vim for Beginners
BSDCan 2022 Tutorial

Benedict Reuschling
bcr@FreeBSD.org
Editors

To configure the system and applications Unix contains human-readable text files which are changeable by an editor. This lecture part describes working with editors under Unix. The focus is on their basic characteristics. Learning an editor is best done by using it on a regular basis. Over time, we’ll learn more features to configure the editor for the users own need: appearance, behavior, shortcuts, etc.
Overview

1. vi
   Basic Design Principles and Fundamentals

2. vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Introducing vi

vi (pronounced vi-ei), written by Bill Joy in 1976, is the standard editor on all Unix systems. One can expect it to be available on any Unix system and work the same everywhere. Learning a few basics is helpful to make editing on any Unix system both easy and familiar without having to re-learn a new editor each time.

**vi is powerful, but needs a bit of getting used to,** because it works with different modes. Even though powerful IDE’s\(^1\) are used for programming nowadays, vi and variants offer similar functionality and editing power which keeps them still relevant. Also, they’re comparably small and have a light memory footprint when compared to a full-blown IDE.

---

\(^1\)Integrated Development Environments
Basic Design Principles and Fundamentals

vi knows three modes of operation:

**Normal mode** This is the mode at the start of an editor session and to which one returns from other modes. In this mode, vi interprets **keyboard input** as **commands**.

**Insert mode** This is the mode to **change or insert** text. Activate this mode by pressing `i` (for Insert). Other keys are a, A, o, O, s, S, R, and c. Exit out of this mode by pressing `Esc`.

**Last Line Mode** In this mode, complex text manipulations are possible (for example, search and replace). In normal mode, enter `:`, `/`, `?`, or `!` to enter it (look at the bottom of the editor). The **ex** editor (sister editor of vi) uses this mode extensively. After exiting this mode, vi returns to normal mode.
A Basic vi Editing Session

Basic editing of a file in vi and saving the changes follows these steps:

1. Enter `vi` textfile at the command line, vi opens
2. Press `i` to enter INSERT mode
3. Enter some text
4. When done, press `Esc` to return to normal mode
5. Enter `:wq` to save and close the editor, returning to the command line

This most basic functionality suffices for most editing tasks in Unix. Once done a couple of times, they’re fairly easy to remember. For more functionality, we’ll look at vim, the improved version of vi in the remaining slides.
Overview

1 vi
   Basic Design Principles and Fundamentals

2 vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Introducing vim

vim (abbrev. for vi improved) by Bram Moolenaar et. al. is a popular, enhanced version of the original vi editor. Compatible with vi, there is even a graphical user interface (GUI) called GVim with mouse support for Mac and Windows systems. Everything that vi can do, vim is capable of doing, but not vice versa. A lot of extra functionality has been added to vim over the years to make it easier to work with. One example is that the current mode is shown in the last line for easier orientation. A lot of complicated editing tasks can be done in vim with only a few keystrokes. Enter vimtutor at the command prompt to start an interactive tutorial that shows how to use vim step by step.
Overview

1. vi
   Basic Design Principles and Fundamentals

2. vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Leaving the Editor

Unix beginners sometimes accidentally end up in vi/vim and don’t know how to exit again. By entering `:wq` vi/vim writes the file and exits. The meaning of each character is as follows:

- `:` enter last line mode
- `w` write, (when needed)
- `q` quit, leave editor. vim warns of any remaining unsaved changes. If needed, add a `!` to force the quit!

When editing a file with insufficient privileges (i.e. editing a file owned by root or another user), the editor will refuse to save. In this case, quit the editor and discard the changes by entering `:q!` to leave.
Other Commands to Quit with or without Saving

Z Z  Save and exit from normal mode (valid with known filename)
:w filename  saves to file filename
:wq  Quitting with saving (write and quit)
:x  Exiting with saving (Exit)
:q  Quit (when there are no unsaved changes left)
:q!  Exiting without saving (force quit)
:w !sudo tee %  Save as root (when editing i.e., system config files), confirm with 0
Overview

1. vi
   Basic Design Principles and Fundamentals

2. vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Positioning the Cursor in Normal Mode

The following keys move the cursor around:

- ← or h
- ↓ or j
- ↑ or k
- → or l

This may help remembering the functionality of those keys for a US/German keyboard:

Look at the j key on your keyboard. It's shaped like a down arrow with one side missing. Next to it is the k key, which looks like an arrow pointing upwards. If j moves the cursor down, then k moves the cursor up.

The key furthest to the right is the l key, and that moves the cursor to the right. Only one key is missing now, which is h. It is the leftmost of those four keys and that's why it moves the cursor to the left.

Practice using these keys for a while and you'll find that you are much faster. In fact, these are so popular that their functionality is also found in applications like YouTube and others with similar movement actions.
The Original vi keyboard

Why were keys like $h$, $j$, $k$, $l$, and $\text{Esc}$ chosen for frequent actions in vi? Because of the layout of the original keyboard vi was developed on.

Source: https://twitter.com/jschauma/status/1434265985138270210
Movement Commands

Commands to move the cursor to a specific position in normal mode are:

- `w` Jump to the beginning of the next word
- `b` Jump back to the previous word
- `e` Jump to the end of the next word
- `^` Beginning of the line (like in regular expressions)
- `$` End of the line (like in regular expressions)
- `( ` Jump to the beginning of the sentence
- `) ` Jump to the end of the sentence
- `{ ` Jump to the beginning of the section
- `} ` Jump to the end of the section
Overview

1. vi
   Basic Design Principles and Fundamentals

2. vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Cut, Copy, & Paste

Use the following in normal mode to open a text file for editing:

:e filename Opens filename for editing

Text can be **copied** (called yanked in vi), **deleted**, and **pasted** in normal mode like this:

- `dd` delete the current line at cursor position
- `yy` copies (yanks) the current line into a buffer
- `p` puts/pastes the yanked/deleted text from the buffer into the next line
- `P` (put) see above, but into the current line

**Examples:**

- `dw` Delete word with cursor at first character
- `yw` yank current word into a buffer
- `y3w` yank the next 3 words into a buffer (yank three words)
- `y)` yank whole sentence into a buffer
- `p` paste buffer into next line from cursor position
Search and Replace

The following commands exist to search and replace text:

/   Search forward
?   Search backwards

Examples:

/text   Search for the string text
/^text  Search for text at the beginning of the line
/\$   Search for empty lines
/text\>  Search for text at the end of the word
.ing   Search for any character in front of the string ing
/[Hh]ello  Search for Hello as well as hello (case-insensitive)

Press \text{n} to jump from one search result to the next (if there is another).
Using Regular Expressions in Text Searches

Specify the string to search for with a regular expression. The character \ cancels out the meaning of the following character, so that one can search for / and ?, too. Limit operations for search and replace to certain areas of text:

^ Beginning of the line
$ End of the line
. any single character
* a sequence of any characters (string)
<< Begin of the word
>> End of the word

[abc] matches a, b, and c (alternatives); with a ^ in front of it, one can find everything except a, b, or c.
Search and Replace Text in Last Line Mode

Replace text using `s(substitute)` in Last Line mode. It has the following form:

\[<\text{Area}>s/\text{searchtext}/\text{replacement}/\text{options}\]

**Examples:**

- `s/old/new` Replaces **the first occurrence** of `old` with `new` in the **current line** (cursor position), ignoring other matches that follow.
- `2,5s/old/new` Replaces the first occurrence of `old` with `new` in line 2 to 5.
- `%s/old/new` Replaces in every line **the first occurrence** of `old` with `new` (not in the rest of the line!)
- `%s/old/new/g` Replace **every occurrence** of `old` with `new` (g = global)
Conditional Search and Replace

Vim can also make replacements when a previous search was successful. This is useful to target specific lines with replacements. The following line will first search for lines that contain the word metal. On those lines that match, it replaces all occurrences of silver with gold.

:g/metal/s/silver/gold/g
Overview

1 vi
   Basic Design Principles and Fundamentals

2 vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
The Language of vim

We already saw that we don’t need to change to INSERT mode to make changes to text. In fact, it is faster (requires fewer keystrokes) to do the change in normal mode, because it does not require pressing \texttt{i} before making the change, and \texttt{Esc} afterwards, in addition to the change itself.

Consider a common editing task like changing a whole word to some other word (i.e. fixing a typo). Starting in normal mode with the cursor positioned at the first character of the word, we can “tell” vim what to do: \texttt{change word}. This will remove all characters of the word and change into INSERT mode to type the new word. That operation is much faster than changing into INSERT mode, hit delete until the end of the (long) word, type in the new word and exiting INSERT mode.

The \texttt{cw} command is part of vim’s language and knowing it enables some powerful editing. These commands are modeled after human language, which makes it easy to remember and apply in a flexible way to the editing task at hand.
The general structure of vim’s language in normal mode is like this:

```
action range object
```

or, closer to human language, we would say

```
verb modifier noun
```

In the example of `cw` we can see that not all elements need to be specified to perform an action. In this case, the range/modifier is optional as it is clear from the context that we want to work on a word.

Let’s look at each of these elements separately and then how we can combine them.
# Language Elements

## Action/Verb

<table>
<thead>
<tr>
<th>Action/Verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>change</td>
</tr>
<tr>
<td><code>d</code></td>
<td>delete</td>
</tr>
<tr>
<td><code>y</code></td>
<td>yank (copy)</td>
</tr>
<tr>
<td><code>g u</code></td>
<td>lowercase</td>
</tr>
<tr>
<td><code>g U</code></td>
<td>uppercase</td>
</tr>
<tr>
<td><code>=</code></td>
<td>filter/C-indenting</td>
</tr>
<tr>
<td><code>g ?</code></td>
<td>ROT13 encoding</td>
</tr>
</tbody>
</table>

## Range/Modifier/Operator

<table>
<thead>
<tr>
<th>Range/Modifier/Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i</code></td>
<td>inside</td>
</tr>
<tr>
<td><code>a</code></td>
<td>around</td>
</tr>
<tr>
<td><code>t</code></td>
<td>till, search something and stop before it</td>
</tr>
<tr>
<td><code>/</code></td>
<td>find string or regex</td>
</tr>
</tbody>
</table>

### NUMBER

- a number like 2, 5, 11

## Object/Noun

<table>
<thead>
<tr>
<th>Object/Noun</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>w</code></td>
<td>word</td>
</tr>
<tr>
<td><code>s</code></td>
<td>sentence</td>
</tr>
<tr>
<td><code>p</code></td>
<td>paragraph</td>
</tr>
<tr>
<td><code>t</code></td>
<td>tag (HTML/XML)</td>
</tr>
<tr>
<td><code>b</code></td>
<td>block (in programming)</td>
</tr>
<tr>
<td><code>0</code></td>
<td>start of line</td>
</tr>
<tr>
<td><code>$</code></td>
<td>end of line</td>
</tr>
</tbody>
</table>

Combine elements from left to right (like `c i w`, `d 3 w`, or `d t .`) and see what they can do.
vim Definitions of Word, Sentence, and Paragraph

We find the following definitions in :help motion.txt:

**word** A word consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space (spaces, tabs, `<EOL>`). [...] An empty line is also considered to be a word.

**WORD** A WORD consists of a sequence of non-blank characters, separated with white space. An empty line is also considered to be a WORD.

**sentence** A sentence is defined as ending at a `.`, `!` or `?` followed by either the end of a line, or by a space or tab. Any number of closing `)`, `]`, `"` and `'` characters may appear after the `.`, `!` or `?` before the spaces, tabs or end of line. A paragraph and section boundary is also a sentence boundary.

**paragraph** A paragraph begins after each empty line, [...]. A section boundary is also a paragraph boundary. Note that a blank line (only containing white space) is NOT a paragraph boundary.

**section** A section begins after a form-feed (`<C-L>`) in the first column [...].
Try it out yourself

Write a few sentences and paragraphs or take an existing text (lorem ipsum) and try to find the sequences for the following instructions:

• Delete the next three words
• Change until the next comma
• Uppercase the current sentence
• ROT13 the whole paragraph
• Change the text inside of the
• Delete all words inside the quotes
• Change text inside of and including the ()
• Delete until the next "end" word
• Change all words before the cursor
Overview

1  vi
   Basic Design Principles and Fundamentals

2  vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Autocompletion to help with text editing

The autocomplete feature of vim helps with re-entering text that is either complicated to type or in a different file or location. The goal is to help in the editing to avoiding mistakes and saving a bit of time.

To autocomplete a word, start typing the first characters of the word in question. In its basic form, vim scans the current file for a word to the left of the cursor position up to the top of the file to see if the same word is already present. Completion happens when the words match the remaining characters. If there are ambiguous matches, a menu appears from which the user can chose the proper word. To reduce the list further, type extra characters to narrow the list down to an exact match.

The autocomplete is not limited to simple words in a dictionary, but can also expand to filenames, whole lines and programming language specific keywords and phrases.
Invoking Autocompletion

For a simple example, begin a new document in vim and type a word like "Developer". In a new line or word, enter D and while still in INSERT-mode, press \texttt{Ctrl+p} to complete the whole word.

For an ambiguous match, enter another word like "Development". Enter another D and \texttt{Ctrl+p} will now display a menu of possible word matches. To cycle through the list, press \texttt{Ctrl+p} again to move to the next word and have vim replace the current word in the text. When satisfied, press \texttt{Enter}, \texttt{Esc}, or \texttt{Esc} to make the menu disappear.

Setting the \texttt{infercase} option (set \texttt{infercase} in \texttt{vimrc}) causes inclusion of upper and lowercase matches in the menu. By default, setting the \texttt{ignorecase} option for searching ignores upper and lowercase matches, but they appear in the autocomplete menu nonetheless. Typing a lowercase letter, the menu includes the same words, but preserves the lowercase letter at the beginning.
Expanding beyond words to autocomplete

The vim editor completes more than simple words. We’ll cover the following here:

- Path and filename completion
- Completing words from a dictionary
- Completing whole lines
- Omni-completion

These powerful helpers trigger from INSERT-mode via the `Ctrl+x` key combination, followed by the specific mode for the completion type listed above. More and customizable completions are available, but the built-in ones are already helpful.
Building the autocomplete list

The autocomplete menu vim presents is build from different sources. The more sources are available, the chances are higher that a match is present for completion. Programmers often use header or include files to structure the code and encapsulate commonly used functionality within separate files. When working on a source file (either C/C++, Java, Ruby, etc.) vim can search those include files referenced in the code to include more options in the list. This activates by typing this combination: Ctrl+×, Ctrl+i (for include). The menu will now contain words from those files as well.
Completing Paths and Filenames

Developers often need to reference a filename in source code to either include it or let HTML tags like \texttt{href=} point to their location. With long and complicated filenames this is a big time saver. To avoid typos and provide any subdirectories as well, vim can insert the proper path and filename based on their position in the underlying file system.

The vim editor tracks the current working directory as a point of reference. Using \texttt{:pwd} we can get the current working directory, same as in the shell. Similarly, to change a directory for vim use \texttt{:cd} followed by the target directory (either absolute or relative).

Expanding the filename is always \textbf{relative} to the current working directory and not to the editing file. Insert a path or filename in \texttt{INSERT}-mode with the following key combination: \texttt{Ctrl+x}, \texttt{Ctrl+f} (for \texttt{file}).
Completing Words from a Dictionary

To avoid typos, vim has a built-in spell checker with language-specific dictionaries to use with autocompletion. To activate the spell checker, enter \texttt{set spell} in \texttt{vimrc} or run it from last line mode for the current session. Highlighting marks words not contained in the dictionary (possibly false ones) now. Typing some characters and using the key combination \texttt{Ctrl+x}, \texttt{Ctrl+k} will list matching words from the dictionary. To set a specific language, enter the following:

\texttt{set spell spelllang=en_us}

Define custom dictionaries using the \texttt{dictionary} option, a comma-separated list of filename paths containing the words. More information is available using \texttt{:help dictionary}. 
Completing whole lines

Completing a single, long and complicated word to type is already useful. But vim is also capable of completing a whole line. This is helpful when repeating a line that needs to change only slightly. This saves a lot of typing and programmers avoid entering a lot of identical lines this way.

To trigger the completion of a whole line, start by typing the first characters of the line and use \( \text{Ctrl}+\text{x}, \text{Ctrl}+\text{l} \) (for line or long) to let vim finish the line as a whole.

Again, a menu will help to select matches. Like before, vim includes header files to widen the options. Programmers will appreciate that vim ignores indentation at the beginning of a line to complete non-whitespace characters.
Context-Aware Omni-Completion

Based on the file and cursor position, vim can suggest specific completions that match. Activate this by adding the following in .vimrc:

```
set nocompatible
filetype plugin on
set omnifunc=syntaxcomplete#Complete
```

The filetype plugin identifies what source file type vim currently edits and provides a range of completions based on those programming languages. Setting a proper omnifunc= value enables valid language-specific syntax completions when invoking `Ctrl`+`x`, `Ctrl`+`o` (for omni). This way, you’ll get HTML-specific recommendations in HTML files, but not in Java files and vice versa.

For example, HTML, CSS, and PHP language features are fully available, among many others. When creating a `style.css` file and starting an `h1` tag to markup like this:

```
h1 { co
```

running omni-completion (`Ctrl`+`x`, `Ctrl`+`o`) shows only valid elements within this context to autocomplete.
Working with Numbers

Not only is vim a great text editor, it also supports working with numbers in normal mode.

Ctrl+a, Ctrl+x  Increase/Decrease the next number in the current line
NUMCtrl+a, NUMCtrl+x  Increase/decrease by NUM

The nrformats option controls what numbers are changeable. Possible values include:

- bin (binary),
- hex (hexadecimal),
- octal, and
- alpha (letters of the alphabet)

To increment/decrement a binary number, it needs to have this prefix: 0b. Similarly, hexadecimal numbers need to have 0x in front to be recognized as such. Octal numbers act weird for numbers like 007. Increasing them changes it to 010, while most people would expect it to change to 008. Increasing letters to the next one in the alphabet will probably be of little use, too. For this reason, one can use

set nrformats=bin,hex

to limit this functionality to binary and hex, in addition to decimal numbers.
Overview

1. vi
   - Basic Design Principles and Fundamentals

2. vim
   - Commands to Quit vim
   - Positioning the Cursor
   - Cut, Copy, & Paste
   - Search and Replace
   - The Language of vim
   - Autocompletion

Macros
- Leader Key
- Connecting to the World Outside the Editor
- Arranging your Workspace
- Settings in vimrc
Macros

Vim can record common typing tasks (keystrokes, command executions), stores them in a register, and plays them back to save the user time and typing. A macro is recorded in normal mode like this:

```
q<register><keystrokes>q
```

For example, a simple "Hello World" macro stored in register a would be recorded like this:

```
q a i Hello World Esc q
```

During the macro recording, vim will display recording @a in the lower left corner. Typing pauses are not included as part of the macro.
Viewing and Changing Macros

View the contents of all registers by running `:reg` (short for `:registers`). To look at register a, use `:reg a`. We can see the following macro representation for our recorded register a:

```
iHello Worldq<80>kb^[
```

Some of the more cryptic representations are for keys like Esc, Enter and others. To edit a macro, go to an empty line and paste the contents of the register like this:

```
"ap
```

Edit the macro as needed and then move to the beginning of line (pressing 0 in normal mode) to save (yank) it again into register a:

```
"ay$
```
Macro Execution

To execute the recorded macro, use @ followed by the register: @a. Repeat the last executed macro by pressing @@. To execute a macro multiple times, provide the number of executions in front: 5@@ or 13@a

A macro is applied to the current editor contents (called buffer). However, it is not limited to that and can be applied to a number of files simultaneously. This is a powerful functionality when a lot of the same changes need to be applied over a number of files.

First, use :args to define on which files the macro should be applied:

```
:args a.txt b.txt c.txt
```

Then, apply the macro stored in register a in normal mode to all lines in the file:

```
:argdo normal @a
```

Finally, save all buffers that were changed by the macro to disk:

```
:argdo update
```
Recursive Macros

A recursive macro is an extension to the macros we just learned about. A recursive macro contains instructions within it to call itself. For example, a macro `r` (for recursive) would contain a call to execute macro `r`.

Suppose we have a file containing these lines:

```
123
456
789
```

We want to write a recursive macro that encloses these numbers in quotes ("""). We place our cursor in front of the first number and start recording:

```
i"^[<80><fd>aA"^[<80><fd>aj0@r
```

This macro enters insert mode to place the first " in front of 1. Then it switches back to normal mode to use A to jump to the end of the word and enter insert mode to place the closing ". Then it switches to normal mode again, moves to the next line using j and then jumps to the beginning of the line using 0. Now it executes itself again.

The macro runs until there are no more lines to go down to (pressing j) and then terminates.
Overview

1. vi
   Basic Design Principles and Fundamentals

2. vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros

   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Leader Key

Vim's keyboard focused editing allows the user to define custom key-combinations for personal shortcuts. To initiate such a shortcut and to distinguish them from the built-in ones, the leader key is used. It is named because it leads the key-combination, meaning that it has to be pressed first, followed by the custom keys that execute the defined action. The backslash \ is the default leader key. Commonly redefined leader keys are (Space) or comma ,. Here we change it to comma in .vimrc:

" set leader key to comma
let mapleader = ",,"

We can now define actions by using <Leader> as the placeholder for the leader key. This example allows saving a file by pressing the ,w key-combination in normal mode:

"Save file using Leader + w
nnoremap <Leader>w :w<CR>
Leader Key Combinations for Insert Mode

The following shortcuts are defined for insert mode only (imap = insert mode mapping). When the string is encountered (starting with the leader key), it gets replaced with what’s defined as the action.

A common typing exercise is to insert the current date in various formats. These shortcuts provide some common ones:

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inoremap &lt;Leader&gt;Ymd</code></td>
<td><code>&lt;C-R&gt;=strftime(&quot;%Y%m%d&quot;)&lt;CR&gt;</code></td>
</tr>
<tr>
<td><code>inoremap &lt;Leader&gt;ymd</code></td>
<td><code>&lt;C-R&gt;=strftime(&quot;%y%m%d&quot;)&lt;CR&gt;</code></td>
</tr>
<tr>
<td><code>inoremap &lt;Leader&gt;dmY</code></td>
<td><code>&lt;C-R&gt;=strftime(&quot;%d.%m.%Y&quot;)&lt;CR&gt;</code></td>
</tr>
<tr>
<td><code>inoremap &lt;Leader&gt;dmy</code></td>
<td><code>&lt;C-R&gt;=strftime(&quot;%d.%m.%y&quot;)&lt;CR&gt;</code></td>
</tr>
</tbody>
</table>

To test the formats, go into insert mode and type `ymd` and see how they are replaced automatically. Type them quickly as the default timeout for the leader key is 1000 ms. Taking too much time to type these will not make the replacement happen.

Be careful when defining insert mode mappings as there could be legitimate text inputs being overwritten each time.
Using Leader Key in Normal Mode

A typical typing task for programmers and authors is to wrap word in quotes (single or double). Same goes for all kinds of brackets and braces. Navigating back to the beginning of the word after typing it, starting the opening character, then jumping to the end of the word to place the closing character becomes tedious after a while. Luckily, we know all the navigation commands by now to create such a mapping. We use normal mode (\texttt{nmap} = normal mode mapping) here because we typically place them after typing is done.

The following additions to \texttt{.vimrc} will save us a lot of typing in the future:

\begin{verbatim}
nnoremap <Leader>q bi"<Esc>ea"<Esc> "Enclose word in " using Leader + q (quotes)
nnoremap <Leader>p bi(<Esc>ea)<Esc> "Enclose word in () using Leader + p
nnoremap <Leader>c bi{{{<Esc>ea}}}<Esc> "Enclose word in {{}} using Leader + c (curlies)
\end{verbatim}

Execute them by placing your cursor anywhere within the word in normal mode and run i.e. \texttt{,}".
Overview

1. **vi**
   Basic Design Principles and Fundamentals

2. **vim**
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key

   **Connecting to the World Outside the Editor**
   Arranging your Workspace
   Settings in vimrc
Connecting to the World Outside the Editor

In last line mode one can use the command

`:!<shellcommand>`

to execute any command in the shell. The percent sign `%` represents the editor content. By using `:sh`, vi branches off to a shell from which one can return by entering `exit`. An interesting possibility is the use of Unix filters on areas of text in vi.

**Examples:**

- `%!column -t` Formats the text as a table with spaces as field separators
- `%!sort -k 2` Sorts the text in the second column (counting starts at 1)
- `%!uniq` Removes sorted lines in a text that occur more than once (`unique`)
- `%!fold -sw25` Wraps text after 25 characters after the last space which still fits into the line
- `%!TOhtml` Renders the current text as an HTML page including the editor window

**Man page lookup** Position the cursor on a word to look up and press K in command mode
Overview

1. vi
   Basic Design Principles and Fundamentals

2. vim
   Commands to Quit vim
   Positioning the Cursor
   Cut, Copy, & Paste
   Search and Replace
   The Language of vim
   Autocompletion
   Macros
   Leader Key
   Connecting to the World Outside the Editor
   Arranging your Workspace
   Settings in vimrc
Arranging your Workspace

Same as in a desktop IDE, vim can arrange the editor to split it into windows (called panes) and switch between them. This way, one editor pane looks something up in a file or man page, while the other pane makes changes. This way, vim stays open to switch to a file to look something up. The splits are either horizontally or vertically. A pane can split again to create splits within splits, dividing up the editor window into smaller pieces. Another IDE-like functionality is the use of tabs to arrange workspaces together. These tabs group windows together (for projects) and allows to switch between them. This allows changing the editing context of i.e. programming, writing a paper, and editing a webpage. Each of these tasks can have its own tab and each tab can split into panes.
Splitting the Editor window

The user can decide to split the editor window either horizontally or vertically. The key combination Ctrl-w (for window) initiates the split, followed by either v for vertical or s for horizontal. Each window divides in half to give each new window equal spacing.

Figure: Commands for sub-dividing editor windows.
Split commands

The following list describes split-related useful commands for last line mode:

:sp Splits horizontally from the current window. Passing a filename as an arguments opens a new split with that file in it.

:vsp Same as above, but for a vertical split.

To switch the active window (the status bar highlights the active one), use these keystrokes:

\[\text{Ctrl-w w}\] Cycle between windows

\[\text{Ctrl-w h}\] Focus window to the left of the current one

\[\text{Ctrl-w j}\] Focus window below the current one

\[\text{Ctrl-w k}\] Focus window above the current one

\[\text{Ctrl-w k}\] Focus window to the right of the current one

To close the current window, press \[\text{Ctrl-w c}\]. The happens using :clo (for close) in last line mode. To close all but the current window, execute :on (only).
Creating Workspaces with Tabs

To create a new workspace called tab, use `:tabedit <filename>`. This creates a new tab to edit the filename provided as a parameter and switches to it. Providing no filename an empty tab opens. Split each tab like in any vim window. Find the list of current tabs at the top of the window. Move an existing window to its own, new tab using `Ctrl-w T`.

Here are some more useful commands in last line mode when working with tabs:

- **:tabn** Switch to the next tab (:tabnext).
  In normal mode, press `g t` to do the same.

- **:tabp** Switch to the previous tab (:tabprevious).
  `g T` does the same in normal mode.

- **:tabc** Close the current tab including all windows contained in it (:tabclose).

- **:tabo** Close all but the current tab (:tabonly).
Settings

Create settings, macros, abbreviations, and other options in each user’s home directory. For vi, the file is called .exrc, for vim it’s .vimrc. Some options for vim are here:

- `syntax on`  Activate syntax highlighting for common programming languages
- `set wrap`  automatic text wrap when the line becomes too long
- `set showmatch`  when closing a bracket, briefly show the matching opening bracket
- `set number`  Show line numbers
- `set hlsearch`  Highlight search results
- `set incsearch`  Show search results when typing and jump directly to them
- `set nocompatible`  cursor keys work as normal when in insert mode
- `iab WBG`  With best regards  Replaces WBG with With best regards

With these settings editing source code with vim is comfortable already.