# How config Open WATCOM Compiler C_C++ (32 and 64 bits) into CodeBlocks

Name of tutorial : How config Open WATCOM Compiler C/C++ (32 and 64 bits) into Code::Blocks on Windows 11 64 bits.

```
Code::Blocks : the best and great free IDE for Windows, Linux and ... Mac OS
```

During first run of CB on Windows, this IDE detect automatically some compilers, or present one list of them pre-configured.
It's very good functionnality, but, sometimes, you must "force" these configurations proposed by default to run correctly.
This tuto describe how configure one compiler C\C++ on Windows : Open WATCOM Compiler C/C++ (version 32 and 64 bits).

How to install Open WATCOM Compiler C/C++ (32 and 64 bits) ?

You can download it from Internet site GITHUB : https://github.com/open-watcom/open-watcom-v2/releases

Then, you select last "pre-release" version V2.0c, and donwload file "open-watcom-2_0-c-win-x64.exe", click on these files to
install Open Watcom C/C++ compiler on "C:\WATCOM" directory (by default). This "pre-release" is good update of initial version
available on SourceForge in "stable" version 1.9 very outdated, and activity on GitHub about this fork is very sustained.

Big interest about Open WATCOM C\C++ compiler provide it can be run on multiple Intel x86 platforms and generate targets on multiple platforms, including 16-bits :
a) Host Platforms :
DOS (command line only)
32-bit OS/2 (IDE and command line)
Windows 3.x (IDE)
Windows 95/98/Me (IDE and command line)
Windows NT/2000/XP upto Windows 11 (IDE and command line)
Linux (command line only)
b) 16-bits target platforms :
DOS
Windows 1 upto Windows 3.x

OS/2 1.x
c) 32-bits target platforms
Extended DOS
Win32s
Windows 95/98/Me
Windows NT/2000/XP ... up to Windows 11
32-bit OS/2
Novell NLMs
Linux 32/64-bit Intel CPU

WARNING : Pre-processor test of presence of this compiler by only "#ifdef **WATCOMC**" don't inform about platform used, only
inform about type of compiler C\C++ used ... and it's normal ... (Is it not ? -) )
In your code, you can find this variable equally on Win32 platforms that on Linux platforms...
Good tests are :
"#if defined(_WIN32) && defined(**WATCOMC**) / *Test ok to detect OW run on Win32 platforms (x86 or x64)* /
"#if defined(**linux**) && defined(**WATCOMC**) / *test ok to detect OW run on linux platforms (x86 or x64)* /

Yes, today, with generalization of 64 bits platforms, it can be considered "obsolete", but it can be important to conserve
these "old" possibilities in specific context.
But, big restriction, same with 64 bits version of this compiler on Windows 11, you can't generate 64 bits version of targets.

Normally, after that, next run of CB detect presence of these compilers and proposed it in list of available compiler in main
menu "Settings" and after submenu "Compiler..." : "OpenWatcom Compiler" (autodetect by CB).
If you select this, verify that fields describe next are parametered into CB.

In tab "Toolchain executable", you must find in field "Compiler installation directory" :
C:\WATCOM (subdirectory "\bin" automatically searched after this "top" directory),
and in subtab "Program Files", list next :
compilateur C : wcl386.exe
compilateur C++ : wcl386.exe
linker for dynamic lib : wlink.exe
linker for static lib : wlib.exe
debugger :
resource compiler : wrc.exe
make program : wmake.exe

It's not enougth, because binaries of Open Watcom Compiler are not in subdirectory ".\bin" but in ".\binnt", then you must add
in subtab "Additional Paths" and click on button "Add" to type in new subwindows :
"C:\WATCOM\binnt".

If CB propose different values of fields described below, you can change/force it.

After, you select tab "Search directories", and into each subtab, you write with "add" button, if not searched by default :
to compiler : C:\WATCOM\h\nt and C:\WATCOM\h
to linker : C:\WATCOM\lib386\nt
to resource compiler : C:\WATCOM\h\nt and C:\WATCOM\h

It's recommended then to select an option in tab "Compiler Settings" and in subtab "Compiler Flags" to select "Compile and Link
for NT (includes Win32) [-bcl=nt]" if you want generate and test your target on Windows NT/7/8/10/11 operating systems.

And, with simply source "hellowworld.c", you can test generation of program into IDE CB, choosing "create new project" in main
windows of CB, and choose "console application" with no source proposed by default, because named "main.c" by default, and choose
compiler "OpenWatcom Compiler".
You can select good directory/source with option "add file" after first creation of project into CB.

One time project created, you can generate it with selecting main menu "Build" and choose submenu "Rebuild..." (or CTRL-F11).

If, you apply all of precedent instructions, compile and link of your program must be succeeded.

For fun, you can also define version 64 bits of Open WATCOM compiler, into CB. You must first return in main menu "Settings" and
after submenu "Compiler..." to choose "OpenWatcom Compiler", after click on button "Copy" and to terminate rename it with type
"OpenWatcom Compiler (64b)" by button "Rename", by example (=> force identification of "new" compiler into CB).

After, verify that in tab "Toolchain executable", you must find in field "Compiler installation directory" :
C:\WATCOM (subdirectory "\bin" automatically searched after this "top" directory),
and in subtab "Program Files", list next :
compilateur C : wcl386.exe
compilateur C++ : wcl386.exe

linker for dynamic lib : wlink.exe
linker for static lib : wlib.exe
debugger :
resource compiler : wrc.exe
make program : wmake.exe

Then you must change in subtab "Additional Paths" and click on button "Edit" to retype in new subwindows : "C:\WATCOM\binnt64".

You can verify also presence in tab "Search directories", and into each subtab (same with version 32 bits) :
to compiler : C:\WATCOM\h\nt and C:\WATCOM\h
to linker : C:\WATCOM\lib386\nt
to resource compiler : C:\WATCOM\h\nt and C:\WATCOM\h

And, with simply source "hellowworld.c", you can test generation of program into IDE CB, choosing "create new project" in main
windows of CB, and choose "console application" with no source proposed by default, because named "main.c" by default, and choose
compiler "OpenWatcom Compiler (64b)".
You can select good directory/source with option "add file" after first creation of project into CB.

One time project created, you can generate it with selecting main menu "Build" and choose submenu "Rebuild..." (or CTRL-F11).

Note that target rest in version 32 bits !!! Little interest ...

Pleasure of programming is open for you, your imagination is illimited, at your keyboard ! Enjoy !

PS : source file "hellowworld.c" :

/ *Basic example in language C : hellowworld.c* /

```
#include <stdio.h>

int main(int argc, char argv[]) {
/ printf() displays the string inside quotation */
printf("Hello, World!");
return 0;
}
```

PS2 : Open Watcom compiler can be used directly into command console of Windows (CMD.EXE) and next command lines configure it :

SET PATHSAV=%PATH%
SET INCSAV=%INCLUDE%
SET LIBSAV=%LIB%
SET WATCOM=C:\WATCOM
REM Choose if you use binary directory of OW beetween 32 bits or 64 bits by suppress "[64]" after or conserve only "64" beetween []
SET PATH=%WATCOM%\binnt[64];%WATCOM%\BINW;%PATH%
SET INCLUDE=%WATCOM%\h\nt;%WATCOM%\h;%INCLUDE%
SET LIB=%WATCOM%\lib386\nt;%LIB%

REM Generate console application in one pass
wcl386 hellowworld.c -fe=hellowworld.exe
REM Generate console application in two pass
wcl386 -c hellowworld.c -fo=hellowworld.obj
wlink system nt file hellowworld.obj name hellowworld.exe

After, work with Open Watcom compiler, but, at the end of your work, think to return in initial state ... to avoid difficulties :

.....
SET PATH=%PATHSAV%
SET INCLUDE=%INCSAV%
SET LIB=%LIBSAV%

PS3 : Options of command line utilities "wcl386" and "wlink" for Open Watcom compiler :

Options are prefixed with a slash (/) or a dash (−) and may be specified in any order.

To "wcl386.exe" command ("wcc386" is same but restrict to compilation only) :

```
        [General options]
c                               compile the files only, do not link them
cc                              treat source files as C code
cc++                    treat source files as C++ code
y                               ignore the WCL/WCL386 environment variable
        [Compiler options]
0 (16-bit only) 8088 and 8086 instructions (default for 16-bit)
1 (16-bit only) 188 and 186 instructions
2 (16-bit only) 286 instructions
3 (16-bit only) 386 instructions
4 (16-bit only) 486 instructions
5 (16-bit only) Pentium instructions
```

```
6 (16-bit only) Pentium Pro instructions
3r (32-bit only) generate 386 instructions based on 386 instruction timings
and use
                                register-based argument passing conventions
3s (32-bit only) generate 386 instructions based on 386 instruction timings
and use
                                stack-based argument passing conventions
4r (32-bit only) generate 386 instructions based on 486 instruction timings
and use
                                register-based argument passing conventions
4s (32-bit only) generate 386 instructions based on 486 instruction timings
and use
                                stack-based argument passing conventions
5r (32-bit only) generate 386 instructions based on Intel Pentium instruction
timings and use
                                register-based argument passing conventions
(default for 32-bit)
5s (32-bit only) generate 386 instructions based on Intel Pentium instruction
timings and use
                                stack-based argument passing conventions
6r (32-bit only) generate 386 instructions based on Intel Pentium Pro
instruction timings and
                                use register-based argument passing
conventions
6s (32-bit only) generate 386 instructions based on Intel Pentium Pro
instruction timings and
                                use stack-based argument passing conventions
aa (C only)     allow non-constant initializers for local aggregates or unions
ad[=<file_name>] generate make style automatic dependency file
adbs                    force path separators generated in auto-dependency
files to backslashes
add[=<file_name>] specify source dependency name generated in make style auto-
dependency file
adhp[=<file_name>] specify path to use for headers with no path given
adfs                    force path separators generated in auto-dependency
files to forward slashes
adt[=<target_name>] specify target name generated in make style auto-
dependency file
bc                              build target is a console application
bd                              build target is a Dynamic Link Library (DLL)
```

```
bg                              build target is a GUI application
bm                              build target is a multi-thread environment
br                              build target uses DLL version of C/C++ run-
time libraries
bt[=<os>]               build target for operating system <os>
bw                              build target uses default windowing support
d0 (C++ only)   no debugging information
d1                              line number debugging information
d1+ (C only)    line number debugging information plus typing information for
global symbols
                                and local structs and arrays
d2                              full symbolic debugging information
d2i (C++ only)  d2 and debug inlines; emit inlines as external out-of-line
functions
d2s (C++ only)  d2 and debug inlines; emit inlines as static out-of-line
functions
d2t (C++ only)  full symbolic debugging information, without type names
d3                              full symbolic debugging with unreferenced type
names
d3i (C++ only)  d3 plus debug inlines; emit inlines as external out-of-line
functions
d3s (C++ only)  d3 plus debug inlines; emit inlines as static out-of-line
functions
d<name>[=text]  preprocessor #define name [text]
d+                              allow extended -d macro definitions
db                              generate browsing information
e<number>               set error limit number (default is 20)
ecc                     set default calling convention to __cdecl
ecd                     set default calling convention to __stdcall
ecf                     set default calling convention to __fastcall
ecp                     set default calling convention to __pascal
ecr                     set default calling convention to __fortran
ecs                     set default calling convention to __syscall
ecw                     set default calling convention to __watcall (default)
ee                              call epilogue hook routine
ef                              use full path names in error messages
ei                              force enum base type to use at least an int
em                              force enum base type to use minimum
en                              emit routine name before prologue
ep[<number>]    call prologue hook routine with number of stack bytes
```

```
available
eq                              do not display error messages (they are still
written to a file)
er (C++ only)   do not recover from undefined symbol errors
et                              Pentium profiling
ew (C++ only)   generate less verbose messages
ez (32-bit only) generate Phar Lap Easy OMF-386 object file
fc=<file_name> (C++ only) specify file of command lines to be batch processed
fh[q][=<file_name>] use precompiled headers
fhd                     store debug info for pre-compiled header once (DWARF
only)
fhr (C++ only)  force compiler to read pre-compiled header
fhw (C++ only)  force compiler to write pre-compiled header
fhwe (C++ only) don't include pre-compiled header warnings when "we" is used
fi=<file_name>  force file_name to be included
fo=<file_name>  set object or preprocessor output file specification
fpc                     generate calls to floating-point library
fpi (16-bit only) generate in-line 80x87 instructions with emulation (default)
        (32-bit only) generate in-line 387 instructions with emulation
(default)
fpi87 (16-bit only) generate in-line 80x87 instructions
        (32-bit only) generate in-line 387 instructions
fp2                     generate in-line 80x87 instructions
fp3                     generate in-line 387 instructions
fp5                     generate in-line 80x87 instructions optimized for
Pentium processor
fp6                     generate in-line 80x87 instructions optimized for
Pentium Pro processor
fpd                     enable generation of Pentium FDIV bug check code
fpr                     generate 8087 code compatible with older versions of
compiler
fr=<file_name>  set error file specification
ft                              try truncated (8.3) header file specification
fti (C only)    track include file opens
fx                              do not try truncated (8.3) header file
specification
fzh (C++ only)  do not automatically append extensions for include files
fzs (C++ only)  do not automatically append extensions for source files
g=<codegroup>   set code group name
h{w,d,c}                set debug output format (Open Watcom, Dwarf, Codeview)
```

```
i=<directory>   add directory to list of include directories
j                               change char default from unsigned to signed
k (C++ only)    continue processing files (ignore errors)
m{f,s,m,c,l,h}  memory model
                                        mf=flat
                                        ms=small
                                        mm=medium
                                        mc=compact
                                        ml=large
                                        mh=huge
                                (default is "ms" for 16-bit and Netware, "mf"
for 32-bit)
nc=<name>               set name of the code class
nd=<name>               set name of the "data" segment
nm=<name>               set module name different from filename
nt=<name>               set name of the "text" segment
o{a,b,c,d,e,f,f+,h,i,i+,k,l,l+,m,n,o,p,r,s,t,u,x,z} control optimization
pil                     preprocessor ignores #line directives
p{e,l,c,w=<num>} preprocess file only, sending output to standard output
                                        "c" include comments
                                        "e" encrypt identifiers (C++ only)
                                        "l" include #line directives
                                        "w=<num>" wrap output lines at <num>
columns (zero means no wrap)
q                               operate quietly
r                               save/restore segment registers
ri                              return chars and shorts as ints
s                               remove stack overflow checks
sg                              generate calls to grow the stack
st                              touch stack through SS first
t=<num> (C++ only) set tab stop multiplier
u<name>                 preprocessor #undef name
v                               output function declarations to .def file
(with typedef names)
vc... (C++ only) VC++ compatibility options
w<number>               set warning level number (default is w1)
wcd=<num>               warning control: disable warning message <num>
wce=<num>               warning control: enable warning message <num>
we                              treat all warnings as errors
wo (C only) (16-bit only) warn about problems with overlaid code
```

```
wx                              set warning level to maximum setting
x                               preprocessor ignores environment variables
xd (C++ only)   disable exception handling (default)
xdt (C++ only)  disable exception handling (same as "xd")
xds                     no exception handling: space
xs                      exception handling: balanced
xss                     exception handling: space
xst                     exception handling: time
    [Preprocessor options]
d+              extend syntax of -d option
d<name>[=text] define a macro
fo=<file>       set object file name
i=<path>        include directory
p{c,l,w=<num>} Preprocess source file
                                c               - preserve comments
                                l               - insert #line directives
                                w=<num>         - wrap output at column <num>
pil             ignore #line directives
t=<num> (C++ only) <num> of spaces in tab stop
tp=<name> (C only) set #pragma on( <name> )
u<name>         undefine macro name
    [Linker options]
"linker_directives" additional linker directives
@=<file>         additional directive file
bcl=<os>         compile and link for <os>
bd               build Dynamic link library
bm               build Multi-thread application
br               build with dll run-time library
bw               build default Windowing app.
fd[=<file>]      write directives
fe=<file>        name executable file
fm[=<file>]      generate map file
k<stack_size>    set stack size
l=<target>       link for the specified <target>
```

Options of command "wlink" can be seen simply with type "wlink" without parameters. It's a complex command, very
verbose, but you can focalize to "Windows NT" generation. By example, you can type "wlink > command_wlink.txt" to
see all content in text file, and after select that you want use.

Just to illustrate, next command is used to generated an console application after one step of compilation :

"wlink debug all system nt LIBP "%LIB%" file %OBJS% objOW32\Debug%NAME_APPLI%.obj option resource=objOW32\Debug%NAME_APPLI%.res
name binOW32\Debug%NAME_APPLI%.exe library glu32,opengl32,advapi32,comdlg32,gdi32,winmm,user32,kernel32"

with many variables to adapt at context, where parameters are :

```
"debug all"             to generate version "Debug" (suppress it to "Release"
version)
"system nt"                     to generate console application on Win32
platforms ("system nt_win" to GUI application Win32)
"LIBP ...."             to list library directories (separator is ";")
"file ...."             to list all object files in input (here contents first
in %OBJ% variable, and close by objOW32\Debug\%NAME_APPLI%.obj)
                                                (separator is " ")
"option resource=..." to provide name of resource file used by linker (many
other options can be defined)
"name ..."                      to force name of output file (here an
executable file)
"library ..."           to list all libraries needed by linker (separator is
",")
```

All documentation (updated) is available on https://open-watcom.github.io/open-watcom-v2-wikidocs (files pdf or html).

Another example to generate an "Release"" DLL (and import library "ad hoc" in parallel) after one step of compilation :

"wlink system nt_dll LIBP "%LIB%" IMPLIB binOW32\Release%NAME_APPLI%.lib file %OBJS% objOW32\Release%NAME_APPLI%.obj
option resource=objOW32\Release%NAME_APPLI%.res name binOW32\Release%NAME_APPLI%.dll library glu32,opengl32,advapi32,comdlg32,gdi32,winmm,user32,kernel32"

And use of OW into IDE CB is very simplified than use into command line ... -)

PS4 : Open Watcom Compiler use by default a specific "calling convention" called "watcall". If you want shared
your development beetween another compiler and Open Watcom, it's better to use another like

"**cdecl" or "**stdcall"
by positionning "-ecc" or "-ecd" flag during compilation/generation. Idem for format of debugging, by default "Open
Watcom", but you can select beetween "Dwarf" or "Codeview" by positionning "-hd" or "-hc" during compilation/generation.