

# vscode




## 调试





[1] 配置 +

[2] 操作 +

## 利用**调试**功能可以观察程序运行的流程

-  可以观察代码的逐步执行。
-  可以观察变量在程序运行中值的变化。
-  是查找程序中错误的利器。

## 调试功能不支持中文路径

-  程序文件名不能够包含中文。
-  程序文件路径上的文件夹名称不能够包含中文。



[1] 配置

+

## C++ 调试启动配置



可以进行**全局**配置或者**工作区**配置。

1. 工作区配置放在工作区**直属**子目录.vscodex中的launch.json文件中。
2. 全局配置放在设置文件的launch项中。



采用全局配置更加方便。




将以下内容放入设置文件中。


```
"launch": {
  "configurations": [
    {
      "name": "C++ debug (global)",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
      "cwd": "${fileDirname}",
      "preLaunchTask": "C++ compile"
    }
  ],
}
```

 name: 调试任务的名称, 将作为标识符在调试窗口中显示。

```
"name": "C++ debug (global)",
```

 type: 调试器类型, GDB采用cppdbg。

```
"type": "cppdbg",
```

 request: 请求类型, 启动新的程序进行调试。

```
"request": "launch",
```

 program: 调试程序的位置, 就是**编译任务**输出的程序位置。

```
"program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
```

 `${}`中的内容为**占位符**, 会根据**文件信息**进行**替换**。

 `${fileDirname}`: 文件所属目录。

 `${fileBasenameNoExtension}`: 不带扩展名的文件名。



 cwd: 调试运行时的工作目录。

```
"cwd": "${fileDirname}",
```

 采用**文件**输入输出时，该目录就是文件的**根目录**。

 preLaunchTask: 调试的**前置**任务，即编译任务。

```
"preLaunchTask": "C++ compile"
```

 任务名称就是编译的任务名称。



## C++ 编译任务配置



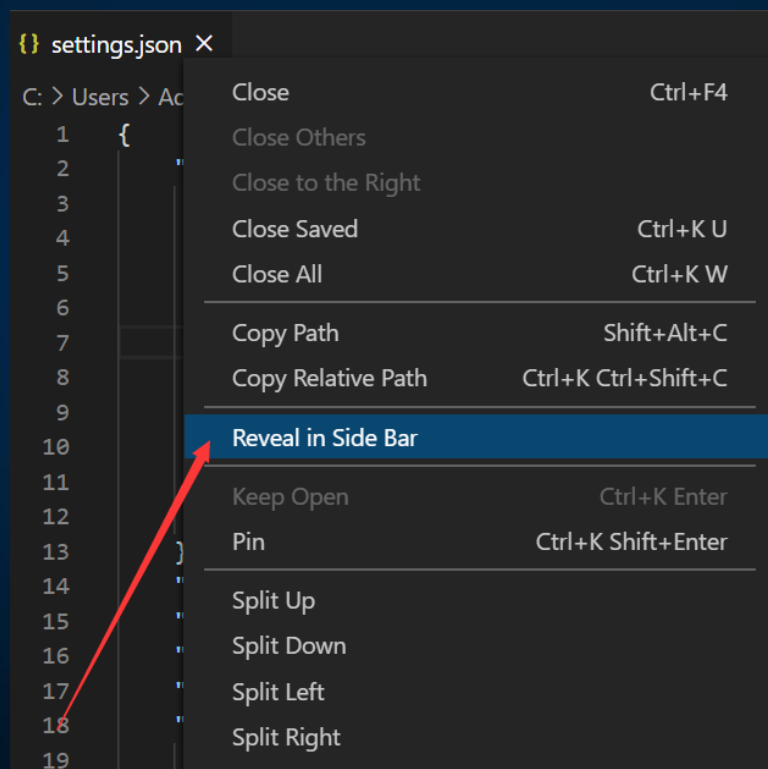
可以进行**全局**配置或者**工作区**配置。

1. 工作区配置放在工作区**直属**子目录.vscode中的tasks.json文件中。
2. 全局配置在设置文件**同目录**下的tasks.json文件中。

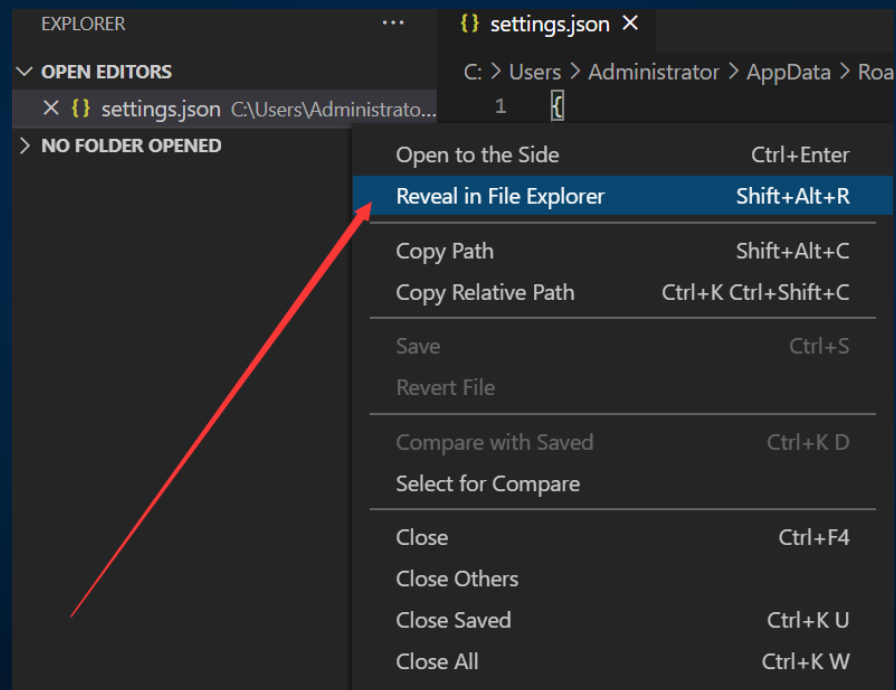


采用全局配置更加方便。



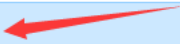
1. 在设置文件的页标签上点击右键，然后点击Reveal in Side Bar，在侧边栏中显示设置文件。



↖ 2. 右键点击侧边栏中的设置文件，然后点击Reveal in File Explorer，打开设置文件所在目录。



↖ 3. 在目录中新建一个文件，并命名为tasks.json。

 settings.json	2020/11/13 16:10	JSON 文件	1 KB
 tasks.json 	2020/11/13 16:21	JSON 文件	0 KB

↖ 4. 将以下内容放入任务文件中。

```
{
  "tasks": [
    {
      "label": "C++ compile",
      "type": "shell",
      "command": "g++",
      "args": [
        "-g",
        "-std=c++14",
        "${file}",
        "-o",
        "${fileDirname}\\${fileBasenameNoExtension}.exe"
      ]
    }
  ]
}
```

 label: 编译任务的名称, 调试启动配置的前置任务将使用此名称。

```
"label": "C++ compile",
```

 type: 任务类型, 通过终端来执行。

```
"type": "shell",
```

 command: 任务命令, 使用g++进行编译。

```
"command": "g++",
```

 args: 编译命令g++的参数。

```
"args": [  
  "-g",  
  "-std=c++14",  
  "${file}",  
  "-o",  
  "${fileDirname}\\${fileBasenameNoExtension}.exe"  
]
```

 与command搭配，就是终端中的编译命令。

 -g用于产生更多调试可用的信息。

 -o指定输出程序的**路径**，调试启动配置中的程序位置将使用此路径。





[2]

操作

+

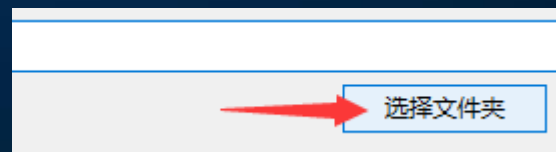
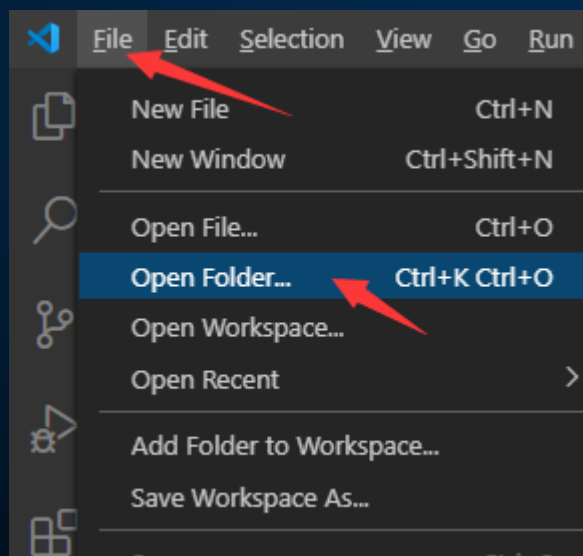


## 调试需要在**工作区**中进行

-  可以将包含代码文件的**文件夹**作为工作区。
-  代码可以存放在工作区中的**子文件夹**中。

## 打开工作区

1. 点击左上角File菜单栏。
2. 点击Open Folder...选项。
3. 在文件浏览器中找到文件夹，并点击选择文件夹。





## 断点：对一行代码的标记



每一行代码上都可以设置断点。



调试过程中，如果程序在运行时遇到断点，  
就会在断点处**暂停**，并等待下一步执行的命令。

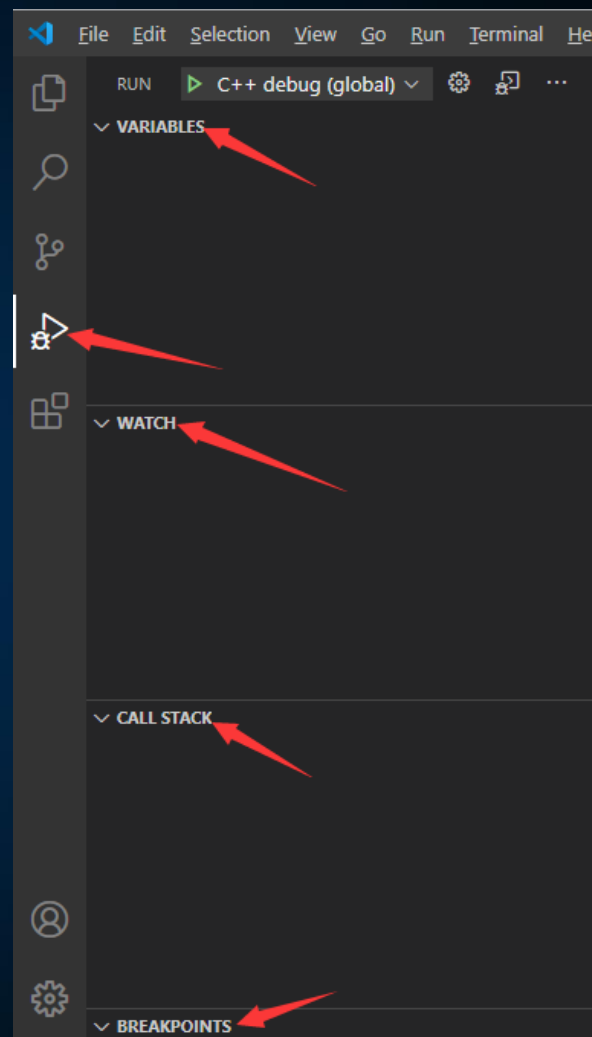


可以在程序调试的过程中**动态**的添加、删除断点。



## 调试在**运行**侧边栏中进行

- ✎ 点击侧边栏中的运行按钮，打开运行侧边栏。
- ✎ VARIABLES栏中会显示系统自动监视的变量。
- ✎ WATCH栏用于设定主动监视的**表达式**。
- ✎ CALL STACK栏中会显示当前暂停处所属的**调用栈**。
- ✎ BREAKPOINTS栏中会显示当前工作区中的**断点**。





## 操作断点

1. 将鼠标**悬停**在一行代码**行号**的**左侧**，出现**半透明**红点，表示断点标记的位置。鼠标移开则红点消失。

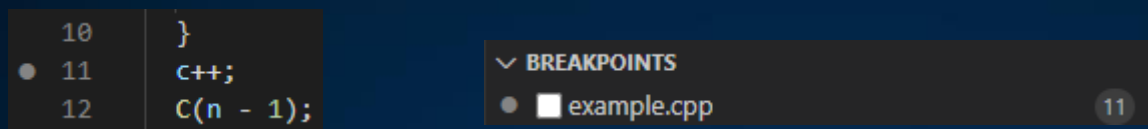
```
9      return;  
10     }  
11     c++;
```

2. 点击半透明红点，红点变成**不透明**，表示断点设置成功，同时断点出现在断点栏中。

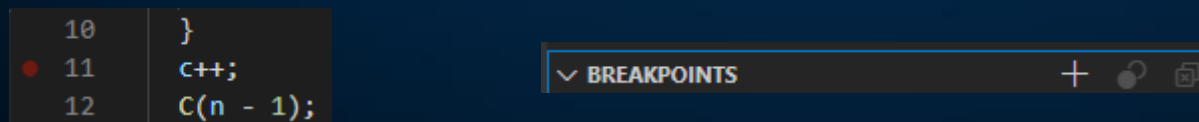
```
10     }  
11     c++;  
12     C(n - 1);
```

```
▼ BREAKPOINTS  
● ☒ example.cpp 11
```

- ↖ 3. 点击断点栏中断点前的**勾选框**，**勾选**时断点标记呈**红色**，表示**启用**；**不勾选**时断点标记呈**灰色**，表示**禁用**。

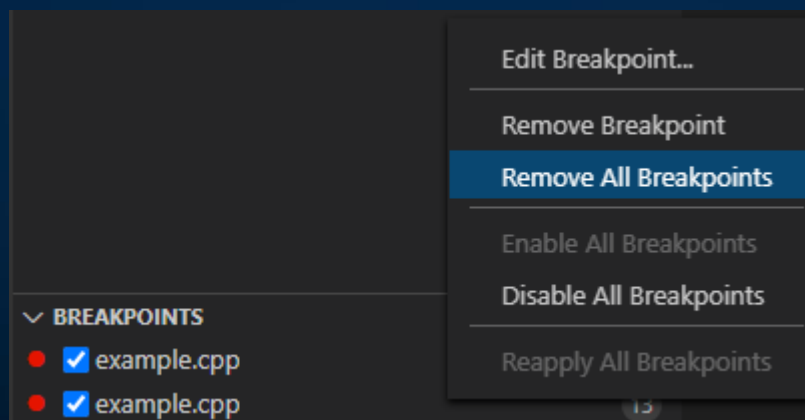


- ↖ 4. 点击断点标记，标记变回**半透明红点**，鼠标移开则消失，表示断点删除成功，同时从断点栏中消失。



## 定时清理不用的断点，防止调试启动出现问题

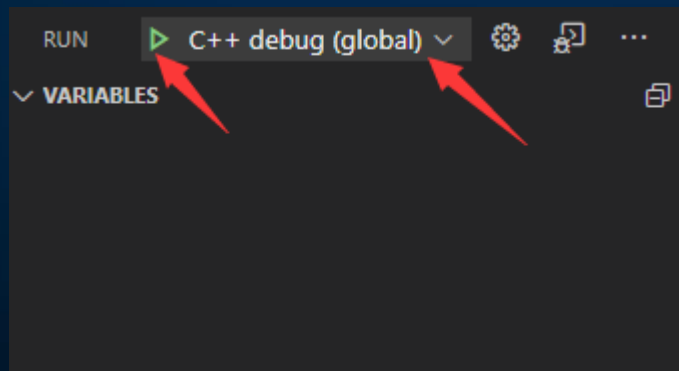
 可以在断点栏点鼠标右键，然后选择Remove All Breakpoints。



Remove All Breakpoints: 删除所有断点

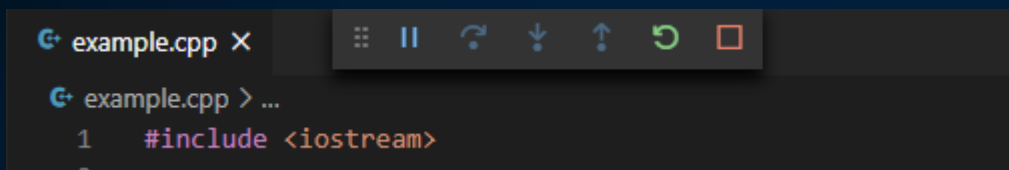
## 启动调试

1. 找到运行侧边栏上方的任务运行框，选择之前配置的调试启动项。
2. 点击调试启动左侧的**绿色箭头**运行按钮，启动调式。

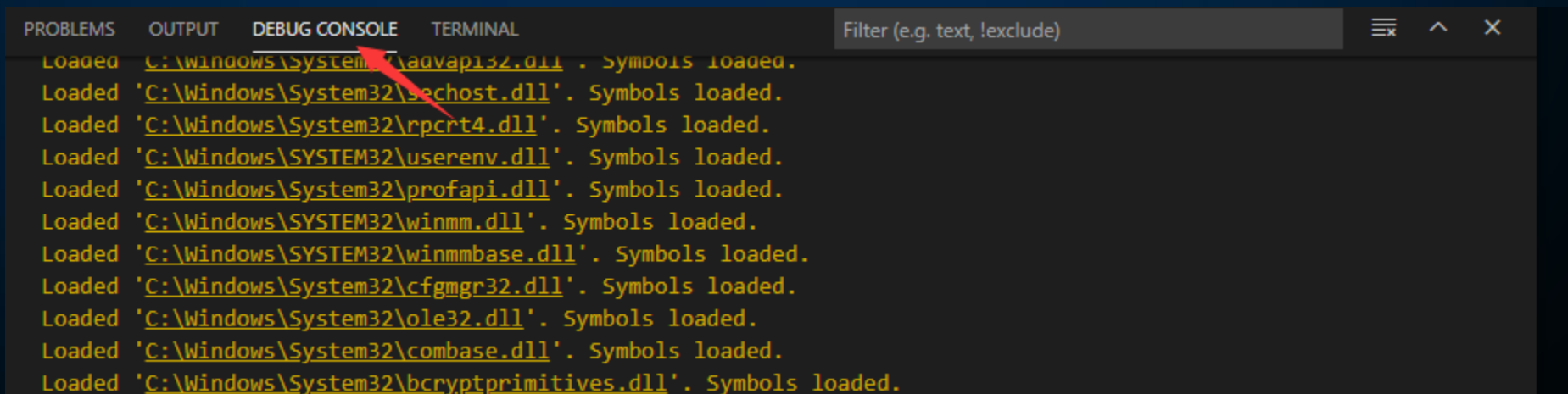




 代码区上方会出现调试控制条。

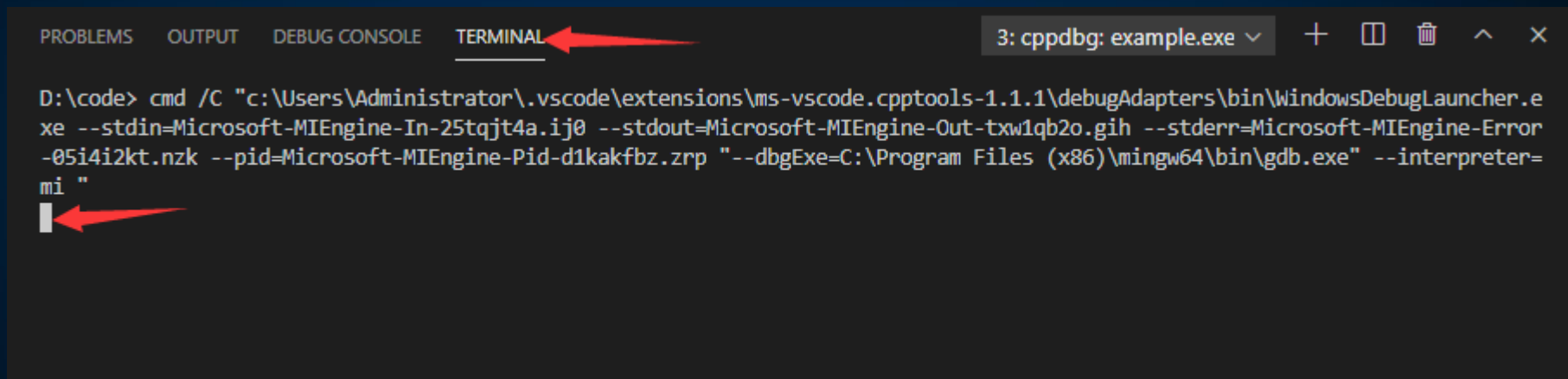


 面板被打开，调试监控标签页出现调试启动信息。





切换到终端标签页，可以看到调试启动的命令。  
在这里可以与程序进行**输入输出交互**。



The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal displays the following command:

```
D:\code> cmd /C "c:\Users\Administrator\.vscode\extensions\ms-vscode.cpptools-1.1.1\debugAdapters\bin\WindowsDebugLauncher.exe --stdin=Microsoft-MIEngine-In-25tqjt4a.ij0 --stdout=Microsoft-MIEngine-Out-txw1qb2o.gih --stderr=Microsoft-MIEngine-Error-05i4i2kt.nzk --pid=Microsoft-MIEngine-Pid-d1kakfbz.zrp "--dbgExe=C:\Program Files (x86)\mingw64\bin\gdb.exe" --interpreter=mi "
```

A red arrow points to the 'TERMINAL' tab, and another red arrow points to the cursor at the end of the command line.

🔗 如果程序中既没有断点，也没有输入，则程序会**直接**执行完毕，调试结束。

 如果程序中设有断点，那么启动调试后会暂停在**第一个**断点处。

```
15  
16 int main() {  
17     cin >> n;  
18     C(n);  
19     cout << c;  
20     return 0;  
21 }
```

 程序暂停时，**待执行**的行背景用**黄色**高亮显示，同时行号左边出现**黄色空心箭头**。

 如果程序在等待**输入**，则黄色高亮消失，需要在**终端**中进行输入。

```
16  int main() {  
17      cin >> n;  
18      C(n);  
19      cout << c;  
20      return 0;  
21  }
```

```
16  int main() {  
17      cin >> n;  
18      C(n);  
19      cout << c;  
20      return 0;  
21  }
```

 输入完毕，程序**继续**运行时，黄色高亮重新出现。

 注意：程序未在等待输入时，终端的输入都**不会**进行显示。  
程序在等待输入时，之前在终端的输入会**一次性**全部显示出来。



## 调试控制



 **Continue**: 继续执行程序。

 如果遇到断点，则会在下一个断点处暂停。

```
7 void C(int n) {
8     if (!n) {
9         return;
10    }
11    c++;
12    C(n - 1);
13    C(n - 1);
14 }
15
16 int main() {
17     cin >> n;
18    C(n);
19    cout << c;
20    return 0;
21 }
```

```
7 void C(int n) {
8     if (!n) {
9         return;
10    }
11    c++;
12    C(n - 1);
13    C(n - 1);
14 }
15
16 int main() {
17     cin >> n;
18    C(n);
19    cout << c;
20    return 0;
21 }
```

continue: 继续



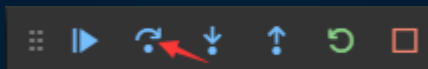
 **Step Into**: 执行一行程序，然后暂停。

 如果遇到函数调用，则进入函数。

```
7 void C(int n) {  
8     if (!n) {  
9         return;  
10    }  
11    c++;  
12    C(n - 1);  
13    C(n - 1);  
14 }
```

```
7 void C(int n) {  
8     if (!n) {  
9         return;  
10    }  
11    c++;  
12    C(n - 1);  
13    C(n - 1);  
14 }
```

```
7 void C(int n) {  
12    C(n - 1);  
13    C(n - 1);  
14 }
```



- ✎ **Step Over**: 执行**一行**程序，然后暂停。
- ✎ 如果遇到函数调用，则直接**执行完**该函数。  
如果遇到断点，则会在下一个断点处暂停。

```
7 void C(int n) {  
8     if (!n) {  
9         return;  
10    }  
11    c++;  
12    C(n - 1);  
13    C(n - 1);  
14 }
```

```
7 void C(int n) {  
8     if (!n) {  
9         return;  
10    }  
11    c++;  
12    C(n - 1);  
13    C(n - 1);  
14 }
```

```
7 void C(int n) {  
8     if (!n) {  
9         return;  
10    }  
11    c++;  
12    C(n - 1);  
13    C(n - 1);  
14 }
```




 **Step Out**: 执行完当前行所在的函数，在函数调用的**出口**处暂停。

 如果遇到断点，则会在下一个断点处暂停。

```
7 void C(int n) {
8     if (!n) {
9         return;
10    }
11    c++;
12    C(n - 1);
13    C(n - 1);
14 }
15
16 int main() {
17     cin >> n;
18     C(n);
19     cout << c;
20     return 0;
21 }
```

```
7 void C(int n) {
8     if (!n) {
9         return;
10    }
11    c++;
12    C(n - 1);
13    C(n - 1);
14 }
15
16 int main() {
17     cin >> n;
18     C(n);
19     cout << c;
20     return 0;
21 }
```

 注意：**递归**调用时同理，只会执行完调用栈中**栈顶**的函数，**不会**把调用栈中的**所有**函数都执行完。





 **Stop**: 结束调试。



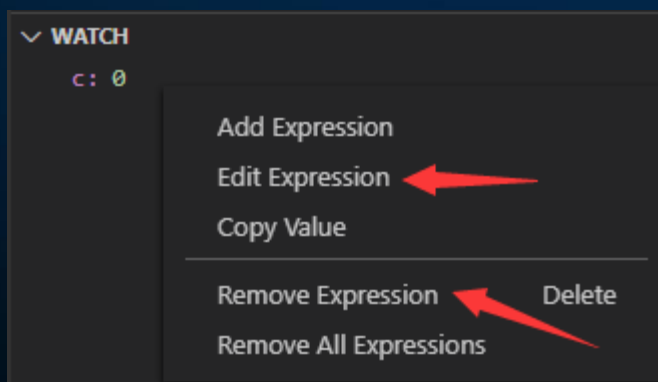
 **Restart**: 重新开始调试。  
等价于停止后再启动调试。

## 监视表达式

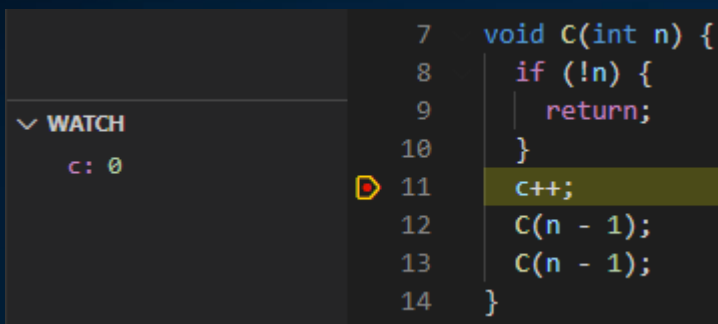
 点击WATCH栏中的加号，可以添加要监视的表达式。



 在被监视的表达式上点击鼠标右键，可以修改、删除表达式。

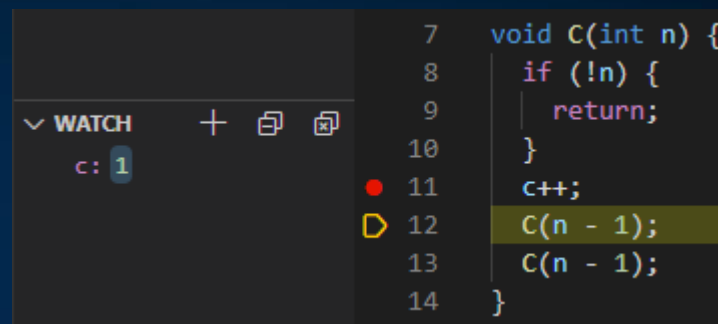


✎ 程序暂停时，会计算被监视的表达式在**此时**的值。



```
7 void C(int n) {
8     if (!n) {
9         return;
10    }
11    c++;
12    C(n - 1);
13    C(n - 1);
14 }
```

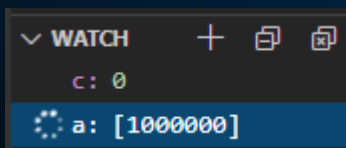
WATCH  
c: 0



```
7 void C(int n) {
8     if (!n) {
9         return;
10    }
11    c++;
12    C(n - 1);
13    C(n - 1);
14 }
```

WATCH  
c: 1

✎ 如果监视的表达式内容过多，则计算时间会很长，甚至导致程序**失去响应**。  
所以如果要监视数组，建议**临时**减少数组的长度。



WATCH  
c: 0  
a: [1000000]

# 谢谢

