Systems, Tools, and Terminal Science

BLOG ARCHIVES

Unix as IDE: Introduction

Posted on February 9, 2012

This series has been independently translated into Chinese and Russian, and formatted as an ebook.

Newbies and experienced professional programmers alike appreciate the concept of the IDE, or integrated development environment. Having the primary tools necessary for organising, writing, maintaining, testing, and debugging code in an integrated application with common interfaces for all the different tools is certainly a very valuable asset. Additionally, an environment expressly designed for programming in various languages affords advantages such as autocompletion, and syntax checking and highlighting.

With such tools available to developers on all major desktop operating systems including Linux and BSD, and with many of the best free of charge, there's not really a good reason to write your code in Windows Notepad, or with `nano` or `cat`.

However, there's a minor meme among devotees of Unix and its modern-day derivatives that "Unix is an IDE", meaning that the tools available to developers on the terminal cover the major features in cutting-edge desktop IDEs with some ease. Opinion is quite divided on this, but whether or not you feel it's fair to call Unix an IDE in the same sense as Eclipse or Microsoft Visual Studio, it may surprise you just how comprehensive a development environment the humble Bash shell can be.

How is UNIX an IDE?

The primary rationale for using an IDE is that it gathers all your tools in the same place, and you can use them in concert with roughly the same user interface paradigm, and without having to exert too much effort to make separate applications cooperate. The reason this becomes especially desirable with GUI applications is because it's very difficult to make windowed applications speak a common language or work well with each other; aside from cutting and pasting text, they don't share a *common interface*.

The interesting thing about this problem for shell users is that well-designed and enduring Unix tools already share a common user interface in *streams of text* and *files as persistent objects*, otherwise expressed in the axiom "everything's a file". Pretty much everything in Unix is built around these two concepts, and it's this common user interface, coupled with a forty-year history of high-powered tools whose users and developers have especially prized interoperability, that goes a long way to making Unix as powerful as a full-blown IDE.

The right idea

This attitude isn't the preserve of battle-hardened Unix greybeards; you can see it in another form in the way the modern incarnations of the two grand old text editors Emacs and Vi (GNU Emacs and Vim) have such active communities developing plugins to make them support pretty much any kind of editing task. There are plugins to do pretty much anything you could really want to do in programming in both editors, and like any Vim junkie I could spout off at least six or seven that I feel are "essential".

However, it often becomes apparent to me when reading about these efforts that the developers concerned are trying to make these text editors into IDEs in their own right. There are posts about never needing to leave Vim, or never needing to leave Emacs. But I think that trying to shoehorn Vim or Emacs into becoming something that it's not isn't quite thinking about the problem in the right way. Bram Moolenaar, the author of Vim, appears to agree to some extent, as you can see by reading `:help design-not`. The shell is only ever a Ctrl+Z away, and its mature, highly composable toolset will afford you more power than either editor ever could.

About this series

In this series of posts, I will be going through six major features of an IDE, and giving examples showing how common tools available in Linux allow you to use them together with ease. This will by no means be a comprehensive survey, nor are the tools I will demonstrate the only options.

- **File and project management** — `ls`, `find`, `grep`/`ack`, `bash`
- **Text editor and editing tools** — `vim`, `awk`, `sort`, `column`
- **Compiler and/or interpreter** — `gcc`, `perl`
- **Build tools** — `make`
- **Debugger** — `gdb`, `valgrind`, `ltrace`, `lsof`, `pmap`
- **Version control** — `diff`, `patch`, `svn`, `git`

What I'm not trying to say

I don't think IDEs are bad; I think they're brilliant, which is why I'm trying to convince you that Unix can be used as one, or at least thought of as one. I'm also not going to say that Unix is always the best tool for any programming task; it is arguably much better suited for C, C++, Python, Perl, or Shell development than it is for more "industry" languages like Java or C#, especially if writing GUI-heavy applications. In particular, I'm not going to try to convince you to scrap your hard-won Eclipse or Microsoft Visual Studio knowledge for the sometimes esoteric world of the command line. All I want to do is show you what we're doing on the other side of the fence.

This entry is part 1 of 7 in the series Unix as IDE.

Posted in **Linux** | Tagged **development**, **environment**, **ide**, **integrated development environment**, **programming**, **tools**, **unix**

Unix as IDE: Files

Posted on **February 10, 2012**

One prominent feature of an IDE is a built-in system for managing files, both the elementary functions like moving, renaming, and deleting, and ones more specific to development, like compiling and checking syntax. It may also be useful to have operations on sets of files, such as finding files of a certain extension or size, or searching files for specific patterns. In this first article, I'll explore some useful ways to use tools that will be familiar to most Linux users for the purposes of working with sets of files in a project.

Listing files

Using `ls` is probably one of the first commands an administrator will learn for getting a simple list of the contents of the directory. Most administrators will also know about the `-a` and `-l` switches, to show all files including dot files and to show more detailed data about files in columns, respectively.

There are other switches to GNU `ls` which are less frequently used, some of which turn out to be very useful for programming:

- `-t` — List files in order of last modification date, newest first. This is useful for very large directories when you want to get a quick list of the most recent files changed, maybe piped through `head` or `sed 10q`. Probably most useful combined with `-l`. If you want the *oldest* files, you can add `-r` to reverse the list.
- `-X` — Group files by extension; handy for polyglot code, to group header files and source files separately, or to separate source files from directories or build files.
- `-v` — Naturally sort version numbers in filenames.
- `-S` — Sort by filesize.
- `-R` — List files recursively. This one is good combined with `-l` and pipedthrough a pager like `less`.

Since the listing is text like anything else, you could, for example, pipe the output of this command into a `vim` process, so you could add explanations of what each file is for and save it as an `inventory` file or add it to a README:

```
$ ls -XR | vim -
```

This kind of stuff can even be automated by `make` with a little work, which I'll cover in another article later in the series.

Finding files

Funnily enough, you can get a complete list of files including relative paths with no decoration by simply typing `find` with no arguments, though it's usually a good idea to pipe it through `sort`:

```
$ find | sort
.
./Makefile
./README
./build
./client.c
./client.h
./common.h
./project.c
./server.c
./server.h
./tests
./tests/suite1.pl
./tests/suite2.pl
./tests/suite3.pl
./tests/suite4.pl
```

If you want an `ls -l` style listing, you can add `-ls` as the action to `find` results:

```
$ find -ls | sort -k 11
1155096    4 drwxr-xr-x   4 tom      tom          4096 Feb 10 09:37 .
1155152    4 drwxr-xr-x   2 tom      tom          4096 Feb 10 09:17 .
1155155    4 -rw-r--r--   1 tom      tom          2290 Jan 11 07:21 .
1155157    4 -rw-r--r--   1 tom      tom          1871 Jan 11 16:41 .
1155159   32 -rw-r--r--   1 tom      tom         30390 Jan 10 15:29 .
1155153   24 -rw-r--r--   1 tom      tom         21170 Jan 11 05:43 .
1155154   16 -rw-r--r--   1 tom      tom         13966 Jan 14 07:39 .
1155080   28 -rw-r--r--   1 tom      tom         25840 Jan 15 22:28 .
1155156   32 -rw-r--r--   1 tom      tom         31124 Jan 11 02:34 .
1155158    4 -rw-r--r--   1 tom      tom          3599 Jan 16 05:27 .
1155160    4 drwxr-xr-x   2 tom      tom          4096 Feb 10 09:29 .
1155161    4 -rw-r--r--   1 tom      tom           288 Jan 13 03:04 .
1155162    4 -rw-r--r--   1 tom      tom          1792 Jan 13 10:06 .
1155163    4 -rw-r--r--   1 tom      tom           112 Jan  9 23:42 .
1155164    4 -rw-r--r--   1 tom      tom           144 Jan 15 02:10 .
```

Note that in this case I have to specify to `sort` that it should sort by the 11th column of output, the filenames; this is done with the `-k` option.

`find` has a complex filtering syntax all of its own; the following examples show some of the most useful filters you can apply to retrieve lists of certain files:

- `find -name '*.c'` — Find files with names matching a shell-style pattern. Use `-iname` for a case-insensitive search.
- `find -path '*test*'` — Find files with paths matching a shell-style pattern. Use `-ipath` for a case-insensitive search.
- `find -mtime -5` — Find files edited within the last five days. You can use `+5` instead to find files edited *before* five days ago.
- `find -newer server.c` — Find files more recently modified than `server.c`.
- `find -type d` — Find directories. For files, use `-type f`; for symbolic links, use `-type l`.

Note, in particular, that all of these can be combined, for example to find C source files edited in the last two days:

```
$ find -name '*.c' -mtime -2
```

By default, the action `find` takes for search results is simply to list them on standard output, but there are several other useful actions:

- `-ls` — Provide an `ls -l` style listing, as above
- `-delete` — Delete matching files
- `-exec` — Run an arbitrary command line on each file, replacing `{}` with the appropriate filename, and terminated by `\;`; for example:

  ```
  $ find -name '*.pl' -exec perl -c {} \;
  ```

  You can use `+` as the terminating character instead if you want to put all of the results on one invocation of the command. One trick I find myself using often is using `find` to generate lists of files that I then edit in vertically split Vim windows:

  ```
  $ find -name '*.c' -exec vim {} +
  ```

Earlier versions of Unix as IDE suggested the use of `xargs` with `find` results. In most cases this should not really be necessary, and it's more robust to handle filenames with whitespace using `-exec` or a `while read -r` loop.

Searching files

More often than *attributes* of a set of files, however, you want to find files based on their *contents*, and it's no surprise that `grep`, in particular `grep -R`, is useful here. This searches the current directory tree recursively for anything matching 'someVar':

```
$ grep -FR someVar .
```

Don't forget the case insensitivity flag either, since by default `grep` works with fixed case:

```
$ grep -iR somevar .
```

Also, you can print a list of files that match without printing the matches themselves with `grep -l`:

```
$ grep -lR someVar .
```

If you write scripts or batch jobs using the output of the above, use a `while` loop with `read` to handle spaces and other special characters in filenames:

```
grep -lR someVar | while IFS= read -r file; do
    head "$file"
done
```

If you're using version control for your project, this often includes metadata in the `.svn`, `.git`, or `.hg` directories. This is dealt with easily enough by *excluding* (`grep -v`) anything matching an appropriate fixed (`grep -F`) string:

```
$ grep -R someVar . | grep -vF .svn
```

Some versions of `grep` include `--exclude` and `--exclude-dir` options, which may be tidier.

With all this said, there's a very popular alternative to grep called `ack`, which excludes this sort of stuff for you by default. It also allows you to use Perl-compatible regular expressions (PCRE), which are a favourite for many hackers. It has a lot of utilities that are generally useful for working with source code, so while there's nothing wrong with good old `grep` since you know it will always be there, if you can install `ack` I highly recommend it. There's a Debian package called `ack-grep`, and being a Perl script it's otherwise very simple to install.

Unix purists might be displeased with my even mentioning a relatively new Perl script alternative to classic `grep`, but I don't believe that the Unix philosophy or using Unix as an IDE is dependent on sticking to the same classic tools when alternatives with the same spirit that solve new problems are available.

File metadata

The `file` tool gives you a one-line summary of what kind of file you're looking at, based on its extension, headers and other cues. This is very handy used with `find` when examining a set of unfamiliar files:

```
$ find -exec file {} \;
.:            directory
./hanoi:      Perl script, ASCII text executable
./.hanoi.swp: Vim swap file, version 7.3
./factorial:  Perl script, ASCII text executable
./bits.c:     C source, ASCII text
./bits:       ELF 32-bit LSB executable, Intel 80386, version ...
```

Matching files

As a final tip for this section, I'd suggest learning a bit about pattern matching and brace expansion in Bash, which you can do in my earlier post entitled Bash shell expansion.

All of the above make the classic UNIX shell into a pretty powerful means of managing files in programming projects.

This entry is part 2 of 7 in the series Unix as IDE.

Posted in **Linux** | Tagged **ack**, **batch**, **file**, **find**, **grep**, **list**, **ls**, **management**, **sort**, **unix**

Unix as IDE: Editing

Posted on **February 11, 2012**

The text editor is the core tool for any programmer, which is why choice of editor evokes such tongue-in-cheek zealotry in debate among programmers. Unix is the operating system most strongly linked with two enduring favourites, Emacs and Vi, and their modern versions in GNU Emacs and Vim, two editors with very different editing philosophies but comparable power.

Being a Vim heretic myself, here I'll discuss the indispensable features of Vim for programming, and in particular the use of Linux shell tools called from *within* Vim to complement the editor's built-in functionality. Some of the principles discussed here will be applicable to those using Emacs as well, but probably not for underpowered editors like Nano.

This will be a very general survey, as Vim's toolset for programmers is *enormous*, and it'll still end up being quite long. I'll focus on the essentials and the things I feel are most helpful, and try to provide links to articles with a more comprehensive treatment of the topic. Don't forget that Vim's `:help` has surprised many people new to the editor with its high quality and usefulness.

Filetype detection

Vim has built-in settings to adjust its behaviour, in particular its syntax highlighting, based on the filetype being loaded, which it happily detects and generally does a good job at doing so. In particular, this allows you to set an indenting style conformant with the way a particular language is usually written. This should be one of the first things in your `.vimrc` file.

```
if has("autocmd")
  filetype on
  filetype indent on
  filetype plugin on
endif
```

Syntax highlighting

Even if you're just working with a 16-color terminal, just include the following in your `.vimrc` if you're not already:

```
syntax on
```

The colorschemes with a default 16-color terminal are not pretty largely by necessity, but they do the job, and for most languages syntax definition files are available that work very well. There's a tremendous array of colorschemes available, and it's not hard to tweak them to suit or even to write your own. Using a 256-color terminal or gVim will give you more options. Good syntax highlighting files will show you definite syntax errors with a glaring red background.

Line numbering

To turn line numbers on if you use them a lot in your traditional IDE:

```
set number
```

You might like to try this as well, if you have at least Vim 7.3 and are keen to try numbering lines relative to the current line rather than absolutely:

```
set relativenumber
```

Tags files

Vim works very well with the output from the `ctags` utility. This allows you to search quickly for all uses of a particular identifier throughout the project, or to navigate straight to the declaration of a variable from one of its uses, regardless of whether it's in the same file. For large C projects in multiple files this can save huge amounts of otherwise wasted time, and is probably Vim's best answer to similar features in mainstream IDEs.

You can run `:!ctags -R` on the root directory of projects in many popular languages to generate a `tags` file filled with definitions and locations for identifiers throughout your project. Once a `tags` file for your project is available, you can search for uses of an appropriate tag throughout the project like so:

```
:tag someClass
```

The commands `:tn` and `:tp` will allow you to iterate through successive uses of the tag elsewhere in the project. The built-in tags functionality for this already covers most of the bases you'll probably need, but for features such as a tag list window, you could try installing the very popular Taglist plugin. Tim Pope's Unimpaired plugin also contains a couple of useful relevant mappings.

Calling external programs

There are two major methods of calling external programs during a Vim session:

- `:!<command>` — Useful for issuing commands from within a Vim context particularly in cases where you intend to record output in a buffer.
- `:shell` — Drop to a shell as a subprocess of Vim. Good for interactive commands.

A third, which I won't discuss in depth here, is using plugins such as Conque to emulate a shell within a Vim buffer. Having tried this myself and found it nearly unusable, I've concluded it's simply bad design. From `:help design-not`:

Vim is not a shell or an Operating System. You will not be able to run a shell inside Vim or use it to control a debugger. This should work the other way around: Use Vim as a component from a shell or in an IDE.

Lint programs and syntax checkers

Checking syntax or compiling with an external program call (e.g. `perl -c`, `gcc`) is one of the calls that's good to make from within the editor using `:!` commands. If you were editing a Perl file, you could run this like so:

```
:!perl -c %

/home/tom/project/test.pl syntax OK
```

```
Press Enter or type command to continue
```

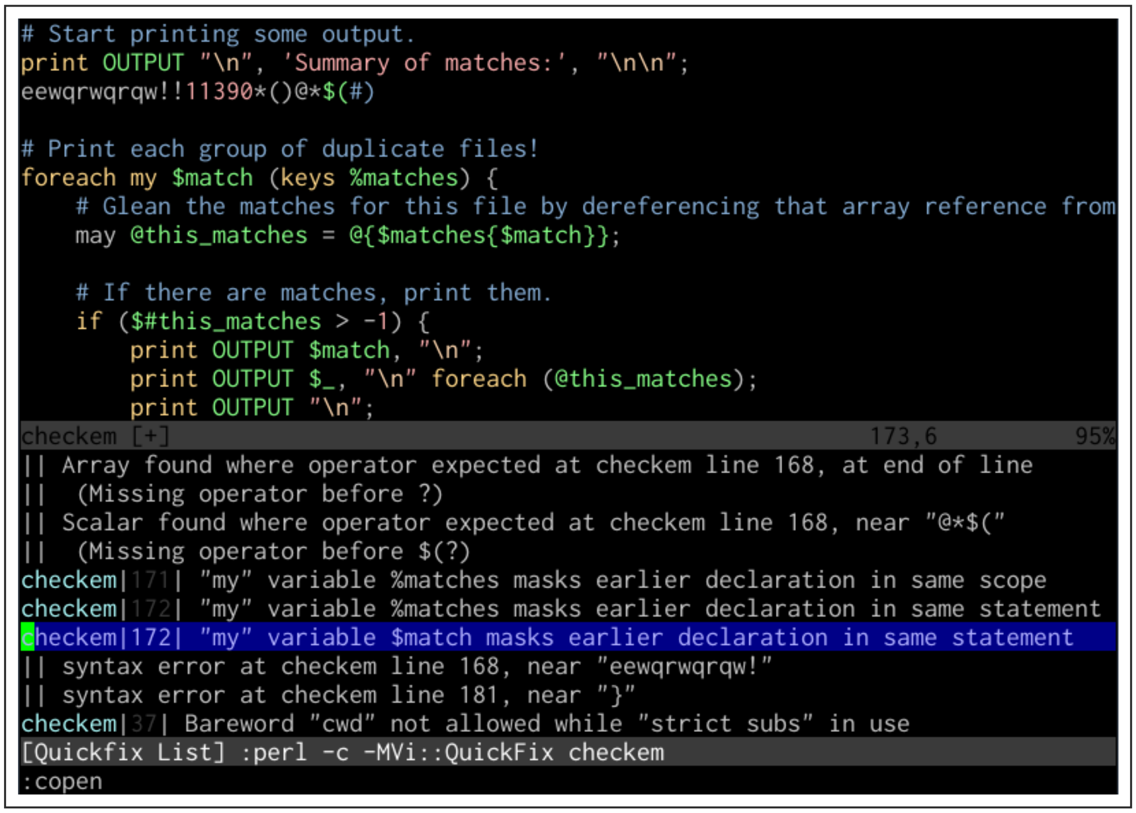
The `%` symbol is shorthand for the file loaded in the current buffer. Running this prints the output of the command, if any, below the command line. If you wanted to call this check often, you could perhaps map it as a command, or even a key combination in your `.vimrc` file. In this case, we define a command `:PerlLint` which can be called from normal mode with `\l`:

```
command PerlLint !perl -c %
nnoremap <leader>l :PerlLint<CR>
```

For a lot of languages there's an even better way to do this, though, which allows us to capitalise on Vim's built-in quickfix window. We can do this by setting an appropriate `makeprg` for the filetype, in this case including a module that provides us with output that Vim can use for its quicklist, and a definition for its two formats:

```
:set makeprg=perl\ -c\ -MVi::QuickFix\ %
:set errorformat+=%m\ at\ %f\ line\ %l\.
:set errorformat+=%m\ at\ %f\ line\ %l
```

You may need to install this module first via CPAN, or the Debian package `libvi-quickfix-perl`. This done, you can type `:make` after saving the file to check its syntax, and if errors are found, you can open the quicklist window with `:copen` to inspect the errors, and `:cn` and `:cp` to jump to them within the buffer.



```
# Start printing some output.
print OUTPUT "\n", 'Summary of matches:', "\n\n";
eewqrwqrqw!!11390*()@*$(#)

# Print each group of duplicate files!
foreach my $match (keys %matches) {
    # Glean the matches for this file by dereferencing that array reference from
    may @this_matches = @{$matches{$match}};

    # If there are matches, print them.
    if ($#this_matches > -1) {
        print OUTPUT $match, "\n";
        print OUTPUT $_, "\n" foreach (@this_matches);
        print OUTPUT "\n";
checkem [+]                                                   173,6         95%
|| Array found where operator expected at checkem line 168, at end of line
||  (Missing operator before ?)
|| Scalar found where operator expected at checkem line 168, near "@*$("
||  (Missing operator before $(?)
checkem|171| "my" variable %matches masks earlier declaration in same scope
checkem|172| "my" variable %matches masks earlier declaration in same statement
checkem|172| "my" variable $match masks earlier declaration in same statement
|| syntax error at checkem line 168, near "eewqrwqrqw!"
|| syntax error at checkem line 181, near "}"
checkem|37| Bareword "cwd" not allowed while "strict subs" in use
[Quickfix List] :perl -c -MVi::QuickFix checkem
:copen
```

— Vim quickfix working on a Perl file

This also works for output from `gcc`, and pretty much any other compiler syntax checker that you might want to use that includes filenames, line numbers, and error strings in its error output. It's even possible to do this with web-focused languages like PHP, and for tools like JSLint for JavaScript. There's also an excellent plugin named Syntastic that does something similar.

Reading output from other commands

You can use `:r!` to call commands and paste their output directly into the buffer with which you're working. For example, to pull a quick directory listing for the current folder into the buffer, you could type:

```
:r!ls
```

This doesn't just work for commands, of course; you can simply read in other files this way with just `:r`, like public keys or your own custom boilerplate:

```
:r ~/.ssh/id_rsa.pub
:r ~/dev/perl/boilerplate/copyright.pl
```

Filtering output through other commands

You can extend this to actually filter text in the buffer through external commands, perhaps selected by a range or visual mode, and replace it with the command's output. While Vim's visual block mode is great for working with columnar data, it's very often helpful to bust out tools like `column`, `cut`, `sort`, or `awk`.

For example, you could sort the entire file in reverse by the second column by typing:

```
:%!sort -k2 -r
```

You could print only the third column of some selected text where the line matches the pattern `/vim/` with:

```
:'<,'>!awk '/vim/ {print $3}'
```

You could arrange keywords from lines 1 to 10 in nicely formatted columns like:

```
:1,10!column -t
```

Really *any kind* of text filter or command can be manipulated like this in Vim, a simple interoperability feature that expands what the editor can do by an order of magnitude. It effectively makes the Vim buffer into a text stream, which is a language that all of these classic tools speak.

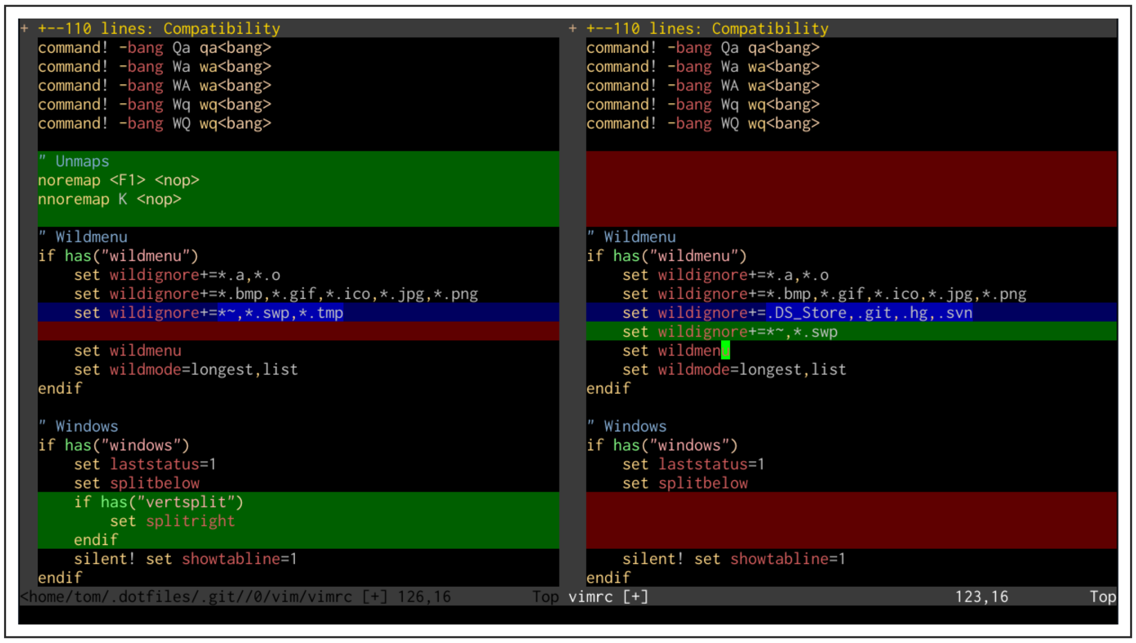
Built-in alternatives

It's worth noting that for really common operations like sorting and searching, Vim has built-in methods in `:sort` and `:grep`, which can be helpful if you're stuck using Vim on Windows, but don't have nearly the adaptability of shell calls.

Diffing

Vim has a *diffing* mode, `vimdiff`, which allows you to not only view the differences between different versions of a file, but also to resolve conflicts via a three-way merge and to replace differences to and fro with commands like `:diffput` and `:diffget` for ranges of text. You can call `vimdiff` from the command line directly with at least two files to compare like so:

```
$ vimdiff file-v1.c file-v2.c
```



```
+ +--110 lines: Compatibility
command! -bang Qa qa<bang>
command! -bang Wa wa<bang>
command! -bang WA wa<bang>
command! -bang Wq wq<bang>
command! -bang WQ wq<bang>

" Unmaps
noremap <F1> <nop>
nnoremap K <nop>

" Wildmenu
if has("wildmenu")
    set wildignore+=*.a,*.o
    set wildignore+=*.bmp,*.gif,*.ico,*.jpg,*.png
    set wildignore+=*~,*.swp,*.tmp

    set wildmenu
    set wildmode=longest,list
endif

" Windows
if has("windows")
    set laststatus=1
    set splitbelow
    if has("vertsplit")
        set splitright
    endif
    silent! set showtabline=1
endif
<home/tom/.dotfiles/.git//0/vim/vimrc [+] 126,16          Top
```

```
+ +--110 lines: Compatibility
command! -bang Qa qa<bang>
command! -bang Wa wa<bang>
command! -bang WA wa<bang>
command! -bang Wq wq<bang>
command! -bang WQ wq<bang>




" Wildmenu
if has("wildmenu")
    set wildignore+=*.a,*.o
    set wildignore+=*.bmp,*.gif,*.ico,*.jpg,*.png
    set wildignore+=.DS_Store,.git,.hg,.svn
    set wildignore+=*~,*.swp
    set wildmenu
    set wildmode=longest,list
endif

" Windows
if has("windows")
    set laststatus=1
    set splitbelow



    silent! set showtabline=1
endif
vimrc [+]                                123,16          Top
```

— Vim diffing a .vimrc file

Version control

You can call version control methods directly from within Vim, which is probably all you need most of the time. It's useful to remember here that `%` is always a shortcut for the buffer's current file:

```
:!svn status
:!svn add %
:!git commit -a
```

Recently a clear winner for Git functionality with Vim has come up with Tim Pope's Fugitive, which I highly recommend to anyone doing Git development with Vim. There'll be a more comprehensive treatment of version control's basis and history in Unix in Part 7 of this series.

The difference

Part of the reason Vim is thought of as a toy or relic by a lot of programmers used to GUI-based IDEs is its being seen as just a tool for editing files on servers, rather than a very capable editing component for the shell in its own right. Its own built-in features being so composable with external tools on Unix-friendly systems makes it into a text editing powerhouse that sometimes surprises even experienced users.

This entry is part 3 of 7 in the series Unix as IDE.

Posted in **Linux** | Tagged **commands**, **diff**, **editing**, **editor**, **external**, **tools**, **unix**, **version control**

Unix as IDE: Compiling

Posted on February 12, 2012

There are a lot of tools available for compiling and interpreting code on the Unix platform, and they tend to be used in different ways. However, conceptually many of the steps are the same. Here I'll discuss compiling C code with `gcc` from the GNU Compiler Collection, and briefly the use of `perl` as an example of an interpreter.

GCC

GCC is a very mature GPL-licensed collection of compilers, perhaps best-known for working with C and C++ programs. Its free software license and near ubiquity on free Unix-like systems like Linux and BSD has made it enduringly popular for these purposes, though more modern alternatives are available in compilers using the LLVM infrastructure, such as Clang.

The frontend binaries for GNU Compiler Collection are best thought of less as a set of complete compilers in their own right, and more as *drivers* for a set of discrete programming tools, performing parsing, compiling, and linking, among other steps. This means that while you can use GCC with a relatively simple command line to compile straight from C sources to a working binary, you can also inspect in more detail the steps it takes along the way and tweak it accordingly.

I won't be discussing the use of `make` files here, though you'll almost certainly be wanting them for any C project of more than one file; that will be discussed in the next article on build automation tools.

Compiling and assembling object code

You can compile object code from a C source file like so:

```
$ gcc -c example.c -o example.o
```

Assuming it's a valid C program, this will generate an unlinked binary object file called `example.o` in the current directory, or tell you the reasons it can't. You can inspect its assembler contents with the `objdump` tool:

```
$ objdump -D example.o
```

Alternatively, you can get `gcc` to output the appropriate assembly code for the object directly with the `-S` parameter:

```
$ gcc -c -S example.c -o example.s
```

This kind of assembly output can be particularly instructive, or at least interesting, when printed inline with the source code itself, which you can do with:

```
$ gcc -c -g -Wa,-a,-ad example.c > example.lst
```

Preprocessor

The C preprocessor `cpp` is generally used to include header files and define macros, among other things. It's a normal part of `gcc` compilation, but you can view the C code it generates by invoking `cpp` directly:

```
$ cpp example.c
```

This will print out the complete code as it would be compiled, with includes and relevant macros applied.

Linking objects

One or more objects can be linked into appropriate binaries like so:

```
$ gcc example.o -o example
```

In this example, GCC is not doing much more than abstracting a call to `ld`, the GNU linker. The command produces an executable binary called `example`.

Compiling, assembling, and linking

All of the above can be done in one step with:

```
$ gcc example.c -o example
```

This is a little simpler, but compiling objects independently turns out to have some practical performance benefits in not recompiling code unnecessarily, which I'll discuss in the next article.

Including and linking

C files and headers can be explicitly included in a compilation call with the `-I` parameter:

```
$ gcc -I/usr/include/somelib.h example.c -o example
```

Similarly, if the code needs to be dynamically linked against a compiled system library available in common locations like `/lib` or `/usr/lib`, such as `ncurses`, that can be included with the

`-l` parameter:

```
$ gcc -lncurses example.c -o example
```

If you have a lot of necessary inclusions and links in your compilation process, it makes sense to put this into environment variables:

```
$ export CFLAGS=-I/usr/include/somelib.h
$ export CLIBS=-lncurses
$ gcc $CFLAGS $CLIBS example.c -o example
```

This very common step is another thing that a `Makefile` is designed to abstract away for you.

Compilation plan

To inspect in more detail what `gcc` is doing with any call, you can add the `-v` switch to prompt it to print its compilation plan on the standard error stream:

```
$ gcc -v -c example.c -o example.o
```

If you don't want it to actually generate object files or linked binaries, it's sometimes tidier to use `-###` instead:

```
$ gcc -### -c example.c -o example.o
```

This is mostly instructive to see what steps the `gcc` binary is abstracting away for you, but in specific cases it can be useful to identify steps the compiler is taking that you may not necessarily want it to.

More verbose error checking

You can add the `-Wall` and/or `-pedantic` options to the `gcc` call to prompt it to warn you about things that may not necessarily be errors, but could be:

```
$ gcc -Wall -pedantic -c example.c -o example.o
```

This is good for including in your `Makefile` or in your `makeprg` definition in Vim, as it works well with the quickfix window discussed in the previous article and will enable you to write more readable, compatible, and less error-prone code as it warns you more extensively about errors.

Profiling compilation time

You can pass the flag `-time` to `gcc` to generate output showing how long each step is taking:

```
$ gcc -time -c example.c -o example.o
```

Optimisation

You can pass generic optimisation options to `gcc` to make it attempt to build more efficient object files and linked binaries, at the expense of compilation time. I find `-O2` is usually a happy medium for code going into production:

- `gcc -O1`
- `gcc -O2`
- `gcc -O3`

Like any other Bash command, all of this can be called from within Vim by:

```
:!gcc % -o example
```

Interpreters

The approach to interpreted code on Unix-like systems is very different. In these examples I'll use Perl, but most of these principles will be applicable to interpreted Python or Ruby code, for example.

Inline

You can run a string of Perl code directly into the interpreter in any one of the following ways, in this case printing the single line "Hello, world." to the screen, with a linebreak following. The first one is perhaps the tidiest and most standard way to work with Perl; the second uses a heredoc string, and the third a classic Unix shell pipe.

```
$ perl -e 'print "Hello world.\n";'
$ perl <<<'print "Hello world.\n";'
$ echo 'print "Hello world.\n";' | perl
```

Of course, it's more typical to keep the code in a file, which can be run directly:

```
$ perl hello.pl
```

In either case, you can check the syntax of the code without actually running it with the `-c` switch:

```
$ perl -c hello.pl
```

But to use the script as a *logical binary*, so you can invoke it directly without knowing or caring what the script is, you can add a special first line to the file called the “shebang” that does some magic to specify the interpreter through which the file should be run.

```
#!/usr/bin/env perl
print "Hello, world.\n";
```

The script then needs to be made executable with a `chmod` call. It’s also good practice to rename it to remove the extension, since it is now taking the shape of a logic binary:

```
$ mv hello{.pl,}
$ chmod +x hello
```

And can thereafter be invoked directly, as if it were a compiled binary:

```
$ ./hello
```

This works so well that many of the common utilities on modern Linux systems, such as the `adduser` frontend to `useradd`, are actually Perl or even Python scripts.

In the next post, I’ll describe the use of `make` for defining and automating building projects in a manner comparable to IDEs, with a nod to newer takes on the same idea with Ruby’s `rake`.

This entry is part 4 of 7 in the series Unix as IDE.

Posted in **Linux** | Tagged **assembler**, **compiler**, **gcc**, **interpreter**, **linker**, **perl**, **python**, **ruby**, **unix**

Unix as IDE: Building

Posted on **February 13, 2012**

Because compiling projects can be such a complicated and repetitive process, a good IDE provides a means to abstract, simplify, and even automate software builds. Unix and its descendents accomplish this process with a `Makefile`, a prescribed recipe in a standard format for generating executable files from source and object files, taking account of changes to only rebuild what’s necessary to prevent costly recompilation.

One interesting thing to note about `make` is that while it’s generally used for compiled software build automation and has many shortcuts to that effect, it can actually effectively be used for any situation in which it’s required to generate one set of files from another. One possible use is to generate web-friendly optimised graphics from source files for deployment for a website; another use is for generating static HTML pages from code, rather than generating pages on the fly. It's on the basis of this more flexible understanding of software "building" that modern takes on the tool like `Ruby's rake` have become popular, automating the general tasks for producing and installing code and files of all kinds.

Anatomy of a `Makefile`

The general pattern of a `Makefile` is a list of variables and a list of *targets*, and the sources and/or objects used to provide them. Targets may not necessarily be linked binaries; they could also constitute actions to perform using the generated files, such as `install` to instate built files into the system, and `clean` to remove built files from the source tree.

It's this flexibility of targets that enables `make` to automate any sort of task relevant to assembling a production build of software; not just the typical parsing, preprocessing, compiling proper and linking steps performed by the compiler, but also running tests (`make test`), compiling documentation source files into one or more appropriate formats, or automating deployment of code into production systems, for example, uploading to a website via a `git push` or similar content-tracking method.

An example `Makefile` for a simple software project might look something like the below:

```
all: example

example: main.o example.o library.o
    gcc main.o example.o library.o -o example

main.o: main.c
    gcc -c main.c -o main.o

example.o: example.c
    gcc -c example.c -o example.o

library.o: library.c
    gcc -c library.c -o library.o

clean:
    rm *.o example

install: example
    cp example /usr/bin
```

The above isn't the most optimal `Makefile` possible for this project, but it provides a means to build and install a linked binary simply by typing `make`. Each *target* definition contains a list of the *dependencies* required for the command that follows; this means that the definitions can appear in any order, and the call to `make` will call the relevant commands in the appropriate order.

Much of the above is needlessly verbose or repetitive; for example, if an object file is built directly from a single C file of the same name, then we don't need to include the target at all, and `make` will sort things out for us. Similarly, it would make sense to put some of the more repeated calls into variables so that we would not have to change them individually if our choice of compiler or flags changed. A more concise version might look like the following:

```
CC = gcc
OBJECTS = main.o example.o library.o
BINARY = example

all: example

example: $(OBJECTS)
    $(CC) $(OBJECTS) -o $(BINARY)

clean:
    rm -f $(BINARY) $(OBJECTS)

install: example
    cp $(BINARY) /usr/bin
```

More general uses of `make`

In the interests of automation, however, it's instructive to think of this a bit more generally than just code compilation and linking. An example could be for a simple web project involving deploying PHP to a live webserver. This is not normally a task people associate with the use of `make`, but the principles are the same; with the source in place and ready to go, we have certain targets to meet for the build.

PHP files don't require compilation, of course, but web assets often do. An example that will be familiar to web developers is the generation of scaled and optimised raster images from vector source files, for deployment to the web. You keep and version your original source file, and when it comes time to deploy, you generate a web-friendly version of it.

Let's assume for this particular project that there's a set of four icons used throughout the site, sized to 64 by 64 pixels. We have the source files to hand in SVG vector format, safely tucked away in version control, and now need to *generate* the smaller bitmaps for the site, ready for deployment. We could therefore define a target `icons`, set the dependencies, and type out the commands to perform. This is where command line tools in Unix really begin to shine in use with `Makefile` syntax:

```
icons: create.png read.png update.png delete.png

create.png: create.svg
    convert create.svg create.raw.png && \
    pngcrush create.raw.png create.png

read.png: read.svg
    convert read.svg read.raw.png && \
    pngcrush read.raw.png read.png

update.png: update.svg
    convert update.svg update.raw.png && \
    pngcrush update.raw.png update.png

delete.png: delete.svg
    convert delete.svg delete.raw.png && \
    pngcrush delete.raw.png delete.png
```

With the above done, typing `make icons` will go through each of the source icons files in a Bash loop, convert them from SVG to PNG using ImageMagick's `convert`, and optimise them with `pngcrush`, to produce images ready for upload.

A similar approach can be used for generating help files in various forms, for example, generating HTML files from Markdown source:

```
docs: README.html credits.html

README.html: README.md
    markdown README.md > README.html

credits.html: credits.md
    markdown credits.md > credits.html
```

And perhaps finally deploying a website with `git push web`, but only *after* the icons are rasterized and the documents converted:

```
deploy: icons docs
    git push web
```

For a more compact and abstract formula for turning a file of one suffix into another, you can use the `.SUFFIXES` pragma to define these using special symbols. The code for converting icons could look like this; in this case, `$<` refers to the source file, `$*` to the filename with no extension, and `$@` to the target.

```
icons: create.png read.png update.png delete.png

.SUFFIXES: .svg .png

.svg.png:
    convert $< $*.raw.png && \
    pngcrush $*.raw.png $@
```

Tools for building a `Makefile`

A variety of tools exist in the GNU Autotools toolchain for the construction of `configure` scripts and `make` files for larger software projects at a higher level, in particular `autoconf` and `automake`. The use of these tools allows generating `configure` scripts and `make` files covering very large source bases, reducing the necessity of building otherwise extensive makefiles manually, and automating steps taken to ensure the source remains compatible and compilable on a variety of operating systems.

Covering this complex process would be a series of posts in its own right, and is out of scope of this survey.

Thanks to user samwyse for the `.SUFFIXES` *suggestion in the comments.*

This entry is part 5 of 7 in the series Unix as IDE.

Posted in **Linux** | Tagged **build**, **clean**, **dependency**, **generate**, **install**, **make**, **makefile**, **target**, **unix**