Using GCC with MinGW

In this tutorial, you configure Visual Studio Code to use the GCC C++ compiler (g++) and GDB debugger from mingw-w64 to create programs that run on Windows. After configuring VS Code, you will compile, run, and debug a Hello World program.

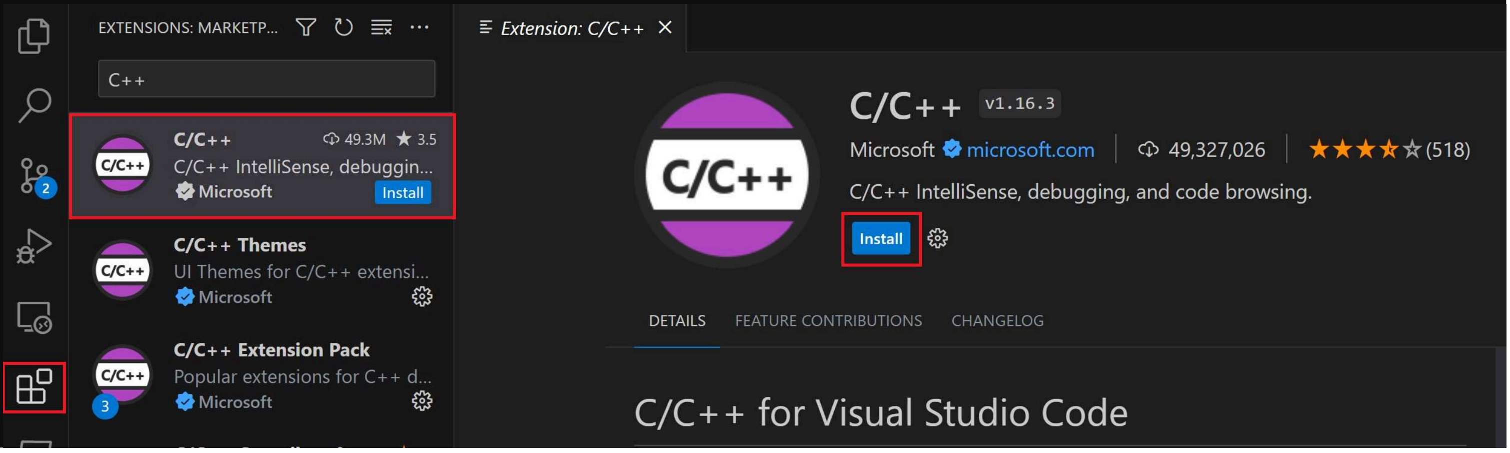
This tutorial does not teach you about GCC, GDB, minGW-w64, or the C++ language. For those subjects, there are many good resources available on the Web.

If you have any problems, feel free to file an issue for this tutorial in the VS Code documentation repository.

Prerequisites

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

1. Install Visual Studio Code.
2. Install the C/C++ extension for VS Code. You can install the C/C++ extension by searching for 'C++' in the Extensions view (`Ctrl+Shift+X`).



Installing the MinGW-w64 toolchain

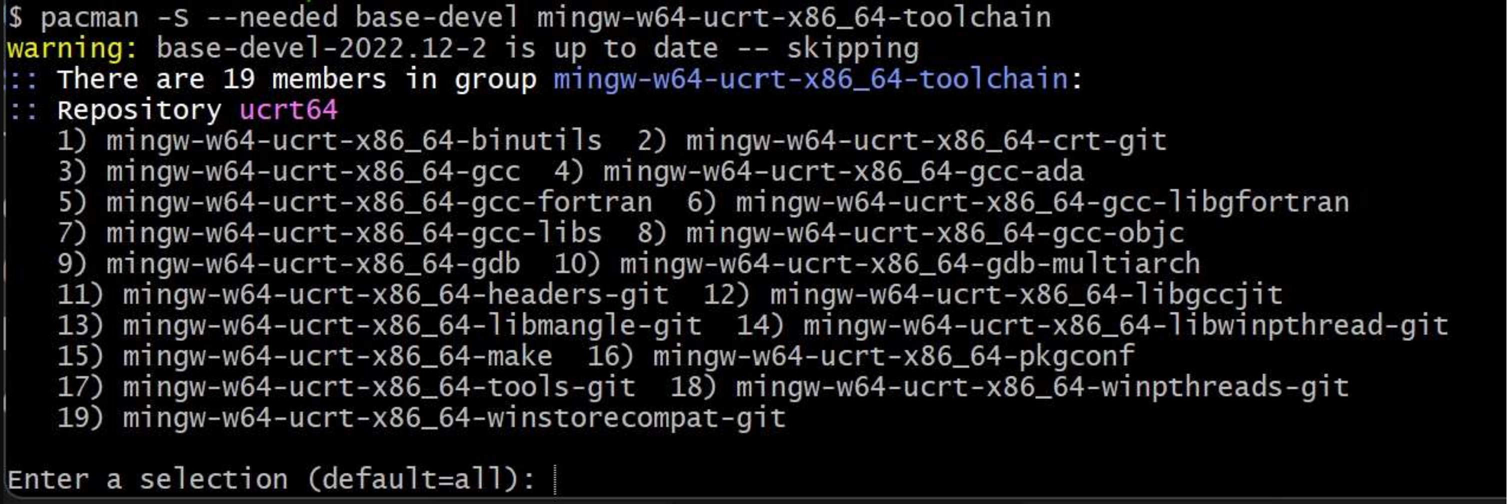
Get the latest version of MinGW-w64 via MSYS2, which provides up-to-date native builds of GCC, MinGW-w64, and other helpful C++ tools and libraries. This will provide you with the necessary tools to compile your code, debug it, and configure it to work with IntelliSense.

To install the MinGW-w64 toolchain, check out this video or follow the steps below:

1. You can download the latest installer from the MSYS2 page or use this **direct link to the installer**.
2. Run the installer and follow the steps of the installation wizard. Note that MSYS2 requires 64 bit Windows 8.1 or newer.
3. In the wizard, choose your desired Installation Folder. Record this directory for later. In most cases, the recommended directory is acceptable. The same applies when you get to setting the start menu shortcuts step. When complete, ensure the **Run MSYS2 now** box is checked and select **Finish**. This will open a MSYS2 terminal window for you.
4. In this terminal, install the MinGW-w64 toolchain by running the following command:

```
pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
```

5. Accept the default number of packages in the `toolchain` group by pressing `Enter`.



```
$ pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
warning: base-devel-2022.12-2 is up to date -- skipping
:: There are 19 members in group mingw-w64-ucrt-x86_64-toolchain:
:: Repository ucrt64
   1) mingw-w64-ucrt-x86_64-binutils  2) mingw-w64-ucrt-x86_64-crt-git
   3) mingw-w64-ucrt-x86_64-gcc  4) mingw-w64-ucrt-x86_64-gcc-ada
   5) mingw-w64-ucrt-x86_64-gcc-fortran  6) mingw-w64-ucrt-x86_64-gcc-libgfortran
   7) mingw-w64-ucrt-x86_64-gcc-libs  8) mingw-w64-ucrt-x86_64-gcc-objc
   9) mingw-w64-ucrt-x86_64-gdb  10) mingw-w64-ucrt-x86_64-gdb-multiarch
   11) mingw-w64-ucrt-x86_64-headers-git  12) mingw-w64-ucrt-x86_64-libgccjit
   13) mingw-w64-ucrt-x86_64-libmangle-git  14) mingw-w64-ucrt-x86_64-libwinpthread-git
   15) mingw-w64-ucrt-x86_64-make  16) mingw-w64-ucrt-x86_64-pkgconf
   17) mingw-w64-ucrt-x86_64-tools-git  18) mingw-w64-ucrt-x86_64-winpthreads-git
   19) mingw-w64-ucrt-x86_64-winstorecompat-git

Enter a selection (default=all):
```

6. Enter `Y` when prompted whether to proceed with the installation.
7. Add the path of your MinGW-w64 `bin` folder to the Windows `PATH` environment variable by using the following steps:
   1. In the Windows search bar, type **Settings** to open your Windows Settings.
   2. Search for **Edit environment variables for your account**.
   3. In your **User variables**, select the `Path` variable and then select **Edit**.
   4. Select **New** and add the MinGW-w64 destination folder you recorded during the installation process to the list. If you used the default settings above, then this will be the path: `C:\msys64\ucrt64\bin`.
   5. Select **OK**, and then select **OK** again in the **Environment Variables** window to update the `PATH` environment variable. You have to reopen any console windows for the updated `PATH` environment variable 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:

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

You should see output that states which versions of GCC, g++ and GDB you have installed. If this is not the case:

1. Make sure your PATH variable entry matches the MinGW-w64 binary location where the toolchain was installed. If the compilers do not exist at that PATH entry, make sure you followed the previous instructions.
2. If `gcc` has the correct output but not `gdb`, then you need to install the packages you are missing from the MinGW-w64 toolset.
   - If on compilation you are getting the "The value of miDebuggerPath is invalid." message, one cause can be you are missing the `mingw-w64-gdb` package.

Create a Hello World app

First, lets get a project set up.

1. Launch a Windows command prompt (Enter **Windows command prompt** in the Windows search bar).
2. Run the following commands. These will create an empty folder called `projects` where you can place all your VS Code projects. There, the next commands will create and navigate to a subfolder called `helloworld`. From there, you will open `helloworld` directly in VS Code.

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

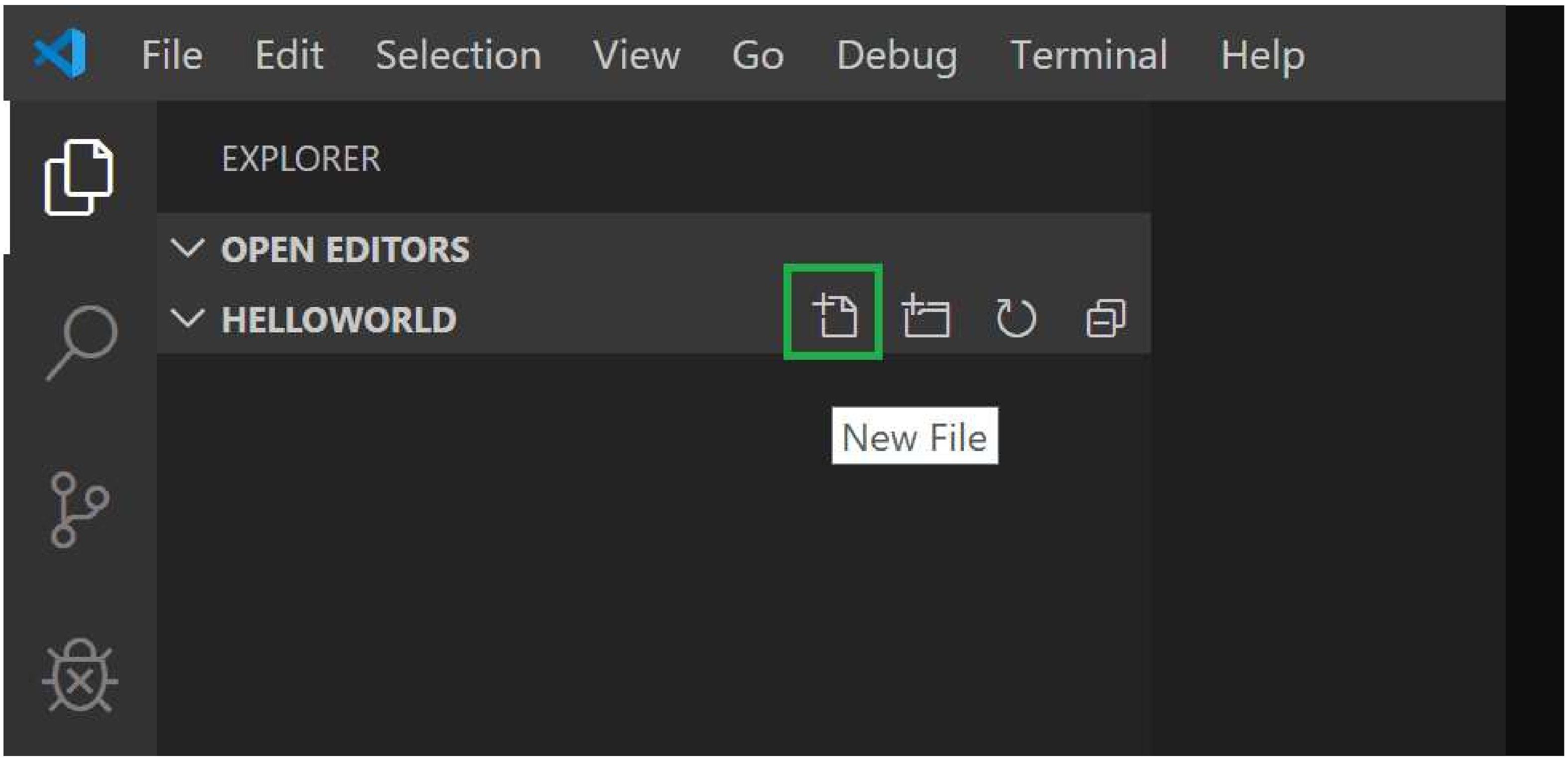
The "code ." command opens VS Code in the current working folder, which becomes your "workspace". Accept the Workspace Trust dialog by selecting **Yes, I trust the authors** since this is a folder you created.

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 Hello World source code file

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



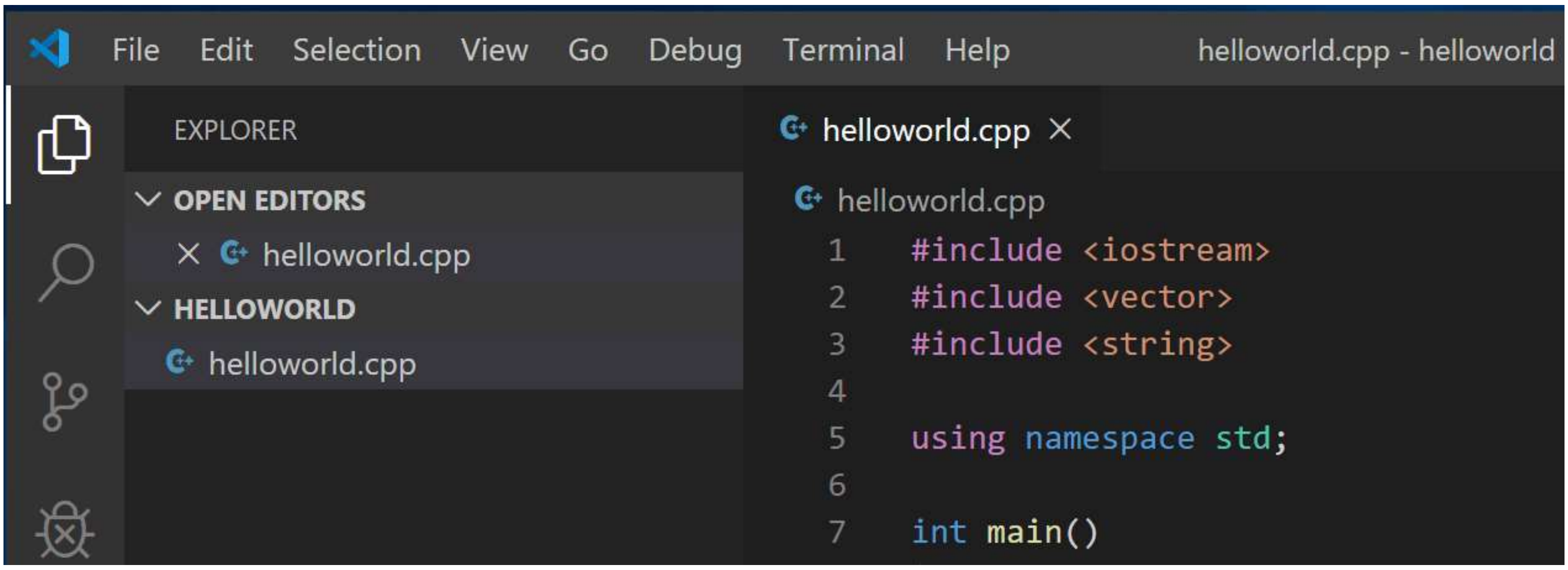
Add hello world source code

Now paste in this source code:

```cpp
#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;
}
```

Explicar

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:



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

using namespace std;

int main()
```

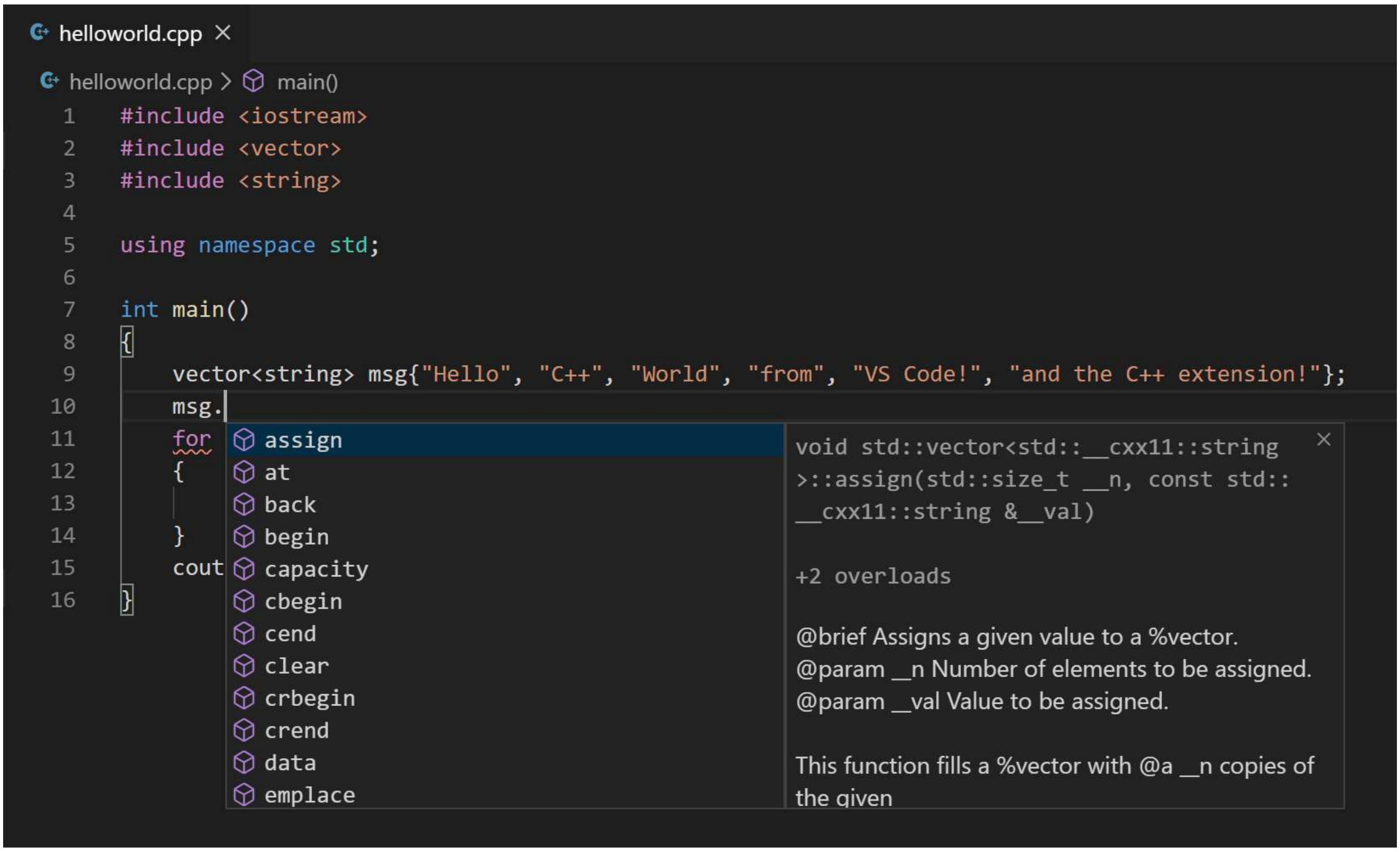
You can also enable Auto Save to automatically save your file changes, by selecting **File** > **Auto Save**. You can find out more about the other views in the VS Code User Interface documentation.

> **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

IntelliSense is a tool to help you code faster and more efficiently by adding code editing features such as code completion, parameter info, quick info, and member lists.

To see IntelliSense in action, hover over `vector` or `string` to see their type information. If you type `msg.` in line 10, you can see a completion list of recommended member functions to call, all generated by IntelliSense:



```cpp
#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
```

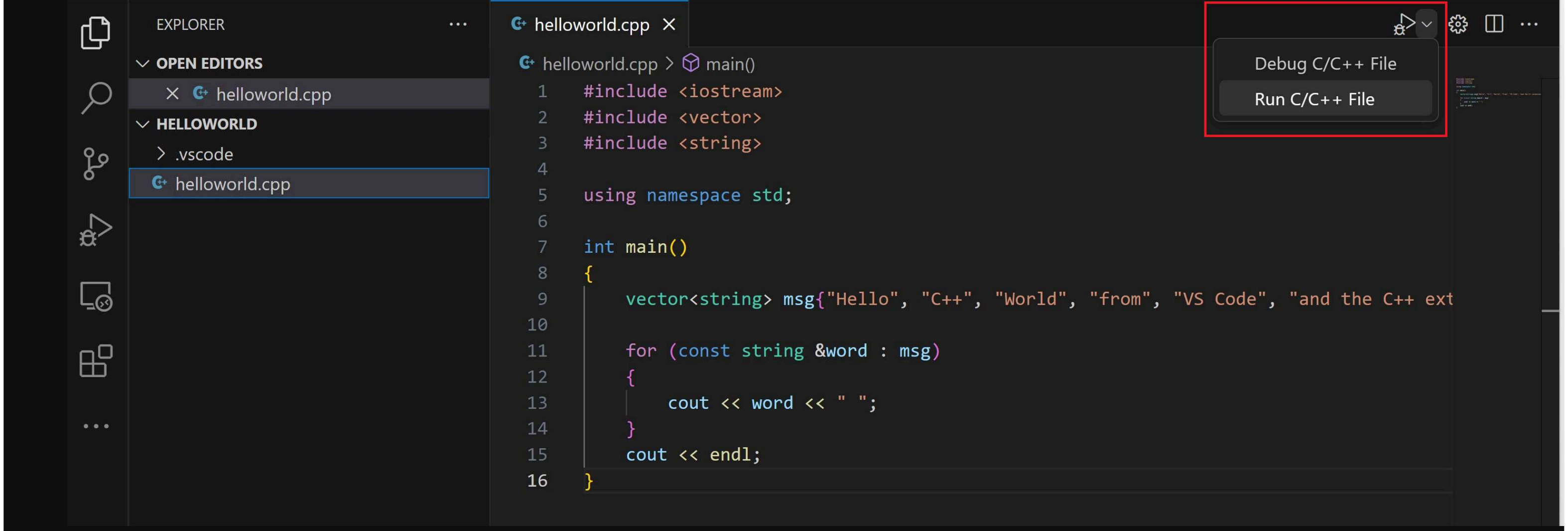
You can press the `Tab` key to insert a selected member. If you then add open parenthesis, IntelliSense will show information on which arguments are required.

If IntelliSense is not already configured, open the Command Palette (`Ctrl+Shift+P`) and enter **Select IntelliSense Configuration**. From the dropdown of compilers, select `Use gcc.exe` to configure. More information can be found in the IntelliSense configuration documentation.

Run helloworld.cpp

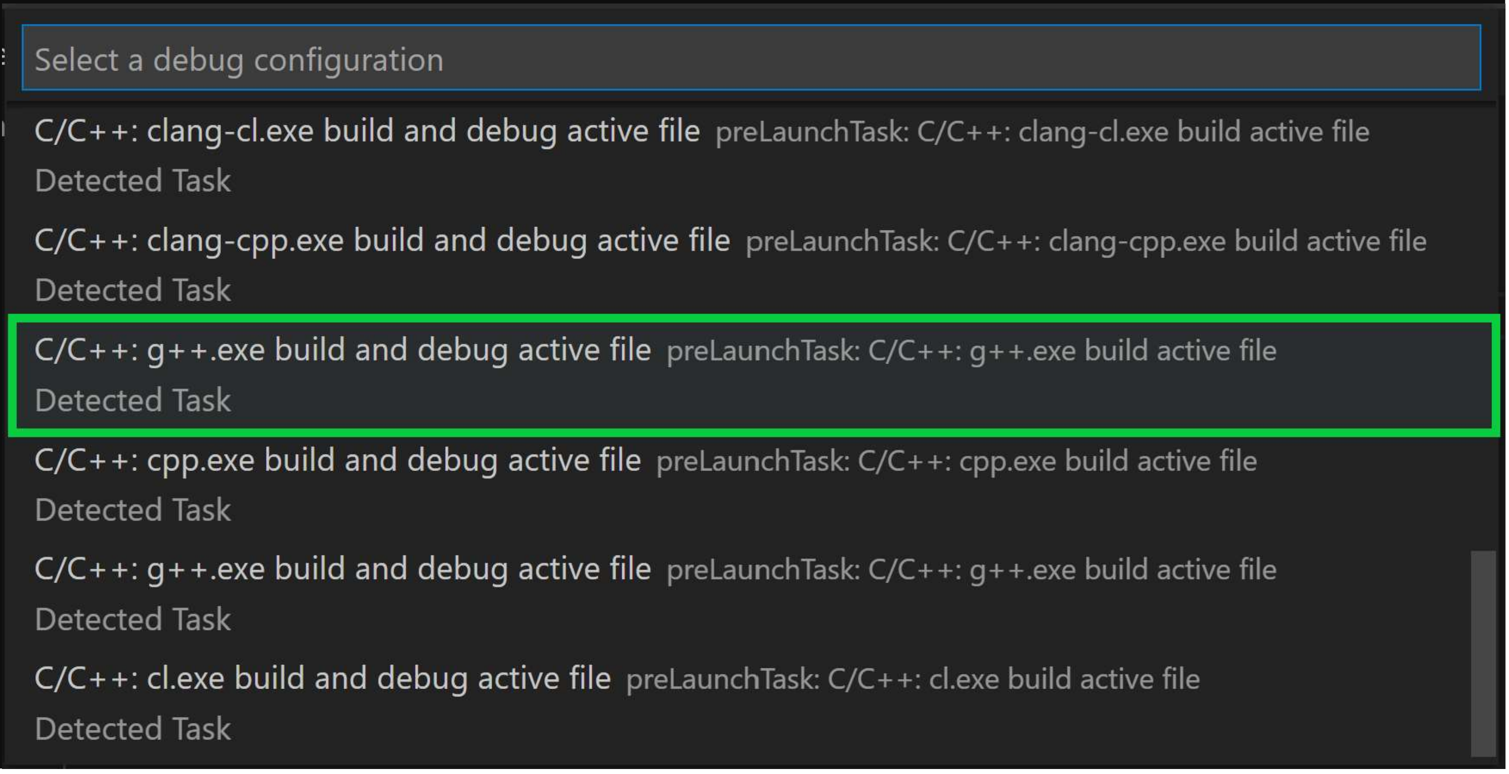
Remember, the C++ extension uses the C++ compiler you have installed on your machine to build your program. Make sure you have completed the "Installing the MinGW-w64 toolchain" step before attempting to run and debug `helloworld.cpp` in VS Code.

1. Open `helloworld.cpp` so that it is the active file.
2. Press the play button in the top right corner of the editor.



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

using namespace std;

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

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

3. Choose **C/C++: g++.exe build and debug active file** from the list of detected compilers on your system.



Select a debug configuration

C/C++: clang-cl.exe build and debug active file preLaunchTask: C/C++: clang-cl.exe build active file
Detected Task

C/C++: clang-cpp.exe build and debug active file preLaunchTask: C/C++: clang-cpp.exe build active file
Detected Task

C/C++: g++.exe build and debug active file preLaunchTask: C/C++: g++.exe build active file
Detected Task

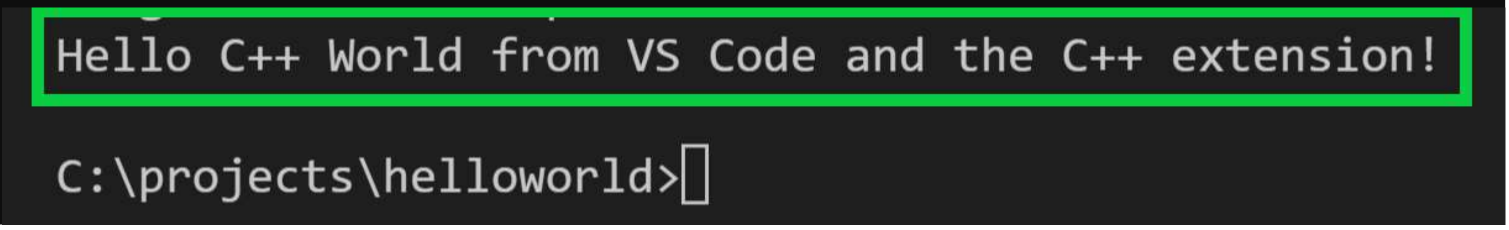
C/C++: cpp.exe build and debug active file preLaunchTask: C/C++: cpp.exe build active file
Detected Task

C/C++: g++.exe build and debug active file preLaunchTask: C/C++: g++.exe build active file
Detected Task

C/C++: cl.exe build and debug active file preLaunchTask: C/C++: cl.exe build active file
Detected Task

You'll only be asked to choose a compiler the first time you run `helloworld.cpp`. This compiler will be set as the "default" compiler in `tasks.json` file.

4. After the build succeeds, your program's output will appear in the integrated **Terminal**.



```
Hello C++ World from VS Code and the C++ extension!

C:\projects\helloworld>
```

Congratulations! You've just run your first C++ program in VS Code!

Understanding tasks.json

The first time you run your program, the C++ extension creates a `tasks.json` file, which you'll find in your project's `.vscode` folder. `tasks.json` stores your build configurations.

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

```json
{
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: g++.exe build active file",
      "command": "C:\\msys64\\ucrt64\\bin\\g++.exe",
      "args": [
        "-fdiagnostics-color=always",
        "-g",
        "${file}",
        "-o",
        "${fileDirname}\\${fileBasenameNoExtension}.exe"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": ["$gcc"],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "detail": "Task generated by Debugger."
    }
  ],
  "version": "2.0.0"
}
```

> **Note**: You can learn more about `tasks.json` variables in the variables reference.

The `command` setting specifies the program to run; in this case that is `g++`.

The `args` array specifies the command-line arguments passed to g++. These arguments are listed in this file in the specific order expected by the compiler.

This task tells g++ to take the active file (`${file}`), compile it, and create an output file (`-o` switch) in the current directory (`${fileDirname}`) with the same name as the active file but with the `.exe` extension (`${fileBasenameNoExtension}.exe`). For us, this results in `helloworld.exe`.

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

The `detail` value is what you will see as the description of the task in the tasks list. It's highly recommended to rename this value to differentiate it from similar tasks.

The `problemMatcher` value selects the output parser to use for finding errors and warnings in the compiler output. For GCC, you'll get the best results if you use the `$gcc` problem matcher.

From now on, the play button will read from `tasks.json` to figure out how to build and run your program. You can define multiple build tasks in `tasks.json`, and whichever task is marked as the default will be used by the play button. In case you need to change the default compiler, you can run **Tasks: Configure Default Build Task** in the Command Palette. Alternatively you can modify the `tasks.json` file and remove the default by replacing this segment:

```
    "group": {
        "kind": "build",
        "isDefault": true
    },
```

with this:

```
    "group": "build",
```

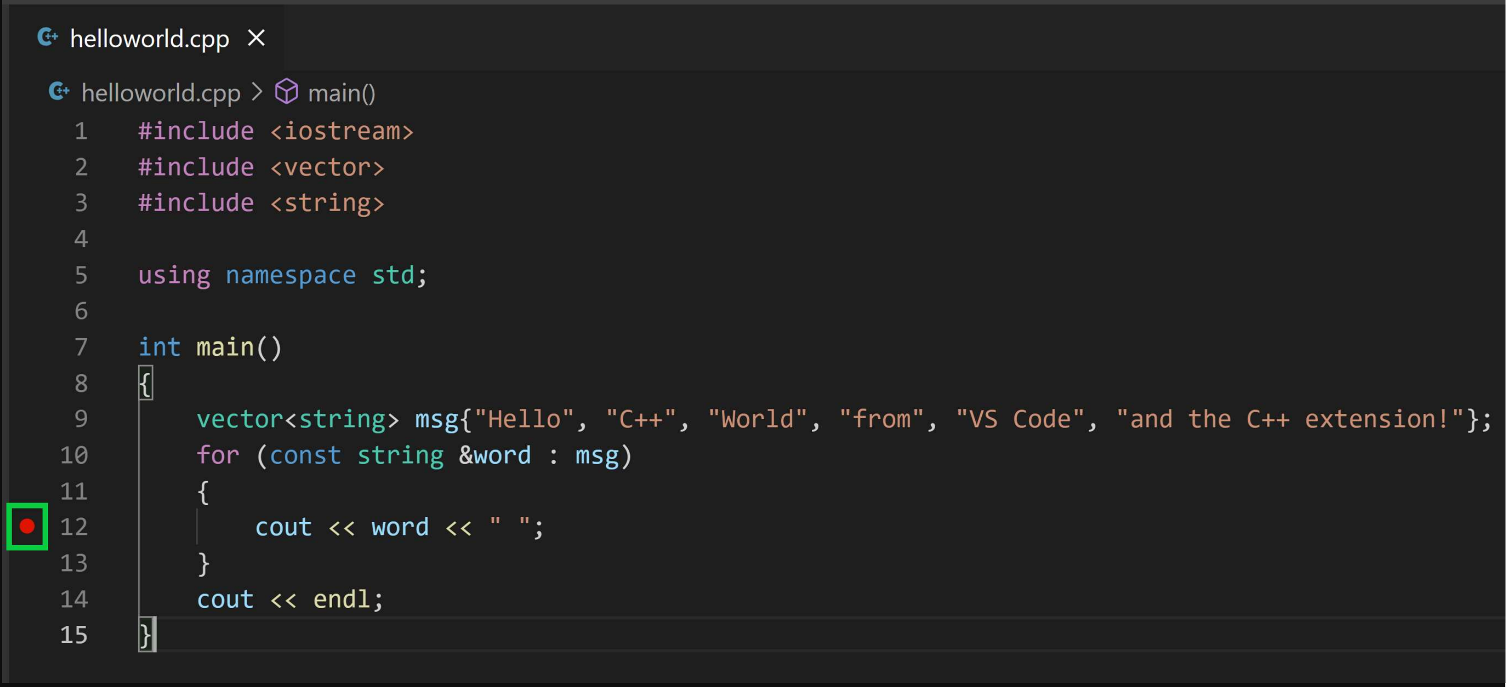
Modifying tasks.json

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

To debug your code,

1. Go back to `helloworld.cpp` so that it is the active file.
2. Set a breakpoint by clicking on the editor margin or using F9 on the current line.

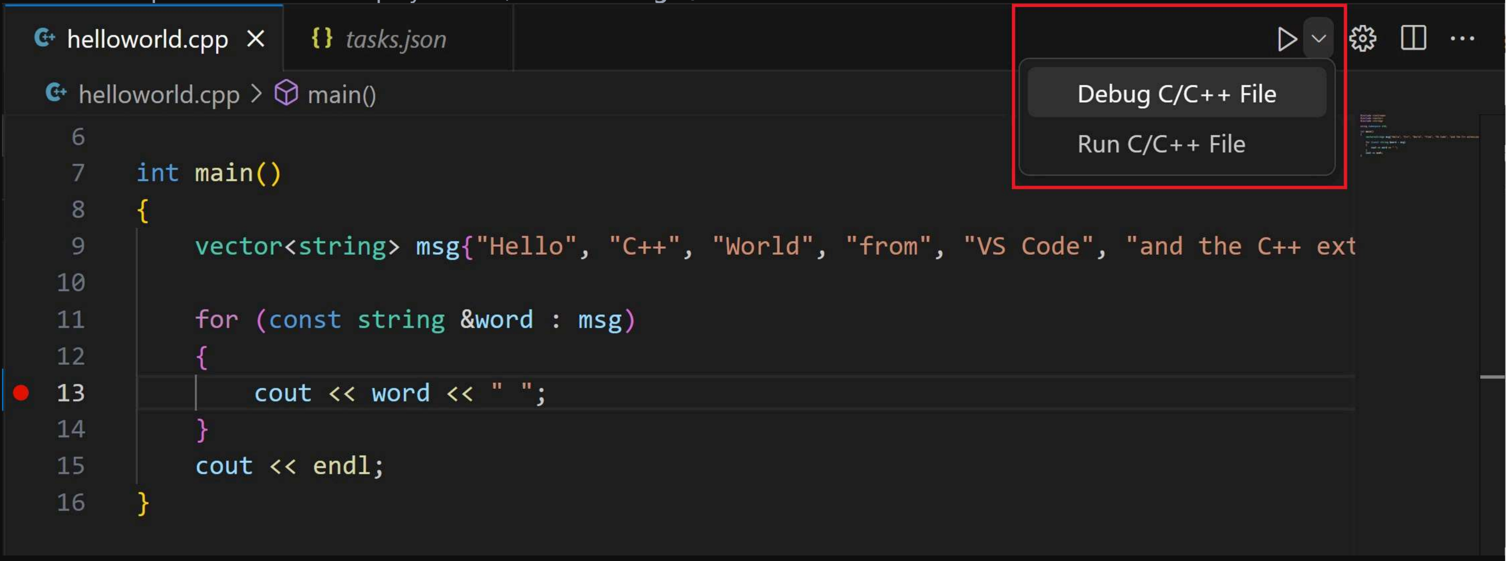


helloworld.cpp ×

helloworld.cpp > main()

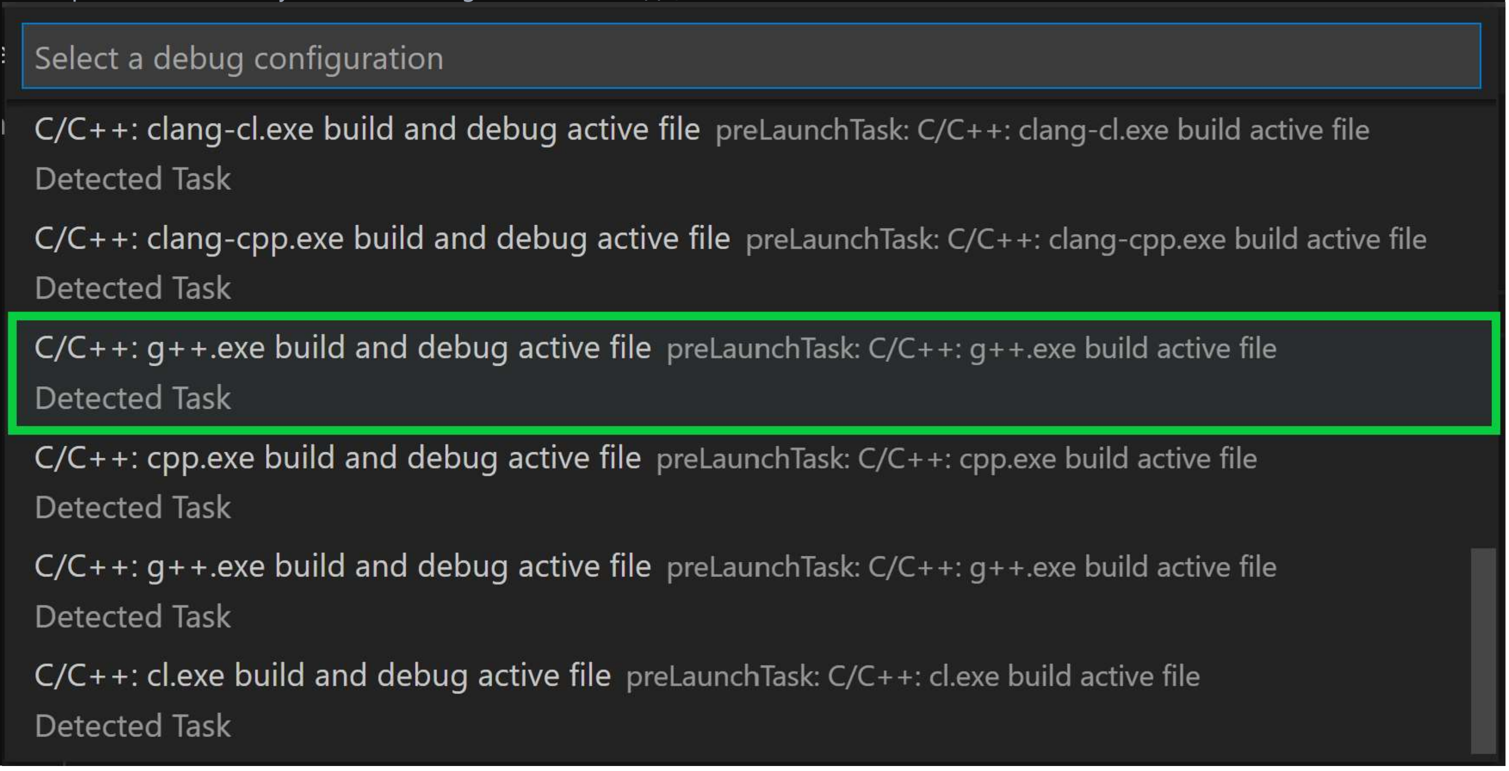
```cpp
#include <iostream>
#include <vector>
#include <string>

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;
}
```

3. From the drop-down next to the play button, select **Debug C/C++ File**.



```cpp
int main()
{
    vector<string> msg{"Hello", "C++", "World", "from", "VS Code", "and the C++ ext

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

4. Choose **C/C++: g++ build and debug active file** from the list of detected compilers on your system (you'll only be asked to choose a compiler the first time you run or debug `helloworld.cpp`).



Select a debug configuration

C/C++: clang-cl.exe build and debug active file preLaunchTask: C/C++: clang-cl.exe build active file
Detected Task

C/C++: clang-cpp.exe build and debug active file preLaunchTask: C/C++: clang-cpp.exe build active file
Detected Task

C/C++: g++.exe build and debug active file preLaunchTask: C/C++: g++.exe build active file
Detected Task

C/C++: cpp.exe build and debug active file preLaunchTask: C/C++: cpp.exe build active file
Detected Task

C/C++: g++.exe build and debug active file preLaunchTask: C/C++: g++.exe build active file
Detected Task

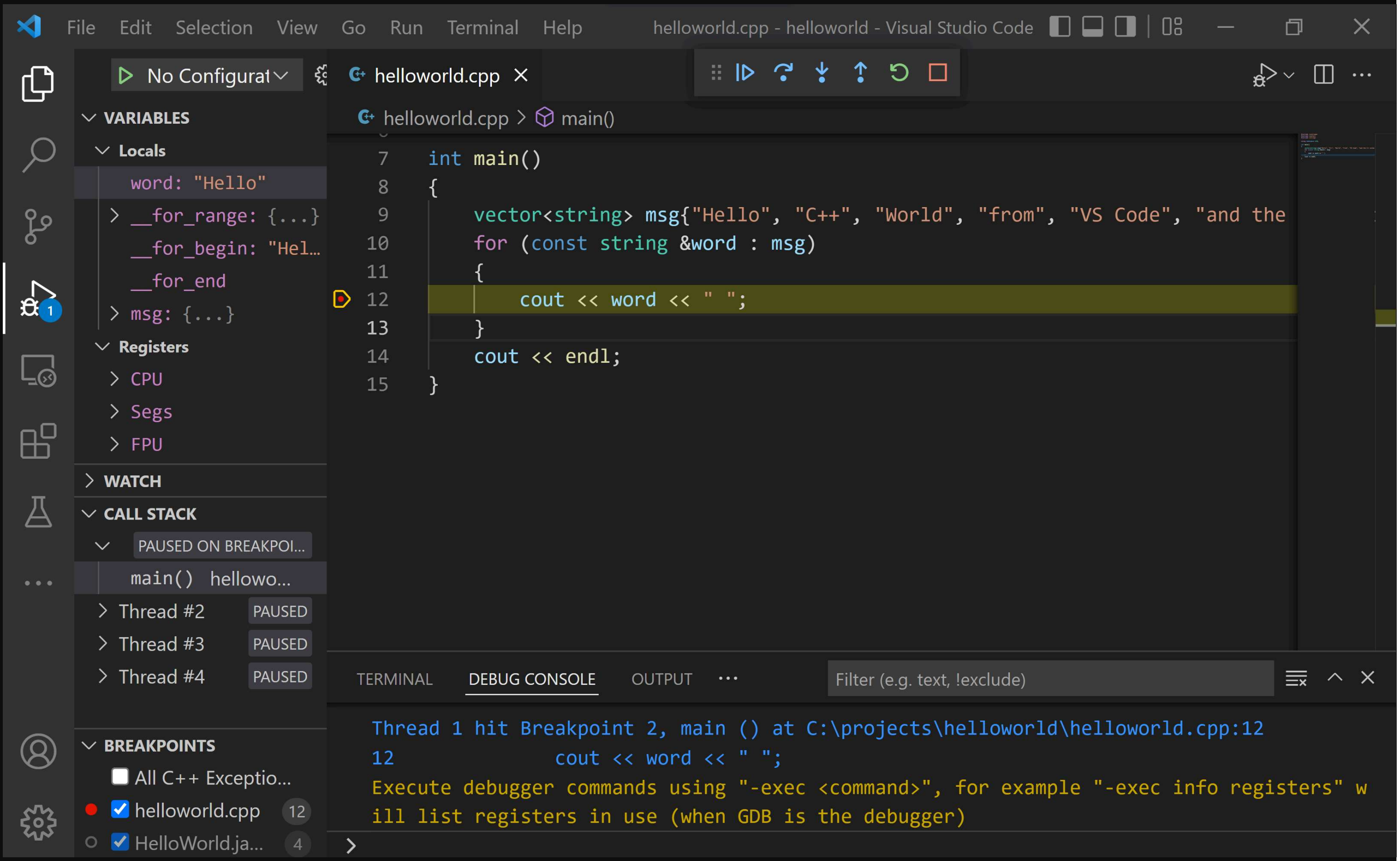
C/C++: cl.exe build and debug active file preLaunchTask: C/C++: cl.exe build active file
Detected Task

The play button has two modes: **Run C/C++ File** and **Debug C/C++ File**. It will default to the last-used mode. If you see the debug icon in the play button, you can just select the play button to debug, instead of using the drop-down.

Explore the debugger

Before you start stepping through the 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 Console** tab, you see output that indicates the debugger is up and running.
- The editor highlights the line where you set a breakpoint before starting the debugger:

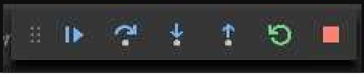


```
word: "Hello"
> __for_range: {...}
  __for_begin: "Hel…
  __for_end
> msg: {...}
```

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

```
Thread 1 hit Breakpoint 2, main () at C:\projects\helloworld\helloworld.cpp:12
12              cout << word << " ";
Execute debugger commands using "-exec <command>", for example "-exec info registers" w
ill list registers in use (when GDB is the debugger)
```

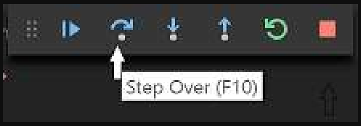
- The **Run and Debug** view on the left shows debugging information. You'll see an example later in the tutorial.
- At the top of the code editor, a debugging control panel appears. You can move this around the screen by grabbing the dots on the left side.



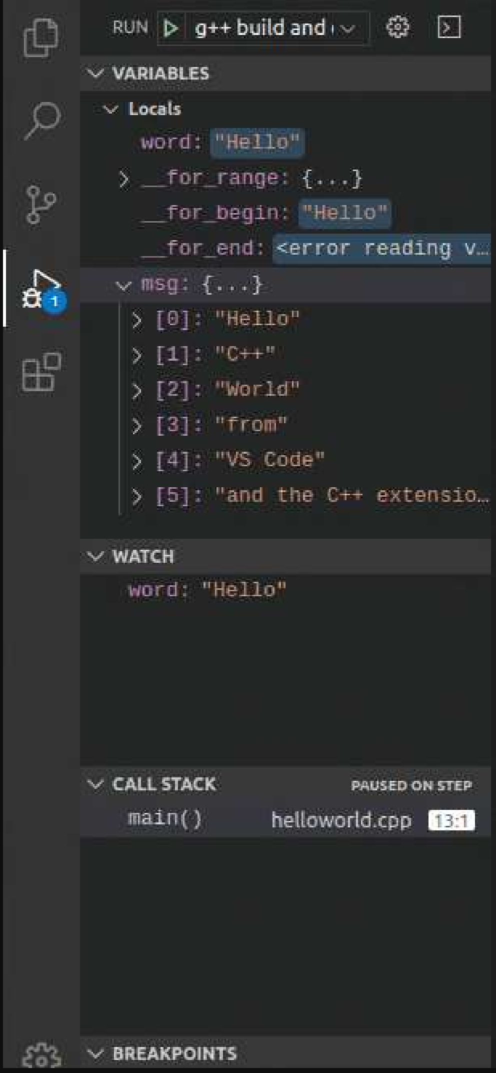
Step through the code

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

1. Select 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 v…
v msg: {...}
  > [0]: "Hello"
  > [1]: "C++"
  > [2]: "World"
  > [3]: "from"
  > [4]: "VS Code"
  > [5]: "and the C++ extensio…
```

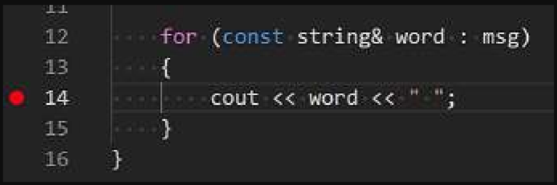
```
word: "Hello"
```

```
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 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!

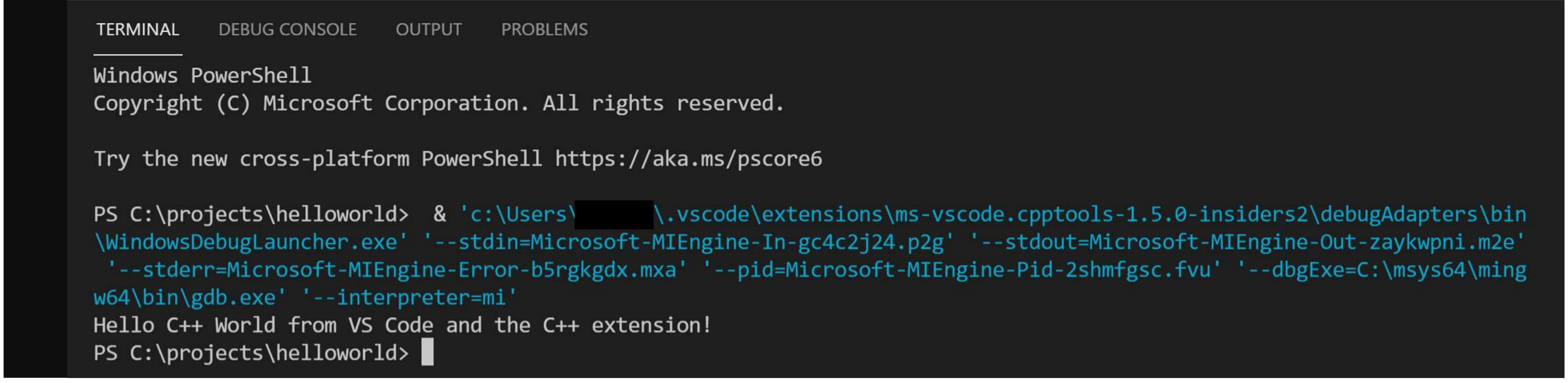
   To return to your own code, one way is to keep pressing **Step over**. Another way is to set a breakpoint in your code by switching to the `helloworld.cpp` tab in the code editor, putting the insertion point somewhere on the `cout` statement inside the loop, and pressing `F9`. A red dot appears in the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

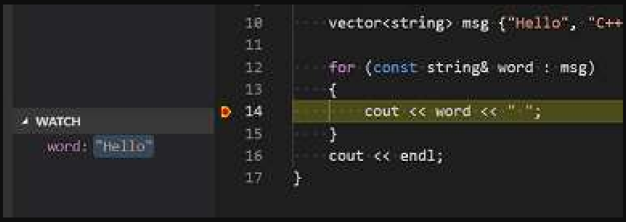


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\projects\helloworld>  & 'c:\Users\[redacted]\.vscode\extensions\ms-vscode.cpptools-1.5.0-insiders2\debugAdapters\bin
\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-gc4c2j24.p2g' '--stdout=Microsoft-MIEngine-Out-zaykwpni.m2e'
 '--stderr=Microsoft-MIEngine-Error-b5rgkgdx.mxa' '--pid=Microsoft-MIEngine-Pid-2shmfgsc.fvu' '--dbgExe=C:\msys64\ming
w64\bin\gdb.exe' '--interpreter=mi'
Hello C++ World from VS Code and the C++ extension!
PS C:\projects\helloworld> 
```

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.

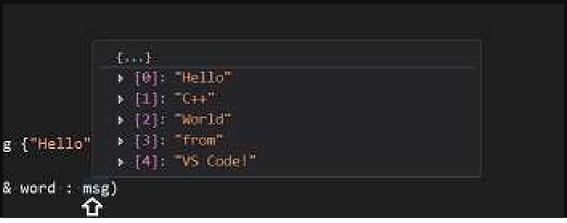
1. Place the insertion point inside the loop. In the **Watch** window, select the plus sign and in the text box, type `word`, which is the name of the loop variable. Now view the Watch window as you step through the loop.



```
    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.



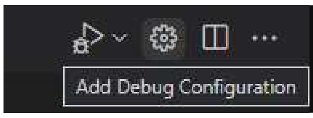
```
g {"Hello"    {...}
              [0]: "Hello"
              [1]: "C++"
              [2]: "World"
              [3]: "from"
              [4]: "VS Code!"
& word : msg)
```

Customize debugging with launch.json

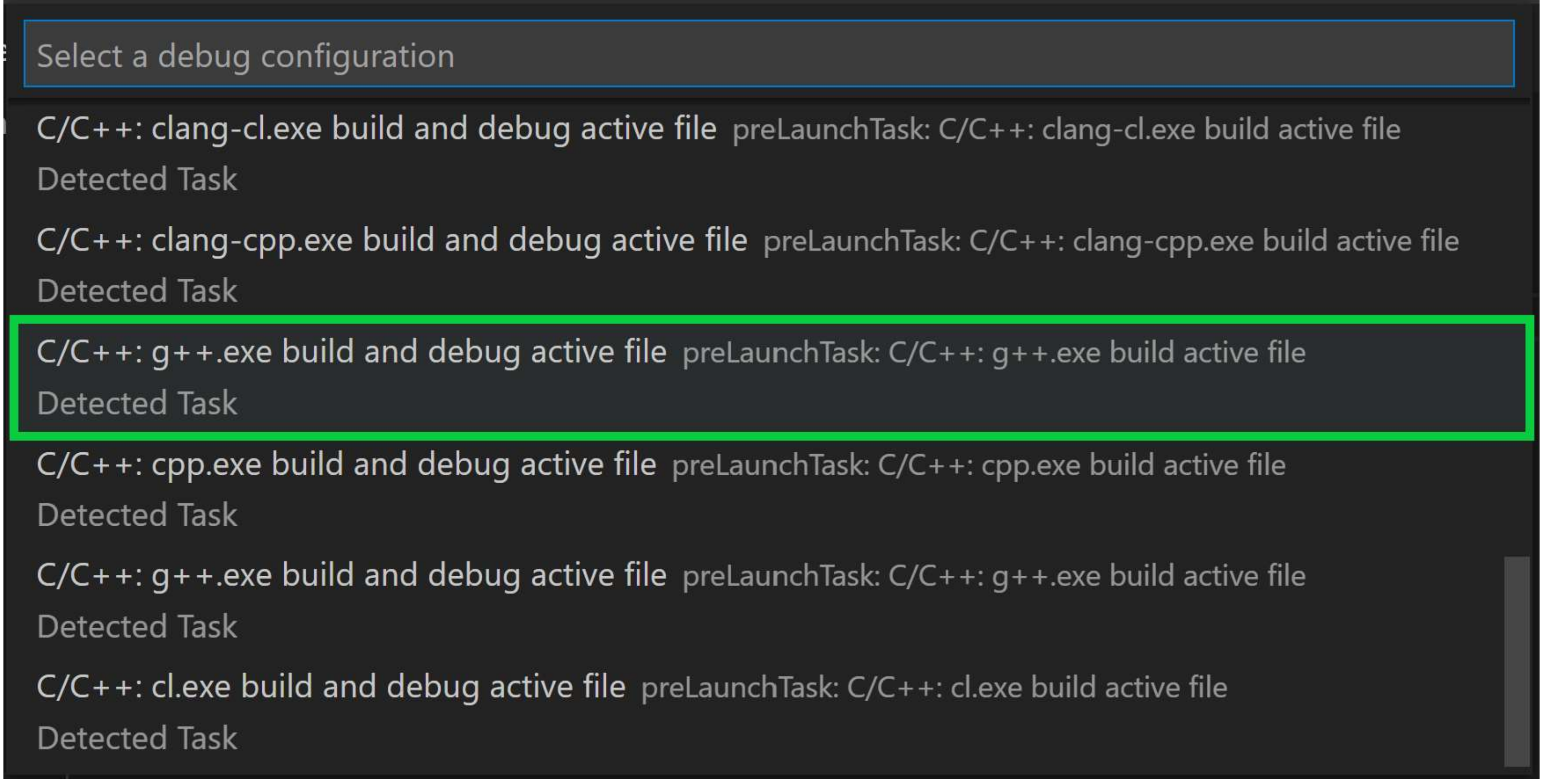
When you debug with the play button or `F5`, the C++ extension creates a dynamic debug configuration on the fly.

There are cases where you'd want to customize your debug configuration, such as specifying arguments to pass to the program at runtime. You can define custom debug configurations in a `launch.json` file.

To create `launch.json`, choose **Add Debug Configuration** from the play button drop-down menu.



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


Select a debug configuration

C/C++: clang-cl.exe build and debug active file preLaunchTask: C/C++: clang-cl.exe build active file
Detected Task

C/C++: clang-cpp.exe build and debug active file preLaunchTask: C/C++: clang-cpp.exe build active file
Detected Task

C/C++: g++.exe build and debug active file preLaunchTask: C/C++: g++.exe build active file
Detected Task

C/C++: cpp.exe build and debug active file preLaunchTask: C/C++: cpp.exe build active file
Detected Task

C/C++: g++.exe build and debug active file preLaunchTask: C/C++: g++.exe build active file
Detected Task

C/C++: cl.exe build and debug active file preLaunchTask: C/C++: cl.exe build active file
Detected Task

VS Code creates a `launch.json` file in the `.vscode` folder`, which looks something like this:

```
{
  "configurations": [
    {
      "name": "C/C++: g++.exe build and debug active file",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${fileDirname}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "C:\\msys64\\ucrt64\\bin\\gdb.exe",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        },
        {
          "description": "Set Disassembly Flavor to Intel",
          "text": "-gdb-set disassembly-flavor intel",
          "ignoreFailures": true
        }
      ],
      "preLaunchTask": "C/C++: g++.exe build active file"
    }
  ],
  "version": "2.0.0"
}
```

Explicar

In the JSON above, `program` 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`. The `args` property is an array of arguments to pass to the program at runtime.

By default, the C++ extension won't add any breakpoints to your source code and the `stopAtEntry` value is set to `false`.

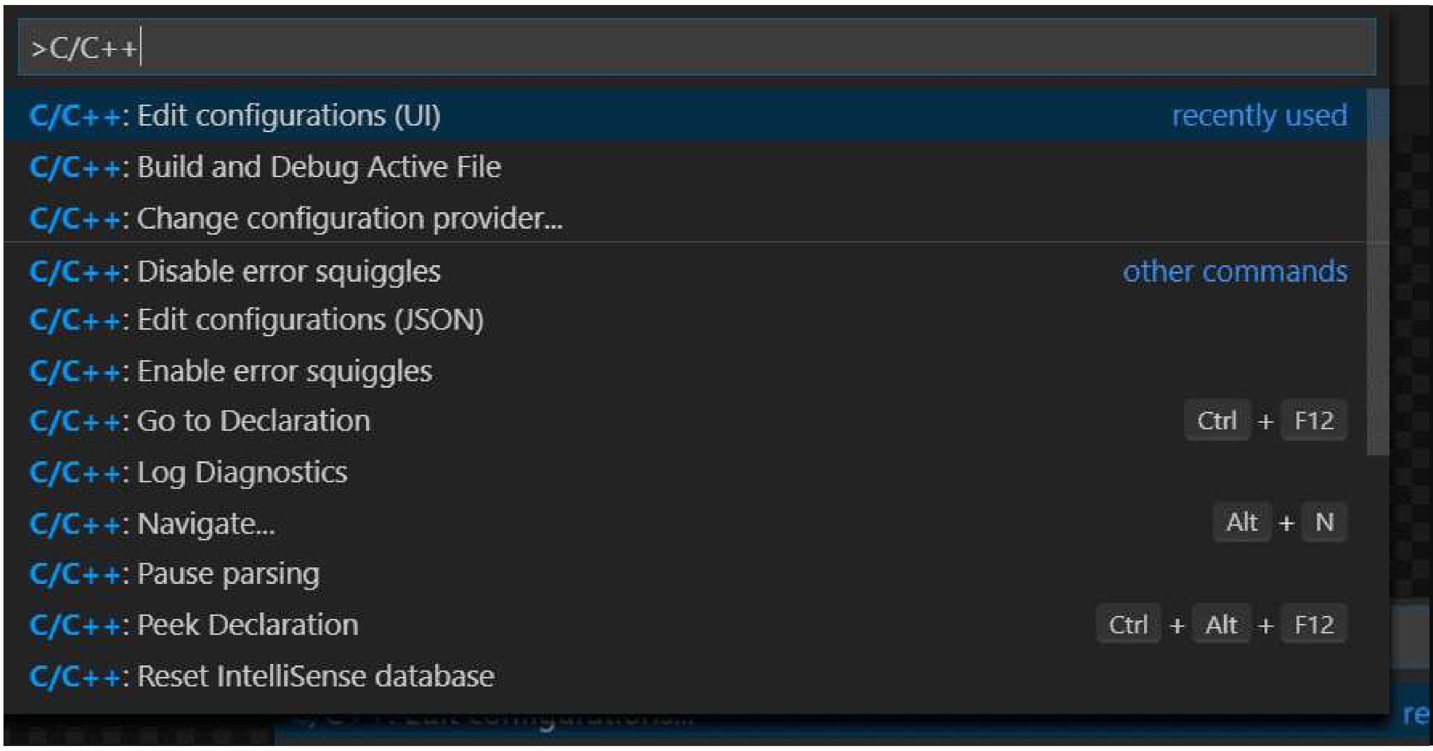
Change the `stopAtEntry` value to `true` to cause the debugger to stop on the `main` method when you start debugging.

> From now on, the play button and `F5` will read from your `launch.json` file when launching your program for debugging.

Adding additional C/C++ settings

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.

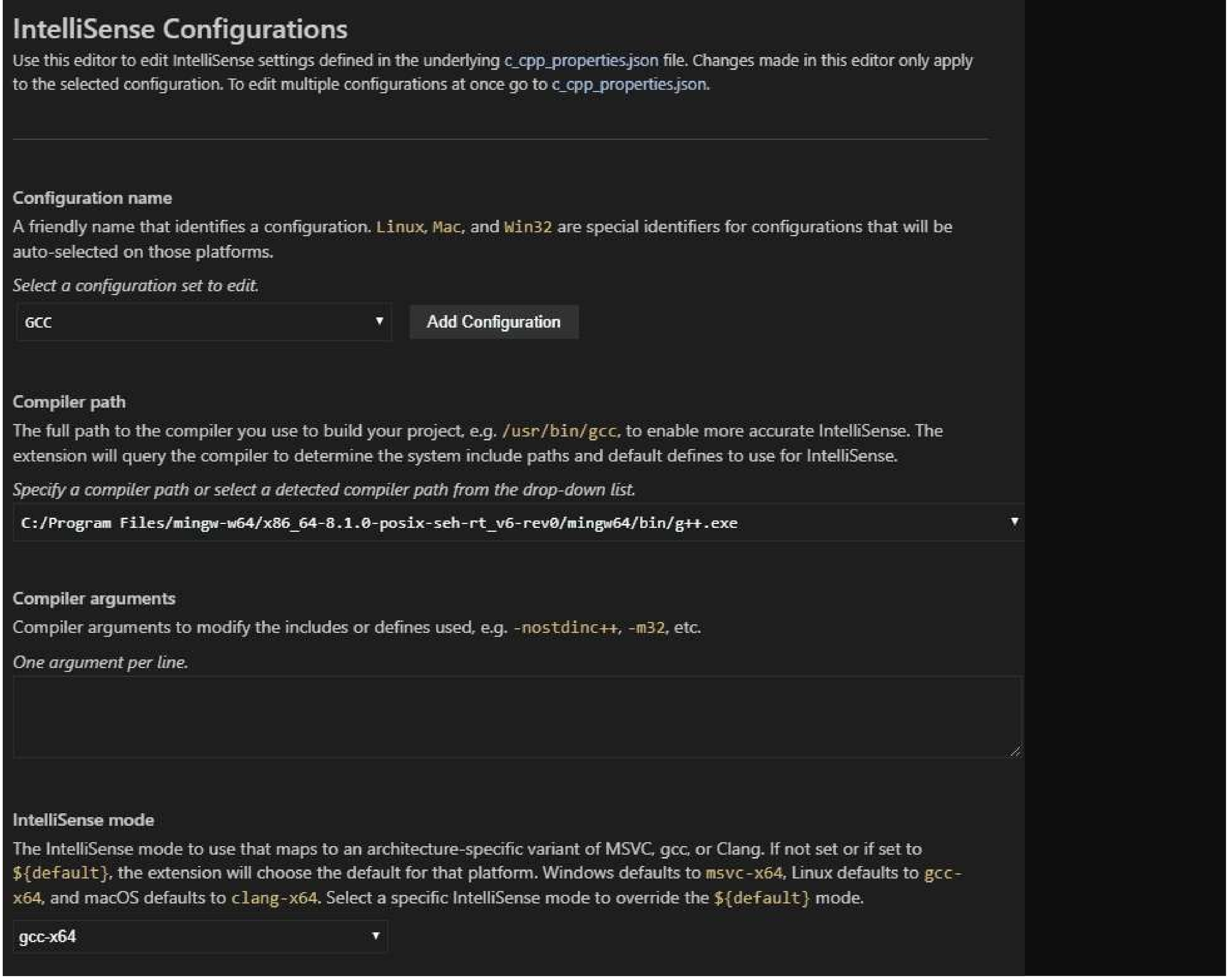
You can view the C/C++ configuration UI by running the command **C/C++: Edit Configurations (UI)** from the Command Palette (`Ctrl+Shift+P`).



```
>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.

Here, we've changed the **Configuration name** to **GCC**, set the **Compiler path** dropdown to the g++ compiler, and the **IntelliSense mode** to match the compiler (**gcc-x64**).


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.

GCC

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:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/bin/g++.exe`

Compiler arguments

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

One argument per line.

IntelliSense mode

The IntelliSense mode to use that maps to an architecture-specific variant of MSVC, gcc, or Clang. If not set or if set to `${default}`, the extension will choose the default for that platform. Windows defaults to `msvc-x64`, Linux defaults to `gcc-x64`, and macOS defaults to `clang-x64`. Select a specific IntelliSense mode to override the `${default}` mode.

gcc-x64

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

```json
{
  "configurations": [
    {
      "name": "GCC",
      "includePath": ["${workspaceFolder}/**"],
      "defines": ["_DEBUG", "UNICODE", "_UNICODE"],
      "windowsSdkVersion": "10.0.22000.0",
      "compilerPath": "C:/msys64/mingw64/bin/g++.exe",
      "cStandard": "c17",
      "cppStandard": "c++17",
      "intelliSenseMode": "windows-gcc-x64"
    }
  ],
  "version": 4
}
```

Explicar

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 extension uses the `compilerPath` setting to infer the path to the C++ standard library header files. When the extension knows where to find those files, it can provide 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 might need to change `compilerPath` to match the preferred compiler for your project. For example, if you installed MinGW-w64 version 8.1.0 using the i686 architecture, Win32 threading, and sjlj exception handling install options, the path would look like this: `C:\Program Files (x86)\mingw-w64\i686-8.1.0-win32-sjlj-rt_v6-rev0\mingw64\bin\g++.exe`.

Troubleshooting

MSYS2 is installed, but g++ and gdb are still not found

You must follow the steps on the MSYS2 website to use the MSYS CLI to install the full MinGW-w64 toolchain(`pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`), as well as all required prerequisites. The toolchain includes g++ and gdb.

As a Windows user, running the pacman command gives me an error

UCRT on Windows machines is only included in Windows 10 or later. If you are using another version of Windows, run the following command that does not use UCRT:

```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```

When adding the the MinGW-w64 destination folder to your list of environment variables, the default path will then be: `C:\msys64\mingw64\bin`.

MinGW 32-bit

If you need a 32-bit version of the MinGW toolset, consult the Downloading section on the MSYS2 wiki. It includes links to both 32-bit and 64-bit installation options.

Next steps

- Explore the VS Code User Guide.
- Review the Overview of the C++ extension.
- Create a new workspace, copy your `.vscode` JSON files to it, adjust the necessary settings for the new workspace path, program name, etc. and start coding!