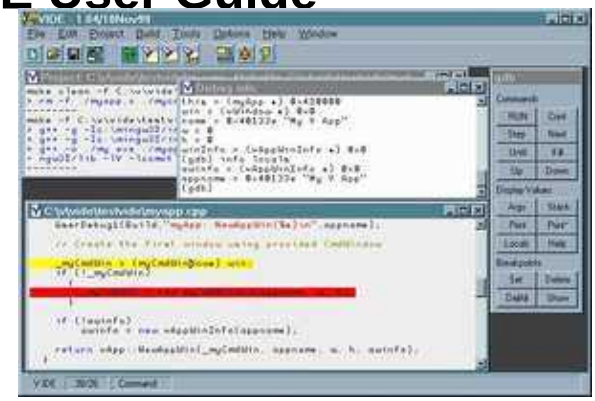
VIDE

Table of Contents

V IDE User Guide



The V Integrated Development Environment

VIDE is the **V** Integrated Development Environment for GNU gcc (Gnu compiler collection), the free Borland C++ Compiler 5.5 for MS−Windows, and the standard Sun Java Development Kit. VIDE is available both as a ready to run package for MS−Windows 9x/NT and Linux, and as part of the **V C++ GUI Framework**. Executables for MS−Windows 9x/NT and Linux (glibc) are available for download at http://www.objectcentral.com/vide.htm.

VIDE has been designed by a programmer for programmers. It makes the task of developing software for C/C++, Java, and HTML much easier than using command line mode. It is easy to learn, so it is a good tool for the beginner. It also has the critical features needed to enhance the productivity of the experienced programmer.

The source code is available under the GNU Public License (GPL), and many parts of its design reflect the philosophy of GNU and Open Source. Whenever possible, VIDE takes advantage of existing GPL or freely available software. It is designed to use the GNU gcc compiler and the free Sun Java kit. It also uses the GPL ctags program, and the addition of more integrated support for other GNU tools is planned.

While VIDE doesn't have every feature found in many commercial development systems, it is an ongoing project, with more features included in each release. And best of all, VIDE is *free*! And since it is GPLed, you can help add even more features if you want.

The main features in the current release of VIDE include:

- **A great editor** − The VIDE editor is a very good editor designed for the programmer. Editor features include:
  - ♦ Syntax Highlighting for C/C++, Java, Perl, Fortran, TeX and HTML.
  - ♦ Several menu−selectable editor command sets, including:
    - ◊ A **generic** modeless command set, similar to many Windows editors.

    - ◊ ***Vi*** − the standard Unix editor, with extensions.
    - ◊ The **See** editor command set, an editor designed and used by Bruce Wampler, the author of **V** and VIDE.
    - ◊ Others easily added by extending a C++ class.
  - ♦ Beautifies C/C++ and Java code
  - ♦ Powerful command macro capability
  - ♦ Complete support for ctags (symbol lookup)

The Editor Reference has a short section of tips that will help you get the most out of the editor.

- **Project Files** − specify source files, compiler options, and other details required for g++ or Java. Project files simplify and hide most of the details of using the underlying tools.

- **gcc and Sun JDK** − Supports development of both C/C++ with the **GNU gcc/g++** compiler for MS−Windows and Linux, (OS/2 environment soon), as well as the Java development using the **Sun JDK**.

- **Borland C++ Compiler 5.5** − VIDE has very good support for the recently released free Borland command line compiler tools. It can build Console, GUI, and static library project files. I also includes some extra documentation about the Borland environment. VIDE does *not* add a debugger, however.

- **Building Projects** − Uses standard GNU *make* to build projects for g++, and the standard features of the JDK to build Java projects.

- **Syntax errors** − Point and click to go to errors in source files.

- **Supports gdb and jdb** − Integrated support for the GNU *gdb* debugger for C/C++ and Sun's *jdb* debugger for Java. The most common debugging tasks, such as stepping through a program, are fully integrated, yet all the more advanced features of *gdb* and *jdb* are available through a command line window.

- **V GUI** − Integrated support for the **V** GUI for C++, including the **V** app generator and the **V** icon editor.

- **HTML** − Extra support for HTML development. While VIDE doesn't support WYSIWYG HTML development, you can send the current HTML file to your browser for immediate viewing. A comprehensive HTML help document is included. Future versions will include more HTML features such as table generation and image sizing.

- **The VIDE Help System** − Includes extensive HTML based help. Covers VIDE, GNU utilities, C/C++ libraries, HTML, and more. The help files are available for separate download. You can see a complete online version of the help package **here**.

- **Future enhancements** − VIDE is under active development and will continue to improve. Additions planned include an interface to *grep*, CVS version control support, spelling checking, C++ class browser, gcc profiler interface, and other features as supplied or requested from the VIDE user community.

The executable version of VIDE is totally freeware. Use it, share it, do whatever you want. The source of

VIDE falls under the GNU General Public License, and is normally included with the **V** GUI distribution. See the file COPYING included with the distribution for more information. Its development is not always in phase with the current **V** distribution, so there will be additional releases of executable versions as they become available. With the added support for Java, it is likely that the standalone executable version will see broader use than the source version included with V.

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

Features that are planned for the near future include:

- Support for more editor command sets, including Emacs.
- Integrated spelling checking.
- Support for CVS/RCS.
- Code templates.
- Predefined Project files for various configurations − GUI, console, Mingw32, Cygnus, etc.

This document describes how to use VIDE. Because the process is slightly different for C/C++ and Java, there is a brief tutorial section for each language. Following the tutorials, all the commands available from the menus are described.

VIDE Overview

The design of VIDE has been somewhat evolutionary, but you should find that it is not that much different than other IDEs you may have used. Because VIDE is a free, open source program, it probably lacks some of the polish of commercial IDEs. However, it is still quite functional, and it is really easier to develop programs with it than it is to use a command line interface.

Generally, any application you write will consist of various source files (with associated header files for C/C++), and required data files. These files are generally dependent on each other. By defining a VIDE Project for your application, all the file dependencies are automatically handled. The standard tool *make* is used for C/C++ files, while the JDK Java compiler automatically handles dependencies.

Using VIDE, the normal work cycle goes:

1. Design your application.
   VIDE currently has no capabilities to help with this stage.
2. Start VIDE, and create a Project File.
   This will include all source files, compiler options, and other information needed to compile your application.
3. Build your project.
   This stage compiles your source into object code. Compilation errors are displayed in the status window, and you can simply right−click on the error to go to the offending line in your source code. After making corrections, you repeat this step until all compilation and linking errors are removed.
4. Run your program.
   You can start your program from within VIDE.
5. Debug your program.
   VIDE for MS−Windows has integrated support for the **gdb** debugger for GNU C/C++. Because the *DDD* debugger available for Linux is so good, VIDE for Linux does not have integrated debugging

support, but will automatically launch DDD. There is no support for debugging Borland BCC32 programs.

6. Write documentation for your application.
   VIDE has syntax highlighting for HTML to make that job easier. You can also automatically launch your web browser to view the resulting HTML pages. Really neat.

VIDE Help System

VIDE is distributed with this complete VIDE documentation. A complete set of HTML documents with useful help topics are available at www.objectcentral.com. VIDE also knows about the documentation that comes with the Sun Java distribution. All this help is easily available from the VIDE Help menu. The vide help system is described in the VIDE command reference.

Debugging with VIDE

VIDE supports GNU **gdb** and the Sun JDK **jdb** debuggers. The VIDE interface to the debuggers has been designed to make the most common debugging tasks easy. The goal is to make using the native debuggers as easy as possible for casual users, while maintaining the full power of the debugger for experienced users. VIDE accomplishes this by showing a command window interface to the debugger. You can enter any native debugger command in this window, and thus have full access to all debugger features.

VIDE makes using the debugger easier by providing an easy to use dialog with the most often used commands. Breakpoints are highlighted in yellow. And as you debug, VIDE will open the source file in an editor window and highlight in red the current execution line on breakpoints or steps. It is very easy to trace program execution by setting breakpoints, and clicking on the *Step* or *Next* dialog buttons. VIDE also allows you to inspect variable values by highlighting the variable in the source and clicking the print button.

VIDE for MS−Windows

VIDE for Windows is distributed as a self−installing executable file. You might want to create a desktop icon to start VIDE. As of 1.08, VIDE supports drag and drop to edit files. If you want any files with particular extensions to automatically start in VIDE, use the **Start−>>Settings−>Folder Options−>File Types** dialog to associate the file type with VIDE.

VIDE for Linux

The Linux version of VIDE is distributed as a statically linked binary version for recent versions of Linux. It is based on the Motif version of V, and is statically linked to the MetroLink Motif library. To install, unzip and untar the distribution. You can install the binary almost anywhere you want.

Because this version is based on Motif, it isn't as well behaved as it might be with the KDE or Gnome windows managers. It will start with the default Motif blue color scheme. It is easy to get a better looking color scheme by using the `−bg` startup switch. This changes all the Motif decorations to be based on the color

you specify. For example, starting VIDE with `VIDE -bg gray75` gives a nice gray based color scheme. You can make a desktop shortcut with either KDE or Gnome that will automatically use the −bg switch.

Release Notes

- **Version 1.08 − 04Mar2000**
  Version 1.08 is an important release. It corrects some bugs in the Borland BCC32 5.5 interface. It adds drag and drop for the MS−Windows version. But it also finally supports the Motif Linux version. This is the first binary release in some time for a Linux version. It is also important because it corresponds to the finally released V GUI Version 1.24, which contains the source for VIDE 1.08.
- **Version 1.07 − 25Feb2000**
  Version 1.07 is a major upgrade. The main new additon is support for the new free version of the Borland C++ Compiler 5.5. In the process of adding this support, some features were improved.

    - *BCC32 support* − fully integrated to VIDE. A separate reference document for the Borland support has been added.
    - *Easier resource files* − It is now easier to include `.rc` resource files on Windows.
    - *Improved Project Editor* − The project editor is a bit more intelligent about rebuilding the Makefile now, and won't automatically regenerate it as often.

- **Version 1.06 − 8Feb2000**
  Version 1.06 has several significant new features added. Gee, this thing is starting to get real!

    - *ctags* support − you can easily find the declaration or definition of any program symbol.
    - *C++ Project Wizard* − When you create a new C++ project, you can easily specify options such as project type (console, GUI, library), compiler options, and more.
    - Significant upgrades of the documentation, including a new editor tips section.
    - Syntax highlighting added for Fortran. There is no project support for Fortran, however.
    - Syntax highlighing for LaTeX files. Additional support for LaTeX is likely in the future.

- **Version 1.05 − 21Jan2000**

    - Autoindent for code files has been added. This option is found on the Options−>Editor dialog.
    - Code beautifier now supports KRbrace placement in addition to braces on separate lines.
    - Syntax highlighting added for Perl. There is no project support for Perl, however.
    - Screen updating for Windows version significantly faster.
    - A few bugs have been fixed.

- **Version 1.04 − 18Nov1999**
  Version 1.04 includes major enhancements to the debugger interfaces. Debug commands have been removed from the tool bar, and now are in a pop up dialog when you run the debugger. The biggest improvement is that breakpoints are now highlighted in yellow in the source file windows. The current execution line is shown in red. The Debug menu has been removed.
  With the release of gdb 4.18, programs developed with the V GUI can now be debugged reliably. With the release of gcc 2.95 for mingw32, and the new Cygnus Version 1.0, all the problems associated with earlier releases seem to have been resolved. You should be using the latest versions of gcc!

- **Version 1.03 − 25Oct1999**
  This version has some minor enhancements with the gdb and jdb interfaces. The documentation HTML has been changed to use the default browser background colors.

- **Version 1.02 − 11Oct1999**
  This version includes new support for different syntax highlighting color schemes and different background colors. There are six choices under the Options:Editor menu dialog.

Installing VIDE

It is probably easiest to use a pre−compiled version of VIDE. The VIDE executable binary is a complete static program. It does not use any external files or DLLs. You can place the binary where ever you want. However, you might want to install VIDE to its own directory tree. This is most likely to be compatible with later versions of VIDE. For now, the tree just includes /vide/bin and /vide/help. You can run VIDE from the desktop without setting a PATH, but if you use V and want to start the Appgen and Icon editor from within VIDE, then those programs must be on your PATH. If you are using mostly GNU gcc/g++, then you may want to install the VIDE executables on the same /bin directory the compiler is on. If you are using Java, then you can create a /vide directory for the VIDE system. You may want to add a desktop shortcut if you use it a lot.

If you want to use the help files other than VIDE help, you will need to download the separate help file archive. It is best to install the help files in the /vide/help directory. After you've installed the help files, you will need to use the VIDE Options:Editor command to set the path to the help files.

VIDE doe not provide Java help. You will need to find and download the Sun JDK yourself. Then use the VIDE Options−>Editor menu command to set the Java directory.

If you are building VIDE from the V distribution, then apply the above comments to the version you build. If you want to use the V appgen program, or the V Icon Editor, you need to get those from the V distribution.

Known VIDE problems

Errors generated by the compiler are passed via two temporary files. Sometimes these files are not deleted, and are left behind on the hard disk. If two instances of VIDE are running, it is possible to have an access conflict to these files from one of the instances. This can prevent builds with error messages from displaying those messages in the message window.

If g++ has a fatal error, it can hang VIDE. Because all files are saved right before running g++, there is no data loss.

The syntax highlighting doesn't work right for multi−line "/* comment */" style comments. If the lines between the opening "/*" and the closing "*/" have a "*" as the first character (which is the normal convention for Java programs!), syntax highlighting works correctly. Otherwise, those lines will not get the comment highlight. This is purely a cosmetic problem, but is unlikely to be fixed because of difficulties imposed by the internal structure used by the editor. (Note (3Sep99): Gee, I feel better! I was just playing with Microsoft VC++ 6.0, and guess what? Their editor doesn't do /* */ comments right, either! I guess if Microsoft can sell a commercial product that can't do proper syntax highlighting, then I can do it for a free

product.)

The tidy/prettyprint command doesn't handle the last case of a switch properly. This won't be fixed. Also note that the formatting is based on the indent of the previous line, so you must start at a known good indent point.

Help make VIDE better!

The V IDE is GPL freeware. It has been written using the V C++ GUI framework. I get no compensation for either V or VIDE, so please don't get too fussy about features or problems. I welcome bug reports and requests for new features, but I make no promises.

My goal for VIDE is to make it a great alternative to Emacs. I know Emacs will do almost anything you would ever want, but the learning curve is huge. VIDE is much more GUI oriented, and I want to keep it simple enough for beginning programmers to use.

So, even more welcome than bug reports would be offers to help add features to VIDE! Since the source code for both V and VIDE are GPL and LGPL, they are available for enhancement. If you would like to make contributions to VIDE, please contact me directly by e−mail. VIDE is currently ahead of the V GUI release cycle, so I will need to provide you with the latest versions of all the source code.

I welcome any contributions or ideas, but right now the following projects:

- Support for other editor command sets. My goal is not to provide exact clones of other editors, but to provide the basic commands of other editors at a "finger memory" level. It is fairly easy to add a new editor command interpreter. I spent less than two days writing the Vi emulation. Because I use the See command set myself, it is a bit hard for me to write a command interpreter for other editors that provide true "finger memory" compatibility. I hope the Vi emulation meets this kind of compatibility. I'd like to see an emulation for Emacs, but any emulation is welcome. I think the emulation will be better if it is written by someone who actually uses that editor.

- Spelling checking.

- An interface to CVS.

- An interface to grep.

- A class browser for both C++ and Java. I have a stand alone V C++ browser that could serve as a starting point.

- An interface to other Java tools such as Javadoc.

- An interface to Jikes.

If you'd like to help on any of these projects, please contact me and I will do everything I can to help you get started.

No Warranty

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

V IDE Reference Manual − Version 1.08 − 04Mar2000

Bruce E. Wampler, Ph.D.
bruce@objectcentral.com
www.objectcentral.com

VIDE − Command Reference

This section provides a summary of the all the VIDE commands available from the menus.

File Menu top

The File menu is used for source files. Use the Project menu to open and edit project files.

File:New 

Create a new source file. Syntax highlighting doesn't take effect until you've done a **File:Save as** and repainted the screen.

File:Open... 

Open an existing file.

File:View...

Open an existing file for viewing. The file is read−only, and you won't be able to make any changes.

File:Save 

Save the current file to disk.

File:Save as...

Save the current file using a new name you will specify.

File:Close

This will close the existing file. If you've made any changes, you will be prompted if you want to save them.

File:Save All

Saves all currently open files.

File:Save/Close

Save the current file, then close it.

File:Send to Browser

Save the current file, then open it with default browser. This command has a quick and dirty implementation, and it passes just the default file name to the system routine that opens the browser. Thus, this command can fail if the file doesn't have a full path qualification. This can happen when you type a file name in directly to the file open dialog. On MS−Windows, you must have .htm and .html files associated properly with your browser. This command won't do anything for non−HTML files. This association will usually be set automatically when you install your browser.

File:Exit

Exit from VIDE.

Edit Menu top

The edit menu has some basic commands to edit text in the current file.

Edit:Undo 

Restores the last text deleted. Only one level. Doesn't undo insertions or position changes. Also, doesn't undo deletions greater than 8K characters.

Edit:Cut

Delete the highlighted text, and copy it to the clipboard. Standard GUI operation − use mouse to highlight region of text, then cut

Edit:Copy

Copy highlighted text to clipboard.

Edit:Paste

Paste the text on the clipboard to the current text position.

Edit:Find... 

Find a text pattern in the file. Brings up a dialog.

Edit:Find Next 

Find the next occurrence of the current pattern in the file.

Edit:Replace... 

Find a pattern in the file, and replace it with a new one. Brings up a dialog.

Edit:Find Matching Paren

If the cursor is over a paren character, i.e., ()[]{}, the cursor will be moved to the matching opposite paren.

Edit:Set BP

This command will set (or preset if the debugger isn't running) a breakpoint on the current line. Breakpoints will be highlighted in yellow. Usually, you set breakpoints after you run the debugger, but VIDE remembers breakpoints across debugger sessions. (However, VIDE does not remember breakpoints across VIDE sessions!)

Edit:Delete BP

This command will delete the breakpoint on the current line.

Edit:Edit Help

Displays a list of the command supported by the current editor command set.

Project Menu top

Project:Open 

Open an existing project file. Project files all have a `.vpj` extension. VIDE automatically detects C/C++ or Java projects.

Project:New C++ Project

This will create a new C/C++ project. The options on the dialog are described in the C/C++ section of this documentation.

Project:New Java Project

This will create a new Java project. The submenu allows you to create a new Applet, Windowed App, or a Console App. The details are described in the Java section of this document.

Project:Edit

Edit the currently open project. See C/C++ or Java sections for details.

Project:Close

Save and close the currently open project.

Project:Save as...

Save the current project under a new name. This is useful for creating "template" projects that have specific settings for your development environment. These templates can be opened later, then saved under a new name again for the real project.

Project:Rebuild Makefile

OK, I admit it. VIDE doesn't handle all cases of changes to your files. If you add a new `#include` to a source file, for example, VIDE won't automatically rebuild the Makefile to add this new dependency. This command helps get around that problem.

Project:Select Makefile or Java file

Instead of using a VIDE project, you can simply use an existing Makefile, or even a Java source file. Use this menu item to specify the Makefile or Java source file instead of a VIDE project. When you have a Makefile or Java source file selected, the Makefile will be run, or the Java source passed to the Java compiler when you click the make tool bar button.

Build Menu top

Used to build and compile projects.

Build:Make C++/Compile Java 

Build the project. This command first saves all your open files. It then runs `make` for C/C++ projects, or the Java compiler for Java projects. Errors are displayed in the message window, and you can go directly to the error by right−clicking on the error line in the message window.

Build:Make clean

Build:Make cleanall

Build:Make cleanbin

Runs `make` with the given target: clean to clean object files, cleanall to clean objects and binaries, and cleanbin to clean binaries only. Used only for C/C++ projects.

Build:Make <Target>

Runs `make` to make the target you specify. Used only for C/C++ projects.

Build:Generate ctags

Use this command to generate or regenerate the ctags file for the current directory. See editor ctags for more information.

Build:Stop Make

For C/C++ makes, will stop the make after the current file is finished being compiled.

Tools Menu top

Tools:Run program w/ args

Runs a specified program. Allows you to specify arguments to the program.

Tools:Run project 

Runs the program from the existing project. Note: VIDE does not check to recompile before running an object.

Tools:Start Debugger 

Opens a new command window to interface to the debugger. Will use the current executable file.

Tools:Run OS Shell

Runs a basic OS shell.

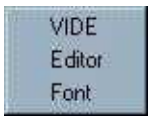
Tools:V App Gen

Runs the V tool V App Gen.

Tools:V Icon Editor

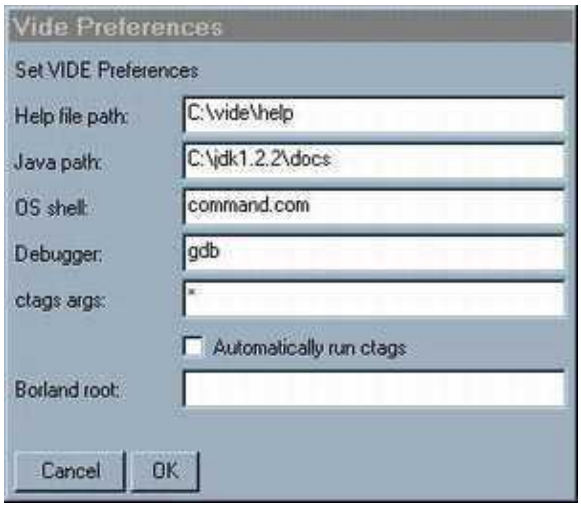
Runs the V Icon Editor.

Options Menu top



The Options menu allows you to customize various aspects of VIDE, including paths, editor attributes, and font. These settings are saved in a standard system place. For example, they are saved in `C:/windows/vide.ini` on MS−Windows. They will be in `$(HOME)/.Viderc` on Linux or other Unix−like systems.

Options:VIDE



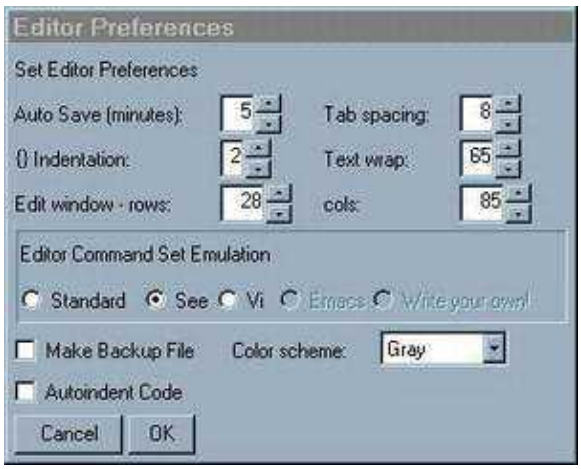
This item allows you to set the paths VIDE uses to find the VIDE help system files as well as the standard help file included with the Sun JDK.

This item also allows you to specify which command shell and debugger are used. These two options are most useful on Linux/Unix systems.

You can also set the default args used when running ctags. See editor ctags. If you set the "Automatically run ctags" box, then VIDE will automatically generate a new ctags file whenever you open a project. Note that running ctags is a very fast operation.

The Windows version also includes a line to specify the "Borland root". This is used for support of the Borland C++ Compiler 5.5. See the Borland reference.

Options:Editor



The options include:

- **AutoSave** − VIDE will automatically save changed versions of edited files every N minutes. Use 0 for no autosave. Note: while autosave might save a lot of lost work for you sometime, it can have problems, too. Unless you check the "Make Backup File" option, VIDE only keeps the current copy of your file. After you have saved the file, either by a manual save, or by autosave, previous versions

of your file are lost. If you quit the editor, and answer no to saving changes, you won't necessarily get back to your original file if you have autosave on.

- **Tab Spacing** − the number of spaces for each TAB character. The most widely used value is 8.
- **{} Indentation** − This applies to beautifying Java and C++ code. With a value of 0, braces { and } will be lined up with the outer indentation level. With a value of 2, braces are indented 2 from the outer level. It is most common to line up with the outer level (value 0) in code that lines up braces, but I find the extra 2 spaces makes the braces stand out better, and is easier to read. This should be 0 for KRstyle code.
- **Text wrap** − If you are editing text or HTML files, each editor emulation will provide a command that will automatically fill text. The text wrap value is the column used to determine the wrapping.
- **Editor emulation** − Select which editor command set you want to use.

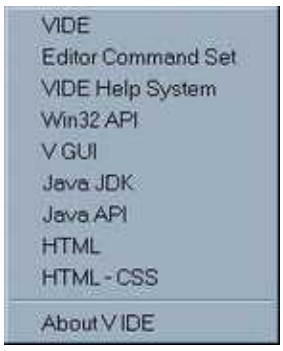
    - Standard Editor
    - See
    - Vi

  The editor emulation changes for new edit windows you open. The editor you chose will be saved in the preferences file.
- **Make backup file** − if this is checked, VIDE will make a backup version of the original file. The backup will retain the original name and extension, but add a ".bak" to the name. For example, "foo.cpp" will be saved as "foo.cpp.bak". This feature can be handy when used in conjunction with AutoSave to be sure there is an unchanged copy of the original file.
- **Color scheme** − By popular request, VIDE now includes several color schemes for the editor windows. The standard version is black letters on a white background, with various other colors used for syntax highlighting. The color scheme option lets you pick from several other color schemes. You can only pick from the fixed schemes. If you have a color combination you'd really like to see, send e−mail and I will consider adding it for the next version. When you select a color scheme, it is not applied until you open a new editor window.
- **Autoindent Code** − When this is checked, VIDE will autoindent when you are entering new code. When you enter a newline, VIDE will automatically insert the same indentation as the previous line (spaces and tabs). This will be done only for code files (C++, Perl, Java).

Options:Font

Lets you specify the font used in the display window. The font will change in the current window, and in future windows, but not in already open windows. The font you chose will be saved in the preferences file.

Help Menu top



Help:VIDE

Opens your browser with this file.

Help:Editor Command Set

Shows a dialog box with a command summary of the editor command set currently being used.

Help:GNU, Other Tools

Opens your browser with the VIDE Help System page. This contains links to various GNU software, g++, libraries, and HTML. This is the page distributed as the VIDE Help Package.

Help:WIN32 API

This will try to open the Borland WIN32 API .hlp file. The hlp format file should be available at Windows API Reference from Borland.

Help:V GUI

Opens the V GUI Reference Manual.

Help:Java JDK

Opens the top level Sun JDK Help pages. You must set the Java help path in the Options:Editor menu, and download the help files from Sun.

Help:Java API

Opens the Sun JDK API Help pages. These cover all the standard elements and library classes of Java, and is probably the reference you will most often use. You must set the Java help path in the Options:Editor menu, and download the help files from Sun.

Help:HTML

Opens a guide to HTML tags.

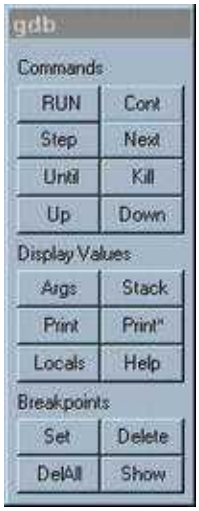
Help:HTML − CSS

Opens a guide to HTML Cascading Style Sheets.

Window Menu

Standard MS−Windows Window menu.

Debug Dialog top



gdb Dialog jdb Dialog

In general, the commands on the debug dialog work very similarly for the **gdb** and **jdb** debuggers. For more specifics for **gdb** and **jdb**, see the C++ Tutorial or the Java Tutorial.

Commands:RUN

Run the program being debugged from the start. You will usually want to set some breakpoints first.

Commands:Cont

Continue running program until the next breakpoint is reached.

Commands:Step

Step into the current statement. This will break at the first statement of a called function. For non−function calls, this will have the same effect as the Next command. When you use Step or Next, the new current program line will be highlighted in red.

Commands:Next

Step over the current statement. If the statement is a function call, the program will break after the function has returned.

Commands:Until

Continue running the program until the current line in the editor window is reached. (gdb only)

Commands:Kill

Stop execution of the running program. (gdb only)

Commands:Up

Move execution up stack frame.

Commands:Down

Move execution down stack frame.

Commands:StepUp

Execute until the current method returns to its caller. (jdb only)

Display Values:Args

Display args to current function. (gdb only)

Display Values:Stack

Display the call stack.

Display Values:Print

Display the value of the variable highlighted in the editor window. The variable must be available in the current context of the running program. To use this command, use the mouse to highlight the variable name you want to inspect. It is easiest to double click over the symbol to highlight it. Then click this command in the dialog.

Display Values:Print*

Line Print, but does indirection. Useful for C/C++ pointers. (gdb only)

Display Values:Locals

Print all local variables in current stack frame.

Display Values:Class

List currently known classes. (jdb only)

Display Values:Memory

Report memory usage. (jdb only)

Breakpoints:Set

Set a breakpoint at the current line in the editor window. To use this (and other commands that use the "current line"), first get focus to the editor window of the source file you want to work with. Then go to the line you want to work with, either with the mouse or cursor movement commands. Finally, click the *Set* button in the dialog box (or the Edit:Set DB menu). Breakpoints will be highlighted in yellow. Usually, you set breakpoints after you run the debugger, but VIDE remembers breakpoints across debugger sessions. (However, VIDE does not remember breakpoints across VIDE sessions!) When you hit a breakpoint, the current program line will be highlighted in red.

Breakpoints:Delete

Delete the breakpoint set at the current line. This command isn't as easy to use as it could be because the editor doesn't highlight lines with breakpoints. The current version doesn't keep its own list of breakpoints, but relies on the debugger. Thus, you have to know in advance which line you want to debug. It is sometimes easier to just use the debugger command line interface for this. I hope this one gets better in the future.

Breakpoints:DelAll

Delete all set breakpoints.

Breakpoints:Show

Show all set breakpoints. Uses native debugger commands for this.

(Breakpoints,Display Values):Help

Show debugger help. Uses debugger native help command.

No Warranty top

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

Bruce E. Wampler, Ph.D.
bruce@objectcentral.com
www.objectcentral.com

VIDE − Editor Reference

The VIDE Editor

What is the most important part of any IDE? Why, the editor, of course! No matter how easy an IDE makes it to compile and debug your programs, you will still spend most of your time in the editor. The VIDE editor is based on the V TextEditor class, and has a long history.

I wrote the first version of the editor over twenty years ago in a language called Ratfor. Over the years, this editor code has evolved. It went from Ratfor to C to its current C++ version, and from supporting a simple console terminal to the current GUI version in an IDE. Over all this time, I always have found it easier to port the editor rather than learn a new one. This editor has the features that I've found the most useful as a programmer. For most of its life, the editor supported my own command interface called See, but the current version supports a standard GUI interface, as well as a Vi command emulation. No matter which command interface you use, the VIDE editor has a long history and includes many features very useful to a programmer.

Getting Started − Tips top

As I just noted, the VIDE editor has evolved into a fully featured programmer's editor. But in order for you to get the full value of these features, you have to know about them. This section will help you get started using the VIDE editor. It contains some tips for getting the most out of the VIDE editor −− understanding both its

features, and its quirks. Do yourself a favor, and read this section!

- **Everybody Does It!** − General Editing

  Of course the editor supports all the standard things you usually do with an editor − moving the cursor, adding text, changing text, finding text, and so on. Just how you do these actions depends on the editor command emulation you are using. (See next section.) VIDE has a very extensive help system, but there is an extra menu command that you will likely find useful as you are first learning the editor. Using the **Edit−>Editor Help** menu command will bring up a dialog box with a list of the commands supported by the current editor emulation.

- **Do It Your Way!** − Emulations

  The current version of VIDE supports several command sets. The default is a generic modeless command set typical of most GUI editors. The second command set is based on the See editor which is the editor I've been using for 20 years. It is not well known − yet. There is also a very good emulation for Vi that will keep your fingers happy if you use Vi. It is possible to add a new emulation with just several hours of work. Adding an emacs emulation has been on my list for a long time, but I've yet to receive even one request to do it. The main idea of any emulation is to support "finger memory" − allowing your fingers to do what they have learned as you've used your favorite editor. Thus, I hope I've succeeded making the generic GUI editor like most other GUI based editors, and having the Vi emulation work as you would expect. You can select the emulation from the Options−>Editor dialog.

  Some features of the editor are available mainly through menus and mouse actions. For example, all emulations use the mouse to access the ctags features. Some of the searching and replace features are available only through the Edit menu.

- **Do It To What?** − Selecting text

  GUI applications all allow you to highlight text for some kind of action − copy to the clipboard, delete, and so on. VIDE supports standard mouse based selection. You can also use Shift+<arrow−keys> to highlight selections. Because of limitations in the V library, you can use the mouse only to select currently showing text. You *can* use the Shift+:<arrow−keys> to select text off the display window.

  You can left−click to select text. The first click positions the cursor. The second click will select the word (a−z, 0−9, and _) under the cursor. A third click will try to select a complete programming symbol (although the selection algorithm still needs some improvement). A fourth click will select the entire line. This selection method is very useful for looking up ctags and setting breakpoints.

- **Where Is It?** − Finding text and symbols

  Of course, one of the main things an editor lets you do is find text within a file by using a *find* or *search* command. The VIDE editor supports this, but it also lets you easily find the definition of a program symbol using another program call *ctags*.

  - ♦ *Searching*
    The VIDE editor supports finding (and replacing) text two ways. There are three Edit menu options: Find, Find Next, and Replace. The menu forms use dialogs to enter the patterns.

Each editor will also have keyboard commands that let you enter the patterns without a dialog. The patterns are echoed on the status line. VIDE does not support regular expressions in its find patterns.

- *ctags*
  The *ctags* program has been used for years on Unix systems. Essentially, ctags builds an extensive cross−reference of the symbols (variables, function names, defines, etc.) in your program files. The VIDE editor uses the ctags file to let you find the original declaration of any symbol in your source file. It is simple to use. Highlight the symbol you want to look up (double left−clicking over the symbol is the easiest way to do this). Then right click on the highlit symbol. The VIDE status window will then show locations where the symbol has been defined or declared. If you then want to see the actual definition, right click on the line in the status window, and the file will be opened right on that line. Neat!

- **Move It!** − Moving Text

  Moving big blocks of text around is a common editing operation. The standard GUI way to do this is to cut and paste the selection. The See and Vi emulations also support other very powerful ways to move blocks of text, including reading and writing files. See the See "Save buffer commands" and the Vi "Yank buffer commands" for more details. Currently, the standard GUI editor does not support this concept.

- **Make It Pretty!** − Looking good

  Having nice looking code is a *very* important part of the programming process. A well formatted, properly indented program makes it far easier to read, understand, and maintain. The VIDE has several features that make it easier for you to write good looking code. It even has some features to let you write documentation − in plain old text or HTML.

  - *Auto−indent*
    If you are editing code files, the VIDE editor has an auto−indent mode (enabled in the **Options−>Editor** dialog). When auto−indent is on, the editor will automatically indent the same number of tabs and spaces on the previous line whenever you insert a new line. If you need to unindent, simply press the backspace key.

  - *Beautifier*
    I personally don't like auto−indent that much. Instead, I prefer to use the VIDE beautifier. This feature will automatically beautify (set the proper indentation) source code, and fill text. The neat thing is that you can do this any time, and it will clean up your code even after you've done extensive editing. A beautifier may not sound really important, but over the years, I've found that it is one of the most important features of the editor, and I would find it difficult to program without it. There are more details in theCode Beautifier section.

  - *Syntax highlighting*
    The VIDE editor knows how to highlight the syntax of C, C++, Java, Perl, and HTML. Syntax highlighting makes it much easier to read your code. There are several standard color choices available in the **Options−>Editor** dialog.

  - *HTML support*
    VIDE is not a full blown HTML environment, but it does have a few features that help. Is use it for most of my HTML editing. As a programmer, I find it pretty easy to use the

standard HTML tags, and find that the editor I use (VIDE/See) is more important. In addition to HTML syntax highlighting, the **File−>Send To Browser** command is most useful. It will close the current file, and then use the default browser to open it. This is pretty effective. Once the current file has been loaded into the browser, it is often easier to click the save file tool bar button, and then the reload button in the browser. I've been doing that a lot as I write this, and it is almost as good as a full HTML editor.

- **Do It Again!** − Macros

  There are a lot of editing tasks that are repetitive. VIDE has a couple of features that makes these tasks easier. First, most editor commands all you to add a count at the beginning. Thus, you can enter a command that says go down 53 lines instead of pressing the down arrow 53 times. The mechanism for entering counts depends on the command syntax of the given emulation.

  Even more powerful than simple counts is the VIDE macro facility. Essentially, a macro is a small editing program that you can execute. You first define a macro consisting of editor commands needed to perform the task, and then execute the macro as many times as needed. See the macros section for more details.

- **I Did It Wrong!** − Undoing

  You mean you make mistakes when editing? VIDE has a few features to help you undo editing errors.

  - Undo
    VIDE has an undo command, but it is only one level, which works for the most common situations. Essentially, you can undo the last delete operation, which most of the time will be enough. You can't undo a long insertion (do it by hand!), or a series of single character deletes. VIDE's undo facility is one place where the 20 year heritage of the code limits it functionality − the internal editor code just doesn't lend itself to multiple undos and redos.
  - Autosave
    Probably the biggest safety feature of VIDE is autosave. You set the value in the **Options−>Editor** dialog. The current state of your file will be automatically saved at the frequency you specify. This is five minutes by default.
  - Backup file You can also use the **Options−>Editor** dialog to tell VIDE to make a backup copy of the original file when you open it. With this enabled, you will always have the original copy of the file to revert to in case of a major editing error.

VIDE features top

This section describes some of the features of the VIDE editor emulations that are especially useful for programming. While most of these features are found in other editors, some are not native to the original editors VIDE emulates. In those cases, VIDE tries to fit the features into the natural command syntax of the original editor.

VIDE Keyboard Macros top

The Standard, See, and Vi emulations all support a simple, yet powerful macro facility. There are 26 buffers called "Q−Registers". You can record character keystrokes into any of the Q−registers, and then execute those characters as a set of commands. A set of commands like this is often called a macro, or in VIDE terminology, Q−Macros. (Note: when you record special keys such as the arrow keys, they are all echoed as "{fn}" −− you won't be able to tell from the echo which key you entered.) You can enter any command into the editor, including find patterns and insertion strings.

The general procedure for using Q−Macros is the same on the currently supported emulations (the commands for using Q−Macros are different). First, you record a sequence of keystrokes into one of the 26 Q−Registers. The keystrokes you enter will be echoed on the status line. You can use backspace to edit your input. You terminate the Q−Macro by entering the special end−macro character defined for the emulation.

Once you have a macro recorded, you can then execute it using the execute−macro command of the emulation. You can provide a count, and thus execute the macro many times. Q−Macros terminate when any given command fails. For example, if a find fails, the macro will end. There are commands to specify which Q−Macro to execute.

Code Beautifier top

The VIDE editor has a code beautifier for C, C++, and Java. How the beautifier works for Perl has not been fully tested. The editor will also fill text lines for text files.

To use the beautifier, you place the cursor on the line after a line that is already properly indented. Then enter the beautify command (as defined by the given emulation), and your code will be automatically indented, subject to a few limitation described later. Some parameters of the filling and beautification process can be adjusted in the **Options−>Editor** dialog.

Code Beautifier

VIDE will format your C, C++, and Java source code by following a fairly simple set of rules. Because VIDE doesn't fully parse the code, you will have to follow some coding conventions, and may have to "help" manually sometimes. On the whole, however, VIDE's beautify command works better than using auto−indent. It is especially useful after you've revised a section of code.

- Indentation is based on steps of *four* spaces. Following the normal Unix convention of 8 spaces per tab, your code will be indented to an even multiple of 4 spaces, even tab stops, or tab stops plus 4 spaces.
- The standard language keywords plus curly braces ({}) are used to determine indentation.
- Lines with non−whitespace in the first column are left as they are.
- Formatting is based on the indentation of the previous non−blank line. Thus, when you beautify your code, you must start at a place where the indentation is already correct.
- Because VIDE doesn't completely parse the source code, it doesn't always correctly handle `case` and `default` statements. You can usually work around this by manually formatting the first line of a case group or default group.
- Beginning with version 1.05, the VIDE beautifier can automatically handle two coding conventions. The preferred style assumes you have braces ({}) on lines by themselves, perhaps with a trailing

comment. VIDE also tries formatting source code that uses the convention of placing the opening brace ({) at the end of a control statement such as `if` or `while` (sometimes known as KRstyle). Support for KRstyle is not quite as robust as keeping braces on separate lines, but it is still good. Having a pair of braces on the same line can confuse VIDE's formatting.

Hint: when you are entering new code, it is often easier to not worry too much about the indentation. Just leave some whitespace at the beginning of each line, then go back and beautify it after you have your statement structure complete.

Hint: Using a command count often is especially useful when beautifying blocks of code. Remember to start on a line that is indented the way you want already.

Text Filling

When you use the beautify command on text files, VIDE will fill the text to the column set in the **Options:Editor** dialog. If the first column of the text has certain special characters or character sequences, VIDE will skip filling that line. This feature is intended to make filling of HTML and other markup language files work better. VIDE won't fill if the line is blank, or begins with a space, a period, a tab, a latex keyword, or a block oriented HTML command.

Syntax Highlighting top

C, C++, Java, Perl

VIDE editors will highlight C, C++, Perl, and Java source code. The following default conventions are used: keywords in blue; constants in red; comments in green; C/C++ preprocessor directives in cyan; remainder in black. Other colors are used for alternate color schemes.

HTML

Highlighting of HTML files is very simple minded, but can really help you to read the HTML source code. Angle brackets (<, >) are always highlighted. If the first character sequence after the opening < is a valid HTML command, then it is also highlighted. Other parameter keywords within an HTML command are not highlighted. String constants and numbers are also highlighted. The ""character is also highlighted.

ctags top

Beginning with version 1.06, VIDE supports the *ctags* program. Ctags will generate a cross−reference tag file of C++ and Java source files. VIDE can read this file and locate the original definition or declaration of a symbol.

To use the ctags feature, first use the **Build−>Generate ctags** command. This will generate a file called "tags" in the current directory. Then, to locate a symbol's definition, highlight it anywhere in a source file. (It is easiest to do this by double clicking over the symbol.) Any instances of that symbol will then be displayed in the status window. If the symbol is defined in multiple files, there will be multiple lines shown. There is

extra information supplied about the symbol, and you can usually tell which is the instance you want. Often the information shown in the status window will beenoughh to help. If you need to see the actual definition or declaration, then right click the appropriate line in the status window, and the file will be opened.

Sometimes ctags will not include the symbol in the tags file. It does not generate entries for symbols local to a function, or for function parameters. And it will not automatically include symbols from libraries you use. If you want library symbols included in the tags file, you need to include the path to the library on the ctags args line.

By default, the arguments supplied to ctags are "−n", which is required for VIDE to use the tags file properly, and "*", which will make ctags use all the source and header files in the current directory. Most of the time, this will be just what you need. However, ctags has many options, and can tag files from other directories given the proper options.

You can change the default "*" argument using the **Options−>VIDE** dialog. You can also supply a project specific ctags argument list using the project editor. Note that the "−n" switch will always be used.

To use the ctags feature, you normally don't have do anything other than be sure ctags is available. The Windows distribution of VIDE will install the ctags executable on the VIDE directory, and ctags is normally found on Linux systems. (You may have to add the VIDE directory to your AUTOEXEC.BAT PATH.) VIDE provides the ctags version known as Exuberant Ctags. A local *text* copy of the ctags man page is also included with VIDE. The source of Exuberant Ctags is available at its web site.

Other features top

Wide lines

One attribute of the VIDE editor is how it handles lines wider than the window. It does not use a horizontal scroll bar. Instead, as you move right on long lines, the text will automatically shift to show more of the line. The last character displayed in the window of a wide line will be the last character in the line, and not the character that actually is in that column.

Auto save

You can set the Auto save value in the **Options:Editor** dialog to tell VIDE to automatically save open files at a given interval. This approach is different than some programmer's editors. You can end up with files that are in a state of transition. VIDE also supports making a backup file of the original when you edit. Note that whenever you do a project build, open files are automatically saved.

Time stamp

If you put a comment containing the string "date:" anywhere in the first 12 lines of your source code file (C++, Java, Perl, HTML), VIDE will automatically add a time stamp after the "date:" whenever you make changes to the file.

Commands common to all editors top

In an effort to comply with standard interface design, especially as it applies to MS−Windows, there are several commands that are common to all editor emulations. These common commands are explained in this section.

In addition to the following keyboard commands, the action of the mouse to move the cursor and select text conforms to normal interface design.

Key	**Command Description**
	Selection Highlighting
*n*Shift−UpArrow	Extend selection n lines up.
*n*Shift−DownArrow	Extend selection n lines down.
Shift−LeftArrow	Extend selection left one character.
Shift−RightArrow	Extend selection right one character.
Shift−Home	Extend selection beginning of line.
Ctrl−Shift−Home	Extend selection beginning of file.
Shift−End	Extend selection end of line.
Ctrl−Shift−End	Extend selection end of file.
	Clipboard
^X [Menu Edit:Cut]	Cut selection to clipboard.
^C [Menu Edit:Copy]	Copy selection to clipboard.
^V [Menu Edit:Paste]	Paste clipboard to insertion point.
	Menu Commands
File:New	Create a new file. You will be prompted for the name of the new file.
File:Open	Open an existing file.
File:View	Open an existing file for read only access.
File:Save	Save (write) current file.
File:Save As	Save current file under a new name. New file becomes current file.

File:Close	Close current file.
File:Save All	Save all open files.
File:Save / Close	Save and close current file.
File:Send to Browser	Send current HTML file to browser. This is an effective way to edit HTML files and view the results.
Edit:Undo	VIDE's undo is a bit limited. Undo will undo the last text delete. It does not undo inserts, clipboard operations, or cursor movement.
Edit:Find	Opens the VIDE Find dialog. This dialog gives you all the options available for finding text. The emulations will support both dialog based finds, and command line finds.
Edit:Find next [F3]	Find next occurrence of find pattern.
Edit:Replace	This opens the Find and Replace dialog. If you check the Confirm Replace option, you will be prompted before the replace is done. That confirmation dialog will also allow you to go ahead and replace all or cancel the find and replace.
Edit:Find Matching Paren	Why is this a menu command? Because the command is useful for beginners who will often use menu commands over keyboard commands.
Edit:Editor Help	Brings up a dialog with a command summary for the editor emulation.

Note about selections. You can use the mouse to highlight an area of text as well as the keyboard commands listed above. If you need to select more text than shows in the window, you will have to use the key selection commands (e.g., Shift−Down).

The Standard Editor top

The generic command set of the VIDE editor is very similar to those found in many GUI based text editors. It is modeless. Commands are either special keys (e.g., arrow keys), or control keys (indicated by a ^ or Ctrl). Several commands use a meta modifier, Ctrl−A (^A) as a prefix. You can enter counts (noted by an "n") for commands by first pressing the Esc key, then a count, then the command. Not all commands are supported by command keys, and require you to use the menus (replace, for example). Note that the See and Vi editors have some features not found in the standard editor.

The Standard Command Set top

Key	**Command Description**
Esc	Prefix to enter count *n*
	Movement Commands
Arrow keys	Standard function

*n*Left	Move left [^L]
*n*Ctl−Left	Move left a word
*n*Up	Move up [^U]
*n*Right	Move right [^R]
*n*Ctl−Right	Move right a word
*n*Down	Move down [^D]
Home	Goto beg of line [^A,]
Ctrl−Home	Goto beg of file
End	Goto end of line [^A.]
Ctrl−End	Goto end of file
*n*PgUp	Move a screenful up
*n*Ctrl−PgUp	Scroll a screenful up
*n*PgDn	Move a screenful down
*n*Ctrl−PgDn	Scroll a screenful down
	Searching commands
^A]	Balance match
^F	Find pattern (non−dialog). This form of find allows you to enter the find pattern directly from the keyboard. The find pattern is terminated with the `Esc` key. The pattern is echoed on the status bar. This form of find is useful for Q−Macros.
^A^F	Find pattern − use find dialog.
Shift−^F [F3]	Find next
	Insertion Commands
n^AIns	Insert char with value of n
n^O	Open a new blank line
Ins	Toggle insert/overtype
	Editing commands
^ABkspace	Delete to line begin [^A']

^ADel	Delete to line end [^A\]
*n*Shft−^C	Fold case
^C	Copy highlight to clipboard
^V	Paste from clipboard
^X	Cut highlight to clipboard
*n*Bkspace	Delete previous char
*n*Del	Delete next char
*n*Shft−Del	Delete line
	Macros
^Aq<a−z	Set register to use (a−z)
^Q	Record keystrokes in Q−Register until ^Q
n^E	Execute current Q−Register N times
	Misc. commands
^AM	Center Cursor in screen
^Av	Repaint screen
n^G	Goto line n
n^K	Kill line
n^B	Beautify code. This beautifies code or fills text. The beautify command will format C/C++, and Java code according to the V style conventions. To use this command, start on a code line that is indented how you want it. After that, 'n' lines will be formatted based on the starting indentation. This command is very useful if you use the V indent style. You can set the indentation for braces in the **Options:Editor** dialog. For text files (including HTML), this command will fill lines to the column specified in the **Options:Editor** dialog.

The See Editor top

The command set of the See editor dates back to the late 1970's. The editor was originally called TVX, and the command set was modeled after the TECO editor. See, like Vi, has command mode and insert mode, and is normally in command mode. The commands are mnemonic. U for up, L for left, F for find, etc. It is very good for touch typists, and minimizes the need to move your fingers from the home row of the keyboard. If

you don't have a favorite editor yet, or if you don't have a command mode editor you like, consider giving the See command set a try. It is really an efficient way to edit.

To use the See command set, start VIDE and select 'See' in the Options:Editor dialog. VIDE will remember your selection.

See is normally in Command mode. You can supply a count value to many commands by entering a value before the command. For example, entering `35d` will move down 35 lines. When you enter insert mode, keys you type are inserted until you press the `Esc` key. The `f` find command lets you enter a find pattern that is echoed on the status line, and can include tabs. Press `Esc` to begin the search. The `F` version of find displays a dialog to enter the pattern.

In most cases, you can use a standard dedicated key (such as the arrow keys) as well as the equivalent mnemonic See command. You can highlight text with the mouse, and cut and paste in the usual fashion.

The See Command Set top

Key	Command Description
	Movement Commands
*n*l	Move left [Left Arrow]
*n*r	Move right [Right Arrow]
*n*u	Move up to beginning of line/TD>
*n*d	Move down to beginning of line
n^U	Move up [Up Arrow]
n^D	Move down [Down Arrow]
n[	Move left a word
*n*Tab	Move right a word [Ctrl−Right]
n^P	Move a screenful up [PgUp]
*n*p	Move a screenful down [PgDn]
,	Goto beginning of line [Home]
.	Goto end of line [End]
b	Goto beginning of file [Ctrl−Home]
e	Goto end of file [Ctrl−End]
j	Jump back to previous location. This is an easy way to get back to where you were before a find command, or a large movement.
n^L	Goto line n

m	Center Cursor in screen.
*n*n	note (mark) location n. Locations go from 1 to 25. This is a bookmark feature.
n^N	Goto noted location n
*n*Ctrl−PgUp	Scroll a screenful up
*n*Ctrl−PgDn	Scroll a screenful down
	Searching commands
n]	Balance match. Find the matching
	paren, bracket, or brace.
f	Find pattern (non−dialog). This form of find allows you to enter the find pattern directly from the keyboard. The find pattern is terminated with the `Esc` key. The pattern is echoed on the status bar. This form of find is useful for Q−Macros.
F	Find pattern (dialog)
;	Find next
^F	Find/replace (dialog)
	Insertion Commands
*n*i	Insert char with value of n. This lets you enter Escapes or other non−printing characters.
i	Enter insert mode
Esc	Exit from Insert Mode
^O	Toggle insert/overtype [Ins]
*n*o	Open a new blank line
	Kill/change commands
^C	Copy highlight to clipboard
^V	Paste from clipboard
^X	Cut highlight to clipboard
'	Delete to line beginning
\"	Delete to line end
n^K	Kill line

n~	Toggle case
/	Kill 'last thing'. The 'last thing' is determined by the previous command. Commands that set the last thing include find, move a word, save, append, and get. Thus, a common way to perform a replace is to find a pattern, then use '=' to insert the replacement.
=	Change 'last thing'. Delete last thing and enter insert mode.
*n*t	Tidy (beautify) n lines. This will beautify C/C++, and Java code according to the V style conventions. To use this command, start on a code line that is indented how you want it. After that, 'n' lines will be formatted based on the starting indentation. This command is very useful if you use the V indent style. You can set the indentation for braces in the **Options:Editor** dialog. For text files (including HTML), this command will fill lines to the column specified in the **Options:Editor** dialog.
*n*Backspace	Delete previous character
*n*Del	Delete next character
	Save buffer commands
*n*s	Save n lines in save buffer. See provides a buffer that allows you to save lines. Using the save buffer is often easier than cut and paste. After saving lines, you can use the '/' delete last thing command to delete the lines you just saved. The See save buffer is independent of the clipboard, and is local for each file being edited.
*n*a	Append n lines to save buffer
g	Get contents of save buffer. Inserts the contents of the save buffer at the current location.
y	Yank file to save buffer. This is an easy way to import text from an external file.
^y	Write save buffer to file. This lets you save part of your file in a new external file. Using 'y' and '^y' is often an easy to copy parts of one file to another, or include standard text into new files.
	Macros
\<a−z	Set register to use (a−z)
q	Record keystrokes in Q−Register until ^Q
n@	Execute current Q−Register N times
	Misc. commands

v	Repaint screen
?	Help
^Q	Save and close

Macros for See top

This section gives some specific examples of using Q−Macros with the See emulation.

First, consider search and replace. Even though VIDE has a search and replace function on the menu, you can do the same thing with a Q−register macro. Consider the following set of *See* commands:

```
fthe$=THE$
```

The '$' represents the `Esc` key. You would enter this macro by selecting a Q−Register (e.g., `\a` to select Q−Register 'a'), entering a 'q' command, then entering the command sequence, and ending the recording the sequence with a Control−Q (^Q). The characters you enter will be echoed on the status bar (which will indicate you are recording). Then execute the macro (22 times, for example):

```
\a22@
```

The next 22 occurrences of 'the' will be changed to 'THE'. The corresponding command in Vi would be: `/the<CR>xiTHE$` where '<CR>' is a Return, and '$' is 'Escape'. Then execute it with `22@a`. Note that in this emulation of Vi, 'x' will delete the pattern just found.

If you want to execute the macro for the whole file, give a very large count. The macro will exit after any command, such as a find, fails.

This is a See example for placing a '** ' at the beginning of every line that contains the word 'special'.

`\aqfspecial$,i** $d^Q` and execute `\a1000@`

The same function in Vi:

`qa/special<CR>0i** $j0^Q` and execute `1000@a`

Future additions top

Things missing from the editor that WILL be included in future versions:

- Auto indent for C/C++ code
- Selectable highlight colors
- Formatted source code printing

The Vi Editor Emulation top

VIDE now includes an emulation of the Vi command set. The emulation is not yet complete, and it is likely it will never be a complete emulation of Vi. However, it is a pretty good "finger memory" emulation. For the most part, the right things will happen when you edit using your automatic "finger memory" of Vi commands. This emulation should improve over time. See the limitations section for a description of the current limitations and differences of this emulation.

To use the Vi emulation set, start VIDE and select 'Vi' in the Options−>Editor dialog. VIDE will remember your selection.

The Vi Command Set

	(* after cmd means emulation not exact)
Key	**Command Description**
	***** Movement Commands *****
h,<Left>	cursor N chars to the left
j,<Down>	cursor N lines downward
k,<Up>	cursor N lines up
l,<Right>,<Space>	cursor N chars to the right
m<a−z>	set mark <a−z> at cursor position
CTRL−D	scroll Down N lines (default: half a screen)
CTRL−U	scroll N lines Upwards (default: half a screen)
CTRL−B,<PageUp>	scroll N screens Backwards
CTRL−F,<PageDown>	scroll N screens Forward
<CR>	cursor to the first CHAR N lines lower
0	cursor to the first char of the line
$	cursor to the end of Nth next line
<Home>	line beginning
<CTRL−Home>	file beginning
<End>	line end
<CTRL−End>	file end
B*	cursor N WORDS backward ['word' not same]
"b	cursor N words backward

`<a−z>	cursor to the first CHAR on the line with mark <a−z>
``	cursor to the position before latest jump
'<a−z>	cursor to the first CHAR on the line with mark <a−z>
''	cursor to first CHAR of line where cursor was before latest jump
<MouseClick>	move cursor to the mouse click position
	***** Searching commands *****
/{pattern}<CR>	search forward for {pattern}
/<CR>	search forward for {pattern} of last search
?{pattern}<CR>	search backward for {pattern}
?<CR>	search backward for {pattern} of last search
N	repeat the latest '/' or '?' in opposite direction
n	repeat the latest '/' or '?'
%	go to matching paren (){}[]
	***** Insertion Commands *****
A	append text after the end of the line
a	append text after the cursor
i,<Insert>	insert text before the cursor (until Esc)
O	begin a new line above cursor and insert text
o	begin a new line below the cursor and insert text
R*	toggle replace mode: overtype existing characters [just toggle]
CTRL−C	Copy to clipboard
CTRL−V	Paste from clipboard
CTRL−X	Cut to clipboard
	***** Kill/change commands *****
C	change from cursor position to end of line, and N−1 more
cc	delete N lines and start insert
c[bBhjklwW$0]	delete Nmotion and start insert

D	delete chars under cursor until end of line and N−1 more
dd	delete N lines
d[bBhjklwW$0]	delete Nmotion
J*	Join 2 lines [2 lines only]
S	delete N lines and start insert;
s	(substitute) delete N characters and start insert
u	undo changes
X, <BS>*	delete N characters before the cursor
x, <Del>	delete N characters under and after cursor
~	switch case of N characters under cursor
	***** Yank buffer commands *****
P	put the text on line before cursor
p	put the text on line after the cursor
Y*	yank N lines; synonym for 'yy' [cursor at end]
y<	read file into yank buffer
y>	write yank buffer to file
	***** Misc. commands *****
CTRL−L	redraw screen
ZZ	save file and close window
gb	beautify N lines of C/C++/Java code, fill text
gm	center cursor in screen
"<a−z>	set q−reg/buff for next op
["<a−z>]q	record to q−register until ^Q
["<a−z>]@	execute q−register N times

Emulation Limitations top

As much as possible, this emulation tries to do the same thing Vi would do with the same command. Probably one of the main things missing in this emulation is the total lack of **Ex** support, i.e., there are no ":" commands supported. In fact, this should be a minor limitation as there are usually menu commands that

support the most important ":" equivalents. You can also record macros that can duplicate many of the functions typically done with ":" commands.

Another major difference is the lack of support for a count for insert and find operations. Thus, "5ifoo$" will not insert 5 instances of "foo". This is true for some other commands as well. Generally, if you enter a command expecting some kind of repeated operation, and the VIDE Vi emulation does not support repeat, then the operation will be done only once. Again, you can use macros to get around this limitation.

The following list summarizes some of the other differences in how this emulation works.

- **c** and **d** commands are somewhat limited. The currently motions supported after the 'c' or 'd' include: 'b', 'B', 'h', 'j', 'k', 'l', 'w', 'W', '$', '0', as well as the 'cc' and 'dd' forms. *Currently, only the full line motions add text to the yank buffer.* Most of the other somewhat obscure motion commands used with 'c' and 'd' can often be done using the mouse to cut and paste.
- The **y** command currently supports only the 'yy' form of the command. In addition, the new VIDE version commands 'y<' and 'y>' have been added. These are used to support external files. Use 'y<' to read an external file into the yank buffer, and 'y>' to save the contents of the yank buffer to an external file.
- The **Backspace** key deletes the character in front of the cursor like almost all other apps you are likely to use instead of moving left. A little note: The backspace key is really CTRL−H. For consistency, the original Vi used CTRL−H and 'h' as left cursor. Now why are 'h', 'j', 'k', and 'l' used for cursor movements? Because the computer lab at UCB where Vi was written had mostly ADM−3a terminals. (ADM−3a terminals were far cheaper than most other terminals available at the time.) And the 'H', 'J', 'K', and 'L' keys had cursor arrows printed on each of them. Thus, the "hjkl" commands for cursor motion.
- Cut, Copy, and Paste are implemented using the standard system clipboard. The standard 'CTRL−X', 'CTRL−C', and 'CTRL−V' commands are used to support clipboard operations.
- The find commands will highlight the pattern found. If you press 'X' or 'x' while the pattern is still highlighted, the pattern will be deleted. While this is different than standard Vi, it is really handy for search and replace.
- This emulation doesn't use the visual '$' marker for some kinds of edits on partial lines, it just does the edits. Another left over from the ADM−3a days, I think.
- Left and right cursor movements flow to the adjacent lines. The V editor class just doesn't support movement limited to one line.
- The 'J' command will only join 2 lines.
- The find command ('/') echoes the pattern on the status bar, and not in the text window. You can use an 'Esc' to terminate the pattern as well as a 'CR'. Find does not support regular expressions. Selecting find from the tool bar or menu brings up a dialog box instead of using the status bar.
- Use menu command or the tool bar button for find/replace.
- The 'gt' command formats C, C++, and Java code. See a more complete description of the See 't' command. Right now, the formatting is limited to the **V** standard, but support for other styles will likely be added.
- The 'R' command toggles Insert and Overtype mode. It doesn't force overtype (replace) mode.
- The '%' command works a bit differently. It currently only supports paren−like characters, and you must place the cursor on the paren you want to match.
- The cursor doesn't change shape for Insert and Normal modes. The mode is shown on the status bar.
- You need to use the menus to load and save files.
- VIDE's concept of a 'word' is different than Vi's. There is no difference between 'w' and 'W', for example. This might get fixed. Because you get visual feedback, this probably isn't a big problem.
- The Vi emulation was built starting with the See command interpreter code. This saved lots and lots of work with minimal compatibility issues. One however, involves counts. The current code can't

distinguish from the default count of 1 and an explicitly entered count of 1. Thus, commands that use default counts might not work correctly when explicitly given a count of 1. For example, the 'G' command can't go to line 1 −− it goes to the last line.
- The 'gm' command is used to center text on the screen.

The following list summarizes most of the Vi features that have not yet been implemented. They are likely to be added in future releases.

- The '.' command is not supported. I hope I can figure out a way to add it, but the basic V editing class is not very compatible with this command.
- There is only one yank buffer. Named buffers are not yet supported.
- If there are Vi features that are important to you, please let me know by e−mail, and I will try to add them. I won't add ':' command support, but I may add the equivalent support via an extended command set. If some of the emulated commands don't quite work how you expect, let me know and I will try to make them work more closely to Vi.
- If you really want to add some features (like ':' support), the source code is available! Fix away!

No Warranty top

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

Bruce E. Wampler, Ph.D.
bruce@objectcentral.com
www.objectcentral.com

VIDE C/C++ Tutorial

VIDE Overview

The design of VIDE has been somewhat evolutionary, but you should find that it is not that much different than other IDEs you may have used. Because VIDE is a free, open source program, it probably lacks some of the polish of commercial IDEs. However, it is still quite functional, and it is really easier to develop programs with it than it is to use a command line interface.

Generally, any application you write will consist of various source files (with associated header files for C/C++), and required data files. These files are generally dependent on each other. By defining a VIDE Project for your application, all the file dependencies are automatically handled. The standard tool `make` is used for C/C++ files, while the JDK Java compiler automatically handles dependencies.

Using VIDE, the normal work cycle goes:

1. Design your application.
   VIDE currently has no capabilities to help with this stage.
2. Start VIDE, and create a Project File.
   This will include all source files, compiler options, and other information needed to compile your application.
3. Build your project.
   This stage compiles your source into object code. Compilation errors are displayed in the status window, and you can simply right−click on the error to go to the offending line in your source code. After making corrections, you repeat this step until all compilation and linking errors are removed.
4. Run your program.
   You can start your program from within VIDE.
5. Debug your program.
   VIDE for MS−Windows has integrated support for the **gdb** debugger for C/C++. Because the `DDD` debugger available for Linux is so good, VIDE for Linux does not have integrated debugging support, but will automatically launch DDD.
   VIDE currently does not support the jdb debugger for Java, but it will "any day now."
6. Write documentation for your application.
   VIDE has syntax highlighting for HTML to make that job easier. You can also automatically launch your web browser to view the resulting HTML pages. Really neat.

Using VIDE with GNU gcc/g++ top

The main C/C++ compiler `VIDE` is designed to work with is the GNU Compiler Collection (gcc), either on a Unix−like system, or on MS−Windows with the GNU GCC compiler. (VIDE will work with either the MinGW version of GCC, or the Cygnus version. Try the latest merged GCC 2.95.2!) Functionality of `VIDE` for gcc is based on standard GNU makefiles. VIDE uses a standard GNU make Makefile to build your project. Thus, you must have a Makefile defined. This Makefile can be one created automatically by VIDE itself from your Project file, one generated by the `vgen` V application generator, or even one you've written yourself. If you have your own Makefile, then you probably won't need to use a VIDE Project.

VIDE assumes you have your gcc/g++ compiler already installed on your system and the PATH correctly set. For Unix/Linux systems, this is a given. If you are using a MS−Windows version (Mingw32 or Cygnus), then you must follow the instructions provided with their distributions. You might find it helpful to copy the VIDE executables to the /bin directory that has your g++ compiler. VIDE requires GNU make (it is not compatible with some other versions of make), the GNU C/C++ compiler (preferably the latest GCC 2.95) and associated utilities, and the GNU gdb debugger. You also may want the V utilities vgen and viconed.

Using VIDE with Borland C++ 5.5 top

VIDE 1.07 has added support for the free Borland compiler. This support does *not* include a debugger! More complete information about using VIDE with the Borland compiler is found in the VIDE Borland reference guide. It is *essential* that you read the information in that reference to get a properly working version of VIDE with Borland BCC32.

Functionality of `VIDE` for BCC32 is based on the non−standard Borland MAKE program. VIDE uses a Borland make Makefile to build your project. Thus, you must have a Makefile defined. This Makefile can be one created automatically by VIDE itself from your Project file, or one you've written yourself. If you have your own Makefile, then you probably won't need to use a VIDE Project.

VIDE assumes you have your Borland compiler already installed on your system and the PATH correctly set. When working with the Borland compiler, VIDE requires the Borland make.

VIDE Projects top

General procedures

Once you start `VIDE`, you will see a blank window labelled "No Makefile, Project, or .java file Specified." This opening window is the message window, and is used to output the results of your make. A typical first step after starting VIDE is to open a VIDE Project or select an existing Makefile.

Once you've opened a project or specified a Makefile, you can build your project with the **Build:Make C++** menu command, or click on the Make button on the tool bar. This runs the make utility with the default target (often "all"). VIDE first runs the Makefile in dry run mode. It uses that output to then run the commands generated. It intercepts the error messages for g++ and put them in the message window. You can then right−click the error line, and VIDE will open up the file in question, and put the cursor on the offending line. This all assumes that the source and makefile are in the same directory. VIDE also assumes all your files have unique names (i.e., you don't have files in your project with the same name but in different directories).

After you correct the problem, rerun make.

You can also make a specific target in your makefile by using the Make menu: **Make:Make <target>**. If you include a "clean" target in your Makefile, **Make:Make Clean** will run make clean. (The Makefile generated by VIDE from a Project has a "clean" target.)

The tools menu allows you to run your program. If you are using a VIDE Project, then **Tools:Run project** will run the project you just built. If you are using your own Makefile, then **Tools:Run project** will prompt you for the file to run.

VIDE C/C++ Projects top

When you are first creating a new project (or moving an existing program to VIDE), click on the **Project:New C++ Project** menu. After you select a name for your project, a "wizard" dialog will open that will let you begin defining a project. The first thing to fill in is the name of the program you are building (*Target name*). If you are building an application, this will be something like "foo.exe" on Windows, or simply "foo" on Linux or Unix systems. Libraries end with a ".a" extension: "foo.a" on both platforms, or a ".lib" for Borland.

Next, select the type of project you want to build. A *Console Application* runs in a shell or console window, and does not use any GUI components. A *GUI* application uses a graphical interface. On Windows, this will be the standard WIN32 API, while on X systems, you will have a choice of Athena, Motif, or gtk. You can also use the V GUI. You will be given a choice between using the static V library or a dynamically loaded library. If you are using OpenGL or Mesa, check that box.

You can also build a static library. Static libraries are usually easy to build. Both Windows and Linux support dynamic libraries (called DLLs on Windows and shared libraries on Linux). However, the rules for defining and getting a shared library (especially DLLs) is somewhat beyond the scope of VIDE. The VIDE project file can support DLLs and shared libraries, but you have to just how to put a dynamic library together first. See the later section on Advanced options for a few more details of building a DLL.

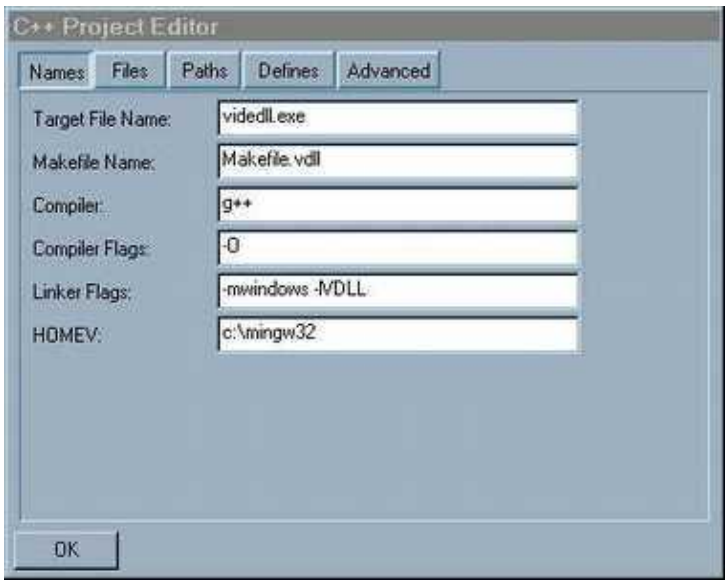
You can also select which you are building, a release version, or simply a debugging version of your project. These options simply determine some switches to the compiler. VIDE does not support both a release and a debug build within the same session. The easiest work around for this is to first build one version, then use **File−>Save As** to save the alternate version.

The Windows version also lets you select which compiler you are using − mingw32, Cygnus, or Borland. Setting the "−mno−cygwin" option will build an application under Cygwin without using the Cygwin DLL.

Once the initial project attributes have been selected, You will get the project editor dialog box with various tabbed items. Most of the fields will be filled in according to the values you set in the opening dialog. The main thing you will probably want to do is add source files using the Files tab. You can set defines as needed in the other tabs. Once you have added the files needed and click "Done," VIDE will create a Makefile suitable to compile your project with gcc or Borland BCC32.

Each of the project editor tabs are described in more detail in the following sections. (These screen shots were taken from the project file that builds VIDE itself using the V GUI library.)

Names



This pane lets you set the target name for the executable or library. You can also change the name of the generated Makefile. Usually, you will use g++ as the compiler. The Borland compiler is BCC32. The The target name determines what kind of final target is built. A ".exe" extension on Windows, or no extension on Linux, causes an executable to be built. If the target name ends in ".a" (or ".lib" for Borland), then a static library will be built. It is important to get these extensions right to generate the correct kind of target.

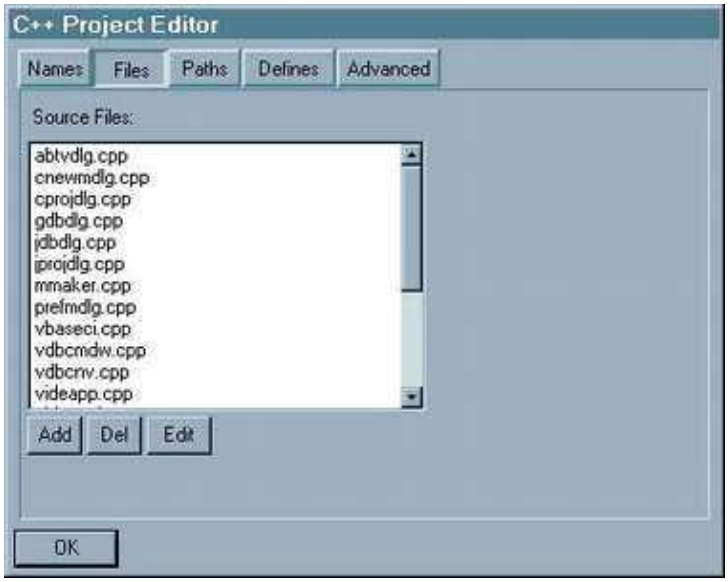
Compiler Flags line lets you pass switches to the compiler, such as −O for optimize, or whatever. The linker flags are passed to the linker, and usually consist of a set of library references. The new project wizard will usually fill these fields in as needed for console, GUI, and V apps. The HOMEV value is required for programs that use the V library if the V GUI system has not been installed in the same places as other libraries and include files on your system.

You may have to change some of the default switches for your specific compiler or operating system. You should only have to do this once for a project. If you will be creating other projects, you can save a template

project in a file of your choice, and then use it as a starting point for new projects.

The Borland guide gives some specific details for the Borland version.

Files



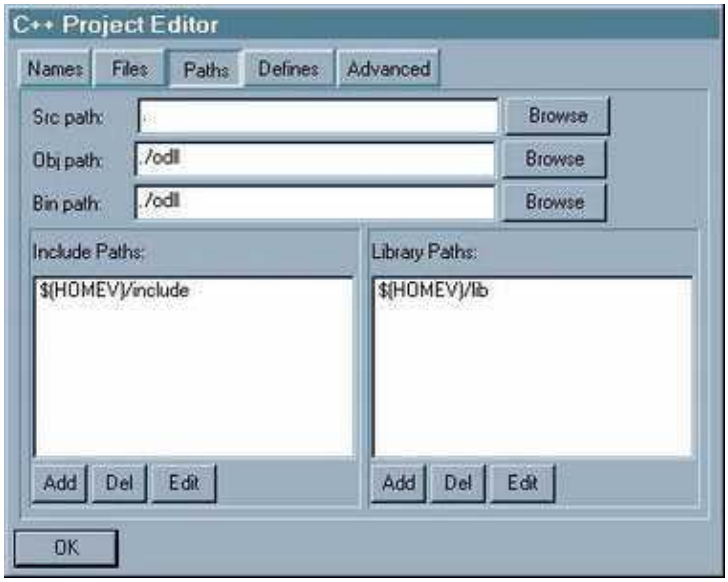
This lets you add the names of the source files included in the project. Clicking ADD brings up a file selection dialog. When you select a file, the file is added without any path name. To delete the selected entry, use the Del button. Until V adds multi−line selection (someday soon), you have to add files one at a time.

On Windows, the source file can be a ".rc" resource file as well. If you include a ".rc" file, VIDE will automatically add the dependency to the Makefile, and use the appropriate resource compiler (WINDRES for gcc or BRCC32 for Borland) to compile a ".o" file for gcc or a ".res" file for Borland.

Note: Adding relative file paths
The current mechanism for adding files does not support adding relative or absolute paths in front of a file name using the file selection dialog. Normally, VIDE assumes that you will have your source files in the directory specified in the source directory path. Thus it strips any leading path. However, you *can* add relative paths. If you need to add a relative (or absolute) path, first add the file using the file selection dialog. Then, select the file from the list file, and click Edit to hand edit the entry. Then add the relative path to the file.

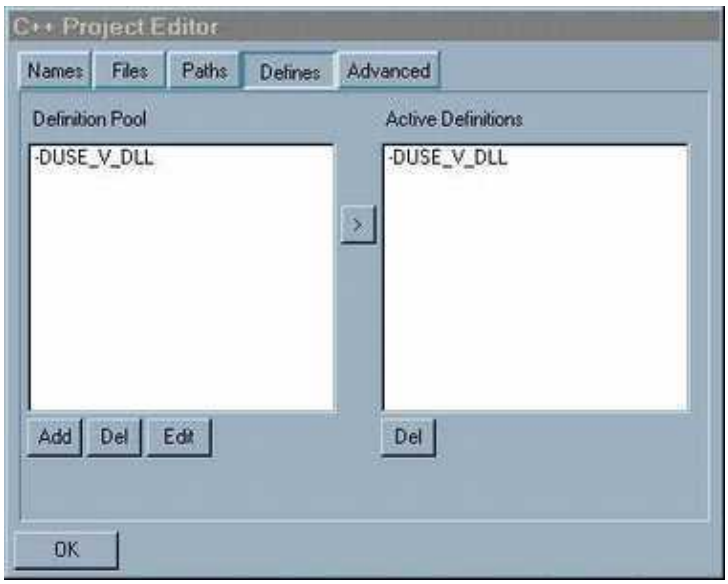
(Note: using the file selection dialog can leave the current directory set to something other than the default source directory, and when you exit the project editor, the generation of the Makefile may fail because things get started in the wrong directory. This problem is a bit hard to fix, and only happens when you use the file dialog box to select a file from a different directory. Just edit the project again, and the Makefile will be generated correctly. Someday I may fix this, but I have higher priorities for now. Sorry.)

Paths



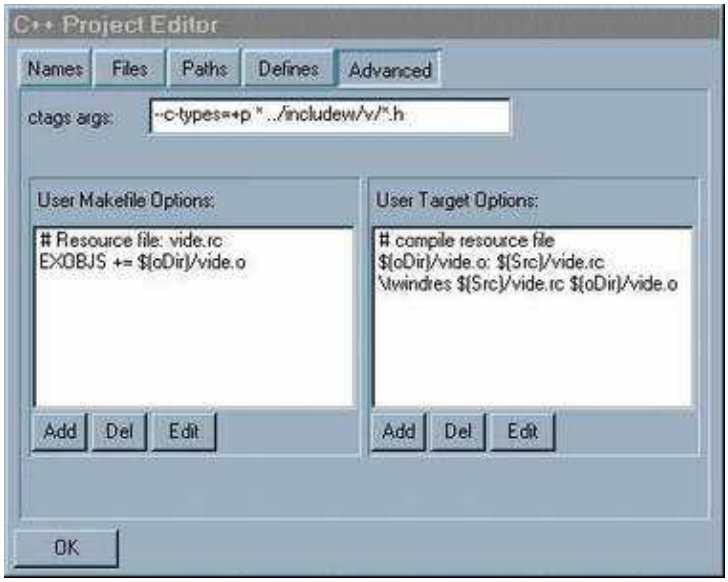
This lets you specify the directory for the source files, the directory where you want object files to be generated, and the directory where the binary should be written to. It is best to use relative paths for these whenever possible. You can also list paths for include and library directories. These are passed to g++ as the appropriate switches.

Defines



It is often helpful to provide compile time defines for C and C++ programs. This tab lets you add compile time defines. The left list shows a a pool of definitions that you might want to use. To have them included at compile time add them to the Active Definitions list using the ">" button between the lists. The pool will stay constant, while the Active side can be a bit more dynamic as you work with your project. Note that you have to supply the full g++ definition switch: −DFOO or −UNOTFOO.

Advanced



The first field of the Advanced pane lets you define a project specific argument for the ctags program. For example, the entry "−c−types=+p * ../includew/v/*.h" does several things. First, it adds extra information about function prototypes. Then it includes the V library headers in the tags file for easy lookup of functions used from the library. Check the ctags documentation for other switches you might want to use.

The other advanced panels let you add lines to the Makefile in one of two places. You will have to have a pretty good understanding of Makefile in order to make effective use of these options.

Anything you add to the "User Makefile Options" list will be written to the generated Makefile immediately after the standard definitions. You could use it to define your own symbols, or whatever.

The "User Targets" list lets you add new targets other than the defaults to make. You can use these to define specific values you want to add to your Makefile. The Borland support uses these to define the default runtime libraries.

If you want to build something other than an executable or a library, there is one important feature provided by the user targets list. If the *first* entry has "#all" in the it, then VIDE will not generate an "all" target (usually the same name as the target name) for the make file. It assumes you are providing the all target here instead.

Note that when you need a leading tab for the makefile, enter a '\t' into the project. It will be automatically converted to a real tab in the final makefile.

Direct Project File Editing

A VIDE `.vpj` project files is in fact regular text file. It is laid out in clearly labeled sections. While you can add entries to any section using the Project Editor, advanced users may find it easier to edit the V project file directly to add definitions and targets that aren't generated automatically. Using these two mechanisms (defines and targets), you can build complicated makefiles which will be automatically generated from the project file. It is possible to define practically anything you might need to include in a makefile. For example,

all the options needed to generate the Windows DLL version of V are included in a VIDE project file.

I'm not going to explain every detail of the project file format here. Any programmer reasonably familiar with makefiles will be able to see some of the potential of the VIDE project file. For example, VIDE uses a set of standard variable names such as "oDir" and "EXOBJS". The standard make targets are defined using symbols, too. It is easy to define entries in the user defined symbol section that use the makefile "+=" operator to modify and add to the standard symbols. You can add very complicated targets to the user target section, especially when you override the "all" target with the "#all" convention. Perhaps the best thing to do is look as a VIDE Project File, and study the VIDE and VGUI project files for examples of some of the things you can do. The main advantage of using a project file is the automatic generation of dependencies and other features that will be eventually included in VIDE.

VIDE Help System top

VIDE now includes a Help menu. Most of the help is supplied in a separate distribution file, and is in HTML format. V Help uses your default Web Browser to show the help files.

I have attempted to collect the most useful documentation I could find for the various GNU C++ tools. If you download and install the Help files, you should have a very complete and useful set of documents for C, C++, and Java programming at your finger tips. See Installing VIDE for more instructions on how to install VIDE help.

If there are other HTML based documents you would like added to the VIDE distribution, please let me know.

Debugging with VIDE top

VIDE supports the standard GNU **gdb** and Sun **jdb** debuggers. The VIDE interface to the debuggers makes it far easier to debug your code, but is of minimalist design. The goal is to make using the native debuggers as easy as possible for casual users, while maintaining the full power of the debugger for experienced users. VIDE accomplishes this by showing a command window interface to the debugger. You can enter any native debugger command in this window, and thus have full access to all debugger features.

VIDE makes using the debugger easier by providing a popup dialog with the most often used commands. And most importantly, VIDE will open the source file in an editor window and highlight the current execution line on breakpoints or steps. It is very easy to trace program execution by setting breakpoints, and clicking on the *Step over* or *Step into* dialog buttons. VIDE also allows you to inspect variable values by highlighting the variable in the source and right clicking the mouse.

A description of debug dialog commands is provided in the VIDE Command Reference section.

Debugging C/C++ with gdb

To debug C/C++ programs with **gdb**, you must first compile the program with debugging information. This is accomplished with the **−g** switch on the compile line. The current version of VIDE does not provide automatic generation of debug or release makefiles. The easiest way to define VIDE projects for both debug and release versions is to use the **Project:Save Project as...** command. First, define a release version of the project. Then, using that project as a template, change the switches as needed for your debug version, and save the project under a different name.

The full power of gdb is available in the debugger command window. You may enter any standard gdb command after the "(gdb)" prompt. In fact, there really is limited interaction between VIDE and gdb, mostly handling breakpoints. VIDE starts gdb using the "−f" switch, which causes gdb to send a special output sequence after each break, which VIDE then uses to open and display the highlighted break line.

VIDE maintains its own list of breakpoints, which it keeps even if you start and stop the debugger. It is important that you use VIDE commands to set and delete breakpoints. If you enter breakpoints directly into the gdb command window, VIDE won't know about them, and won't highlight them in your source code.

Limitations with gdb

- [MS−Windows 9x/NT] Output from console applications isn't properly displayed in the gdb command window. If you run gdb from a command window, the output is displayed as it is generated. With the VIDE gdb command window, the output from the running program is not displayed until the very end when the program terminates. I have no idea why gdb behaves like this, but it must have something to do with the way I use CreateProcess. Any help on this problem would be appreciated.

- [MS−Windows NT] On Windows NT, gdb seems to bring up a message dialog trying to open some file over the network. I don't have a networked NT machine, so I don't know if this is a general gdb feature or what, but gdb seems to work if you just press the appropriate button to ignore the problem when the dialog is displayed.

No Warranty top

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

Bruce E. Wampler, Ph.D.
bruce@objectcentral.com
www.objectcentral.com


GUI
IDE

Using VIDE with the Sun JDK

VIDE Overview top

An Integrated Development Environment (IDE) is a software tool intended to make the process of writing programs easier. An IDE typically works as a unit with your compiler environment. It allows you to edit your program source code, compile it with the compiler, fix syntax errors, run it, and debug it, all from inside a consistent and convenient environment.

VIDE has been designed to work with Sun's JDK (Java Development Kit). Instead of using the compiler and interpreter from command shells (MS−DOS prompts), you interact with them using VIDE. The entire development process is simplified.

I will discuss the specifics of Sun JDK later (if you have programmed before, or if you have used an IDE before, you can probably skip to that section right now). But first, here's the general procedure for building a working Java application using VIDE and Sun's JDK.

Generally, any application you write will consist of various source files, and required data files. In order to get your program to compile and run, you must have all the files available. To track all the files, and to set various options required by the compiler, you will create a VIDE Project for your application. The VIDE Project file contains all the information needed to build and run your application using the JDK tools.

Thus, using VIDE, the normal work cycle goes something like this:

1. **Design your application.**
   Sorry, you have to do this part yourself. While there are software tools that can help you do this, VIDE currently has no capabilities to help with this stage.

2. **Start VIDE, and create a Project.**
   You should almost always start with a new Project, even if you don't yet have any source code entered. It is best to name the project the same name as your Java application name. By creating a new project, you will define the application name, and thus the name of the main source file. You also can define compiler options, and other information needed to compile your application.

3. **Create your source code.**
   Typically you will need to create the source code. VIDE provides an excellent programmer's editor. Programmer's editors are really somewhat different than a generic text−pad like editor, or even a word processor. Typically, they have commands appropriate for programming, and support special features like syntax highlighting and automatic program formatting. VIDE is no exception. It has syntax highlighting, program formatting, and an extensive help system designed just for programmers.

4. **Build your project.**
   Once you have your programs entered, you will need to compile your source to object code. It is almost certain that your code will have both typos and logic errors. When you compile your code from within VIDE, you will see compilation errors displayed in the status window. You can simply right−click on the error, and VIDE will open the source code file, and go to the offending line. After making corrections, you repeat this step until all compilation and linking errors are removed.

5. **Run your program.**
   You can start your program from within VIDE by clicking the run icon.

6. **Debug your program.**
   Your program will almost certainly have logic errors in it as well. A debugger is used to help you find the logic errors in your program.

7. **Write documentation for your application.**
   Once you have an application running and tested, you will likely need to write documentation for it. VIDE provides some extra support for HTML, including syntax highlighting. You can also automatically launch your web browser to view the resulting HTML pages. Really neat.

Java and the Sun JDK top

Java Basics

Getting a Java program to run on your machine normally takes several steps. First you create the source file (using the VIDE editor). That source code is then compiled by the Java compiler `javac`, which produces Java bytecode output. Java bytecode can be run on any computer that supports Java. Using the JDK, you typically execute your bytecode by running the Java bytecode interpreter. There are *two* versions for MS−Windows: `java` (for console apps), and `javaw` (for GUI window apps). On Linux, because applications are started much differently than on MS−Windows, there is just one: `java`.

If you have an applet, you need an HTML file to go with the applet bytecode, and then run the applet either by running your usual web browser, or using the JDK's `appletviewer`.

Typically, you run the compiler and interpreter from a command window. This is usually an `MS−DOS Prompt` on Windows, or the normal command shell or console window on Linux. (Note that using VIDE,

however, eliminates the need to explicitly use a command window.)

To summarize:

- Java console application:
  Source code −> `javac` −> `java`

- Java MS−Windows GUI window application:
  Source code −> `javac` −> `javaw`

- Java Linux GUI window application:
  Source code −> `javac` −> `java`

- Java applet:
  Source code −> `javac` −> + *HTML* −> *browser*
  −or−
  Source code −> `javac` −> + *HTML* −> `appletviewer`

Important Note: MS−Windows vs. Linux consoles

Linux and MS−Windows start programs in a fundamentally different way. Because of this, Sun supplies both java and javaw for MS−Windows. And because Sun supported the JDK for MS−Windows first, VIDE was designed to work best for that environment. This means there is a little difference between the Linux and the MS−Windows version.

When you run a Java console app from inside VIDE, you need to create a console shell to see your output. VIDE uses the fact that you've specified `java` as the interpreter to decide to run your app in a console. If you've specifed `javaw`, then VIDE will launch your app so that it does not have a console. Even though the Linux version of the Sun JDK does not have a `javaw` interpreter, you still need to specify `javaw` for Linux GUI apps. When you create a new Java project, VIDE will automatically fill in `javaw`. When you actually run your app, VIDE will run the correct `java`, even though the project file says `javaw`. This slightly strange convention allows VIDE project files to be identical on MS−Windows or Linux.

Sun JDK Specifics top

As distributed, VIDE has been designed to use the default conventions of Sun's JDK 1.2 (Java 2). It should be easy to change the default values to whatever is required by earlier JDKs. VIDE assumes you have correctly installed JDK according to Sun's instructions, and have appropriately set whatever environment variables you need in your `AUTOEXEC.BAT` file on Windows, or the appropriate shell initialization file on Linux.

VIDE allows you to interact with the JDK tools from within VIDE rather than running all the JDK components directly from a command window. VIDE also takes advantage of some of the advanced features found in the JDK component. VIDE lets the JDK compiler `javac` handle all the file dependencies for projects that require more than one source file.

Once you start `VIDE`, the first action normally is to open a VIDE Java Project or select a Java source file.

The opening window is the message window, and is used to output the results of your compiles. If you have a small Java application (not an applet), you can simply specify the name of the top level Java class file from the **Project:Select Makefile or Java file** menu. However, it is almost always better to use a VIDE Java Project. If you have an applet or need to specify compiler options, you must create a new Java Project file.

Once you've opened a project or specified a Java source file, you can compile your project with the**Build:Compile Java** menu command, or click on the Make/Compile button on the tool bar. This runs the Java compiler, `javac`. The results of the compile are shown in the message window. If you get an error, you can usually right click on the error line and the source file will be loaded into an edit window, and the cursor placed on the offending line.

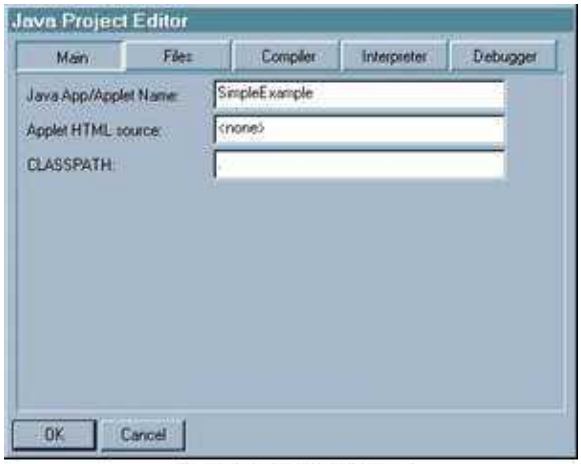
The Development Cycle using VIDE top

Defining the VIDE Java Project top

When you are first creating a new project (or moving an existing program to VIDE), click on the **Project:New Java Project** menu. You will get a dialog box with various tabbed items. You will only need to use the *Main* and *Files* tabs for most projects. Once you've specified the required information, you can compile your project as described earlier.

HINT: When you open a new project file, VIDE assumes some options and switch settings that are commonly appropriate to use with the compiler. It is likely that the defaults will not be the one *you* want. There is an easy solution. Simply create a new project, change the settings to be just what you want, and then save that project as a "template" using an appropriate name. Next time, open that template project file, and immediately use **Project:Save Project As...** to save it under the working name of the new project.

Main

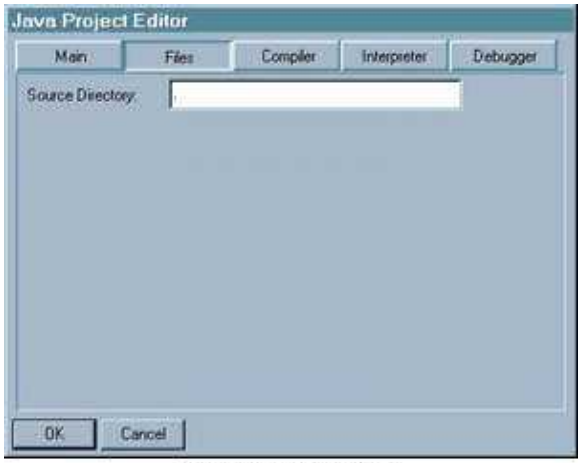


Java Project Editor: Main

This pane lets you specify the name of your application or applet. This should be the Java Class name of the top class. You don't need to add .java or other extension. If you are creating an applet, then you will need to specify the name of the file with the HTML code that runs your applet. When you run an applet, VIDE uses JDK's `appletviewer`.

If you have JDK 1.2, you usually won't need to use the CLASSPATH setting. However, earlier JDKs and some circumstances may require the CLASSPATH to be specified. This information is passed to the compiler and interpreter via switches. See Sun's documentation for more information about CLASSPATH.

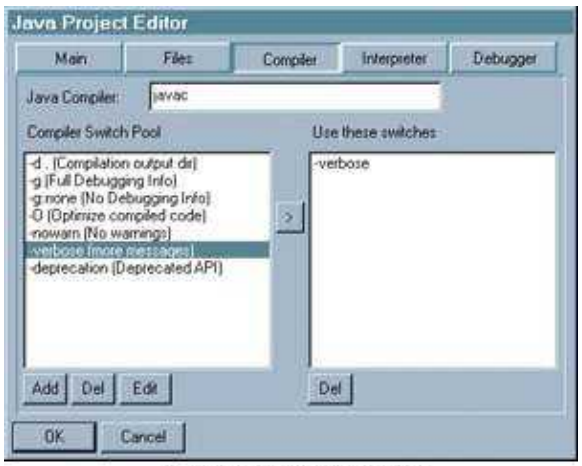
Files



Java Project Editor: Files

This pane is unused in the current version of VIDE. It will eventually let you specify which files are included in your project, as well as the source directory. In this version, VIDE simply uses the app name specified in the Main tab. However, the information will be used in later versions.

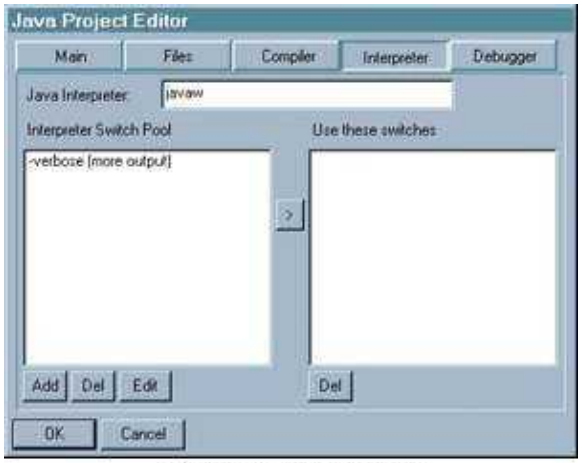
Compiler



Java Project Editor: Compiler

This pane lets you specify which Java compiler to use. It is almost always `javac`, but you can change it for other development environments. The Compiler Switch Pool has the standard switches supported by JDK 1.2. Click the > button to use a switch. You can add your own switches to the Pool if you need to (for other JDKs, for example).

Interpreter



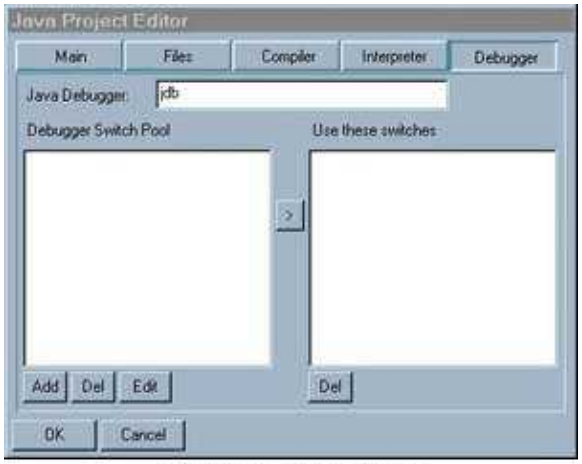
Java Project Editor: Interpreter

This pane lets you specify the interpreter to use to run you app. The default value is `java`, which is suitable for running console−type apps. If you use `awt` or `swing` for GUI base apps, you should use `javaw`. You can still use `java` to run GUI apps, but you will get an extra command window on the screen as well as your GUI window.

Note that you specify `java` or `javaw` on Linux, even though the Linux JDK does not have a `javaw`. VIDE will start your app correctly automatically: `java` means a console app, `javaw` means a GUI app.

Java applets are handled differently. To view an applet, you must specify the HTML source in the Main pane, which will cause VIDE to launch `appletviewer`. If you want to see your applet inside a browser window, you can edit the associated HTML file, and use **File:Send to Browser** to start your browser.

Debugger



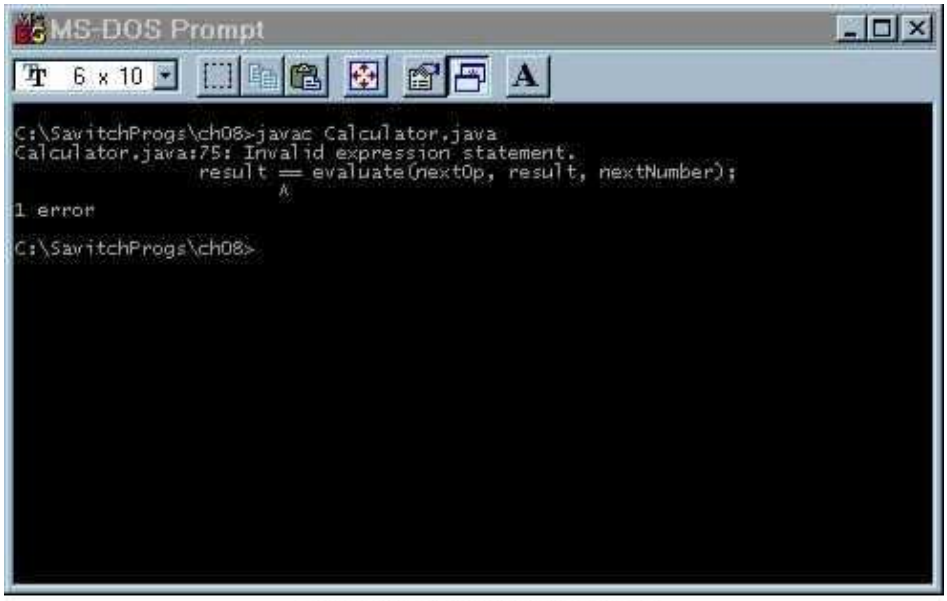
Java Project Editor: Debugger

This pane will let you specify options you may need for the debugger.

Compiling your program top

Once you have created your source files and created your project, you will compile your Java code from inside VIDE. It is this part of the development cycle that VIDE really shines and makes your life easier. All programmers, but especially beginning programmers, make errors when entering the source program. These errors result in what are called *syntax errors*, and cause the compiler to generate error output.

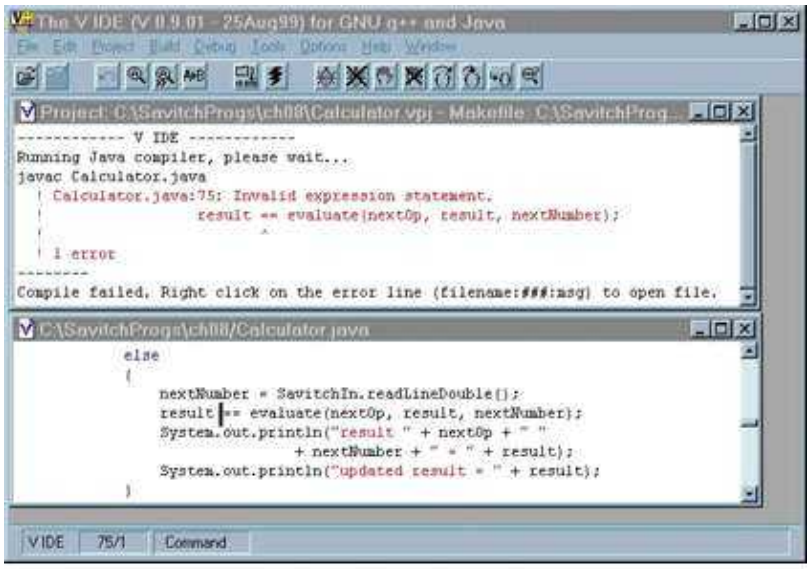
When you use the compiler from a command window, you first must enter the compiler command and the name of the file. For example, `javac Calculator.java`. The output of the compiler is then displayed in the command window as shown in the following figure:



Java Syntax Error in Command Window

Often, a single syntax error will cause the compiler to generate multiple errors. If there are too many errors, the error output often scrolls off the command window. And then you have to edit the source code by finding each line with an error.

Using VIDE is much simpler. To compile, you either use the **Build:MakeC++/Compile Java** menu command, or click on the build button  . When you run the compiler from within VIDE, the resulting error output is put in the VIDE status window:

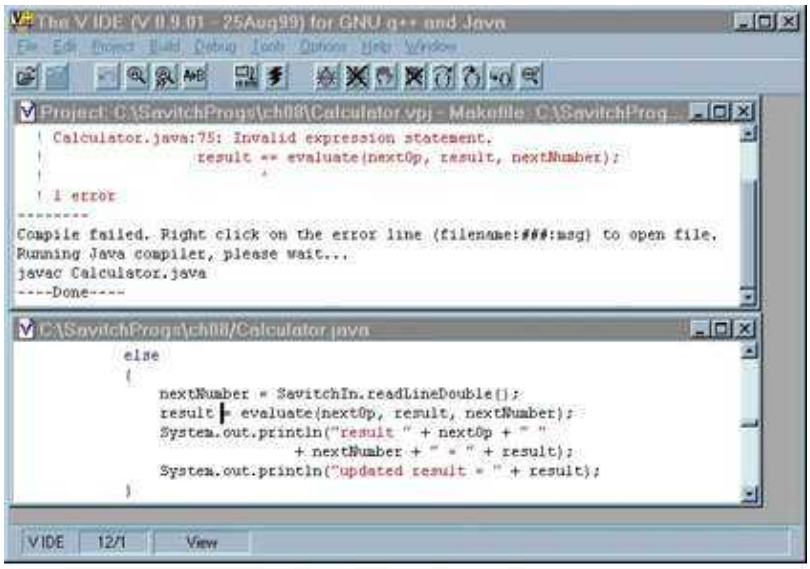


Java Syntax Error in VIDE

If there are too many errors, you can scroll the window to see them all. But best of all, you can right−click on error line, and VIDE will open the source file, and put the cursor on the line with the error.

(Note: you must put the cursor on a line that has the form "Filename:line#:message". If you click on a different line, VIDE will not open the source file.)

Once you've corrected the syntax error, you can simply click the build button again, and recompile the program. You continue this process until your program compiles successfully.



Java Syntax Error in VIDE - Fixed!

Running your program top

Once you have your program compiled, you need to run it with the Java interpreter. When you enter the **Tools:Run Project** menu command, or click the run icon  , you will run your program with the interpreter set in the *Interpreter* pane of the project dialog. The `java` interpreter will run both console−type apps and GUI apps, but is really more suitable for running console−type apps. If you use `awt` or `swing` for GUI base apps, you should use `javaw`.

Java applets are handled differently. To view an applet, you must specify the HTML source in the *Main* pane of the project dialog. VIDE will then use the JDK `appletviewer` to run your applet. If you want to see your applet inside a browser window, you can edit the associated HTML file, and use **File:Send to Browser** to start your browser. Run your program by clicking the run button.

Debugging with VIDE top

Just as most program have syntax errors when they are first created, they have logic errors when you first run them. There are three main ways to find logic errors.

Most programmers neglect what is often the best way to find logic bugs − simply reading the source code carefully. While reading code isn't as much fun as running a debugger, it often is faster and easier. You should try to look at the code carefully first!

The second common way to find errors is to insert well placed print statements into your code. Print statements can easily show how far your code has gotten, and show values of variables. It also works well because you have to read the code to decide where to put the print statements.

The third way to find bugs is with the debugger. This way seems like the most fun, but often takes longer to find the bug than the more traditional code reading or trace print statements.

The most common way to find bugs with a debugger is to set breakpoints. By setting a breakpoint, the debugger will stop your program when it tries to execute that statement. By setting breakpoints on statements just before and after the spot where the program seems to be going wrong, and then examining the values of variables after the breakpoint hits, you can often find out what is going wrong, and then fix the logic error in the code.

Besides examining variable values at a breakpoint, you can also step through the code a statement at at time. This will show you exactly which statements your program is executing.

As valuable as a debugger is for finding logic errors, it often is a real time waster. It is very easy to waste lots of time setting breakpoints where there is no error, or single stepping through perfectly correct code. Even so, there are many bugs that just can't be found without a debugger.

VIDE supports the standard GNU **gdb** and Sun **jdb** debuggers. The VIDE interface to the debuggers makes it far easier to debug your code, but is of minimalist design. The goal is to make using the native debuggers as easy as possible for casual users, while maintaining the full power of the debugger for experienced users. VIDE accomplishes this by showing a command window interface to the debugger. You can enter any native debugger command in this window, and thus have full access to all debugger features.

VIDE makes using the debugger easier by providing a dialog box with the most often used commands. And most importantly, VIDE will open the source file in an editor window and highlight the current execution line on breakpoints or steps. It is very easy to trace program execution by setting breakpoints, and clicking on the *Step over* or *Step into* debug dialog buttons. VIDE also allows you to inspect variable values by highlighting the variable in the source and right clicking the mouse.

A description of debug dialog commands and tool bar buttons is provided in the VIDE Command Reference section.

Debugging Java with jdb top

To effectively use jdb, you need a pretty good understanding of Java. One of the most confusing aspects of using jdb is threads. Many java apps, especially Java GUI apps, use threads, and this can lead to some confusion of just what you are debugging, and what will be displayed. This is just part of using Java.

To debug Java programs with jdb, you must first compile the program with debugging information. This is accomplished with the **−g** switch on the compile line. The current version of VIDE does not provide automatic generation of debug or release versions. The easiest way to define VIDE projects for both debug and release versions is to use the **Project:Save Project as...** command. First, define a release version of the project. Then, using that project as a template, change the switches as needed for your debug version, and save the project under a different name.

The full power of jdb is available in the debugger command window. You may enter any standard jdb command after the ">" prompt. In fact, there really is limited interaction between VIDE and jdb, mostly handling breakpoints. VIDE maintains its own list of breakpoints, which it keeps even if you start and stop the debugger. It is important that you use VIDE commands to set and delete breakpoints. If you enter breakpoints directly into the jdb command window, VIDE won't know about them, and won't highlight them in your source code.

Applets vs. Apps

Depending on whether you are debugging a Java app or a Java applet, you must start jdb differently. The standard way to start jdb from a command line is `jdb AppName`. To debug an applet, you would start jdb with `appletviewer −debug Applet.html`. What does this mean to you? It means when you want to debug an applet, you need to use the Java Project Editor (Project−>Edit) and set the debugger name to "appletviewer −debug" on the Debugger tab. Normally, the debugger name is set to "jdb". If you created an applet project initially, this will be done automatically. Note that the appletviewer with the debugger is likely to take a *long* time to begin execution. Be patient.

Limitations with jdb

VIDE works quite well with jdb. In fact, it extends some of the capabilities by allowing you to delete all breakpoints at once.

VIDE Help System top

The VIDE help system has been designed to provide an excellent help environment for the programmer. The help files are supplied in HTML format. Different files are supplied depending on which VIDE distribution you have. The entire set of help files are available in a separate download from the ObjectCentral web site. V Help uses your default Web Browser to show the help files.

There are several help topics available from the VIDE help menu.

VIDE

This is the documentation for VIDE, including this file. It is included with all versions of VIDE.

Editor Command Set

This will display a dialog with a summary of the commands available for the current editor emulation. It is not an HTML file, but is internal to VIDE.

VIDE Help System

This will display the full VIDE Help System, which includes the best free documentation on GNU utilities, the GNU gcc compiler, C++, and HTML. The actual content for this help is available only as a separate download from the ObjectCentral web site. Clicking on this help before it is has been installed shows an HTML file with instructions for downloading the full version.

Win32 API

This item will bring up an HTML file telling how to download a Windows .HLP file that contains the WIN32 API reference. After you've installed that help file, this item will show it.

V GUI

This will show the V Reference Manual if you have downloaded the full V GUI distribution.

Java JDK

If you have installed the Sun JDK documentation, and set the Java Help path in the Options menu, this will bring up the entire Sun JDK help documentation in your browser.

HTML

This will bring up an HTML reference manual if it is installed. The HTML reference is normally included in the VIDE/Java distribution.

HTML − CSS

This will bring up an HTML reference manual if it is installed. The HTML reference is normally included in the VIDE/Java distribution.

No Warranty top

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

Bruce E. Wampler, Ph.D.
bruce@objectcentral.com
www.objectcentral.com

VIDE User Guide

The Borland C++ Compiler 5.5

This document is intended to help you use the free version of the Borland C++ compiler with VIDE. The Borland compiler is a good compiler, but the free version has some serious deficiencies. VIDE helps make up for many of the problems, but the most serious is the lack of a debugger. This issue is not likely to be resolved soon. The free version is also a bit thin on its documentation. This situation is likely to improve, and this document is a small step in that direction.

Getting and Using the Free Borland C++ Compiler 5.5 top

The Borland C++ Compiler 5.5 is available at http://www.borland.com/bcppbuilder/freecompiler/. You have to fill out a bunch of web forms, and eventually get to download the compiler. The download is about 8 Megabytes long.

Once you download, you have to install the compiler. The download is a self−extracting installer. It is probably a good idea to install it in the default location: `c:\borland\bcc55`. Once you've installed it, you need to read the `README.TXT` file.

Setting up BCC 5.5 top

Configuration Files

The most important thing you need to do is set up two configuration files on the `\bin` directory. The Borland instructions don't make that location clear. Assuming you installed the compiler to the default locations, you need to create two files. The first, `C:\Borland\BCC55\bin\bcc32.cfg` should contain:

```
−I"c:\Borland\Bcc55\include"
−L"c:\Borland\Bcc55\lib;c:\Borland\Bcc55\lib\psdk"
```

The second, `C:\Borland\BCC55\bin\ilink32.cfg` should contain:

```
−L"c:\Borland\Bcc55\lib;c:\Borland\Bcc55\lib\psdk"
```

The purpose of these two files is to allow the compiler to find the standard system include and library files. Note that the Borland README.TXT leaves out the `\psdk` entry. If you leave that out, then the compiler won't be able to find all the standard Windows API files contained there.

Also note that you can add other entries to these files to change the default behavior of the compiler.

For example, you might want to add `-wuse-` to `bcc32.cfg` to stop the compiler from issuing warnings about variables that are declared but never used. See the next section on specific switches recommended for VIDE.

Environment Path

In addiion to these two configuration files, you need to add the compiler `\bin` directory to the PATH environment variable. On Windows9x, you edit the file `C:\autoexec.bat`. Simply add `c:\borland\bcc55\bin` to the `PATH` command. On NT, you use the system settings menu off the Start menu to change the PATH in the environment. Note that if you are using VIDE, you will need to have the VIDE directory on your path, too.

General Notes

Note that many switches can be negated by following it with a '-'. For example, '-v-' means no debugging information.

If you want to make any of these switches the default behavior, you can add them to the BCC32.CFG and ILINK32.CFG files in the `/bin` directory of the Borland command line tools.

Using Borland C++ with VIDE top

VIDE hides most of the details of using the command line tools from you. However, underneath it all, the command line tools are still there. This section explains some of the details of using VIDE with BCC32.

Borland Configuration files

It is *essential* that you have the two compiler `.cfg` files set up in the `\bin` directory. The following files are suggested:

bcc32.cfg

```
-w
-I"c:\Borland\Bcc55\include"
-L"c:\Borland\Bcc55\lib;c:\Borland\Bcc55\lib\psdk"
```

The '-w' switch turn on warnings. You might want to refine the with some '-wxxx-' switches to supress some of the warnings.

ilink32.cfg

```
-x
-L"c:\Borland\Bcc55\lib;c:\Borland\Bcc55\lib\psdk"
```

The '-x' switch turn off the map file. If you want to supress incremental linking, you can add the

'−Gn' switch.

VIDE Options

To use VIDE with the Borland compiler, you *MUST* set the path to the root of the compiler directory in the **Options−>VIDE** dialog. You should set the `Borland root:` value to the directory of the Borland compiler (not the \bin directory). If you installed BCC32 to the default directory, then this would be `c:\Borland\bcc55`.

Default Project Values

Depending on whether you generate GUI or a Console application, the VIDE project file sets some default values. These are visible in the **Project−>Edit** project editor dialog.

The default compiler flags look like: `−P −O1 −v−`. The `−P` switch means C++ files, `−O1` is optimization for size, and `−v−` turns off debugging. Remember that you may have other switches in the `bcc32.cfg` file.

The linker flags line looks like: `−v− −Tpe −ap −c −limport32 −l$(BCC32RTLIB)`. These switches control the linker, and may change depending if you have a Console or GUI app. The last two values are the names of the run time libraries needed. `Import32` is always needed, and the other, `BCC32RTLIB` is the It can be a static or dynamic version, and `cw32.lib` static version is used by default.

The linker also must include a startup object code file, which varies for GUI ("c0w32.obj") and console apps ("c0x32.obj"). There are also wide−char versions of these two startup libraries. You can override the defaults by changing the value of `BCC32STARTUP` in the project editor.

Runtime Libraries

BCC32 comes with 4 runtime libraries. There are single threaded and multithreade versions, and a static and dynamic version of each. The default library is "cw32.lib", the single−threaded static library. You can use the dyanmic version of this library by changing the value of `BCC32RTLIB` in the advanced tab of the project editor to "cw32i" (no .lib, which is added automatically by VIDE). You also must either add the `−D_RTLDLL` define from the defines tab, or add the `−tWR` switch to the compiler flags line, and recompile your program. You can do switch to the static multi−threaded library ("cw32mt") or dynamic library ("cw32mti") in a similar fashion.

Specifying Libraries

VIDE allows you to specify libraries to link with on the **Linker flags** line of the Names tab of the project editor. This line is used for linker flags, *and* the names of libraries you need to add. You can see the default `−limport32 −l$(BCC32RTLIB)` when you create a new project. You can add your own library names to this line, preferably *before* the `−limport32` entry. This `−l` syntax is not part of the Borland command line options, but is converted by VIDE to the form appropriate in the

generated Makefile.

BCC32 Quick Reference

BCC32.EXE Switches top

+filename

Use alternate configuration file named filename

@filename

Read compiler options from the response file filename

−3

Generate 80386 protected−mode compatible instructions. (Default for 32−bit compiler)

−4

Generate 80386/80486 protected−mode compatible instructions.

−5

Generate Pentium protected−mode compatible instructions.

−6

Generate Pentium Pro protected−mode compatible instructions.

−a

Default (−a4) data alignment; −a− is byte.

−an

Align to n. 1=byte, 2=word (2 bytes), 4=double word (default), 8=quad word (8 bytes), 16=paragraph (16 bytes)

−A

Use only ANSI keywords. (Extensions like the far and near modifier no longer recognized.)

−A− *(Default)*

Enable Borland C++ keyword extensions: near, far, huge, asm, cdecl, pascal, interrupt, _export, _ds, _cs, _ss, _es.

−AK

Use only KRkeywords.

−AT

Use Borland C++ keywords (Alternately specified by −A−)

−AU

Use UNIX V keywords. (Extensions like the far and near modifier no longer recognized.)

−b

Make enums always integer−sized. (Default: −b make enums integer size)

−B

Compiles assembly and calls TASM or TASM32. If you don't have TASM in your path, checking this option generates an error. Also, old versions of TASM might have problems with 32−bit generated assembler code.

−c

Compileource files, but does not execute a link command.

−C

Turn nested comments on. (Default: −C− turn nested comments off.)

−d

Merge duplicate strings. (Default)

−Didentifier

Define identifier to the null string.

−Didentifier=string

Define identifier to string.

−efilename

Derives the executable program's name from filename by adding the file extension .EXE (the program name is then filename.EXE). filename must immediately follow the −e, with no intervening whitespace. Without this option, the linker derives the

.EXE file's name from the name of the first source or object file in the file name list.

−Efilename

Use filename as the name of the assembler to use. (Default = TASM)

−f

Emulate floating point. (Default)

−f−

No Floating Point

−ff

Fast floating point. (Default)

−F

Uses fast huge pointers.

−Ff

Create far variables automatically.

−Ff=1

Array variable 'identifier' is near warning. (Default)

−Fm

Enables all the other −F options (−Fc, −Ff, and −Fs). Use this to quickly port code from other 16−bit compilers.

−gb

Stop batch compilatoin after first file with warnings (Default: −gb−).

−gn

Stop compiling after n messages. (Default: 255.)

−G

Optimize code for speed. (Default: −G− optimize code for size.)

−H

Generate and use precompiled headers. It might be called BC32DEF.CSM.

−H−*(Default)*

Does not generate and use precompiled headers.

−Hfilename

Sets the name of the file for precompiled headers

−H=filename

Set the name of the file for precompiled headers to filename.

−Hc

Cache precompiled headers. Use with −H, −Hxxx, −Hu, or −Hfilename. This option is useful when compiling more than one precompiled header.

−Hu

Use but do not generate precompiled headers.

−in

Make significant identifier length to be n, where n is between 8 and 250. (Default = 250)

−Ipath

Set search path for directories for include files to path.

−jb

Stop batch compilation after first file with errors. (default: off)

−jn

Errors: stop after n messages. (Default = 25)

−Ja

Expand all template members, including unused members.

−Jg

Generate definitions for all template instances and merge duplicates. (Default)

−Jgd

Generate public definitions for all template instances; duplicates result in redefinition errors.

−Jgx

Generate external references for all template instances.

−k

Turn on standard stack frame. (Default)

−k−

Turn off standard stack frame. Generates smaller code, but it can't be easily debugged.

−K

Default character type unsigned. (Default: −K− default character type signed.)

−lx

Pass option x to the linker. More than one option can appear after the −l (which is a lowercase L).

−l−x

Disable option x for linker.

−Lpath

Set search path for library files.

−M

Instruct linker to create a full link map.

−npath

Set the output directory to path.

−O

Optimize jumps. (Default: on)

−O1

Generate smallest possible code.

−O2

Generate fastest possible code.

−Od

Disable all optimizations.

−Ox

There are a bunch of tiny optimizations, but it is probably only necessary to used −O1 and −O2, so they are not covered here.

−OS

Pentium instruction schedluling. (Default: off: −O−S)

−p

Use Pascal calling convention. (This is a lowercase p.)

−pc

Use C calling convention. (Default same as −pc or −p−)

−pr

Use fastcall calling convention for passing parameters in registers.

−ps

Use stdcall calling convention (32−bit compiler only).

−P

Perform a C++ compile regardless of source file extension. (Default when extension is not specified. This is an uppercase P.)

−Pext

Perform a C++ compile regardless of source file extension and set the default extension to ext. This option is available because some programmers use .C or another extension as their default extension for C++ code.

−q

Quiet − suppress compiler banner.

−r

Use register variables. (Default)

−rd

Allow only declared register variables to be kept in registers.

−R

Include browser information in generated .OBJ files.

−RT

Enable runtime type information. (Default)

−S

Generate assembler source compiles the named source files and produces assembly language output files (.ASM), but does not assemble. When you use this option, Borland C++ includes the C or C++ source lines as comments in the produced .ASM file.

−tW

Make the target a Windows .EXE with all functions exportable. (Default)

−tWC

Make the target a console .EXE.

−tWD

Make the target a Windows .DLL with all functions exportable.

−tWM

Make a multithreaded application or DLL.

−tWR

Target uses the dynamic runtime lib. Can use −D_RTLDLL instead.

−T−

Remove all previous assembler options.

−Tstring

Pass string as an option to TASM, TASM32, or assembler specified with −E.

−u

Generate underscores for symbols. (Default)

−Uname

Undefines any previous definitions of the named identifier name.

−v

Turn on source debugging.

−vi

Turn on inline expansion (−vi− turns off inline expansion).

−V

Create smart C++ virtual tables. (Default) This means the .objs are compatible only with Borland tools. The V0 and V1 can apparently be used with other tools, but why?

−V0

Create external C++ virtual tables.

−V1

Create public C++ virtual tables.

−Vmd

Use the smallest representation for member pointers.

−Vmm

Member pointers support multiple inheritance.

−Vmp

Honor the declared precision for all member pointer types.

−Vms

Member pointers support single inheritance.

−Vmv

Member pointers have no restrictions (most general representation). (Default)

−Vx

Zero−length empty class member functions.

−w

Display warnings on.

−w!

Do not compile to .OBJ if warnings were found. (Note: there is no space between the

−w and the !.)

−wmsg

Enable user define #pragma messages.

−w−xxx

Disable xxx warning message.

−wxxx

Enable xxx warning message.
amb −Ambiguous operators need parentheses.
amp −Superfluous with function.
asm − Unknown assembler instruction.
aus − 'identifier' is assigned a value that is never used. (Default)
bbf − Bit fields must be signed or unsigned int.
bei − Initializing 'identifier' with 'identifier'. (Default)
big − Hexadecimal value contains more than three digits. (Default)
ccc − Condition is always true OR Condition is always false. (Default)
cln − Constant is long.
cpt − Nonportable pointer comparison. (Default)
def − Possible use of 'identifier' before definition.
dpu − Declare type 'type' prior to use in prototype (Default)
dsz − Array size for 'delete' ignored. (Default)
dup − Redefinition of 'macro' is not identical (Default)
eas − 'type' assigned to 'enumeration'. (Default)
eff − Code has no effect. (Default)
ext − 'identifier' is declared as both external and static (Default)
hch − Handler for '<type1>' Hidden by Previous Handler for '<type2>'
hid − 'function1' hides virtual function 'function2' (Default)
ibc − Base class '<base1>' is also a base class of '<base2>' (Default)
ill − Ill−formed pragma. (Default)
inl − Functions containing reserved words are not expanded inline (Default)
lin − Temporary used to initialize 'identifier'. (Default)
lvc − Temporary used for parameter 'parameter' in call to 'function' (Default)
mpc − Conversion to type fails for members of virtual base class base. (Default)
mpd − Maximum precision used for member pointer type type. (Default)
msg − User−defined warnings . This option allows user−defined messages to appear in the IDE message window.
nak − Non−ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement for ANSI conformance.)
ncf − Non−const function 'function' called for const object. (Default)
nci − The constant member 'identifier' is not initialized. (Default)
nod − No declaration for function 'function'
nsf − Declaration of static function 'func(...)' ignored.
nst − Use qualified name to access nested type 'type' (Default)
ntd − Use '> >' for nested templates instead of '>>'. (Default)
nvf − Non−volatile function function called for volatile object. (Default)
obi − Base initialization without a class name is now obsolete (Default)

obs − Identifier is obsolete. (Default)
ofp − Style of function definition is now obsolete. (Default)
ovl − Overload is now unnecessary and obsolete. (Default)
par − Parameter 'parameter' is never used. (Default)
pch − Cannot create precompiled header: header. (Default)
pia − Possibly incorrect assignment. (Default)
pin − Initialization is only partially bracketed.
pre − Overloaded prefix operator 'operator' used as a postfix operator.
pro − Call to function with no prototype. (Default)
rch − Unreachable code. (Default)
ret − Both return and return of a value used. (Default)
rng − Constant out of range in comparison. (Default)
rpt − Nonportable pointer conversion. (Default)
rvl − Function should return a value. (Default)
sig − Conversion may lose significant digits.
stu − Undefined structure 'structure'
stv − Structure passed by value.
sus − Suspicious pointer conversion. (Default)
ucp − Mixing pointers to different 'char' types.
use − 'identifier' declared but never used.
voi − Void functions may not return a value. (Default)
xxx − Enable xxx warning message. (Default)
zdi − Division by zero (Default)

−W

Creates a Windows GUI application. (same as −tW)

−WC

Creates a 32−bit console mode application. (same as −tWC)

−WD

Creates a GUI DLL with all functions exportable. (same as −tWD)

−WM

Make a multithreaded application or DLL. (same as −tWM)

−WU

Generates Unicode application. Uses −txxxx macros in tchar.h.

−x

Enable exception handling. (Default)

−xd

Enable destructor cleanup. (Default)

−xf

Enable fast exception prologs.

−xp

Enable exception location information.

−xp

Enable slow exception epilogues.

−X

Disable compiler autodependency output. (Default: −X− use compiler autodependency output.)

−y

Line numbers on.

−zAname

Code class set to name.

−zBname

BSS class set to name.

−zCname

Code segment class set to name.

−zDname

BSS segment set to name.

−zGname

BSS group set to name.

−zPname

Code group set to name.

−zRname

Data segment set to name.

−zSname

Data group set to name.

−zTname

Data class set to name.

−zX*

Use default name for X. For example, −zA assigns the default class name CODE to the code segment class.

−Z

Enable register load suppression optimization.

ILINK32.EXE − Switches top

ILINK32 objfiles, exefile, mapfile, libfiles, deffile, resfiles

@xxxx indicates use response file xxxx

−ax

Specify application type (known x's follow)

−aa

Generate a protected−mode executable that runs using the 32−bit Windows API

−ap

Generate a protected−mode executable file that runs in console mode

−Ao:nnnn

Specify object alignment

−b:xxxx

Specify image base addr

−c

Case sensitive linking

−C

Clear state before linking

−Dstring

Set image description

−d

Delay load a .DLL

−Enn

Max number of errors

−Gi

Generate import library

−Gl

Static package

−Gn

No state files

−Gpd

Design time only package

−Gpr

Runtime only package

−Gt

Fast TLS

−Gz

Do image checksum

−GC

Specify image comment string

−GD

Generate .DRC file

−GF

Set image flags

−GS

Set section flags

−H:xxxx

Specify heap reserve size

−Hc:xxxx

Specify heap commit size

−I

Intermediate output dir

−j

Specify object search paths

−L

Specify library search paths

−m

Map file with publics

−M

Map with mangled names

−q

Supress banner

−r

Verbose linking

−Rr

Replace resources

−s

Detailed segment map

−S:xxxx

Specify stack reserve size

−Sc:xxxx

Specify stack commit size

−Txx

Display time spent on link

−Txx

Specify output file type

−Tpd

Target a Windows .DLL file

−Tpe

Target a Windows .EXE file

−Tpp

Generate package

−Ud.d

Specify image user version

−v

Full debug information

−Vd.d

Specify subsystem version

−w−

Dilable all warnings.

−wxxx

Warning control
def − No .DEF file
dpl − Duplicate symbol in lib
imt − Import does not match previous definition
msk − Multiple stack segment
bdk − using based linking in .DLL
dll − .EXE module built with .DLL extension
dup − Duplicate symbol
ent − No entry point

inq − Extern not qualified with __import
srf − Self−relative fixup overflow
stk − No stack

−x

No map

BCC32 Libraries top

The following is an incomplete listing of the main startup and runtime libraries included with BCC32. If you have more details, please send them, and I will include them here.

OBJ Files

C0D32.OBJ

32−bit DLL startup module (cod32w: wide−char version; cod32x: no exception handling)

C0S32.OBJ

Unknown

C0W32.OBJ

32−bit GUI EXE startup module (c0w32w: wide−char)

C0X32.OBJ

32−bit console−mode EXE startup module (c0x32w: wide−char)

FILEINFO.OBJ

Passes open file−handle information to child processes. Include this file in your link to pass full information about open files to child processes created with exec and spawn. Works with the C++ runtime library to inherit this information.

GP.OBJ

Prints register−dump information when an exception occurs.

WILDARGS.OBJ

If you want wild−card expansion for you console−mode applications, then you should also link in this file when you link your console−mode application. It apparently doesn't work for GUI apps. It does the normal DOS−like wild card expansion and passes them to argv and argc of your main.

LIB Files

CW32.LIB

32−bit single−threaded static library

CW32I.LIB

32−bit single−thread, dynamic RTL import library for CW3250.DLL. To use this import library, you must compile your programs with either −D_RTDLL or −tWR options to the compiler. This probably applies to the other "i" libs as well.

CW32MT.LIB

32−bit multithread static library

CW32MTI.LIB

Import lib for 32−bit multithread dynamic RTL import library for CW3250MT.DLL

IMPORT32.LIB

32−bit import library

dxextra.lib

DirectX static library

inet.lib

Import lib for MS Internet DLLs

noeh32.lib

No exception handling support lib

ole2w32.lib

Import lib for 32−bit OLE 2.0 API

oleaut32.lib

Unknown

uuid.lib

GUID static lib for Direct3d, DirectDraw, DirectSound, Shell extensions, DAO, Active Scripting, etc.

wininet.lib

Unknown

ws2_32.lib

Import lib for WinSock 2.0 API

Help Improve VIDE for BCC top

Please note that VIDE will reamain primarily focused on the GNU MinGW gcc compiler. However, I do plan to continue support for the free Borland BCC 5.5. Remember that VIDE is GPLed, so the code is avaialble for modification.

First, if I've gotten some default behavior wrong, please suggest a reasonable alternative. But remember, I chose the defaults here mainly to simplify typical, simple applications. Advanced users are expected to edit the Project file or makefile to get more options.

If you don't like the layout of this document (like tables might be better), chip in and fix it. I'll fold it back into the standard distriubtion.

The BIG hole is the missing debugger. But the tools emit debugging information. That mean it would be possible to write a free debugger to go with this compiler. I have some ideas on how to approach such a project, but no time to do it. I think even a very simple interface that just had breakpoints and variable value inspection would be a GIANT improvement. If you are interested in this, send me some e−mail, and I'll be happy to discuss the project with you.

Disclaimer top

First, the Borland C++ 5.5 compiler is not the main compiler supported by VIDE (gcc is). I tried to get most of the stuff right with the first release (1.07), but there may be glitches. Help me out, and report them. I'll try to fix them a soon as I can.

This information was assembled for publicly available sources and is intended merely to help use VIDE with Borland BCC32. There is no guarantee of its accuracy, although it seems to be correct, but may be incomplete. Last revision: 3−4−00.

No Warranty top

This program is provided on an "as is" basis, without warranty of any kind. The entire risk as to the quality and performance of the program is borne by you.

Bruce E. Wampler, Ph.D.
bruce@objectcentral.com

www.objectcentral.com