University of Zagreb

Faculty of Electrical Engineering and Computing

Department of Applied Computing

**Instructions for compiling programs written in programming language C**

Zagreb, 2023.

**Content**

[1 Introduction 1](#_Toc147787081)

[2 Installing MinGW 1](#_Toc147787082)

[2.1 Opening the Command Prompt 2](#_Toc147787083)

[2.1.1 Setting the system variable Path in the command window 4](#_Toc147787084)

[3 Writing programs with a simple editor, compiling and executing programs 7](#_Toc147787085)

[3.1 Configuration for MacOS 8](#_Toc147787086)

[3.2 Possible problems when compiling or executing a program 8](#_Toc147787087)

[4 Writing and compiling programs in an integrated development environment 11](#_Toc147787088)

[4.1.1 Optional: Universal *Makefile* 18](#_Toc147787089)

[5 VSC configuration files 22](#_Toc147787090)

# Introduction

Electronic computer programs are prepared in a symbolic, human-understandable programming language. In the course Introduction to programming, for this purpose will use the C programming language.

The program, written in symbolic language and stored in a file with the extension .c, needs to be converted to commands understandable to the processor, or executable machine code (in Windows operating system, files with the extension *.exe*, in Unix / Linux operating systems, the format of the extension is not prescribed). The conversion process is carried out in two steps (the concept of preprocessing and assembly code creation will be omitted here because it is indivisible as part of translation):

1. *The translation into object code* is done by a program called the *compiler*. The compiler translates the source file and creates the object code file (usually with the extension *.o*) that cannot yet be run because it does not yet have links to functions and variables from other object code files and libraries where standard functions (input / output, mathematical functions, etc.) have been prepared.
2. Linking to the executable code is done by a program called the linker. This program merges all the required object code files with the pre-programmed libraries and creates executable code that can be run directly from the operating system.

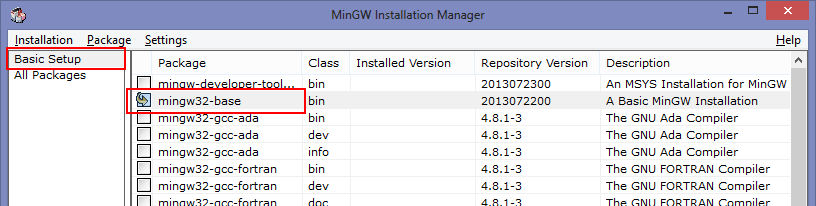
Any text editor already installed on the computer can be used to prepare the source code (write the program). The editor must not create additional hidden formatting tags, such as done by Word. An example of such a program is the basic *Notepad* editor or the slightly improved *Notepad*++. The compiler and linker are often not part of the operating system and need to be installed. For the purpose of Introduction to programming, ww will install MinGW packages in 32-bit version (<https://en.wikipedia.org/wiki/MinGW>).

# Installing MinGW

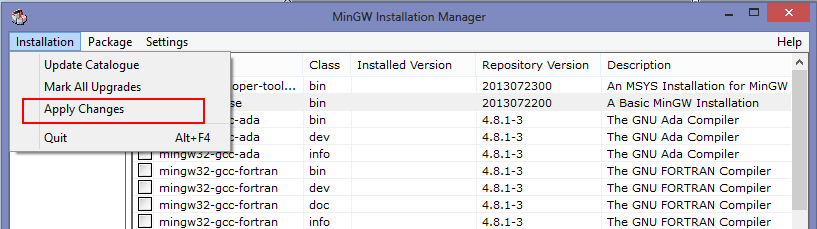
The installer *mingw-get-setup.exe* will be downloaded automatically from the page <http://sourceforge.net/projects/mingw/files/latest/download?source=files>.

The downloaded program must be executed. On the first form shown, press the *Install* button. On the next form, press *Continue*. It will initiate download of MinGW Installation Manager. When downloaded, press Continue to start MinGW Installation Manager.

In left column, select *Basic Setup.* In next column, select(*Mark for Installation*) item *mingw32-base:*



Next, from the menu, select *Installation* -> *Apply Changes*



and in the newly opened window, press the button *Apply.*

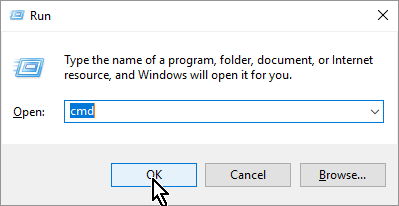
After the installation is completed, the system variable Path must be set by adding a path to the folder bin of the MinGW package. This system variable contains a list of folders where the operating system looks for executable programs that the user attempts to run by specifying their name.

If MinGW is installed with the default settings, the path to the folder bin will be C:\MinGW\bin.

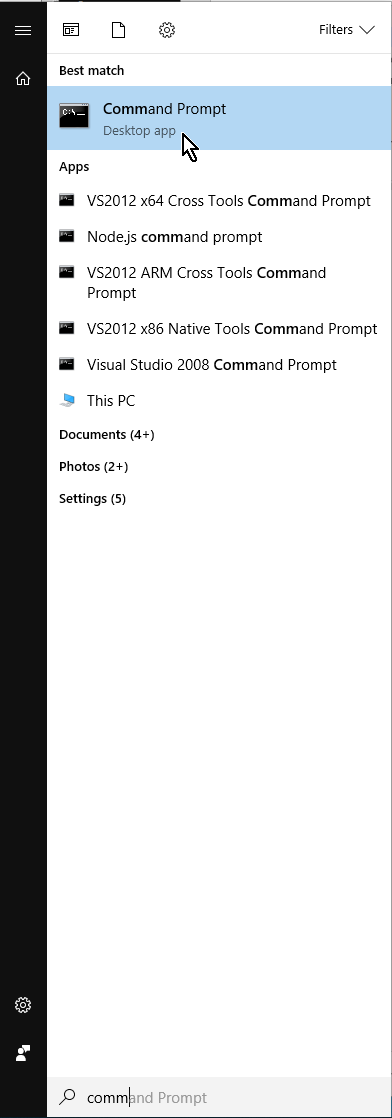
This can be done in several ways, but the simplest way to do it is in the command prompt.

## Opening the Command Prompt

The command prompt is opened by opening the Windows menu (Start menu, ), then typing and selecting Run, typing cmd in the newly opened window, and pressing *OK*



or typing command in the Start menu and selecting Command Prompt:



It is useful to add the command prompt to the Start menu by right-clicking on *Command Prompt* and selecting *Pin to Start* from the pop-up menu.

The command prompt allows you to directly execute operating system commands, which generally looks like this:

program argument1 argument2 … argumentn

program is the name of the program (or internal operating system command) to be executed. If the program is not an internal operating system command (such as a command DIR that prints the contents of the current folder), it must either be in the current folder or in one of the folders listed in the variable *Path* set as described in the next section. A program can be called without arguments like DIR, or with arguments that define the names of the files over which the program will act or options that modify how the program is executed. Options usually begin with characters /, -, or --.

In the following example, the file t1.txt will be copied to t2.txt:

copy t1.txt t2.txt

This requires confirmation if *t2.txt* already exists, but if option /y is added to the call, the confirmation will not be required:

copy /y t1.txt t2.txt

It should be noted that some operating systems distinguish between uppercase and lowercase letters in the names of programs, options, or files, while others do not. Operating system Windows does not distinguish between upper and lower case letters (is *case insensitive*), while operating systems Unix and Linux do (are *case sensitive*).

### Setting the system variable Path in the command window

The system variable *Path* is easily set in a command window by a command

setx Path "%Path%;C:\MinGW\Bin"

The *setx* command written in this way sets the new value of the variable *Path* by adding text *;C:\MinGW\Bin* to it's old value.

Unfortunately, the command *setx* has a built-in restriction on the length of a variable to 1024 characters, so before the call it is necessary to check that the existing content is not too long, which can be checked by the command echo:

echo %Path%

If the following message appears when setting the variable *Path:*

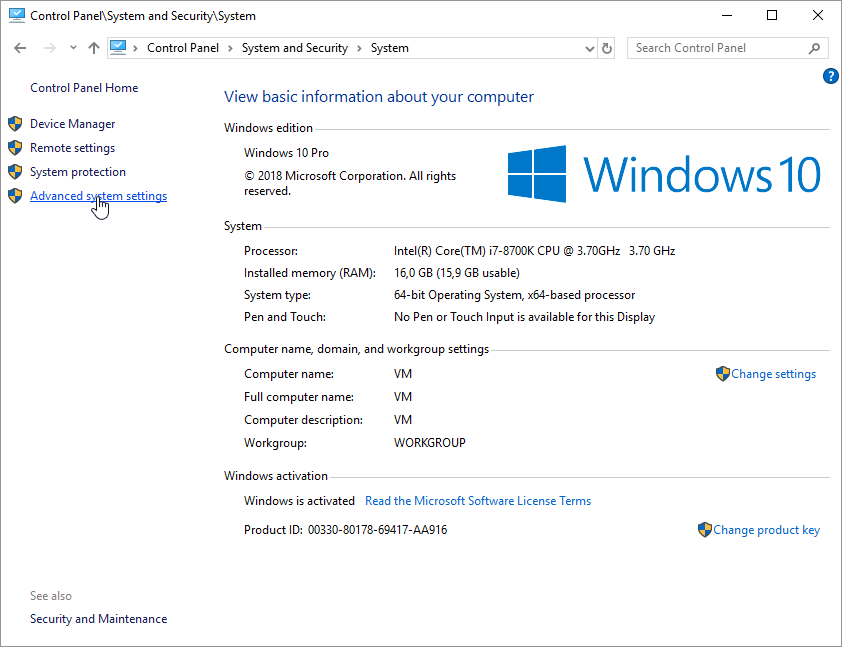
WARNING: The data being saved is truncated to 1024 characters.

it means that the *setx* command limit is exceeded, so the variable *Path* will need to be set through the GUI and checked for content.

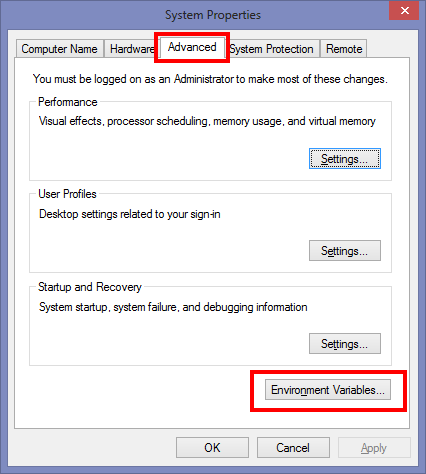
**Setting the system variable Path through a graphical interface**

The system variable *Path* can be set in Control Panel:

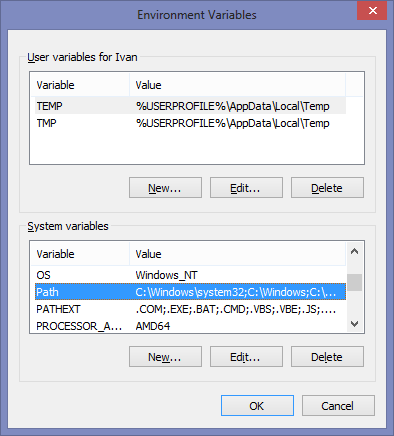
Control Panel -> System and Security -> System  
You need to click the link Advanced system settings



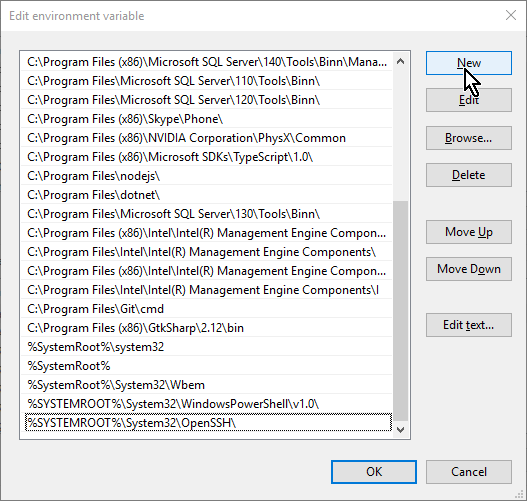
In the newly opened window, select the tab *Advanced* and then press the button *Environment Variables…* as shown in the following figure:



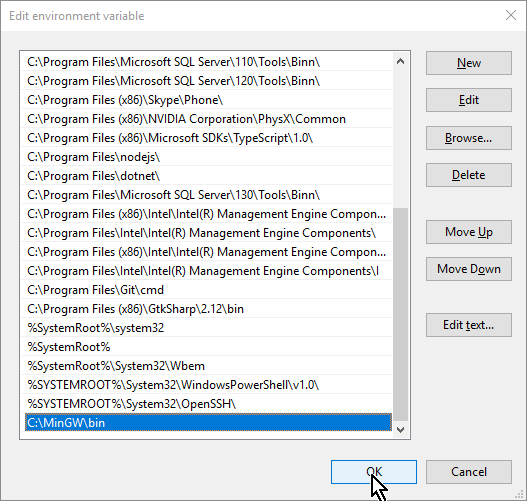
After that, choose *Path* from the list *System variables* and click *Edit:*



A window will open with all the folders where the operating system looks for executable programs, where you need to click New.



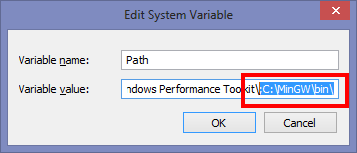
In the newly created blank field at the bottom of the list, enter the path to the folder where the MinGW package is installed to (normally *C:\MinGW\bin*), and press OK:



In older versions of Windows, a window will open. There, in the Variable value field, the text

; C:\MinGW\bin\

should be appended to existing text:



Note the sign ; at the beginning of the text to be added to the variable Path. This character separates the paths contained in the variable Path. The above text should be added to the end of the existing contents of the variable, taking care not to delete the existing contents of the variable.

If MinGW is installed in another folder (instead of C:\MinGW), appropriate folder name should be entered.

**An alternative way of installation**

Students who have problems installing the latest version of MinGW on their Windows computer can try this alternative method:

1. Create a folder on your computer to place MinGW files (e.g. *C:\MinGW*)
2. Download the archive *MinGW.zip* from the file repository and extract it into the folder created above
3. Set the system variable *Path* according to the instructions above

# Writing programs with a simple editor, compiling and executing programs

The whole process of writing and compiling programs will be illustrated in the command prompt, where you first need to create a folder where the program file will be stored, and then make it current e.g.

mkdir C:\IPRO

cd C:\IPRO

and start editing program there with a text editor:

notepad program.c

The following code should be entered in this file:

#include <stdio.h>

int main(void) {

printf("Introduction to Programming");

return 0;

}

and save the changes via the *File/Save* menu or the standard keyboard shortcut *Ctrl-S* (press the *Ctrl* key and then, without releasing it, the *S* key). After that it is necessary to check whether elementary compilation works:

gcc program.c -o program.exe

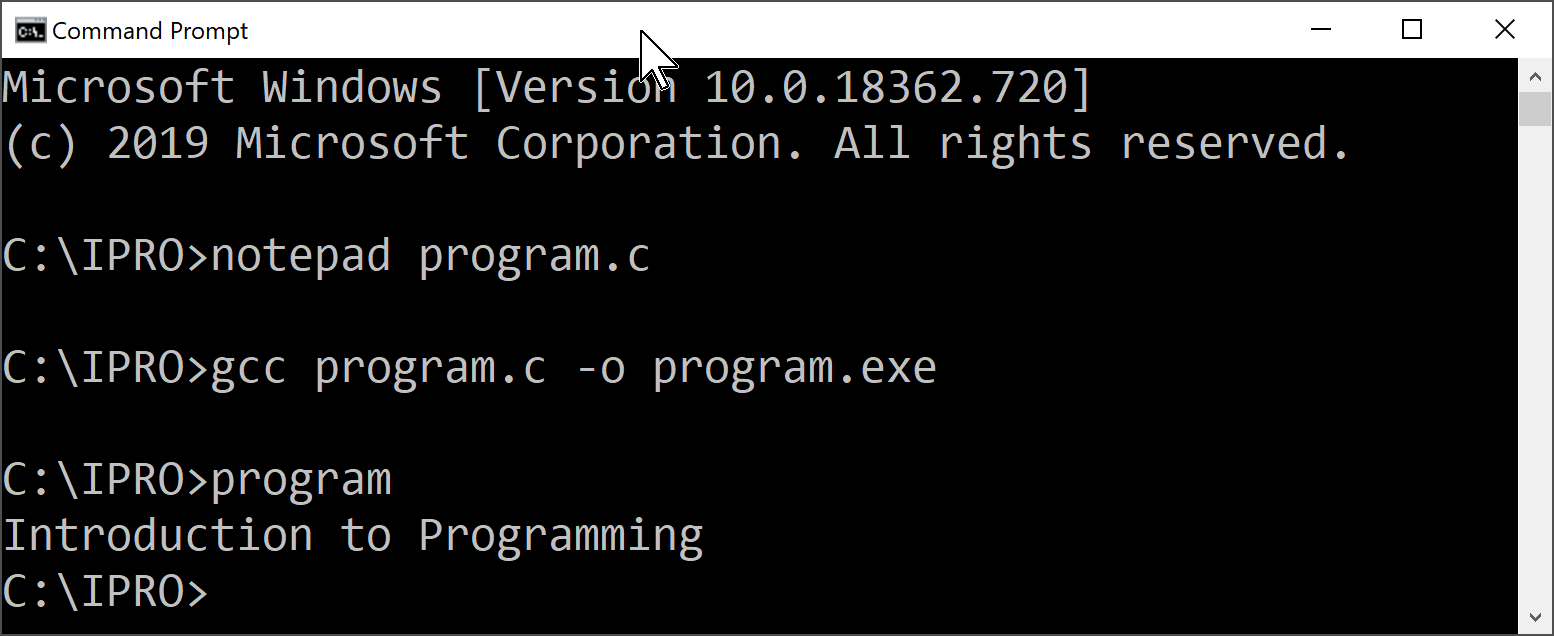
This simplest *gcc* compiler call integrates compiler and linker functions. The program.c file is compiled. Because of the -o (*output*) option, it creates an executable file named *program.exe* which can then be executed with the command

program.exe

or

program

The result should be as in the following figure:



If an error occurred while calling the compiler or starting the program, you should look for the solution in the next section.

Basically, the previous compiler call is internally executed in the same way as the following sequence:

gcc -c program.c

gcc program.o -o program.exe

The first call, because of the argument or the *-c* (*compile*) option, is just a compilation to *program.o*, which in this case remains stored in the current folder (in the case of an elementary call, that object code is automatically deleted). The second call links the object code with the program libraries and creates the executable *program.exe.*

Additional compiler options to use in the Introduction to Programming course are:

-std=c11 – a parameter that ensures compilation compliant to ISO/IEC 9899:2011 standard (there is a newer ISO/IEC 9899:2018 standard, option -c++17, but gcc currently only applies it to C++.

-pedantic-errors – a parameter that ensures generation of errors if the code is not strictly compliant to the standard

-Wall – a parameter that requires printing of detailed warnings

The command that translates the program with the options listed above looks like this:

gcc -std=c11 -pedantic-errors -Wall program.c -o program.exe

## Configuration for MacOS

Install gcc using Homebrew: <https://brew.sh/>

1. Install Homebrew by executing in terminal:

/bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"

1. Install gcc:

brew install gcc

1. Check whether elementary compilation works:

gcc -o <output-file-name> <input-file-name>

npr.

gcc -o output\_program program.c

1. Execute with the command:

./output\_program

## Possible problems when compiling or executing a program

*Problem with variable Path*

If the following error occurs when starting *gcc*:

'gcc' is not recognized as an internal or external command, operable program or batch file.

then probably the variable *Path* is not set or refreshed correctly. In this case, the following steps should be taken. After each step, you can check if the problem is resolved.

1. Check that the variable *Path* is set correctly by typing the command

echo %Path%

If the error is a path, repair it as described in the previous section.

1. Close all command prompts and reopen the command prompt.
2. In Task manageru restart explorer.exe
3. Restart the computer

#### Problem with antivirus software:

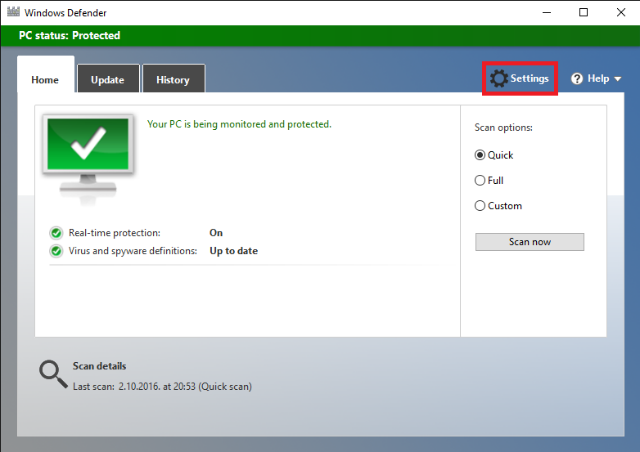
If the following error occurs when starting the compiled program (executable .exe file):

The system cannot execute the specified program

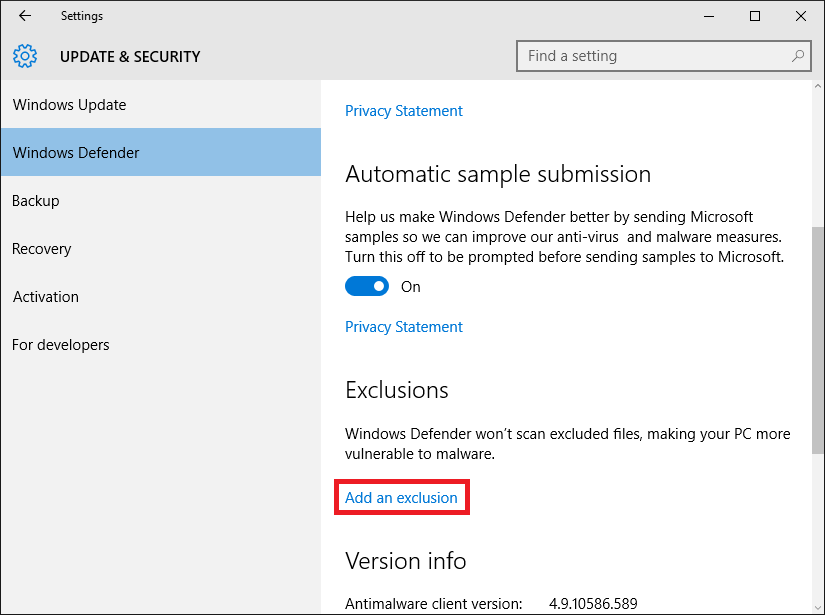
it is possible that the built-in security mechanism (Windows Defender) or other antivirus software automatically blocks the launch of an unknown executable file. It is then necessary to disable the blocking of programs running from the folder where the .exe is created.

For Windows Defender, this can be done as follows:

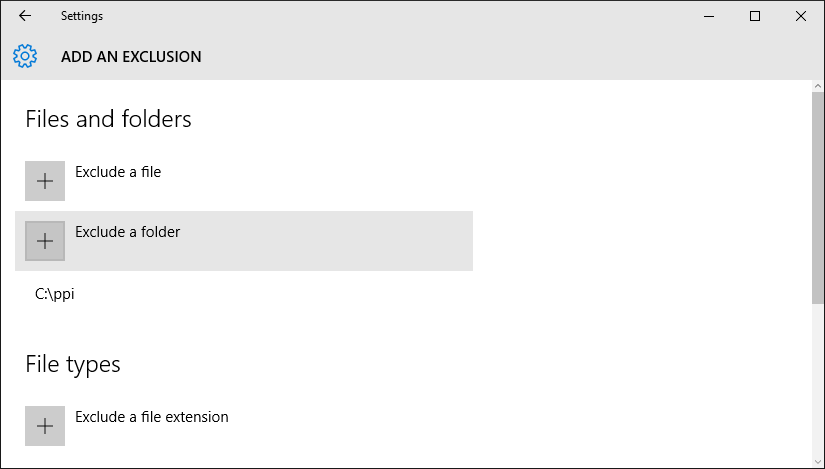
Select *Control Panel -> Windows Defender,* and in the upper right corner, click *Settings*:



In the newly-opened window, under *Exclusions,* choose *Add an exclusion:*



In the following window, select *Exclude a folder* and then select the folder where the program code is located (e.g. *C:\IPRO*):



You should then reattempt the program compilation using the *gcc* command and run the executable program (*.exe*).

# Writing and compiling programs in an integrated development environment

Knowledge of elementary editing of a program by notepad or a similar text editor, compiling programs from the command line, and running programs from the command line, is mandatory and will be tested during laboratory exercises.

Integrated Development Environments (IDEs) are used to make writing, compiling, and testing programs easier. In the Introduction to Programming course, we will use Visual Studio Code development environment, because it is an open source program, runs on multiple platforms (Windows, Linux, Mac), the installation package is relatively small compared to other similar environments, installation is fast and can be used for programming in any programming language, provided that a suitable compiler is installed on your computer, such as the MinGW package whose installation and use is described in the previous sections. The following assumes that MinGW is installed in accordance with the instructions in the previous sections.

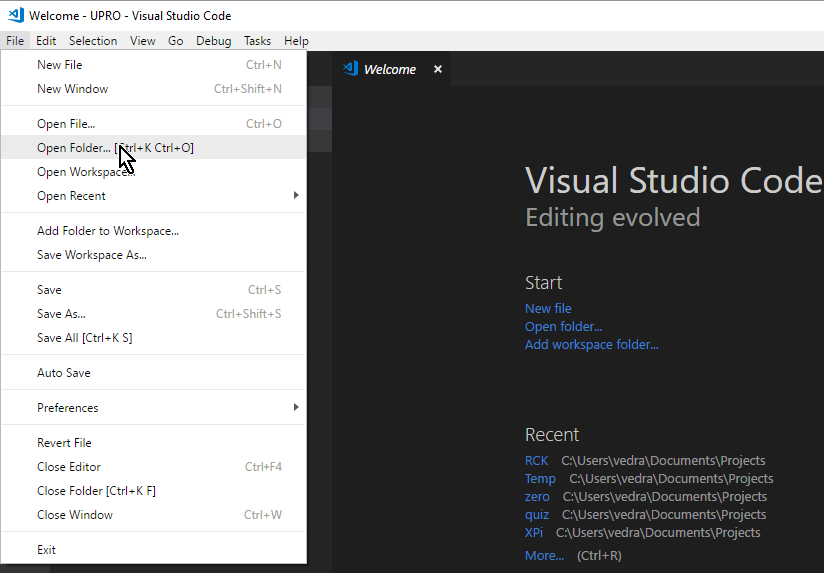
**Installation of the environment**

Visual Studio Code (VSC) is downloaded from the link

<https://code.visualstudio.com/download>

After selecting the platform, downloading and running the installer and successfully completing the installation, VSC is ready to launch.

The VSC desktop treats a single folder within the file system as a project. Therefore, the first action to take is to open the folder in which the program files are or will be located, such as the folder where we tested the compiler gcc:



The VSC toolbar consists of the following icons:



Explorer (work with files in the currently open folder)

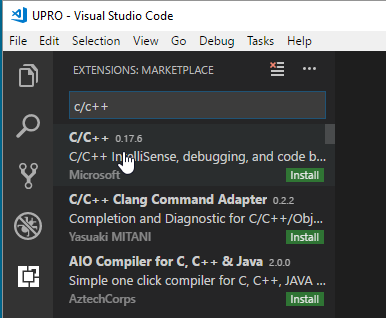
Search (search for content in the currently open folder and subfolders)

Source control (working with GIT version control system)

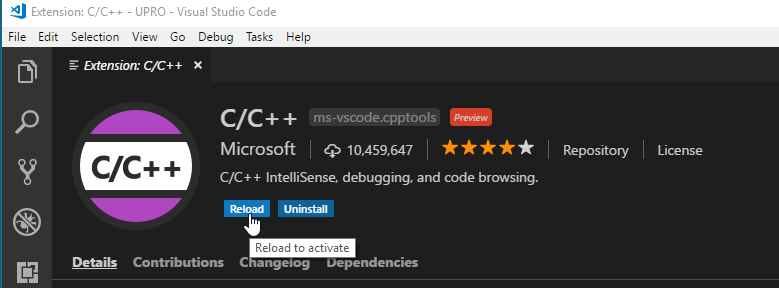
Debug (running a program and troubleshooting a program)

Extensions (installing a VSC extension)

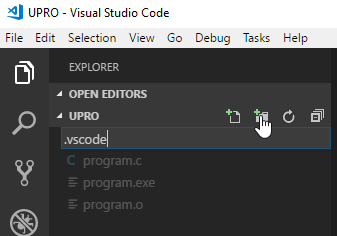
First, you need to install an extension that allows you to run programs in the C or C ++ programming language. In the extension search box, type C / C ++ and then select the Microsoft extension according to the following figure:



After downloading, it is necessary to restart the VSC by pressing a button Reload.



**Configure the environment for simple programs**

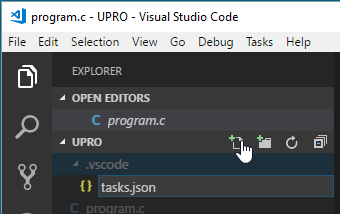
Unfortunately, VSC is not yet configured to compile C programs. Configuration can be done through menus, but the simplest way is to create a *.vscode* subfolder (note that the name begins with a dot):

and in this colder, create two files written in *Java Script Object Notation, JSON*:

tasks.json describes the tasks to be performed as part of the program compilation

launch.json describes how to run the executable program

Files are created by selecting the folder within which the files should be created, selecting the appropriate icon and typing in the name of the new file.



The following code should be copied to the window on the right in the *tasks.json* file:

{

"version": "2.0.0",

"tasks": [{

"label": "Build with gcc",

"type": "shell",

"command": "gcc",

"args": ["-g", "-std=c11", "-pedantic-errors",

"-Wall", "-o", "program.exe", "${file}"

]

}]

}

Basically, we are creating a task that will be called Build with gcc ("label"), which is an operating system command ("type":"Shell"), and the specific command that is executed is gcc ("Command"). "args" is a list of arguments of an external program call. ${file} is the name of the currently open program file, so the *program.c* is currently open in the window, the previous description will result in the execution of the following operating system command:

gcc -g -std=c11 -pedantic-errors -Wall -o program.exe program.c

that is, a known compiler call which will create *program.exe*.

The only novelty here is an option -g which adds an interaction with the debugger to the executable code.

It should be noted that this configuration is only good for programs that consist of a single source file, but therefore allows the same folder contain an unlimited number of such simple programs.

The file launch.json should be created in the same way, with the following content

{

"version": "0.2.0",

"configurations": [

{

"name": "C Launch",

"preLaunchTask": "Build with gcc",

"type": "cppdbg",

"request": "launch",

"program": "${workspaceRoot}/program.exe",

"args": [],

"stopAtEntry": false,

"cwd": "${workspaceRoot}",

"environment": [],

"externalConsole": true,

"miDebuggerPath": "gdb.exe",

"linux": {

"MIMode": "gdb"

},

"osx": {

"MIMode": "lldb"

},

"windows": {

"MIMode": "gdb"

}

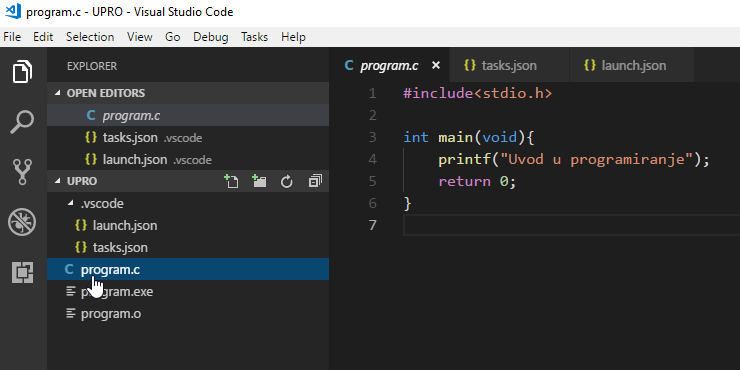
}

]

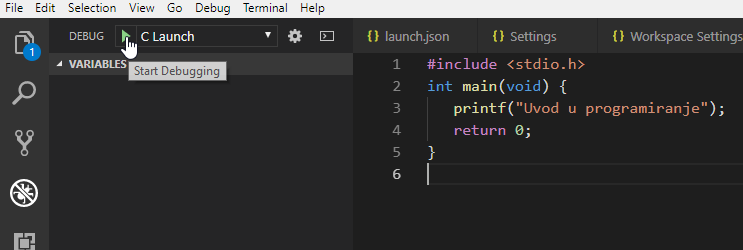
}

In short, we have given the name of the job that must be started before our program is started ("preLaunchTask"), the path to our program ("program") and the path to the debugger("miDebuggerPath"). The name of the job ("preLaunchTask") must match the text in the option "*label*" in *tasks.json*. It should be noted that in this case, the character separating the levels of folders in the slash (/) not the backslash (\).

In the above configuration("miDebuggerPath") it is understood that system variable *Path* contains the folder in which *debugger* *gdb.exe* is installed. If this is not the case, the correct absolute path to the file *gdb.exe* must be entered.

Now all we have to do is open our source program file (because the configuration specifies that the program currently open in the editor is compiled)

select *Debug* and start the program by pressing Start debugger or simply by pressing the key F5 on the keyboard



Consequently, the program will be compiled as described in *prelaunch task* in *tasks.json* and then, if compilation is successful, the program will be executed described in launch.json.

If the program has been modified in any way in the meantime, mark *x* after the program name

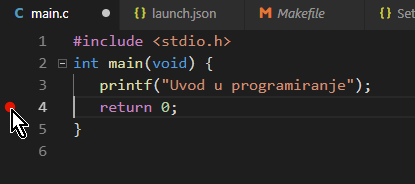


will change into a circle



and changes will need to be saved before the compilation with File/Save or Ctrl-S.

This action will print the text Introduction to Programming in a new window, which will then close immediately. To prevent the window from closing immediately, a breakpoint should be placed on the last line of the program by clicking the mouse at the very beginning of the line containing the return command.



If a new folder is created for another program or programs, *.vscode* folder should be copied there and all the preparatory actions are done.

**The basics of working with a debugger**

Program behavior is easily monitored using debugger. To demonstrate how such a program works, we will replace the content of *program.c* with

#include<stdio.h>

int main(void){

int i;

i = 0;

i = i + 1;

printf("Value of i: %d\n", i);

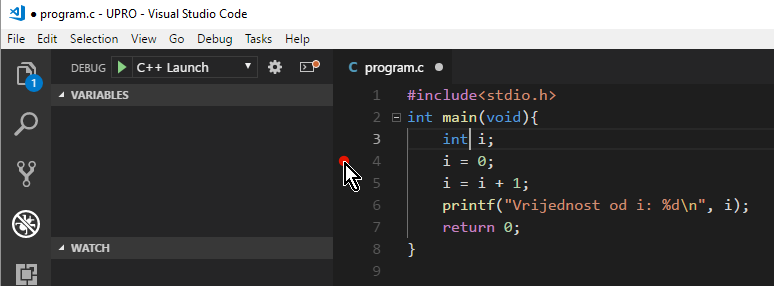
return 0;

}

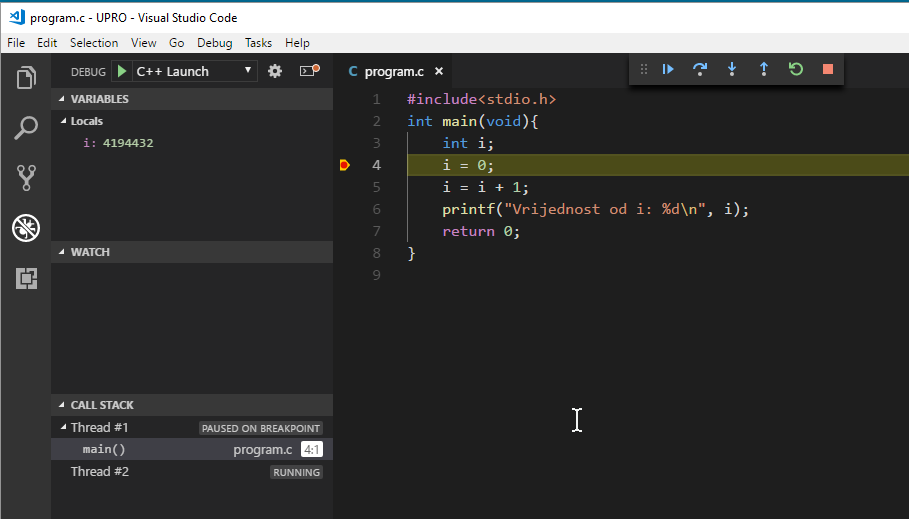
and monitor the execution of that program step by step.

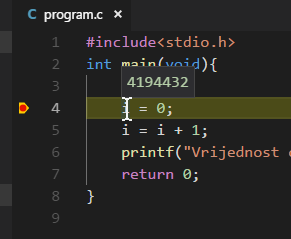
In short, we define one variable here, set it’s initial value, increase it by 1, and finally print its final value in a window with accompanying text.

In the program, we first set a breakpoint at the statement where to pause the execution, by left-clicking at the very beginning of the corresponding line.



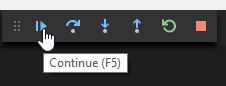
After compilation, the program starts executing and then stops at the breakpoint:



Bringing the mouse cursor over a variable name gives you an insight into its value:

The number 4194432 should not be surprising because the definition of a variable only provides memory space, but not a default value.

Debugger’s toolbar responds with a popup window that lets you see the meaning of each icon together with the keyboard shortcut which triggers the corresponding action:



and contains the following commands (action names and corresponding shortcuts are listed in parentheses):

Continue running to the end or next breakpoint (Continue, F5)

Execute the following statement without entering the function code (Step over, F10)

Execute the following statement entering the function code (Step into, F11)

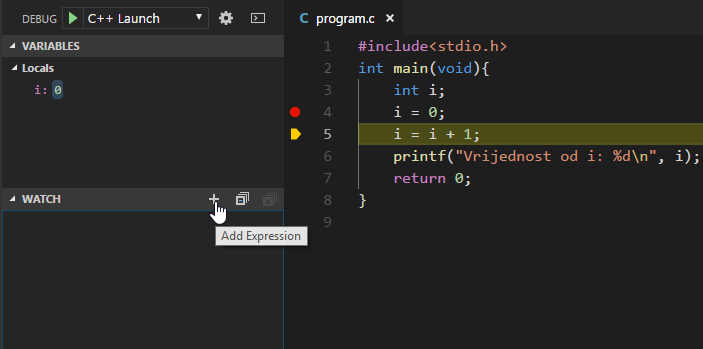
Exit the current function (Step out, Shift-F11)

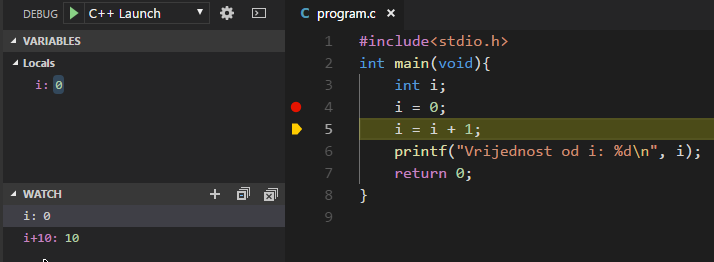
Restart the program with recompilation (Restart, Ctrl-Shift-F5)

Stop the program (Stop, Shift-F5)

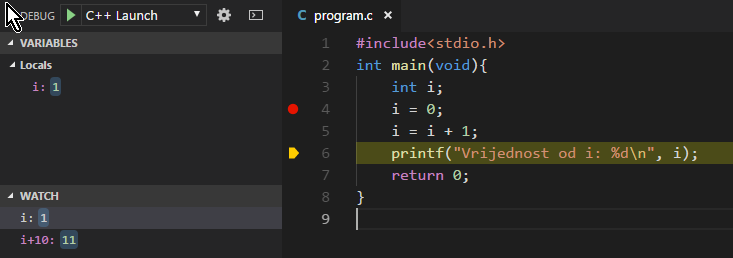
If the following statement is executed with the *Step over* action, the program stops after executing the command, and hovering over the variable name indicates that the content has changed.

The state of several variables in the surrounding of statement where the program stopped can be monitored in the *Locals window*. In larger programs, some variables that are not in immediate surrounding of the current statement may disappear from the *Locals window*. In such a case, the variable or expression whose value we want to watch constantly can be added to the Watch window:





The following figure shows the status after performing the statement i = i + 1:



If *Step into* action is triggered at this point, the debugger will enter *printf* system function and the statements comprising *printf* will be shown. This is rarely necessary. *Step ou*t action exits such a function, executing the rest of the commands in the function.

**Configure the environment for more complex programs**

More complex programs regularly consist of multiple source files (modules), because orientation in a large source file is difficult. If the program consists of files (modules) e.g. p1.c, p2.c i p3.c, in tasks.json, in line

"args": ["-g", "-std=c11", "-pedantic-errors",

"-Wall", "-o", "program.exe", "${file}"]

"${file}" shoud be replaced with list of modules comprising the program:

"args": ["-g", "-std=c11", "-pedantic-errors",

"-Wall", "-o", "program.exe", "p1.c", "p2.c", "p3.c"]

The disadvantage of this approach is that all modules listed here are compiled every timy, regardless of whether they have been modified in the meantime.

**Configure the environment for complex programs using *make***

Unnecessary compilations can be avoided using the program make, which utilizes executable program dependencies on object code files, and object code files on source code files. If the program detects that, for example, the source code file is newer than the object code file that depends on it, it will be compiled. Files whose source code is older than the corresponding object code will not be compiler.

Program make can be found in C: \MinGW\bin (or in the appropriate folder if MinGW is not installed with default destination) under a non-standard name mingw32-make.exe, so it is useful to copy it first to make.exe with copy

copy C:\MinGW\bin\mingw32-make.exe C:\MinGW\bin\make.exe

The dependencies are described by rules in the *Makefile* file, which should be located in the same folder as the source code files.

Makefile which describes the compilation of program.c to program.exe:

program.exe: program.o

gcc -o program.exe program.o

program.o: program.c

gcc -c -g -std=c11 -pedantic-errors -Wall program.c

Here we have defined that program.exe (executable code) depends on program.o (object code), and that program.o depends on program.c. If program.c is changed, this file will become newer than program.o, wich results, due to the option -c (*compile*), in compilation into object code, which will make the object code newer than the executable one, causing linking.

It is important to know that the indentation of a command that is executed for an appropriate rule is mandatory, with a tab character, and that the line describing the dependency of the executable code on the object code must be written first.

In practice, there will also be dependencies on the object code file and header (.h) files that are included in the compilation by the preprocessor command #include.

If, for example, a program consists of two modules, p1.c i p2.c where the first includes h1.h i h2.h, and the second includes h3.h, this will be defined in the following *Makefile*:

program.exe: p1.o p2.o

gcc -o program.exe program.o

p1.o: p1.c h1.h h2.h

gcc -c -g -std=c11 -pedantic-errors -Wall p1.c

p2.o: p2.c h3.h

gcc -c -g -std=c11 -pedantic-errors -Wall p2.c

tasks.json must now describe the call of program make. I can be done by replacing the *gcc* call with a *make* program call while deleting the argument list (make does not require arguments if dependencies are described in the file Makefile, otherwise the arguments at the call should be -f name\_of\_make\_file). Alternatively, in tasks.json a new task can be added as follows:

{

"version": "2.0.0",

"tasks": [{

"label": "Build with gcc",

"type": "shell",

"command": "gcc",

"args": ["-g", "-std=c11", "-pedantic-errors",

"-Wall", "-o", "program.exe", "${file}"

]

},

{

"label": "Build with make",

"type": "shell",

"command": "make"

}

]

}

In any case, care should be taken that the option"preLaunchTask" in launch.json references the corresponding task, in this example like this:

"preLaunchTask": "Build with make"

### Optional: Universal *Makefile*

In the previous examples, only the basic functionality of the program make was shown. With knowledge of more advanced functionalities, the details of which we will not dwell into here, it is possible to write *Makefile* which can be used to compile programs of any complexity, without explicitly specifying individual components and their dependencies. Very superficially explained, all the files with the extension *.c* are source files for the program, and all the files with the extension *.o* are object code and are derived from the source code of the same base name. The compiler name and options are stored in variables CC and CFLAGS.

CC = gcc

CFLAGS=-c -g -std=c11 -pedantic-errors -Wall

SRCS=$(wildcard \*.c)

INCLUDES=$(wildcard \*.h)

OBJS=$(SRCS:.c=.o)

main.exe: $(OBJS)

$(CC) -o $@ $(OBJS)

$(OBJS) : %.o: %.c $(INCLUDES)

$(CC) $(CFLAGS) -c $<

clean:

del \*.o \*.exe

There last rule allows us to delete all object and executable code files. This, like any other single rule, can be executing by calling the program make specifying the name of the rule to be executed, in this example make clean. The first subsequent *make* program call will cause all the program modules to be compiled.

**Formatting the code**

Properly formatted code is a prerequisite for the readability of the program. The C compiler will seamlessly compile the following program:

#include <stdio.h>

int main(void) { if

(1) { printf ("1"); } else {

printf ("2"); }}

but such a program it is almost impossible for a human to manage. Let’s compare the previous program with the equivalent but neatly formatted one:

#include <stdio.h>

int main(void) {

if (1) {

printf("1");

} else {

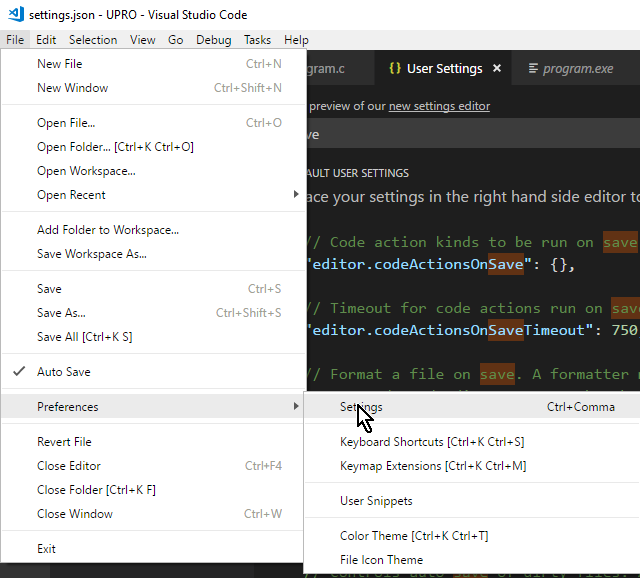
printf("2");

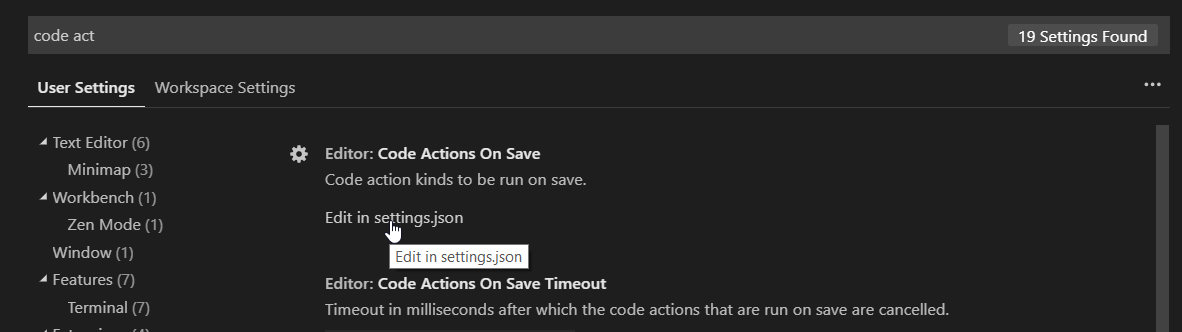
}

}

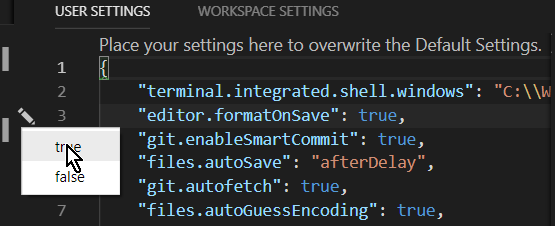
The VSC extension for C/C ++ will format the code with a command *Alt-Shift-F*.

The automatic code formatting while saving changes is built into this extension, but is turned off by default. The automatic code formatting setting while saving changes can be set in File/Preferences/Settings:



This, as well as all VSC settings, can be set at the user level (User settings, will be valid in any project/folder) or at the folder level (Workspace settings, will only be valid in the current folder). It is therefore necessary to first indicate the level (in the following example User settings to make the setting valid in other projects as well). Then search for the setting by typing the name of the setting in the search box. Some settings can be edited in the appropriate fields, some, for now (the environment is in constant development) by editing the JSON object. The specific setting is edited by editing the JSON object:

Click the pencil icon in the line *editor.formatOnSave* and select *true.*



The default formatting style is somewhat wasteful. While saving the above program code, it will be automatically formatted as follows:

#include <stdio.h>

int main(void)

{

if (1)

{

printf("1");

}

else

{

printf("2");

}

}

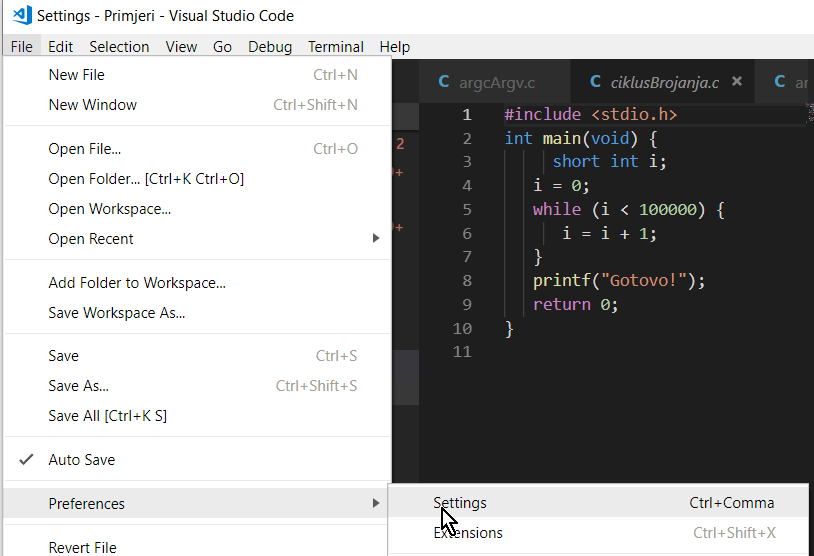
This can be changed by adding a file named .clang-format to the folder containing the source code.

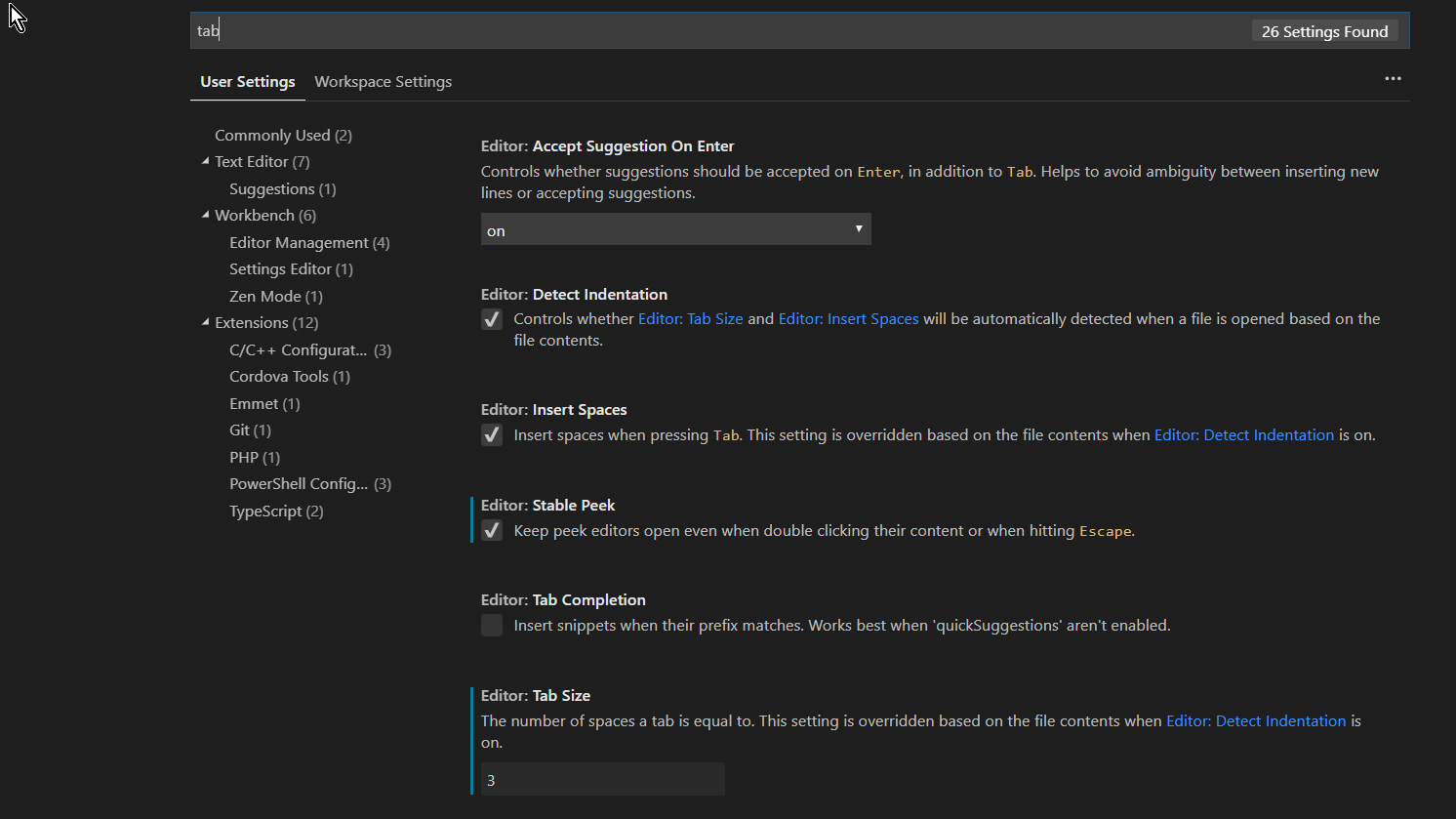
This file should contain the setting IndentWidth by which we determine the depth of indentation of each level of program statements. It is recommended to indent by 3 letters

IndentWidth: 3

The blank behind the colon is required.

The tab width in the VSC settings also needs to be changed to format properly:



In the search box in the window that opens to the right, type a tab and set the Tab size to 3 (default value is 4):

With the existence of the file .clang-format, the program will be formatted optimally when saved, in terms of the number of program lines consumed and its readability:

#include <stdio.h>

int main(void) {

if (1) {

printf("1");

} else {

printf("2");

}

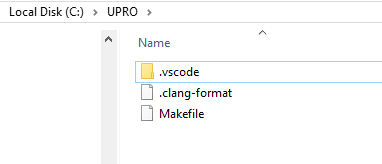
}

It is also possible to enter the *BasedOnStyle* setting in this file to determine the formatting style. Allowed values are LLVM (this is also the default value), Google, Chromium, Mozilla and WebKit.

We leave it to the readers to experiment with these settings.

# VSC configuration files

The *configuration.zip* file contains all the configuration files described in this text. The entire contents of this compressed file should be copied to a folder containing the original programs, for example:

*C:\IPRO*, so after copying the folder looks like this.

The folder can then be opened in the VSC and all is ready to edit and compile a program.

There are different configurations built into these configuration files, so you need to select the *C Launch* configuration to be used for compilation and execution.

