Contents

Download, install, and set up the Linux workload

9/21/2022 • 5 minutes to read • Edit Online

Linux projects are supported in Visual Studio 2017 and later. To see the documentation for these versions, set the Visual Studio **Version** selector control for this article to Visual Studio 2017 or Visual Studio 2019. It's found at the top of the table of contents on this page.

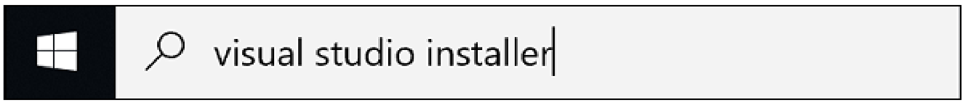
You can use the Visual Studio IDE on Windows to create, edit, and debug C++ projects that execute on a remote Linux system, virtual machine, or the Windows Subsystem for Linux.

You can work on your existing code base that uses CMake without having to convert it to a Visual Studio project. If your code base is cross-platform, you can target both Windows and Linux from within Visual Studio. For example, you can edit, build, and debug your code on Windows using Visual Studio. Then, quickly retarget the project for Linux to build and debug in a Linux environment. Linux header files are automatically copied to your local machine. Visual Studio uses them to provide full IntelliSense support (Statement Completion, Go to Definition, and so on).

For any of these scenarios, the **Linux development with C++** workload is required.

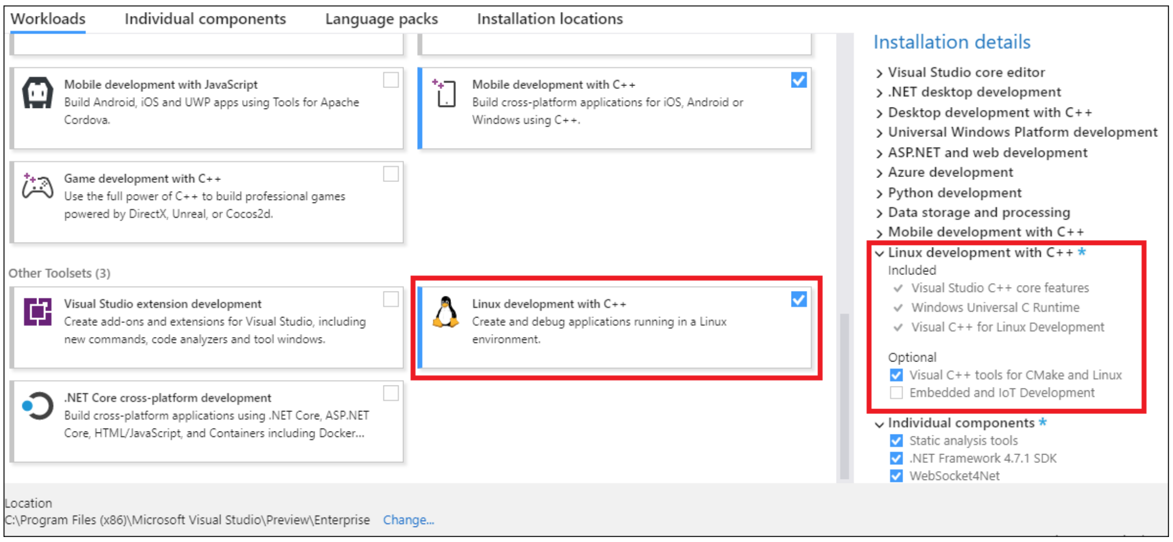
Visual Studio setup

1. Type "Visual Studio Installer" in the Windows search box:



visual studio installer

2. Look for the installer under the **Apps** results and double-click it. When the installer opens, choose **Modify**, and then click on the **Workloads** tab. Scroll down to **Other toolsets** and select the **Linux development with C++** workload.



Workloads | Individual components | Language packs | Installation locations

Mobile development with JavaScript
Build Android, iOS and UWP apps using Tools for Apache Cordova.

Mobile development with C++ ✓
Build cross-platform applications for iOS, Android or Windows using C++.

Game development with C++
Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Other Toolsets (3)

Visual Studio extension development
Create add-ons and extensions for Visual Studio, including new commands, code analyzers and tool windows.

Linux development with C++ ✓
Create and debug applications running in a Linux environment.

.NET Core cross-platform development
Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker...

Installation details

- Visual Studio core editor
- .NET desktop development
- Desktop development with C++
- Universal Windows Platform development
- ASP.NET and web development
- Azure development
- Python development
- Data storage and processing
- Mobile development with C++
- Linux development with C++ *
  - Included
    - ✓ Visual Studio C++ core features
    - ✓ Windows Universal C Runtime
    - ✓ Visual C++ for Linux Development
  - Optional
    - ☑ Visual C++ tools for CMake and Linux
    - ☐ Embedded and IoT Development
- Individual components *
  - ☑ Static analysis tools
  - ☑ .NET Framework 4.7.1 SDK
  - ☑ WebSocket4Net

Location
C:\Program Files (x86)\Microsoft Visual Studio\Preview\Enterprise Change...

3. If you're targeting IoT or embedded platforms, go to the **Installation details** pane on the right. Under **Linux development with C++**, expand **Optional Components**, and choose the components you need. CMake support for Linux is selected by default.

4. Click **Modify** to continue with the installation.

Options for creating a Linux environment

If you don't already have a Linux machine, you can create a Linux Virtual Machine on Azure. For more information, see Quickstart: Create a Linux virtual machine in the Azure portal.

On Windows 10 and later, you can install and target your favorite Linux distro on the Windows Subsystem for Linux (WSL). For more information, see Windows Subsystem for Linux Installation Guide for Windows 10. If you're unable to access the Windows Store, you can manually download the WSL distro packages. WSL is a convenient console environment, but it's not recommended for graphical applications.

Linux projects in Visual Studio require the following dependencies to be installed on your remote Linux system or WSL:

- **A compiler** - Visual Studio 2019 and later have full support for GCC and Clang.
- **gdb** - Visual Studio automatically launches gdb on the Linux system, and uses the front end of the Visual Studio debugger to provide a full-fidelity debugging experience on Linux.
- **rsync** and **zip** - the inclusion of rsync and zip allows Visual Studio to extract header files from your Linux system to the Windows filesystem for use by IntelliSense.
- **make**
- **openssh-server** (remote Linux systems only) - Visual Studio connects to remote Linux systems over a secure SSH connection.
- **CMake** (CMake projects only) - You can install Microsoft's statically linked CMake binaries for Linux.
- **ninja-build** (CMake projects only) - Ninja is the default generator for Linux and WSL configurations in Visual Studio 2019 version 16.6 or later.

The following commands assume you're using g++ instead of clang.

Linux projects in Visual Studio require the following dependencies to be installed on your remote Linux system or WSL:

- **gcc** - Visual Studio 2017 has full support for GCC.
- **gdb** - Visual Studio automatically launches gdb on the Linux system and uses the front end of the Visual Studio debugger to provide a full-fidelity debugging experience on Linux.
- **rsync** and **zip** - the inclusion of rsync and zip allows Visual Studio to extract header files from your Linux system to the Windows filesystem to use for IntelliSense.
- **make**
- **openssh-server** - Visual Studio connects to remote Linux systems over a secure SSH connection.
- **CMake** (CMake projects only) - You can install Microsoft's statically linked CMake binaries for Linux.

Linux setup: Ubuntu on WSL

When you're targeting WSL, there's no need to add a remote connection or configure SSH to build and debug. **zip** and **rsync** are required for automatic syncing of Linux headers with Visual Studio for Intellisense support. **ninja-build** is only required for CMake projects. If the required applications aren't already present, you can install them using this command:

```
sudo apt-get install g++ gdb make ninja-build rsync zip
```

Ubuntu on remote Linux systems

The target Linux system must have **openssh-server**, **g++**, **gdb**, and **make** installed. **ninja-build** is required for CMake projects only. The **ssh** daemon must be running. **zip** and **rsync** are required for automatic syncing of remote headers with your local machine for Intellisense support. If these applications aren't already present, you can install them as follows:

1. At a shell prompt on your Linux computer, run:

   ```
   sudo apt-get install openssh-server g++ gdb make ninja-build rsync zip
   ```

   You may be prompted for your root password to run the sudo command. If so, enter it and continue. Once complete, the required services and tools are installed.

2. Ensure the ssh service is running on your Linux computer by running:

   ```
   sudo service ssh start
   ```

   This command starts the service and runs it in the background, ready to accept connections.

Fedora on WSL

Fedora uses the **dnf** package installer. To download **g++**, **gdb**, **make**, **rsync**, **ninja-build**, and **zip**, run:

```
sudo dnf install gcc-g++ gdb rsync ninja-build make zip
```

zip and **rsync** are required for automatic syncing of Linux headers with Visual Studio for Intellisense support. **ninja-build** is only required for CMake projects.

Fedora on remote Linux systems

The target machine running Fedora uses the **dnf** package installer. To download **openssh-server**, **g++**, **gdb**, **make**, **ninja-build**, **rsync**, and **zip**, and restart the ssh daemon, follow these instructions. **ninja-build** is only required for CMake projects.

1. At a shell prompt on your Linux computer, run:

   ```
   sudo dnf install openssh-server gcc-g++ gdb ninja-build make rsync zip
   ```

   You may be prompted for your root password to run the sudo command. If so, enter it and continue. Once complete, the required services and tools are installed.

2. Ensure the ssh service is running on your Linux computer by running:

   ```
   sudo systemctl start sshd
   ```

   This command starts the service and runs it in the background, ready to accept connections.

Next Steps

You're now ready to create or open a Linux project and configure it to run on the target system. For more information, see:

- Create a new Linux MSBuild C++ project
- Configure a Linux CMake project

Connect to your target Linux system in Visual Studio

9/21/2022 • 10 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

You can configure a Linux project to target a remote machine or the Windows Subsystem for Linux (WSL). For both remote machines and for WSL, you need to set up a remote connection in Visual Studio 2017.

You can configure a Linux project to target a remote machine or the Windows Subsystem for Linux (WSL). For a remote machine, you need to set up a remote connection in Visual Studio. To connect to WSL, skip ahead to the Connect to WSL section.

When using a remote connection, Visual Studio builds C++ Linux projects on the remote machine. It doesn't matter if it's a physical machine, a VM in the cloud, or WSL. To build the project, Visual Studio copies the source code to your remote Linux computer. Then, the code gets compiled based on Visual Studio settings.

NOTE

Starting in Visual Studio 2019 version 16.5, Visual Studio supports secure, Federal Information Processing Standard (FIPS) 140-2 compliant cryptographic connections to Linux systems for remote development. To use a FIPS-compliant connection, follow the steps in Set up FIPS-compliant secure remote Linux development instead.

Set up the SSH server on the remote system

If `ssh` isn't already set up and running on your Linux system, follow these steps to install it. The examples in this article use Ubuntu 18.04 LTS with OpenSSH server version 7.6. However, the instructions should be the same for any distro using a moderately recent version of OpenSSH.

1. On the Linux system, install and start the OpenSSH server:

   ```
   sudo apt install openssh-server
   sudo service ssh start
   ```

2. If you'd like the ssh server to start automatically when the system boots, enable it using systemctl:

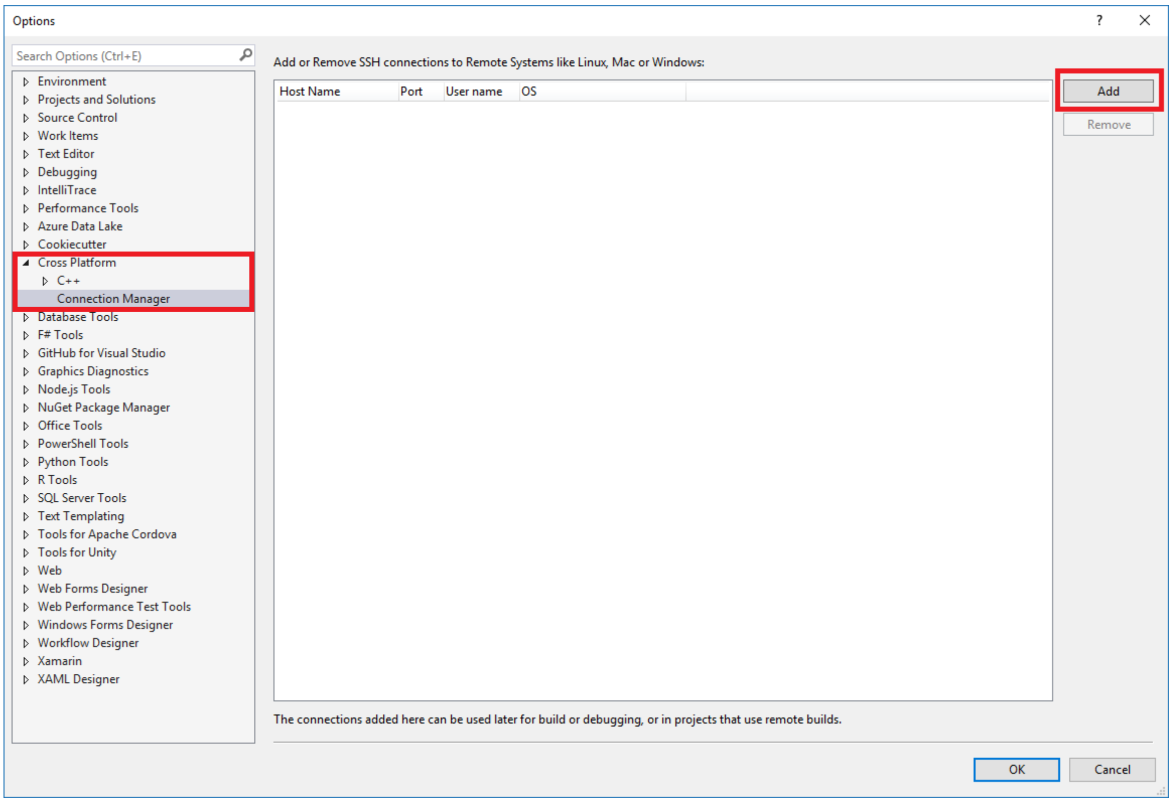
   ```
   sudo systemctl enable ssh
   ```

Set up the remote connection

1. In Visual Studio, choose **Tools** > **Options** on the menu bar to open the **Options** dialog. Then select **Cross Platform** > **Connection Manager** to open the Connection Manager dialog.

   If you haven't set up a connection in Visual Studio before, when you build your project for the first time, Visual Studio opens the Connection Manager dialog for you.

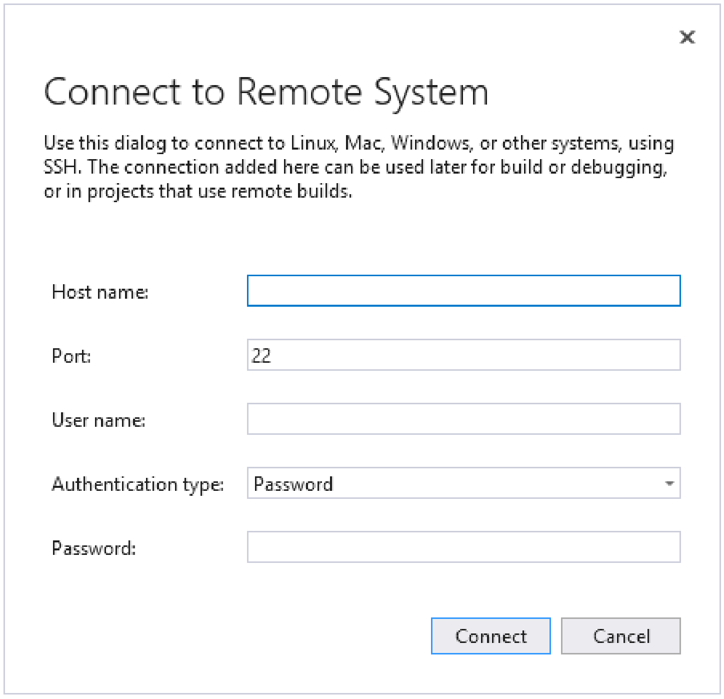
2. In the Connection Manager dialog, choose the **Add** button to add a new connection.



Add or Remove SSH connections to Remote Systems like Linux, Mac or Windows:

The connections added here can be used later for build or debugging, or in projects that use remote builds.

In either scenario, the **Connect to Remote System** window is displayed.



Use this dialog to connect to Linux, Mac, Windows, or other systems, using SSH. The connection added here can be used later for build or debugging, or in projects that use remote builds.

Port: 22

Authentication type: Password

3. Enter the following information:

ENTRY	DESCRIPTION
Host Name	Name or IP address of your target device
Port	Port that the SSH service is running on, typically 22
User name	User to authenticate as
Authentication type	Password and Private Key are both supported

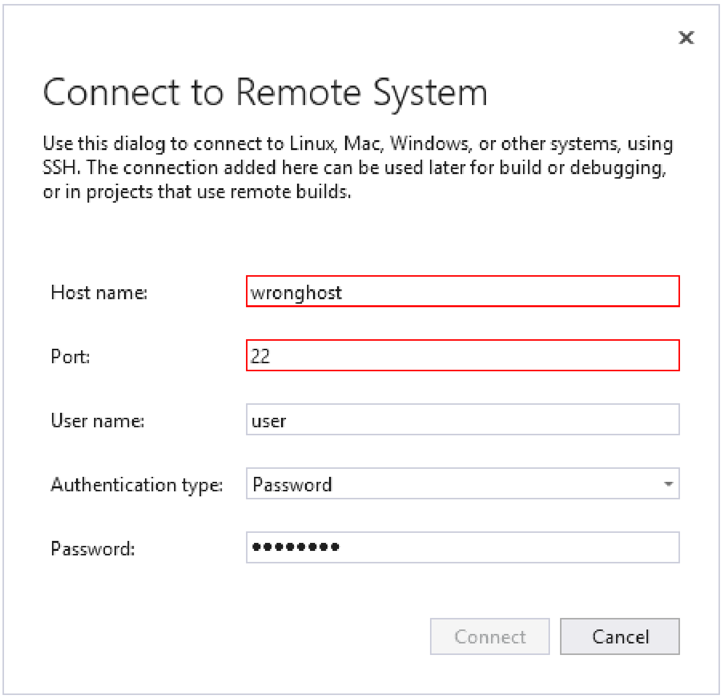
ENTRY	DESCRIPTION
Password	Password for the entered user name
Private key file	Private key file created for ssh connection
Passphrase	Passphrase used with private key selected above

You can use either a password or a key file and passphrase for authentication. For many development scenarios, password authentication is sufficient, but key files are more secure. If you already have a key pair, it's possible to reuse it. Currently Visual Studio only supports RSA and DSA keys for remote connections.

4. Choose the **Connect** button to attempt a connection to the remote computer.

   If the connection succeeds, Visual Studio configures IntelliSense to use the remote headers. For more information, see IntelliSense for headers on remote systems.

   If the connection fails, the entry boxes that need to be changed are outlined in red.



Connect to Remote System

Use this dialog to connect to Linux, Mac, Windows, or other systems, using SSH. The connection added here can be used later for build or debugging, or in projects that use remote builds.

Host name:	wronghost
Port:	22
User name:	user
Authentication type:	Password
Password:	●●●●●●●●●

   If you use key files for authentication, make sure the target machine's SSH server is running and configured properly.

   If you have trouble connecting to WSL on `localhost`, see Fix WSL `localhost` connection problems.

Host key verification

In Visual Studio version 16.10 or later, you'll be asked to verify the server's host key fingerprint whenever Visual Studio connects to a remote system for the first time. You may be familiar with this process if you've used the OpenSSH command-line client or PuTTY before. The fingerprint identifies the server. Visual Studio uses the fingerprint to ensure it's connecting to the intended and trusted server.

The first time Visual Studio establishes a new remote connection, you'll be asked to accept or deny the host key fingerprint presented by the server. Or, anytime there are changes to a cached fingerprint. You can also verify a fingerprint on demand: select a connection in the Connection Manager and choose **Verify**.

If you upgrade to Visual Studio 16.10 or later from an older version, it treats any existing remote connections as new connections. You'll be prompted to accept the host key fingerprint first. Then, Visual Studio establishes a connection and caches the accepted fingerprint.

You can also update remote connections from `ConnectionManager.exe` using the `update` argument.

Supported SSH algorithms

Starting in Visual Studio version 16.9, support for older, insecure SSH algorithms used to encrypt data and exchange keys, has been removed. Only the following algorithms are supported. They're supported for both client-to-server and server-to-client SSH communication:

ALGORITHM TYPE	SUPPORTED ALGORITHMS

Encryption	`aes128-cbc`
	`aes128-cbc`
	`aes192-cbc`
	`aes192-ctr`
	`aes256-cbc`
	`aes256-ctr`

HMAC	`hmac-sha2-256` `hmac-sha2-256`

Key exchange	`diffie-hellman-group14-sha256`
	`diffie-hellman-group16-sha512`
	`diffie-hellman-group-exchange-sha256`
	`ecdh-sha2-nistp256`
	`ecdh-sha2-nistp384`
	`ecdh-sha2-nistp521`

Host key	`ecdsa-sha2-nistp256`
	`ecdsa-sha2-nistp384`
	`ecdsa-sha2-nistp521`
	`ssh-dss`
	`ssh-rsa`

Configure the SSH server

First, a little background. You can't select the SSH algorithm to use from Visual Studio. Instead, the algorithm is determined during the initial handshake with the SSH server. Each side (client and server) provides a list of algorithms it supports, and then the first algorithm common to both is selected. The connection succeeds as long as there's at least one algorithm in common between Visual Studio and the server for encryption, HMAC, key exchange, and so on.

The Open SSH configuration file (*`sshd_config`*) doesn't configure which algorithm to use by default. The SSH server should use secure defaults when no algorithms are specified. Those defaults depend on the version and vendor of the SSH server. If Visual Studio doesn't support those defaults, you'll likely see an error like: "Could not connect to the remote system. No common client to server HMAC algorithm was found." The error may also appear if the SSH server is configured to use algorithms that Visual Studio doesn't support.

The default SSH server on most modern Linux distributions should work with Visual Studio. However, you may be running an older SSH server that's configured to use older, insecure algorithms. The following example explains how to update to more secure versions.

In the following example, the SSH server uses the insecure `hmac-sha1` algorithm, which isn't supported by Visual Studio 16.9. If the SSH server uses OpenSSH, you can edit the `/etc/ssh/sshd_config` file as shown below to enable more secure algorithms. For other SSH servers, refer to the server's documentation for how to configure them.

First, verify that the set of algorithms your server is using includes algorithms supported by Visual Studio. Run the following command on the remote machine to list the algorithms supported by the server:

```
ssh -Q cipher; ssh -Q mac; ssh -Q kex; ssh -Q key
```

The command produces output like:

```
3des-cbc
aes128-cbc
aes192-cbc
aes256-cbc
...
ecdsa-sha2-nistp521-cert-v01@openssh.com
sk-ecdsa-sha2-nistp256-cert-v01@openssh.com
```

The output lists all the encryption, HMAC, key exchange, and host key algorithms supported by your SSH server. If the list doesn't include algorithms supported by Visual Studio, then you'll need to upgrade your SSH server before proceeding.

You can enable algorithms supported by Visual Studio by editing `/etc/ssh/sshd_config` on the remote machine. The following examples show how to add various types of algorithms to that configuration file.

These examples can be added anywhere in `/etc/ssh/sshd_config`. Ensure that they're on their own lines.

After editing the file, restart the SSH server (`sudo service ssh restart` on Ubuntu) and attempt to connect again from Visual Studio.

Cipher example

Add: `Ciphers <algorithms to enable>`
For example: `Ciphers aes128-cbc,aes256-cbc`

HMAC example

Add: `MACs <algorithms to enable>`
For example: `MACs hmac-sha2-256,hmac-sha2-512`

Key exchange example

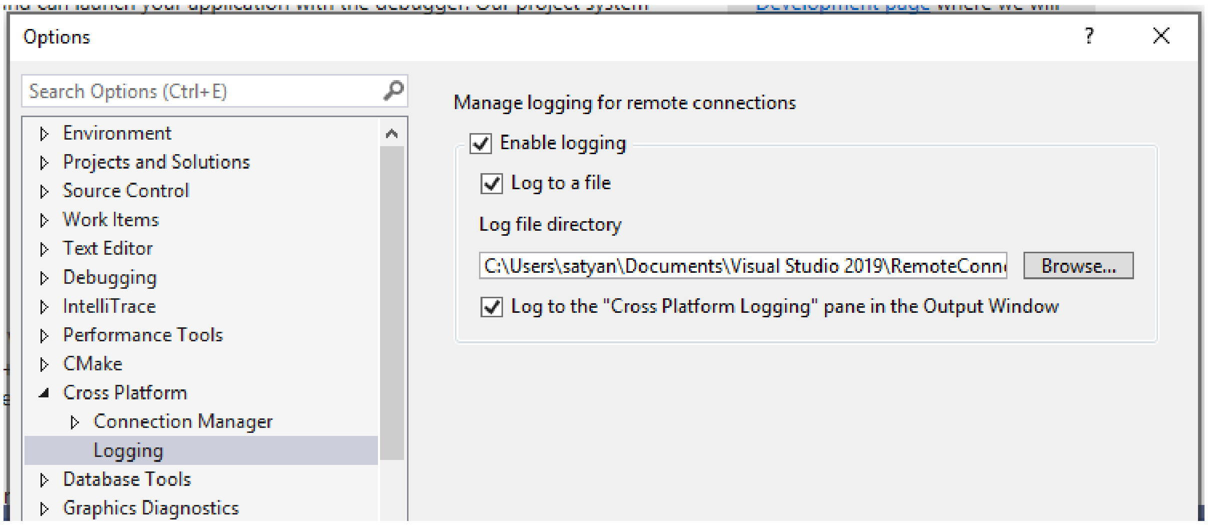
Add: `KexAlgorithms <algorithms to enable>`
For example: `KexAlgorithms ecdh-sha2-nistp256,ecdh-sha2-nistp384`

Host key example

Add: `HostKeyAlgorithms <algorithms to enable>`
For example: `HostKeyAlgorithms ssh-dss,ssh-rsa`

Logging for remote connections

You can enable logging to help troubleshoot connection problems. On the menu bar, select **Tools > Options**. In the **Options** dialog, select **Cross Platform > Logging**:


Options

Manage logging for remote connections

☑ Enable logging

☑ Log to a file

Log file directory

C:\Users\satyan\Documents\Visual Studio 2019\RemoteConn

☑ Log to the "Cross Platform Logging" pane in the Output Window

Logs include connections, all commands sent to the remote machine (their text, exit code and execution time), and all output from Visual Studio to the shell. Logging works for any cross-platform CMake project or MSBuild-based Linux project in Visual Studio.

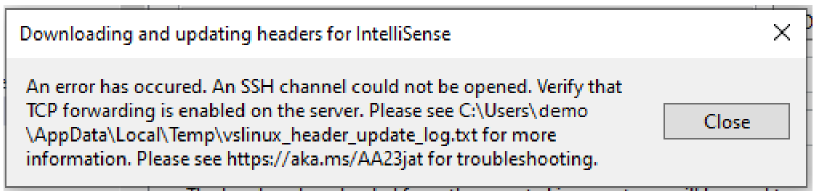
You can configure the output to go to a file or to the **Cross Platform Logging** pane in the Output window. For MSBuild-based Linux projects, MSBuild commands sent to the remote machine aren't routed to the **Output Window** because they're emitted out-of-process. Instead, they're logged to a file, with a prefix of "msbuild_".

Command-line utility for the Connection Manager

Visual Studio 2019 version 16.5 or later: `ConnectionManager.exe` is a command-line utility to manage remote development connections outside of Visual Studio. It's useful for tasks such as provisioning a new development machine. Or, you can use it to set up Visual Studio for continuous integration. For examples and a complete reference to the ConnectionManager command, see ConnectionManager reference.

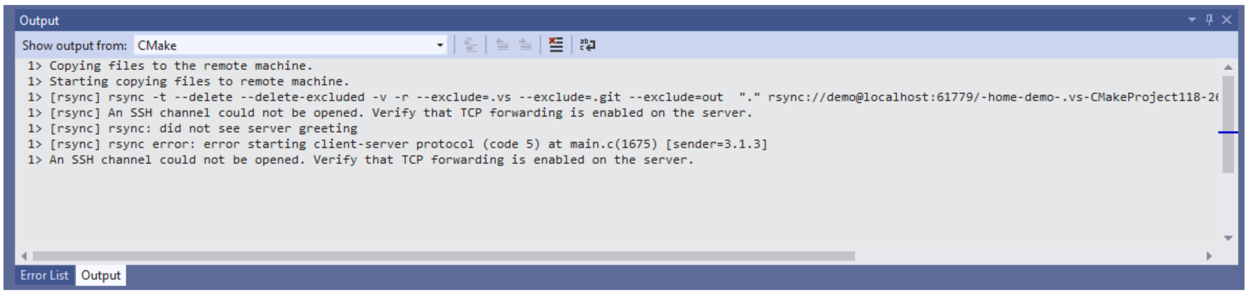
TCP Port Forwarding

The `rsync` command is used by both MSBuild-based Linux projects and CMake projects to copy headers from your remote system to Windows for use by IntelliSense. When you can't enable TCP port forwarding, disable the automatic download of remote headers. To disable it, use **Tools > Options > Cross Platform > Connection Manager > Remote Headers IntelliSense Manager**. If the remote system doesn't have TCP port forwarding enabled, you'll see this error when the download of remote headers for IntelliSense begins:


Downloading and updating headers for IntelliSense

An error has occured. An SSH channel could not be opened. Verify that TCP forwarding is enabled on the server. Please see C:\Users\demo \AppData\Local\Temp\vslinux_header_update_log.txt for more information. Please see https://aka.ms/AA23jat for troubleshooting.

`rsync` is also used by Visual Studio's CMake support to copy source files to the remote system. If you can't enable TCP port forwarding, you can use `sftp` as your remote copy sources method. `sftp` is often slower than `rsync`, but doesn't have a dependency on TCP port forwarding. You can manage your remote copy sources method with the `remoteCopySourcesMethod` property in the CMake Settings Editor. If TCP port forwarding is disabled on your remote system, you'll see an error in the CMake output window the first time it invokes `rsync`.



```
1> Copying files to the remote machine.
1> Starting copying files to remote machine.
1> [rsync] rsync -t --delete --delete-excluded -v -r --exclude=.vs --exclude=.git --exclude=out  "." rsync://demo@localhost:61779/-home-demo-.vs-CMakeProject118-2
1> [rsync] An SSH channel could not be opened. Verify that TCP forwarding is enabled on the server.
1> [rsync] rsync: did not see server greeting
1> [rsync] rsync error: error starting client-server protocol (code 5) at main.c(1675) [sender=3.1.3]
1> An SSH channel could not be opened. Verify that TCP forwarding is enabled on the server.
```

`gdbserver` can be used for debugging on embedded devices. If you can't enable TCP port forwarding, then you must use `gdb` for all remote debugging scenarios. `gdb` is used by default when debugging projects on a remote system.

Visual Studio's Linux support has a dependency on TCP port forwarding. Both `rsync` and `gdbserver` are affected if TCP port forwarding is disabled on your remote system. If this dependency impacts you, vote for this suggestion ticket on Developer Community.

Connect to WSL

In Visual Studio 2017, you use the same steps to connect to WSL as you use for a remote Linux machine. Use `localhost` for the **Host Name**.

Starting in Visual Studio 2019 version 16.1, Visual Studio has native support for using C++ with the Windows Subsystem for Linux (WSL). That means you can build and debug on your local WSL installation directly. You no longer need to add a remote connection or configure SSH. You can find details on how to install WSL here.

To configure your WSL installation to work with Visual Studio, you need the following tools installed: `gcc` or `clang`, `gdb`, `make`, `ninja-build` (only required for CMake projects using Visual Studio 2019 version 16.6 or later), `rsync`, and `zip`. You can install them on distros that use `apt` by using this command, which also installs the g++ compiler:

```
sudo apt install g++ gdb make ninja-build rsync zip
```

Fix WSL `localhost` connection problems

When connecting to Windows Subsystem for Linux (WSL) on `localhost`, you may run into a conflict with the Windows `ssh` client on port 22. In WSL, change the port that `ssh` expects requests from to 23 in `/etc/ssh/sshd_config`:

```
Port 23
```

If you're connecting using a password, ensure that the following is set in `/etc/ssh/sshd_config`:

```
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
```

After making these changes, restart the SSH server (`sudo service ssh restart` on Ubuntu).

Then retry your connection to `localhost` using port 23.

For more information, see Download, install, and set up the Linux workload.

To configure an MSBuild project for WSL, see Configure a Linux project. To configure a CMake project for WSL, see Configure a Linux CMake project. To follow step-by-step instructions for creating a simple console application with WSL, check out this introductory blog post on C++ with Visual Studio 2019 and the Windows

Subsystem for Linux (WSL).

See Also

Configure a Linux project
Configure a Linux CMake project
Deploy, run, and debug your Linux project
Configure CMake debugging sessions

Set up FIPS-compliant secure remote Linux development

9/21/2022 • 5 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later. FIPS-compliant secure remote Linux development is available in Visual Studio 2019 version 16.5 and later.

Federal Information Processing Standard (FIPS) Publication 140-2 is a U.S. government standard for cryptographic modules. Implementations of the standard are validated by NIST. Windows has validated support for FIPS-compliant cryptographic modules. In Visual Studio 2019 version 16.5 and later, you can use a secure, FIPS-compliant cryptographic connection to your Linux system for remote development.

Here's how to set up a secure, FIPS-compliant connection between Visual Studio and your remote Linux system. This guide is applicable when you build CMake or MSBuild Linux projects in Visual Studio. This article is the FIPS-compliant version of the connection instructions in Connect to your remote Linux computer.

Prepare a FIPS-compliant connection

Some preparation is required to use a FIPS-compliant, cryptographically secure ssh connection between Visual Studio and your remote Linux system. For FIPS-140-2 compliance, Visual Studio only supports RSA keys.

The examples in this article use Ubuntu 18.04 LTS with OpenSSH server version 7.6. However, the instructions should be the same for any distro using a moderately recent version of OpenSSH.

To set up the SSH server on the remote system

1. On the Linux system, install and start the OpenSSH server:

```
sudo apt install openssh-server
sudo service ssh start
```

2. If you'd like the `ssh` server to start automatically when the system boots, enable it using `systemctl`:

```
sudo systemctl enable ssh
```

3. Open `/etc/ssh/sshd_config` as root. Edit (or add, if they don't exist) the following lines:

```
Ciphers aes256-cbc,aes192-cbc,aes128-cbc,3des-cbc
HostKeyAlgorithms ssh-rsa
KexAlgorithms diffie-hellman-group-exchange-sha256,diffie-hellman-group-exchange-sha1,diffie-hellman-group14-sha1
MACs hmac-sha2-256,hmac-sha1
```

> **NOTE**
> `ssh-rsa` is the only FIPS compliant host key algorithm VS supports. The `aes*-ctr` algorithms are also FIPS compliant, but the implementation in Visual Studio isn't approved. The `ecdh-*` key exchange algorithms are FIPS compliant, but Visual Studio doesn't support them.

You're not limited to these options. You can configure `ssh` to use other ciphers, host key algorithms, and so on. Some other relevant security options you may want to consider are `PermitRootLogin`, `PasswordAuthentication`, and `PermitEmptyPasswords`. For more information, see the `man` page for `sshd_config` or the article SSH Server Configuration.

4. After saving and closing `sshd_config`, restart the ssh server to apply the new configuration:

   ```
   sudo service ssh restart
   ```

Next, you'll create an RSA key pair on your Windows computer. Then you'll copy the public key to the remote Linux system for use by `ssh`.

To create and use an RSA key file

1. On the Windows machine, generate a public/private RSA key pair by using this command:

   ```
   ssh-keygen -t rsa -b 4096 -m PEM
   ```

   The command creates a public key and a private key. By default, the keys are saved to *`%USERPROFILE%\.ssh\id_rsa`* and *`%USERPROFILE%\\.ssh\\id_rsa.pub`*. (In PowerShell, use `$env:USERPROFILE` instead of the cmd macro `%USERPROFILE%`) If you change the key name, use the changed name in the steps that follow. We recommend you use a passphrase for increased security.

2. From Windows, copy the public key to the Linux machine:

   ```
   scp %USERPROFILE%\.ssh\id_rsa.pub user@hostname:
   ```

3. On the Linux system, add the key to the list of authorized keys, and ensure the file has the correct permissions:

   ```
   cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
   chmod 600 ~/.ssh/authorized_keys
   ```

4. Now, you can test to see if the new key works in `ssh`. Use it to sign in from Windows:

   ```
   ssh -i %USERPROFILE%\.ssh\id_rsa user@hostname
   ```

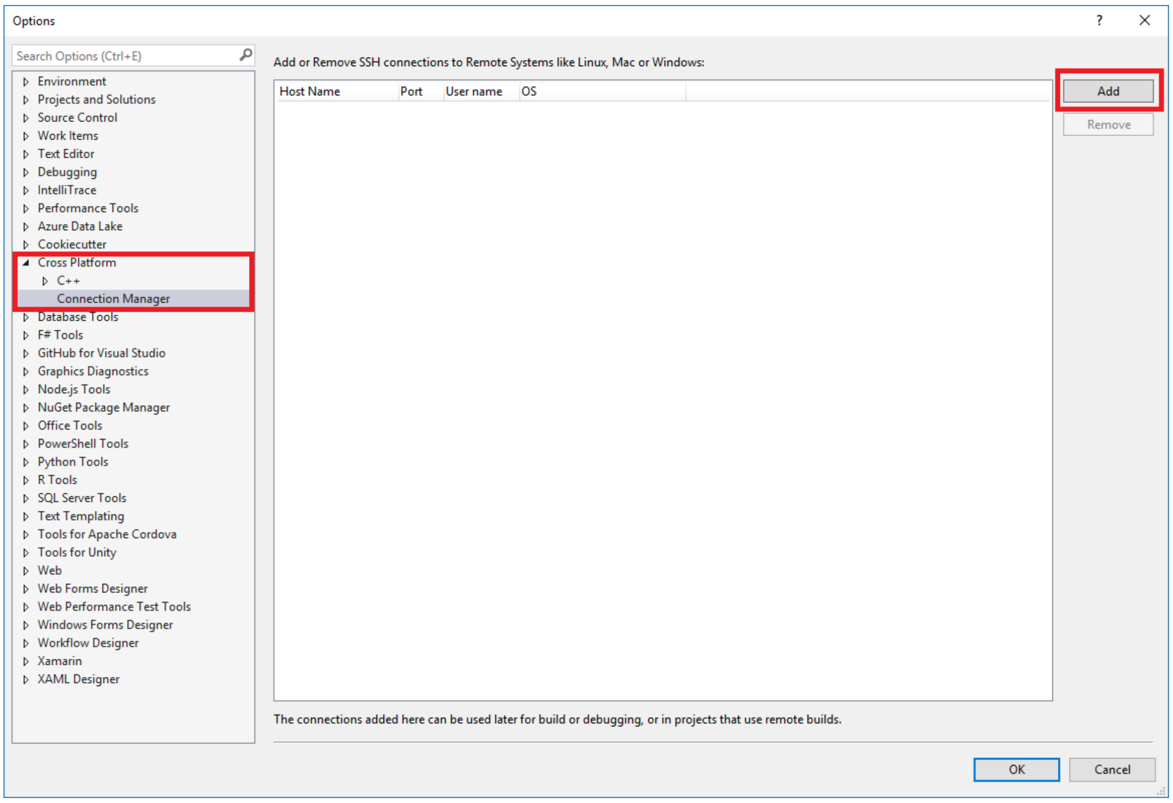
You've successfully set up `ssh`, created and deployed encryption keys, and tested your connection. Now you're ready to set up the Visual Studio connection.

Connect to the remote system in Visual Studio

1. In Visual Studio, choose **Tools > Options** on the menu bar to open the **Options** dialog. Then select **Cross Platform > Connection Manager** to open the Connection Manager dialog.

   If you haven't set up a connection in Visual Studio before, when you build your project for the first time, Visual Studio opens the Connection Manager dialog for you.

2. In the Connection Manager dialog, choose the **Add** button to add a new connection.


Options

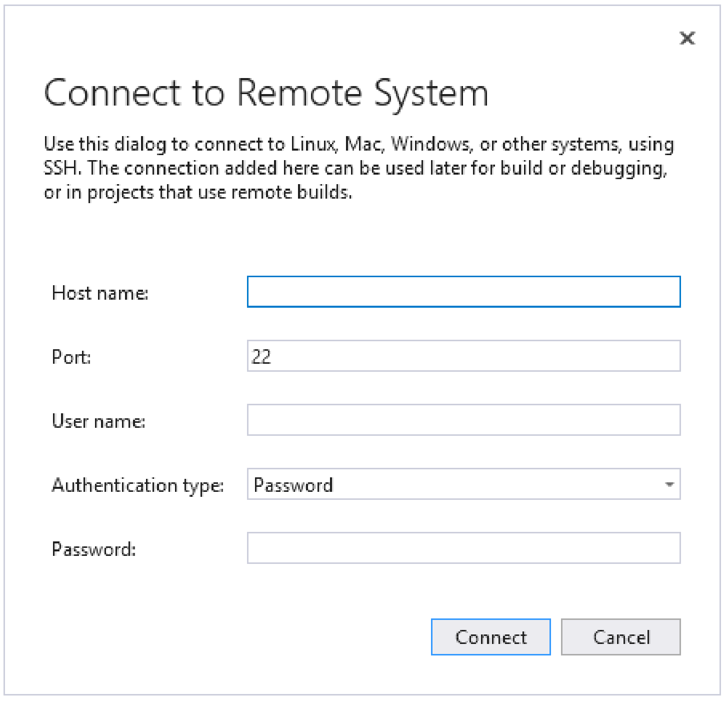
Search Options (Ctrl+E)

Add or Remove SSH connections to Remote Systems like Linux, Mac or Windows:

Host Name	Port	User name	OS

The connections added here can be used later for build or debugging, or in projects that use remote builds.

The **Connect to Remote System** window is displayed.


Connect to Remote System

Use this dialog to connect to Linux, Mac, Windows, or other systems, using SSH. The connection added here can be used later for build or debugging, or in projects that use remote builds.

Host name:	
Port:	22
User name:	
Authentication type:	Password
Password:	

3. In the **Connect to Remote System** dialog, enter the connection details of your remote machine.

ENTRY	DESCRIPTION
Host Name	Name or IP address of your target device
Port	Port that the SSH service is running on, typically 22
User name	User to authenticate as
Authentication type	Choose **Private Key** for a FIPS-compliant connection

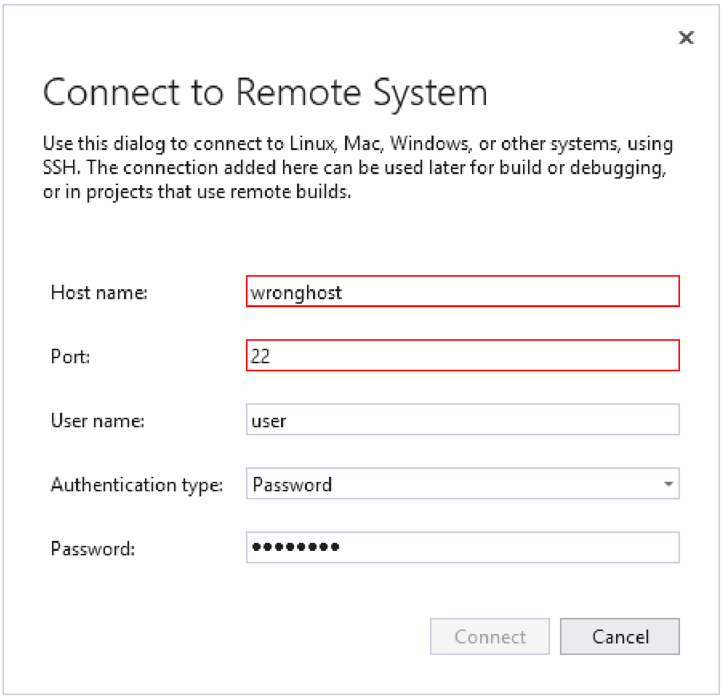
ENTRY	DESCRIPTION
Private key file	Private key file created for ssh connection
Passphrase	Passphrase used with private key selected above

Change the authentication type to **Private Key**. Enter the path to your private key in the **Private key file** field. You can use the **Browse** button to navigate to your private key file instead. Then, enter the passphrase used to encrypt your private key file in the **Passphrase** field.

4. Choose the **Connect** button to attempt a connection to the remote computer.

   If the connection succeeds, Visual Studio configures IntelliSense to use the remote headers. For more information, see IntelliSense for headers on remote systems.

   If the connection fails, the entry boxes that need to be changed are outlined in red.


Connect to Remote System

Use this dialog to connect to Linux, Mac, Windows, or other systems, using SSH. The connection added here can be used later for build or debugging, or in projects that use remote builds.

Host name:	wronghost
Port:	22
User name:	user
Authentication type:	Password
Password:	•••••••••

   For more information on troubleshooting your connection, see Connect to your remote Linux computer.

Command-line utility for the Connection Manager

Visual Studio 2019 version 16.5 or later: `ConnectionManager.exe` is a command-line utility to manage remote development connections outside of Visual Studio. It's useful for tasks such as provisioning a new development machine. Or, you can use it to set up Visual Studio for continuous integration. For examples and a complete reference to the ConnectionManager command, see ConnectionManager reference.

Optional: Enable or disable FIPS mode

It's possible to enable FIPS mode globally in Windows.

1. To enable FIPS mode, press **Windows+R** to open the **Run** dialog, and then run `gpedit.msc`.
2. Expand **Local Computer Policy > Computer Configuration > Windows Settings > Security Settings > Local Policies** and select **Security Options**.
3. Under **Policy**, select **System cryptography: Use FIPS-compliant algorithms for encryption, hashing, and signing**, and then press **Enter** to open its dialog box.

4. In the **Local Security Setting** tab, select **Enabled** or **Disabled**, and then choose **OK** to save your changes.

> **WARNING**
>
> Enabling FIPS mode may cause some applications to break or behave unexpectedly. For more information, see the blog post Why We're Not Recommending "FIPS mode" Anymore.

Additional resources

Microsoft documentation on FIPS 140 validation

FIPS 140-2: Security Requirements for Cryptographic Modules (from NIST)

Cryptographic Algorithm Validation Program: Validation Notes (from NIST)

Microsoft blog post on Why We're Not Recommending "FIPS mode" Anymore

SSH Server Configuration

See Also

Configure a Linux project
Configure a Linux CMake project
Connect to your remote Linux computer
Deploy, run, and debug your Linux project
Configure CMake debugging sessions

ConnectionManager reference

9/21/2022 • 4 minutes to read • Edit Online

ConnectionManager.exe is available in Visual Studio 2019 version 16.5 and later.

ConnectionManager.exe is a command-line utility to manage remote development connections outside of Visual Studio. It's useful for tasks such as provisioning a new development machine. Or, use it to set up Visual Studio for continuous integration. You can use it in a Developer Command Prompt window. For more information about the Developer Command Prompt, see Use the Microsoft C++ toolset from the command line.

ConnectionManager.exe is available in Visual Studio 2019 version 16.5 and later. It's part of the **Linux development with C++** workload in the Visual Studio Installer. It's also installed automatically when you choose the **Connection Manager** component in the installer. It's installed in *%VCIDEInstallDir%\Linux\bin\ConnectionManagerExe\ConnectionManager.exe*.

The functionality of ConnectionManager.exe is also available in Visual Studio. To manage remote development connections in the IDE, on the menu bar, choose **Tools** > **Options** to open the Options dialog. In the Options dialog, select **Cross Platform** > **Connection Manager**.

Syntax

`ConnectionManager.exe` *command* [*arguments*] [*options*]

Commands and arguments

- `add` *user@host* [`--port` *port*] [`--password` *password*] [`--privatekey` *privatekey_file*]

  Authenticates and adds a new connection. By default, it uses port 22 and password authentication. (You'll be prompted to enter a password.) Use both - `-password` and `--privatekey` to specify a password for a private key.

- `clean`

  Deletes header cache for connections that no longer exist.

- `help`

  Displays a help screen.

- `list` [`--properties`]

  Displays information, IDs, and properties of all stored connections.
  For examples, see Commonly used properties.

- `modify` [*default* | *connection_id* | *user@host* [`--port` *port*]] [`--property` *key=value*]

  Defines or modifies a property on a connection.
  If *value* is empty, then the property *key* is deleted.
  If authentication fails, no changes will be made.
  If no connection is specified (what is meant by *default*, above), the user's default remote connection is used.

- `remove` [*connection_id* | *user@host* [`--port` *port*]]

  Removes a connection. If no arguments are specified, you're prompted to specify which connection to remove.

- `remove-all`

  Removes all stored connections.

- `update` [*default* | *all* | *connection_id* | *user@host* [`--port` *port*]] [`--previous`] [`--fingerprint`]

  Added in Visual Studio 16.10. Updates the host key fingerprint of the specified connection(s).

- `version`

  Displays version information.

Options

- `--file` *filename*

  Read connection information from the provided *filename*.

- `--fingerprint`

  The host key fingerprint presented by the server. Use this option with `list` to view a connection's fingerprint.

- `-i`

  Same as `--privatekey`.

- `-n`, `--dry-run`

  Does a dry run of the command.

- `--no-prompt`

  Fail instead of prompt, when appropriate.

- `--no-telemetry`

  Disable sending usage data back to Microsoft. Usage data is collected and sent back to Microsoft unless the `--no-telemetry` flag is passed.

- `--no-verify`

  Add or modify a connection without authentication.

- `--p`

  Same as `--password`.

- `--previous`

  Indicates that the connection(s) will be read from the previous version of connection manager, updated, and written to the new version.

- `-q`, `--quiet`

  Prevents output to `stdout` or `stderr`.

Examples

This command adds a connection for a user named "user" on localhost. The connection uses a key file for authentication, found in *%USERPROFILE%.ssh\id_rsa*.

```
ConnectionManager.exe add user@127.0.0.1 --privatekey "%USERPROFILE%\.ssh\id_rsa"
```

This command removes the connection that has ID 1975957870 from the list of connections.

```
ConnectionManager.exe remove 1975957870
```

Commonly used properties

PROPERTY	DESCRIPTION
authentication type	The type of authentication used for the connection such as: `"password"`, `"privatekey"`. To create a connection with the authentication type set to `"privatekey"`: `ConnectionManager.exe add user@127.0.0.1 --privatekey "%USERPROFILE%\.ssh\id_rsa"`
`default`	A boolean indicating whether this is the default connection. The default connection is used when there's more than one connection available and the one to use isn't specified. To set the specified connection to be the default connection: `ConnectionManager.exe modify -21212121 --property default=true`
`host`	The name or IP address of the remote computer. To change the host for the specified connection to another machine, in this case, local host: `ConnectionManager.exe modify -21212121 --property host=127.0.0.1`
`isWsl`	Returns true if the remote session is running Windows Subsystem for Linux.
`password`	The password for the connection. Change the password for the specified connection with: `ConnectionManager.exe modify -21212121 --property password="xyz"`
`platform`	The platform of the remote computer such as `"ARM"`, `"ARM64"`, `"PPC"`, `"PPC64"`, `"x64"`, `"x86"`.
`port`	The port used for the connection. Change the port for the specified connection: `ConnectionManager.exe modify -21212121 --property port=22`
`shell`	The preferred shell to use on the remote system. Supported shells are `sh, csh, bash, tcsh, ksh, zsh, dash` To set the preferred shell to be zsh for the remote machine on the specified connection: `ConnectionManager.exe modify -21212121 --property shell=zsh` If the shell found on the Linux system isn't supported, then `sh` is used for all commands.
`systemID`	The remote system type, such as `"OSX"`, `"Ubuntu"`.

PROPERTY	DESCRIPTION
`timeout`	The connection timeout in milliseconds. Change the timeout for the specified connection with: `ConnectionManager.exe modify -21212121 --property timeout=100`
`username`	The name of the user logged into the remote computer. To add a connection for a user named `"user"` on localhost: `ConnectionManager.exe add user@127.0.0.1`

See also

Connect to your target Linux system in Visual Studio

Create a Linux MSBuild C++ project in Visual Studio

9/21/2022 • 2 minutes to read • Edit Online

Linux projects are available in Visual Studio 2017 and later.

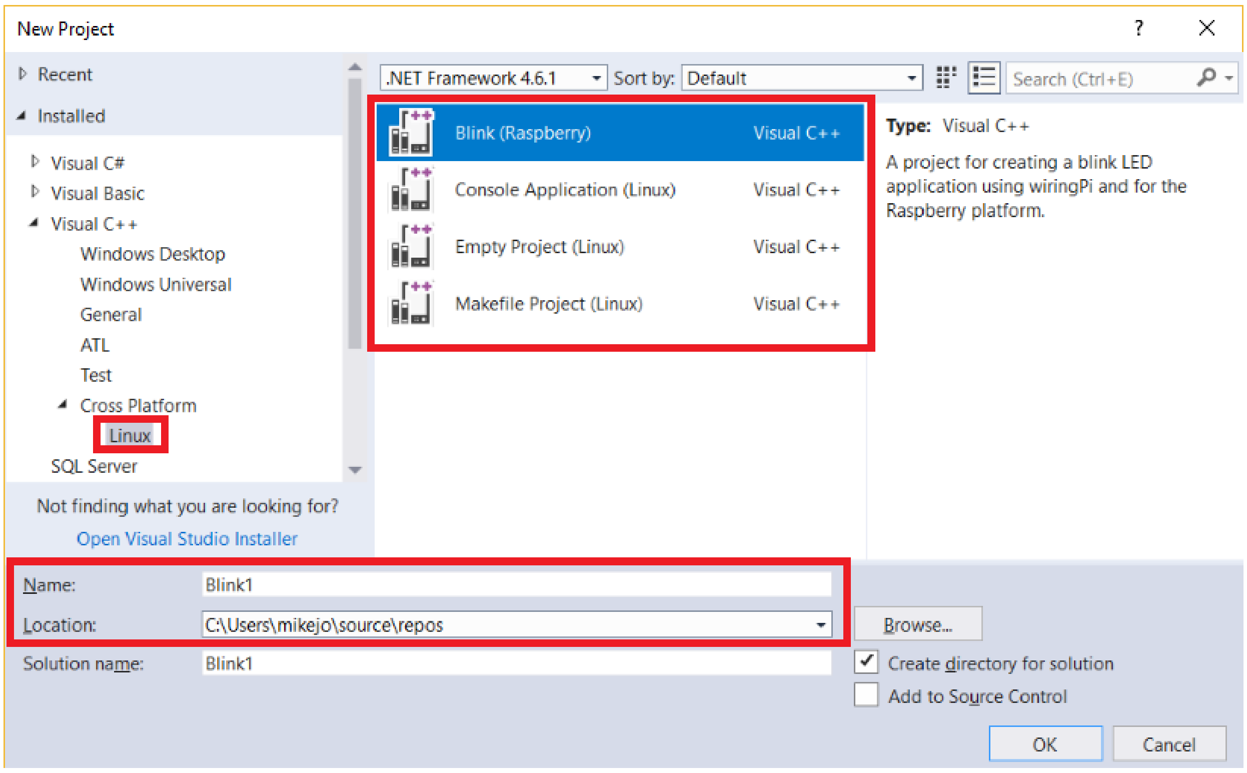
First, make sure you have the **Linux Development Workload** for Visual Studio installed. For more information, see Download, install, and setup the Linux workload.

For cross-platform compilation, we recommend using CMake. CMake support is more complete in Visual Studio 2019. If CMake isn't an option, and you have an existing Windows Visual Studio solution that you would like to extend to compile for Linux, you can add a Visual Studio Linux project to the Windows solution, along with a **Shared Items** project. Put the code that is shared between both platforms in the Shared Items project, and add a reference to that project from the Windows and Linux projects.

To create a new Linux project

To create a new Linux project in Visual Studio 2017, follow these steps:

1. Select **File** > **New Project** in Visual Studio, or press **Ctrl** + **Shift** + **N**.
2. Select the **Visual C++** > **Cross Platform** > **Linux** node, and then select the project type to create. Enter a **Name** and **Location**, and choose **OK**.


New Project
Recent
Installed
Visual C#
Visual Basic
Visual C++
Windows Desktop
Windows Universal
General
ATL
Test
Cross Platform
Linux
SQL Server
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.1
Sort by: Default
Search (Ctrl+E)
Blink (Raspberry) Visual C++
Console Application (Linux) Visual C++
Empty Project (Linux) Visual C++
Makefile Project (Linux) Visual C++
Browse...
Create directory for solution
Add to Source Control
OK
Cancel

Type: Visual C++

A project for creating a blink LED application using wiringPi and for the Raspberry platform.

Name: Blink1

Location: C:\Users\mikejo\source\repos

Solution name: Blink1

PROJECT TYPE	DESCRIPTION
Blink (Raspberry)	Project targeted for a Raspberry Pi device, with sample code that blinks an LED

PROJECT TYPE	DESCRIPTION
Console Application (Linux)	Project targeted for any Linux computer, with sample code that outputs text to the console
Empty Project (Linux)	Project targeted for any Linux computer, with no sample code
Makefile Project (Linux)	Project targeted for any Linux computer, built using a standard Makefile build system

First, make sure you have the **Linux Development Workload** for Visual Studio installed. For more information, see Download, install, and set up the Linux workload.

When you create a new C++ project for Linux in Visual Studio, you can choose to create a Visual Studio project or a CMake project. This article describes how to create a Visual Studio project. In general, for new projects that might include open-source code or you intend to compile for cross-platform development, we recommend you use CMake with Visual Studio. With a CMake project, you can build and debug the same project on both Windows and Linux. For more information, see Create and configure a Linux CMake Project.

If you have an existing Windows Visual Studio solution that you would like to extend to compile for Linux, and CMake isn't an option, then you can add a Visual Studio Linux project to the Windows solution, along with a **Shared Items** project. Put the code that is shared between both platforms in the Shared Items project, and add a reference to that project from the Windows and Linux projects.

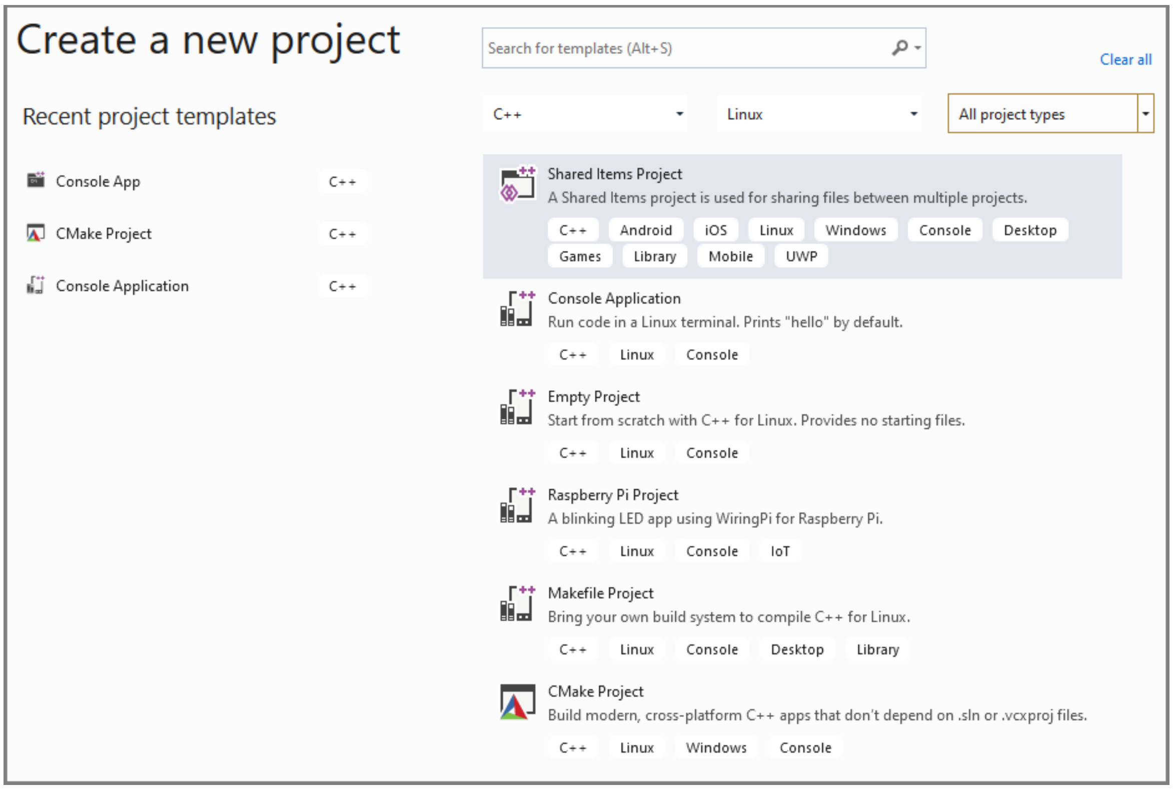
Create a new Linux project

To create a new Linux project in Visual Studio, follow these steps:

1. Select **File** > **New Project** in Visual Studio, or press **Ctrl** + **Shift** + **N**. The Create a new project dialog appears.

2. In the **Search for templates** textbox, enter **Linux** to list the available templates for Linux projects.

3. Select the project type to create, for example **Console Application**, and then choose **Next**. Enter a **Name** and **Location**, and choose **Create**.


Create a new project

Search for templates (Alt+S)

Clear all

C++ | Linux | All project types

Recent project templates

- Console App C++
- CMake Project C++
- Console Application C++

Shared Items Project
A Shared Items project is used for sharing files between multiple projects.
C++ Android iOS Linux Windows Console Desktop Games Library Mobile UWP

Console Application
Run code in a Linux terminal. Prints "hello" by default.
C++ Linux Console

Empty Project
Start from scratch with C++ for Linux. Provides no starting files.
C++ Linux Console

Raspberry Pi Project
A blinking LED app using WiringPi for Raspberry Pi.
C++ Linux Console IoT

Makefile Project
Bring your own build system to compile C++ for Linux.
C++ Linux Console Desktop Library

CMake Project
Build modern, cross-platform C++ apps that don't depend on .sln or .vcxproj files.
C++ Linux Windows Console

PROJECT TYPE	DESCRIPTION
Raspberry Pi project	Project targeted for a Raspberry Pi device, with sample code that blinks an LED
Console Application	Project targeted for any Linux computer, with sample code that outputs text to the console
Empty Project	Project targeted for any Linux computer, with no sample code
Makefile Project	Project targeted for any Linux computer, built using a standard Makefile build system
CMake Project	Project targeted for any Linux computer, built using the CMake build system

Next steps

Configure a Linux MSBuild project

Configure a Linux MSBuild C++ project in Visual Studio

9/21/2022 • 5 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

This topic describes how to configure a MSBuild-based Linux project as described in Create a Linux MSBuild C++ project in Visual Studio. For Linux CMake projects, see Configure a Linux CMake project.

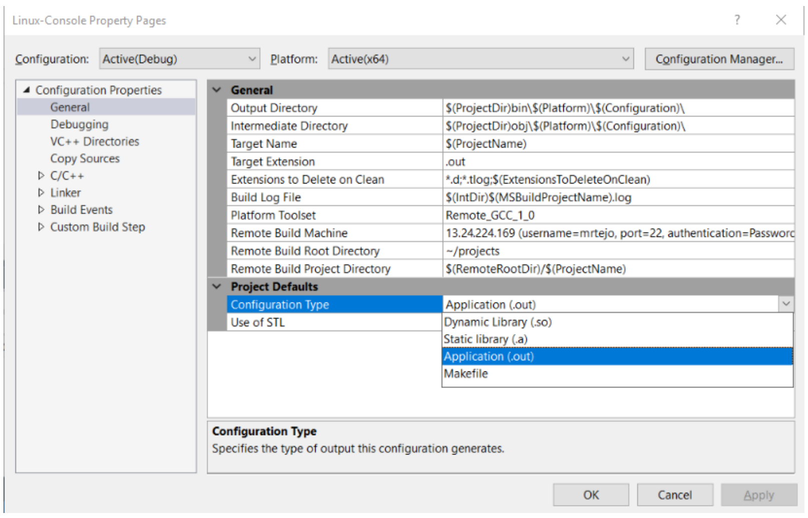
You can configure a Linux project to target a physical Linux machine, a virtual machine, or the Windows Subsystem for Linux (WSL).

Visual Studio 2019 version 16.1 and later:

- When you target WSL, you can avoid the copy operations needed to build and get IntelliSense that are required when you target a remote Linux system.
- You can specify separate Linux targets for building and debugging.

General settings

To view configuration options, select the **Project > Properties** menu, or right-click on the project in **Solution Explorer** and select **Properties** from the context menu. The **General** settings appear.



Configuration: Active(Debug)   Platform: Active(x64)

General	
Output Directory	$(ProjectDir)bin\$(Platform)\$(Configuration)\
Intermediate Directory	$(ProjectDir)obj\$(Platform)\$(Configuration)\
Target Name	$(ProjectName)
Target Extension	.out
Extensions to Delete on Clean	*.d;*.tlog;$(ExtensionsToDeleteOnClean)
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	Remote_GCC_1_0
Remote Build Machine	13.24.224.169 (username=mrtejo, port=22, authentication=Passworc
Remote Build Root Directory	~/projects
Remote Build Project Directory	$(RemoteRootDir)/$(ProjectName)
Project Defaults	
Configuration Type	Application (.out)
Use of STL	

Configuration Type
Specifies the type of output this configuration generates.

By default, an executable (.out) is built. To build a static or dynamic library, or to use an existing Makefile, use the **Configuration Type** setting.

If you're building for Windows Subsystem for Linux (WSL), WSL Version 1 is limited to 64 parallel compilation processes. This is governed by the **Max Parallel Compilation Jobs** setting in **Configuration properties > C/C++ > General**.

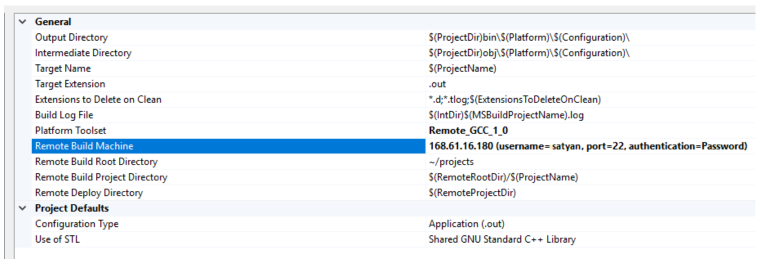
Regardless of the WSL version you are using, if you intend to use more than 64 parallel compilation processes, we recommend that you build with Ninja--which generally will be faster and more reliable. To build with Ninja, use the **Enable Incremental Build** setting in **Configuration properties > General**.

For more information about the settings in the property pages, see Linux Project Property Page Reference.

Remote settings

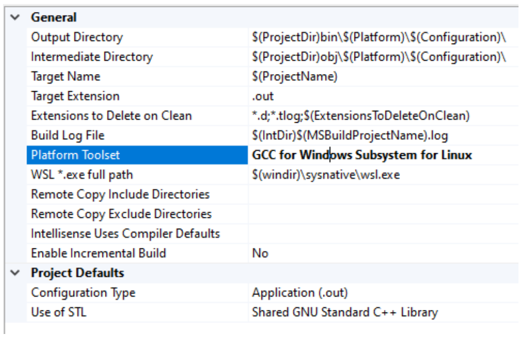
To change settings related to the remote Linux computer, configure the remote settings that appear under General.

- To specify a remote target Linux computer, use the **Remote Build Machine** entry. This will allow you to select one of the connections created previously. To create a new entry, see the Connecting to Your Remote Linux Computer section.



General	
Output Directory	$(ProjectDir)bin\$(Platform)\$(Configuration)\
Intermediate Directory	$(ProjectDir)obj\$(Platform)\$(Configuration)\
Target Name	$(ProjectName)
Target Extension	.out
Extensions to Delete on Clean	*.d;*.tlog;$(ExtensionsToDeleteOnClean)
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**Remote_GCC_1_0**
Remote Build Machine	**168.61.16.180 (username=satyan, port=22, authentication=Password)**
Remote Build Root Directory	~/projects
Remote Build Project Directory	$(RemoteRootDir)/$(ProjectName)
Remote Deploy Directory	$(RemoteProjectDir)
Project Defaults	
Configuration Type	Application (.out)
Use of STL	Shared GNU Standard C++ Library

Visual Studio 2019 version 16.7 and later: To target Windows Subsystem for Linux (WSL), set the **Platform Toolset** drop-down to **GCC for Windows Subsystem for Linux**. The other remote options will disappear and the path to the default WSL shell will appear in their place:



General	
Output Directory	$(ProjectDir)bin\$(Platform)\$(Configuration)\
Intermediate Directory	$(ProjectDir)obj\$(Platform)\$(Configuration)\
Target Name	$(ProjectName)
Target Extension	.out
Extensions to Delete on Clean	*.d;*.tlog;$(ExtensionsToDeleteOnClean)
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**GCC for Windows Subsystem for Linux**
WSL *.exe full path	$(windir)\sysnative\wsl.exe
Remote Copy Include Directories	
Remote Copy Exclude Directories	
Intellisense Uses Compiler Defaults	
Enable Incremental Build	No
Project Defaults	
Configuration Type	Application (.out)
Use of STL	Shared GNU Standard C++ Library

If you have side-by-side WSL installations, you can specify a different path here. For more information about managing multiple distros, see Manage and configure Windows Subsystem for Linux.

You can specify a different target for debugging on the **Configuration Properties** > **Debugging** page.

- The **Remote Build Root Directory** determines the root location of where the project is built on the remote Linux computer. This will default to **~/projects** unless changed.

- The **Remote Build Project Directory** is where this specific project will be built on the remote Linux computer. This will default to **$(RemoteRootDir)/$(ProjectName)**, which will expand to a directory named after the current project, under the root directory set above.

> **NOTE**
>
> To change the default C and C++ compilers, or the Linker and Archiver used to build the project, use the appropriate entries in the **C/C++ > General** section and the **Linker > General** section. You can specify a certain version of GCC or Clang, for example. For more information, see C/C++ Properties (Linux C++) and Linker Properties (Linux C++).

Copy sources (remote systems only)

> **NOTE**
>
> This section doesn't apply when targeting WSL.

When building on remote systems, the source files on your development PC are copied to the Linux computer and compiled there. By default, all sources in the Visual Studio project are copied to the locations set in the settings above. However, additional sources can also be added to the list, or copying sources can be turned off entirely, which is the default for a Makefile project.

- **Sources to copy** determines which sources are copied to the remote computer. By default, the **@(SourcesToCopyRemotely)** defaults to all source code files in the project, but doesn't include any asset/resource files, such as images.

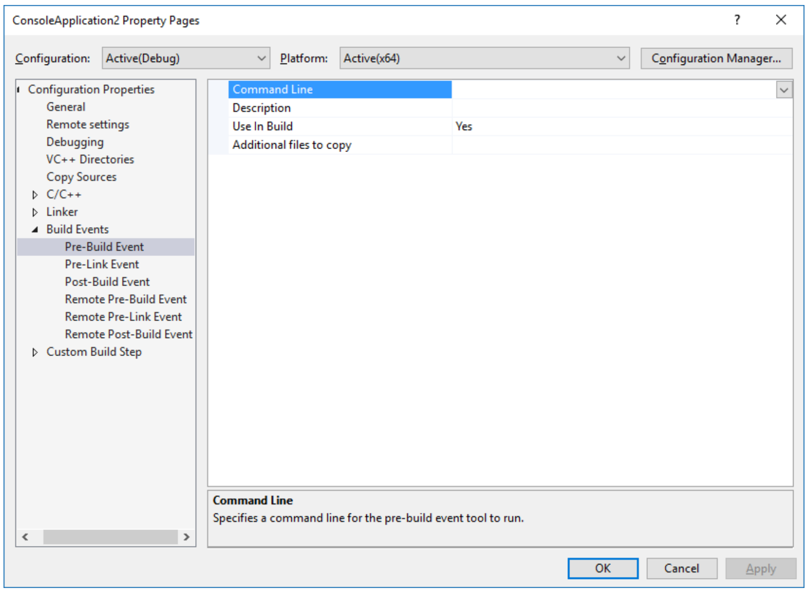
- **Copy sources** can be turned on and off to enable and disable the copying of source files to the remote computer.

- **Additional sources to copy** allows you to add additional source files, which will be copied to the remote system. You can specify a semi-colon delimited list, or you can use the := syntax to specify a local and remote name to use:

```
C:\Projects\ConsoleApplication1\MyFile.cpp:=~/projects/ConsoleApplication1/ADifferentName.cpp;C:\Projects\ConsoleApplication1\MyFile2.cpp:=~/projects/Consol
```

Build events

Since all compilation is happening on a remote computer (or WSL), several additional Build Events have been added to the Build Events section in Project Properties. These are **Remote Pre-Build Event**, **Remote Pre-Link Event**, and **Remote Post-Build Event**, and will occur on the remote computer before or after the individual steps in the process.


ConsoleApplication2 Property Pages

Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

Configuration Properties
- General
- Remote settings
- Debugging
- VC++ Directories
- Copy Sources
- C/C++
- Linker
- Build Events
  - Pre-Build Event
  - Pre-Link Event
  - Post-Build Event
  - Remote Pre-Build Event
  - Remote Pre-Link Event
  - Remote Post-Build Event
- Custom Build Step

Command Line	
Description	
Use In Build	Yes
Additional files to copy	

Command Line
Specifies a command line for the pre-build event tool to run.

OK Cancel Apply

IntelliSense for headers on remote systems

When you add a new connection in **Connection Manager**, Visual Studio automatically detects the include directories for the compiler on the remote system. Visual Studio then zips up and copies those files to a directory on your local Windows machine. After that, whenever you use that connection in a Visual Studio or CMake project, the headers in those directories are used to provide IntelliSense.

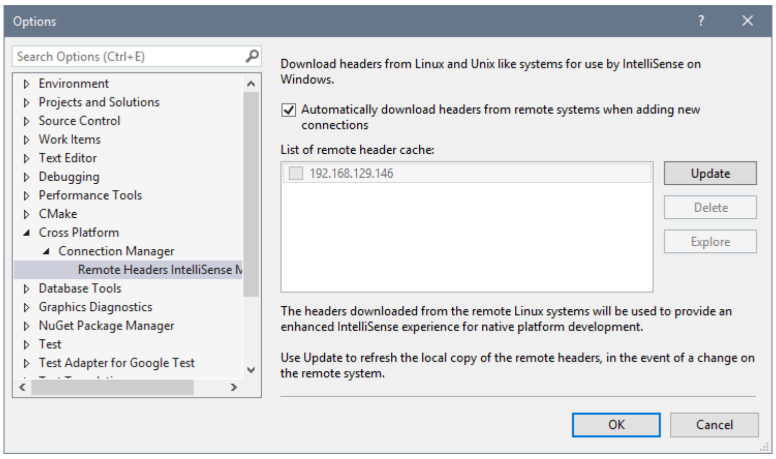
NOTE

In Visual Studio 2019 version 16.5 and later, the remote header copy has been optimized. Headers are now copied on-demand when opening a Linux project or configuring CMake for a Linux target. The copy occurs in the background on a per-project basis, based on the project's specified compilers. For more information, see Improvements to Accuracy and Performance of Linux IntelliSense.

This functionality depends on the Linux machine having zip installed. You can install zip by using this apt-get command:

```
sudo apt install zip
```

To manage your header cache, navigate to **Tools > Options, Cross Platform > Connection Manager > Remote Headers IntelliSense Manager**. To update the header cache after making changes on your Linux machine, select the remote connection and then select **Update**. Select **Delete** to remove the headers without deleting the connection itself. Select **Explore** to open the local directory in **File Explorer**. Treat this folder as read-only. To download headers for an existing connection that was created before Visual Studio 2017 version 15.3, select the connection and then select **Download**.


Options

Search Options (Ctrl+E)

- Environment
- Projects and Solutions
- Source Control
- Work Items
- Text Editor
- Debugging
- Performance Tools
- CMake
- Cross Platform
  - Connection Manager
    - Remote Headers IntelliSense M
- Database Tools
- Graphics Diagnostics
- NuGet Package Manager
- Test
- Test Adapter for Google Test

Download headers from Linux and Unix like systems for use by IntelliSense on Windows.

[✓] Automatically download headers from remote systems when adding new connections

List of remote header cache:

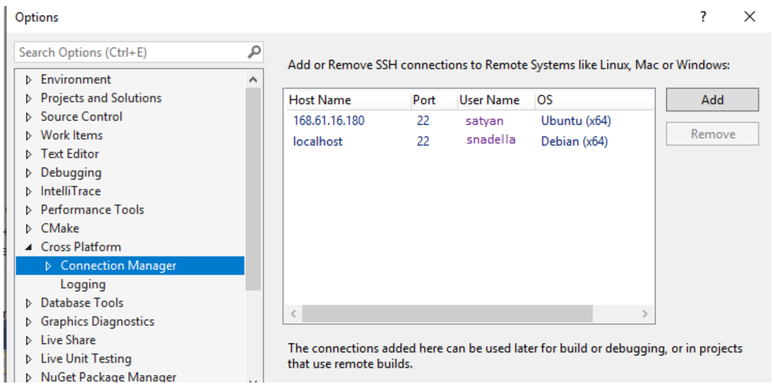
192.168.129.146

Update Delete Explore

The headers downloaded from the remote Linux systems will be used to provide an enhanced IntelliSense experience for native platform development.

Use Update to refresh the local copy of the remote headers, in the event of a change on the remote system.

OK Cancel

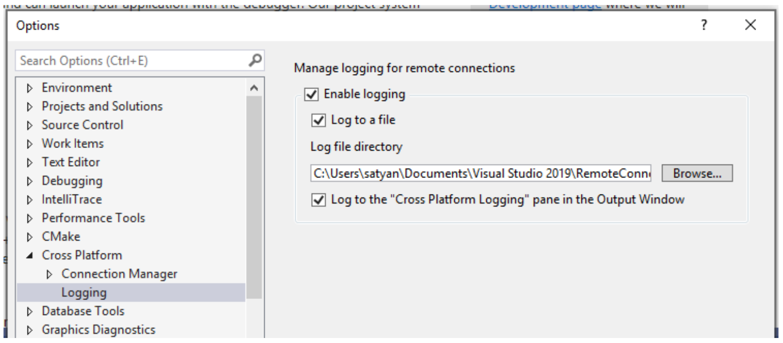


Add or Remove SSH connections to Remote Systems like Linux, Mac or Windows:

Host Name	Port	User Name	OS
168.61.16.180	22	satyan	Ubuntu (x64)
localhost	22	snadella	Debian (x64)

The connections added here can be used later for build or debugging, or in projects that use remote builds.

You can enable logging to help troubleshoot problems:



Manage logging for remote connections

- [✓] Enable logging
- [✓] Log to a file

Log file directory

C:\Users\satyan\Documents\Visual Studio 2019\RemoteConn

- [✓] Log to the "Cross Platform Logging" pane in the Output Window

Linux target locale

Visual Studio language settings aren't propagated to Linux targets because Visual Studio doesn't manage or configure installed packages. Messages shown in the **Output** window, such as build errors, are shown using the language and locale of the Linux target. You'll need to configure your Linux targets for the desired locale.

See also

Set compiler and build properties
C++ General Properties (Linux C++)
VC++ Directories (Linux C++)
Copy Sources Project Properties (Linux C++)
Build Event Properties (Linux C++)

Deploy, run, and debug your Linux MSBuild project

9/21/2022 • 7 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later. To see the documentation for these versions, set the **Version** drop-down located above the table of contents to **Visual Studio 2017** or **Visual Studio 2019**.

Once you've created a MSBuild-based Linux C++ project in Visual Studio and you've connected to the project using the Linux Connection Manager, you can run and debug the project. You compile, execute, and debug the code on the remote target.

Visual Studio 2019 version 16.1 and later: You can target different Linux systems for debugging and building. For example, you can cross-compile on x64 and deploy to an ARM device when targeting IoT scenarios. For more information, see Specify different machines for building and debugging later in this article.

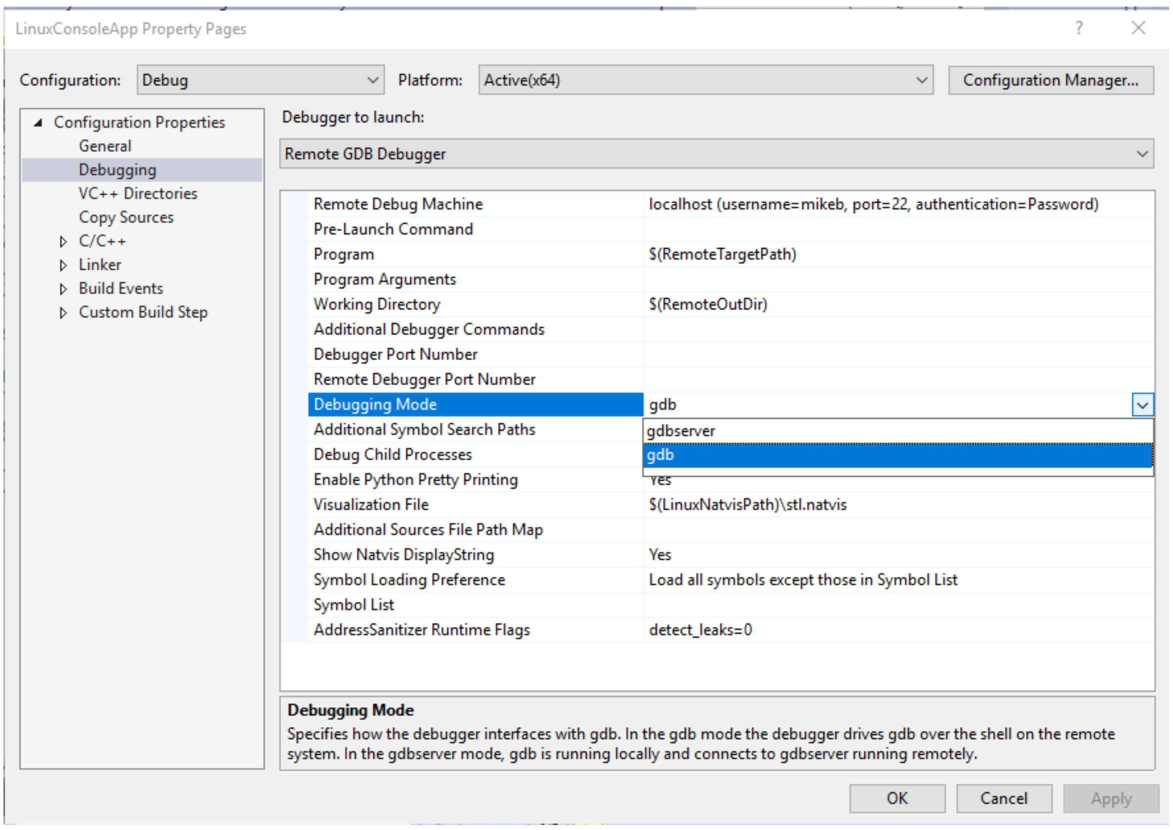
There are several ways to interact with and debug your Linux project.

- Debug using traditional Visual Studio features, such as breakpoints, watch windows, and hovering over a variable. Using these methods, you may debug as you normally would for other project types.
- View output from the target computer in the Linux Console window. You can also use the console to send input to the target computer.

Debug your Linux project

1. Select debugging mode in the **Debugging** property page.

   GDB is used to debug applications running on Linux. When debugging on a remote system (not WSL) GDB can run in two different modes, which can be selected from the **Debugging Mode** option in the project's **Debugging** property page:



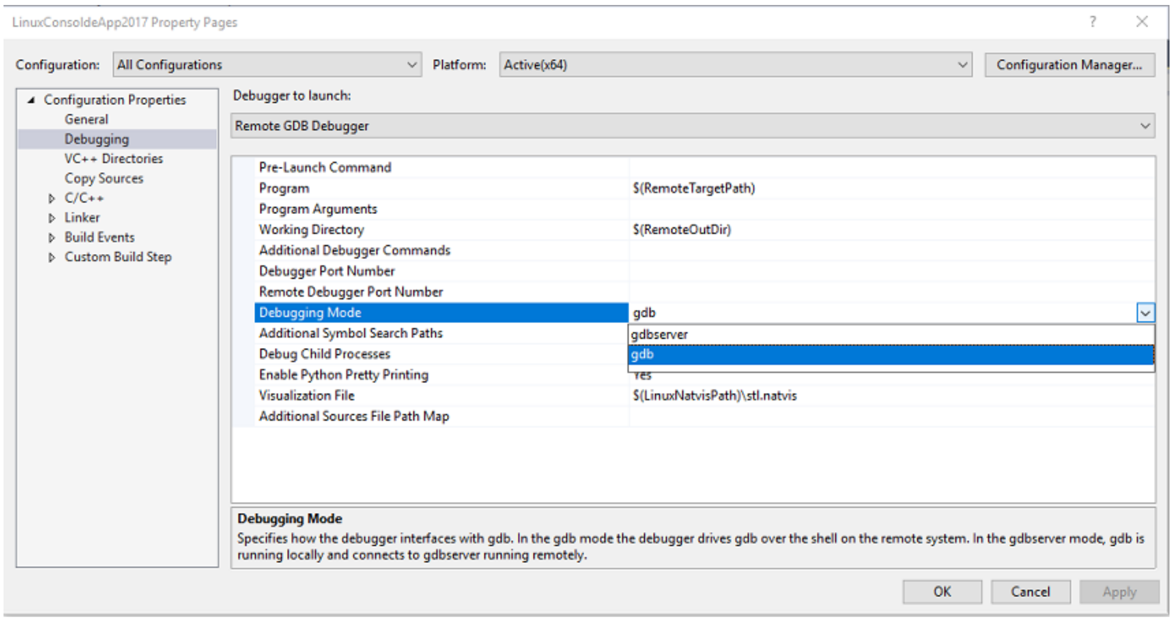
Configuration: Debug
Platform: Active(x64)

Debugger to launch:
Remote GDB Debugger

Remote Debug Machine	localhost (username=mikeb, port=22, authentication=Password)
Pre-Launch Command	
Program	$(RemoteTargetPath)
Program Arguments	
Working Directory	$(RemoteOutDir)
Additional Debugger Commands	
Debugger Port Number	
Remote Debugger Port Number	
Debugging Mode	gdb
Additional Symbol Search Paths	
Debug Child Processes	
Enable Python Pretty Printing	Yes
Visualization File	$(LinuxNatvisPath)\stl.natvis
Additional Sources File Path Map	
Show Natvis DisplayString	Yes
Symbol Loading Preference	Load all symbols except those in Symbol List
Symbol List	
AddressSanitizer Runtime Flags	detect_leaks=0

Debugging Mode
Specifies how the debugger interfaces with gdb. In the gdb mode the debugger drives gdb over the shell on the remote system. In the gdbserver mode, gdb is running locally and connects to gdbserver running remotely.

GDB is used to debug applications running on Linux. GDB can run in two different modes, which can be selected from the **Debugging Mode** option in the project's **Debugging** property page:



Configuration: All Configurations
Platform: Active(x64)

Debugger to launch: Remote GDB Debugger

Pre-Launch Command	
Program	$(RemoteTargetPath)
Program Arguments	
Working Directory	$(RemoteOutDir)
Additional Debugger Commands	
Debugger Port Number	
Remote Debugger Port Number	
Debugging Mode	gdb
Additional Symbol Search Paths	gdbserver
Debug Child Processes	gdb
Enable Python Pretty Printing	Yes
Visualization File	$(LinuxNatvisPath)\stl.natvis
Additional Sources File Path Map	

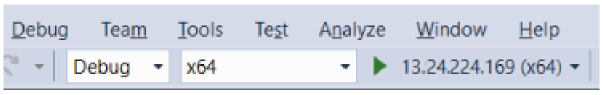
Debugging Mode
Specifies how the debugger interfaces with gdb. In the gdb mode the debugger drives gdb over the shell on the remote system. In the gdbserver mode, gdb is running locally and connects to gdbserver running remotely.

- In **gdbserver** mode, GDB is run locally, which connects to gdbserver on the remote system.
- In **gdb** mode, the Visual Studio debugger drives GDB on the remote system. This is a better option if the local version of GDB isn't compatible with the version installed on the target computer. This is the only mode that the Linux Console window supports.

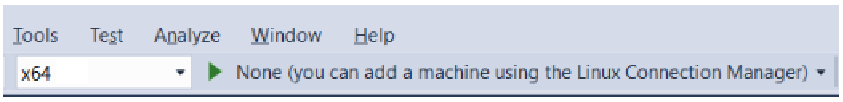
> **NOTE**
>
> If you are unable to hit breakpoints in gdbserver debugging mode, try gdb mode. gdb must first be installed on the remote target.

2. Select the remote target using the standard **Debug** toolbar in Visual Studio.

   When the remote target is available, you'll see it listed by either name or IP address.



   If you haven't connected to the remote target yet, you'll see an instruction to use Linux Connection Manager to connect to the remote target.



None (you can add a machine using the Linux Connection Manager)

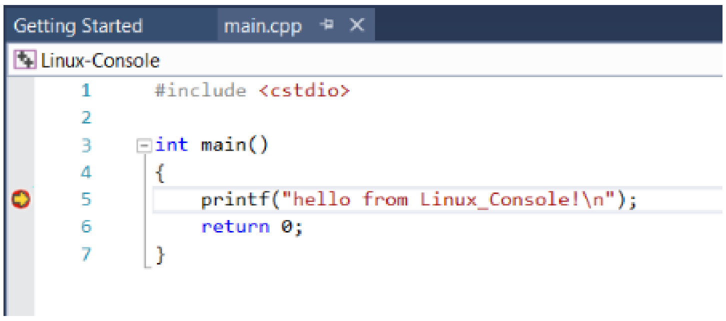
3. Set a breakpoint by clicking in the left gutter of some code that you know will execute.

   A red dot appears on the line of code where you set the breakpoint.

4. Press **F5** (or **Debug > Start Debugging**) to start debugging.

   When you start debugging, the application is compiled on the remote target before it starts. Any compilation errors will appear in the **Error List** window.

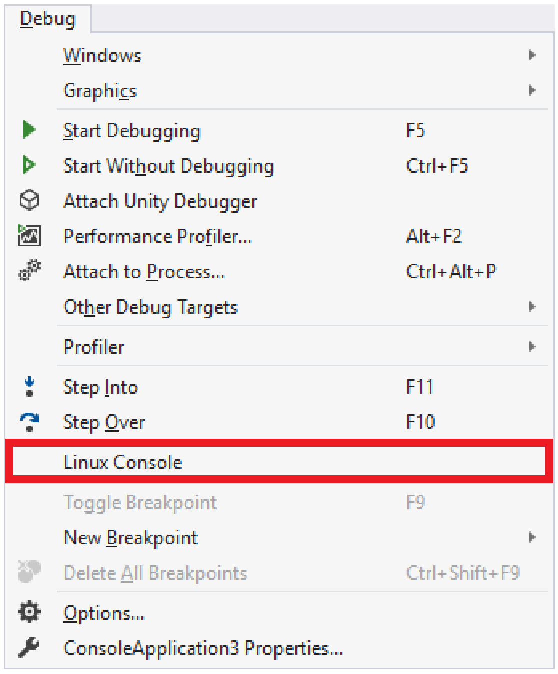
   If there are no errors, the app will start and the debugger will pause at the breakpoint.



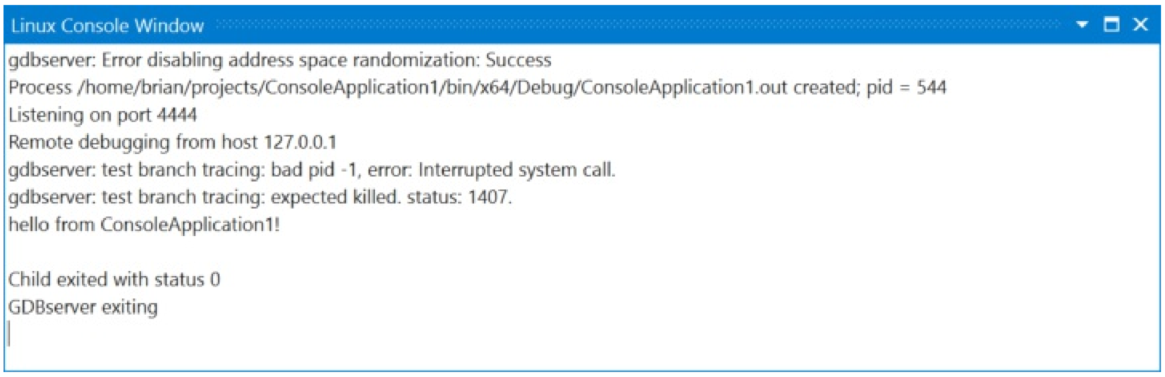
```
#include <cstdio>

int main()
{
    printf("hello from Linux_Console!\n");
    return 0;
}
```

Now, you can interact with the application in its current state, view variables, and step through code by pressing command keys such as **F10** or **F11**.

5. If you want to use the Linux Console to interact with your app, select **Debug** > **Linux Console**.



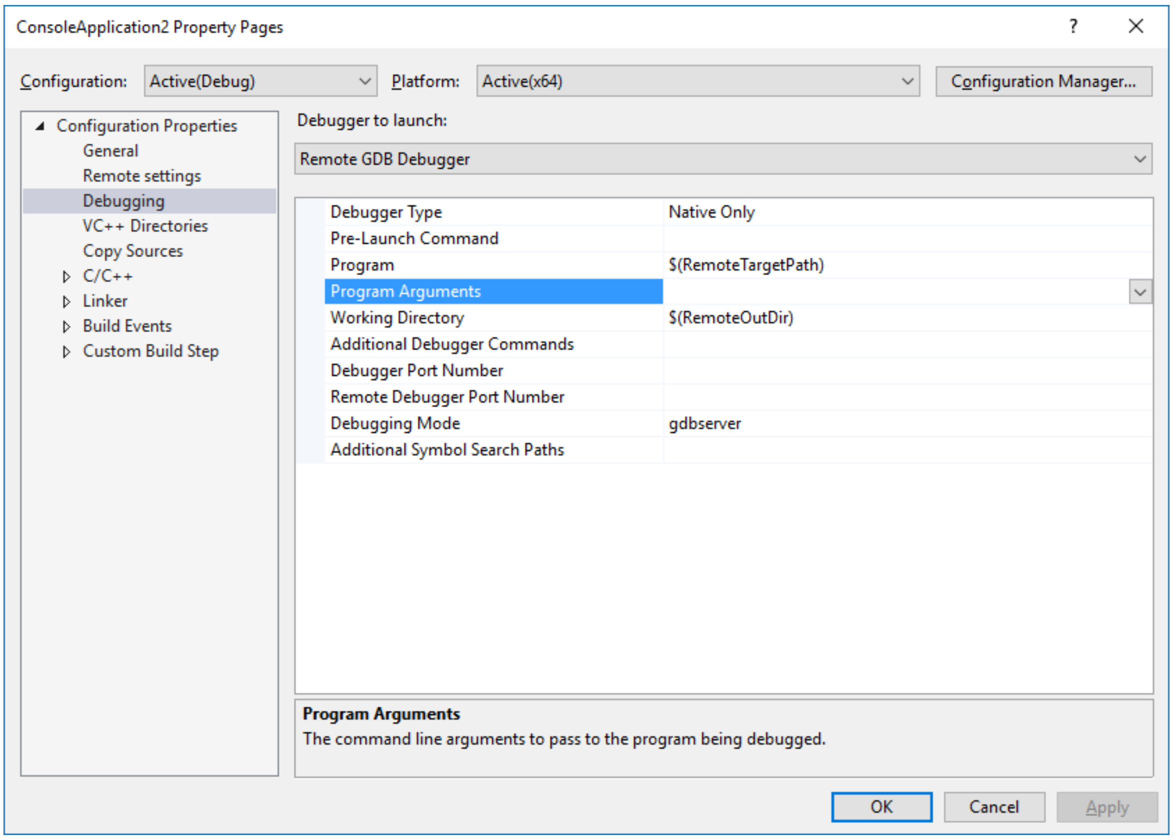
This console will display any console output from the target computer and take input and send it to the target computer.



```
gdbserver: Error disabling address space randomization: Success
Process /home/brian/projects/ConsoleApplication1/bin/x64/Debug/ConsoleApplication1.out created; pid = 544
Listening on port 4444
Remote debugging from host 127.0.0.1
gdbserver: test branch tracing: bad pid -1, error: Interrupted system call.
gdbserver: test branch tracing: expected killed. status: 1407.
hello from ConsoleApplication1!

Child exited with status 0
GDBserver exiting
```

Configure other debugging options (MSBuild projects)

- Command-line arguments can be passed to the executable using the **Program Arguments** item in the project's **Debugging** property page.
- You can export the `DISPLAY` environment variable by using the **Pre-Launch Command** in the project's **Debugging** property pages. For example: `export DISPLAY=:0.0`


ConsoleApplication2 Property Pages

Configuration: Active(Debug) Platform: Active(x64)

Debugger to launch: Remote GDB Debugger

Debugger Type	Native Only
Pre-Launch Command	
Program	$(RemoteTargetPath)
Program Arguments	
Working Directory	$(RemoteOutDir)
Additional Debugger Commands	
Debugger Port Number	
Remote Debugger Port Number	
Debugging Mode	gdbserver
Additional Symbol Search Paths	

Program Arguments
The command line arguments to pass to the program being debugged.

- Specific debugger options can be passed to GDB using the **Additional Debugger Commands** entry. For example, you might want to ignore SIGILL (illegal instruction) signals. You could use the **handle** command to achieve this by adding the following to the **Additional Debugger Commands** entry as shown above:

  ```
  handle SIGILL nostop noprint
  ```

- You can specify the path to the GDB used by Visual Studio using the **GDB Path** item in the project's **Debugging** property page. This property is available in Visual Studio 2019 version 16.9 and later.

Debug with Attach to Process

The Debugging property page for Visual Studio projects, and the **Launch.vs.json** settings for CMake projects, have settings that enable you to attach to a running process. If you require more control beyond what is provided in those settings, you can place a file named `Microsoft.MIEngine.Options.xml` in the root of your solution or workspace. Here is a simple example:

```
<?xml version="1.0" encoding="utf-8"?>
<SupplementalLaunchOptions>
    <AttachOptions>
      <AttachOptionsForConnection AdditionalSOLibSearchPath="/home/user/solibs">
        <ServerOptions MIDebuggerPath="C:\Program Files (x86)\Microsoft Visual
Studio\Preview\Enterprise\Common7\IDE\VC\Linux\bin\gdb\7.9\x86_64-linux-gnu-gdb.exe"
ExePath="C:\temp\ConsoleApplication17\ConsoleApplication17\bin\x64\Debug\ConsoleApplication17.out"/>
        <SetupCommands>
          <Command IgnoreFailures="true">-enable-pretty-printing</Command>
        </SetupCommands>
      </AttachOptionsForConnection>
    </AttachOptions>
</SupplementalLaunchOptions>
```

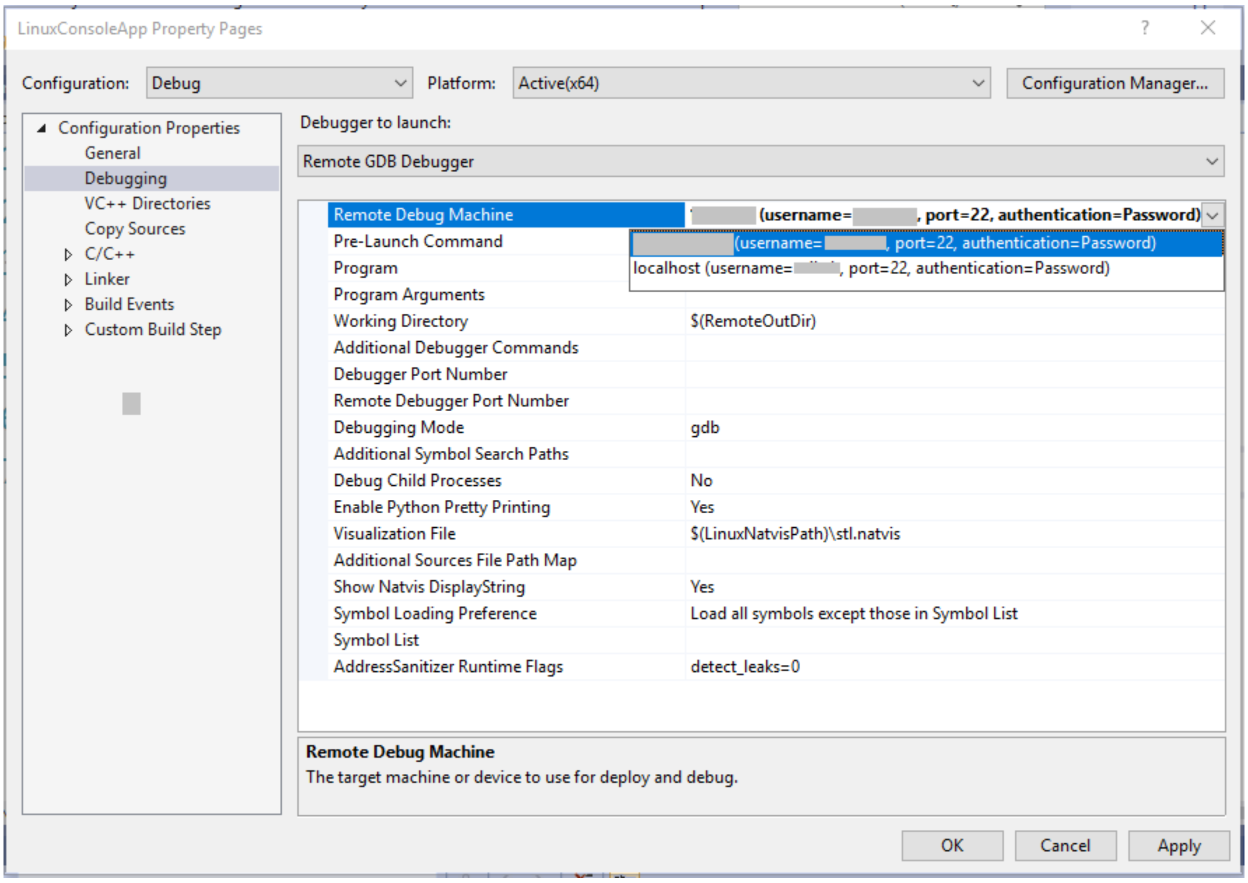
The **AttachOptionsForConnection** has most of the attributes you might need. The example above shows how to specify a location to search for more .so libraries. The child element **ServerOptions** enables attaching to the remote process with gdbserver instead. To do that, you need to specify a local gdb client (the one shipped in

Visual Studio 2017 is shown above) and a local copy of the binary with symbols. The **SetupCommands** element enables you to pass commands directly to gdb. You can find all the options available in the LaunchOptions.xsd schema on GitHub.

Specify different machines for building and debugging in MSBuild-based Linux projects

You can separate your remote build machine from your remote debug machine for both MSBuild-based Linux projects and CMake projects that target a remote Linux machine. For example, you can now cross-compile on x64 and deploy to an ARM device when targeting IoT scenarios.

By default, the remote debug machine is the same as the remote build machine (**Configuration Properties** > **General** > **Remote Build Machine**). To specify a new remote debug machine, right-click on the project in **Solution Explorer** and go to **Configuration Properties** > **Debugging** > **Remote Debug Machine**.


LinuxConsoleApp Property Pages

Configuration: Debug   Platform: Active(x64)   Configuration Manager...

- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - Copy Sources
  - C/C++
  - Linker
  - Build Events
  - Custom Build Step

Debugger to launch:

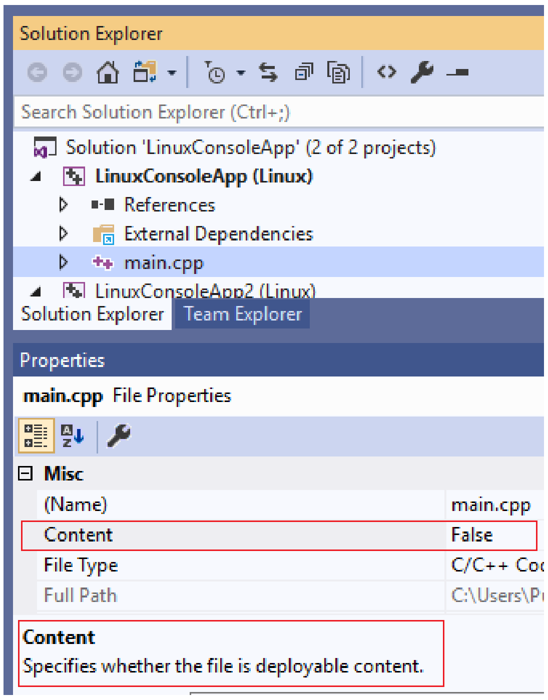
Remote GDB Debugger

Remote Debug Machine	[redacted] (username=[redacted], port=22, authentication=Password)
Pre-Launch Command	[redacted] (username=[redacted], port=22, authentication=Password)
Program	localhost (username=[redacted], port=22, authentication=Password)
Program Arguments	
Working Directory	$(RemoteOutDir)
Additional Debugger Commands	
Debugger Port Number	
Remote Debugger Port Number	
Debugging Mode	gdb
Additional Symbol Search Paths	
Debug Child Processes	No
Enable Python Pretty Printing	Yes
Visualization File	$(LinuxNatvisPath)\stl.natvis
Additional Sources File Path Map	
Show Natvis DisplayString	Yes
Symbol Loading Preference	Load all symbols except those in Symbol List
Symbol List	
AddressSanitizer Runtime Flags	detect_leaks=0

Remote Debug Machine
The target machine or device to use for deploy and debug.

The drop-down menu for **Remote Debug Machine** is populated with all established remote connections. To add a new remote connection, navigate to **Tools** > **Options** > **Cross Platform** > **Connection Manager** or search for "Connection Manager" in **Quick Launch**. You can also specify a new remote deploy directory in the project's Property Pages (**Configuration Properties** > **General** > **Remote Deploy Directory**).

By default, only the files necessary for the process to debug will be deployed to the remote debug machine. You can use **Solution Explorer** to configure which source files will be deployed to the remote debug machine. When you click on a source file, you'll see a preview of its File Properties directly below the Solution Explorer.


Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'LinuxConsoleApp' (2 of 2 projects)
LinuxConsoleApp (Linux)
References
External Dependencies
main.cpp
LinuxConsoleApp2 (Linux)

Solution Explorer | Team Explorer

Properties

main.cpp File Properties

Misc

(Name)	main.cpp
Content	False
File Type	C/C++ Coc
Full Path	C:\Users\P

Content
Specifies whether the file is deployable content.

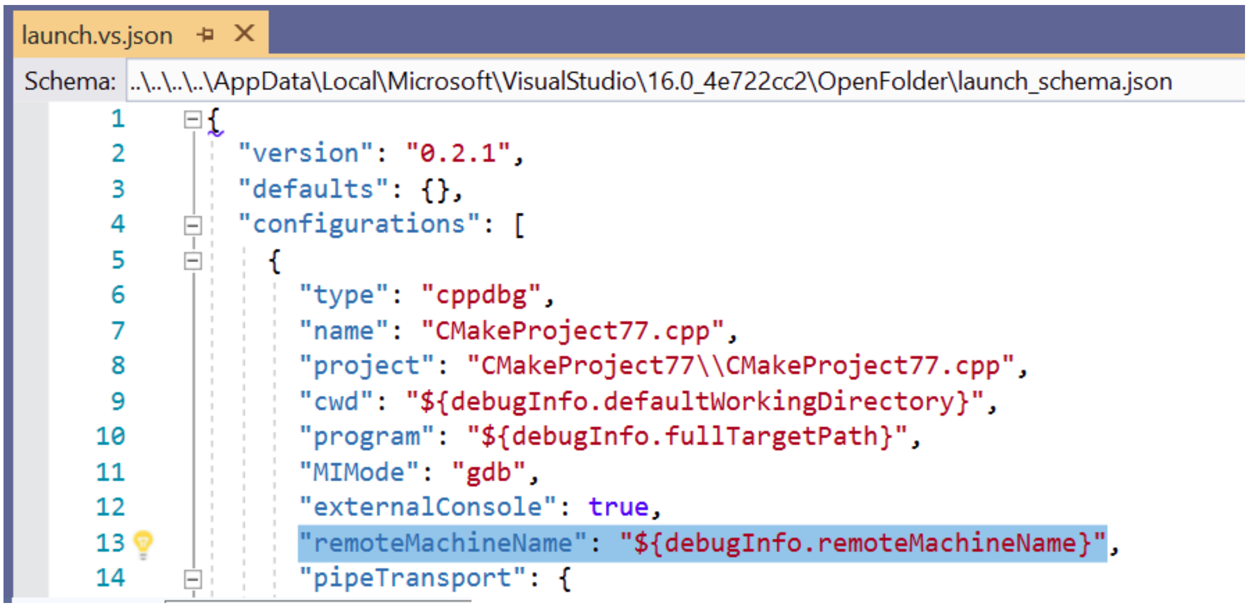
The **Content** property specifies whether the file will be deployed to the remote debug machine. You can disable deployment entirely by navigating to **Property Pages** > **Configuration Manager** and unchecking **Deploy** for the desired configuration.

In some cases, you may require more control over your project's deployment. For example, some files that you want to deploy might be outside of your solution or you want to customize your remote deploy directory per file or directory. In these cases, append the following code block(s) to your .vcxproj file and replace "example.cpp" with the actual file names:

```
<ItemGroup>
   <RemoteDeploy Include="__example.cpp">
<!-- This is the source Linux machine, can be empty if DeploymentType is LocalRemote -->
      <SourceMachine>$(RemoteTarget)</SourceMachine>
      <TargetMachine>$(RemoteDebuggingTarget)</TargetMachine>
      <SourcePath>~/example.cpp</SourcePath>
      <TargetPath>~/example.cpp</TargetPath>
<!-- DeploymentType can be LocalRemote, in which case SourceMachine will be empty and SourcePath is a local
file on Windows -->
      <DeploymentType>RemoteRemote</DeploymentType>
<!-- Indicates whether the deployment contains executables -->
      <Executable>true</Executable>
   </RemoteDeploy>
</ItemGroup>
```

CMake projects

For CMake projects that target a remote Linux machine, you can specify a new remote debug machine in launch.vs.json. By default, the value of "remoteMachineName" is synchronized with the "remoteMachineName" property in CMakeSettings.json, which corresponds to your remote build machine. These properties no longer need to match, and the value of "remoteMachineName" in launch.vs.json will dictate which remote machine is used for deploy and debug.


launch.vs.json

Schema: ..\..\..\..\AppData\Local\Microsoft\VisualStudio\16.0_4e722cc2\OpenFolder\launch_schema.json

```
{
  "version": "0.2.1",
  "defaults": {},
  "configurations": [
    {
      "type": "cppdbg",
      "name": "CMakeProject77.cpp",
      "project": "CMakeProject77\\CMakeProject77.cpp",
      "cwd": "${debugInfo.defaultWorkingDirectory}",
      "program": "${debugInfo.fullTargetPath}",
      "MIMode": "gdb",
      "externalConsole": true,
      "remoteMachineName": "${debugInfo.remoteMachineName}",
      "pipeTransport": {
```

IntelliSense will suggest all a list of all established remote connections. You can add a new remote connection by navigating to **Tools** > **Options** > **Cross Platform** > **Connection Manager** or searching for "Connection Manager" in **Quick Launch**.

If you want complete control over your deployment, you can append the following code block(s) to the launch.vs.json file. Remember to replace the placeholder values with real values:

```
"disableDeploy": false,
"deployDirectory": "~\foo",
"deploy" : [
   {
      "sourceMachine": "127.0.0.1 (username=example1, port=22, authentication=Password)",
      "targetMachine": "192.0.0.1 (username=example2, port=22, authentication=Password)",
      "sourcePath": "~/example.cpp",
      "targetPath": "~/example.cpp",
      "executable": "false"
   }
]
```

Next steps

- To debug ARM devices on Linux, see this blog post: Debugging an embedded ARM device in Visual Studio.

See also

C++ Debugging Properties (Linux C++)

Linux Project Property Page Reference

9/21/2022 • 2 minutes to read • Edit Online

This section contains reference content for property pages in a Visual C++ Linux project.

- General Properties (Linux)
- Debugging Properties (Linux)
- VC++ Directories Properties (Linux)
- Copy Sources Properties (Linux)
- C/C++ Properties (Linux)
- Linker Properties (Linux)
- Build Event Properties (Linux)
- Custom Build Step Properties (Linux)
- Makefile Project Properties (Linux)

General properties (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

PROPERTY	DESCRIPTION
Output Directory	Specifies a relative path to the output file directory. It can include environment variables.
Intermediate Directory	Specifies a relative path to the intermediate file directory. It can include environment variables.
Target Name	Specifies the file name that this project generates.
Target Extension	Specifies the file extension (for example, `.a`) that this project generates.
Extensions to Delete on Clean	Semi-colon-delimited wildcard specification for which files in the intermediate directory to delete on clean or rebuild.
Build Log File	Specifies the build log file to write to when build logging is enabled.
Platform Toolset	Specifies the toolset used for building the current configuration. If not set, the default toolset is used.
WSL *.exe full path	**Visual Studio 2019 version 16.1** Full path to the Windows Subsystem for Linux (WSL) executable used to build and debug.
Remote Build Machine	Displays the target machine or device to use for remote build, deploy, and debug. You can add or edit a target machine connection by using **Tools** > **Options** > **Cross Platform** > **Connection Manager**. **Visual Studio 2019 version 16.1** You can specify a different machine for debugging on the Debugging page.
Remote Build Root Directory	Specifies a path to a directory on the remote machine or device.
Remote Build Project Directory	Specifies a path to a directory on the remote machine or device for the project.
Remote Deploy Directory	**Visual Studio 2019 version 16.1** Specifies the directory path on the remote machine or device to deploy the project.
Enable Incremental Build	**Visual Studio 2019 version 16.7** Specifies whether to do incremental builds using the Ninja build system. Builds will usually be faster for most projects with this setting enabled.

PROPERTY	DESCRIPTION
Remote Copy Include Directories	**Visual Studio 2019 version 16.5** A list of directories to copy recursively from the Linux target. This property affects the remote header copy for IntelliSense, but not the build. It can be used when **IntelliSense Uses Compiler Defaults** is set to false. Use **Additional Include Directories** under the C/C++ General tab to specify additional include directories to use for both IntelliSense and build.
Remote Copy Exclude Directories	**Visual Studio 2019 version 16.5** A list of directories *not* to copy from the Linux target. Usually, this property is used to remove subdirectories of the include directories.
IntelliSense Uses Compiler Defaults	**Visual Studio 2019 version 16.5** Whether to query the compiler referenced by this project for its default list of include locations. These locations are automatically added to the list of remote directories to copy. Only set this property to false if the compiler doesn't support gcc-like parameters. Both gcc and clang compilers support queries for the include directories (for example, `g++ -x c++ -E -v -std=c++11`).
Configuration Type	Specifies the type of output this configuration generates, such as: **Dynamic Library (.so)**, **Static library (.a)**, **Application (.out)**, and **Makefile**
Use of STL	Specifies which C++ Standard Library to use for this configuration, such as: **Shared GNU Standard C++ Library**, or **Static GNU Standard C++ Library (-static)**

C++ Debugging Properties (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

PROPERTY	DESCRIPTION	CHOICES
Remote debug machine	**Visual Studio 2019 version 16.1**: Specifies the machine to debug the program on. Can be different than the remote build machine that's specified on the General page. You can add or edit a target machine connection by using **Tools** > **Options** > **Cross Platform** > **Connection Manager**.	
Pre-Launch Command	A command that's run on the shell before the debugger starts, that can be used to affect the debugging environment.	
Program	The full path on the remote system to the program to debug. If left empty or unchanged, it defaults to the current project output.	
Program Arguments	The command-line arguments to pass to the program being debugged.	
Working Directory	The remote application's working directory. By default, the user home directory.	
Additional Debugger Commands	Additional `gdb` commands for the debugger to run before starting debugging.	
Debugger Port Number	The port number for debugger communication with the remote debugger. The port must not be in use locally. This value must be positive, and between 1 and 65535. If not supplied, a free port number is used.	
Remote Debugger Port Number	The port number on which the remote debugger server `gdbserver` is listening on the remote system. The port must not be in use on the remote system. This value must be positive, and between 1 and 65535. If not supplied, a free port number starting from 4444 is used.	

PROPERTY	DESCRIPTION	CHOICES
Debugging Mode	Specifies how the debugger interfaces with `gdb`. In *gdb mode*, the debugger drives `gdb` over the shell on the remote system. In *gdbserver mode*, `gdb` runs locally and connects to `gdbserver` running remotely.	**gdbserver** **gdb**
Additional Symbol Search Paths	Additional search path for debug symbols (solib-search-path).	
Debug Child Processes	Specifies whether to enable debugging of child processes.	
Enable Python Pretty Printing	Enable pretty printing of expression values. Only supported in gdb debugging mode.	
Visualization File	Default native visualization file (.natvis) containing visualization directives for SLT types. Other .natvis files that belong to the current solution are loaded automatically.	
Additional Sources File Path Map	Additional path equivalences for the debugger to use to map Windows source file names to Linux source file names. The format is "<windows-path>=<linux-path>;...". A source file name found under the Windows path is referenced as if it's found in the same relative position under the Linux path. Files found in the local project don't require additional mapping.	
GDB Path	**Visual Studio 2019 version 16.9**: Specifies the path to the GDB executable to be used by Visual Studio.	

VC++ Directories (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

PROPERTY	DESCRIPTION
Include Directories	Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.
Library Directories	Path to use when searching for library files while building a VC++ project. Corresponds to environment variable LIB.
Source Directories	Path to use when searching for source files to use for IntelliSense.
Exclude Directories	Path to skip when searching for scan dependencies.

Copy Sources Project Properties (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

The properties set on this property page apply to all files in the project except whose file-level properties are set.

PROPERTY	DESCRIPTION
Sources To Copy	Specifies the sources to copy to the remote system. Changing this list might shift or otherwise affect the directory structure where files are copied to on the remote system.
Copy Sources	Specifies whether to copy the sources to the remote system.
Additional Sources To Copy	Specifies additional sources to copy to the remote system. Optionally specify local to remote mapping pairs using a syntax like this: fulllocalpath1:=fullremotepath1;fulllocalpath2:=fullremotepath2, where a local file can be copied to the specified remote location on the remote system.

@SourcesToCopyRemotely and @DataFilesToCopyRemotely refer to item groups in the project file. To modify which sources or data files are copied remotely, edit the *vcxproj* file like this:

```
<ItemGroup>
   <MyItems Include="foo.txt" />
   <MyItems Include="bar.txt" />
   <DataFilesToCopyRemotely Include="@(MyItems)" />
</ItemGroup>
```

C/C++ Properties (Linux C++)

9/21/2022 • 4 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

General

PROPERTY	DESCRIPTION	CHOICES
Additional Include Directories	Specifies one or more directories to add to the include path. Use semi-colons to separate multiple directories. (-I[path]).	
Debug Information Format	Specifies the type of debugging information generated by the compiler.	**None** - Produces no debugging information, so compilation may be faster. **Minimal Debug Information** - Generate minimal debug information. **Full Debug Information (DWARF2)** - Generate DWARF2 debug information.
Object File Name	Specifies a name to override the default object file name. It can be a file or directory name. (-o [name]).	
Warning Level	Selects how strict you want the compiler to be about code errors. Add other flags directly to **Additional Options**. (/w, /Weverything).	**Turn Off All Warnings** - Disables all compiler warnings. **EnableAllWarnings** - Enables all warnings, including ones disabled by default.
Treat Warnings As Errors	Treats all compiler warnings as errors. For a new project, it may be best to use /Werror in all compilations. Resolve all warnings to ensure the fewest possible hard-to-find code defects.	
C Additional Warnings	Defines a set of additional warning messages.	
C++ Additional Warnings	Defines a set of additional warning messages.	
Enable Verbose mode	When Verbose mode is enabled, prints out more information to diagnose the build.	
C Compiler	Specifies the program to invoke during compilation of C source files, or the path to the C compiler on the remote system.	

PROPERTY	DESCRIPTION	CHOICES
C++ Compiler	Specifies the program to invoke during compilation of C++ source files, or the path to the C++ compiler on the remote system.	
Compile Timeout	Remote compilation timeout, in milliseconds.	
Copy Object Files	Specifies whether to copy the compiled object files from the remote system to the local machine.	
Max Parallel Compilation Jobs	The number of processes to create in parallel during compilation. The default is 1. If you're using Windows Subsystem for Linux (WSL) version 1, the limit is 64.	
Validate Architecture	Specify whether to check if the platform the project targets matches the remote system.	
Enable Address Sanitizer	Compile the program with Address Sanitizer, which is a fast memory error detector that can find runtime memory issues such as use-after-free, and perform out of bounds checks.	

Optimization

PROPERTY	DESCRIPTION	CHOICES
Optimization	Specifies the optimization level for the application.	**Custom** - Custom optimization. **Disabled** - Disable optimization. **Minimize Size** - Optimize for size. **Maximize Speed** - Optimize for speed. **Full Optimization** - Expensive optimizations.
Strict Aliasing	Assumes the strictest aliasing rules. An object of one type is never assumed to have the same address as an object of a different type.	
Unroll Loops	Unrolls loops to make the application faster by reducing the number of branches executed, at the cost of larger code size.	
Link Time Optimization	Enables inter-procedural optimizations by allowing the optimizer to look across object files in your application.	

PROPERTY	DESCRIPTION	CHOICES
Omit Frame Pointer	Suppresses creation of frame pointers on the call stack.	
No Common Blocks	Allocates even uninitialized global variables in the data section of the object file, rather than generate them as common blocks.	

Preprocessor

PROPERTY	DESCRIPTION
Preprocessor Definitions	Defines preprocessing symbols for your source file. (-D)
Undefine Preprocessor Definitions	Specifies one or more preprocessor undefines. (-U [macro])
Undefine All Preprocessor Definitions	Undefines all previously defined preprocessor values. (-undef)
Show Includes	Generates a list of include files with compiler output. (-H)

Code Generation

PROPERTY	DESCRIPTION	CHOICES
Position Independent Code	Generates position-independent code (PIC) for use in a shared library.	
Statics are thread safe	Emits extra code to use routines specified in the C++ ABI for thread-safe initialization of local statics.	**No** - Disable thread-safe statics. **Yes** - Enable thread-safe statics.
Floating Point Optimization	Enables floating-point optimizations by relaxing IEEE-754 conformance.	
Inline Methods Hidden	When enabled, out-of-line copies of inline methods are declared `private extern`.	
Symbols Hidden By Default	All symbols are declared `private extern` unless explicitly marked for export by using the `__attribute` macro.	
Enable C++ Exceptions	Specifies the exception-handling model used by the compiler.	**No** - Disable exception handling. **Yes** - Enable exception handling.

Language

PROPERTY	DESCRIPTION	CHOICES
Enable Run-Time Type Information	Adds code for checking C++ object types at run time (runtime type information). (frtti, fno-rtti)	
C Language Standard	Determines the C language standard.	**Default** **C89** - C89 Language Standard. **C99** - C99 Language Standard. **C11** - C11 Language Standard. **C99 (GNU Dialect)** - C99 (GNU Dialect) Language Standard. **C11 (GNU Dialect)** - C11 (GNU Dialect) Language Standard.
C++ Language Standard	Determines the C++ language standard.	**Default** **C++03** - C++03 Language Standard. **C++11** - C++11 Language Standard. **C++14** - C++14 Language Standard. **C++03 (GNU Dialect)** - C++03 (GNU Dialect) Language Standard. **C++11 (GNU Dialect)** - C++11 (GNU Dialect) Language Standard. **C++14 (GNU Dialect)** - C++14 (GNU Dialect) Language Standard.

Advanced

PROPERTY	DESCRIPTION	CHOICES
Compile As	Selects compilation language option for .c and .cpp files. (-x c, -x c++)	**Default** - Detect based on the .c or .cpp extension. **Compile as C Code** - Compile as C code. **Compile as C++ Code** - Compile as C++ code.
Forced Include Files	Specifies one or more forced include files (-include [name])	

Linker Properties (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

General

PROPERTY	DESCRIPTION	CHOICES
Output File	The option overrides the default name and location of the program that the linker creates. (-o)	
Show Progress	Prints Linker Progress Messages.	
Version	The -version option tells the linker to put a version number in the header of the executable.	
Enable Verbose Output	The -verbose option tells the linker to output verbose messages for debugging.	
Trace	The --trace option tells the linker to output the input files as are processed.	
Trace Symbols	Print the list of files in which a symbol appears. (--trace-symbol=symbol)	
Print Map	The --print-map option tells the linker to output a link map.	
Report Unresolved Symbol References	This option when enabled will report unresolved symbol references.	
Optimize For Memory Usage	Optimize for memory usage, by rereading the symbol tables as necessary.	
Shared Library Search Path	Allows the user to populate the shared library search path. (-rpath-link=path)	
Additional Library Directories	Allows the user to override the environmental library path. (-L folder).	
Linker	Specifies the program to invoke during linking, or the path to the linker on the remote system.	
Link Timeout	Remote linking timeout, in milliseconds.	

PROPERTY	DESCRIPTION	CHOICES
Copy Output	Specifies whether to copy the build output file from the remote system to the local machine.	

Input

PROPERTY	DESCRIPTION	CHOICES
Ignore Specific Default Libraries	Specifies one or more names of default libraries to ignore. (--exclude-libs lib,lib)	
Ignore Default Libraries	Ignore default libraries and only search libraries explicitly specified.	
Force Undefined Symbol References	Force symbol to be entered in the output file as an undefined symbol. (-u symbol --undefined=symbol)	
Library Dependencies	This option allows specifying additional libraries to be added to the linker command line. The additional library will be added to the end of the linker command line prefixed with 'lib' and end with the '.a' extension. (-lFILE)	
Additional Dependencies	Specifies additional items to add to the link command line.	

Debugging

PROPERTY	DESCRIPTION	CHOICES
Debugger Symbol Information	Debugger symbol information from the output file.	**Include All** **Omit Debugger Symbol Information Only** **Omit All Symbol Information**
Map File Name	The Map option tells the linker to create a map file with the user specified name. (-Map=)	

Advanced

PROPERTY	DESCRIPTION	CHOICES
Mark Variables ReadOnly After Relocation	This option marks variables read-only after relocation.	
Enable Immediate Function Binding	This option marks object for immediate function binding.	

PROPERTY	DESCRIPTION	CHOICES
Do Not Require Executable Stack	This option marks output as not requiring executable stack.	
Whole Archive	Whole Archive uses all code from Sources and Additional Dependencies.	

Build Event Properties (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

Pre-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the pre-build event tool to run.
Description	Specifies a description for the pre-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy to the remote system. Optionally specify local to remote mapping pairs using a syntax like this: fulllocalpath1:=fullremotepath1;fulllocalpath2:=fullremotepath2, where a local file can be copied to the specified remote location on the remote system.

Pre-Link Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the pre-link event tool to run.
Description	Specifies a description for the pre-link event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy to the remote system. Optionally specify local to remote mapping pairs using a syntax like this: fulllocalpath1:=fullremotepath1;fulllocalpath2:=fullremotepath2, where a local file can be copied to the specified remote location on the remote system.

Post-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the post-build event tool to run.
Description	Specifies a description for the post-build event tool to display.

PROPERTY	DESCRIPTION
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy to the remote system. Optionally specify local to remote mapping pairs using a syntax like this: fulllocalpath1:=fullremotepath1;fulllocalpath2:=fullremotepath2, where a local file can be copied to the specified remote location on the remote system.

Remote Pre-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the pre-build event tool to run on the remote system.
Description	Specifies a description for the pre-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy from the remote system. Optionally specify remote to local mapping pairs using a syntax like this: fullremotepath1:=fulllocalpath1;fullremotepath2:=fulllocalpath2, where a remote file can be copied to the specified location on the local machine.

Remote Pre-Link Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the pre-link event tool to run on the remote system.
Description	Specifies a description for the pre-link event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy from the remote system. Optionally specify remote to local mapping pairs using a syntax like this: fullremotepath1:=fulllocalpath1;fullremotepath2:=fulllocalpath2, where a remote file can be copied to the specified location on the local machine.

Remote Post-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the post-build event tool to run on the remote system.
Description	Specifies a description for the post-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy from the remote system. Optionally specify remote to local mapping pairs using a syntax like this: fullremotepath1:=fulllocalpath1;fullremotepath2:=fulllocalpath2, where a remote file can be copied to the specified location on the local machine.

Custom Build Step Properties (Linux C++)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

PROPERTY	DESCRIPTION
Command Line	The command to be executed by the custom build step.
Description	A message that's displayed when the custom build step runs.
Outputs	The output file that the custom build step generates. This setting is required so that incremental builds work correctly.
Additional Dependencies	A semicolon-delimited list of any additional input files to use for the custom build step.
Execute After and Execute Before	These options define when the custom build step is run in the build process, relative to the listed targets. The most commonly listed targets are BuildGenerateSources, BuildCompile, and BuildLink, because they represent the major steps in the build process. Other often-listed targets are Midl, CLCompile, and Link.
Treat Output As Content	This option is only meaningful for Microsoft Store or Windows Phone apps, which include all content files in the .appx package.

Makefile Project Properties (Linux C++)

9/21/2022 • 3 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

This is a partial list of the properties available in a Linux Makefile project. Many Makefile project properties are identical to the Linux C++ Console Application project properties.

General

PROPERTY	DESCRIPTION	CHOICES
Output Directory	Specifies a relative path to the output file directory; can include environment variables.	
Intermediate Directory	Specifies a relative path to the intermediate file directory; can include environment variables.	
Build Log File	Specifies the build log file to write to when build logging is enabled.	
Configuration Type	Specifies the type of output this configuration generates.	**Dynamic Library (.so)** - Dynamic Library (.so) **Static library (.a)** - Static Library (.a) **Application (.out)** - Application (.out) **Makefile** - Makefile
Remote Build Machine	The target machine or device to use for remote build, deploy and debug.	
Remote Build Root Directory	Specifies a path to a directory on the remote machine or device.	
Remote Build Project Directory	Specifies a path to a directory on the remote machine or device for the project.	

Debugging

See Debugger Properties (Linux C++)

Copy Sources

See Copy Sources Project Properties (Linux C++).

Build Events

Pre-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the pre-build event tool to run.
Description	Specifies a description for the pre-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy to the remote system. Optionally the list can be provided as a local to remote mapping pairs using a syntax like this: fulllocalpath1:=fullremotepath1;fulllocalpath2:=fullremotepath2, where a local file can be copied to the specified remote location on the remote system.

Post-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the post-build event tool to run.
Description	Specifies a description for the post-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy to the remote system. Optionally the list can be provided as a local to remote mapping pairs using a syntax like this: fulllocalpath1:=fullremotepath1;fulllocalpath2:=fullremotepath2, where a local file can be copied to the specified remote location on the remote system.

Remote Pre-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the pre-build event tool to run on the remote system.
Description	Specifies a description for the pre-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy from the remote system. Optionally the list can be provided as a remote to local mapping pairs using a syntax like this: fullremotepath1:=fulllocalpath1;fullremotepath2:=fulllocalpath2, where a remote file can be copied to the specified location on the local machine.

Remote Post-Build Event

PROPERTY	DESCRIPTION
Command Line	Specifies a command line for the post-build event tool to run on the remote system.
Description	Specifies a description for the post-build event tool to display.
Use In Build	Specifies whether this build event is excluded from the build for the current configuration.
Additional files to copy	Specifies additional files to copy from the remote system. Optionally the list can be provided as a remote to local mapping pairs using a syntax like this: fullremotepath1:=fulllocalpath1;fullremotepath2:=fulllocalpath2, where a remote file can be copied to the specified location on the local machine.

C/C++

IntelliSense

The IntelliSense properties can be set at the project or file level to provide clues to the IntelliSense engine. They do not affect compilation.

PROPERTY	DESCRIPTION
Include Search Path	Specifies the include search path for resolving included files.
Forced Includes	Specifies the files that are forced included.
Preprocessor Definitions	Specifies the preprocessor defines used by the source files.
Undefine Preprocessor Definitions	Specifies one or more preprocessor undefines. (/U[macro])
Additional Options	Specifies additional compiler switches to be used by IntelliSense when parsing C++ files.

Build

PROPERTY	DESCRIPTION
Build Command Line	Specifies the command line to run for the 'Build' command.
Rebuild All Command Line	Specifies the command line to run for the 'Rebuild All' command.
Clean Command Line	Specifies the command line to run for the 'Clean' command.

Remote Build

PROPERTY	DESCRIPTION
Build Command Line	Specifies the command line to run for the 'Build' command. This is executed on the remote system.

PROPERTY	DESCRIPTION
Rebuild All Command Line	Specifies the command line to run for the 'Rebuild All' command. This is executed on the remote system.
Clean Command Line	Specifies the command line to run for the 'Clean' command. This is executed on the remote system.
Outputs	Specifies the outputs generated by the remote build on the remote system.

Remote Archive Properties (C++ Linux)

9/21/2022 • 2 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later.

PROPERTY	DESCRIPTION
Create an archive index	Create an archive index (as done by ranlib). This option can speed up linking and reduce dependency within its own library.
Create Thin Archive	Create a thin archive. A thin archive contains relative paths to the objects instead of embedding the objects. Switching between Thin and Normal requires deleting the existing library.
No Warning on Create	Doesn't warn if or when the library is created.
Truncate Timestamp	Use zero for timestamps and uids/gids.
Suppress Startup Banner	Don't show the version number.
Verbose	Verbose
Additional Options	Additional options.
Output File	The /OUT option overrides the default name and location of the program that the lib creates.
Archiver	Specifies the program to invoke during linking of static objects, or the path to the archiver on the remote system.
Archiver Timeout	Remote archiver timeout, in milliseconds.
Copy Output	Specifies whether to copy the build output file from the remote system to the local machine.

Create a CMake Linux project in Visual Studio

9/21/2022 • 3 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later. To see the documentation for these versions, set the **Version** drop-down located above the table of contents to **Visual Studio 2017** or **Visual Studio 2019**.

We recommend you use CMake for projects that are cross-platform or will be made open-source. You can use CMake projects to build and debug the same source code on Windows, the Windows Subsystem for Linux (WSL), and remote systems.

Before you begin

First, make sure you have the Visual Studio Linux workload installed, including the CMake component. That's the **Linux development with C++** workload in the Visual Studio installer. See Install the C++ Linux workload in Visual Studio if you aren't sure you have that installed.

Also, make sure the following are installed on the remote machine:

- gcc
- gdb
- rsync
- zip
- ninja-build (Visual Studio 2019 or above)

The CMake support in Visual Studio requires server mode support introduced in CMake 3.8. For a Microsoft-provided CMake variant, download the latest prebuilt binaries at https://github.com/Microsoft/CMake/releases.

The binaries are installed in `~/.vs/cmake`. After deploying the binaries, your project automatically regenerates. If the CMake specified by the `cmakeExecutable` field in *CMakeSettings.json* is invalid (it doesn't exist or is an unsupported version), and the prebuilt binaries are present, Visual Studio ignores `cmakeExecutable` and uses the prebuilt binaries.

Visual Studio 2017 can't create a CMake project from scratch, but you can open a folder that contains an existing CMake project, as described in the next section.

You can use Visual Studio 2019 to build and debug on a remote Linux system or WSL, and CMake will be invoked on that system. Cmake version 3.14 or later should be installed on the target machine.

Make sure that the target machine has a recent version of CMake. Often, the version offered by a distribution's default package manager isn't recent enough to support all the features required by Visual Studio. Visual Studio 2019 detects whether a recent version of CMake is installed on the Linux system. If none is found, Visual Studio shows an info-bar at the top of the editor pane. It offers to install CMake for you from https://github.com/Microsoft/CMake/releases.

With Visual Studio 2019, you can create a CMake project from scratch, or open an existing CMake project. To create a new CMake project, follow the instructions below. Or skip ahead to Open a CMake project folder if you already have a CMake project.

Create a new Linux CMake project

To create a new Linux CMake project in Visual Studio 2019:

1. Select **File** > **New Project** in Visual Studio, or press **Ctrl** + **Shift** + **N**.

2. Set the **Language** to **C++** and search for "CMake". Then choose **Next**. Enter a **Name** and **Location**, and choose **Create**.

Alternatively, you can open your own CMake project in Visual Studio 2019. The following section explains how.

Visual Studio creates a minimal *CMakeLists.txt* file with only the name of the executable and the minimum CMake version required. You can manually edit this file however you like; Visual Studio will never overwrite your changes.

To help you make sense of, edit, and author your CMake scripts in Visual Studio 2019, refer to the following resources:

- In-editor documentation for CMake in Visual Studio
- Code navigation for CMake scripts
- Easily Add, Remove, and Rename Files and Targets in CMake Projects

Open a CMake project folder

When you open a folder that contains an existing CMake project, Visual Studio uses variables in the CMake cache to automatically configure IntelliSense and builds. Local configuration and debugging settings get stored in JSON files. You can optionally share these files with others who are using Visual Studio.

Visual Studio doesn't modify the *CMakeLists.txt* files. This allows others working on the same project to continue to use their existing tools. Visual Studio does regenerate the cache when you save edits to *CMakeLists.txt*, or in some cases, to *CMakeSettings.json*. If you're using an **Existing Cache** configuration, then Visual Studio doesn't modify the cache.

For general information about CMake support in Visual Studio, see CMake projects in Visual Studio. Read that before continuing here.

To get started, choose **File** > **Open** > **Folder** from the main menu or else type `devenv.exe <foldername>` in a developer command prompt window. The folder you open should have a *CMakeLists.txt* file in it, along with your source code.

The following example shows a simple *CMakeLists.txt* file and .cpp file:

```
// hello.cpp

#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello from Linux CMake \n";
}
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.8)
project (hello-cmake)
add_executable(hello-cmake hello.cpp)
```

Next steps

Configure a Linux CMake project

See also

Configure a Linux CMake project in Visual Studio

9/21/2022 • 4 minutes to read • Edit Online

Linux support is available in Visual Studio 2017 and later. To see the documentation for these versions, set the **Version** drop-down located above the table of contents to **Visual Studio 2017** or **Visual Studio 2019**.

This topic describes how to add a Linux configuration to a CMake project that targets either a remote Linux system or Windows Subsystem for Linux (WSL). It continues the series that began with Create a Linux CMake project in Visual Studio. If you're using MSBuild, instead, see Configure a Linux MSBuild Project in Visual Studio

Add a Linux configuration

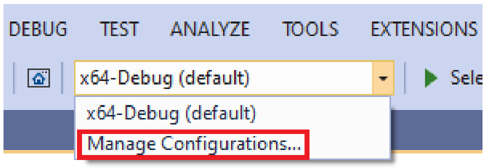
A configuration can be used to target different platforms (Windows, WSL, a remote system) with the same source code. A configuration is also used to set your compilers, pass environment variables, and customize how CMake is invoked. The *CMakeSettings.json* file specifies some or all of the properties listed in Customize CMake settings, plus other properties that control the build settings on the remote Linux machine.

To change the default CMake settings in Visual Studio 2017, choose **CMake** > **Change CMake Settings** > **CMakeLists.txt** from the main menu. Or, right-click *CMakeLists.txt* in **Solution Explorer** and choose **Change CMake Settings**. Visual Studio then creates a new *CMakeSettings.json* file in your root project folder. To make changes, open the file and modify it directly. For more information, see Customize CMake settings.

The default configuration for Linux-Debug in Visual Studio 2017 (and Visual Studio 2019 version 16.0) looks like this:

```
{
  "configurations": [
    {
      "name": "Linux-Debug",
      "generator": "Unix Makefiles",
      "remoteMachineName": "${defaultRemoteMachineName}",
      "configurationType": "Debug",
      "remoteCMakeListsRoot": "/var/tmp/src/${workspaceHash}/${name}",
      "cmakeExecutable": "/usr/local/bin/cmake",
      "buildRoot": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "remoteBuildRoot": "/var/tmp/build/${workspaceHash}/build/${name}",
      "remoteInstallRoot": "/var/tmp/build/${workspaceHash}/install/${name}",
      "remoteCopySources": true,
      "remoteCopySourcesOutputVerbosity": "Normal",
      "remoteCopySourcesConcurrentCopies": "10",
      "remoteCopySourcesMethod": "rsync",
      "remoteCopySourcesExclusionList": [
        ".vs",
        ".git"
      ],
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [
        "linux_x64"
      ]
    }
  ]
}
```

To change the default CMake settings in Visual Studio 2019 or later, from the main toolbar, open the **Configuration** dropdown and choose **Manage Configurations**.



This command opens the **CMake Settings Editor**, which you can use to edit the *CMakeSettings.json* file in your root project folder. You can also open the file with the JSON editor by clicking the **Edit JSON** button in the upper-right of the **CMake Settings** dialog. For more information, see Customize CMake Settings.

The default Linux-Debug configuration in Visual Studio 2019 version 16.1, and later, looks like this:

```
{
  "configurations": [
    {
      "name": "Linux-GCC-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "cmakeExecutable": "cmake",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_x64" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCMakeListsRoot": "$HOME/.vs/${projectDirName}/${workspaceHash}/src",
      "remoteBuildRoot": "$HOME/.vs/${projectDirName}/${workspaceHash}/out/build/${name}",
      "remoteInstallRoot": "$HOME/.vs/${projectDirName}/${workspaceHash}/out/install/${name}",
      "remoteCopySources": true,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync",
      "variables": []
    }
  ]
}
```

In Visual Studio 2019 version 16.6 or later, Ninja is the default generator for configurations targeting a remote system or WSL, as opposed to Unix Makefiles. For more information, see this post on the C++ Team Blog.

For more information about these settings, see CMakeSettings.json reference.

When you do a build:

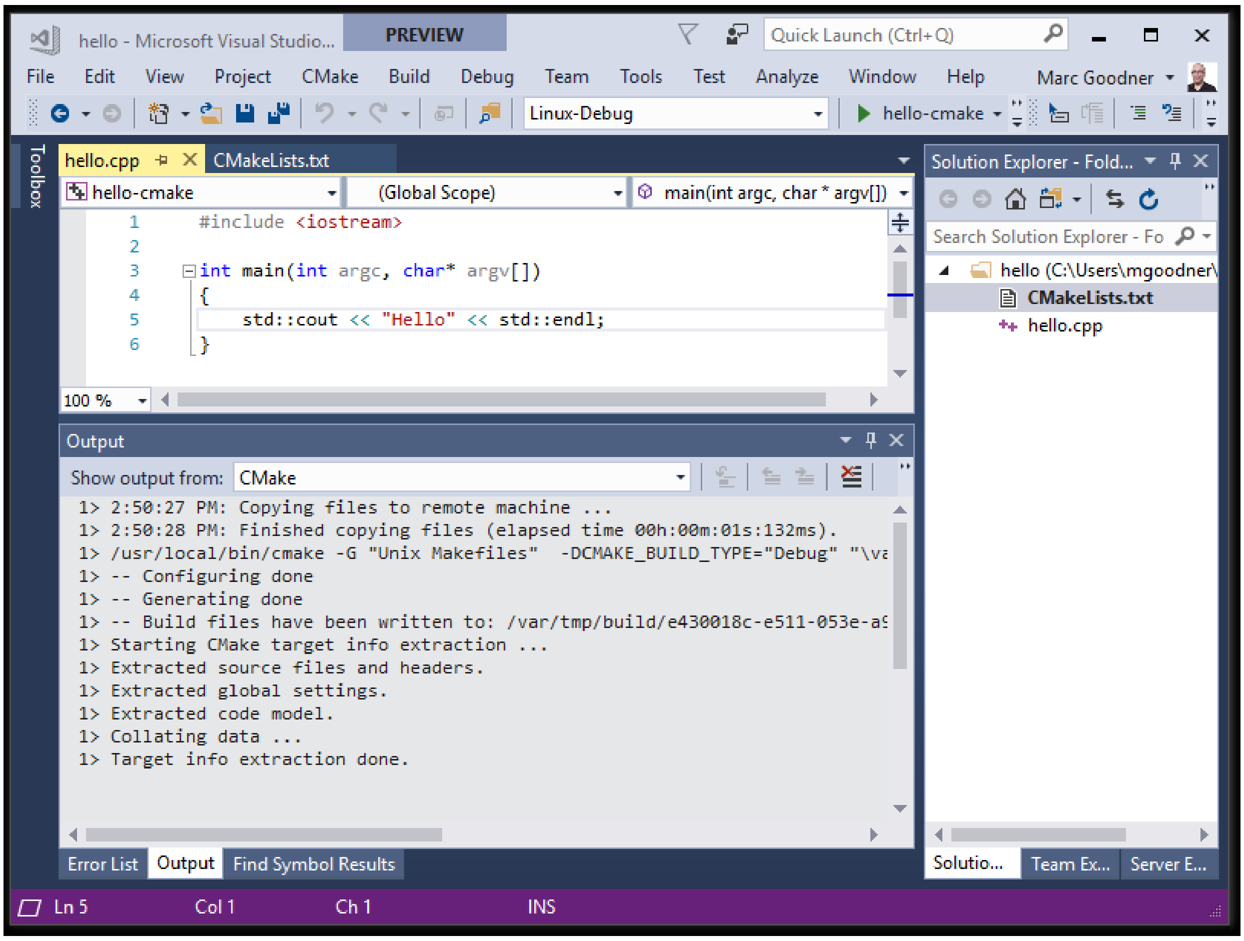
- If you're targeting a remote system, Visual Studio chooses the first remote system in the list under **Tools** > **Options** > **Cross Platform** > **Connection Manager** by default for remote targets.
- If no remote connections are found, you're prompted to create one. For more information, see Connect to your remote Linux computer.

Choose a Linux target

When you open a CMake project folder, Visual Studio parses the *CMakeLists.txt* file, and specifies a Windows target of **x86-Debug**. To target a remote Linux system, you'll change the project settings based on your Linux compiler. For example, if you're using GCC on Linux and compiling with debug info, choose: **Linux-GCC-Debug** or **Linux-GCC-Release**.

If you specify a remote Linux target, your source is copied to the remote system.

After you select a target, CMake runs automatically on the Linux system to generate the CMake cache for your project:



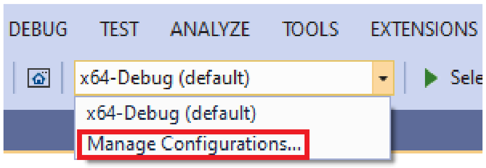
```
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello" << std::endl;
}
```

```
1> 2:50:27 PM: Copying files to remote machine ...
1> 2:50:28 PM: Finished copying files (elapsed time 00h:00m:01s:132ms).
1> /usr/local/bin/cmake -G "Unix Makefiles"  -DCMAKE_BUILD_TYPE="Debug" "\va
1> -- Configuring done
1> -- Generating done
1> -- Build files have been written to: /var/tmp/build/e430018c-e511-053e-a
1> Starting CMake target info extraction ...
1> Extracted source files and headers.
1> Extracted global settings.
1> Extracted code model.
1> Collating data ...
1> Target info extraction done.
```

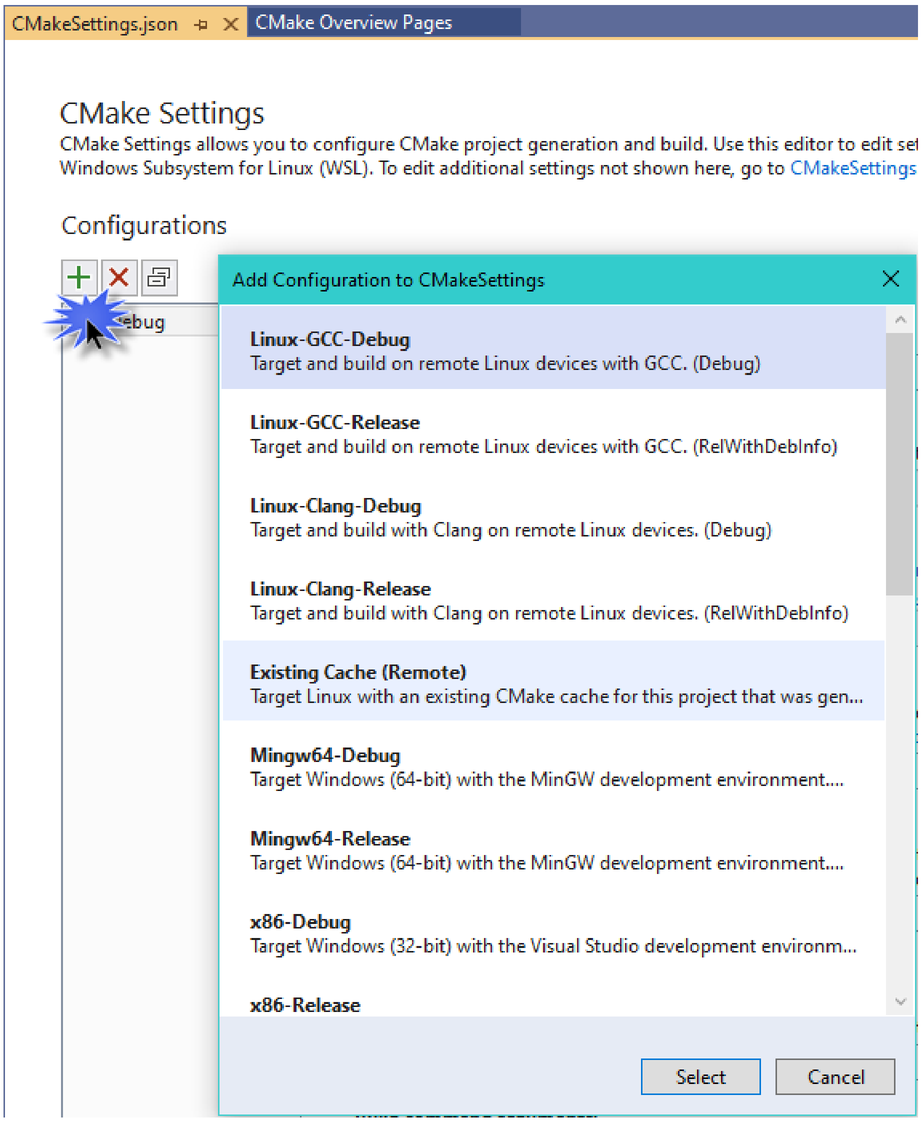
Target Windows Subsystem for Linux

If you're targeting Windows Subsystem for Linux (WSL), you don't need to add a remote connection.

To target WSL, select **Manage Configurations** in the configuration dropdown in the main toolbar:



The **CMakeSettings.json** window appears.



CMake Settings

CMake Settings allows you to configure CMake project generation and build. Use this editor to edit se
Windows Subsystem for Linux (WSL). To edit additional settings not shown here, go to CMakeSettings

Add Configuration to CMakeSettings

Linux-GCC-Debug
Target and build on remote Linux devices with GCC. (Debug)

Linux-GCC-Release
Target and build on remote Linux devices with GCC. (RelWithDebInfo)

Linux-Clang-Debug
Target and build with Clang on remote Linux devices. (Debug)

Linux-Clang-Release
Target and build with Clang on remote Linux devices. (RelWithDebInfo)

Existing Cache (Remote)
Target Linux with an existing CMake cache for this project that was gen...

Mingw64-Debug
Target Windows (64-bit) with the MinGW development environment....

Mingw64-Release
Target Windows (64-bit) with the MinGW development environment....

x86-Debug
Target Windows (32-bit) with the Visual Studio development environm...

x86-Release

Press **Add Configuration** (the green '+' button) and then choose **Linux-GCC-Debug** or **Linux-GCC-Release** if using GCC. Use the Clang variants if you're using the Clang/LLVM toolset. Press **Select** and then **Ctrl+S** to save the configuration.

Visual Studio 2019 version 16.1 When you target WSL, Visual Studio doesn't need to copy source files and maintain two synchronous copies of your build tree because the compiler on Linux has direct access to your source files in the mounted Windows file system.

IntelliSense

Accurate C++ IntelliSense requires access to the C++ headers referenced by your C++ source files. Visual Studio automatically uses the headers referenced by a CMake project from Linux to Windows to provide a full-fidelity IntelliSense experience. For more information, see IntelliSense for remote headers.

Locale setting

Visual Studio language settings aren't propagated to Linux targets because Visual Studio doesn't manage or configure installed packages. Messages shown in the Output window, such as build errors, are shown using the language and locale of the Linux target. You'll need to configure your Linux targets for the desired locale.

More settings

Use the following settings to run commands on the Linux system before and after building, and before CMake generation. The values can be any command that is valid on the remote system. The output is piped back to Visual Studio.

```
{
      "remotePrebuildCommand": "",
      "remotePreGenerateCommand": "",
      "remotePostbuildCommand": "",
}
```

Next steps

Configure CMake debugging sessions

See also

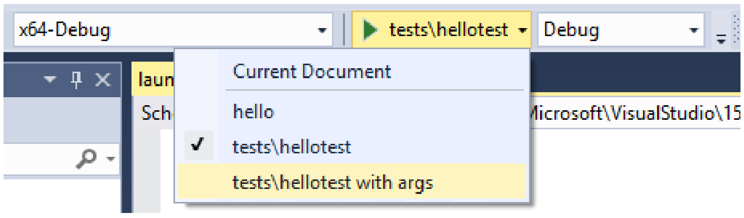
Working with project properties
Customize CMake settings
CMake predefined configuration reference

Configure CMake debugging sessions

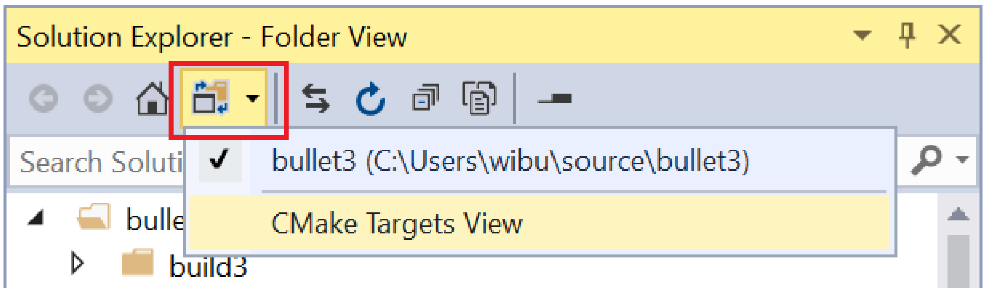
9/21/2022 • 9 minutes to read • Edit Online

Native CMake support is available in Visual Studio 2017 and later. To see the documentation for these versions, set the Visual Studio **Version** selector control for this article to Visual Studio 2017 or later. It's found at the top of the table of contents on this page.

All executable CMake targets are shown in the **Startup Item** dropdown in the toolbar. Select one to start a debugging session and launch the debugger.



You can also start a debug session from Solution Explorer. First, switch to **CMake Targets View** in the **Solution Explorer** window.

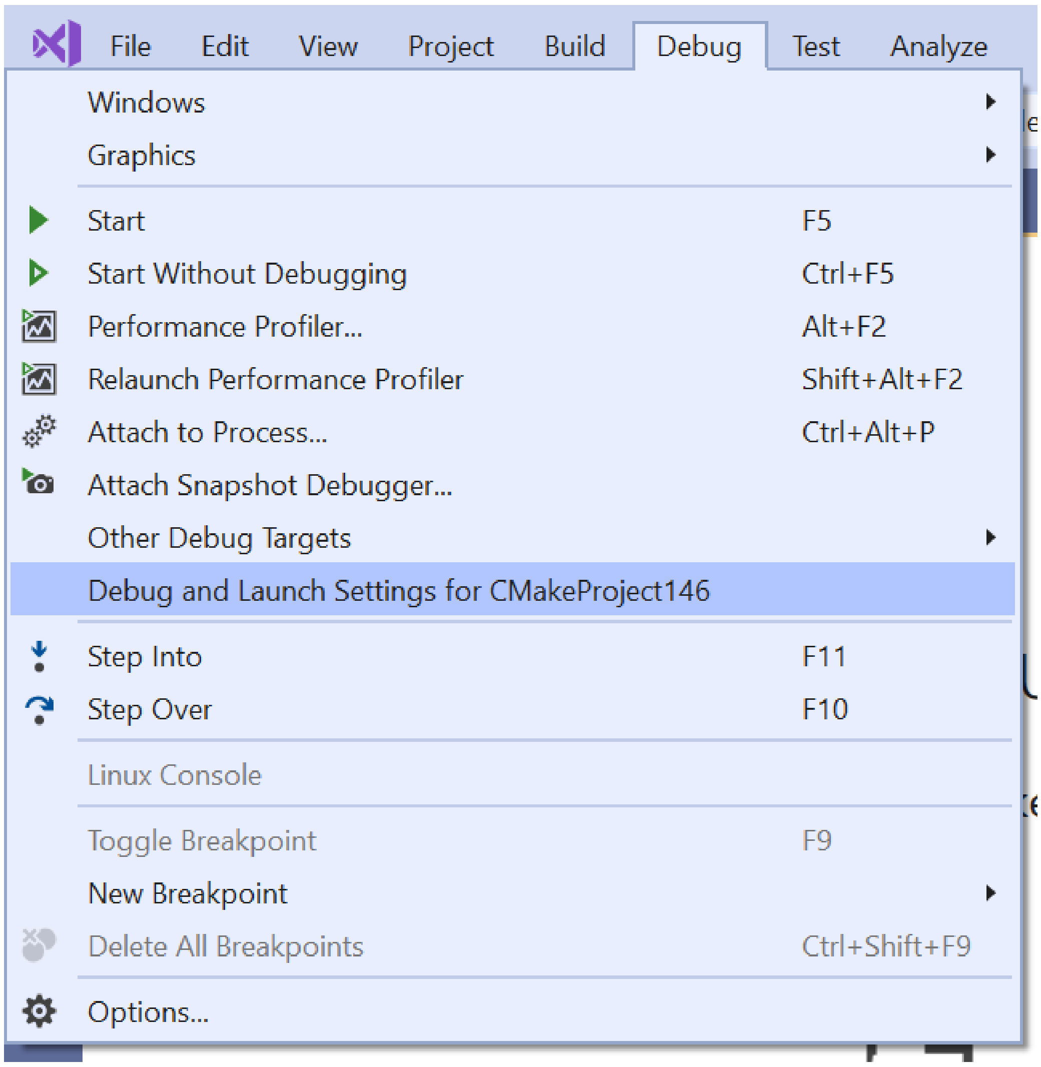

Solution Explorer - Folder View
Search Soluti
bullet3 (C:\Users\wibu\source\bullet3)
CMake Targets View
bulle
build3

Then, right-click on an executable and select **Debug**. This command automatically starts debugging the selected target based on your active configuration.

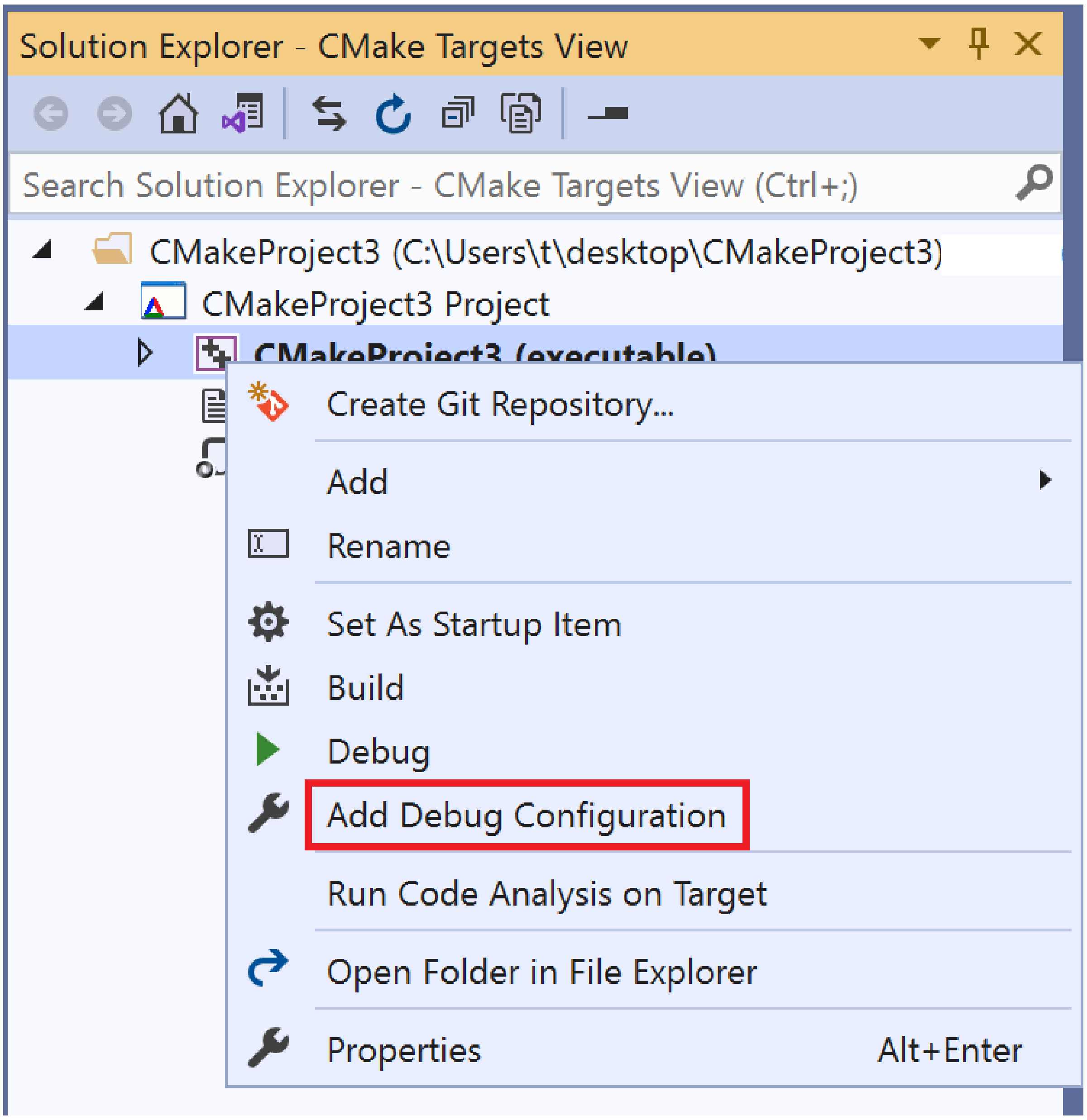
Customize debugger settings

You can customize the debugger settings for any executable CMake target in your project. They're found in a configuration file called *launch.vs.json*, located in a `.vs` folder in your project root. A launch configuration file is useful in most debugging scenarios, because you can configure and save your debugging setup details. There are three entry points to this file:

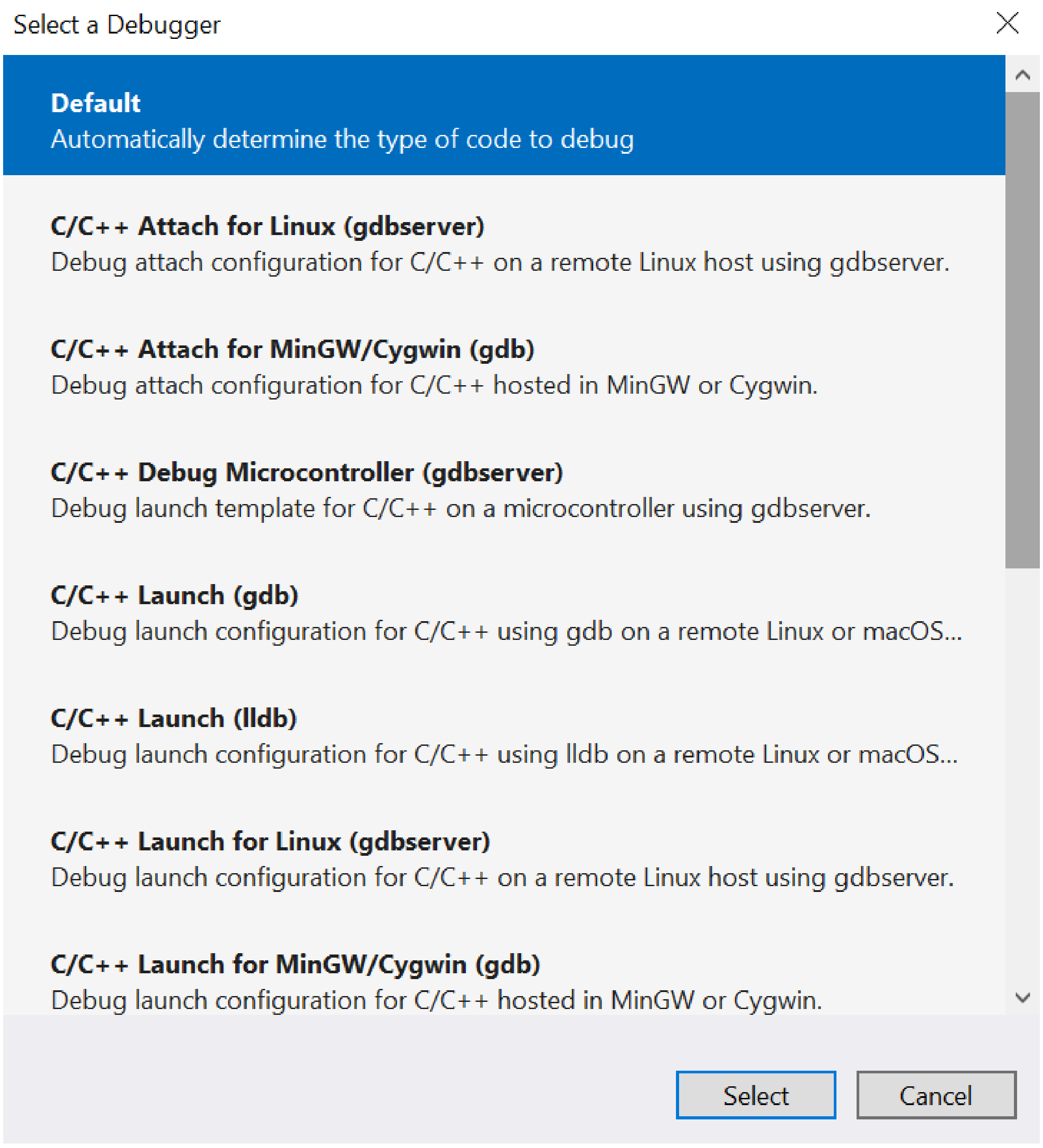
- **Debug Menu**: Select **Debug > Debug and Launch Settings for ${activeDebugTarget}** from the main menu to customize the debug configuration specific to your active debug target. If you don't have a debug target selected, this option is grayed out.


File
Edit
View
Project
Build
Debug
Test
Analyze
Windows
Graphics
Start F5
Start Without Debugging Ctrl+F5
Performance Profiler... Alt+F2
Relaunch Performance Profiler Shift+Alt+F2
Attach to Process... Ctrl+Alt+P
Attach Snapshot Debugger...
Other Debug Targets
Debug and Launch Settings for CMakeProject146
Step Into F11
Step Over F10
Linux Console
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Options...

- **Targets View**: Navigate to **Targets View** in Solution Explorer. Then, right-click on a debug target and select **Add Debug Configuration** to customize the debug configuration specific to the selected target.


Solution Explorer - CMake Targets View
Search Solution Explorer - CMake Targets View (Ctrl+;)
CMakeProject3 (C:\Users\t\desktop\CMakeProject3)
CMakeProject3 Project
CMakeProject3 (executable)
Create Git Repository...
Add
Rename
Set As Startup Item
Build
Debug
Add Debug Configuration
Run Code Analysis on Target
Open Folder in File Explorer
Properties
Alt+Enter

- **Root CMakeLists.txt:** Right-click on a root *CMakeLists.txt* and select **Add Debug Configuration** to open the **Select a Debugger** dialog box. The dialog allows you to add *any* type of debug configuration, but you must manually specify the CMake target to invoke via the `projectTarget` property.


Select a Debugger

Default
Automatically determine the type of code to debug

C/C++ Attach for Linux (gdbserver)
Debug attach configuration for C/C++ on a remote Linux host using gdbserver.

C/C++ Attach for MinGW/Cygwin (gdb)
Debug attach configuration for C/C++ hosted in MinGW or Cygwin.

C/C++ Debug Microcontroller (gdbserver)
Debug launch template for C/C++ on a microcontroller using gdbserver.

C/C++ Launch (gdb)
Debug launch configuration for C/C++ using gdb on a remote Linux or macOS...

C/C++ Launch (lldb)
Debug launch configuration for C/C++ using lldb on a remote Linux or macOS...

C/C++ Launch for Linux (gdbserver)
Debug launch configuration for C/C++ on a remote Linux host using gdbserver.

C/C++ Launch for MinGW/Cygwin (gdb)
Debug launch configuration for C/C++ hosted in MinGW or Cygwin.

You can edit the *launch.vs.json* file to create debug configurations for any number of CMake targets. When you save the file, Visual Studio creates an entry for each new configuration in the **Startup Item** dropdown.

Reference keys in CMakeSettings.json

To reference any key in a *CMakeSettings.json* file, prepend `cmake.` to it in *launch.vs.json*. The following example shows a simple *launch.vs.json* file that pulls in the value of the `remoteCopySources` key in the *CMakeSettings.json* file for the currently selected configuration:

```
{
  "version": "0.2.1",
  "configurations": [
    {
      "type": "default",
      "project": "CMakeLists.txt",
      "projectTarget": "CMakeHelloWorld.exe (Debug\\CMakeHelloWorld.exe)",
      "name": "CMakeHelloWorld.exe (Debug\\CMakeHelloWorld.exe)",
      "args": ["${cmake.remoteCopySources}"]
    }
  ]
}
```

Environment variables defined in *CMakeSettings.json* can also be used in launch.vs.json using the syntax `${env.VARIABLE_NAME}`. In Visual Studio 2019 version 16.4 and later, debug targets are automatically launched using the environment you specify in *CMakeSettings.json*. You can unset an environment variable by setting it to **null**.

Launch.vs.json reference

There are many *launch.vs.json* properties to support all your debugging scenarios. The following properties are common to all debug configurations, both remote and local:

- `projectTarget`: Specifies the CMake target to invoke when building the project. Visual Studio autopopulates this property if you enter *launch.vs.json* from the **Debug Menu** or **Targets View**. This value must match the name of an existing debug target listed in the **Startup Item** dropdown.

- `env`: Additional environment variables to add using the syntax:

  ```
  "env": {
        "DEBUG_LOGGING_LEVEL": "trace;info",
        "ENABLE_TRACING": "true"
      }
  ```

- `args`: Command-line arguments passed to the program to debug.

Launch.vs.json reference for remote projects and WSL

In Visual Studio 2019 version 16.6, we added a new debug configuration of `type: cppgdb` to simplify debugging on remote systems and WSL. Old debug configurations of `type: cppdbg` are still supported.

Configuration type `cppgdb`

- `name`: A friendly name to identify the configuration in the **Startup Item** dropdown.
- `project`: Specifies the relative path to the project file. Normally, you don't need to change this path when debugging a CMake project.
- `projectTarget`: Specifies the CMake target to invoke when building the project. Visual Studio autopopulates this property if you enter *launch.vs.json* from the **Debug Menu** or **Targets View**. This target value must match the name of an existing debug target listed in the **Startup Item** dropdown.
- `debuggerConfiguration`: Indicates which set of debugging default values to use. In Visual Studio 2019 version 16.6, the only valid option is `gdb`. Visual Studio 2019 version 16.7 or later also supports `gdbserver`.
- `args`: Command-line arguments passed on startup to the program being debugged.
- `env`: Additional environment variables passed to the program being debugged. For example, `{"DISPLAY": "0.0"}`.
- `processID`: Linux process ID to attach to. Only used when attaching to a remote process. For more information, see Troubleshoot attaching to processes using GDB.

Additional options for the `gdb` configuration

- `program`: Defaults to `"${debugInfo.fullTargetPath}"`. The Unix path to the application to debug. Only required if different than the target executable in the build or deploy location.
- `remoteMachineName`: Defaults to `"${debugInfo.remoteMachineName}"`. Name of the remote system that hosts the program to debug. Only required if different than the build system. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.
- `cwd`: Defaults to `"${debugInfo.defaultWorkingDirectory}"`. The Unix path to the directory on the remote system where `program` is run. The directory must exist.
- `gdbpath`: Defaults to `/usr/bin/gdb`. Full Unix path to the `gdb` used to debug. Only required if using a

custom version of `gdb`.

- `preDebugCommand`: A Linux command to run immediately before invoking `gdb`. `gdb` doesn't start until the command completes. You can use the option to run a script before the execution of `gdb`.

Additional options allowed with the `gdbserver` configuration (16.7 or later)

- `program`: Defaults to `"${debugInfo.fullTargetPath}"`. The Unix path to the application to debug. Only required if different than the target executable in the build or deploy location.

  > **TIP**
  >
  > Deploy is not yet supported for local cross-compilation scenarios. If you are cross-compiling on Windows (for example, using a cross-compiler on Windows to build a Linux ARM executable) then you'll need to manually copy the binary to the location specified by `program` on the remote ARM machine before debugging.

- `remoteMachineName`: Defaults to `"${debugInfo.remoteMachineName}"`. Name of the remote system that hosts the program to debug. Only required if different than the build system. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.

- `cwd`: Defaults to `"${debugInfo.defaultWorkingDirectory}"`. Full Unix path to the directory on the remote system where `program` is run. The directory must exist.

- `gdbPath`: Defaults to `${debugInfo.vsInstalledGdb}`. Full Windows path to the `gdb` used to debug. Defaults to the `gdb` installed with the Linux development with C/C++ workload.

- `gdbserverPath`: Defaults to `usr/bin/gdbserver`. Full Unix path to the `gdbserver` used to debug.

- `preDebugCommand`: A Linux command to run immediately before starting `gdbserver`. `gdbserver` doesn't start until the command completes.

Deployment options

Use the following options to separate your build machine (defined in CMakeSettings.json) from your remote debug machine.

- `remoteMachineName`: Remote debug machine. Only required if different than the build machine. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.
- `disableDeploy`: Defaults to `false`. Indicates whether build/debug separation is disabled. When `false`, this option allows build and debug to occur on two separate machines.
- `deployDirectory`: Full Unix path to the directory on `remoteMachineName` that the executable gets copied to.
- `deploy`: An array of advanced deployment settings. You only need to configure these settings when you want more granular control over the deployment process. By default, only the files necessary for the process to debug get deployed to the remote debug machine.
  - `sourceMachine`: The machine from which the file or directory is copied. Press **Ctrl+Space** to view a list of all the remote connections stored in the Connection Manager. When building natively on WSL, this option is ignored.
  - `targetMachine`: The machine to which the file or directory is copied. Press **Ctrl+Space** to view a list of all the remote connections stored in the Connection Manager.
  - `sourcePath`: The file or directory location on `sourceMachine`.
  - `targetPath`: The file or directory location on `targetMachine`.
  - `deploymentType`: A description of the deployment type. `LocalRemote` and `RemoteRemote` are supported. `LocalRemote` means copying from the local file system to the remote system specified by `remoteMachineName` in *launch.vs.json*. `RemoteRemote` means copying from the remote build system specified in *CMakeSettings.json* to the different remote system specified in *launch.vs.json*.
  - `executable`: Indicates whether the deployed file is an executable.

Execute custom `gdb` commands

Visual Studio supports executing custom `gdb` commands to interact with the underlying debugger directly. For more information, see Executing custom `gdb` lldb commands.

Enable logging

Enable MIEngine logging to see what commands get sent to `gdb`, what output `gdb` returns, and how long each command takes. Learn more

Configuration type `cppdbg`

The following options can be used when debugging on a remote system or WSL using the `cppdbg` configuration type. In Visual Studio 2019 version 16.6 or later, configuration type `cppgdb` is recommended.

- `name`: A friendly name to identify the configuration in the **Startup Item** dropdown.
- `project`: Specifies the relative path to the project file. Normally, you don't need to change this value when debugging a CMake project.
- `projectTarget`: Specifies the CMake target to invoke when building the project. Visual Studio autopopulates this property if you enter *launch.vs.json* from the **Debug Menu** or **Targets View**. This value must match the name of an existing debug target listed in the **Startup Item** dropdown.
- `args`: Command-line arguments passed on startup to the program being debugged.
- `processID`: Linux process ID to attach to. Only used when attaching to a remote process. For more information, see Troubleshoot attaching to processes using GDB.
- `program`: Defaults to `"${debugInfo.fullTargetPath}"`. The Unix path to the application to debug. Only required if different than the target executable in the build or deploy location.
- `remoteMachineName`: Defaults to `"${debugInfo.remoteMachineName}"`. Name of the remote system that hosts the program to debug. Only required if different than the build system. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.
- `cwd`: Defaults to `"${debugInfo.defaultWorkingDirectory}"`. Full Unix path to the directory on the remote system where `program` is run. The directory must exist.
- `environment`: Additional environment variables passed to the program being debugged. For example,

```
  "environment": [
      {
        "name": "ENV1",
        "value": "envvalue1"
      },
      {
        "name": "ENV2",
        "value": "envvalue2"
      }
    ]
```

- `pipeArgs`: An array of command-line arguments passed to the pipe program to configure the connection. The pipe program is used to relay standard input/output between Visual Studio and `gdb`. Most of this array **doesn't need to be customized** when debugging CMake projects. The exception is the `${debuggerCommand}`, which launches `gdb` on the remote system. It can be modified to:
  - Export the value of the environment variable DISPLAY on your Linux system. In the following example, this value is `:1`.

```
"pipeArgs": [
    "/s",
    "${debugInfo.remoteMachineId}",
    "/p",
    "${debugInfo.parentProcessId}",
    "/c",
    "export DISPLAY=:1;${debuggerCommand}",
    "--tty=${debugInfo.tty}"
  ],
```

  - Run a script before the execution of `gdb`. Ensure execute permissions are set on your script.

```
"pipeArgs": [
    "/s",
    "${debugInfo.remoteMachineId}",
    "/p",
    "${debugInfo.parentProcessId}",
    "/c",
    "/path/to/script.sh;${debuggerCommand}",
    "--tty=${debugInfo.tty}"
  ],
```

- `stopOnEntry`: A boolean that specifies whether to break as soon as the process is launched. The default is false.
- `visualizerFile`: A .natvis file to use when debugging this process. This option is incompatible with `gdb` pretty printing. Also set `showDisplayString` when you set this property.
- `showDisplayString`: A boolean that enables the display string when a `visualizerFile` is specified. Setting this option to `true` can cause slower performance during debugging.
- `setupCommands`: One or more `gdb` command(s) to execute, to set up the underlying debugger.
- `miDebuggerPath`: The full path to `gdb`. When unspecified, Visual Studio searches PATH first for the debugger.
- Finally, all of the deployment options defined for the `cppgdb` configuration type can be used by the `cppdbg` configuration type as well.

Debug using `gdbserver`

You can configure the `cppdbg` configuration to debug using `gdbserver`. You can find more details and a sample launch configuration in the Microsoft C++ Team Blog post Debugging Linux CMake Projects with `gdbserver`.

See also

CMake projects in Visual Studio
Configure a Linux CMake project
Connect to your remote Linux computer
Customize CMake build settings
Configure CMake debugging sessions
Deploy, run, and debug your Linux project
CMake predefined configuration reference

Tutorial: Create C++ cross-platform projects in Visual Studio

9/21/2022 • 10 minutes to read • Edit Online

Visual Studio C and C++ development isn't just for Windows anymore. This tutorial shows how to use Visual Studio for C++ cross platform development on Windows and Linux. It's based on CMake, so you don't have to create or generate Visual Studio projects. When you open a folder that contains a CMakeLists.txt file, Visual Studio configures the IntelliSense and build settings automatically. You can quickly start editing, building, and debugging your code locally on Windows. Then, switch your configuration to do the same on Linux, all from within Visual Studio.

In this tutorial, you learn how to:

- clone an open-source CMake project from GitHub
- open the project in Visual Studio
- build and debug an executable target on Windows
- add a connection to a Linux machine
- build and debug the same target on Linux

Prerequisites

- Set up Visual Studio for Cross Platform C++ Development
  - First, install Visual Studio and choose the **Desktop development with C++** and **Linux development with C++ workloads**. This minimal install is only 3 GB. Depending on your download speed, installation shouldn't take more than 10 minutes.
- Set up a Linux machine for Cross Platform C++ Development
  - Visual Studio doesn't require any specific distribution of Linux. The OS can be running on a physical machine, in a VM, or in the cloud. You could also use the Windows Subsystem for Linux (WSL). However, for this tutorial a graphical environment is required. WSL isn't recommended here, because it's intended primarily for command-line operations.
  - Visual Studio requires these tools on the Linux machine: C++ compilers, gdb, ssh, rsync, make, and zip. On Debian-based systems, you can use this command to install these dependencies:

    ```
    sudo apt install -y openssh-server build-essential gdb rsync make zip
    ```

  - Visual Studio requires a recent version of CMake on the Linux machine that has server mode enabled (at least 3.8). Microsoft produces a universal build of CMake that you can install on any Linux distro. We recommend you use this build to ensure that you have the latest features. You can get the CMake binaries from the Microsoft fork of the CMake repo on GitHub. Go to that page and download the version that matches the system architecture on your Linux machine, then mark it as an executable:

    ```
    wget <path to binary>
    chmod +x cmake-3.11.18033000-MSVC_2-Linux-x86_64.sh
    ```

  - You can see the options for running the script with `--help`. We recommend that you use the

`-prefix` option to specify installing in the **/usr** path, because **/usr/bin** is the default location where Visual Studio looks for CMake. The following example shows the Linux-x86_64 script. Change it as needed if you're using a different target platform.

```
sudo ./cmake-3.11.18033000-MSVC_2-Linux-x86_64.sh --skip-license --prefix=/usr
```

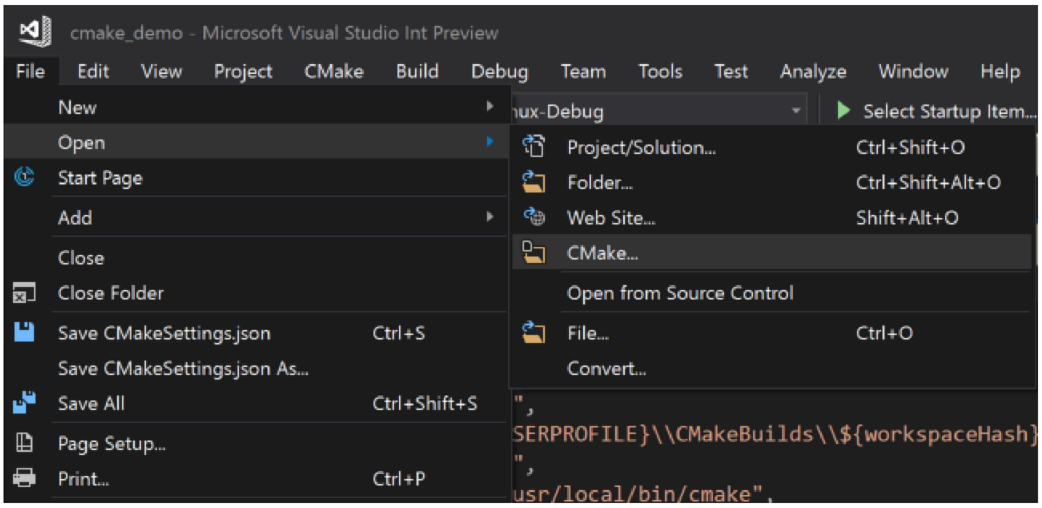
- Git for windows installed on your Windows machine.
- A GitHub account.

Clone an open-source CMake project from GitHub

This tutorial uses the Bullet Physics SDK on GitHub. It provides collision detection and physics simulations for many applications. The SDK includes sample executable programs that compile and run without having to write additional code. This tutorial doesn't modify any of the source code or build scripts. To start, clone the *bullet3* repository from GitHub on the machine where you have Visual Studio installed.

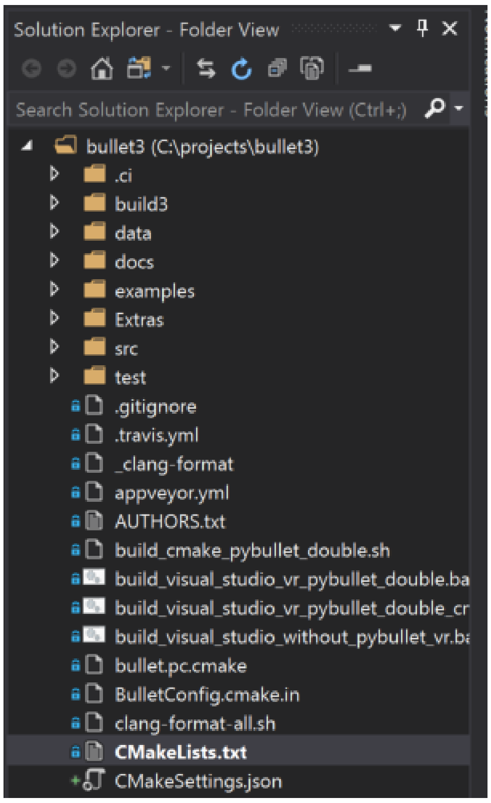
```
git clone https://github.com/bulletphysics/bullet3.git
```

1. On the Visual Studio main menu, choose **File** > **Open** > **CMake**. Navigate to the CMakeLists.txt file in the root of the bullet3 repo you just downloaded.


cmake_demo - Microsoft Visual Studio Int Preview
File Edit View Project CMake Build Debug Team Tools Test Analyze Window Help
New
Open
Start Page
Add
Close
Close Folder
Save CMakeSettings.json Ctrl+S
Save CMakeSettings.json As...
Save All Ctrl+Shift+S
Page Setup...
Print... Ctrl+P
nux-Debug
Select Startup Item...
Project/Solution... Ctrl+Shift+O
Folder... Ctrl+Shift+Alt+O
Web Site... Shift+Alt+O
CMake...
Open from Source Control
File... Ctrl+O
Convert...

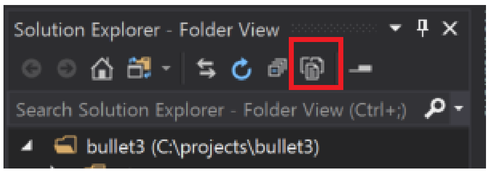
```
",
SERPROFILE}\\CMakeBuilds\\${workspaceHash}
",
usr/local/bin/cmake",
```

As soon as you open the folder, your folder structure becomes visible in the **Solution Explorer**.


Solution Explorer - Folder View
Search Solution Explorer - Folder View (Ctrl+;)
bullet3 (C:\projects\bullet3)
.ci
build3
data
docs
examples
Extras
src
test
.gitignore
.travis.yml
_clang-format
appveyor.yml
AUTHORS.txt
build_cmake_pybullet_double.sh
build_visual_studio_vr_pybullet_double.ba
build_visual_studio_vr_pybullet_double_cr
build_visual_studio_without_pybullet_vr.ba
bullet.pc.cmake
BulletConfig.cmake.in
clang-format-all.sh
CMakeLists.txt
CMakeSettings.json

This view shows you exactly what is on disk, not a logical or filtered view. By default, it doesn't show hidden files.

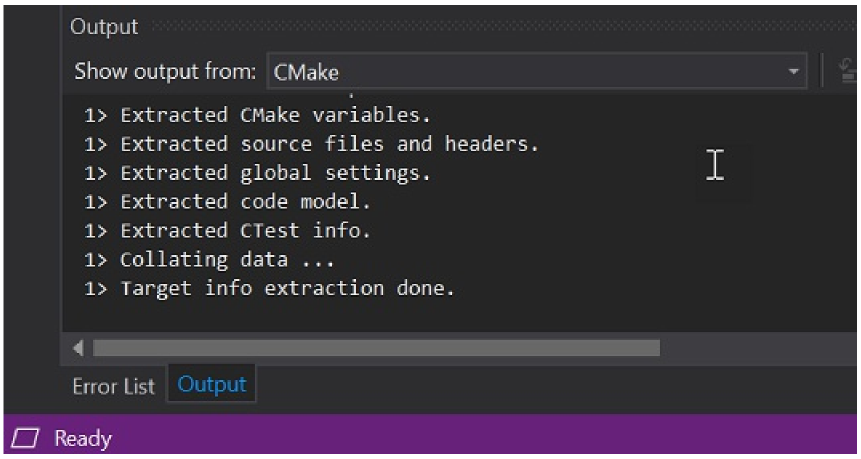
2. Choose the **Show all files** button to see all the files in the folder.


Solution Explorer - Folder View
Search Solution Explorer - Folder View (Ctrl+;)
bullet3 (C:\projects\bullet3)

Switch to targets view

When you open a folder that uses CMake, Visual Studio automatically generates the CMake cache. This operation might take a few moments, depending on the size of your project.

1. In the **Output Window**, select **Show output from** and then choose **CMake** to monitor the status of the cache generation process. When the operation is complete, it says "Target info extraction done".



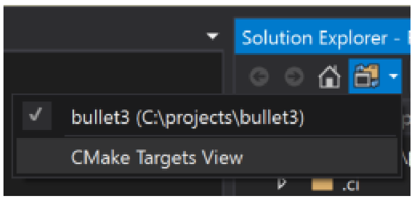
Output

Show output from: CMake

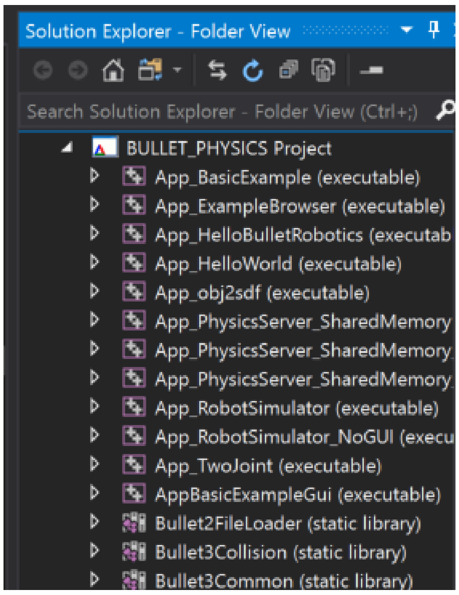
```
1> Extracted CMake variables.
1> Extracted source files and headers.
1> Extracted global settings.
1> Extracted code model.
1> Extracted CTest info.
1> Collating data ...
1> Target info extraction done.
```

After this operation completes, IntelliSense is configured. You can build the project, and debug the application. Visual Studio now shows a logical view of the solution, based on the targets specified in the CMakeLists files.

2. Use the **Solutions and Folders** button in the **Solution Explorer** to switch to CMake Targets View.



Here is what that view looks like for the Bullet SDK:


Solution Explorer - Folder View
Search Solution Explorer - Folder View (Ctrl+;)
BULLET_PHYSICS Project
App_BasicExample (executable)
App_ExampleBrowser (executable)
App_HelloBulletRobotics (executab
App_HelloWorld (executable)
App_obj2sdf (executable)
App_PhysicsServer_SharedMemory
App_PhysicsServer_SharedMemory
App_PhysicsServer_SharedMemory
App_RobotSimulator (executable)
App_RobotSimulator_NoGUI (execu
App_TwoJoint (executable)
AppBasicExampleGui (executable)
Bullet2FileLoader (static library)
Bullet3Collision (static library)
Bullet3Common (static library)

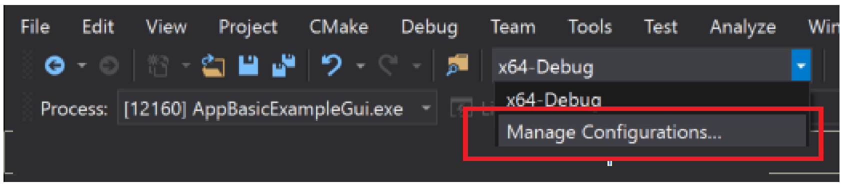
Targets view provides a more intuitive view of what is in this source base. You can see some targets are libraries and others are executables.

3. Expand a node in CMake Targets View to see its source code files, wherever those files might be located on disk.

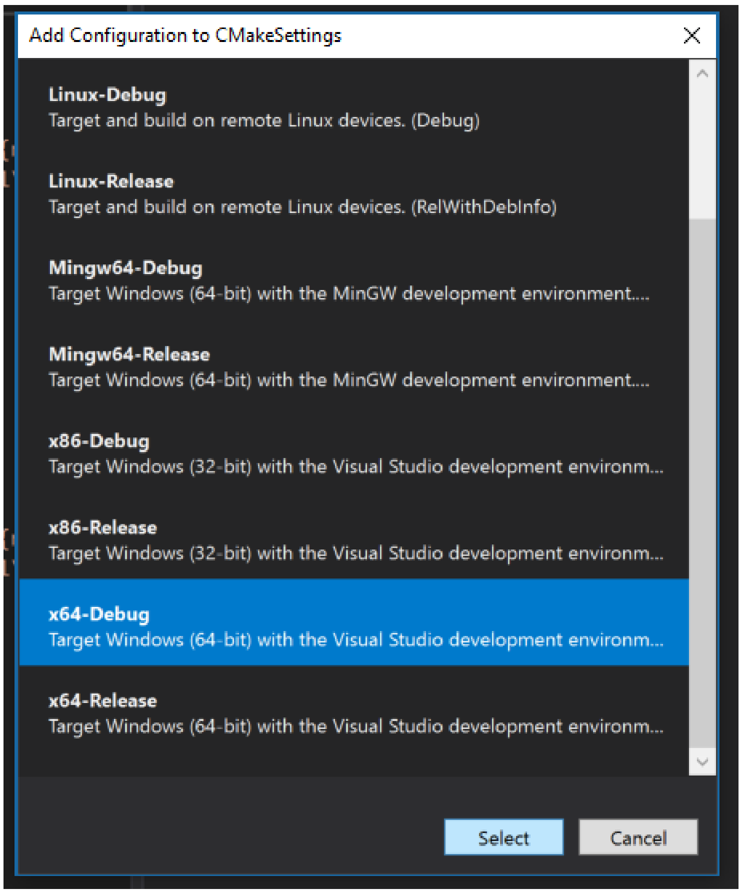
Add an explicit Windows x64-Debug configuration

Visual Studio creates a default **x64-Debug** configuration for Windows. Configurations are how Visual Studio understands what platform target it's going to use for CMake. The default configuration isn't represented on disk. When you explicitly add a configuration, Visual Studio creates a file called *CMakeSettings.json*. It's populated with settings for all the configurations you specify.

1. Add a new configuration. Open the **Configuration** drop-down in the toolbar and select **Manage Configurations**.



The CMake Settings Editor opens. Select the green plus sign on the left-hand side of the editor to add a new configuration. The **Add Configuration to CMakeSettings** dialog appears.


Add Configuration to CMakeSettings

Linux-Debug
Target and build on remote Linux devices. (Debug)

Linux-Release
Target and build on remote Linux devices. (RelWithDebInfo)

Mingw64-Debug
Target Windows (64-bit) with the MinGW development environment....

Mingw64-Release
Target Windows (64-bit) with the MinGW development environment....

x86-Debug
Target Windows (32-bit) with the Visual Studio development environm...

x86-Release
Target Windows (32-bit) with the Visual Studio development environm...

x64-Debug
Target Windows (64-bit) with the Visual Studio development environm...

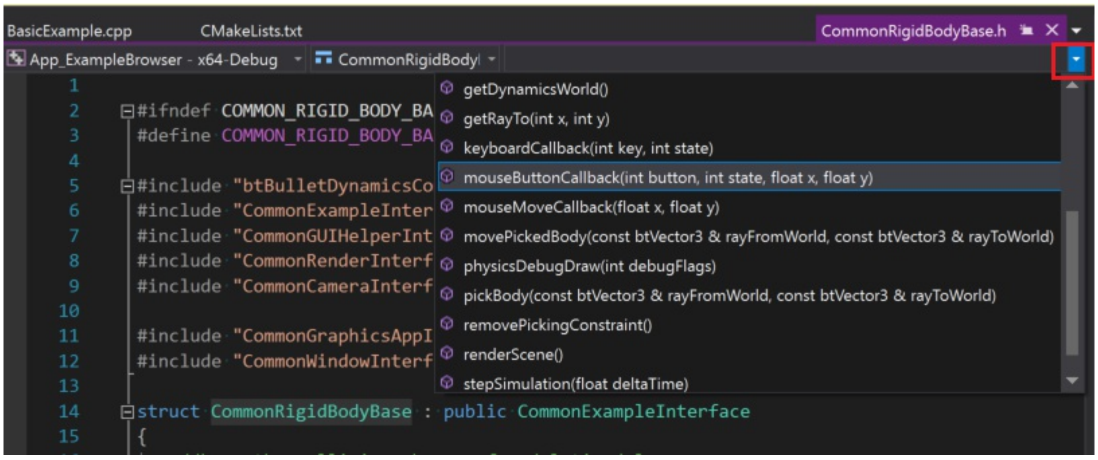
x64-Release
Target Windows (64-bit) with the Visual Studio development environm...

This dialog shows all the configurations included with Visual Studio, plus any custom configurations that you create. If you want to continue to use a **x64-Debug** configuration, that should be the first one you add. Select **x64-Debug**, and then choose the **Select** button. Visual Studio creates the CMakeSettings.json file with a configuration for **x64-Debug**, and saves it to disk. You can use whatever names you like for your configurations by changing the name parameter directly in CMakeSettings.json.

Set a breakpoint, build, and run on Windows

In this step, we'll debug an example program that demonstrates the Bullet Physics library.

1. In **Solution Explorer**, select AppBasicExampleGui and expand it.
2. Open the file `BasicExample.cpp`.
3. Set a breakpoint that gets hit when you click in the running application. The click event is handled in a method within a helper class. To quickly get there:
   a. Select `CommonRigidBodyBase` that the struct `BasicExample` is derived from. It's around line 30.
   b. Right-click and choose **Go to Definition**. Now you're in the header CommonRigidBodyBase.h.
   c. In the browser view above your source, you should see that you're in the `CommonRigidBodyBase`. To the right, you can select members to examine. Open the drop-down and select `mouseButtonCallback` to go to the definition of that function in the header.

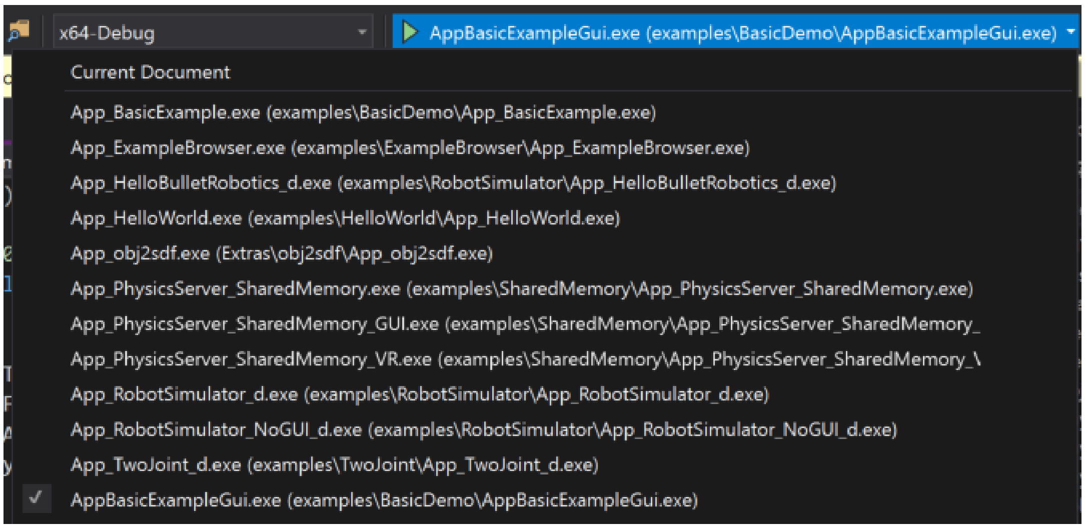


```
#ifndef COMMON_RIGID_BODY_BA
#define COMMON_RIGID_BODY_BA

#include "btBulletDynamicsCo
#include "CommonExampleInter
#include "CommonGUIHelperInt
#include "CommonRenderInterf
#include "CommonCameraInterf

#include "CommonGraphicsAppI
#include "CommonWindowInterf

struct CommonRigidBodyBase : public CommonExampleInterface
{
```

```
getDynamicsWorld()
getRayTo(int x, int y)
keyboardCallback(int key, int state)
mouseButtonCallback(int button, int state, float x, float y)
mouseMoveCallback(float x, float y)
movePickedBody(const btVector3 & rayFromWorld, const btVector3 & rayToWorld)
physicsDebugDraw(int debugFlags)
pickBody(const btVector3 & rayFromWorld, const btVector3 & rayToWorld)
removePickingConstraint()
renderScene()
stepSimulation(float deltaTime)
```

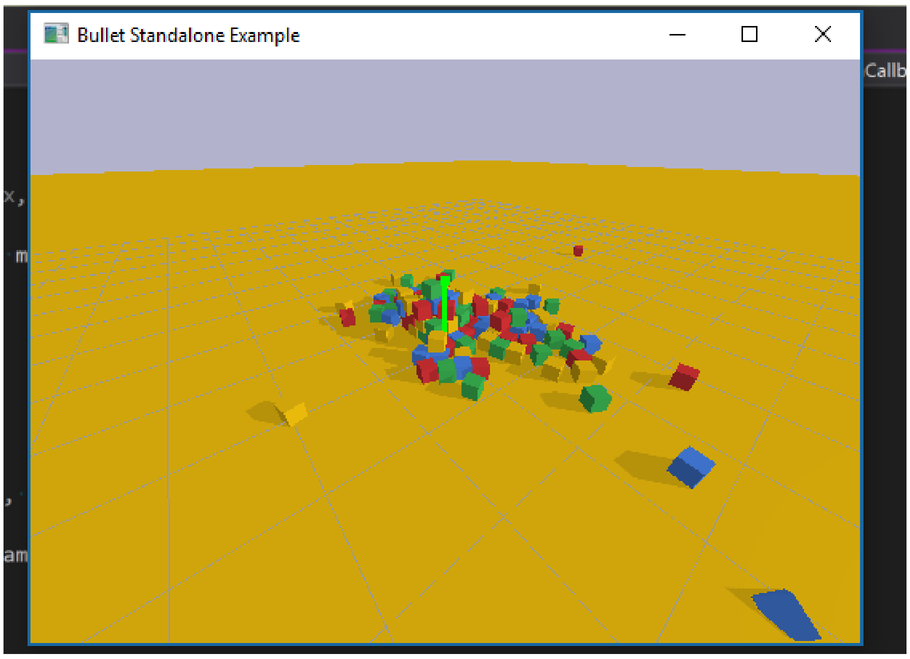
4. Place a breakpoint on the first line within this function. It gets hit when you click a mouse button within the window of the application, when run under the Visual Studio debugger.

5. To launch the application, select the launch drop-down in the toolbar. It's the one with the green play icon that says "Select Startup Item". In the drop-down, select AppBasicExampleGui.exe. The executable name now displays on the launch button:


x64-Debug
AppBasicExampleGui.exe (examples\BasicDemo\AppBasicExampleGui.exe)

```
Current Document
App_BasicExample.exe (examples\BasicDemo\App_BasicExample.exe)
App_ExampleBrowser.exe (examples\ExampleBrowser\App_ExampleBrowser.exe)
App_HelloBulletRobotics_d.exe (examples\RobotSimulator\App_HelloBulletRobotics_d.exe)
App_HelloWorld.exe (examples\HelloWorld\App_HelloWorld.exe)
App_obj2sdf.exe (Extras\obj2sdf\App_obj2sdf.exe)
App_PhysicsServer_SharedMemory.exe (examples\SharedMemory\App_PhysicsServer_SharedMemory.exe)
App_PhysicsServer_SharedMemory_GUI.exe (examples\SharedMemory\App_PhysicsServer_SharedMemory_
App_PhysicsServer_SharedMemory_VR.exe (examples\SharedMemory\App_PhysicsServer_SharedMemory_\
App_RobotSimulator_d.exe (examples\RobotSimulator\App_RobotSimulator_d.exe)
App_RobotSimulator_NoGUI_d.exe (examples\RobotSimulator\App_RobotSimulator_NoGUI_d.exe)
App_TwoJoint_d.exe (examples\TwoJoint\App_TwoJoint_d.exe)
✓ AppBasicExampleGui.exe (examples\BasicDemo\AppBasicExampleGui.exe)
```

6. Choose the launch button to build the application and necessary dependencies, then launch it with the Visual Studio debugger attached. After a few moments, the running application appears:



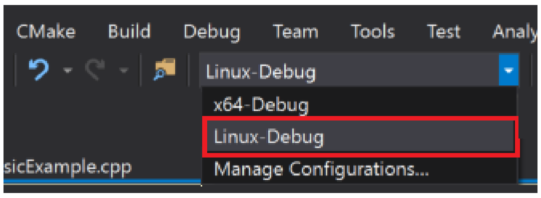
7. Move your mouse into the application window, then click a button to trigger the breakpoint. The breakpoint brings Visual Studio back to the foreground, and the editor shows the line where execution is paused. You can inspect the application variables, objects, threads, and memory, or step through your code interactively. Choose **Continue** to let the application resume, and then exit it normally. Or, halt execution within Visual Studio by using the stop button.

Add a Linux configuration and connect to the remote machine

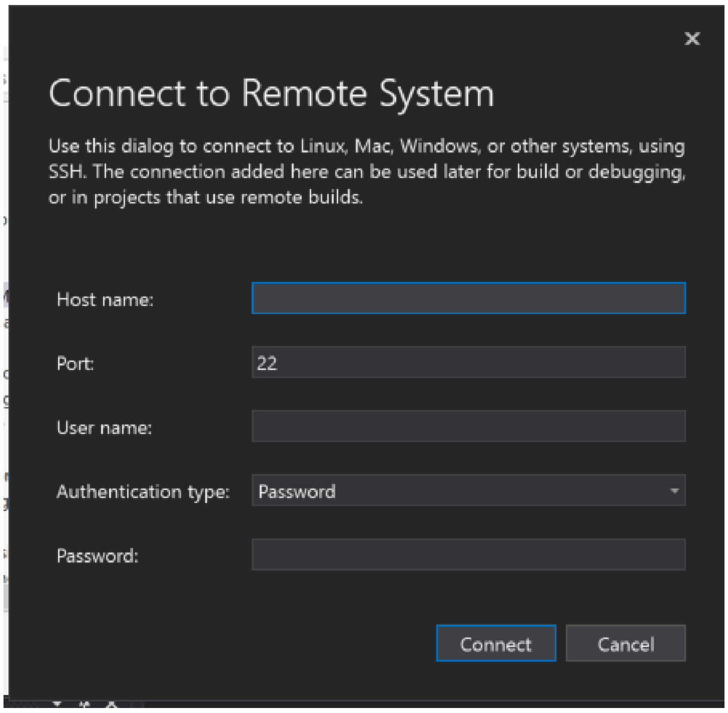
1. Add a Linux configuration. Right-click the CMakeSettings.json file in the **Solution Explorer** view and select **Add Configuration**. You see the same Add Configuration to CMakeSettings dialog as before. Select **Linux-Debug** this time, then save the CMakeSettings.json file (ctrl + s).

2. **Visual Studio 2019 version 16.6 or later** Scroll down to the bottom of the CMake Settings Editor and select **Show advanced settings**. Select **Unix Makefiles** as the **CMake generator**, then save the CMakeSettings.json file (ctrl + s).

3. Select **Linux-Debug** in the configuration drop-down.



If it's the first time you're connecting to a Linux system, the **Connect to Remote System** dialog appears.


Connect to Remote System

Use this dialog to connect to Linux, Mac, Windows, or other systems, using SSH. The connection added here can be used later for build or debugging, or in projects that use remote builds.

Host name:

Port: 22

User name:

Authentication type: Password

Password:

If you've already added a remote connection, you can open this window by navigating to **Tools > Options > Cross Platform > Connection Manager**.

4. Provide the connection information to your Linux machine and choose **Connect**. Visual Studio adds that machine as to CMakeSettings.json as your default connection for **Linux-Debug**. It also pulls down the headers from your remote machine, so you get IntelliSense specific to that remote connection. Next, Visual Studio sends your files to the remote machine and generates the CMake cache on the remote system. These steps may take some time, depending on the speed of your network and power of your remote machine. You'll know it's complete when the message "Target info extraction done" appears in the CMake output window.

Set a breakpoint, build, and run on Linux

Because it's a desktop application, you need to provide some additional configuration information to the debug configuration.

1. In the CMake Targets view, right-click AppBasicExampleGui and choose **Debug and Launch Settings** to open the launch.vs.json file that's in the hidden **.vs** subfolder. This file is local to your development environment. You can move it into the root of your project if you wish to check it in and save it with your team. In this file, a configuration has been added for AppBasicExampleGui. These default settings work in most cases, but not here. Because it's a desktop application, you need to provide some additional information to launch the program so you can see it on your Linux machine.

2. To find the value of the environment variable `DISPLAY` on your Linux machine, run this command:

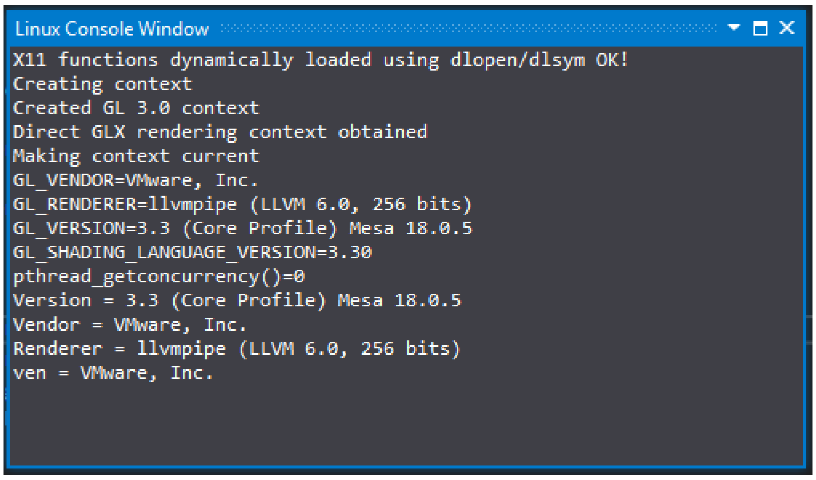
   ```
   echo $DISPLAY
   ```

   In the configuration for AppBasicExampleGui, there's a parameter array, "pipeArgs". It contains a line: "${debuggerCommand}". It's the command that launches gdb on the remote machine. Visual Studio must export the display into this context before that command runs. For example, if the value of your display is `:1`, modify that line as follows:

   ```
   "export DISPLAY=:1;${debuggerCommand}",
   ```

3. Launch and debug your application. Open the **Select Startup Item** drop-down in the toolbar and choose **AppBasicExampleGui**. Next, either choose the green play icon in the toolbar, or press **F5**. The application and its dependencies are built on the remote Linux machine, then launched with the Visual Studio debugger attached. On your remote Linux machine, you should see an application window appear.

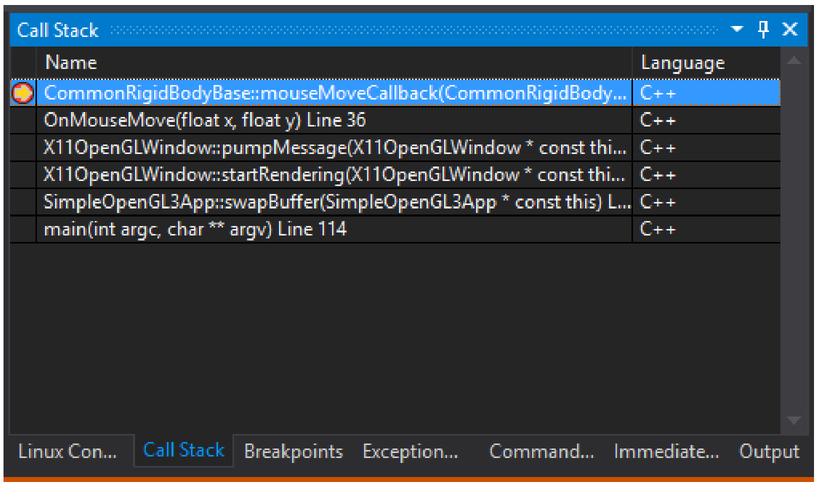
4. Move your mouse into the application window, and click a button. The breakpoint is hit. Program execution pauses, Visual Studio comes back to the foreground, and you see your breakpoint. You should also see a Linux Console Window appear in Visual Studio. The window provides output from the remote Linux machine, and it can also accept input for `stdin`. Like any Visual Studio window, you can dock it where you prefer to see it. Its position is persisted in future sessions.



```
X11 functions dynamically loaded using dlopen/dlsym OK!
Creating context
Created GL 3.0 context
Direct GLX rendering context obtained
Making context current
GL_VENDOR=VMware, Inc.
GL_RENDERER=llvmpipe (LLVM 6.0, 256 bits)
GL_VERSION=3.3 (Core Profile) Mesa 18.0.5
GL_SHADING_LANGUAGE_VERSION=3.30
pthread_getconcurrency()=0
Version = 3.3 (Core Profile) Mesa 18.0.5
Vendor = VMware, Inc.
Renderer = llvmpipe (LLVM 6.0, 256 bits)
ven = VMware, Inc.
```

5. You can inspect the application variables, objects, threads, memory, and step through your code interactively using Visual Studio. But this time, you're doing it all on a remote Linux machine instead of your local Windows environment. You can choose **Continue** to let the application resume and exit normally, or you can choose the stop button, just as with local execution.

6. Look at the Call Stack window and view the Calls to `x11OpenGLWindow` since Visual Studio launched the application on Linux.



Name	Language
CommonRigidBodyBase::mouseMoveCallback(CommonRigidBody...	C++
OnMouseMove(float x, float y) Line 36	C++
X11OpenGLWindow::pumpMessage(X11OpenGLWindow * const thi...	C++
X11OpenGLWindow::startRendering(X11OpenGLWindow * const thi...	C++
SimpleOpenGL3App::swapBuffer(SimpleOpenGL3App * const this) L...	C++
main(int argc, char ** argv) Line 114	C++

What you learned

In this tutorial, you cloned a code base directly from GitHub. You built, ran, and debugged it on Windows without modifications. Then you used the same code base, with minor configuration changes, to build, run, and debug on a remote Linux machine.

Next steps

Learn more about configuring and debugging CMake projects in Visual Studio:

CMake Projects in Visual Studio

Configure a Linux CMake project

Connect to your remote Linux computer

Customize CMake build settings

Configure CMake debugging sessions

Deploy, run, and debug your Linux project

CMake predefined configuration reference

Configure Linux projects to use Address Sanitizer

9/21/2022 • 2 minutes to read • Edit Online

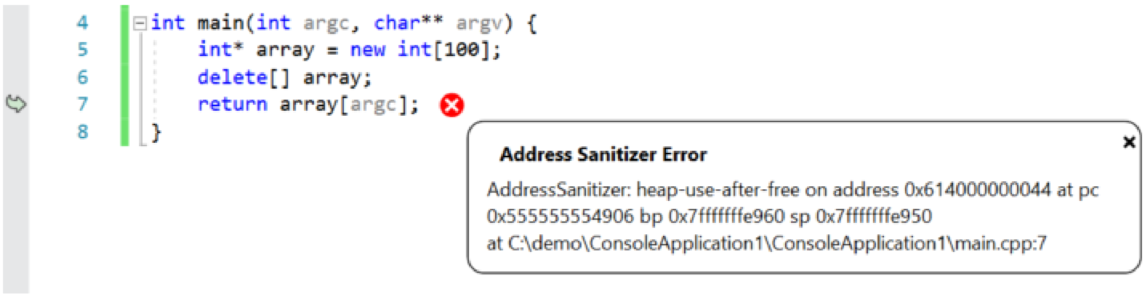
In Visual Studio 2019 version 16.1, AddressSanitizer (ASan) support is integrated into Linux projects. You can enable ASan for both MSBuild-based Linux projects and CMake projects. It works on remote Linux systems and on Windows Subsystem for Linux (WSL).

About ASan

ASan is a runtime memory error detector for C/C++ that catches the following errors:

- Use after free (dangling pointer reference)
- Heap buffer overflow
- Stack buffer overflow
- Use after return
- Use after scope
- Initialization order bugs

When ASan detects an error, it stops execution immediately. If you run an ASan-enabled program in the debugger, you see a message that describes the type of error, the memory address, and the location in the source file where the error occurred:



```
int main(int argc, char** argv) {
    int* array = new int[100];
    delete[] array;
    return array[argc];
}
```

Address Sanitizer Error

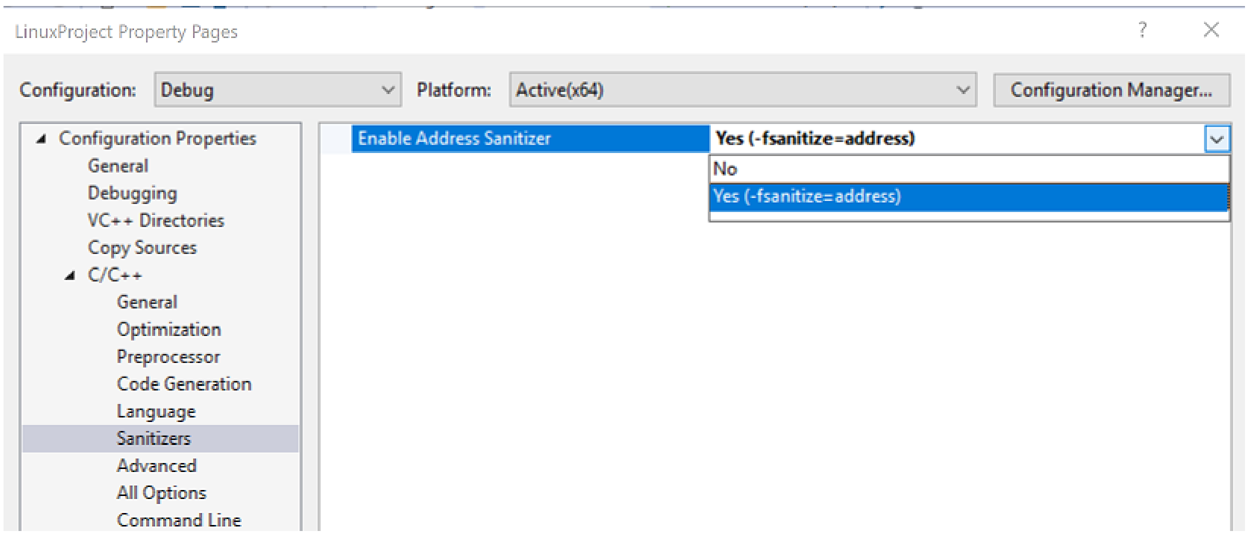
AddressSanitizer: heap-use-after-free on address 0x614000000044 at pc 0x555555554906 bp 0x7fffffffe960 sp 0x7fffffffe950
at C:\demo\ConsoleApplication1\ConsoleApplication1\main.cpp:7

You can also view the full ASan output (including where the corrupted memory was allocated/deallocated) in the Debug pane of the output window.

Enable ASan for MSBuild-based Linux projects

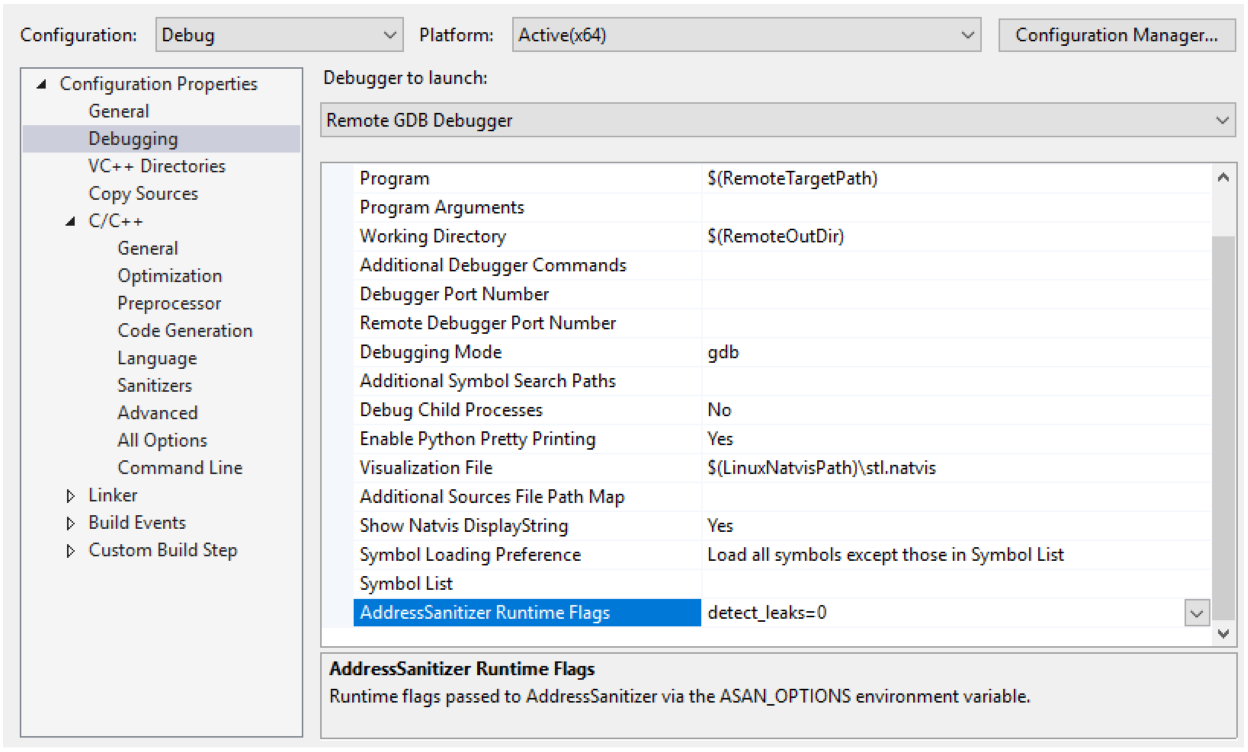
> **NOTE**
>
> Starting in Visual Studio 2019 version 16.4, AddressSanitizer for Linux projects is enabled via **Project properties** > **Configuration Properties** > **C/C++** > **Enable Address Sanitizer**.

To enable ASan for MSBuild-based Linux projects, right-click on the project in **Solution Explorer** and select **Properties**. Next, navigate to **Configuration Properties** > **C/C++** > **Sanitizers**. ASan is enabled via compiler and linker flags, and requires your project to be recompiled to work.



Enable Address Sanitizer: Yes (-fsanitize=address)

You can pass optional ASan runtime flags by navigating to **Configuration Properties** > **Debugging** > **AddressSanitizer Runtime Flags**. Click the down-arrow to add or remove flags.



Debugger to launch:

Remote GDB Debugger

Program	$(RemoteTargetPath)
Program Arguments	
Working Directory	$(RemoteOutDir)
Additional Debugger Commands	
Debugger Port Number	
Remote Debugger Port Number	
Debugging Mode	gdb
Additional Symbol Search Paths	
Debug Child Processes	No
Enable Python Pretty Printing	Yes
Visualization File	$(LinuxNatvisPath)\stl.natvis
Additional Sources File Path Map	
Show Natvis DisplayString	Yes
Symbol Loading Preference	Load all symbols except those in Symbol List
Symbol List	
AddressSanitizer Runtime Flags	detect_leaks=0

AddressSanitizer Runtime Flags
Runtime flags passed to AddressSanitizer via the ASAN_OPTIONS environment variable.

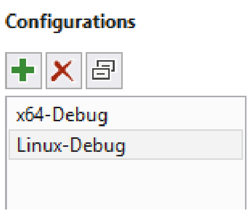
Enable ASan for Visual Studio CMake projects

NOTE

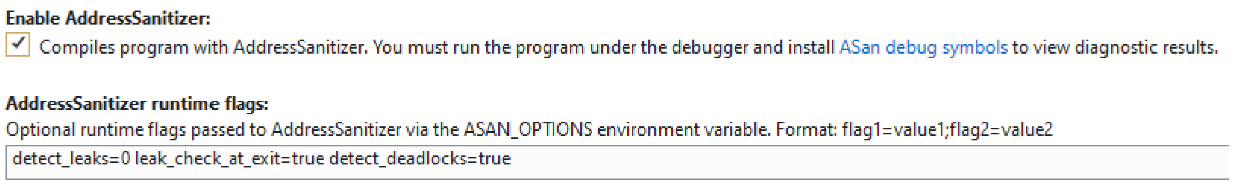
To build with CMake Presets, first enable ASan in your CMakeLists.txt file. For more information, see Enable AddressSanitizer for Windows and Linux.

To enable ASan for CMake, right-click on the `CMakeLists.txt` file in **Solution Explorer** and choose **CMake Settings for Project**.

Make sure you have a Linux configuration (for example, **Linux-Debug**) selected in the left pane of the dialog:



The ASan options are under **General**. Enter the ASan runtime flags in the format "flag=value", separated by spaces. The UI incorrectly suggests using semi-colons. Use spaces or colons to separate flags.



Enable AddressSanitizer:

✓ Compiles program with AddressSanitizer. You must run the program under the debugger and install ASan debug symbols to view diagnostic results.

AddressSanitizer runtime flags:

Optional runtime flags passed to AddressSanitizer via the ASAN_OPTIONS environment variable. Format: flag1=value1;flag2=value2

```
detect_leaks=0 leak_check_at_exit=true detect_deadlocks=true
```

Install the ASan debug symbols

To enable the ASan diagnostics, you must install its debug symbols (libasan-dbg) on your remote Linux machine or WSL installation. The version of libasan-dbg that you load depends on the version of GCC installed on your Linux machine:

ASAN VERSION	GCC VERSION
libasan0	gcc-4.8
libasan2	gcc-5
libasan3	gcc-6
libasan4	gcc-7
libasan5	gcc-8

You can determine which version of GCC you have by using this command:

```
gcc --version
```

To view the version of libasan-dbg you need, run your program, and then look at the **Debug** pane of the **Output** window. The version of ASan that's loaded corresponds to the version of libasan-dbg needed on your Linux machine. You can use **Ctrl** + **F** to search for "libasan" in the window. If you have libasan4, for example, you see a line like this:

```
Loaded '/usr/lib/x86_64-linux-gnu/libasan.so.4'. Symbols loaded.
```

You can install the ASan debug bits on Linux distros that use apt with the following command. This command installs version 4:

```
sudo apt-get install libasan4-dbg
```

Full instructions for installing debug symbol packages on Ubuntu can be found at Debug symbol packages.

If ASan is enabled, Visual Studio prompts you at the top of the **Debug** pane of the **Output** window to install the

ASan debug symbols.