TOPICS GCC on Linux

Using C++ on Linux in VS Code

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

In this tutorial, you will configure Visual Studio Code to use the GCC C++ compiler (g++) and GDB debugger on Linux. GCC stands for GNU Compiler Collection; GDB is the GNU debugger.

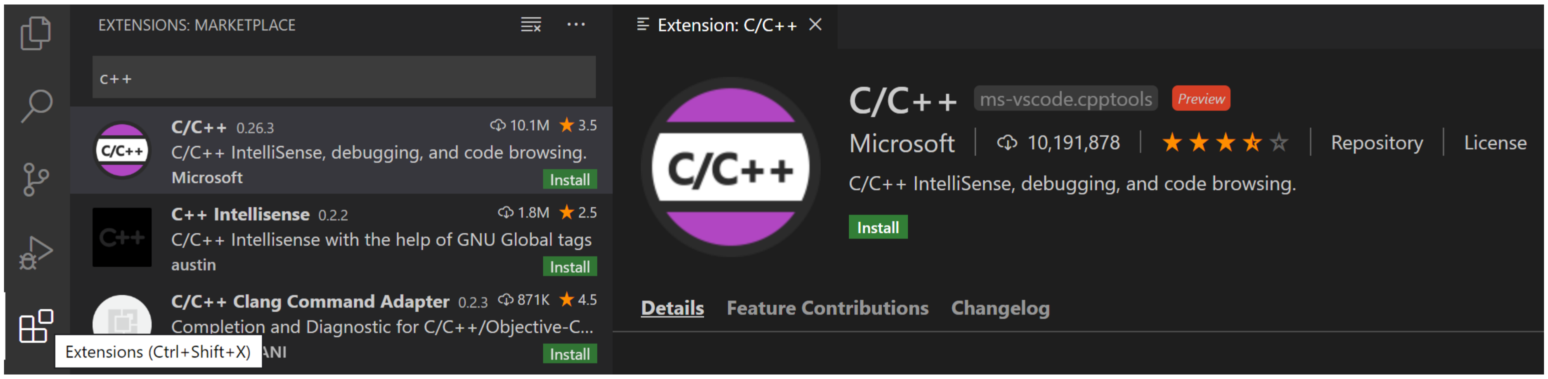
After configuring VS Code, you will compile and debug a simple C++ program in VS Code. This tutorial does not teach you GCC, GDB, Ubuntu or the C++ language. For those subjects, there are many good resources available on the Web.

If you have trouble, 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:

1. Install Visual Studio Code (/download).
2. Install the 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++ 0.26.3 10.1M 3.5
C/C++ IntelliSense, debugging, and code browsing.
Microsoft Install
C++ Intellisense 0.2.2 1.8M 2.5
C/C++ Intellisense with the help of GNU Global tags
austin Install
C/C++ Clang Command Adapter 0.2.3 871K 4.5
Completion and Diagnostic for C/C++/Objective-C...
ANI Install
Extensions (Ctrl+Shift+X)
Extension: C/C++
C/C++ ms-vscode.cpptools Preview
Microsoft 10,191,878 Repository License
C/C++ IntelliSense, debugging, and code browsing.
Install
Details Feature Contributions Changelog

Ensure GCC is installed

Although you'll use VS Code to edit your source code, you'll compile the source code on Linux using the g++ compiler. You'll also use GDB to debug. These tools are not installed by default on Ubuntu, so you have to install them. Fortunately, that's easy.

First, check to see whether GCC is already installed. To verify whether it is, open a Terminal window and enter the following command:

```
gcc -v
```

If GCC isn't installed, run the following command from the terminal window to update the Ubuntu package lists. An out-of-date Linux distribution can sometimes interfere with attempts to install new packages.

```
sudo apt-get update
```

Next install the GNU compiler tools and the GDB debugger with this command:

```
sudo apt-get install build-essential gdb
```

Create Hello World

From the terminal window, create an empty folder called `projects` to store your VS Code projects. Then create a subfolder called `helloworld`, navigate into it, and open VS Code in that folder by entering the following commands:

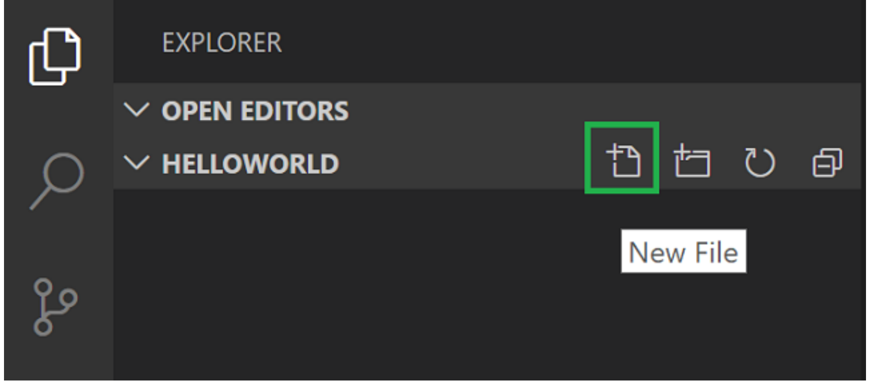
```
mkdir projects
cd projects
mkdir helloworld
cd helloworld
code .
```

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

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

Add hello world source code file

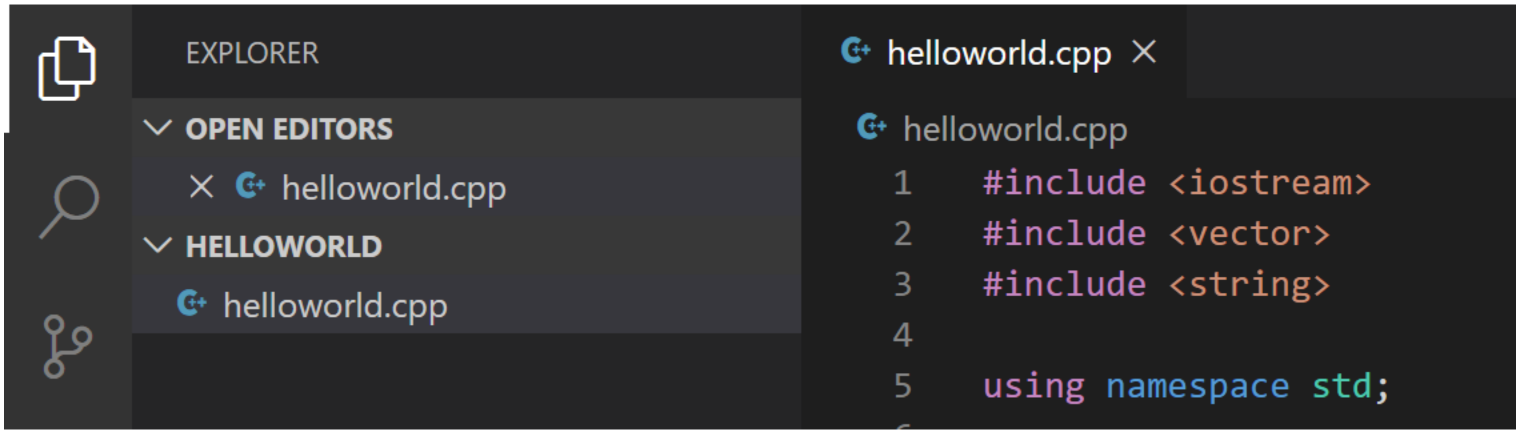
In the File Explorer title bar, select **New File** and name the file `helloworld.cpp`.



Paste in the following 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 that your files are listed in the **File Explorer** view (`Ctrl+Shift+E`) in the side bar of VS Code:



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

using namespace std;
```

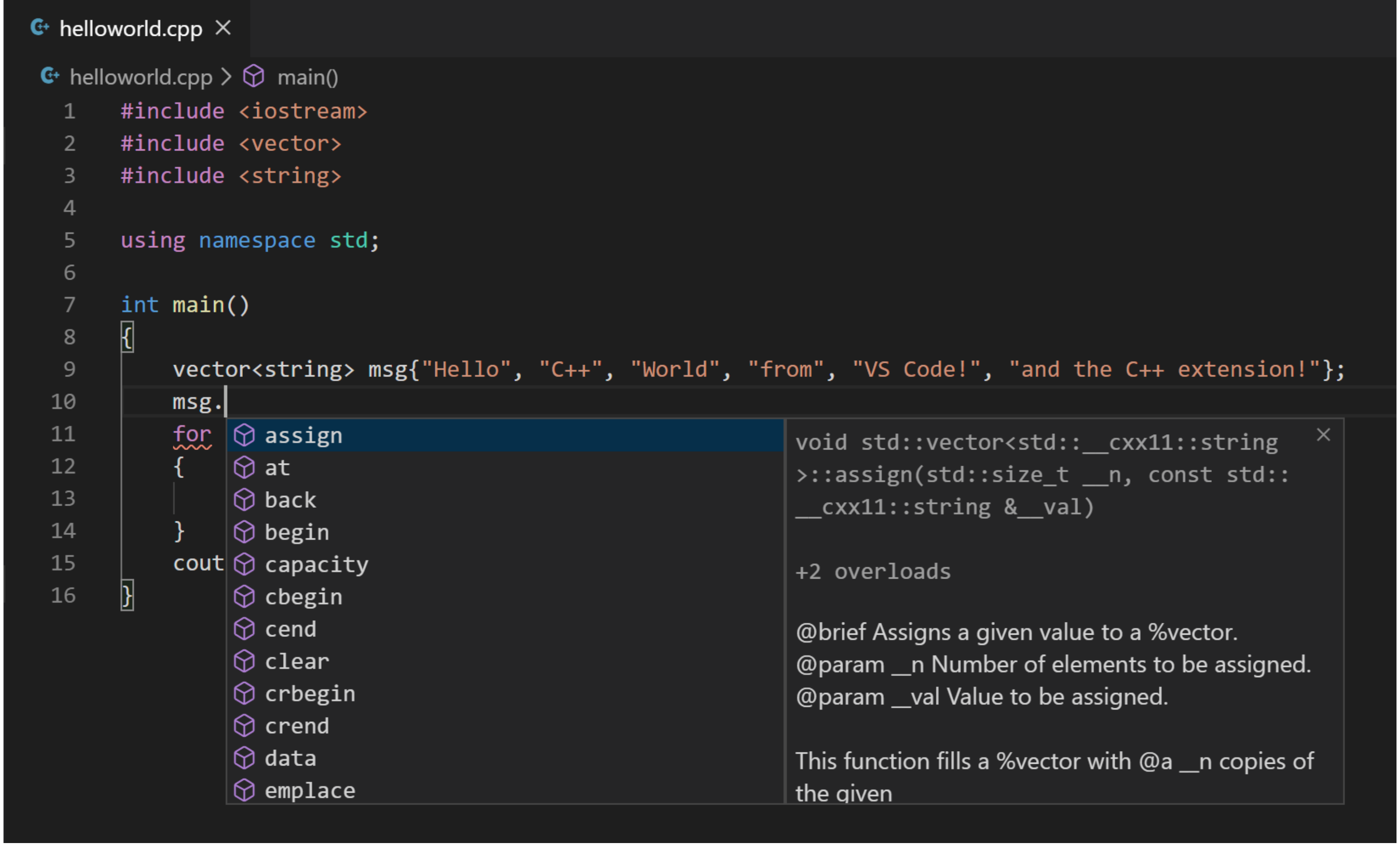
You can also enable Auto Save (/docs/editor/codebasics#_saveauto-save) to automatically save your file changes, by checking **Auto Save** in the main **File** menu.

The Activity Bar on the edge of Visual Studio Code lets you open different views such as **Search**, **Source Control**, and **Run**. You'll look at the **Run** 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 the `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:



```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

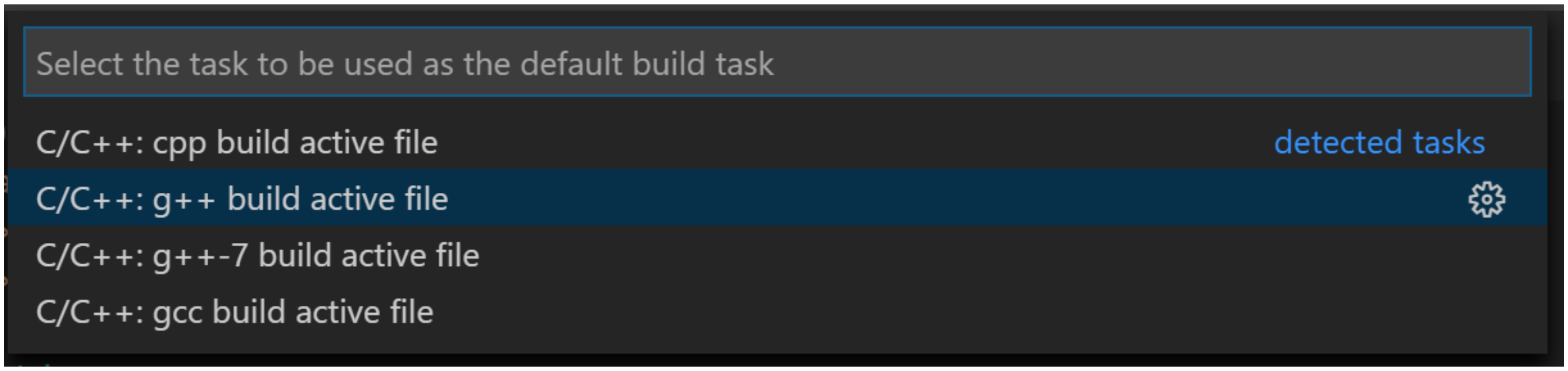
You can press the `Tab` key to insert the selected member. Then, when you add the opening parenthesis, you'll see information about arguments that the function requires.

Build helloworld.cpp

Next, you'll 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 from the source code.

It's important to have `helloworld.cpp` open in the editor because the next step uses the active file in the editor for context to create the build task in the next step.

From the main menu, choose **Terminal** > **Configure Default Build Task**. A dropdown appears showing various predefined build tasks for C++ compilers. Choose **C/C++: g++ build active file**.


Select the task to be used as the default build task
C/C++: cpp build active file
detected tasks
C/C++: g++ build active file
C/C++: g++-7 build active file
C/C++: gcc 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++ build active file",
      "command": "/usr/bin/g++",
      "args": ["-g", "${file}", "-o", "${fileDirname}/${fileBasenameNoExtension}"],
      "options": {
        "cwd": "/usr/bin"
      },
      "problemMatcher": ["$gcc"],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

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

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.

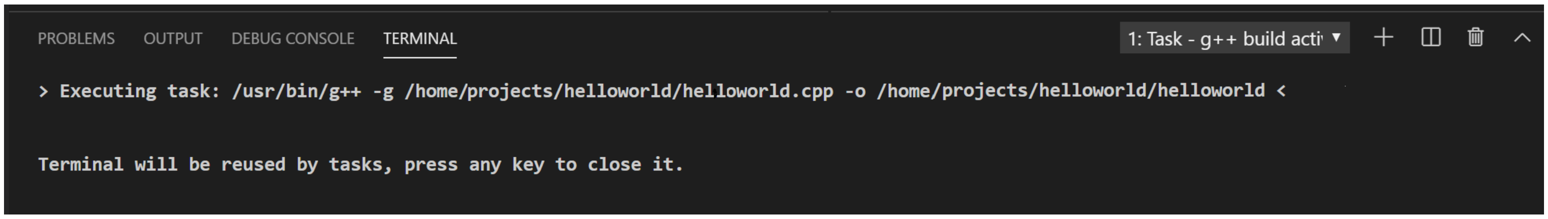
This task tells g++ to take the active file (`${file}`), compile it, and create an executable file in the current directory (`${fileDirname}`) with the same name as the active file but without an extension (`${fileBasenameNoExtension}`), resulting in `helloworld` for our example.

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**.

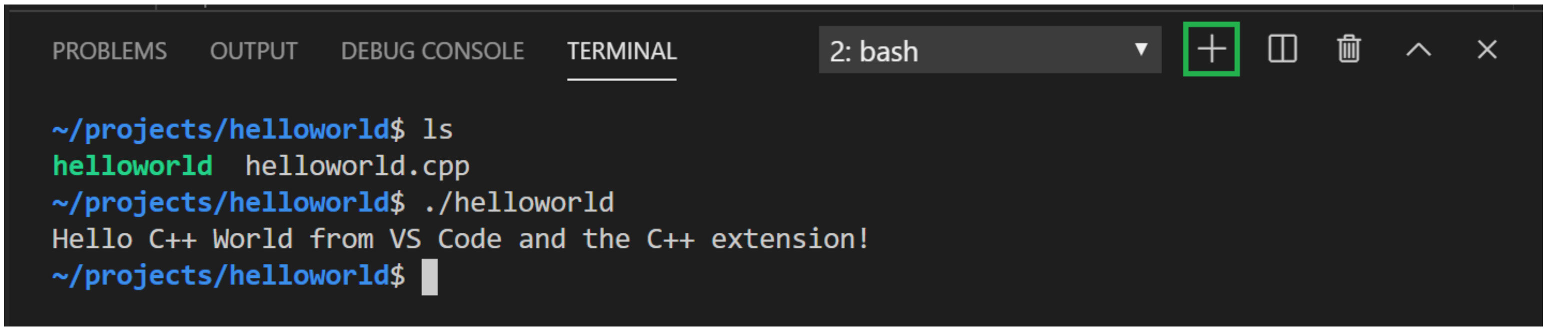
Running the build

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 **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: /usr/bin/g++ -g /home/projects/helloworld/helloworld.cpp -o /home/projects/helloworld/helloworld <


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 bash terminal running in the context of WSL with the `helloworld` folder as the working directory. Run `ls` and you should now see the executable `helloworld` (no file extension).



```
~/projects/helloworld$ ls
helloworld  helloworld.cpp
~/projects/helloworld$ ./helloworld
Hello C++ World from VS Code and the C++ extension!
~/projects/helloworld$
```

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

Modifying tasks.json

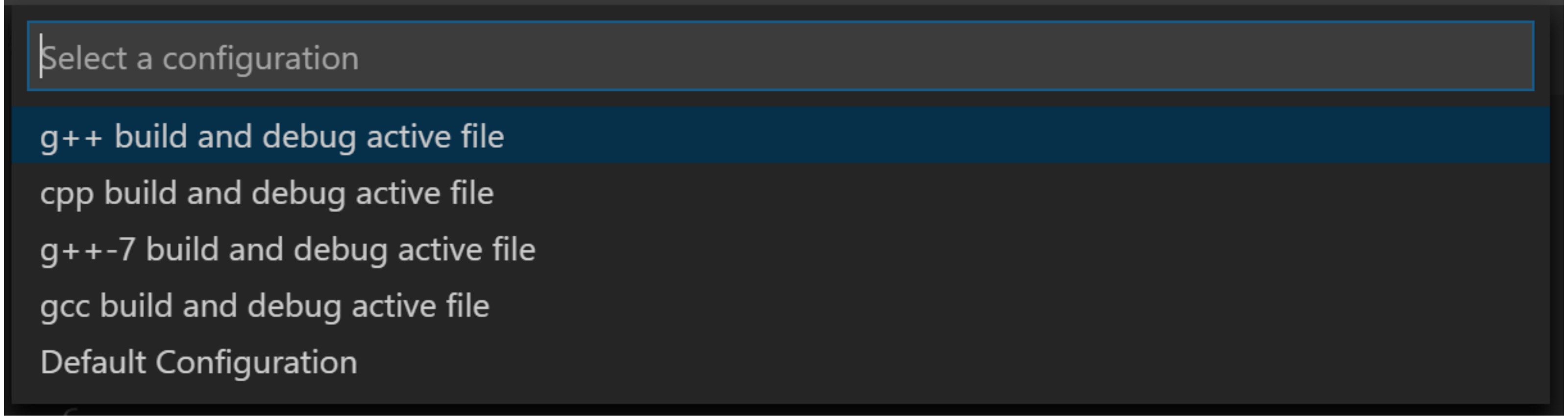
You can modify your `tasks.json` to build multiple C++ files by using an argument like `"${workspaceFolder}/*.cpp"` instead of `${file}` . You can also modify the output filename by replacing `"${fileDirname}/${fileBasenameNoExtension}"` with a hard-coded filename (for example 'helloworld.out').

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 **Run** > **Add Configuration...** and then choose **C++ (GDB/LLDB)**.

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


Select a configuration
g++ build and debug active file
cpp build and debug active file
g++-7 build and debug active file
gcc 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++ build and debug active file",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}/${fileBasenameNoExtension}",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ],
      "preLaunchTask": "g++ build active file",
      "miDebuggerPath": "/usr/bin/gdb"
    }
  ]
}
```

In the JSON above, `program` specifies the program you want to debug. Here it is set to the active file folder `${fileDirname}` and active filename without an extension `${fileBasenameNoExtension}`, which if `helloworld.cpp` is the active file will be `helloworld`.

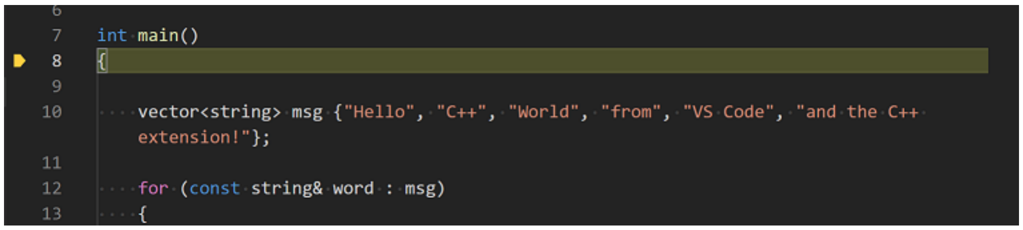
By default, the C++ extension won't add any breakpoints to your source code and the `stopAtEntry` value is set to `false`.

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

Start a debugging session

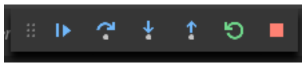
1. Go back to `helloworld.cpp` so that it is the active file.
2. Press `F5` or from the main menu choose **Run** > **Start Debugging**. Before you start stepping through the code, let's take a moment to notice several changes in the user interface:

- The Integrated Terminal appears at the bottom of the source code editor. In the **Debug 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:



```
int main()
{

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

    for (const string& word : msg)
    {
```

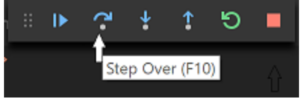
- The Run view on the left shows debugging information. You'll see an example later in the tutorial.
- At the top of the code editor, a debugging control panel appears. You can move this around the screen by grabbing the dots on the left side.



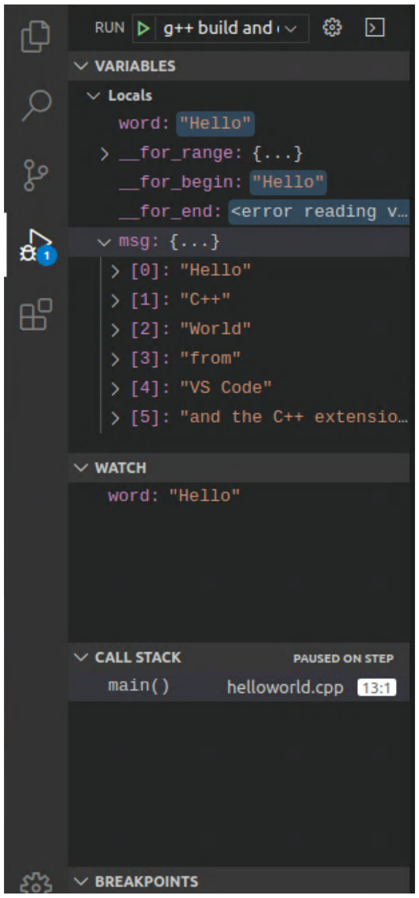
Step through the code

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 side.



```
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…
```

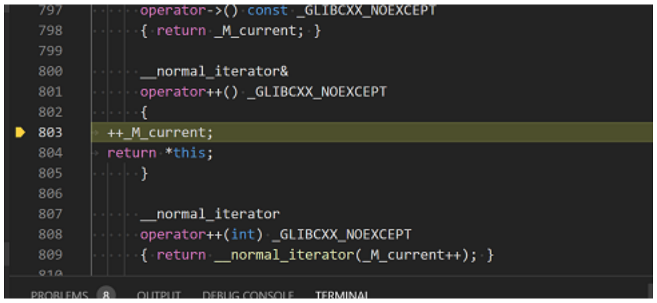
```
word: "Hello"
```

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

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.

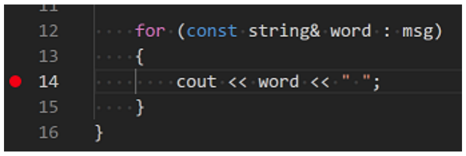
3. Press **Step over** again to execute the `cout` statement. (Note that as of the March 2019 release, the C++ extension does not print any output to the **Debug Console** until the last cout executes.)

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++); }
```

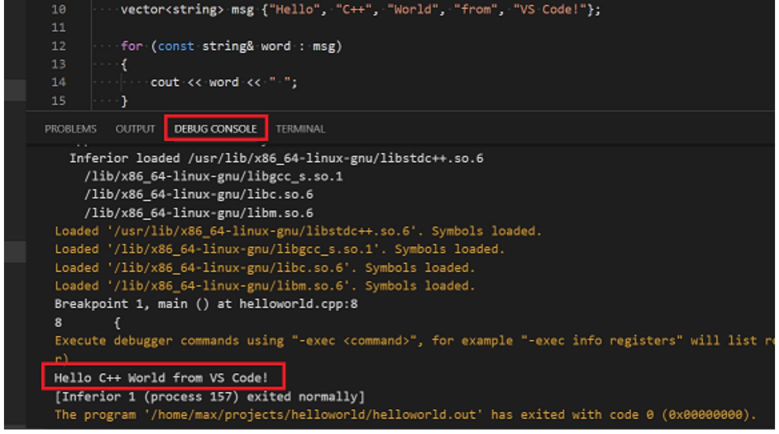
To return to your own code, one way is to keep pressing **Step over**. Another way is to set a breakpoint in your code by switching to the `helloworld.cpp` tab in the code editor, putting the insertion point somewhere on the `cout` statement inside the loop, and pressing `F9`. A red dot appears in the 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 **Debug Console** tab of the integrated terminal, along with some other diagnostic information that is output by GDB.



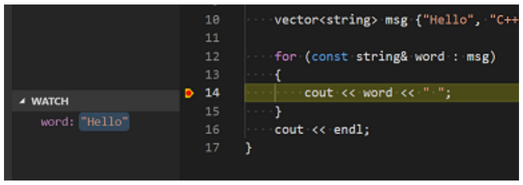
```
    vector<string> msg {"Hello", "C++", "World", "from", "VS Code!"};

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

```
   Inferior loaded /usr/lib/x86_64-linux-gnu/libstdc++.so.6
     /lib/x86_64-linux-gnu/libgcc_s.so.1
     /lib/x86_64-linux-gnu/libc.so.6
     /lib/x86_64-linux-gnu/libm.so.6
Loaded '/usr/lib/x86_64-linux-gnu/libstdc++.so.6'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libgcc_s.so.1'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libc.so.6'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libm.so.6'. Symbols loaded.
Breakpoint 1, main () at helloworld.cpp:8
8       {
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list r
r)
Hello C++ World from VS Code!
[Inferior 1 (process 157) exited normally]
The program '/home/max/projects/helloworld/helloworld.out' has exited with code 0 (0x00000000).
```

Set a watch

To keep track of the value of a variable as your program executes, set a **watch** on the variable.

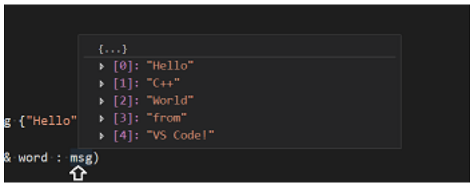
1. Place the insertion point inside the loop. In the **Watch** window, click the plus sign and in the text box, type `word`, which is the name of the loop variable. Now view the Watch window as you step through the loop.



```
word: "Hello"
```

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

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

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

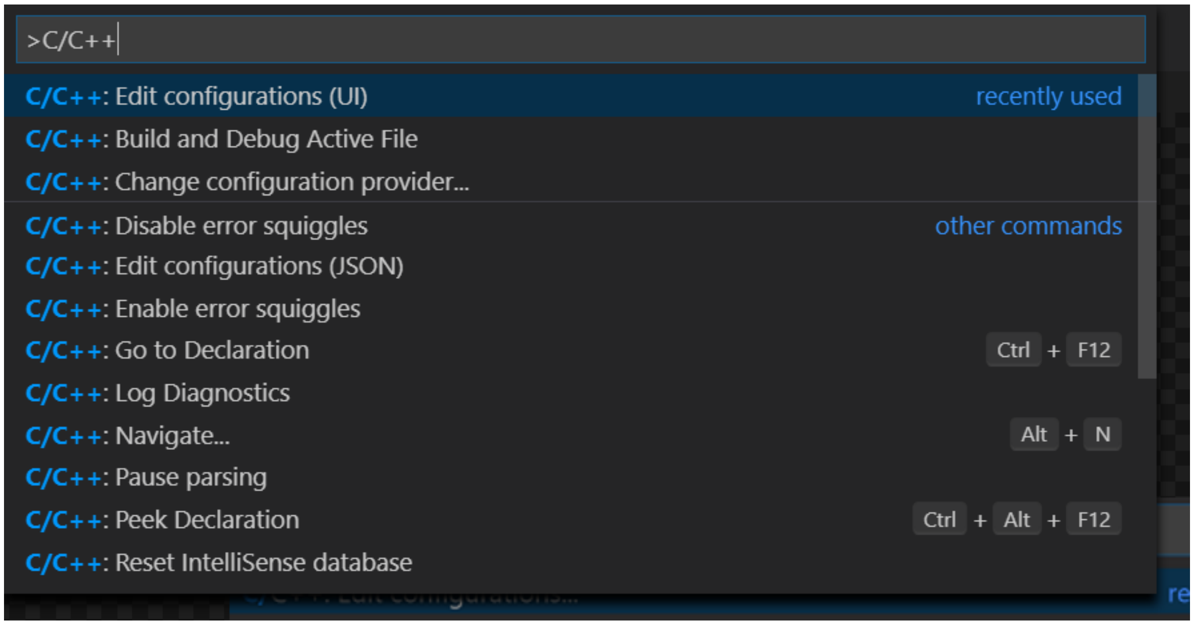


```
{...}
[0]: "Hello"
[1]: "C++"
[2]: "World"
[3]: "from"
[4]: "VS Code!"
g {"Hello"
& word : msg)
```

C/C++ configurations

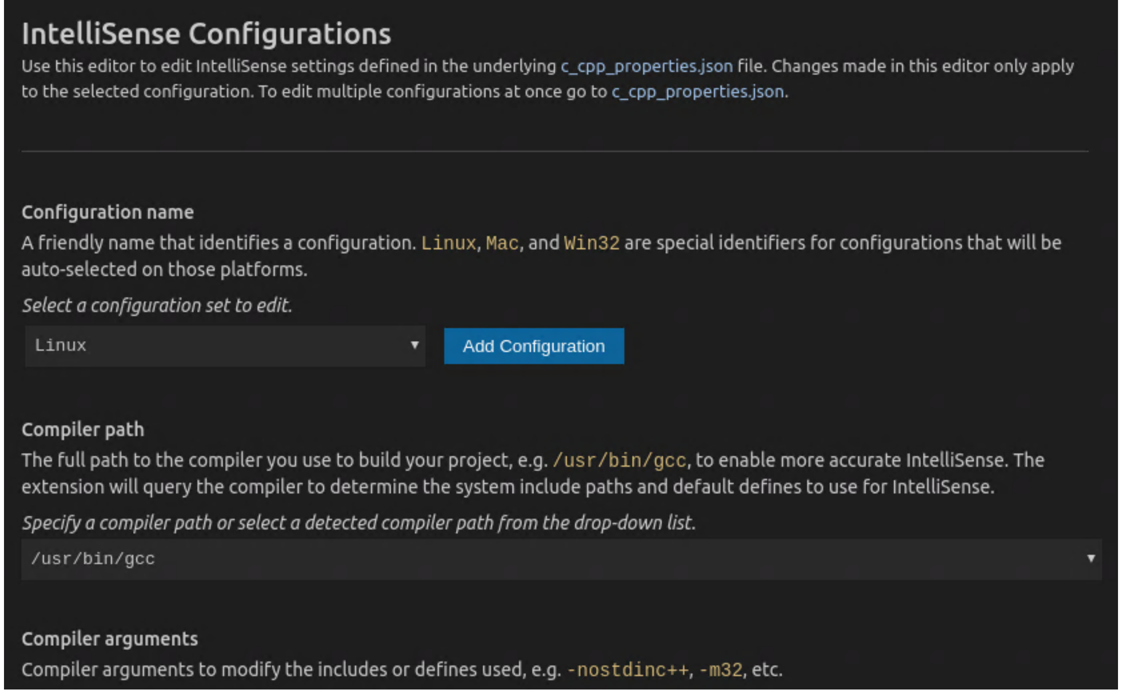
If you want more control over the C/C++ extension, you can create a `c_cpp_properties.json` file, which will allow you to change settings such as the path to the compiler, include paths, C++ standard (default is C++17), and more.

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.

Linux

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.

/usr/bin/gcc

Compiler arguments

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

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

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": "Linux",
      "includePath": ["${workspaceFolder}/**"],
      "defines": [],
      "compilerPath": "/usr/bin/gcc",
      "cStandard": "c11",
      "cppStandard": "c++17",
      "intelliSenseMode": "clang-x64"
    }
  ],
  "version": 4
}
```

Reusing your C++ configuration

VS Code is now configured to use gcc on Linux. The configuration applies to the current workspace. To reuse the configuration, just copy the JSON files to a `.vscode` folder in a new project folder (workspace) and change the names of the source file(s) and executable as needed.

Troubleshooting

Compiler and linking errors

The most common cause of errors (such as `undefined _main`, or `attempting to link with file built for unknown-unsupported file format`, and so on) occurs when `helloworld.cpp` is not the active file when you start a build or start debugging. This is because the compiler is trying to compile something that isn't source code, like your `launch.json`, `tasks.json`, or `c_cpp_properties.json` file.

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 .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**

3/19/2020

IN THIS ARTICLE

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

Hello from Seattle. Follow @code (https://go.microsoft.com/fwlink/?LinkID=533687) Star 94,133