Version 1.41 (/updates) is now available! Read about the new features and fixes from November. ✕

TOPICS Mingw-w64 on Windows ▾

Using Mingw-w64 in VS Code

(https://github.com/Microsoft/vscode-docs/blob/master/docs/cpp/config-mingw.md)

In this tutorial, you configure Visual Studio Code to use the GCC C++ compiler (g++) and GDB debugger from Mingw-w64 (http://mingw-w64.org/doku.php/start) to create programs that run on Windows.

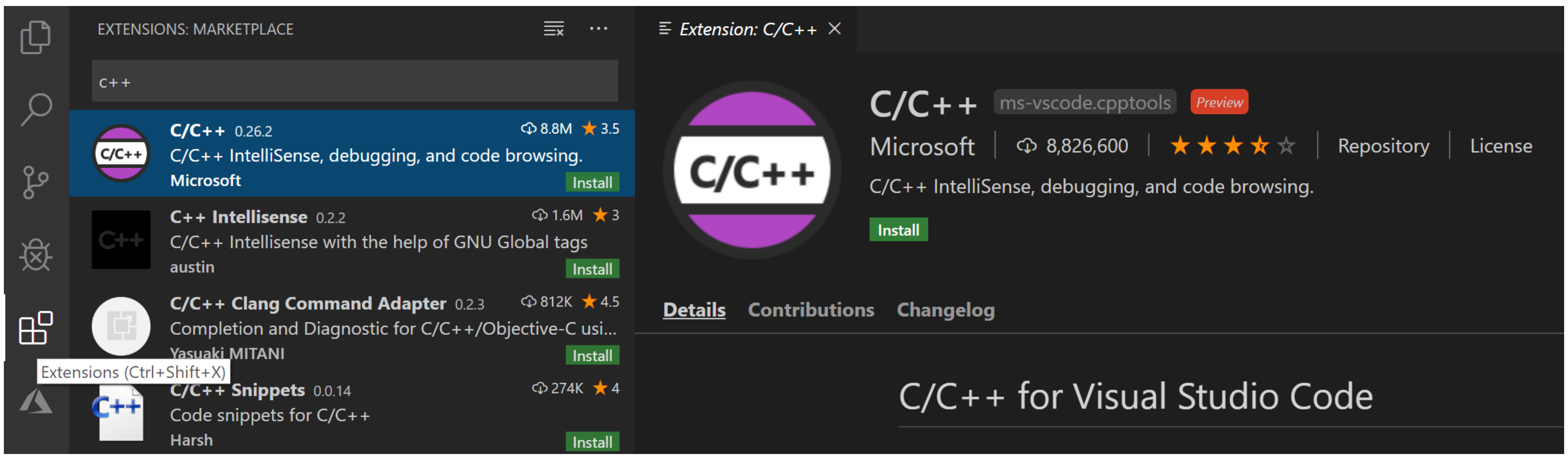
After configuring VS Code, you will compile and debug a simple Hello World program in VS Code. This tutorial does not teach you about GCC or Mingw-w64 or the C++ language. For those subjects, there are many good resources available on the Web.

If you have any problems, feel free to file an issue for this tutorial in the VS Code documentation repository (https://github.com/Microsoft/vscode-docs/issues).

Prerequisites

To successfully complete this tutorial, you must do the following steps:

1. Install Visual Studio Code (/download).
2. Install the C/C++ extension for VS Code (https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools). You can install the C/C++ extension by searching for 'c++' in the Extensions view (`Ctrl+Shift+X`).


EXTENSIONS: MARKETPLACE
c++
C/C++ 0.26.2 8.8M 3.5
C/C++ IntelliSense, debugging, and code browsing.
Microsoft Install
C++ Intellisense 0.2.2 1.6M 3
C/C++ Intellisense with the help of GNU Global tags
austin Install
C/C++ Clang Command Adapter 0.2.3 812K 4.5
Completion and Diagnostic for C/C++/Objective-C usi...
Yasuaki MITANI Install
Extensions (Ctrl+Shift+X)
C/C++ Snippets 0.0.14 274K 4
Code snippets for C/C++
Harsh Install
Extension: C/C++
C/C++ ms-vscode.cpptools Preview
Microsoft 8,826,600 Repository License
C/C++ IntelliSense, debugging, and code browsing.
Install
Details Contributions Changelog
C/C++ for Visual Studio Code

3. Install Mingw-w64 (http://mingw-w64.org/doku.php/download/mingw-builds) via the SourceForge website to a folder that has no spaces in its path (in other words, NOT the default location of C:/Program Files/). In this tutorial, we assume it is installed under `C:\mingw-w64` .
4. Add the path to your Mingw-w64 `bin` folder to the Windows PATH environment variable.
   1. In the Windows search bar, type 'settings' to open your Windows Settings.
   2. Search for **Edit environment variables for your account**.
   3. Choose the `Path` variable and then select **Edit**.
   4. Select **New** and add the Mingw-w64 path to the system path. The exact path depends on which version of Mingw-w64 you have installed and where you installed it. Here is an example: `c:\mingw-w64\x86_64-8.1.0-win32-seh-rt_v6-rev0\mingw64\bin` .
   5. Select **OK** to save the `Path` update. You will need to reopen any console windows for the new PATH location to be available.

Check your MinGW installation

To check that your Mingw-w64 tools are correctly installed and available, open a new Command Prompt and type:

```
g++ --version
gdb --version
```

If you don't see the expected output or `g++` or `gdb` is not a recognized command, check your installation (Windows **Control Panel** > **Programs**) and make sure your PATH entry matches the Mingw-w64 location.

Create Hello World

From a Windows command prompt, create an empty folder called `projects` where you can place all your VS Code projects. Then create a sub-folder called `helloworld` , navigate into it, and open VS Code in that folder by entering the following commands:

```
mkdir projects
cd projects
mkdir helloworld
cd helloworld
code .
```

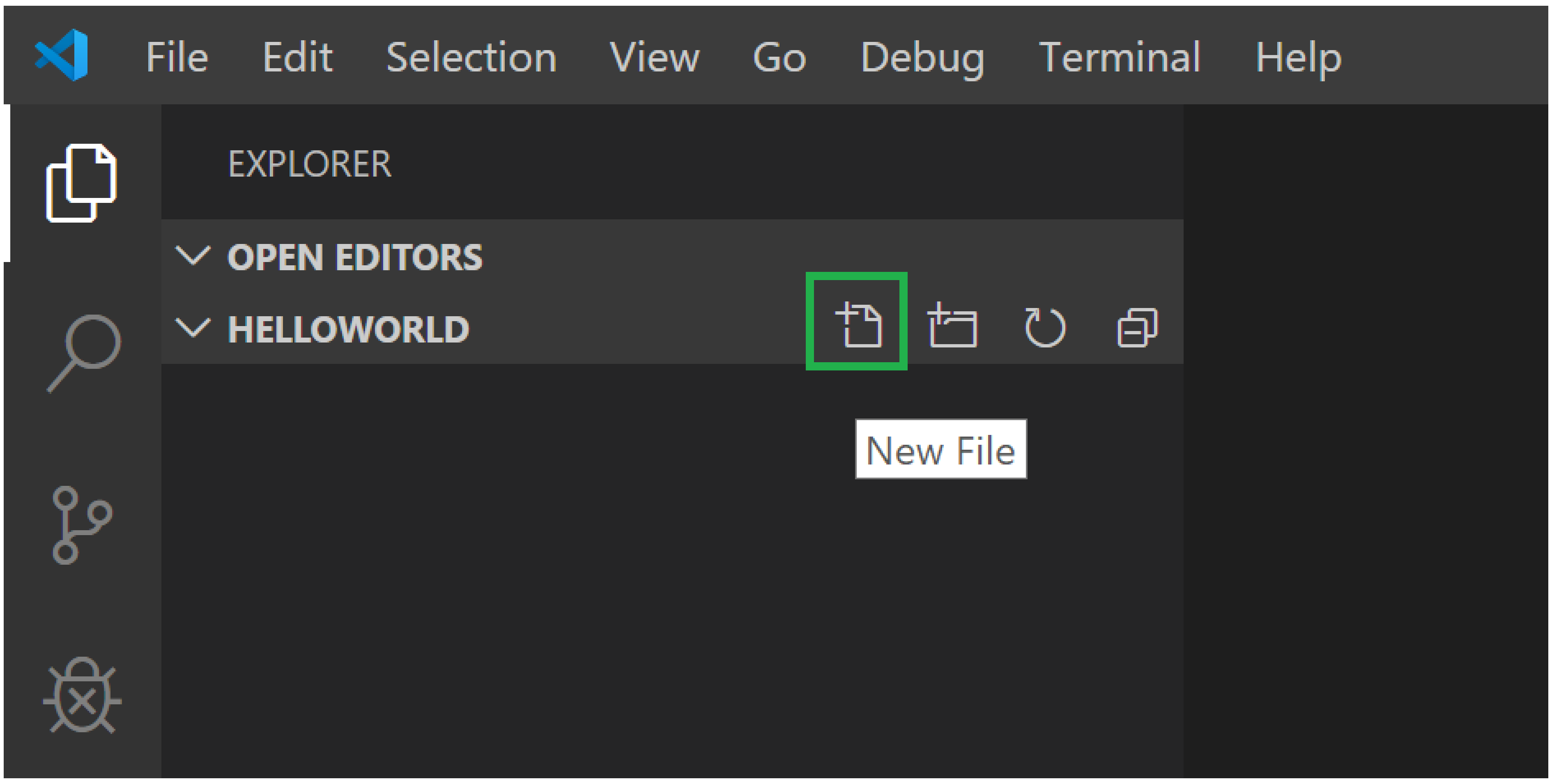
The **code .** command opens VS Code in the current working folder, which becomes your "workspace". As you go through the tutorial, you will see three files created in a `.vscode` folder in the workspace:

- `tasks.json` (build instructions)

- `launch.json` (debugger settings)
- `c_cpp_properties.json` (compiler path and IntelliSense settings)

Add a source code file

In the File Explorer title bar, select the **New File** button and name the file `helloworld.cpp` .

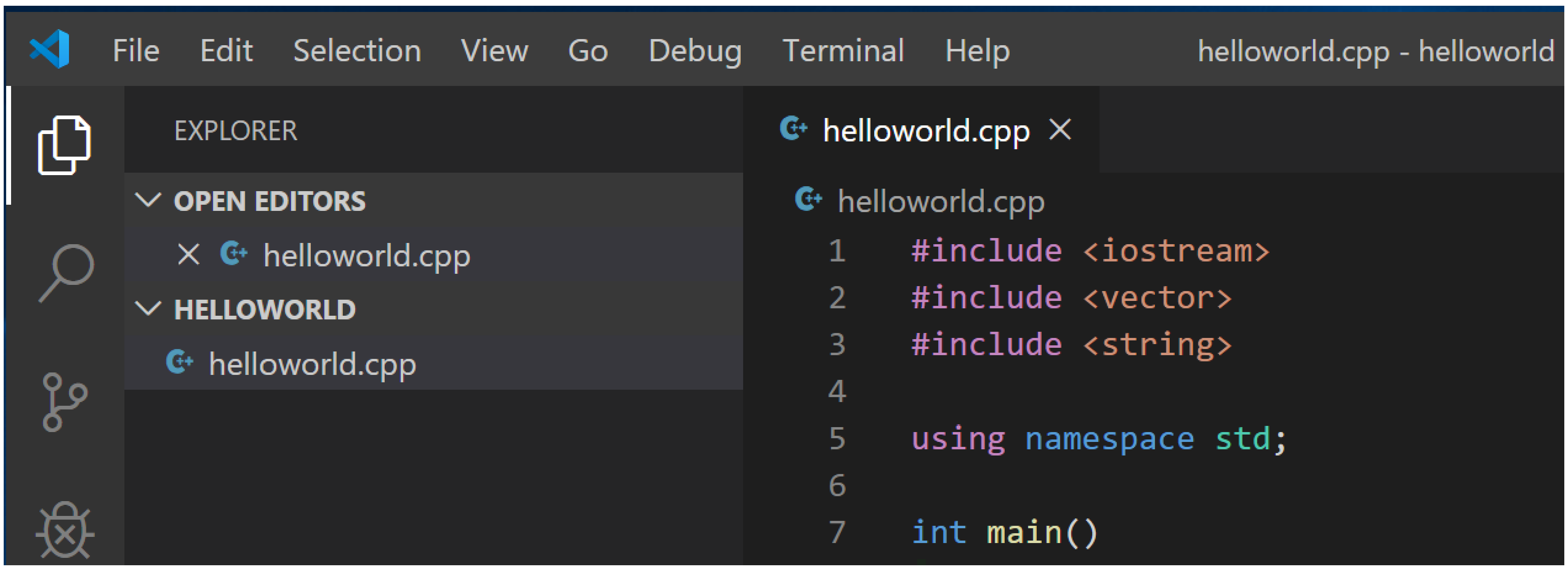


Add hello world source code

Now paste in this source code:

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
   vector<string> msg {"Hello", "C++", "World", "from", "VS Code", "and the C++ extension!"};

   for (const string& word : msg)
   {
      cout << word << " ";
   }
   cout << endl;
}
```

Now press `Ctrl+S` to save the file. Notice how the file you just added appears in the **File Explorer** view (`Ctrl+Shift+E`) in the side bar of VS Code:



```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
```

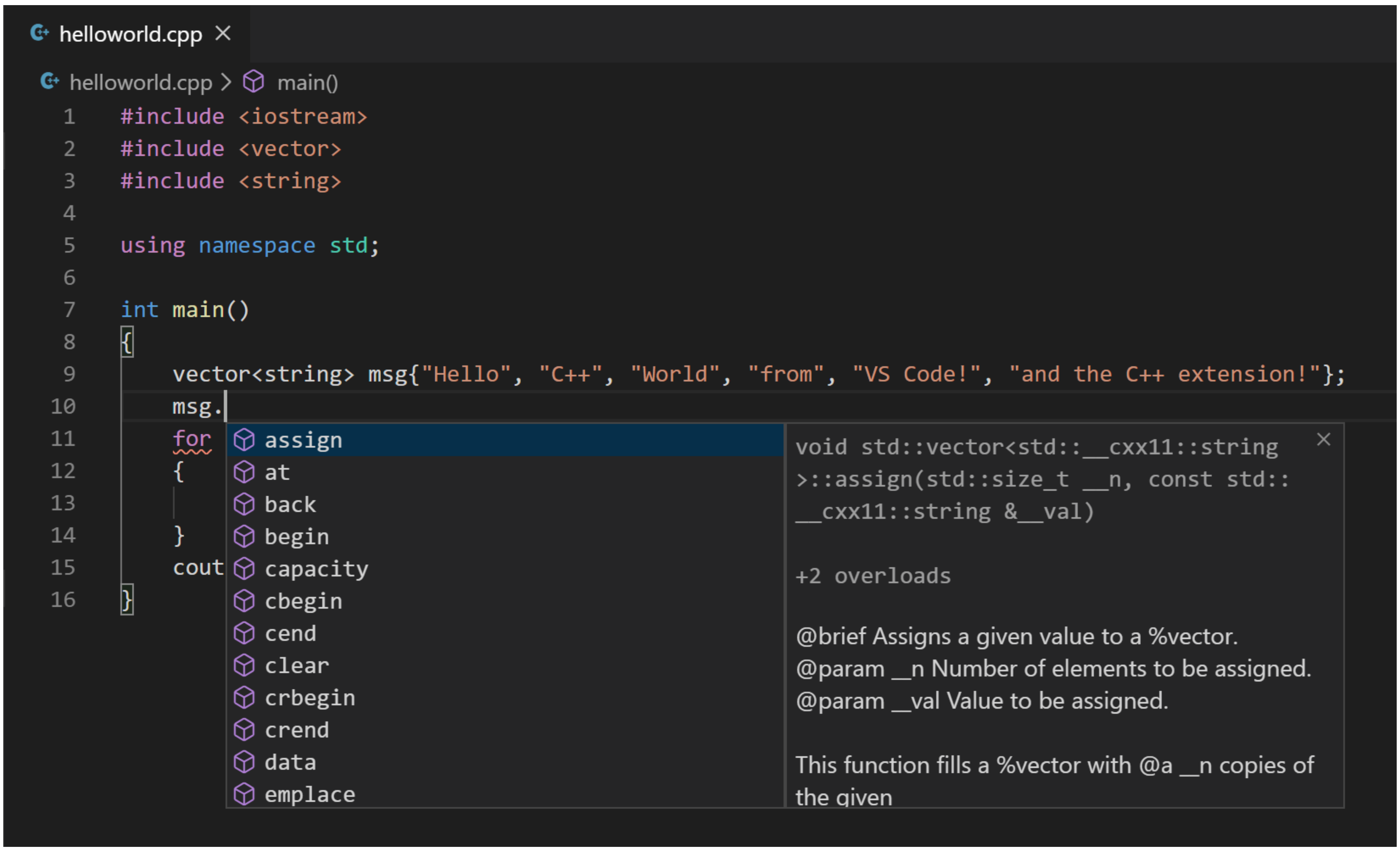
You can also enable Auto Save (/docs/editor/codebasics#_saveauto-save) to automatically save your file changes, by checking **Auto Save** in the main **File** menu.

The Activity Bar on the far left lets you open different views such as **Search**, **Source Control**, and **Debug**. You'll look at the **Debug** view later in this tutorial. You can find out more about the other views in the VS Code User Interface documentation (/docs/getstarted/userinterface).

> **Note**: When you save or open a C++ file, you may see a notification from the C/C++ extension about the availability of an Insiders version, which lets you test new features and fixes. You can ignore this notification by selecting the `X` (**Clear Notification**).

Explore IntelliSense

In your new `helloworld.cpp` file, hover over `vector` or `string` to see type information. After the declaration of the `msg` variable, start typing `msg.` as you would when calling a member function. You should immediately see a completion list that shows all the member functions, and a window that shows the type information for the `msg` object:



```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> msg{"Hello", "C++", "World", "from", "VS Code!", "and the C++ extension!"};
    msg.
    for
    {

    }
    cout
}
```

```
void std::vector<std::__cxx11::string
>::assign(std::size_t __n, const std::
__cxx11::string &__val)

+2 overloads
```

@brief Assigns a given value to a %vector.
@param __n Number of elements to be assigned.
@param __val Value to be assigned.

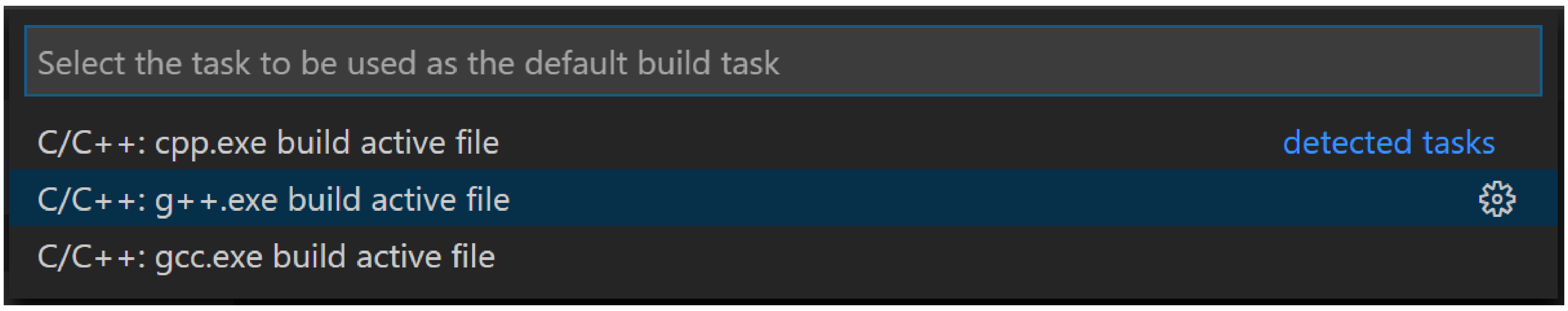
This function fills a %vector with @a __n copies of the given

You can press the `Tab` key to insert the selected member; then, when you add the opening parenthesis, you will see information about any arguments that the function requires.

Build helloworld.cpp

Next, you will create a `tasks.json` file to tell VS Code how to build (compile) the program. This task will invoke the g++ compiler to create an executable file based on the source code.

From the main menu, choose **Terminal** > **Configure Default Build Task**. In the dropdown, which will display a tasks dropdown listing various predefined build tasks for C++ compilers. Choose **g++.exe build active file**, which will build the file that is currently displayed (active) in the editor.


Select the task to be used as the default build task
C/C++: cpp.exe build active file
detected tasks
C/C++: g++.exe build active file
C/C++: gcc.exe build active file

This will create a `tasks.json` file in a `.vscode` folder and open it in the editor.

Your new `tasks.json` file should look similar to the JSON below:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "shell",
      "label": "g++.exe build active file",
      "command": "C:\\mingw-w64\\i686-8.1.0-posix-dwarf-rt_v6-rev0\\mingw32\\bin\\g++.exe",
      "args": ["-g", "${file}", "-o", "${fileDirname}\\${fileBasenameNoExtension}.exe"],
      "options": {
        "cwd": "C:\\mingw-w64\\i686-8.1.0-posix-dwarf-rt_v6-rev0\\mingw32\\bin"
      },
      "problemMatcher": ["$gcc"],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

The `command` setting specifies the program to run; in this case that is g++. The `args` array specifies the command-line arguments that will be passed to g++. These arguments must be specified in the order expected by the compiler. This task tells g++ to take the active file (`${file}`), compile it, and create an executable file in the current directory (`${fileDirname}`) with the same name as the active file but with the `.exe` extension (`${fileBasenameNoExtension}.exe`), resulting in `helloworld.exe` for our example.

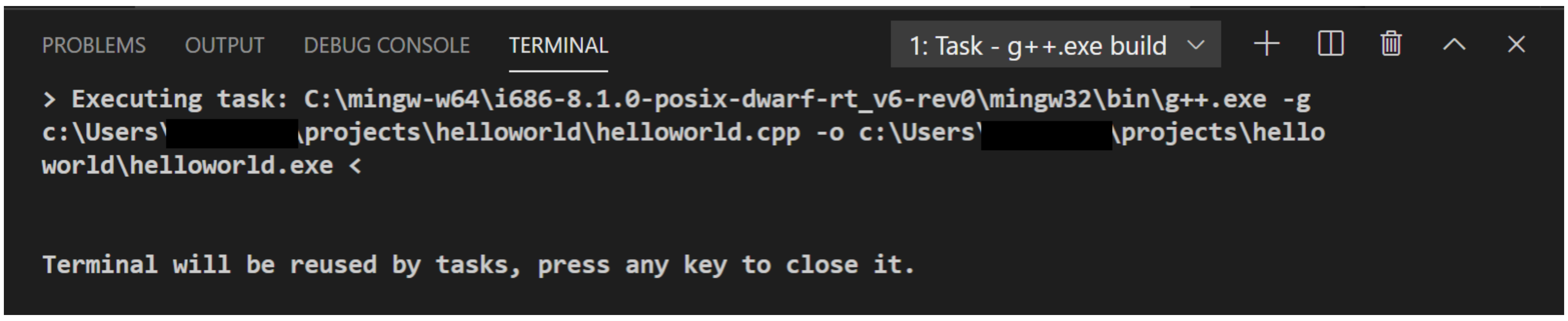
> **Note**: You can learn more about `task.json` variables in the variables reference (/docs/editor/variables-reference).

The `label` value is what you will see in the tasks list; you can name this whatever you like.

The `"isDefault": true` value in the `group` object specifies that this task will be run when you press `Ctrl+Shift+B` . This property is for convenience only; if you set it to false, you can still run it from the Terminal menu with **Tasks: Run Build Task**.

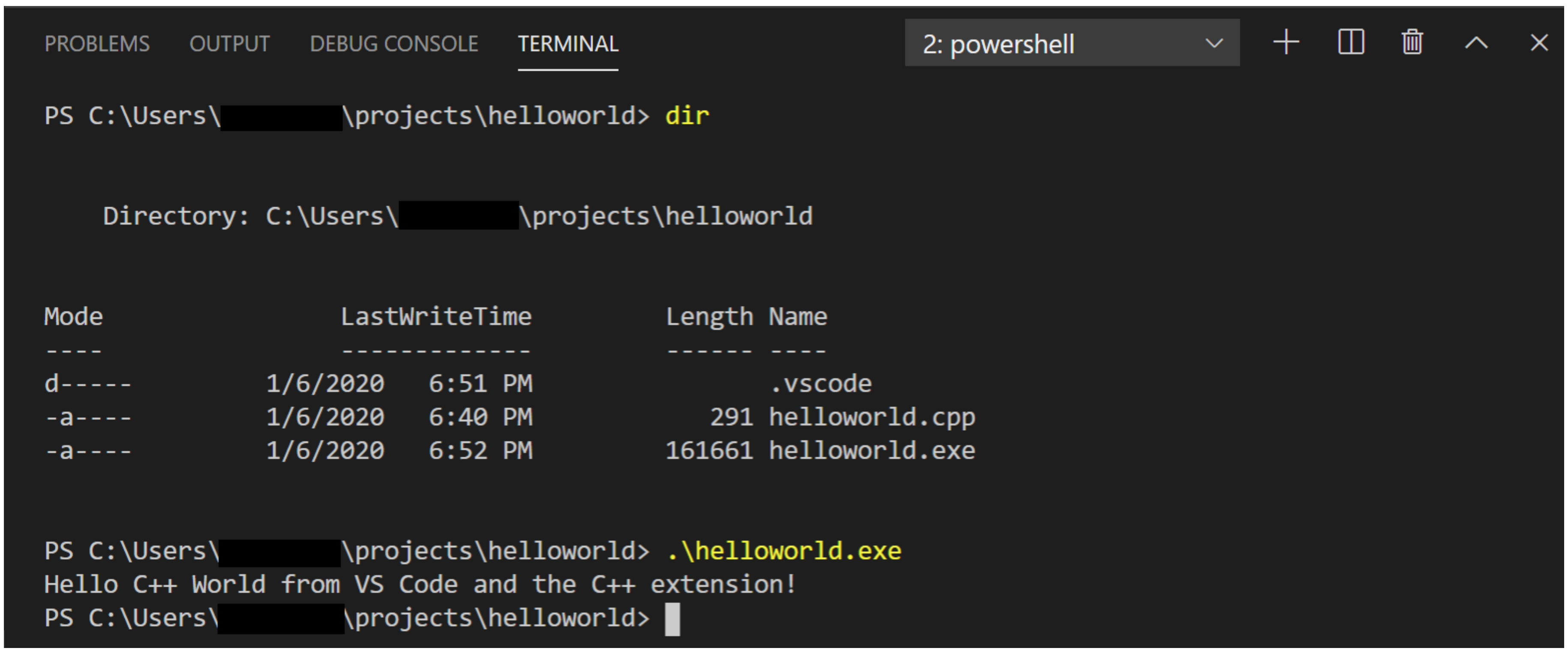
Running the build

1. Go back to `helloworld.cpp` . Your task builds the active file and you want to build `helloworld.cpp` .
2. To run the build task defined in `tasks.json` , press `Ctrl+Shift+B` or from the **Terminal** main menu choose **Tasks: Run Build Task**.
3. When the task starts, you should see the Integrated Terminal panel appear below the source code editor. After the task completes, the terminal shows output from the compiler that indicates whether the build succeeded or failed. For a successful g++ build, the output looks something like this:



```
> Executing task: C:\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\g++.exe -g
c:\Users\[redacted]\projects\helloworld\helloworld.cpp -o c:\Users\[redacted]\projects\hello
world\helloworld.exe <


Terminal will be reused by tasks, press any key to close it.
```

4. Create a new terminal using the + button and you'll have a new terminal (running PowerShell) with the `helloworld` folder as the working directory. Run `dir` and you should now see the executable `helloworld.exe` .



```
PS C:\Users\[redacted]\projects\helloworld> dir


    Directory: C:\Users\[redacted]\projects\helloworld


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         1/6/2020   6:51 PM                .vscode
-a----         1/6/2020   6:40 PM            291 helloworld.cpp
-a----         1/6/2020   6:52 PM         161661 helloworld.exe


PS C:\Users\[redacted]\projects\helloworld> .\helloworld.exe
Hello C++ World from VS Code and the C++ extension!
PS C:\Users\[redacted]\projects\helloworld> █
```

5. You can run `helloworld` in the terminal by typing `.\helloworld.exe` .

> **Note**: You might need to press `Enter` a couple of times initially to see the PowerShell prompt in the terminal. This issue should be fixed in a future release of Windows.

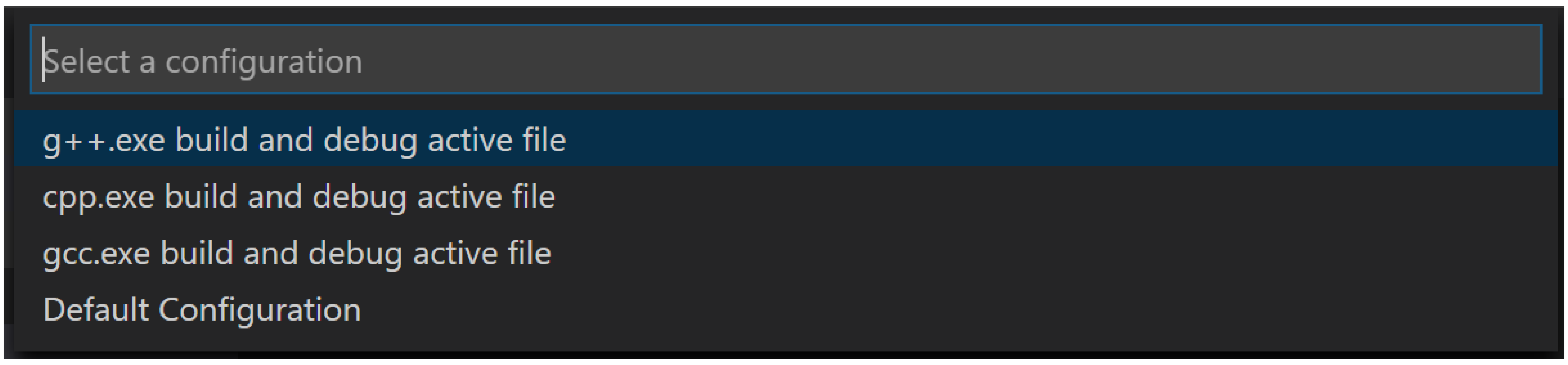
Modifying tasks.json

You can modify your `tasks.json` to build multiple C++ files by using an argument like `"${workspaceFolder}\\*.cpp"` instead of `${file}`. This will build all `.cpp` files in your current folder. You can also modify the output filename by replacing `"${fileDirname}\\${fileBasenameNoExtension}.exe"` with a hard-coded filename (for example `"${workspaceFolder}\\myProgram.exe"`).

Debug helloworld.cpp

Next, you'll create a `launch.json` file to configure VS Code to launch the GDB debugger when you press `F5` to debug the program. From the main menu, choose **Debug** > **Add Configuration...** and then choose **C++ (GDB/LLDB)**.

You'll then see a dropdown for various predefined debugging configurations. Choose **g++.exe build and debug active file**.


Select a configuration
g++.exe build and debug active file
cpp.exe build and debug active file
gcc.exe build and debug active file
Default Configuration

VS Code creates a `launch.json` file, opens it in the editor, and builds and runs 'helloworld'.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "g++.exe build and debug active file",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "C:\\mingw-w64\\i686-8.1.0-posix-dwarf-rt_v6-rev0\\mingw32\\bin\\gdb.exe",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ],
      "preLaunchTask": "g++.exe build active file"
    }
  ]
}
```

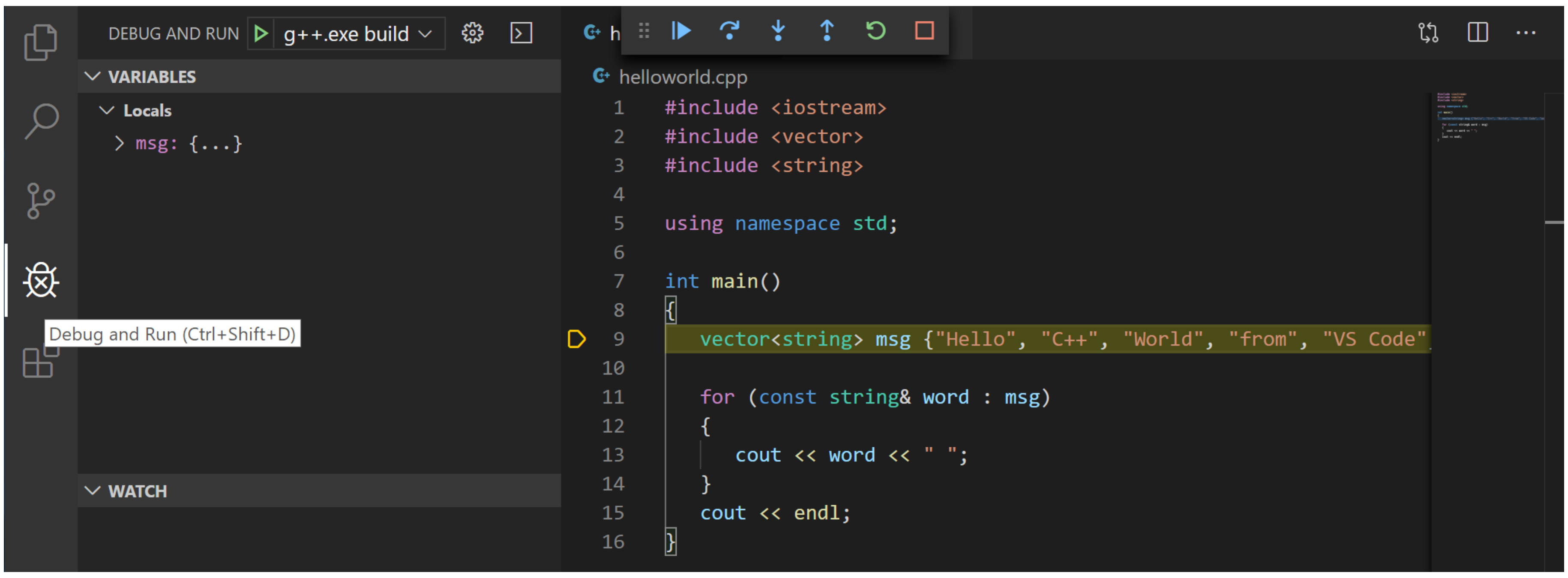
The `program` setting specifies the program you want to debug. Here it is set to the active file folder `${fileDirname}` and active filename with the `.exe` extension `${fileBasenameNoExtension}.exe`, which if `helloworld.cpp` is the active file will be `helloworld.exe`.

By default, the C++ extension won't add any breakpoints to your source code and the `stopAtEntry` value is set to `false`. Change the `stopAtEntry` value to `true` to cause the debugger to stop on the `main` method when you start debugging.

Start a debugging session

1. Go back to `helloworld.cpp` so that it is the active file.
2. Press `F5` or from the main menu choose **Debug** > **Start Debugging**. Before you start stepping through the source code, let's take a moment to notice several changes in the user interface:

- The Integrated Terminal appears at the bottom of the source code editor. In the **Debug Output** tab, you see output that indicates the debugger is up and running.
- The editor highlights the first statement in the `main` method. This is a breakpoint that the C++ extension automatically sets for you:



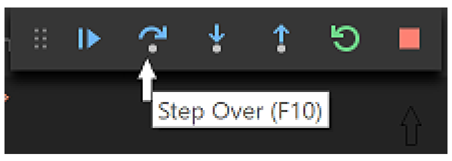
```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> msg {"Hello", "C++", "World", "from", "VS Code"
    
    for (const string& word : msg)
    {
        cout << word << " ";
    }
    cout << endl;
}
```

- The Debug view on the left shows debugging information. You'll see an example later in the tutorial.
- At the top of the code editor, a debugging control panel appears. You can move this around the screen by grabbing the dots on the left side.

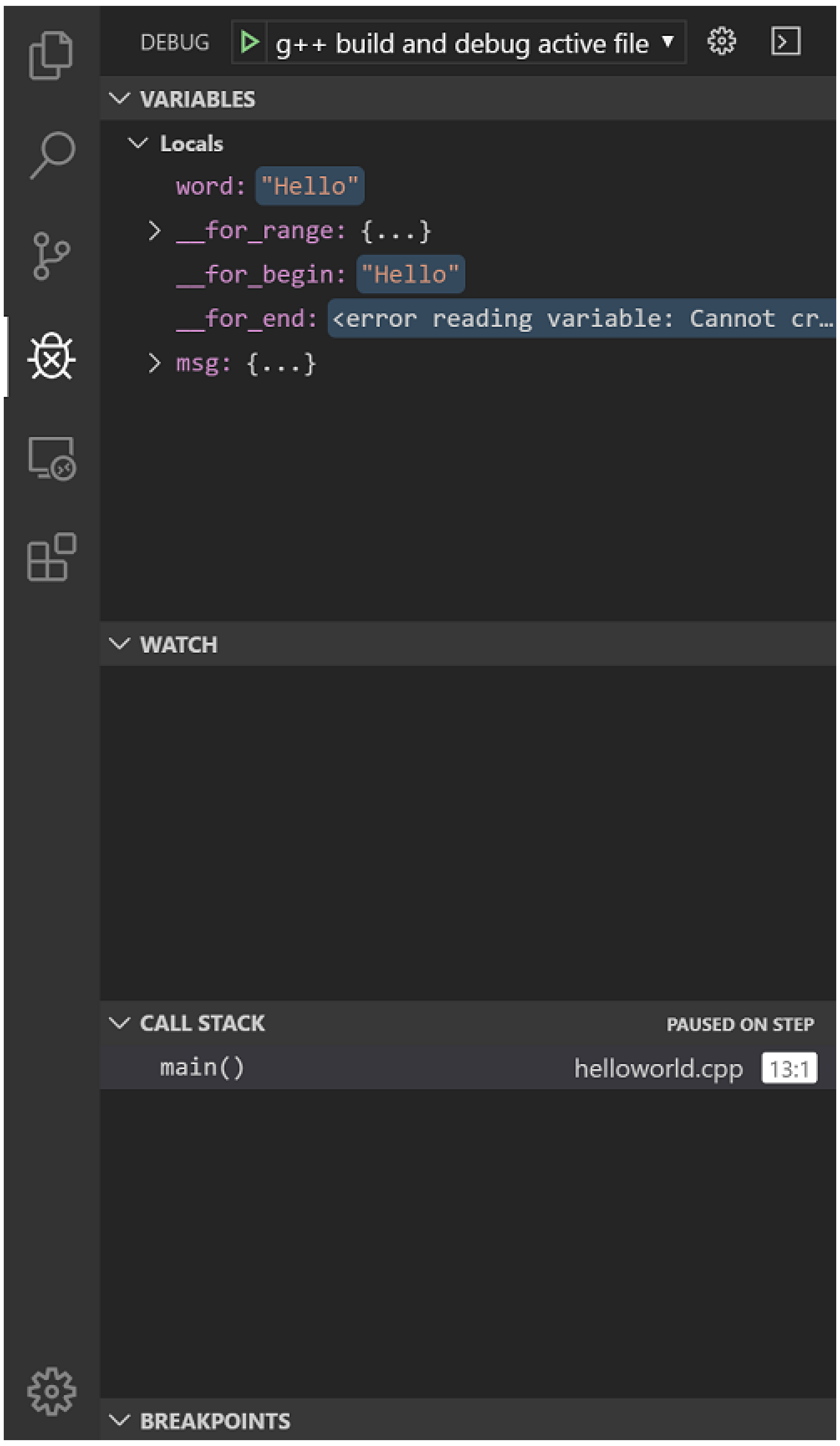
Step through the code

Now you're ready to start stepping through the code.

1. Click or press the **Step over** icon in the debugging control panel.



This will advance program execution to the first line of the for loop, and skip over all the internal function calls within the `vector` and `string` classes that are invoked when the `msg` variable is created and initialized. Notice the change in the **Variables** window on the left.



```
word: "Hello"
> __for_range: {...}
__for_begin: "Hello"
__for_end: <error reading variable: Cannot cr…
> msg: {...}
```

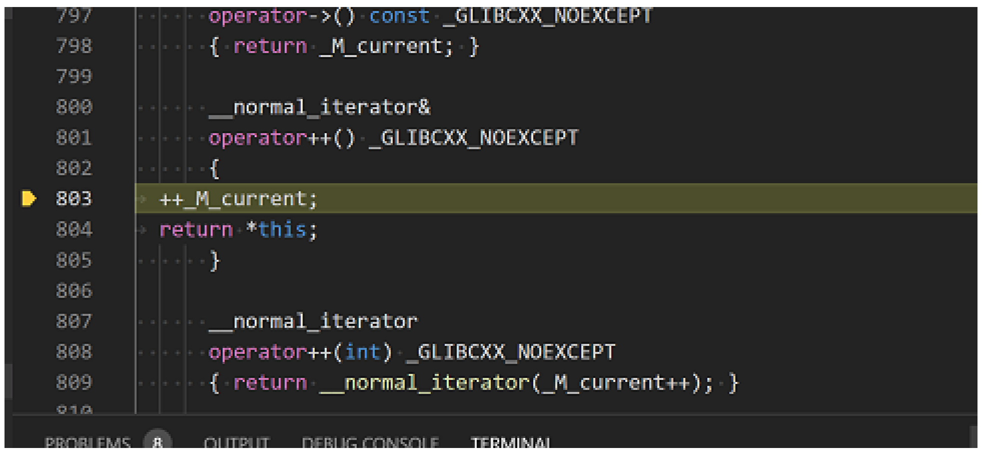
```
main()    helloworld.cpp 13:1
```

In this case, the errors are expected because, although the variable names for the loop are now visible to the debugger, the statement has not executed yet, so there is nothing to read at this point. The contents of `msg` are visible, however, because that statement has completed.

2. Press **Step over** again to advance to the next statement in this program (skipping over all the internal code that is executed to initialize the loop). Now, the **Variables** window shows information about the loop variables.

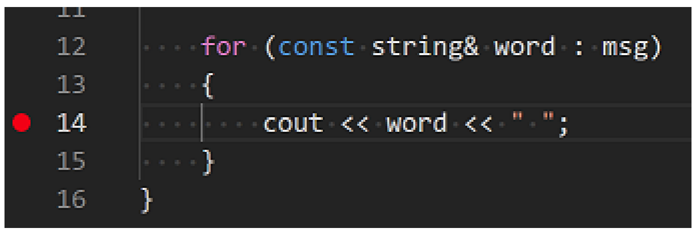
3. Press **Step over** again to execute the `cout` statement. (Note that as of the March 2019 release, the C++ extension does not print any output to the **Debug Console** until the loop exits.)

4. If you like, you can keep pressing **Step over** until all the words in the vector have been printed to the console. But if you are curious, try pressing the **Step Into** button to step through source code in the C++ standard library!



```
      operator->() const _GLIBCXX_NOEXCEPT
      { return _M_current; }

      __normal_iterator&
      operator++() _GLIBCXX_NOEXCEPT
      {
	++_M_current;
	return *this;
      }

      __normal_iterator
      operator++(int) _GLIBCXX_NOEXCEPT
      { return __normal_iterator(_M_current++); }
```

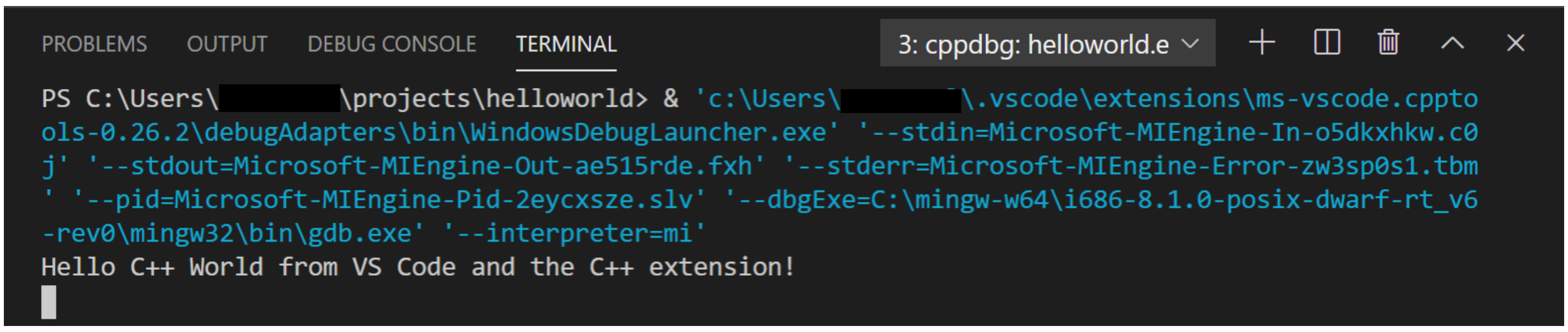
To return to your own code, one way is to keep pressing **Step over**. Another way is to set a breakpoint in your code by switching to the `helloworld.cpp` tab in the code editor, putting the insertion point somewhere on the `cout` statement inside the loop, and pressing `F9`. A red dot appears in the gutter on the left to indicate that a breakpoint has been set on this line.



```
    for (const string& word : msg)
    {
        cout << word << " ";
    }
}
```

Then press `F5` to start execution from the current line in the standard library header. Execution will break on `cout`. If you like, you can press `F9` again to toggle off the breakpoint.

When the loop has completed, you can see the output in the Integrated Terminal, along with some other diagnostic information that is output by GDB.

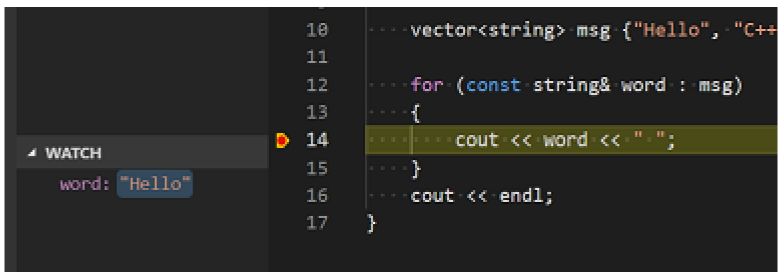


```
PS C:\Users\[redacted]\projects\helloworld> & 'c:\Users\[redacted]\.vscode\extensions\ms-vscode.cpptools-0.26.2\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-o5dkxhkw.c0j' '--stdout=Microsoft-MIEngine-Out-ae515rde.fxh' '--stderr=Microsoft-MIEngine-Error-zw3sp0s1.tbm' '--pid=Microsoft-MIEngine-Pid-2eycxsze.slv' '--dbgExe=C:\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Hello C++ World from VS Code and the C++ extension!
```

Set a watch

Sometimes you might want to keep track of the value of a variable as your program executes. You can do this by setting a **watch** on the variable.

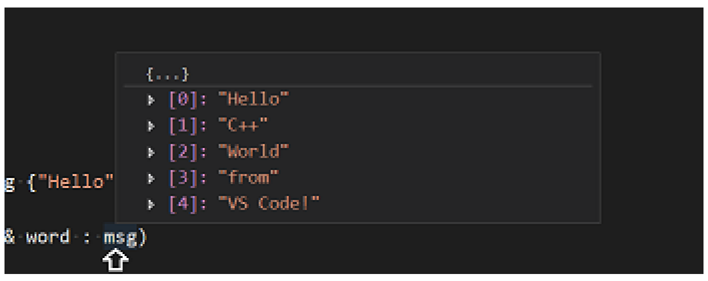
1. Place the insertion point inside the loop. In the **Watch** window, click the plus sign and in the text box, type `word`, which is the name of the loop variable. Now view the Watch window as you step through the loop.



```
    vector<string> msg {"Hello", "C++

    for (const string& word : msg)
    {
        cout << word << " ";
    }
    cout << endl;
}
```

2. Add another watch by adding this statement before the loop: `int i = 0;`. Then, inside the loop, add this statement: `++i;`. Now add a watch for `i` as you did in the previous step.

3. To quickly view the value of any variable while execution is paused on a breakpoint, you can hover over it with the mouse pointer.



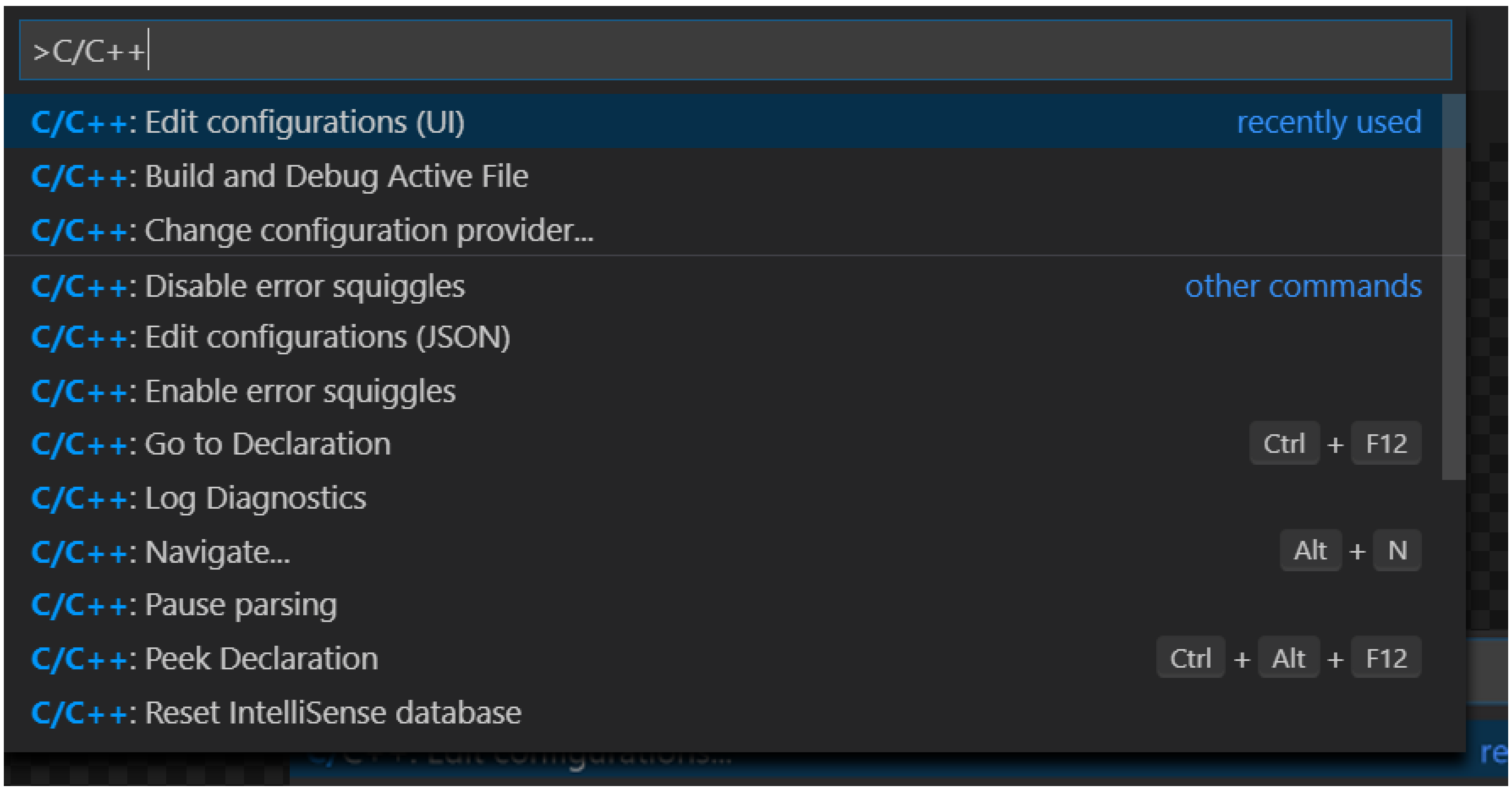
```
g {"Hello"
& word : msg)
```

```
{...}
 [0]: "Hello"
 [1]: "C++"
 [2]: "World"
 [3]: "from"
 [4]: "VS Code!"
```

C/C++ configurations

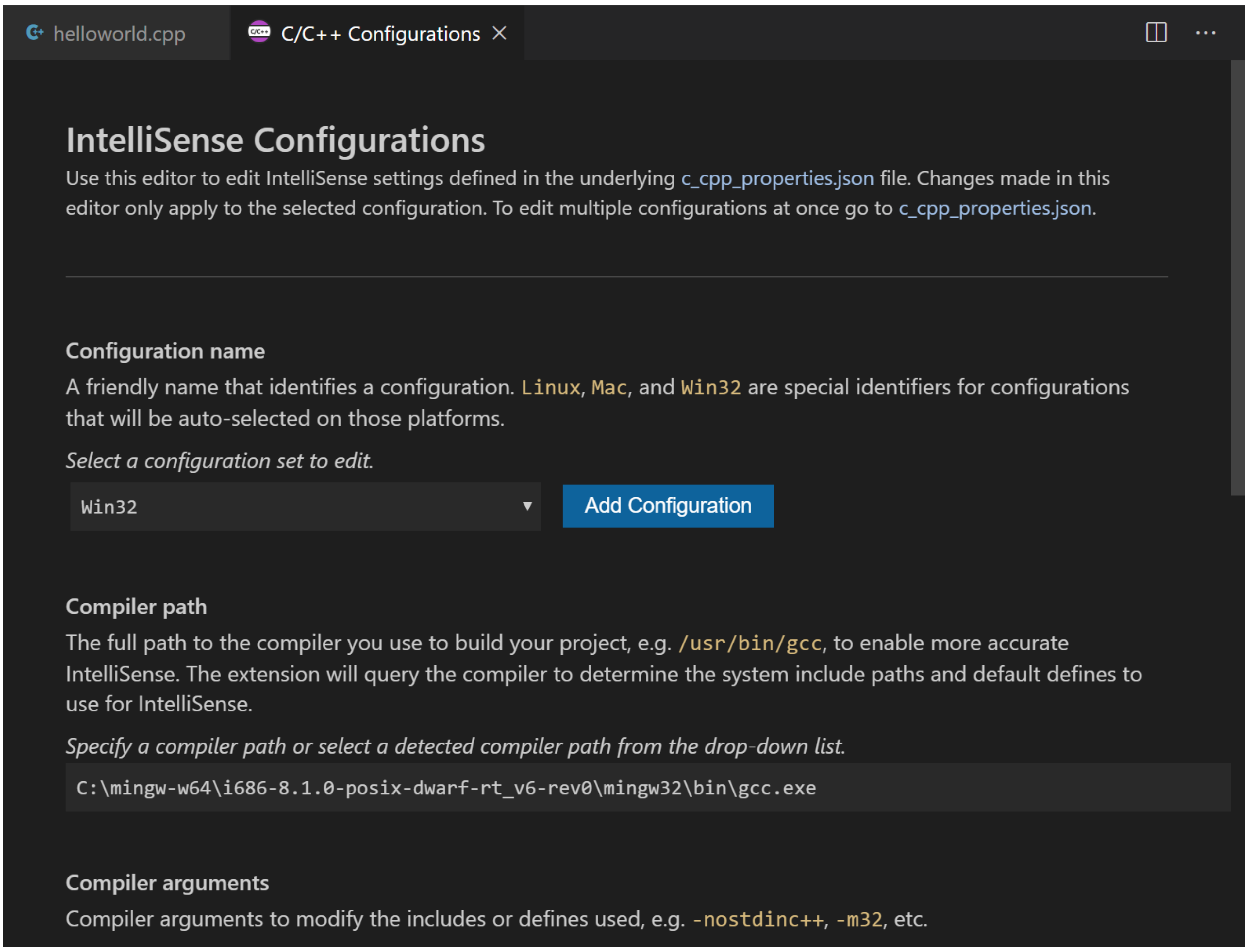
If you want more control over the C/C++ extension, you can create a `c_cpp_properties.json` file, which will allow you to change settings such as the path to the compiler, include paths, C++ standard (default is C++17), and more.

You can view the C/C++ configuration UI by running the command **C/C++: Edit Configurations (UI)** from the Command Palette (`Ctrl+Shift+P`).



>C/C++
C/C++: Edit configurations (UI) recently used
C/C++: Build and Debug Active File
C/C++: Change configuration provider...
C/C++: Disable error squiggles other commands
C/C++: Edit configurations (JSON)
C/C++: Enable error squiggles
C/C++: Go to Declaration Ctrl + F12
C/C++: Log Diagnostics
C/C++: Navigate... Alt + N
C/C++: Pause parsing
C/C++: Peek Declaration Ctrl + Alt + F12
C/C++: Reset IntelliSense database

This opens the **C/C++ Configurations** page. When you make changes here, VS Code writes them to a file called `c_cpp_properties.json` in the `.vscode` folder.



IntelliSense Configurations

Use this editor to edit IntelliSense settings defined in the underlying c_cpp_properties.json file. Changes made in this editor only apply to the selected configuration. To edit multiple configurations at once go to c_cpp_properties.json.

Configuration name

A friendly name that identifies a configuration. `Linux`, `Mac`, and `Win32` are special identifiers for configurations that will be auto-selected on those platforms.

Select a configuration set to edit.

`Win32` Add Configuration

Compiler path

The full path to the compiler you use to build your project, e.g. `/usr/bin/gcc`, to enable more accurate IntelliSense. The extension will query the compiler to determine the system include paths and default defines to use for IntelliSense.

Specify a compiler path or select a detected compiler path from the drop-down list.

```
C:\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gcc.exe
```

Compiler arguments

Compiler arguments to modify the includes or defines used, e.g. `-nostdinc++`, `-m32`, etc.

Visual Studio Code places these settings in `.vscode\c_cpp_properties.json` . If you open that file directly, it should look something like this:

```
{
  "configurations": [
    {
      "name": "Win32",
      "includePath": ["${workspaceFolder}/**"],
      "defines": ["_DEBUG", "UNICODE", "_UNICODE"],
      "compilerPath": "C:\\mingw-w64\\i686-8.1.0-posix-dwarf-rt_v6-rev0\\mingw32\\bin\\gcc.exe",
      "cStandard": "c11",
      "cppStandard": "c++17",
      "intelliSenseMode": "clang-x86"
    }
  ],
  "version": 4
}
```

You only need to add to the **Include path** array setting if your program includes header files that are not in your workspace or in the standard library path.

Compiler path

The `compilerPath` setting is an important setting in your configuration. The extension uses it to infer the path to the C++ standard library header files. When the extension knows where to find those files, it can provide useful features like smart completions and **Go to Definition** navigation.

The C/C++ extension attempts to populate `compilerPath` with the default compiler location based on what it finds on your system. The extension looks in several common compiler locations.

The `compilerPath` search order is:

- First check for the Microsoft Visual C++ compiler
- Then look for g++ on Windows Subsystem for Linux (WSL)
- Then g++ for Mingw-w64.

If you have Visual Studio or WSL installed, you may need to change `compilerPath` to match the preferred compiler for your project. For example, if you installed Mingw-w64 version 8.1.0 under C:\mingw-w64, using the Win32 threads and SEH exception handling options, the path would look like this: `C:\mingw-w64\x86_64-8.1.0-win32-seh-rt_v6-rev0\mingw64\bin\g++.exe`.

Next steps

- Explore the VS Code User Guide (/docs/editor/codebasics).
- Review the Overview of the C++ extension (/docs/languages/cpp).
- Create a new workspace, copy your `.vscode` JSON files to it, adjust the necessary settings for the new workspace path, program name, and so on, and start coding!

Was this documentation helpful?

Yes **No**

07/25/2019

IN THIS ARTICLE

Tweet this link (https://twitter.com/intent/tweet?original_referer=https://code.visualstudio.com/docs/cpp/config-mingw&ref_src=twsrc%5Etfw&text=Get%20Started%20with%20C++%20and%20Mingw-w64%20in%20Visual%20Studio%20Code&tw_p=tweetbutton&url=https://code.visualstudio.com/docs/cpp/config-mingw&via=code)

Subscribe(/feed.xml)

Ask questions(https://stackoverflow.com/questions/tagged/vscode)

Follow @code(https://go.microsoft.com/fwlink/?LinkID=533687)

Request features(https://go.microsoft.com/fwlink/?LinkID=533482)

Report issues(https://www.github.com/Microsoft/vscode/issues)

Watch videos(https://www.youtube.com/channel/UCs5Y5_7XK8HLDX0SLNwkd3w)