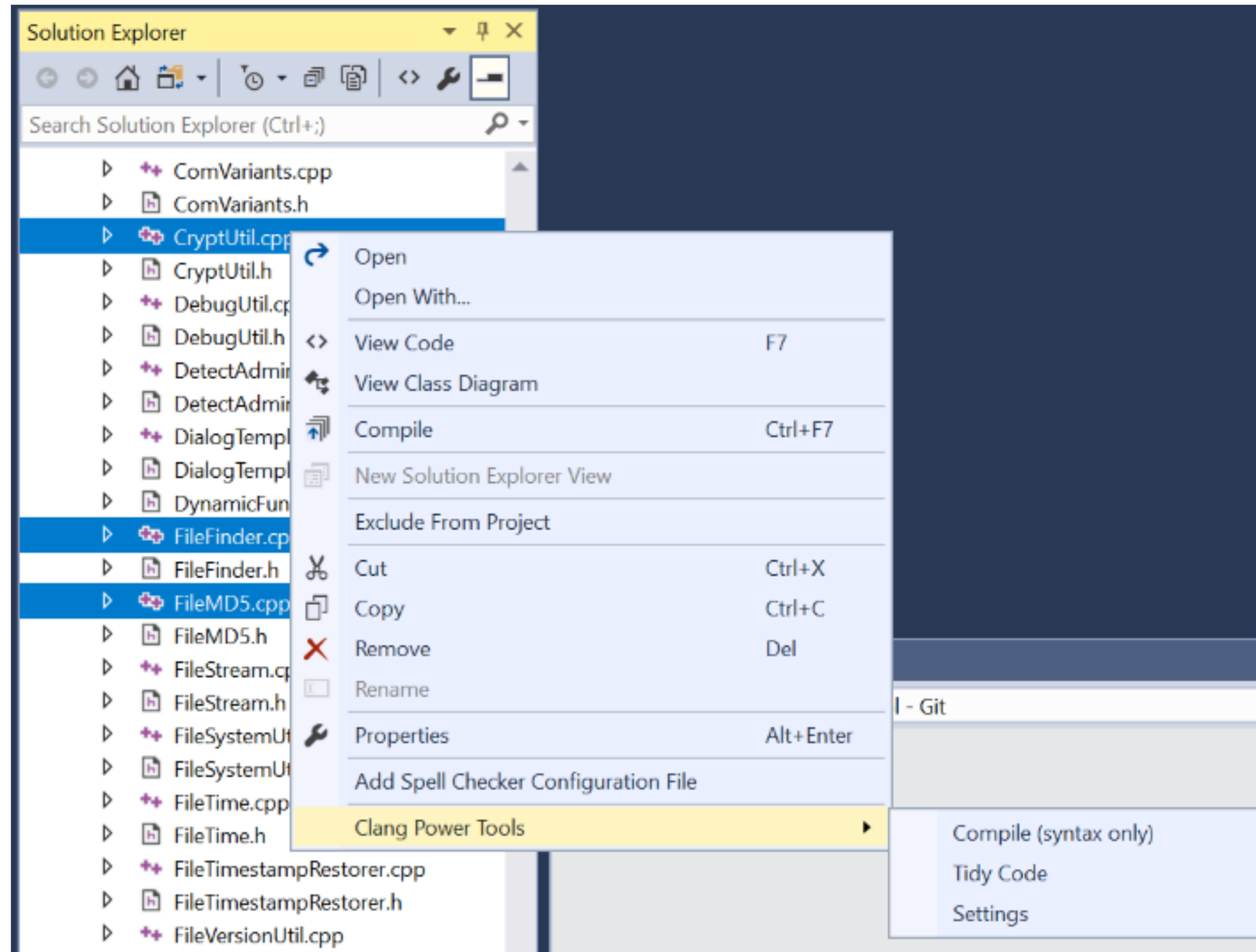
Run Clang Power Tools on an open source file





# Using The "Clang Power Tools" Visual Studio Extension

Run Clang Power Tools on selected files →



← Compile or Tidy code



# Using The "Clang Power Tools" Visual Studio Extension

```
StringProcessing.cpp x StringEncoding.cpp
Platform StringUtil IsRTL(const wstring & aString)
498 {
499     size_t textLength = aString.length();
500
501     CAutoVectorPtr<WORD> charsType;
502     charsType.Allocate(textLength);
503
504     Facet facet = DEFAULT_LOCALE;
505
506     // get type of each character from string
507     BOOL ret = ::GetStringTypeW(CT_CTYPE2, aString.c_str(), (int)textLength, charsType);
508     if (!ret)
509         return false;
510
511     for (size_t i = 0; i < textLength; i++)
512     {
513         // at least one char is RTL so we consider entire string as RTL
514         if (charsType[i] == C2_RIGHTTOLEFT)
```

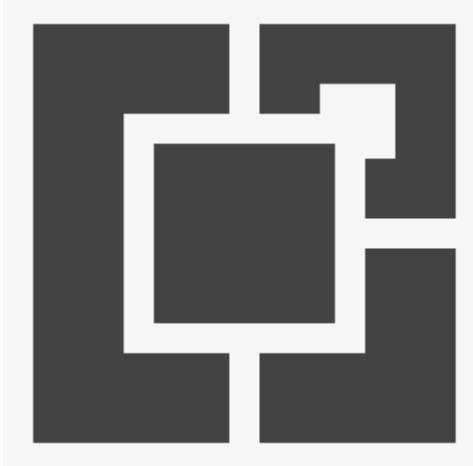
Output

Show output from: Clang Power Tools

```
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 'const class Facet'
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





# Using The "Clang Power Tools" Visual Studio Extension

```
StringProcessing.cpp  Platform  StringUtil  IsRTL(const wstring & aString)
491 // get type of each character from string
492 BOOL ret = ::GetStringTypeW(CT_CTYPE2, aString.c_str(), (int)textLength, charsType);
493
494 if (!ret)
495     return false;
496
497 for (size_t i = 0; i < textLength; i++)
498 {
499     // at least one char is RTL so we consider entire string as RTL
500     if (charsType[i] == C2_RIGHTTOLEFT)
501         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.NullDereference]
    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
```

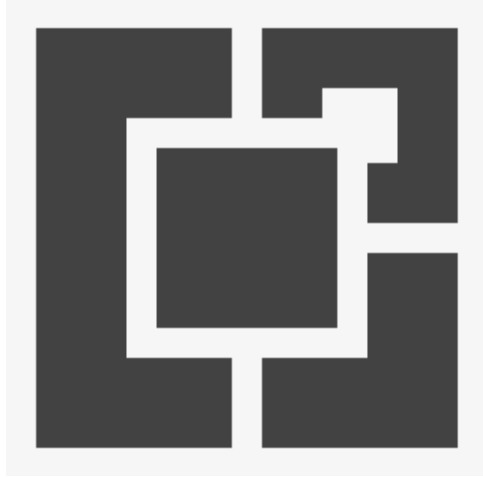
Error List Output Find Symbol Results

← clang-tidy : analyzer report

Eg.

[clang-analyzer-core.NullDereference]

# Where Can I Get It ?



Extension for Visual Studio 2015/2017

**Clang Power Tools** <sup>(Free)</sup> [marketplace.visualstudio.com](https://marketplace.visualstudio.com)

<https://github.com/Caphyon/clang-power-tools>



PowerShell scripts: `sample-clang-build.ps1 => clang-build.ps1`

<https://github.com/Caphyon/clang-power-tools/blob/master/ClangPowerTools/ClangPowerTools/clang-build.ps1>

<https://github.com/Caphyon/clang-power-tools/blob/master/ClangPowerTools/ClangPowerTools/sample-clang-build.ps1>

# Beyond clang-tidy



## LibTooling

- we wrote `custom tools` for our needs (project specific)
- fixed hundreds of member initializer lists with wrong order [-Wreorder]
- removed unused class private fields (references, pointers) [-Wunused-private-field]
- refactored some heavily used class constructors (changed mechanism for acquiring dependencies - interface refs)
- even more on the way...

# Roadmap

- **-Wextra** (a few remaining issues in our code)
- improve **Clang Power Tools** Visual Studio extension
- run more clang-tidy checks (fix more issues with **clang-analyzer-\***)
- re-run previous checks (for new code)
- use **libTooling** for more custom code transformations (project-specific)

# Questions

?