Seven habits of effective text editing

Bram Moolenaar

If you spend a lot of time typing plain text, writing programs or HTML, you can save much of that time by using a good editor and using it effectively. This paper will present guidelines and hints for doing your work more quickly and with fewer mistakes.

The open source text editor Vim (Vi IMproved) will be used here to present the ideas about effective editing, but they apply to other editors just as well. Choosing the right editor is actually the first step towards effective editing. The discussion about which editor is the best for you would take too much room and is avoided. If you don't know which editor to use or are dissatisfied with what you are currently using, give Vim a try; you won't be disappointed.

[Vim commands and options are printed in `this font`]

Part 1: edit a file

1. Move around quickly

Most time is spent reading, checking for errors and looking for the right place to work on, rather than inserting new text or changing it. Navigating through the text is done very often, thus you should learn how to do that quickly.

Quite often you will want to search for some text you know is there. Or look at all lines where a certain word or phrase is used. You could simply use the search command `/pattern` to find the text, but there are smarter ways:

- If you see a specific word and want to search for other occurrences of the same word, use the `*` command. It will grab the word from under the cursor and search for the next one.
- If you set the `'incsearch'` option, Vim will show the first match for the pattern, while you are still typing it. This quickly shows a typo in the pattern.
- If you set the `'hlsearch'` option, Vim will highlight all matches for the pattern with a yellow background. This gives a quick overview of where the search command will take you. In program code it can show where a variable is used. You don't even have to move the cursor to see the matches.

In structured text there are even more possibilities to move around quickly. Vim has specific commands for programs in C (and similar languages like C++ and Java):

- Use `%` to jump from an open brace to its matching closing brace. Or from a "#if" to the matching "#endif". Actually, `%` can jump to many different matching items. It is very useful to check if () and {} constructs are balanced properly.
- Use `[{` to jump back to the "{" at the start of the current code block.
- Use `gd` to jump from the use of a variable to its local declaration.

There are many more, of course. The point is that you need to get to know these commands. You might object that you can't possibly learn all these commands - there are hundreds of different movement commands, some simple, some very clever - and it would take weeks of training to learn them all. Well, you don't need to; instead realise what your specific way of editing is, and learn only those commands that make your editing more effective.

There are **three basic steps**:
1. While you are editing, keep an eye out for actions you repeat and/or spend quite a bit of time on.
2. Find out if there is an editor command that will do this action quicker. Read the documentation, ask a friend, or look at how others do this.
3. Train using the command. Do this until your fingers type it without thinking.

Let's use an example to show how it works:
1. You find that when you are editing C program files, you often spend time looking for where a function is defined. You currently use the `*` command to search for other places where the function name appears, but end up going through a lot of matches for where the function is used instead of defined. You get the idea that there must be a way to do this faster.
2. Looking through the quick reference you find a remark about jumping to tags. The documentation shows how this can be used to jump to a function definition, just what you were looking for!
3. You experiment a bit with generating a tags file, using the ctags program that comes with Vim. You learn to use the `CTRL-]` command, and find you save lots of time using it. To make it easier, you add a few lines to your Makefile to automatically generate the tags file.

A couple of things to watch out for when you are using these three steps:
- "I want to get the work done, I don't have time to look through the documentation to find some new command". If you think like this, you will get stuck in the stone age of computing. Some people use Notepad for everything, and then wonder why other people get their work done in half the time...
- Don't overdo it. If you always try to find the perfect command for every little thing you do, your mind will have no time left to think about the work you were actually doing. Just pick out those actions that take more time than necessary, and train the commands until you don't need to think about it when using them. Then you can concentrate on the text.

In the following sections there will be suggestions for actions that most people have to deal with. You can use these as inspiration for using the **three basic steps** for your own work.

2. Don't type it twice

There is a limited set of words we type. And even a limited number of phrases and sentences. Especially in computer programs. Obviously, you don't want to type the same thing twice.

Very often you will want to change one word into another. If this is to be done in the whole file, you can use the `:s` (substitute) command. If only a few locations needs changing, a quick method is to use the `*` command to find the next occurrence of the word and use `cw` to change the word. Then type `n` to find the next word and `.` (dot) to repeat the `cw` command.

The `.` command repeats the last change. A change, in this context, is inserting, deleting or replacing text. Being able to repeat this is a very powerful mechanism. If you organise your editing around it, many changes will become a matter of hitting just that `.` key. Watch out for making other changes in between, because it will replace the change that you were repeating. Instead you might want to mark the location with the `m` command, continue your repeated change and come back there later.

Some function and variable names can be awkward to type. Can you quickly type "XpmCreatePixmapFromData" without a typo and without looking it up? Vim has a completion mechanism that makes this a whole lot easier. It looks up words in the file you are editing, and also in #include'd files. You can type "XpmCr", then hit `CTRL-N` and Vim will expand it to "XpmCreatePixmapFromData" for you. Not only does this save quite a bit of typing, it also avoids making a typo and having to fix it later when the compiler gives you an error message.

When you are typing a phrase or sentence multiple times, there is an even quicker approach. Vim has a mechanism to record a macro. You type `qa` to start recording into register 'a'. Then you type your commands as usual and finally hit `q` again to stop recording. When you want to repeat the recorded commands you type `@a`. There are 26 registers available for this.

With recording you can repeat many different actions, not just inserting text. Keep this is mind when you know you are going to repeat something.

One thing to watch out for when recording is that the commands will be played back exactly as you typed them. When moving around you must keep in mind that the text you move over might be different when the command is repeated. Moving four characters left might work for the text where you are recording, but it might need to be five characters where you repeat the commands. It's often necessary to use commands to move over text objects (words, sentences) or move to a specific character.

When the commands you need to repeat are getting more complicated, typing them right at once is getting more difficult. Instead of recording them, you should then write a script or macro. This is very useful to make templates for parts of your code; for example, a function header. You can make this as clever as you like.

3. Fix it when it's wrong

It's normal to make errors while typing. Nobody can avoid it. The trick is to quickly spot and correct them. The editor should be able to help you with this. But you need to tell it what's wrong and what's right.

Very often you will make the same mistake again and again. Your fingers just don't do what you intended. This can be corrected with abbreviations. A few examples:

```
:abbr Lunix Linux
:abbr accross across
:abbr hte the
```

The words will be automatically corrected just after you typed them.

The same mechanism can be used to type a long word with just a few characters. Especially useful for words that you find hard to type, and it avoids that you type them wrong. Examples:

```
:abbr pn pinguin
:abbr MS Mandrake Software
```

However, these tend to expand to the full word when you don't want it, which makes it difficult when you really want to insert "MS" in your text. It is best to use short words that don't have a meaning of their own.

To find errors in your text Vim has a clever highlighting mechanism. This was actually meant to be used to do syntax highlighting of programs, but it can catch and highlight errors as well.

Syntax highlighting shows comments in colour. That doesn't sound like an important feature, but once you start using it you will find that it helps a lot. You can quickly spot text that should be a comment, but isn't highlighted as such (you probably forgot a comment marker). Or see a line of code highlighted as comment (you forgot to insert a "*/"). These are errors which are hard to spot in a B&W file and can waste a lot of time when trying to debug the code.

The syntax highlighting can also catch unbalanced braces. An unbalanced ")" is highlighted with a bright red background. You can use the `%` command to see how they match, and insert a "(" or ")" at the right position.

Other common mistakes are also quickly spotted, for example using "#included <stdio.h>" instead of "#include <stdio.h>". You easily miss the mistake in B&W, but quickly spot that "include" is highlighted while "included" isn't.

A more complex example: for English text there is a long list of all words that are used. Any word not in this list could be an error. With a syntax file you can highlight all words that are not in the list. With a few extra macros you can add words to the wordlist, so that they are no longer flagged as an error. This works just as you would expect in a word processor. In Vim it is implemented with scripts and you can further tune it for your own use: for example, to only check the comments in a program for spelling errors.

Part 2: edit more files

4. A file seldom comes alone

People don't work on just one file. Mostly there are many related files, and you edit several after each other, or even several at the same time. You should be able to take advantage of your editor to make working with several files more efficient.

The previously mentioned tag mechanism also works for jumping between files. The usual approach is to generate a tags file for the whole project you are working on. You can then quickly jump between all files in the project to find the definitions of functions, structures, typedefs, etc. The time you save compared with manually searching is tremendous; creating a tags file is the first thing I do when browsing a program.

Another powerful mechanism is to find all occurrences of a name in a group of files, using the `:grep` command. Vim makes a list of all matches, and jumps to the first one.

The `:cn` command takes you to each next match. This is very useful if you need to change the number of arguments in a function call.

Include files contain useful information. But finding the one that contains the declaration you need to see can take a lot of time. Vim knows about include files, and can search them for a word you are looking for. The most common action is to lookup the prototype of a function. Position the cursor on the name of the function in your file and type `[I`: Vim will show a list of all matches for the function name in included files. If you need to see more context, you can directly jump to the declaration. A similar command can be used to check if you did include the right header files.

In Vim you can split the text area in several parts to edit different files. Then you can compare the contents of two or more files and copy/paste text between them. There are many commands to open and close windows, jump between them, temporarily hide files, etc. Again you will have to use the three basic steps to select the set of commands you want to learn to use.

There are more uses of multiple windows. The preview-tag mechanism is a very good example. This opens a special preview window, while keeping the cursor in the file you are working on. The text in the preview window shows, for example, the function declaration for the function name that is under the cursor. If you move the cursor to another name and leave there for a second, the preview window will show the definition of that name. It could also be the name of a structure or a function which is declared in an include file of your project.

5. Let's work together

An editor is for editing text. An e-mail program is for sending and receiving messages. An Operating System is for running programs. Each program has its own task and should be good at it. The power comes from having the programs work together.

A simple example: Select some structured text in a list and sort it: `!sort`. The external "sort" command is used to filter the text. Easy, isn't it? The sorting functionality could be included in the editor. But have a look at "man sort", it has a lot of options. And it's probably a nifty algorithm that does the sorting. Do you want to include all that in an editor? Also for other filter commands? It would grow <u>huge</u>.

It has always been the spirit of Unix to have separate programs that do their job well, and work together to perform a bigger task. Unfortunately, most editors don't work too well together with other programs - you can't replace the e-mail editor in Netscape with another one, for example. You end up usning a crippled editor. Another tendency is to include all kinds of functionality inside the editor; Emacs is a good example of where this can end up. (Some call it an operating system that can also be used to edit text.)

Vim tries to integrate with other programs, but this is still a struggle. Currently it's possible to use Vim as the editor in MS-Developer Studio and Sniff. Some e-mail programs that support an external editor, like Mutt, can use Vim. Integration with Sun Workshop is being worked on. Generally this is an area that has to be improved in the near future. Only then will we get a system that's better than the sum of its parts.

6. Text is structured

You will often work with text that has some kind of structure, but different from what is supported by the available commands. Then you will have to fall back to the "building blocks" of the editor and create your own macros and scripts to work with this text. We are getting to the more complicated stuff here.

One of the simpler things is to speed up the edit-compile-fix cycle. Vim has the `:make` command, which starts your compilation, catches the errors it produces and lets you jump to the error locations to fix the problems. If you use a different compiler, the error messages will not be recognised. Instead of going back to the old "write it down" system, you should adjust the `'errorformat'` option. This tells Vim what your errors look like and how to get the file name and line number out of them. It works for the complicated gcc error messages, thus you should be able to make it work for almost any compiler.

Sometimes adjusting to a type of file is just a matter of setting a few options or writing a few macros. For example, to jump around manual pages, you can write a macro that grabs the word under the cursor, clears the buffer and then reads the manual page for that word into the buffer. That's a simple and efficient way to lookup cross-references.

Using the three basic steps, you can work more effectively with any sort of structured file. Just think about the actions you want to do with the file, find the editor commands that do it and start using them. It's really as simple as it sounds. You just have to do it.

Part 3: sharpen the saw

7. Make it a habit

Learning to drive a car takes effort. Is that a reason to keep driving your bicycle? No, you realise you need to invest time to learn a skill. Text editing isn't different. You need to learn new commands and turn them into a habit.

On the other hand, you should not try to learn every command an editor offers. That would be a complete waste of time. Most people only need to learn 10 to 20 percent of the commands for their work. But it's a different set of commands for everybody. It requires that you lean back now and then, and wonder if there is some repetitive task that could be automated. If you do a task only once, and don't expect having to do it again, don't try to optimise it. But you probably realise you have been repeating something several times in the last hour. Then search the documentation for a command that can do it quicker. Or write a macro to do it. When it's a larger task, like lining out a specific sort of text, you could look around in newsgroups or on the Internet if somebody already solved it for you.

The essential basic step is the last one. You can think of a repetitive task, find a nice solution for it and after the weekend you forgot how you did it. That doesn't work. You will have to repeat the solution until your fingers do it automatically. Only then will you reach the efficiency you need. Don't try to learn too many things at once. But doing a few at the same time will work well. For tricks you don't use often enough to get them in your fingers, you might want to write them down to be able to look them up later. Anyway, if you keep the goal in view, you will find ways to make your editing more and more effective.

One last remark to remind you of what happens when people ignore all the above: I still see people who spend half their day behind a VDU looking up at their screen, then down at two fingers, then up at the screen, etc. - and then wonder why they get so tired... Type with ten fingers! It's not just faster, it also is much less tiresome. Using a computer program for one hour each day, it only takes a couple of weeks to learn to touch-type.

Epilogue

The idea for the title comes from the successful book "The 7 habits of highly effective people" by Stephen R. Covey. I recommend it to everyone who wants to solve personal and professional problems (and who doesn't?). Although some of you will claim it came from the Dilbert book "Seven years of highly defective people" by Scott Adams (also recommended!). See http://www.vim.org/iccf/click1.html and go to "recommended books and CDs".

About the author

Bram Moolenaar is the main author of Vim. He writes the core Vim functionality and selects what code submitted by many others is included. He graduated at the technical university of Delft as a computer technician. Now he mainly works on software, but still knows how to handle a soldering iron. He is founder and treasurer of ICCF Holland, which helps orphans in Uganda. He does free-lance work as a systems architect, but actually spends most time working on Vim. His e-mail address: Bram@Moolenaar.net