# Build and Debug Existing Project Using External Makefile

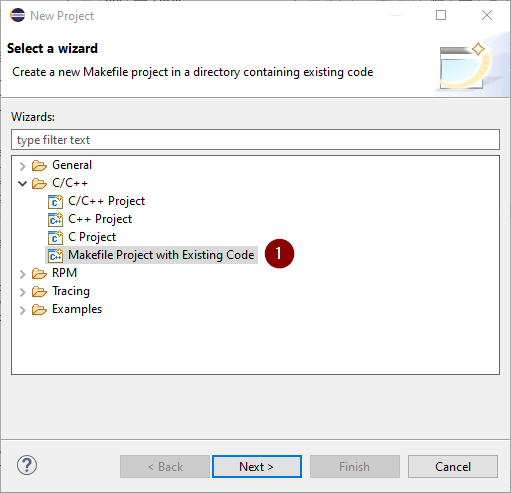
By default, once we create a new C/C++ project, Eclipse automatically generates appropriate settings and makefile for that project.

But what if:

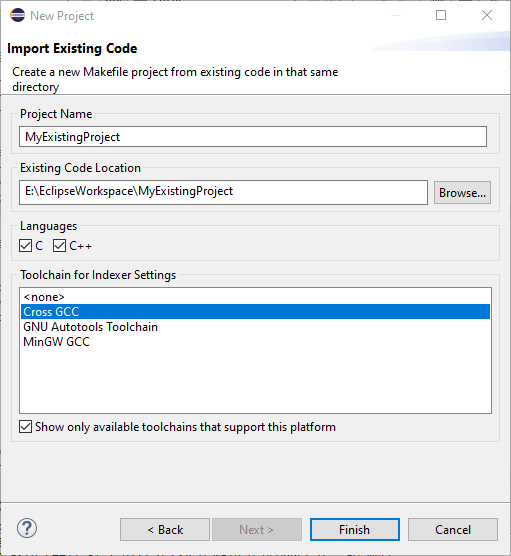
* The project is already created and built using its own makefile.
* And **we want to debug its executable with Eclipse without having to rebuilding it**.
* Or we want to **reuse the existing makefile instead of using the auto-generated makefile by Eclipse**?

Follow these steps:

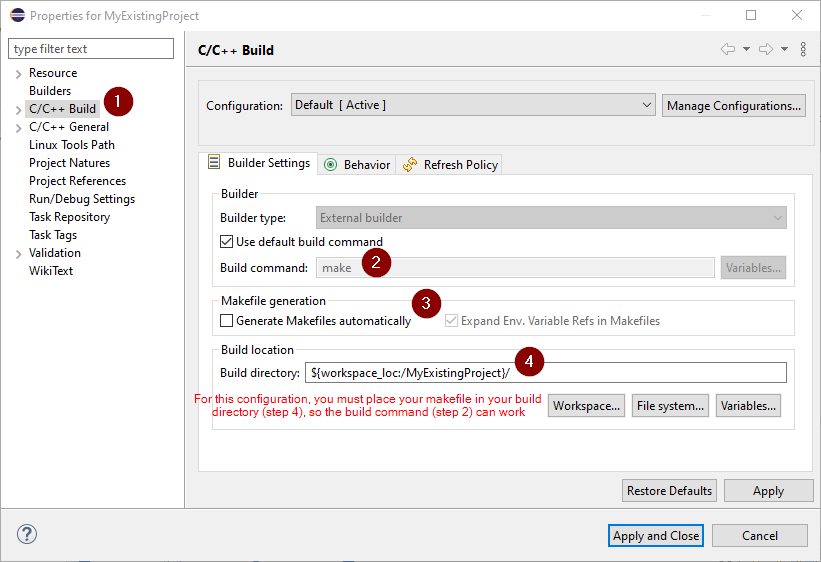
**1. Create project with existing code**



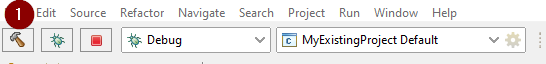
**2. Choose project name, code location, languages and toolchain**



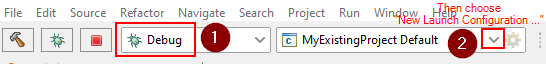
**3. Configure building settings from Project Properties**

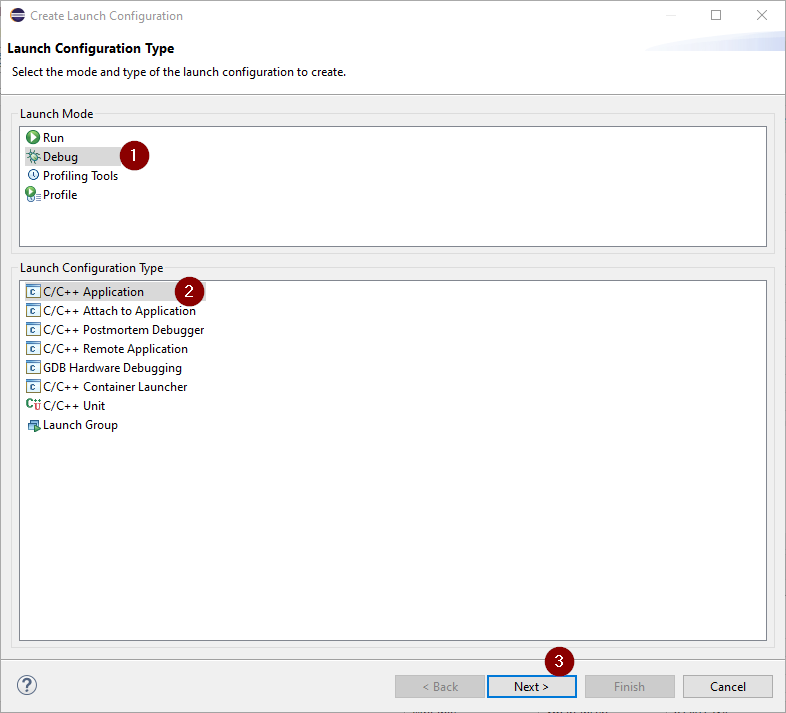


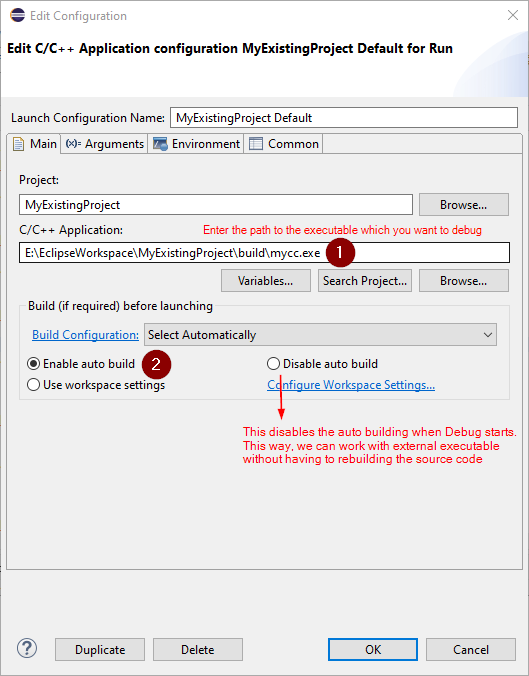
**4. Build the source code**



**5. Configure for debugging**

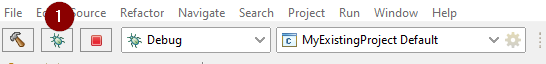






**Note**: If you select "Disable auto build", you can debug existing executable without rebuilding its source code.

**6. Debug your executable**



# Remote Debugging

## Overall Techniques

There are two kinds of setup for remote debugging:

**1. Have the debugger run on the remote system**

Typically, compilation will happen on the remote host, and all tools need to be available remotely including the debugger.

An NFS or Samba mount can be used to make the remote sources visible locally for static analysis and editing.

The make command is changed into "ssh <remote-host> make -C <remote-dir>". The debug command is changed into "ssh <remote-host> gdb <remote-executable".

**2. Have the debugger run on the local system, as a cross debugger, with some kind of thin agent on the remote.**

There are multiple options, but they all have one prerequisite in common: you need a **cross toolchain installed locally**. That is, your compiler runs locally (e.g., on Windows) and produces output that the remote (e.g., Linux) system can execute. Similarly, your debugger runs locally (e.g., *gdbppc*) and is capable of debugging the remote system. Such a cross toolchain is available from commercial vendors, or you can build one yourself. See [Wikipedia:Cross-compilation](http://en.wikipedia.org/wiki/Cross-compilation) for a starting point.

When you have your cross toolchain set up (and tested on the command line), you can start integrating it into Eclipse. There're multiple options for this:

1. Use the [Target Management](http://www.eclipse.org/tm) / RSE Remote CDT Launch. It uses any RSE file subsystem and shell subsystem, so it works with SSH, Telnet, FTP or whatever communication mechanism you have plugged into RSE. It requires a *gdbserver* executable installed on the remote, and is capable of uploading your executable to debug to the remote. Here is a [mailing list entry](http://dev.eclipse.org/mhonarc/lists/dsdp-tm-dev/msg01431.html) that should help you get started.

2. Use the [Device Debugging](http://www.eclipse.org/dsdp/dd) / DSF gdb/mi integration's remote launch capabilities. Here is a [mailing list entry](http://dev.eclipse.org/mhonarc/lists/dsdp-dd-dev/msg01126.html). The [DSDP/DD/GDB](https://wiki.eclipse.org/DSDP/DD/GDB) Wiki gives more information about this component.

3. Use the [Target Communication Framework (TCF)](https://wiki.eclipse.org/TCF) component of Target Management. TCF provides a thin plain-C agent that runs out of the box on Linux and uses ptrace for debugging. There are two separate debugger integrations for Platform/Debug and DD-DSF available. TCF is still incubating, so from a User's perspective the debugging capabilities are still very limited. But if you want to integrate your own debugger or cannot use *gdbserver* for any reason, you should be looking at TCF.

For a general summary about CDT Remote Development, see [this mailing list entry](http://dev.eclipse.org/mhonarc/lists/dsdp-tm-dev/msg01369.html) which has some more pointers into bugzilla, as well as the [PTP/planning/remote](https://wiki.eclipse.org/PTP/planning/remote) Wiki. There was also a [CDT Remote / RDT Presentation](http://www.eclipsecon.org/2008/?page=sub/&id=323) at EclipseCon 2008, which suggests even forming a new Eclipse project for remote development in general.

Ref: <https://wiki.eclipse.org/TM_and_RSE_FAQ#How_can_I_do_Remote_Debugging_with_CDT.3F>

## Way #1: Using Remote System Explorer End-User Runtime (RSE)

Refs:

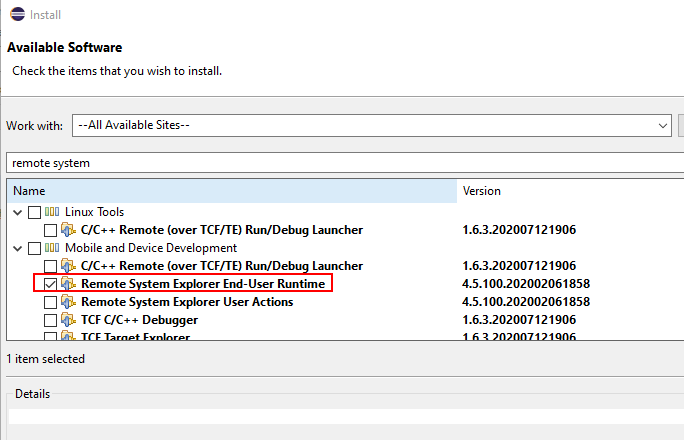
<https://wiki.eclipse.org/CDT/User/FAQ#How_do_I_debug_a_remote_application.3F>

<https://wiki.eclipse.org/TM_and_RSE_FAQ#How_can_I_use_a_remote_workspace_over_SSH.3F>

**Warning:**

In below examples, we configure for **debugging a Linux app from Windows host**. If this is not your case, change the cross toolchain and Eclipse settings based on your needs.

### 1. Install RSE



### 2. Install gdbserver

Make sure the remote machine has *gdbserver* installed. Usually, Linux provides *gdbserver* (along with *gdb*, *gcc*, *make*, etc.) by default, but Windows requires manually installing these C/C++ dev tools (should be via MingW or Cygwin).

### 3. Check SSH Connection

There are several remote connection types we can choose for remote debugging, such as SSH, Serial Port or Telnet. In this guide, we'll only focus on using SSH.

So, make sure your **SSH connection between the local machine and remote machine is established**,before moving to step 3.

How:

1. Install SSH Client on the local machine and SSH Server on the remote machine:

- Windows 10: Install OpenSSH from Apps and Features with this [guide](https://www.thomasmaurer.ch/2017/11/install-ssh-on-windows-10-as-optional-feature/).

- Linux:

2. Start SSH Server on the remote machine:

- Windows 10: Open the Services window, then start *OpenSSH SSH Server*.

- Linux:

3. Check SSH connection between local machine and remote machine by running:

ssh <remote-user>@<remote-hostname-or-ip>

For example: ssh [triho@172.168.11.25](mailto:triho@172.168.11.25)

### 4. Cross Compile

Despite which launchers you choose (at step 5), you must need a **cross toolchain installed locally**. That is, your compiler runs locally (e.g., on Windows) and produces output that the remote (e.g., Linux) system can execute. Similarly, your debugger runs locally (e.g., *gdbppc*) and is capable of debugging the remote system.

Make sure the **app is built with "-g" flag (or -g1, -g3)**. Otherwise, we CANNOT debug it (error such as "no source code found" when starting debugging, etc.).

**Personal experience:**

Cross compiling between Linux distributions is quite easy. But **cross compiling between Windows and Linux is much harder**. So, consider [Way #2: Using Direct Remote C++ Debugging](#_Way_#2:_Using) which works better for different platforms.

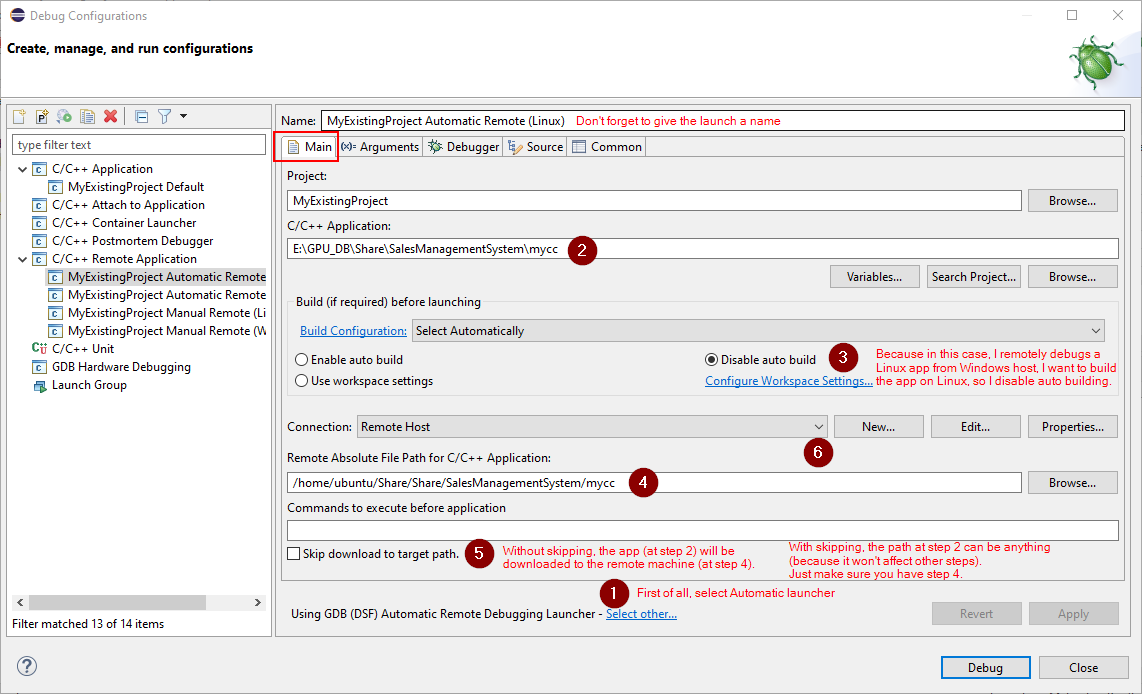
### 5. Configure Debugging Settings in Eclipse

With RSE, there are three launches to debug a remote application:

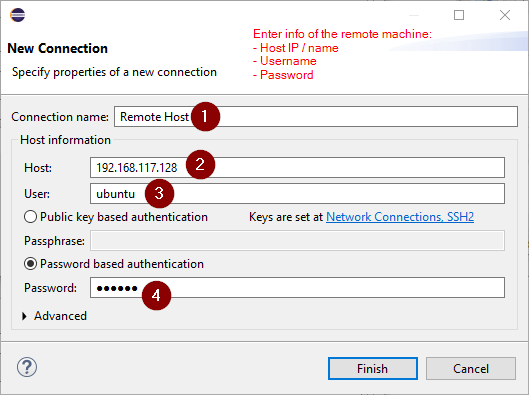
* **Automatic Remote Launcher**: *Start* the application on the remote machine in debug mode. This launch:
  + Automatically downloads your app to the remote machine (unless specifically told not to)
  + Automatically starts *gdbserver* on the remote machine using that app
  + Automatically connects GDB to the *gdbserver* just started. You can also specify some commands to be run before *gdbserver* is started.
* **Manual Remote Launcher**: *Start* the application on the remote machine in debug mode. This launch:
* Expects you to manually download your app to the remote machine
* Expects you to manually start *gdbserver* on the remote machine using that app
* Every time your re-launch, you will need to restart *gdbserver* (as it usually terminates when the debug session terminates).
* **Remote Attach Launcher**: Debug one or more *already running* applications on a remote machine.

#### #1: Automatic Remote Launcher

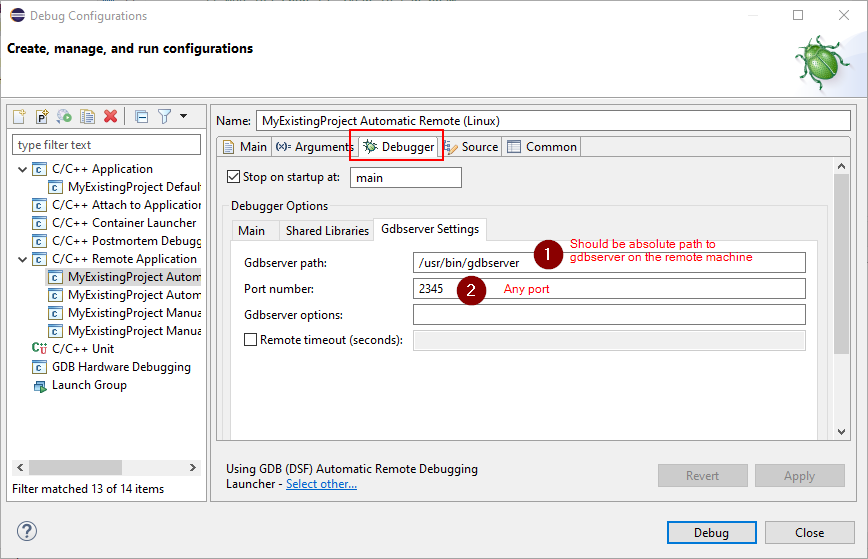
Create a new launch and configure it from Debug Configurations window:



At step 6, choose your connection as SSH, and enter info for that connection:



Move to the Debugger tab:



**Notes:**

* When connecting to the remote machine, the PATH variable may not be set properly (seem like a bug of Eclipse), which causes cannot-find-*gdbserver* error. So, it is safer to use an absolute path for the *gdbserver* location.

Give the launch a target:



Refs:

[Eclipse remote development and debugging - 96Boards](https://www.96boards.org/blog/eclipse-remote-development-debugging/)

#### #2: Manual Remote Launcher

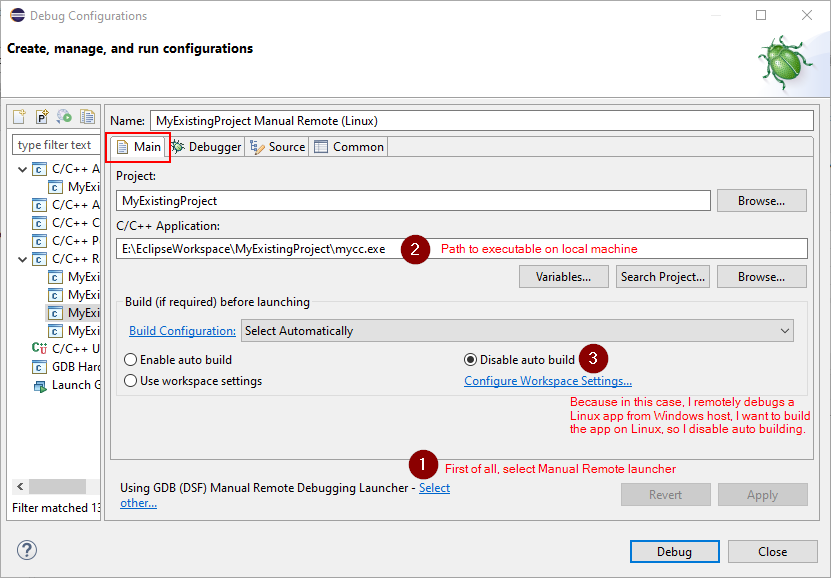
With this launch, we have to do following steps manually:

* Must build the app on the remote machine.
* Start *gdbserver* on the remote machine using that app, using command:

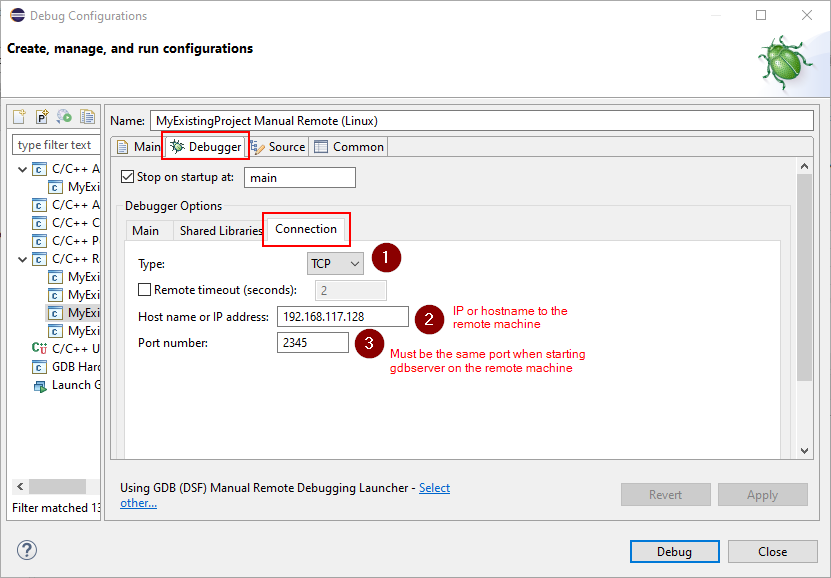
*gdbserver* :<port> <path-to-executable>

* Restart *gdbserver* on the remote machine every time start debugging the app.

Create a new launch and configure it from Debug Configurations window:



Move to the Debugger tab:



Give the launch a target:



#### #3: Remote Attach Launcher

<https://wiki.eclipse.org/CDT/User/FAQ#How_do_I_debug_a_remote_application.3F> (section Remote Attach Launch)

### 6. Start Remote Debugging

Once all settings are configured, you can start remote debugging.

## Way #2: Using Direct Remote C++ Debugging

**Warning:**

In below examples, we configure for **debugging a Linux app from Windows host**. If this is not your case, change your Eclipse settings based on your needs.

This way offers following advantages:

* Only requires *gdb* to be installed in the remote machine (the ***gdbsever* is NOT required**).
* Source code files in the Eclipse project can be a copy of the remote source code used to build the program. In this case, an **automatic mapping between the local and remote source code** is happened so you can set breakpoints in Eclipse as you do in normal debugging.
* Eclipse and the debugged program can be on **different platforms**. For example, you can debug a Linux app from Windows host (our example in this guide).

### 1. Install DRD

<http://marketplace.eclipse.org/content/direct-remote-c-debugging>

### 2. Check SSH Connection

Similar to what described in Way #1 (using RSE).

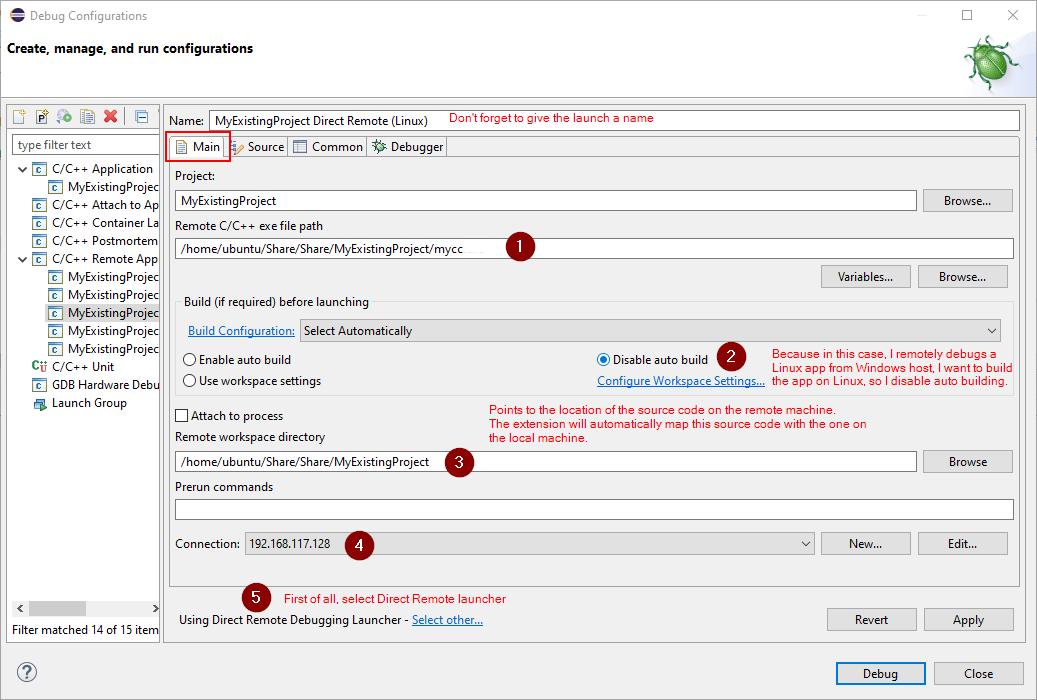
### 3. Compile

Simply compile your source code on the remote machine using its native compiler.

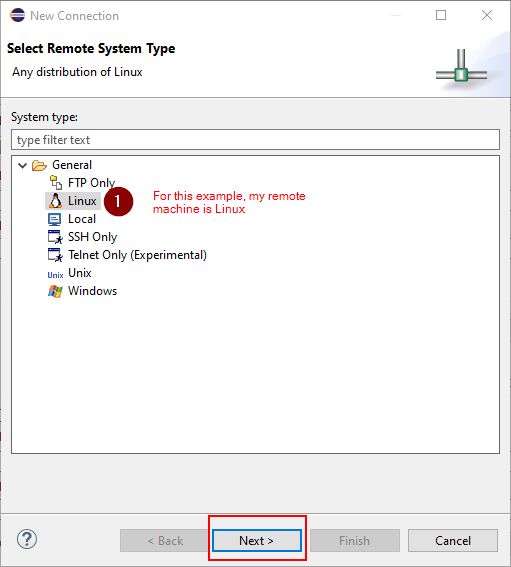
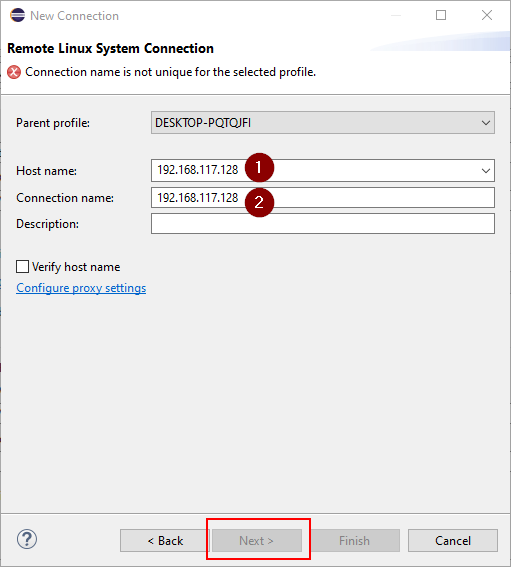
**Note**: Make sure the app is built on the remote machine with "-g" flag (or -g1, -g3). Otherwise, we cannot debug it (error such as "no source code found" when starting debugging, etc.).

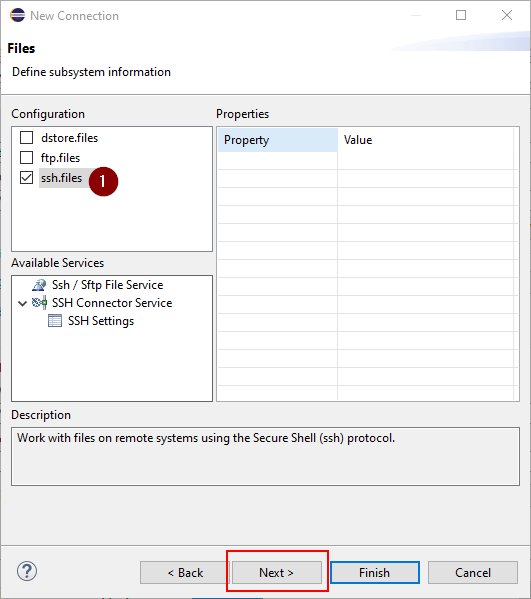
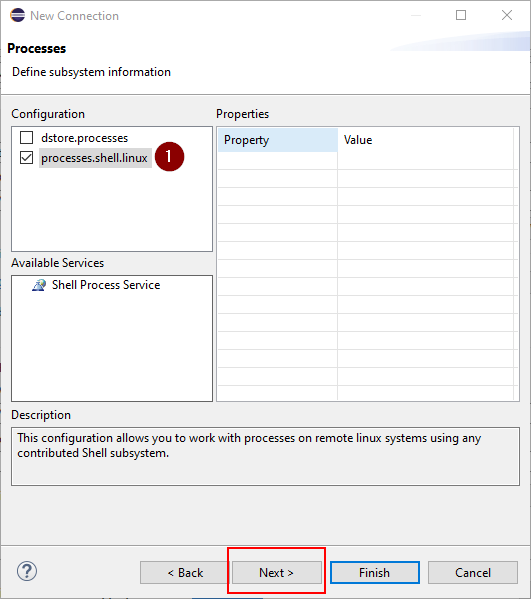
### 4. Configure Debugging Settings in Eclipse

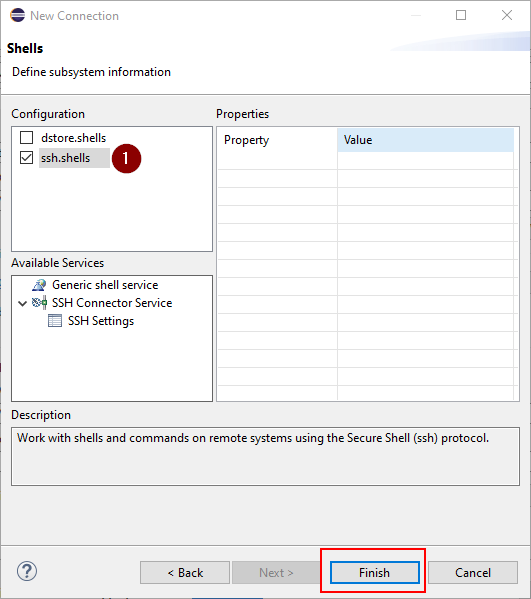
Create a new launch and configure it from Debug Configurations window:

****

At step 5, enter info for the remote machine (system type, IP/hostname, etc.):







Give the launch a target:



### 5. Start Remote Debugging

Once all settings are configured, you can start remote debugging.