MS Visual Developer™

V6.0

Rex Jaeschke

MS Visual Developer V6.0

© 1996, 1998, 2009 Rex Jaeschke.

Edition: 4.0

The information in this book is subject to change without notice, and should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and the publisher assume no responsibility for errors or omissions.

Visual C++ and Windows are trademarks of Microsoft.

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

The on-line help facility for Developer Studio contains a lot of useful information about using the Developer Studio and supporting tools. However, that information really isn't organized in a tutorial fashion, and no source examples are supplied for testing. The following information, along with the accompanying demonstration files, is intended to get people "up to speed" with the Visual C++ development environment in less than a day assuming they have a reasonable grasp of C or C++, and a working knowledge of using Windows-based programs.

Before staring this tutorial, you should create a directory called Source and copy the demonstration directories and files to that directory.

Lines of the following form:

- Select File|Open Workspace.
- Select the directory \Source\PRJ01.
- …

indicate a series of actions you should perform.

 All mouse click instructions assume the mouse is configured for right-hand use.

Rex Jaeschke, November 2009

1.   Introduction to Developer Studio

- Start Developer Studio.
- Right-click on an unoccupied portion of the menubar, such as to the right of the Help menu, and make sure that the only entries checked in the drop-down list are Standard, Output, and Workspace.

Those entries below the separator represent the toolbars that can be displayed or hidden. (An alternative way to display the output and workspace windows is to select View|Output and View|Workspace, respectively. Toolbars can also be displayed and hidden using Tools|Customize|Toolbars.)

You should now see a display something like the following:
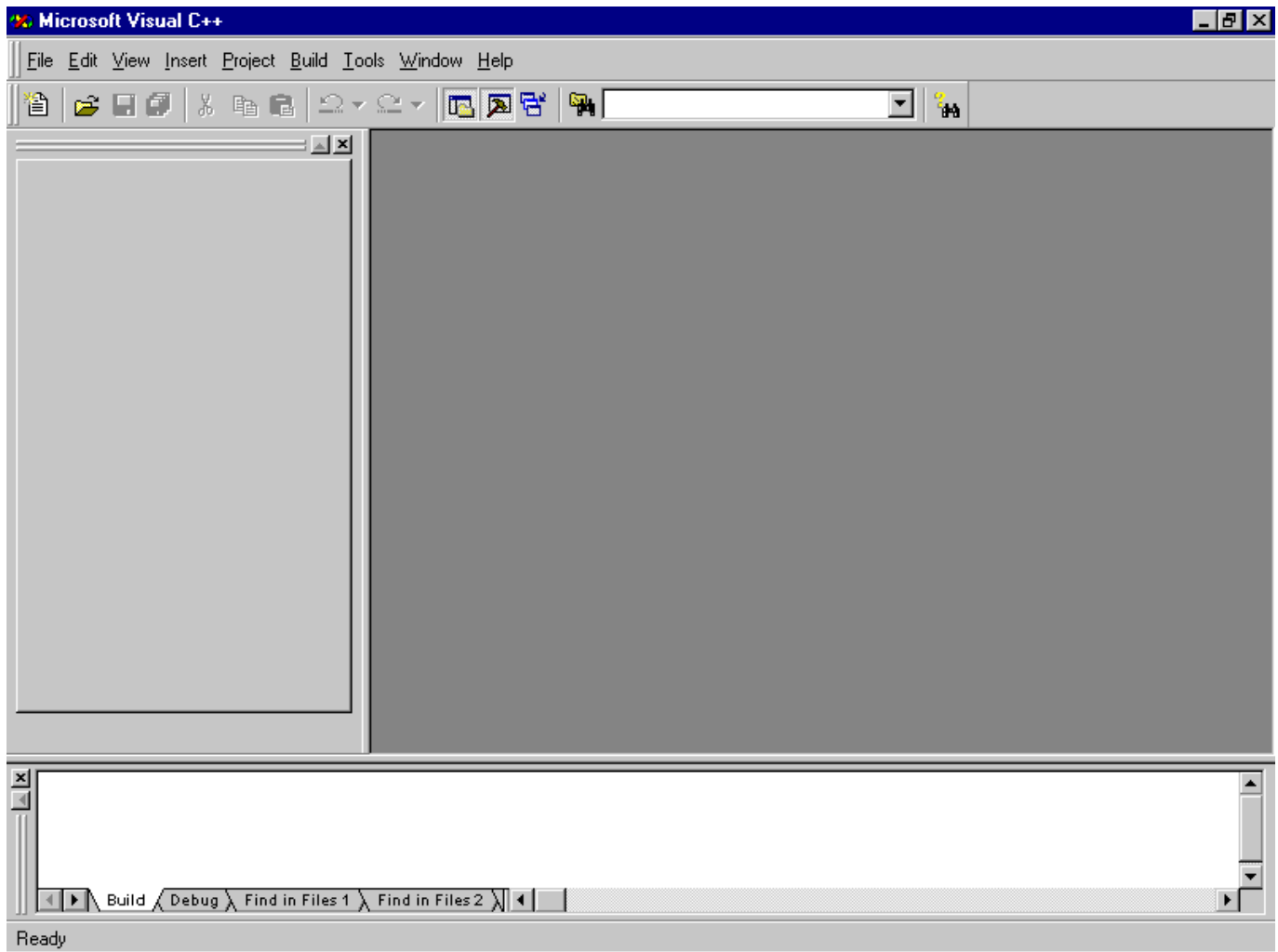
- Select Tools|Options|Format|Reset All. This reinitializes the text typeface and source file element color scheme to the default settings.

1

Across the top is a set of pull-down menus. Under that, there is a row of buttons representing the Standard toolbar. The rest of the display is broken into three separate windows, some with tabs.

Before you can build a program or even compile a single source module, you must create a *workspace* containing at least one *project*. But rather than explain how to do that just yet, you'll simply load an existing workspace.

- Select File|Open Workspace.
- Select the directory \Source\PRJ01.
- Select the file PRJ01.DSW; this loads the previously saved workspace for the first demonstration program.
- Without clicking, hold the cursor over the Standard toolbar button containing the picture of a hammer. After a few seconds, a small window should be displayed telling you the purpose of that button.
- Press that button once; press it again.
- Do likewise for the one to its immediate left. These little windows are called *ToolTips*, and they are available for all toolbar buttons. (If Tooltips are not displayed, they are probably disabled, in which case, select Tools|Customize|Toolbars, and check the box labeled Show ToolTips.)
- Look for some other ToolTips.

The Workspace window that appears on the top left side of the screen has two small tabs on its bottom left side, and each tab leads to a different *pane*.

The pane that is currently displayed is the ClassView, and its tab contains three colored rectangles. The closed folder shown is preceded by a + sign to indicate it contains something.

- Left-click on the + sign to open the folder.
- This results in two more entries; expand both.

The first entry signifies a structure type `struct x` and, as you can see, it has two members, d and y. The second entry is called Globals, and shows all the identifiers in the program that have external linkage (that is, names that are visible outside their parent source file). Object names are preceded with a light blue box that slopes to the right while function names are preceded with a light purple box that slopes to the left, as shown below.

# 3.	Configurations and Subprojects

## 3.1	Project Configurations

Consider the following scenario: You have an application for which you'd like to build three different executables based on the presence or value of one or more macros. And for each of these three, you want both a Debug and a Release version. Therefore, in total, you need to generate six different executables all from the same source, based on compiler and/or linker options. To be more specific, the three main flavors of your application will be internationalized with respect to supporting the ANSI Single-Byte Character Set (SBCS), the Shift-JIS MultiByte Character Set (MBCS) as used by Japanese, and Unicode, respectively. Now if you don't know or care about internationalization, don't panic, the example chosen is simple enough so that it won't matter; the point of this exercise is to learn about configurations, not internationalization.

Here's the source file (i18n.c) from which you'll generate these six versions:

```c
#include <stdio.h>
#include <tchar.h>

main()
{
#if defined(_DEBUG)
    _tprintf(_T("Debug mode\n"));
#endif

#if defined(NDEBUG)
    _tprintf(_T("Nondebug mode\n"));
#endif

#if defined(_UNICODE)
    _tprintf(_T("Unicode mode\n"));
#endif

#if defined(_MBCS)
    _tprintf(_T("MBCS mode\n"));
#endif

#if !defined(_UNICODE) && !defined(_MBCS)
    _tprintf(_T("SBCS mode\n"));
#endif

    return 0;
}
```

The six executables will be built based on the following combinations of attributes:

1.	SBCS/Debug	_DEBUG defined
2.	SBCS/Release	NDEBUG defined
3.	MBCS/Debug	_DEBUG, _MBCS defined
4.	MBCS/Release	NDEBUG, _MBCS defined
5.	Unicode/Debug	_DEBUG, _UNICODE defined
6.	Unicode/Release	NDEBUG, _UNICODE defined

For example, the code produced by the preprocessor using combination 1 is:

```
main()
{
    printf("Debug mode\n");
    printf("SBCS mode\n");

    return 0;
}
```

while that from combination 6 is:

```
main()
{
    wprintf(L"Nondebug mode\n");
    wprintf(L"Unicode mode\n");

    return 0;
}
```

The way in which you'll build this application is to define a workspace that has one project which, in turn, has six configurations. Here are the steps:

- Create a new workspace. (I called it PRJ07, placed it in directory \Source, and made its project a console application.)
- Create the source file i18n.c in \Source\PRJ07 and insert that file into the project just created.
- Select Build|Configurations and you should see the following:

© 1996, 1998, 2009 Rex Jaeschke.
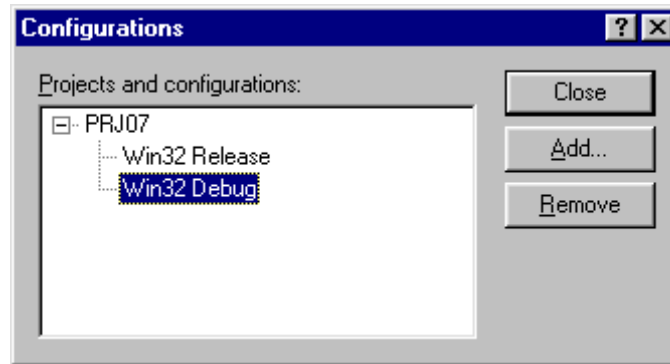
- PRJ0 PRJ07 contains the two configurations, Release and Debug.
- Select the Add project configuration option. In the drop-down box Copy settings from, select the "Debug" configuration. Under Configuration, enter the name "Debug Unicode".
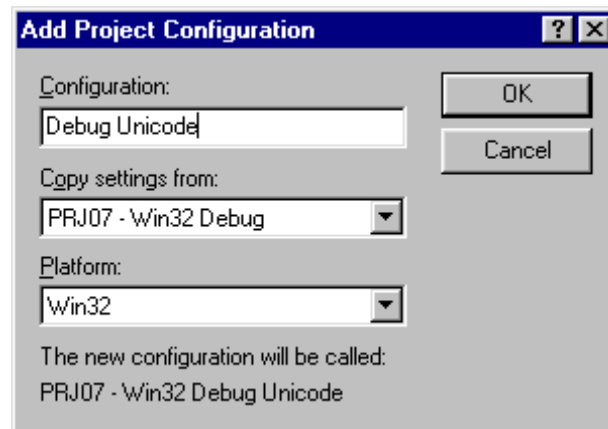
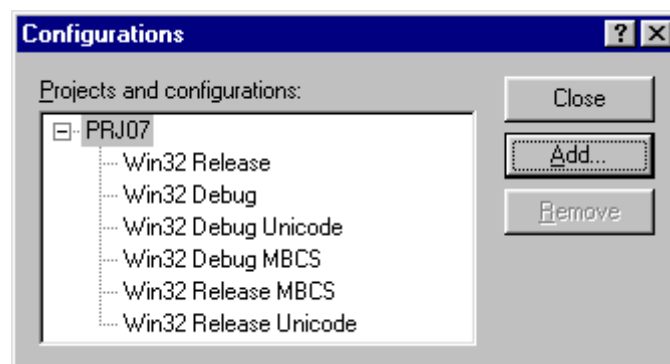- Select OK and the new configuration (cloned from an existing one) gets created.
- Repeat this procedure creating the three other configurations by cloning "Debug" to "Debug MBCS" and by cloning "Release" to "Release Unicode" and "Release MBCS". Now you have six configurations, as follows:

- Close the configuration window.
- The Build toolbar contains a drop-down box called Select Active Configuration. Select it and the following appears:

6. Debugging

6.1 Introduction

- Load the workspace \Source\PRJ01 and build the debug configuration of that project.
- Start the debugger by selecting Build|Start Debug|Step Into. (If you get a complaint about "can't find source file CRT0.C", change Settings|debug from None or line numbers to Program Database instead.)

To specify command-line arguments to a program being debugged, select PRJ0|Settings|Debug|Category General|Program arguments first.

The debug cursor (the yellow arrow on the left) is positioned at the start of `main`, the program's entry point. (For a Windows program, it would be positioned at `WinMain`.) While the debugger is active, the Build menu option is replaced by the Debug menu.

- Look at the entries in Debug menu.
- In the source window, hold the mouse cursor over a variable name without clicking. For scalar objects, the value of that variable is displayed. Try it with `global`.
- Also, try it on an aggregate such as an array or structure.
- Find the statement `table[0][0] = 3.5F` in `main`. Left-click and drag, selecting the expression `table[0][0]`. Now when you position the cursor over this marked area you will see the value of that expression; it's 1.2 because `main` has not yet begun executing.

Display the Debug toolbar. Now you can click on these toolbar buttons instead of making selections from the pull-down Debug menu.

Several useful debugging option buttons should be added to this toolbar if they are not already there: Go (white page with blue arrow pointing down), Insert/Remove Breakpoint (white hand), Remove All Breakpoints (two white hands with red X), Enable/Disable Breakpoint (white/gray hand), and Disable All Breakpoints (two white/gray hands with red X). You might find it convenient to add these to the debug toolbar. You do this as follows:

- Select Tools|Customize.
- Select the Commands tab and highlight Debug to see the list of the debugging buttons. (To learn what each button is for, press it and read the description shown to the left.)
- Position this window such that you can also see the debug toolbar.
- Drag and drop the five buttons mentioned above onto that toolbar, positioning them as you wish.
- To remove a button from the toolbar, simply drag and drop it outside the toolbar.
- Close the Customize window.

When floating, your debug toolbar should look something like this:
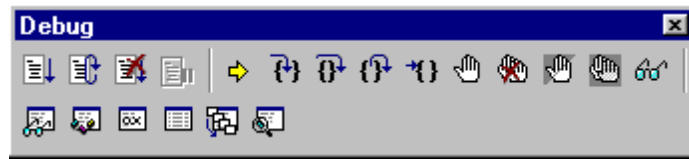
- Position the cursor over each button on the debug toolbar to see what it does. The last six buttons on the toolbar correspond to the last six entries in View|Debug Windows.
- Find the Stop debugging button and press it.
- Find the Restart button and press it. (If the Debug toolbar goes away when debugging is stopped, redisplay it.)
- Select the Variables button and look at the Auto, Locals, and this panes.

You can adjust the width of a pane and, for panes containing multiple columns, you can adjust the column width. The Auto} pane shows the value of the variables being used in the statements in the vicinity of the debug cursor; it usually has only one or two entries. The Local pane shows the value of each automatic or static variable local to the function being executed. Initially, it contains only static variables; however, as execution flows through new automatic variable definitions, those variables are added to this list. The `this` pane is for C++ programs and shows the value of the `this` pointer on entry to a given non-static member function. In all other contexts, its value is meaningless.

- In the Locals pane, expand all aggregate objects making sure you agree with their values. Since these variables are all static, they have been initialized prior to `main`'s being called (typically at compile-time).

© 1996, 1998, 2009 Rex Jaeschke.
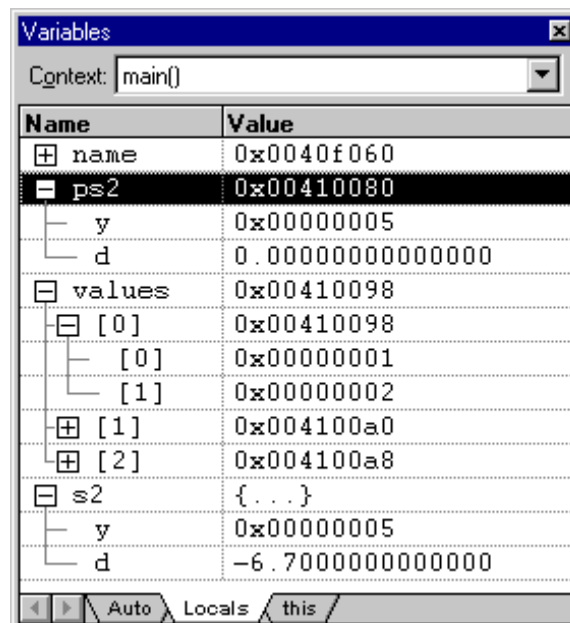
You can step through the program one statement at a time using the Step Into button.

- Press the Step Into button once and watch the debug cursor position at the definition of the first automatic variable, `i`. Since `i` has been allocated memory but has not yet been initialized, you can see its undefined initial value in the Variables pane.
- Step Into again and watch the value of `i` change to -1. Note that the value of `i` is displayed in red; this happens for each variable when its value changes.
- Step again and note that the value reverts to black since its value didn't change.
- Step through the whole program one statement at a time watching the Auto and/or Locals panes. When the debug cursor reaches a call to a library function such as `printf`, press Step Over to avoid single stepping through that function. (You can also step over any call to a user-written function.)

When you step into a `return` statement, the Auto pane shows the value returned by that function. Pressing Step Out causes the program to execute to the end of the current function and then return to its caller.

While single stepping through the code is interesting, it is also tedious, especially when you want to examine something well into the program.

- Press the Restart button.
- Left click on the source label `done`. You have set a temporary *breakpoint*} at that source line.
- Select Run to Cursor and the debugger executes the program up to the selected point. You can verify this by holding the cursor over the name `i` or by looking at the Variables pane; `i` should have the value 4.
- Ordinarily, the Variables} pane shows only those variables in the current function. Right-click in this pane and make sure that Toolbar is checked; this results in a pull-down menu appearing at the top, labeled "Context:".
- Single step into `f1` and then into `f3`. Now the Context menu shows the hierarchy of active functions.
- Select `f1` or `main` to see the current state of their variables. When doing so you will see a green triangular cursor; this indicates the place at which you went down the current execution path.

Since a source line might contain multiple steps, setting a breakpoint at the line can be rather coarse; you might like to set one at a particular part of that line. You can do this by displaying the machine code instead using the Disassembly button. Of course, this is also useful if you step into a function written in assembly language. (If you step into a library function such as `printf`, you likely will automatically have this mode enabled if the C/C++ source for that function is unavailable.) Now you can single-step through individual machine instructions or run to the cursor wherever it is positioned, as before.

- Press Restart.
- Set the cursor to the call to `f0` inside the `for` loop.
- Run to the cursor.
- Single step until `i` has the value 3.
- In the Variables pane left-click on the Value column for `i` until its value is highlighted.
- Change `i`'s value to -1 and continue single-stepping; notice how the loop keeps on going from that new value. You can modify the value of any non-`const` object in this way.
- Right click on any identifier in the Variables pane and select Properties to see the properties of that object. By pressing in the pushpin, you can keep the display up while you select other identifiers. Try it.